FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

**Ph.D. Dissertation**

Tom Hvitved

# Contract Formalisation and Modular Implementation of Domain-Specific Languages

*Submitted to the PhD School of The Faculty of Science,*

*University of Copenhagen*

## Abstract

This dissertation encompasses the topics *contract formalisation*, *domain-specific languages implementation*, and *enterprise resource planning systems*. The dissertation is a collection of six independent chapters, two of which are published papers, two of which are extended versions of published papers, and two of which are unpublished manuscripts.

Our contributions to the field of contract formalisation covers three chapters. The first chapter is a comprehensive, comparative survey of previous work on formal languages and models for legally binding contracts. We provide a list of key requirements for formalising contracts, which serves both as our comparison measure, and as a guideline for future formalisms. We conclude that, although much work has been carried out in the field, much remains to be done. In particular, only very few previous approaches present a clear, formal semantics. To this end, we propose a novel, formal model of legally binding contracts in the second chapter. Besides striving for an unambiguous, formal semantics, we focus on a model that properly accounts for blame assignment. That is, our contract model takes into account that a breach of contract must be attributable to one or more of the contract participants. In the third chapter we shift the focus from legally binding contracts to contracts for distributed programming. We propose a fundamentally new generalisation of the traditional programming-by-contract paradigm, which gives rise to a game-theoretic view on distributed-programming contracts. Perhaps surprisingly, many aspects of contracts for distributed programming and traditional legally binding contracts turn out to coincide.

Our contributions to the field of domain-specific languages implementation covers two chapters. We introduce a Haskell library for constructing data types, and functions on them, in a modular and extendable fashion. Our library targets implementations of domain-specific languages, in which the abstract syntax trees (ASTs) are represented as elements of a recursive algebraic data type. The shortcoming of the traditional approach is that it is missing modularity, and typically we find ourselves implementing AST transformations for which the type system does not properly account for the underlying invariants (also known as the *Expression Problem*). In the first of the two chapters we introduce our library, which enables full modularity and extensibility, as well as seamless support for AST annotations and run-time optimisation in the vein of deforestation. In the second chapter we extend our library with support for variable binders. We use a restricted form of higher-order abstract syntax that permits effective recursion schemes, as well as transformations on higher-order ASTs.

In the final chapter we present a novel software architecture for enterprise resource planning (ERP) systems, based on domain-specific languages. This chapter ties together the previous chapters, by drawing on our domain-specific language for contracts and our Haskell library of compositional data types. We present a detailed overview of our architecture, as well as the domain-specific languages for specifying the data model, reports, and contracts respectively. The data model defines objects that we want to model, such as customers; the reports define the information we want to derive from the transactions in the system, such as current balance; and the contracts define the expected future transactions, such as payments. We demonstrate the validity of our approach by means of a use case, in which we implement a small ERP system from scratch. The implementation that we obtain is but a fraction of the code in normal ERP systems, and the domain-specific style yields specifications that are much closer to the informal, textual requirements than the corresponding implementations in standard ERP systems.

## Resumé (Danish abstract)

Denne afhandling favner over emnerne *kontraktformalisering*, *implementation af domænespecifikke sprog*, samt *virksomhedssystemer*. Afhandlingen består af seks uafhængige kapitler, hvoraf to kapitler er publicerede forskningsartikler, to kapitler er udvidede udgaver af publicerede forskningsartikler, og to kapitler er ikke-publiceret materiale.

Vores bidrag til feltet kontraktformalisering består af tre kapitler. I det første kapitel foretager vi en dybdegående gennemgang af eksisterende litteratur omhandlende formelle modeller og sprog for juridisk bindende kontrakter. Vi præsenterer en liste af nøglekrav, der muliggør en kvalitativ sammenligning af den eksisterende litteratur. Listen kan endvidere fungere som en målestok for fremtidige kontraktformalismer. Konklusionen på vores gennemgang er, at forskningsfeltet stadig er åbent – specielt bærer feltet præg af en manglende matematisk modenhed. På baggrund af denne konklusion introducerer vi en ny, formel model for juridisk bindende kontrakter i det andet kapitel. Udover at hige efter en formel semantik, er vores mål at konstruere en model, der tager højde for skyldtildeling. Med skyldtildeling menes, at alle kontraktbrud kan henføres til én eller flere kontraktdeltagere. I det tredje kapitel flytter vi fokus fra traditionelle juridisk bindende kontrakter til programmeringskontrakter i et distribueret miljø. Vi præsenterer en fundamentalt ny generalisering af *programming-by-contract* paradigmet, hvilket fører til en spilteoretisk formulering af kontrakter for distribueret programmering. Måske overraskende, viser det sig, at mange aspekter af programmeringskontrakter og juridisk bindende kontrakter er identiske.

Vores bidrag til feltet implementation af domænespecifikke sprog består af to kapitler. Vi præsenterer et Haskell bibliotek, der muliggør konstruktion af datatyper – og funktioner på datatyperne – på en modulær og udvidelig måde. Det primære anvendelsesområde er implementation af domænespecifikke sprog, hvor abstrakte syntakstræer (AST'er) traditionelt repræsenteres som elementer af en rekursiv, algebraisk datatype. Ulempen ved denne repræsentation er, at den ikke er modulær og udvidelig, og således vil vi ofte implementere transformationer af AST'er, hvor typesystemet ikke indfanger de underliggende invarianter (også kendt som *The Expression Problem*). I det første af de to kapitler introducerer vi vores bibliotek, der udover at muliggøre modulære, udvidelige datatyper og funktioner, også tilbyder ubesværet håndtering af AST annoteringer og køretidsoptimeringer i stil med *deforestation*. I det andet kapitel udvider vi vores bibliotek til også at håndtere variabelbindere. Vi indfører en begrænset form for højereordens abstrakt syntaks, der tillader såvel effektive rekursionsskemaer som transformationer af højereordens AST'er.

I det sidste kapitel præsenterer vi en nyskabende softwarearkitektur for virksomhedssystemer, baseret på domænespecifikke sprog. Dette kapitel binder de tidligere kapitler sammen, ved at trække på vores domænespecifikke sprog for kontrakter og vores Haskell bibliotek til modulære datatyper. Vi giver et detaljeret overblik over arkitekturen, såvel som en præsentation af de domænespecifikke sprog, der benyttes til at specificere henholdsvis datamodellen, rapporter og kontrakter. Datamodellen definerer objekterne vi ønsker at modellere, såsom kunder; rapporterne specificerer information vi ønsker at aflede fra transaktionerne i systemet, såsom den aktuelle balance; og kontrakterne specificerer forventede fremtidige transaktioner, såsom betalinger. Vi demonstrerer gyldigheden af vores tilgang ved at implementere et minimalt virksomhedssystem fra grunden. Omfanget af vores implementation udgør en brøkdel af koden fundet i normale virksomhedssystemer, og vores domænespecifikke tilgang betyder en langt mindre afstand mellem kravsspecifikation og implementation.

# Contents

# Overview

This dissertation is submitted in partial fulfilment of the requirements for the Ph.D. degree at the Department of Computer Science, University of Copenhagen (DIKU). The work has been supervised by Professor Fritz Henglein and Associate professor Andrzej Filinski, both at DIKU. The Ph.D. studies have been carried out under the 4+4 scheme [24, Appendix 2]. Therefore, some of the work presented in this dissertation is based on the Ph.D. progress report [49], which also qualifies as an extended master's thesis.

The research has been conducted in the context of the research project *3rd generation Enterprise Resource Planning* (3gERP). The objective of the 3gERP project was:

> [...] to develop a fundamentally new high-level software architecture with implementation tools and business models for a standardized, yet highly flexible and configurable global ERP-system for small- and medium-sized enterprises (SMEs), which can be implemented and maintained at a fraction of the cost of current ERP systems. [1]

The results presented in this dissertation are therefore ultimately targeting the objective above, and the work can be qualified as *applied theoretical computer science*.

The dissertation is structured in two parts, which in turn are divided into three chapters, making for a total of six chapters. Each chapter can be read independently, and there is consequently a slight overlap between some chapters. Chapters 2 and 4 are published papers [11, 52], with only minor changes and reformatting compared to the published versions. Chapters 1 and 3 are extended versions of published (short) papers [42, 50]. Chapters 5 and 6 are unpublished manuscripts intended for later publication [10, 51]. Below follows a short overview of the two parts of the dissertation, and how the two parts are tied together. Each chapter contains a more detailed and motivating introduction, as well as a clear summary of the scientific contributions and how they relate to previous work.

## Contract Formalisation

The topic of the first part is *contract formalisation*, and the results we present in this part are mostly of a theoretical nature. By contract formalisation we mean mathematical models and domain-specific languages for specifying and reasoning about contracts. In the first two chapters, a contract refers to a traditional, legally binding agreement between two or more parties, and the specific domain of interest is *business contracts*—for instance a sales contract.

Chapter 1 serves as an introduction to the field of (business) contract formalisation. The research area is characterised by great diversity, ranging from logic based approaches over functional programming based approaches to process calculus based approaches. Rather disappointingly, most research within the area of contract formalisation lacks mathematical rigour, that is a formal semantics, and in some cases complete and consistent presentations. The motivation for our work in Chapter 2 is consequently to give a formal, semantic model for (business) contracts, as well as a domain-specific language for specifying contracts. Moreover, our focus in Chapter 2 is to construct a semantic model that properly accounts for *blame assignment*. With blame assignment we mean that blame can be inferred from the semantics of a contract, in case of a breach of contract. That is, if the execution of a contract fails, then one or more of the involved parties will be blamed.

The topic of Chapter 3 is contracts as well, but rather *programming contracts* than business contracts. In this chapter we investigate an extension of the programming-by-contract [69] paradigm to a distributed environment. Our hypothesis is that in order to properly account for potential conflicts of interest in a distributed environment with different administrative entities, the basic assumptions of traditional programming-by-contract must be reconsidered. These considerations give rise to a game-theoretic formulation of contracts, in which a programming contract between two entities assigns a quantitative measure of contract conformance, modelled as payoffs. The results in Chapter 3 are of a very fundamental character, and they do not give rise to an immediate implementation of distributed programming-by-contract. Rather, they lay the foundation for such an implementation, and they point out issues in generalising traditional programming-by-contract to a distributed setting that, to the best of our knowledge, have not been raised before.

At first sight, the connection between business contracts and contracts for distributed programming may seem absent. However, many aspects are in fact remarkably similar. For instance, in both scenarios a contract has to deal with real-time aspects, and in both scenarios we can assume no common goal of interest between the involved parties, nor access to the internal organisation of each party. Therefore, contracts are—in both scenarios—characterised by the idea that failures always happen in finite time. In other words, contracts are *safety properties* [4]. This, in turn, means that commitments cannot be *eventually guarantees*—commitments must be guarded by absolute deadlines.

The analogy between business contracts and distributed-programming contracts does not stop there. Whereas a programming contract is expected to be fulfilled by a program, a business contract can be fulfilled by a workflow. Hence in both cases programs and workflows represent *strategies* for fulfilling a contract, and the notions of *program correctness* and *contract compliance* coincide. In Chapter 3 we pursue the definition of correctness of a program with respect to a set of contracts, whereas in Chapter 2 we only consider contracts and not workflows. However, we believe that a proper treatment of workflow compliance will follow in the same lines. Namely, rather than defining workflow compliance with respect to a single contract, workflow compliance must be defined with respect to a set of contracts—a *contract portfolio*—where some contracts may be used to fulfil others. That is, a strategy consists both of a workflow (program) and a means of delegating contractual obligations to subcontractors.

## Modular Implementation of Domain-Specific Languages

The topic of the second part of the dissertation is *modular implementation of domain-specific languages* and *domain-specific languages for enterprise systems*. The latter topic is an effort at reaching the goal of the 3gERP project, whereas the former arose as a “by-product” of that attempt. (The results we present in this part of the dissertation are *applied computer science* compared to the first part of the dissertation.)

Modular implementation of domain-specific languages is covered in Chapters 4 and 5, and the results are readily usable in the form of the Haskell library *compositional data types*. Compositional data types target issues of code duplication and loss of invariants maintained by the type system, which may arise when implementing domain-specific languages. Consider for instance a scenario where we want to implement a transformation on abstract syntax trees that desugars an extended language to a core language. In order to reflect this invariant in the type system, we are forced—with traditional algebraic data types—to duplicate the parts of the extended language that corresponds to the core language as a separate type. Then, in turn, we have to duplicate functions that operate on the extended language to the core language. Alternatively, we may reuse the algebraic data type for the extended language also to represent the core language, but then the type no longer reflects the structure of the core language.

Compositional data types target the issues above—dubbed the *Expression Problem* by Wadler [114]—by letting us construct the data types of the extended language and the core language, as well as functions on them, in a modular and reusable fashion. Our approach takes the view of data types as fixed points of functors [67], and it extends Swierstra’s work on *data types à la carte* [104]. We introduce compositional data types in Chapter 4, and in Chapter 5 we introduce the extended *parametric compositional data types*. Parametric compositional data types target languages with variable binders, and we combine previous work by Fegaras and Sheard [25] and Chlipala [19] in order to define appropriate recursion schemes for structures with embedded binders.

The ability to define modular data types and modular functions, while retaining static type safety, is not only a useful tool when we want to implement a single domain-specific language. Compositional data types also provide an ideal tool when we want to implement multiple domain-specific languages that have pairwise, common components. In fact, this is the original motivation for introducing compositional data types, namely in order to implement an enterprise resource planning (ERP) system based on domain-specific languages, which is the topic of Chapter 6.

In Chapter 6 we tie together the two parts of the dissertation. In this chapter we introduce a novel software architecture for ERP systems based on domain-specific languages. The architecture extends the process-oriented event-driven transaction systems (POETS) architecture of Henglein et al. [41], and the system has been fully implemented using Haskell and the compositional data types library. What makes POETS novel—and radical—compared to traditional ERP systems, is a shift away from relational databases and imperative languages on the one hand, and from double-entry bookkeeping on the other hand. Instead, POETS relies on an ontological description of data and domain-specific languages (DSLs) for describing contracts (what should happen) and reports (what has happened) on the one hand,

and the resources, events, and agents (REA) accounting model [64] on the other hand. We use the contract specification language of Chapter 2 in order to specify contracts in our extended POETS architecture.

Besides extending the original POETS architecture and constructing a prototype implementation, our main objective is to conduct a use case that demonstrates how to implement—from scratch—a small, but realistic, ERP system. This amounts to defining an ontology for the ERP domain, as well as standard financial reports and simple purchase contracts and sales contracts. The use case shows, we believe, that the objective of the 3gERP project can be met, and that the extended POETS architecture is a potential candidate for such a new architecture. The amount of code needed to implement the small system is but a fraction of what would have to be implemented in state-of-the-art ERP systems—in fact, we have included the complete code in the dissertation. However, in order to confidently verify the hypothesis of the 3gERP project, with POETS as the constructive proof, a much larger and more realistic use case must be conducted, preferably in a live, industrial setting. At the time of writing, we are pursuing such a use case.

## Contributions

In summary, we see our main contributions as follows:

- We present a novel, trace-based model for multiparty (business) contracts that has blame assignment as its distinguishing feature. We show that our model faithfully captures real-world contracts by means of several examples. We construct a domain-specific language in order to concisely specify contracts, and we provide a formal semantics in terms of a mapping into the trace-based model. The semantics gives rise to incremental run-time monitoring of contracts.

- We investigate a fundamentally new extension of the programming-by-contract paradigm to a distributed setting. In order to convey our ideas, we present a formal model of processes, distributed-programming contracts, and a notion of contract conformance. Our definition of contract conformance takes the possibility of delegation into account, by considering a contract portfolio rather than a single contract. This has—to the best of our knowledge—not been investigated previously.

- We introduce a rich Haskell library for constructing modular and extendable data types, as well as modular and extendable functions on them. The primary application of our library is in the context of language implementation, in which the type of abstract syntax trees can advantageously be represented as a modular data type. Besides eliminating boilerplate code, our library introduces a novel implementation of annotated data types, automatic deforestation, and higher-order variable binders.

- We present a novel software architecture for enterprise resource planning (ERP) systems, based on domain-specific languages. We show how to implement the core features of a small ERP system from scratch. This amounts to defining an ontology (data model) for the ERP domain, report specifications for deriving

information, and contract specifications for describing daily activities—all in domain-specific languages. The amount of code needed for the implementation is but a fraction of the code in normal ERP systems, and the domain-orientation makes it—we believe—much easier for non-programmers to understand, and verify correctness of, the implementation.

## Acknowledgements

Summarising four years studies, there are many people I wish to thank. First and foremost, I thank my supervisor Fritz Henglein for his continuous source of inspiration and dedication. Had it not been for Fritz's compelling visions, I would not have indulged in topics such as business contracts and enterprise systems in the first place. Yet, the focus on conducting applied theoretical research turned out to be a key motivation for me.

I thank my co-supervisor Andrzej Filinski for always taking his time listening to my ideas, and for making me cherish conciseness, clarity, and mathematical rigour. I give all credit to Andrzej and Fritz for me starting the Ph.D. studies, and for opening the fascinating world of programming languages to me.

I gratefully acknowledge the coauthors who have contributed directly to the contents of my dissertation: Jesper Andersen, Patrick Bahr, Felix Klaedtke, and Eugen Zălinescu. Furthermore, I acknowledge Andrzej Filinski and Anders Starcke Henriksen who have contributed ideas, on which some content in the dissertation is based.

A wholehearted thanks goes to the Information Security Group at ETH Zürich, for hosting me for half a year. In particular, I thank Professor David Basin for inviting me to visit his research group in the first place, and for making me feel most welcome and at home at ETH.

On a more personal note, I thank past and current colleagues at DIKU for providing a great social environment, and for making my years at DIKU a memorable experience. I thank my fellow Ph.D. students and members of the 'lunch club' for providing a better social environment than I could possibly have hoped for.

Thanks to my parents for always supporting me, and for always believing in me.

Thank you, Annemette, for your love and support, and for joining me on the journey—not at least the half year in Zürich. Thank you for making all this possible.

Tom Hvitved
*Copenhagen, Denmark*
*November 2011*

# Part I

# Contract Formalisation

# Chapter 1

# A Survey of Formal Languages and Models for Contracts[⋆]

**Abstract**

We present the current status of languages and models for formalising contracts. A contract is a legally binding agreement between two or more parties, and a formalisation is an unambiguous, machine-interpretable representation. Formal representation of contracts is a prerequisite for supporting automatic contract lifecycle management (CLM), which includes business-critical activities such as validation, execution, and analysis of contracts. Starting from example contracts, we derive a set of features that contract formalisms should ideally support, based on which we carry out a comparative analysis of existing contract formalisms. Not surprisingly, none of the existing formalisms support all requirements. More surprisingly, the majority of existing approaches lack formal mathematical underpinnings, rendering these approaches immature for expressing legally binding agreements, as well as for performing contract analysis. We conclude our survey with a brief account of commercial CLM products.

## 1.1 Introduction

Contracts are legally binding agreements between parties. In e-business, contracts play an important role by stipulating commitments between the involved parties, for instance as term of sales agreements between web shops and customers. More generally, contracts serve as descriptions of what a business is expected to "deliver" to its customers, as well as what the customers are expected to "deliver" in return. Consequently, it is crucial for businesses to manage their portfolios of contracts in order to monitor for compliance—both by the businesses themselves, but also their customers. *Contract lifecycle management* (CLM) is the business terminology used to cover the activity of managing a contract portfolio, namely contract creation, contract negotiation, contract approval, contract execution, and contract analysis. Empirical studies conducted by the Aberdeen Group [84, 85] conclude that CLM will be a critical key to success for businesses in the near future. In the studies it is reported that around 80 percent of the surveyed enterprises (220 participants) are exercising only manual, or partially automated contract management activities, the

[⋆] *Extended version of a previous short paper [50].*

implication of which is a lower rate of transactions that are compliant with contracts. This in turn implies potential financial penalties:

> "[...] the average savings of transactions that are compliant with contracts is 22%" [84, page 1].

The conclusion of the Aberdeen Group studies is a list of *required actions* [85], which serve as recommendations for implementing the CLM methodology. The key recommendations are:

(A1) Establish standardised and formal contract management processes, including a standard language for contracts accessible via libraries and templates.

(A2) Clearly define protocols for the complete contracting process and contract administration (such as contract signing and contract execution).

(A3) Use reporting and analytic capabilities on contract data to gain competitive advantage.

Recommendations A1 and A2 capture the technical requirements of a CLM system, and recommendation A3 summarises the business potential of automated CLM. Recommendation A1—which covers the main topic of our survey—is, in fact, a prerequisite for achieving A2 and A3, since representations of the actual contracts are needed in order to execute them and perform analyses on them. As a complement to our survey, Tan et al. [106] provide an overview of CLM features, which focuses on the aspects of recommendations A2 and A3, rather than on formal languages for contracts. We will briefly return to CLM features in Section 1.4, where we supplement the survey of Tan et al. with an overview of current CLM products and their features.

## 1.2 Contract Formalisation Requirements

Formal specification of contracts and automatic reasoning about contracts has drawn interest from a wide variety of research areas within computer science, going back to the late eighties with the pioneering work by Lee [58]. Rather than presenting the existing work on contract formalisation ad hoc, we aim at a comparative analysis in order to differentiate the existing approaches. In this section we therefore present a list of features that "ideal" contract formalisms should support. This list not only allows us to compare existing approaches, but it also provides guidelines for constructing new formalisms.

Before we proceed with identifying requirements, we make an important remark on terminology. A formal *model* refers to a semantic, mathematical model, and a formal *language* refers to a syntactic representation. Ideally, a formal language co-exists with a formal model in terms of the language's *semantics*, which is a mapping of syntactic contracts to objects of the model. Such semantics is a prerequisite if the purpose of a contract language is to write legally binding agreements, since the involved parties must agree on the meaning of the contract. Moreover, in order to perform contract analysis, for instance to check whether two contracts have the same meaning, then a semantics is needed. We include these considerations as our first "meta" requirement (for easy reference, we label the $n$th identified requirement R$n$):

**Paragraph 1.** Seller agrees to transfer and deliver to Buyer, on or before 2011-01-01, the goods: 1 laser printer.

**Paragraph 2.** Buyer agrees to accept the goods and to pay a total of €200 for them according to the terms further set out below.

**Paragraph 3.** Buyer agrees to pay for the goods half upon receipt, with the remainder due within 30 days of delivery.

**Paragraph 4.** If Buyer fails to pay the second half within 30 days, an additional fine of 10% has to be paid within 14 days.

**Paragraph 5.** Upon receipt, Buyer has 14 days to return the goods to Seller in original, unopened packaging. Within 7 days thereafter, Seller has to repay the total amount to Buyer.

Figure 1.1: A sales contract between a buyer and a seller.

*(R1) Contract model, contract language, and a formal semantics.*

We will refer to models and languages collectively as *formalisms*, and most of the requirements that we identify in the following pertain to both models and languages.

In order to identify requirements for contract formalisms, we must first make it clear what we mean by a contract.

**Definition 1.2.1.** A *contract* is a legally binding agreement between two or more parties. The agreement is a normative description of commitments between the parties of the contract, that is a contract describes the expected actions to be performed by the participants of the contract.

We follow Prisacariu and Schneider [93] and restrict contracts to *ought-to-do* rather than *ought-to-be*, compare the definition above. That is, a contract is a description of what may/must/must not be performed, rather than what may/-must/must not be the state of affairs.

As an example of a contract according to Definition 1.2.1, consider the sales contract in Figure 1.1. The contract is a bilateral agreement between a buyer and a seller, and the normative content of the contract describes how—and when—the sale of one laser printer, in return for €200, is to be carried out. We will use the contract in Figure 1.1 to derive our first requirements:

*(R2) Contract participants.*
*(R3) (Conditional) commitments.*
*(R4) Absolute temporal constraints.*
*(R5) Relative temporal constraints.*
*(R6) Reparation clauses.*

Modelling of contract participants (R2) and commitments (R3) are self-nominated requirements, given our previous, informal definition of contracts. Yet, commitments may be conditional on what happens during the execution of a contract, as illustrated in Paragraph 5, where Seller's commitment to repay Buyer is contingent upon Buyer returning the printer. Seller's obligation to deliver goods in Paragraph 1 is an example of a commitment, but it is also an example of an absolute deadline (R4). Paragraph 3, on the other hand, exemplifies relative deadlines (R5), where Buyer's commitment to pay the first half depends on the time of delivery. Finally,

**Paragraph 1.** The term of this lease is for 6 months, beginning on 2011-01-01. At the expiration of said term, the lease will automatically be renewed for a period of one month unless either party (Landlord or Tenant) notifies the other of its intention to terminate the lease at least one month before its expiration date.

**Paragraph 2.** The lease is for 1 apartment, which is provided by Landlord throughout the term.

**Paragraph 3.** Tenant agrees to pay the amount of €1000 per month, each payment due on the 7th day of each month.

**Paragraph 4.** The rent is adjusted annually according to the Consumer Price Index (CPI).

Figure 1.2: A lease agreement between a tenant and a landlord.

**Paragraph 1.** Buyer agrees to pay to Seller the total sum €10000, in the manner following:

**Paragraph 2.** €500 is to be paid at closing, and the remaining balance of €9500 shall be paid as follows:

**Paragraph 3.** €500 or more per month on the first day of each and every month, and continuing until the entire balance, including both principal and interest, shall be paid in full; provided, however, that the entire balance due plus accrued interest and any other amounts due here-under shall be paid in full on or before 24 months.

**Paragraph 4.** Monthly payments shall include both principal and interest with interest at the rate of 10%, computed monthly on the remaining balance from time to time unpaid.

Figure 1.3: An agreement to pay in instalments between a buyer and a seller.

Paragraph 4 is an example of a reparation clause (R6), which comes into effect when another clause is not fulfilled—in this case the second part of Paragraph 3.

Consider next the lease agreement in Figure 1.2. Compared to the contract in Figure 1.1, the lease agreement features three new aspects:

*(R7) Instantaneous and continuous actions.*

*(R8) Potentially infinite and repetitive contracts.*

*(R9) Time-varying, external dependencies (observables).*

The distinction between instantaneous and continuous actions (R7) is witnessed by Paragraphs 2 and 3, respectively. The payment in Paragraph 3 is an instantaneous action, whereas the commitment to provide the apartment in Paragraph 2 is an ongoing action. More interestingly, the lease agreement is an example of a potentially infinite contract (R8), since the contract is renewed by default every month, unless the tenant or the landlord decide to terminate the agreement. By potentially infinite we hence mean that a latest time of termination is not known in advance when the contract is signed—the contract can, in principle, run forever. Last, but not least, Paragraph 4 illustrates how contracts may depend on external, time-varying observables (R9). We make a distinction between observables (R9) and conditional commitments (R3): the latter refers to conditions that can be controlled by the participants of the contract, whereas the former cannot.

The last example we consider is an instalment sale in Figure 1.3. For simplicity, the example only includes the payment part of the sale, and not Seller's obligations, which would be similar to the contract in Figure 1.1. The instalment sale is a slightly more complex contract than the previous example contracts, and it is a good example of a real-world contract, whose exact terms will be easier to understand once it is formalised. Unlike the contract in Figure 1.2, this contract is not potentially infinite, but instead it has a *bounded* repetitive pattern: the instalments must be paid within 24 months. Also, unlike the lease contract that has a fixed repetition pattern, the instalment contract dictates monthly commitments that depend on what has been paid in the previous months (referred to as the balance).

*(R10) History-sensitive commitments.*

*(R11) In-place expressions.*

History-sensitivity (R10) refers to the above-mentioned fact that commitments may depend on what has previously happened during contract execution. History-sensitivity (R10) differs from conditional commitments (R3) in that the latter refers to commitments that may or may not become active, whereas the former refers to commitments where the exact terms depend on previous events. With in-place expressions (R11) we mean the possibility to write (in this case arithmetic) expressions in contracts. In-place expressions are needed in order to disambiguate contracts like the one in Figure 1.3, where for instance the calculation of interest rate should be defined explicitly.

Requirements R2–R11 are devised more or less by a syntactic inspection of the example contracts. The following, last requirements focus instead on properties of a contract formalism than on what it should be able to model:

*(R12) Parametrised contracts.*

*(R13) Isomorphic encoding.*

*(R14) Run-time monitoring.*

*(R15) Blame assignment.*

*(R16) Amenability to (compositional) analysis.*

Parametrised contracts (R12) and isomorphic encodings (R13) are—in some sense—convenience features. Parametrised contracts are useful for defining contract templates, from which concrete instances can be derived, and it is certainly a feature that is required in practice. The isomorphism principle (R13) is originally introduced in the context of legal text formalisation [12]. The principle refers to formal encodings that are in one-to-one correspondence with the informal paper contracts. That is, one component (paragraph) in the paper contract corresponds to one component in the formalisation, and dependencies between components in the paper contract are present between the corresponding components in the formalisation. Like contract templates, the isomorphism principle is convenient in practice: (local) changes in the paper contract correspond to (local) changes in the formalised contract and vice versa. Furthermore, a formal encoding that is reminiscent of the paper contract is likely easier for domain experts to maintain, and errors in the formalisation process are less likely to be introduced.

Run-time monitoring (R14) of contracts is the single most important requirement, compare the Aberdeen group studies [84]. Run-time monitoring [59] is the ability to monitor the execution of a contract in real-time, that is to check whether the contract has been breached and to report upcoming commitments in order to avoid future breaches of contract. Blame assignment (R15) is closely related to run-time monitoring. In the case where a contract is breached, the monitor should not only report a breach of contract, but also who among the contract participants is responsible. If a contract does not uniquely determine who is to be blamed when contract execution fails, then contract parties may have to go to court to solve the dispute—this is exactly the situation we want to avoid with formalised contracts.

The final requirement that we have included is amenability to analysis (R16). Although this requirement is already listed as a general CLM requirement (A3), the language that defines the representation of contracts will determine which analyses are possible. Examples of analyses that are relevant for contracts include (1) satisfiability, that is whether a contract can be fulfilled, (2) satisfiability with respect to a particular party, that is whether a party can avoid breaching a contract in which it is involved, (3) contract valuation, that is what is the expected value of a contract for a given party, and (4) contract entailment, that is whether fulfilling a contract entails the fulfilment of another contract. Lastly, remark that by compositionality we refer to analyses that are defined universally for all possible contracts that can be expressed in the formalism. That is, analyses that are not defined ad hoc, such as valuation of one particular type of instalment sale.

We now have a total of 16 formalism requirements. These requirements are likely not representative for *all* possible contracts, yet each requirement represents an important aspect, and—we believe—covers most aspects found in contracts. We again remind the reader that these requirements are for contract formalisms, and not for CLM systems in general. The latter requires features such as contract drafting, electronic contract signing, contract versioning, etc., which we will return to briefly in Section 1.4.

## 1.3 Contract Formalisms

The majority of existing work on contract formalisation falls into three categories: (deontic) logic based formalisms [35, 58, 93], event-condition-action based formalisms [33, 60], and trace based formalisms [6, 57]. The logic based approaches mainly focus on declarative specification of contracts, and on (meta) reasoning, such as decidability of the logic. On the other hand, the event-condition-action and trace based formalisms focus mainly on contract execution. Other approaches to contract modelling include functional programming [86], defeasible reasoning [34, 36], finite state machines [74], and more informal frameworks [17, 21, 80, 117, 121, 122]. Common to all approaches is the goal of modelling electronic contracts in general, except for Peyton Jones and Eber [86] and Andersen et al. [6] who specifically consider financial contracts and commercial contracts, respectively.

In the following we will go through the contract formalisms above in more detail. We will assess the formalisms in terms of the requirements we identified in the previous section, and—where it is possible—provide sample encodings of the contract in Figure 1.1. Since deontic logic [111]—the logic of obligations, permissions, and prohibitions—is a reoccurring theme, we enclose a brief introduction to deontic logic in Appendix A.1.

### 1.3.1 Logic Programming

The seminal work by Lee on *electronic contracting* [58] is the first attempt to formalise business contracts. Contracts are viewed as Petri net transition systems, where a set of states can be active at any point in time, and events—that is, actions performed by contract parties—can trigger new states to become active.

At the core of Lee's approach is to model the transition system in logic programming, and to model contract concepts such as obligations in terms of transitions. More specifically, the transition relation is modelled as a predicate $\mathrm{trans}(A, B, E)$, which should be read "if precondition $A$—a predicate on the current active states—holds, and event $E$ occurs, then postcondition $B$—defining the new active states—holds". As an example,

$$\mathrm{trans}([s(0)], [s(1), s(2)], X : D : A)$$

models that states 1 and 2 become active, if state 0 is active and action $A$ is taken by party $X$ fulfilling some deadline $D$. (We use a slightly simplified syntax compared to the original article [58].) Note that, as in Petri nets, the active states that are used as preconditions in a transition become inactive after the transition, that is when states 1 and 2 above are activated then state 0 is deactivated.

As an example of how contract aspects are modelled, consider the obligation for party $X$ to perform action $A$ within a deadline specified by some quantifier $D$:

$$\begin{array}{l}\mathrm{trans}([s(0)], [s(1)], X : D : A)\\ \mathrm{trans}([s(0)], [s(\mathrm{default}(X))], \sim X : D : A).\end{array}$$

Here the state $\mathrm{default}(X)$ means that party $X$ has defaulted on the contract, and $\sim X : D : A$ means that party $X$ has not performed action $A$. Hence if party $X$ is obliged to perform action $A$ in state 0, then by doing so the contract progresses to the successor state 1, and by not doing so the contract enters a default state, which means a breach of contract. Note how the absence of the action is encoded explicitly, rather than via negation as failure. The motivation is that only when we know explicitly that action $A$ was not performed by $X$ is the contract breached.

An important point about Lee's modelling of obligations is the relation to standard deontic logic (SDL). Although inspired by SDL, Lee makes a clear point that the semantics is not the usual possible worlds semantics of SDL (Appendix A.1), but rather that it only makes sense to consider one world, namely the actual circumstances of the contract. By abandoning the possible worlds semantics, Lee derives the model of obligations that we saw above, which conforms intuitively with contractual obligations, unlike the more philosophical semantics of SDL.

Similar to obligations, Lee shows how to model a rich set of features in terms of electronic contracts, namely modelling of contract participants (R2), commitments (R3), absolute temporal constraints (R4), relative temporal constraints (R5), instantaneous and continuous actions (R7), potentially infinite and repetitive contracts (R8) and parametrised contracts (R12). There is no account for reparation clauses (R6), time-varying, external dependencies (R9), history-sensitive commitments (R10), and in-place expressions (R11) in electronic contracts. However, with the exception of R11 these aspects are likely definable within the model as well, given the freedom of logic programming.

The logic programming encoding of contracts provides the back end for formalising contracts, and for the purpose of writing contracts and monitoring their execution (R14), Lee sketches how to map an English-like text (R13) to the logic programming model. By utilising the logic programming query engine, the state of a contract can be analysed (R16), for instance to see who (if any) have outstanding commitments, which provides some means of blame assignment as well (R15).

$$\mathrm{trans}([s(1)], [s(2), s(3), s(4)], \mathrm{Seller} : \mathrm{rb}(\text{1-jan-2011}) : \mathrm{deliver}(\mathrm{Buyer}, \mathrm{printer})) \tag{1.1}$$
$$\mathrm{trans}([s(1)], [s(\mathrm{default}(\mathrm{Seller}))], \sim \mathrm{Seller} : \mathrm{rb}(\text{1-jan-2011}) : \mathrm{deliver}(\mathrm{Buyer}, \mathrm{printer})) \tag{1.2}$$
$$\mathrm{trans}([s(2)], [s(5)], \mathrm{Buyer} : \mathrm{rw}(0) : \mathrm{pay}(\mathrm{Seller}, €100)) \tag{1.3}$$
$$\mathrm{trans}([s(2)], [s(\mathrm{default}(\mathrm{Buyer}))], \sim \mathrm{Buyer} : \mathrm{rw}(0) : \mathrm{pay}(\mathrm{Seller}, €100)) \tag{1.4}$$
$$\mathrm{trans}([s(3)], [s(6)], \mathrm{Buyer} : \mathrm{rw}(30) : \mathrm{pay}(\mathrm{Seller}, €100)) \tag{1.5}$$
$$\mathrm{trans}([s(3)], [s(7)], \sim \mathrm{Buyer} : \mathrm{rw}(30) : \mathrm{pay}(\mathrm{Seller}, €100)) \tag{1.6}$$
$$\mathrm{trans}([s(7)], [s(8)], \mathrm{Buyer} : \mathrm{rw}(14) : \mathrm{pay}(\mathrm{Seller}, €110)) \tag{1.7}$$
$$\mathrm{trans}([s(7)], [s(\mathrm{default}(\mathrm{Buyer}))], \sim \mathrm{Buyer} : \mathrm{rw}(14) : \mathrm{pay}(\mathrm{Seller}, €110)) \tag{1.8}$$
$$\mathrm{trans}([s(4)], [s(9)], \mathrm{Buyer} : \mathrm{rw}(14) : \mathrm{return}(\mathrm{Seller}, \mathrm{printer})) \tag{1.9}$$
$$\mathrm{trans}([s(9)], [s(10)], \mathrm{Seller} : \mathrm{rw}(7) : \mathrm{pay}(\mathrm{Buyer}, €100)) \tag{1.10}$$
$$\mathrm{trans}([s(9)], [s(\mathrm{default}(\mathrm{Seller}))], \sim \mathrm{Seller} : \mathrm{rw}(7) : \mathrm{pay}(\mathrm{Buyer}, €100)) \tag{1.11}$$

Figure 1.4: A sales contract between a buyer and a seller (logic programming).

Lee's approach succeeds in capturing most of the aspects we identified in Section 1.2, and—as we shall see—later approaches often fall short on many of the aspects that Lee's model covers. In particular, Lee argues that the possible worlds semantics of standard deontic logic is inappropriate for modelling contracts, which is overlooked in later work. Indeed, none of the later work that claims to be based on deontic logic demonstrates that the possible worlds semantics is appropriate for modelling contracts. The reason—we believe—why the features of Lee's electronic contracts and modelling of deontic modalities are rather overlooked in later work, is the absence of a clear language definition, as well as a direct semantic model of electronic contracts (R1).

We conclude with an example encoding of the sales contract from Figure 1.1 in Figure 1.4 (we refer to the original article [58] for more information on the concrete syntax). In the initial state Seller has to deliver within 2011-01-01, which activates states 2, 3, and 4 (1.1). The quantifier "rb" abbreviates "realised before", and similarly "rw" abbreviates "realised within". As we saw above, failure to comply with an obligation is modelled with a default state (1.2). In state 2 Buyer has to pay the first half immediately (1.3) and in state 3 Buyer has to pay the second half within 30 days (1.5). However, failure to comply with the latter payment does not result in a default state, but rather a new state in which Buyer has to pay €110 (1.6). Lastly, the permission of Buyer to return the goods to Seller is modelled as a transition that enables an obligation on Seller (1.9).

### 1.3.2 Event-condition-action

The event-condition-action (ECA) paradigm from active databases [13] is first used by Goodchild et al. [33] for modelling contracts. The ECA interpretation of contracts is that events trigger actions when certain conditions are met. To be more concrete, the ECA interpretation of for instance Paragraph 1 in Figure 1.1 is (1) when the event "contract start" takes place, then (2) the action "Buyer delivers printer to Seller" should happen, provided that (3) the deadline of 2011-01-01 is not passed.

Goodchild et al. model contracts as sets of policies. A policy specifies that

$S = \mathit{Contract.Seller}$;
$B = \mathit{Contract.Buyer}$;

**when** $\mathit{Contract.State} == \text{'}\mathit{initial}\text{'}$
**action**(*deliver*, $S$, $B$, $t$)
**must occur where** $t \leq 2011-01-01$
**otherwise trigger**(*send_notice_of_breach*, $*$, "Seller failed to deliver goods")

**when** { *delivery has occurred* }
**action**(*pay_first_half*, $B$, $S$, $t$)
**must occur where** { *t same day as delivery date* }
**otherwise trigger**(*send_notice_of_breach*, $*$, "Buyer failed to pay first half")

**when** { *first half paid* }
**action**(*pay_second_half*, $B$, $S$, $t$)
**must occur where** { *t is no later than 30 days after delivery* }
**otherwise trigger**(*send_notice_of_breach*, $*$, "Buyer failed to pay second half")

Figure 1.5: A sales contract between a buyer and a seller (event-condition-action).

a legal entity is either forbidden or obliged to perform an action under certain conditions. The grammar for policies is as follows (keywords are in bold face, [·] denotes optionality, and $\cdot^*$ denotes zero or more occurrences):

$$\begin{array}{lcl}
\mathit{Policy} & ::= & \mathit{VariableDeclaration}^* \\
 & & \textbf{when}\ \mathit{Condition} \\
 & & \mathit{Action} \\
 & & \textbf{must}\ [\textbf{not}]\ \textbf{occur where}\ \mathit{Condition} \\
 & & \textbf{otherwise}\ \mathit{Trigger} \\
\mathit{Action} & ::= & \textbf{action}(\mathit{ActionName}, \mathit{Actor}, \mathit{Audience}, \mathit{Time}) \\
\mathit{Trigger} & ::= & \textbf{trigger}(\mathit{ActionName}, \mathit{Audience})
\end{array}$$

Unfortunately, there is no semantic account for the policy language, nor a detailed description of the syntax besides the incomplete grammar above (R1). In particular, we would expect that it is possible for actions to update the state of the contract, but how this is done is not described. Despite the lack of detail, the policy language provides some hints of how ECA can be used to model contracts. Figure 1.5 sketches how to encode Paragraphs 1–3 from Figure 1.1 as policies. Paragraphs 4–5 cannot be encoded, as the policy language does not include reparation clauses and permissions, respectively. We use pseudo-notation, enclosed in curly braces, to make up for the lacking language constructs.

Given the lack of detail in the presentation of Goodchild et al., it is difficult to asses the policy language with respect to our requirements from Section 1.2. However, the supplied examples [33] indicate that the policy language includes the aspects of commitments (R3), absolute temporal constraints (R4), in-place expressions (R11), and run-time monitoring (R14) in terms of an SQL implementation.

$$\begin{array}{lcll}
l_1 & : & \mathit{init} & \rightarrow O_{\text{Seller,Buyer}}(\mathit{deliver}(\mathit{printer}, t_1) < \text{2011-01-01}) \\
l_{3.1} & : & \mathit{fulfilled}(l_1) & \rightarrow O_{\text{Buyer,Seller}}(\mathit{pay}(100) < t_1) \\
l_{3.2} & : & \mathit{fulfilled}(l_1) & \rightarrow O_{\text{Buyer,Seller}}(\mathit{pay}(100) < t_1 + 30) \\
l_4 & : & \mathit{not\_fulfilled}(l_{3.2}) & \rightarrow O_{\text{Seller,Buyer}}(\mathit{pay}(110) < t_1 + 44) \\
l_{5.1} & : & \mathit{fulfilled}(l_1) & \rightarrow P_{\text{Buyer,Seller}}(\mathit{return}(\mathit{printer}, t_2) < t_1 + 14) \\
l_{5.2} & : & \mathit{fulfilled}(l_{5.1}) & \rightarrow O_{\text{Seller,Buyer}}(\mathit{pay}(200) < t_2 + 7)
\end{array}$$

Figure 1.6: A sales contract between a buyer and a seller (normative statements).

### 1.3.3 Normative Statements

In the e-contracts framework of Boulmakoul and Sallé [17], contracts are modelled as sets of normative statements, similar to Goodchild et al.'s policies (Section 1.3.2). Rather than using event-condition-action principles, Boulmakoul and Sallé start from deontic logic principles, that is from the deontic operators of SDL, but not the semantics.

As for the event-condition-action approach of Goodchild et al., there is no formal semantics of normative statements (R1), and the details are very sparse. Yet, the intuition behind normative statements is interesting, and the language is very compact. Normative statements have the form:

$$l : f \rightarrow D_{i_1,i_2}(a < T),$$

where $l$ is a label, $f$ is a predicate that may refer to other statements via their labels, $D$ is a deontic operator (either obligation $O$, permission $P$, or prohibition $F$), $i_1$ and $i_2$ are roles, $a$ is the action to (not) perform, and $T$ is a deadline. The intuitive reading of the statement above is "when $f$ holds, $i_1$ is obliged/permitted/prohibited by $i_2$ to achieve/perform $a$ before $T$".

Figure 1.6 sketches how to model the sales contract from Figure 1.1 as normative statements. Unlike standard deontic logic, actions may carry values rather than being restricted to propositional atoms. *init* represents the start of the contract, $\mathit{deliver}(g, t)$ represents delivery of goods $g$ at time $t$, $\mathit{pay}(x)$ is payment of amount $x$, and $\mathit{return}(g, t)$ is the returning of goods $g$ at time $t$.

Although lacking semantics and a precise language definition (R1), the encoding sketched above shows that the labelling approach yields a formalisation that is close in structure to the original paper contract (R13). Moreover, by annotating deontic modalities with parties (R2), we should expect to be able to perform some form of blame assignment (R15) when contracts are breached. Besides the requirements R2, R13, and R15, the policy language of Boulmakoul and Sallé supports commitments (R3), absolute temporal constraints (R4), reparation clauses (R6), and run-time monitoring (R14), although none of these features are given a detailed or formal treatment.

### 1.3.4 Functional Programming

Peyton Jones and Eber [86] consider a restricted form of contracts, namely bilateral financial contracts. In order to illustrate the domain of financial contracts, consider the example in Figure 1.7. Each $D_i$ represents—in principle—a contract. That is,

$D$ : The holder of this contract has the right to choose on 30 June 2000 between:

- $D_1$ : Both of:
  - $D_{11}$ : Receive £100 on 29 Jan 2001.
  - $D_{12}$ : Pay £105 on 1 Feb 2002.
- $D_2$ : An option exercisable on 15 Dec 2000 to choose one of:
  - $D_{21}$ : Both of:
    - $D_{211}$ : Receive £100 on 29 Jan 2001.
    - $D_{212}$ : Pay £106 on 1 Feb 2002.
  - $D_{22}$ : Both of:
    - $D_{221}$ : Receive £100 on 29 Jan 2001.
    - $D_{222}$ : Pay £112 on 1 Feb 2003.

Figure 1.7: Financial contract [86, page 1].

$D$ is a contract composed of (sub)contracts $D_1$ and $D_2$, which are in turn composed of (sub)contracts.

The compositional structure of financial contracts lend them appropriate for a formalisation in functional programming, which is compositional in nature. Peyton Jones and Eber construct a compact combinator library in Haskell [62], which consists of the following contract constructors (using non-Haskell syntax):

| | |
|---|---|
| $c ::= \mathit{zero}$ | (no rights/obligations) |
| $\mid \mathit{one}(k)$ | (right to one unit of currency $k$) |
| $\mid \mathit{give}(c)$ | (reverse the rights and obligations of $c$) |
| $\mid \mathit{and}(c_1, c_2)$ | (immediately acquire both $c_1$ and $c_2$) |
| $\mid \mathit{or}(c_1, c_2)$ | (immediately acquire either $c_1$ or $c_2$, but not both) |
| $\mid \mathit{cond}(o, c_1, c_2)$ | (immediately acquire $c_1$ if $o$ holds, otherwise $c_2$) |
| $\mid \mathit{scale}(o, c)$ | (immediately acquire $c$ where all amounts are scaled by $o$) |
| $\mid \mathit{when}(o, c)$ | (immediately acquire $c$ as soon as $o$ holds) |
| $\mid \mathit{anytime}(o, c)$ | (acquire $c$ once, anytime $o$ holds) |
| $\mid \mathit{until}(o, c)$ | (immediately acquire $c$, but abandon $c$ once $o$ holds) |

$k$ ranges over currencies, for instance USD, and $o$ ranges over observables, which are time-varying values (R9), for instance the LIBOR interest rate on a particular date.

Peyton Jones and Eber demonstrate how the compact library suffices for defining standard financial contracts such as *swaps* and *zero-coupon discount bonds*. Rather than having such contracts as atomic constructs, the combinator approach facilitates a uniform treatment of contract analysis. Since the combinator library is tailored specifically to financial contracts, we show how the example from Figure 1.7 is encoded in Figure 1.8, rather than the example contract in Figure 1.1. The function *isdate* is the observable that holds exactly on the supplied date, that is $\mathit{isdate}(d)$ holds only on the date $d$. The function *const* is the constant observable, that is $\mathit{const}(a)$ has the value $a$ at all times.

The strength of Peyton Jones and Eber's approach is the ability to perform compositional analysis of contracts expressed in the library (R16). For instance, Peyton Jones and Eber show how to perform a valuation analysis of any contract that can be expressed in the language, which essentially means interpreting contracts as two-player games, and estimating the expected outcome of the game (we refer to the two parties as Holder and Opponent, respectively):

$$
\begin{aligned}
c = {} & \mathbf{when}(\mathit{isdate}(2000{-}06{-}30),\\
& \quad \mathbf{or}(\mathbf{and}(\mathit{receive\_on}(2001{-}01{-}29,\ 100),\ \mathit{pay\_on}(2002{-}02{-}01,\ 105)),\\
& \quad\quad \mathbf{when}(2000{-}12{-}15,\\
& \quad\quad\quad \mathbf{or}(\mathbf{and}(\mathit{receive\_on}(2001{-}01{-}29,\ 100),\ \mathit{pay\_on}(2002{-}02{-}01,\ 106)),\\
& \quad\quad\quad\quad \mathbf{and}(\mathit{receive\_on}(2001{-}01{-}29,\ 100),\ \mathit{pay\_on}(2003{-}02{-}01,\ 112))))))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{receive\_on}(d,a) &= \mathbf{when}(\mathit{isdate}(d),\ \mathbf{scale}(\mathit{const}(a),\ \mathbf{one}(\mathit{GBP})))\\
\mathit{pay\_on}(d,a) &= \mathbf{give}(\mathit{receive\_on}(d,a))
\end{aligned}
$$

Figure 1.8: Financial contract (functional programming).

| | |
|---|---|
| $zero$ : | The game has ended. |
| $one(k)$ : | Holder wins one unit of currency $k$ from Opponent. |
| $give(c)$ : | Initiate game $c$ where Holder and Opponent switch roles. |
| $and(c_1, c_2)$ : | Games $c_1$ and $c_2$ are initiated (in parallel). |
| $or(c_1, c_2)$ : | Holder chooses one of $c_1$ and $c_2$, which is then initiated. |
| $cond(o, c_1, c_2)$ : | If $o$ holds, game $c_1$ is initiated, otherwise game $c_2$ is initiated. |
| $scale(o, c)$ : | Initiate game $c$ where all amounts are scaled by $o$. |
| $when(o, c)$ : | The game $c$ is initiated as soon as $o$ holds. |
| $anytime(o, c)$ : | Holder can initiate game $c$ (once) anytime $o$ holds. |
| $until(o, c)$ : | Game $c$ is initiated, but immediately ended once $o$ holds. |

With the game-theoretic interpretation, the expected value of for instance $give(c)$ is the negation of the expected value of $c$, while the expected value of $or(c_1, c_2)$ is the maximum of the expected values of $c_1$ and $c_2$. And in the case of observables, statistical forecasts are applied, that is there is a stochastic model for observables.

In terms of our desiderata from Section 1.2, the combinator library of Peyton Jones and Eber supports conditional commitments (R3), absolute temporal constraints (R4), relative temporal constraints (R5), potentially infinite and repetitive contracts (R8), time-varying, external dependencies (R9), in-place expressions (R11), parametrised contracts (R12), isomorphic encoding (R13), and amenability to (compositional) analysis (R16). The fact that financial contracts are expressed in Haskell means that R8, R11, and R12 come for free, for instance a potentially infinite contract is nothing but a recursively defined term, and a parametrised contract is a function.

### 1.3.5 Finite State Machines

Molina-Jimenez et al. [74] consider a finite state machine (FSM) encoding of contracts, referred to as executable contracts (x-contracts). X-contracts have some similarities with Lee's transition system approach (Section 1.3.1), yet x-contracts use FSMs more explicitly, and the modelling involves one FSM per contract party rather than one global contract description. In that respect, x-contracts are projections of global contract descriptions to each of the contract parties, although the projection is not given explicitly as in, for instance, the end-point projection of multiparty session types [46].

A finite state machine is defined as a tuple $(S, I, Z, \delta, \lambda)$, where $S$ is a finite set of states, $I$ and $Z$ are finite sets of input and output symbols respectively, $\delta : S \times I \to S$

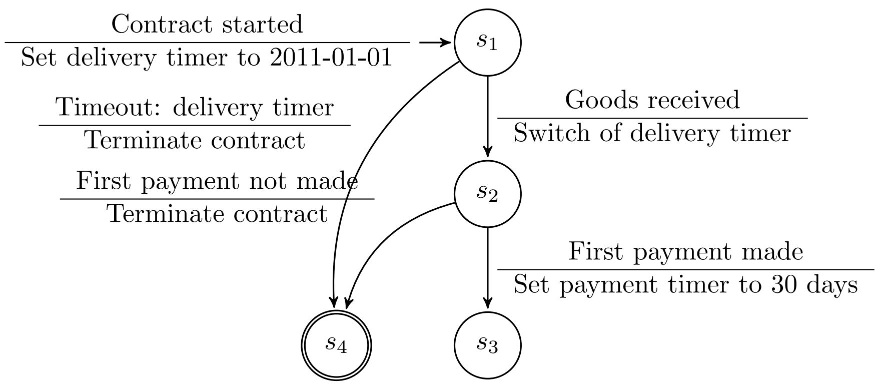


Figure 1.9: A sales contract between a buyer and a seller from the buyer's viewpoint (finite state machines).

is the transition function, and $\lambda : S \times I \to Z$ is the output function (hence the FSM is actually a Mealy machine, but Molina-Jimenez et al. use the FSM terminology). The state models the state of the contract, and the input symbols model events that happen relevant to the contract, for instance delivery of goods. The output symbols model the operation that the contract stipulates on the holder of the executable contract, for instance to pay in return for the delivered goods.

Figure 1.9 presents an encoding of a small fragment of the sales contract from Figure 1.1 as an x-contract, from Buyer's viewpoint. We use the same graphical notation as Molina-Jimenez et al., that is states are depicted as circles, and transitions are labelled arrows between states, where the event (input) is written above the operation (output).

In the initial state $s_1$ Seller is obliged to deliver goods, which from Buyer's viewpoint amounts to starting a timer to verify delivery within the agreed deadline. If the timer runs out, Buyer must terminate the contract, which results in the terminal state $s_4$, similar to Lee's default state (Section 1.3.1). If Seller delivers the goods, Buyer must pay immediately in state $s_2$, and doing so advances the contract to state $s_3$. (We do not present the encoding past state $s_3$.)

The encoding above is rather informal, however a similar encoding is used in the original paper [74]. For instance, we use timers to encompass deadlines even though timing aspects are not part of finite state machines. Moreover, the finite state machine encoding above does not take into account that Buyer can return the goods in state $s_2$. The reason why we have not encoded this option is because it raises a general problem with the FSM approach, namely how to model the case when two subcontracts are active simultaneously. In such cases we need to construct a product automaton, which yields $n \times m$ states when the sub automata have $n$ and $m$ states, respectively, which makes it incomprehensible to depict.

In terms of our requirements from Section 1.2, the FSM encoding used in x-contracts focuses entirely on a contract execution model, and not a language for writing contracts (R1). X-contracts support (conditional) commitments (R3) and relative temporal constraints (R5), and to some extent absolute temporal constraints (R4), even though absolute time is not part of the mathematical FSM model. Furthermore, x-contracts support reparation clauses (R6), potentially infinite and repetitive contracts (R8), run-time monitoring (R14), and amenability to (compositional) analysis (R16). Of these features run-time monitoring is the most prominent, since the execution model is built entirely with the purpose of monitoring contracts. Moreover, being based on finite state machines means that existing tools can be used for contract analysis, for instance to resolve ambiguities [74].

### 1.3.6 Business Contract Language

The business contract language (BCL) of Milosevic et al. [60, 72] is designed with the purpose of enabling event-based monitoring of business activities. A BCL contract consists of a set of roles along with a set of policies, and it is hence similar in structure to the event-condition-action (ECA) approach in Section 1.3.2. The roles define the parties involved in a contract, and the policies define the obligations and rights agreed upon by the parties. The first presentations of BCL [60, 72] contain no formal semantics, and the language is only presented fragment-wise by means of examples. Governatori and Milosevic [35] later seek to formalise BCL by mapping it to a fragment of deontic logic extended with contrary-to-duty obligations. The presentation we give here is based primarily on the later presentation.

The motivation for BCL is event-based monitoring. An event $e \in E$ is either (1) an action performed by one of the signatories of the contract, (2) a temporal occurrence such as the passing of a deadline, (3) a change in the contract state, or (4) a contract violation. (1) covers "real-world" events, that is actions that are actually performed, and (2–4) are "control" events, that is events that are used for executing the contract, and which are defined by the contract in an ECA manner.

An event pattern $ep \in EP$ is either (1) a logical relation between events, such as $\neg e$ and $e_1 \wedge e_2$; (2) a temporal relation between events, such as $e_1$ *before* $e_2$; or (3) a temporal constraint on event patterns, such as *ep before* $t$, where $t$ is a point in time. Whereas events are atomic, event patterns are used to describe complex events. For instance, a policy may require that a certain role is obliged to perform either of two actions $e_1$ and $e_2$, which is achieved via the event pattern $e_1 \vee e_2$.

The main ingredient of BCL is policies. A policy consists of (1) a policy name; (2) a role, that is to whom does the policy apply; (3) the modality of the policy, either an obligation, a permission, or a prohibition; (4) a trigger that defines when the policy is active in terms of a set of event patterns; (5) an optional guard that—like the trigger—specifies when the policy is active, but—unlike the trigger—refers to the contract state and to other policies; and (6) the obliged/permitted/prohibited behaviour dictated by the policy in terms of an event pattern.

The grammar for BCL, which we described informally above, is as follows:

$$
\begin{array}{lcl}
\mathit{Contract} & ::= & \mathit{Policy}^* \\
\mathit{Policy} & ::= & \textbf{Policy} : \ \mathit{Name} \\
 & & \textbf{Role} : \ \mathit{Role} \\
 & & \textbf{Modality} : \ \mathit{Modality} \\
 & & \textbf{Trigger} : \ \mathit{EP}^+ \\
 & & [\mathit{Guard}] \\
 & & \textbf{Behavior} : \ \mathit{EP} \\
\mathit{Modality} & ::= & \textbf{Obligation} \mid \textbf{Permission} \mid \textbf{Prohibition} \\
\mathit{EP} & ::= & \textbf{not}\ E \mid E\ \textbf{and}\ E \mid E\ \textbf{or}\ E \mid E\ \textbf{before}\ E \mid \mathit{EP}\ \textbf{before}\ T \\
\mathit{Guard} & ::= & \textbf{Guard} : \ \mathit{StateExp} \mid \textbf{violated}(\mathit{Name})
\end{array}
$$

**Policy**: *P1*
**Role**: *Seller*
**Modality**: **Obligation**
**Trigger**: *init*
**Behavior**: *deliver* **before** 2011−01−01

**Policy**: *P3*.1
**Role**: *Buyer*
**Modality**: **Obligation**
**Trigger**: *deliver*
**Behavior**: *pay_first_half*

**Policy**: *P3*.2
**Role**: *Buyer*
**Modality**: **Obligation**
**Trigger**: *deliver*
**Behavior**: *pay_second_half* **before** 30

**Policy**: *P4*
**Role**: *Buyer*
**Modality**: **Obligation**
**Guard**: **violated**(*P3*.2)
**Behavior**: *pay_penalty* **before** 14

**Policy**: *P5*.1
**Role**: *Buyer*
**Modality**: **Permission**
**Trigger**: *deliver*
**Behavior**: *return_goods* **before** 14

**Policy**: *P5*.2
**Role**: *Seller*
**Modality**: **Obligation**
**Trigger**: *return_goods*
**Behavior**: *repay* **before** 7

Figure 1.10: A sales contract between a buyer and a seller (business contract language).

As an example of a BCL contract according to the grammar, consider the encoding of the sales contracts from Figure 1.1 in Figure 1.10. The encoding, which is reminiscent of the structure of the paper contract (R13), uses the events *init*, *deliver*, *pay_first_half*, *pay_second_half*, *pay_penalty*, *return_goods*, and *repay*, which are propositional atoms à la standard deontic logic (SDL), that is they carry no values. Unlike SDL, however, BCL uses temporal aspects which means that the semantics of SDL (Appendix A.1) does not apply immediately to BCL.

Besides the isomorphism principle (R13), BCL features contract participants (R2), conditional commitments (R3), absolute temporal constraints (R4), relative temporal constraints (R5), reparation clauses (R6), and run-time monitoring (R14). Run-time monitoring of BCL contracts is referred to as business activity monitoring (BAM) [72]. The BAM engine executes BCL contracts by processing external events, as well as generating internal events, for instance when deadlines pass.

In terms of a semantic model of contracts (R1), Governatori and Milosevic [35] introduce the formal contract logic (FCL). FCL extends SDL by annotating deontic modalities with responsibility and by introducing a restricted form of contrary-to-duty obligations [92]. The former extension entails that obligations have the form $\mathrm{O}_s\phi$, which means that $s$ is responsible for the obligation $\phi$, and the latter extension is the ability to model secondary obligations that become active when primary obligations are violated. Unlike Prakken and Sergot's original extension of SDL with contrary-to-duty obligations [92], FCL has the following restricted grammar:

$$
\begin{array}{lll}
l & ::= p \mid \neg p & \text{(literals)} \\
ml & ::= \mathrm{O}_s l \mid \neg\mathrm{O}_s l \mid \mathrm{P}_s l \mid \neg\mathrm{P}_s l & \text{(modal literals)} \\
\otimes\text{-}exp & ::= ml \mid \mathrm{O}_{s_1} l_1 \otimes \cdots \otimes \mathrm{O}_{s_n} l_n & \text{($\otimes$-expressions)} \\
 & \quad\mid \mathrm{O}_{s_1} l_1 \otimes \cdots \otimes \mathrm{O}_{s_n} l_n \otimes \mathrm{P}_{s_{n+1}} l_{n+1} & \\
\phi & ::= l \to \phi \mid ml \to \phi \mid \otimes\text{-}exp & \text{(policies)}
\end{array}
$$

Literals $l$ are propositional atoms and events, and the binary connective $\otimes$ is used to represent contrary-to-duty structures. Unlike SDL, deontic operators only pertain to literals, which means that FCL is restricted to ought-to-do statements [93] (as opposed to ought-to-be statements). A (binary) contrary-to-duty obligation $\mathrm{O}_{s_1}l_1 \otimes \mathrm{O}_{s_2}l_2$ should be read "$s_1$ is obliged to perform $l_1$, and failure to comply obliges $s_2$ to perform $l_2$". Similarly, a (binary) contrary-to-duty permission $\mathrm{O}_{s_1}l_1 \otimes \mathrm{P}_{s_2}l_2$ means that $s_2$ is permitted to perform $l_2$ if $s_1$ fails to perform $l_1$.

Following the original intent of standard deontic logic with contrary-to-duty obligations [92], a contrary-to-duty obligation $\mathrm{O}_{s_1}l_1 \otimes \mathrm{O}_{s_2}l_2$ is not the same as a simple disjunction $\mathrm{O}_{s_1}l_1 \vee \mathrm{O}_{s_2}l_2$, that is $\otimes$ is non-commutative. Rather, Governatori and Rotolo suggest the intuitive reading that a contrary-to-duty obligation corresponds to "[...] a disjunction where the order of disjuncts matters" [37, page 198]. That is, there is an implicit agreement first and foremost to fulfil the primary obligation, and only if there are no other possibilities must the secondary obligation be fulfilled. Hence, none of our example contracts in Figures 1.1–1.3 contain *actual* contrary-to-duty obligations—for instance the penalty of Paragraph 4 in Figure 1.1 is merely an alternative payment method for the buyer, not a contrary-to-duty obligation. Therefore, the encoding in BCL above of the sales contract in Figure 1.1 is actually not faithful to realities, since we model Paragraph 4 is a contrary-to-duty obligation of Paragraph 3. However, Governatori and Milosevic make similar encodings for what are arguably not contrary-to-duty obligations [35].

Returning to the grammar of FCL, policies have the form:

$$t_1 \to t_2 \to \cdots \to t_n \to \otimes\text{-}exp,$$

where $t$ is used as an abbreviation for the union of the two syntactic categories $l$ and $ml$. Each $t_i$ is the antecedent for the rule ("trigger" in the BCL terminology), which dictates when the $\otimes$-expression ("behaviour" in the BCL terminology) becomes active.

Governatori and Milosevic sketch how to map a subset of BCL to FCL. However, such a mapping does not provide a semantics to BCL, since FCL itself does not have a semantics! Being supposedly based on standard deontic logic, we would expect a semantics for FCL in the style of Appendix A.1, yet such a semantics is not provided. Moreover, FCL neglects temporal aspects, which is a crucial part of BCL contracts.

In relation to contract analysis (R16), Governatori et al. [38] consider an interesting question, namely the analysis of whether a given business process is compliant with a business contract. A business process is characterised by the set of event patterns that it generates, and compliance is then a matter of testing whether all event patterns satisfy the given FCL contract. However, since there is no formal semantics for FCL, there is no result that contract compliance implies formal contract satisfaction in all possible executions.

### 1.3.7 Process Algebra

Andersen et al. [6] consider a restricted form of contracts that govern the exchange of resources—that is, money, goods, and services—between multiple parties. The approach complements McCarthy's resources, events, and agents (REA) accounting model [64], in which the transaction patterns of companies (agents) are modelled as transfers of resources, referred to as events. The approach of Andersen et al.

```
letrec{
 return[buyer,seller,goods,payment,deadline] =
  (transmit(b, s, g, t1 | b = buyer ∧ s = seller ∧ g = goods ∧ t1 ≤ deadline).
   transmit(s, b, p, t2 | s = seller ∧ b = buyer ∧ p = payment ∧ t2 ≤ t1 + 7).
   Success) + Success

 sale[buyer,seller,goods,payment,deadline] =
  transmit(s, b, g, t1 | s = seller ∧ b = buyer ∧ g = goods ∧ t1 ≤ deadline).
  transmit(b, s, p, t2 | b = buyer ∧ s = seller ∧ p = payment/2 ∧ t1 = t2).
  ((transmit(b, s, p, t3 | b = buyer ∧ s = seller ∧ p = payment/2 ∧ t3 ≤ t1 + 30).
    Success
    +
    transmit(b, s, p, t3 | b = buyer ∧ s = seller ∧ p = 1.1∗(payment/2) ∧ t3 ≤ t1 + 44).
    Success)
   ||
   return(buyer, seller, goods, payment, t1 + 14))
}
in
sale(Buyer, Seller, Printer, 200€, 2011−01−01)
```

Figure 1.11: A sales contract between a buyer and a seller (process algebra).

is inspired partly by the compositional approach of Peyton Jones and Eber (Section 1.3.4), partly by the algebraic, behavioural approach of CSP [44].

The grammar of Andersen et al.'s contract calculus is as follows:

$$\begin{array}{lll}
k ::= & \mathbf{letrec}\{f_i[\vec{x_i}] = c_i\}_{i=1}^{m}\ \mathbf{in}\ c & \text{(contract)} \\
c ::= & \mathbf{Success} & \text{(no obligations)} \\
 & \mid \mathbf{Failure} & \text{(failed contract)} \\
 & \mid c_1 + c_2 & \text{(choice)} \\
 & \mid c_1 \parallel c_2 & \text{(parallel)} \\
 & \mid c_1 ; c_2 & \text{(sequence)} \\
 & \mid f(\vec{a}) & \text{(instantiation)} \\
 & \mid \mathbf{transmit}(a_1, a_2, r, t \mid p).c & \text{(transfer obligation)}
\end{array}$$

$x$, $a$, $r$, and $t$ are variables, and $p$ is a predicate (we omit the grammar for predicates).

At top-level, a contract $k$ consists of a set of mutually recursive template definitions along with a contract body. The body of a contract $c$ is either (1) a completed contract **Success**, (2) a failed contract **Failure**, (3) a choice between two subcontracts $c_1 + c_2$, (4) a simultaneous obligation to fulfil two subcontracts $c_1 \parallel c_2$, (5) a sequential obligation to fulfil two subcontracts $c_1; c_2$, (6) an instantiation of a contract template $f(\vec{a})$, or (7) an obligation to transmit a resource that fulfils the predicate $p$, followed by a residual contract $\mathbf{transmit}(a_1, a_2, r, t \mid p).c$.

In order to illustrate the contract calculus, consider the encoding of the sales contract from Figure 1.1 in Figure 1.11. Note how the permission to return goods is modelled as a choice between either returning the goods, or not. Moreover, note that **transmit**s are binders, that is for instance $t_1$ is bound to the time of returning goods, and can be referred to in the continuation contract in order to encode the deadline for return payment.

Unlike previous work, Andersen et al. provide both a language for defining contracts (the above), as well as a semantic model for contracts (R1). The semantic model of contracts is trace-based, that is a contract is denoted by the set of traces that fulfil the contract. A trace is a finite sequence of events, and events have the form:

$$\text{transmit}(a_1, a_2, r, t),$$

denoting the transmission of resource $r$ from agent $a_1$ to agent $a_2$ at time $t$. The link between syntactic contracts and the semantic model is given as a CSP-inspired denotational semantics, that is a compositional mapping of contract syntax to sets of (accepting) traces.

Besides a formal semantics, the contract language has support for contract participants (R2), conditional commitments (R3), absolute temporal constraints (R4), relative temporal constraints (R5), reparation clauses (R6), potentially infinite and repetitive contracts (R8), time-varying, external dependencies (R9), history-sensitive commitments (R10), in-place expressions (R11), parametrised contracts (R12), run-time monitoring (R14), and amenability to (compositional) analysis (R16). Of these requirements, we want to single out run-time monitoring: Andersen et al. construct a sound and complete (with respect to the denotational trace semantics) small-step semantics. The small-step semantics is a labelled transition system $c \xrightarrow{e} c'$, which means that contract $c$ evolves to contract $c'$ under event $e$, or in other words that $c'$ is the residual contract of $c$ after event $e$ has happened. The small-step semantics gives rise to a run-time monitor, since the state of a contract is—at any time—determined by the events that have occurred, and upcoming deadlines can be determined by a syntactic inspection of the current contract state. The reduction semantics furthermore entails that any analysis applicable to initial contracts are also applicable at run-time.

One downside to Andersen et al.'s calculus is the inherent non-determinism of the choice operator $+$. For instance, it is possible to write a contract that obliges either agent $a_1$ to transfer a resource, or agent $a_2$ to transfer a resource, but what if neither of them transfer the resource? In such a contract it is impossible to assign blame (R15), which is also the case for the degenerate contract **Failure**.

### 1.3.8 Dynamic Logic

The contract language $\mathcal{CL}$ introduced by Prisacariu and Schneider [93] is a logic for expressing electronic contracts based on a combination of deontic, dynamic, and temporal logics. As in the logic of Governatori and Milosevic (Section 1.3.6), $\mathcal{CL}$ restricts deontic modalities to ought-to-do statements, that is deontic modalities can only be applied to actions that are performed. Unlike the logic of Governatori and Milosevic, Prisacariu and Schneider present a formal semantics in terms of an extended fragment of the propositional $\mu$-calculus. Similar to the logic approach of Lee (Section 1.3.1), Prisacariu and Schneider abandon the possible worlds semantics of standard deontic logic (Appendix A.1), in favour of a dynamic, action based semantics.

In later work, Fenech et al. [26] consider a revised version of $\mathcal{CL}$, along with a revised, trace semantics, which we base our presentation on. The grammar of $\mathcal{CL}$ is as follows:

1. $O(\textit{deliver})$
2. $[\textit{deliver}]\,\big(O(\textit{pay\_first\_half}) \wedge O_{O(\textit{pay\_fine})}(\textit{pay\_second\_half})\big)$
3. $[\textit{deliver}]P(\textit{return})$
4. $[\textit{return}]O(\textit{repay})$

Figure 1.12: A sales contract between a buyer and a seller (dynamic logic).

$$
\begin{array}{lll}
C & ::= C_O \mid C_P \mid C_F \mid C \wedge C \mid [\beta]C \mid \top \mid \bot & \text{(clause)} \\
C_O & ::= O_C(\alpha) \mid C_O \oplus C_O & \text{(obligation)} \\
C_P & ::= P(\alpha) \mid C_P \oplus C_P & \text{(permission)} \\
C_F & ::= F_C(\alpha) & \text{(prohibition)} \\
\alpha & ::= 0 \mid 1 \mid \overline{a} \mid a \mid \alpha \,\&\, \alpha \mid \alpha ; \alpha \mid \alpha + \alpha & \text{(deontic action)} \\
\beta & ::= \epsilon \mid 0 \mid 1 \mid \overline{a} \mid a \mid \beta \,\&\, \beta \mid \beta ; \beta \mid \beta + \beta \mid \beta^* & \text{(dynamic action)}
\end{array}
$$

A $\mathcal{CL}$ clause is either an obligation $C_O$, a permission $C_P$, a prohibition $C_F$, a conjunction of two clauses $C \wedge C$, a clause preceded by a dynamic condition $[\beta]C$, a trivially fulfilled clause $\top$, or a trivially violated clause $\bot$.

An obligation $O_C(\alpha)$ means that actions $\alpha$ are obligatory, and $C$ is a reparation clause that is active if $\alpha$ is not performed. Similar to Milosevic et al. (Section 1.3.6), the terminology "contrary-to-duty" is used for the reparation clause $C$, which suggests that there is an implicit agreement about primary and secondary obligations, compare the discussion in Section 1.3.6. However, like Milosevic et al., the examples that illustrate contrary-to-duty obligations in $\mathcal{CL}$ [26, 57] are arguably not contrary-to-duties, but merely choices.

A deontic action $\alpha$ is either no action 0, any action 1, any action—except $a$—$\overline{a}$, simultaneous actions $\alpha \,\&\, \alpha$, sequential actions $\alpha ; \alpha$, or any of two actions $\alpha + \alpha$. $C \oplus C$ is the exclusive disjunction of two clauses, that is exactly one of the two clauses must be fulfilled. Permissions $P(\alpha)$ and prohibitions $F_C(\alpha)$ are similar to obligations, only permissions do not have a contrary-to-duty clause as they cannot be violated, and prohibitions cannot be combined disjunctively. The dynamic condition $[\beta]C$ means that if $\beta$ happens, then $C$ must be fulfilled. Dynamic actions $\beta$ extend deontic actions $\alpha$ with Kleene star, which enables clauses of the form $[\beta^*]C$, that is whenever $\beta$ happens then $C$ should be fulfilled.

Actions are propositional atoms as in standard deontic logic (Appendix A.1). Figure 1.12 presents an encoding of the sales contract from Figure 1.1 in $\mathcal{CL}$, using the propositional atoms *deliver*, *pay_first_half*, *pay_second_half*, *pay_fine*, *return*, and *repay*. We omit the implicit conjunction between the clauses, we omit contrary-to-duty subscripts when they are $\bot$, and following previous $\mathcal{CL}$ examples, we model the late payment as a contrary-to-duty obligation, even though it is a choice.

Fenech et al. [26] present a trace semantics for $\mathcal{CL}$. Unlike traditional trace semantics, the fulfilment relation has the form $\sigma, \sigma_d \models C$, which means that the trace $\sigma$ fulfils $C$, provided that the remaining obligations described by $\sigma_d$ are met. Hence, standard trace fulfilment is a special case, that is $\sigma$ fulfils $C$ whenever $\sigma, \varepsilon \models C$ holds, where $\varepsilon$ is the empty trace. The trace semantics gives rise to incremental run-time monitoring by residuation in the style of Andersen et al. (Section 1.3.7). That is, Fenech et al. construct a residuation function $f : \mathcal{CL} \times A \to \mathcal{CL}$ that reduces a clause $C$ to a residual clause $C'$ when action $a$ takes place.

Besides a formal semantics (R1), $\mathcal{CL}$ supports conditional commitments (R3), relative temporal constraints (R5), reparation clauses (R6), potentially infinite and repetitive contracts (R8), run-time monitoring (R14), and (compositional) analysis (R16). Examples of contract analysis include meta results about the logic [93] in particular absence of certain deontic paradoxes, and model checking of contracts [83].

### 1.3.9 Defeasible Reasoning

Starting from defeasible logic [78] and formal contract logic (FCL, Section 1.3.6), Governatori introduces defeasible deontic logic of violation (DDLV) [34] (with a later refinement by Governatori and Pham [36]). The purpose of DDLV is to support overlapping contract clauses, where conflicts are resolved by allowing rules to defeat each other.

Following defeasible logic, DDLV contains four different kinds of knowledge: facts, strict rules, defeasible rules, and a superiority relation. Facts represent indisputable knowledge, that is knowledge which cannot be overruled at a later stage. For instance, in the sales contract in Figure 1.1 it is a fact that the printer costs €200, which in DDLV is modelled as a predicate:

$$\mathit{Price}(\mathit{Printer}, 200).$$

Strict rules enable conclusion of new facts. When the antecedents of a strict rule are known to be facts, then so is the conclusion. As an example, the sales contract could have included the following definition of premium customers:

$$\mathit{TotalExpensesAtLeast}(X, 2000) \to \mathit{PremiumCustomer}(X),$$

which states that customers who have spent at least €2000 are premium customers.

Defeasible rules enable conclusion of new knowledge, which may be overruled by some other (defeasible or strict) rule. For instance the sales contract could have entitled premium customers to a 10% discount, which in DDLV would be modelled as follows:

$$\begin{aligned} &r_1 : \top \Rightarrow \mathit{Price}(\mathit{Printer}, 200) \\ &r_2 : \mathit{PremiumCustomer}(X) \Rightarrow \mathit{Price}(\mathit{Printer}, 180). \end{aligned}$$

The last component—the superiority relation—defines how rules defeat each other. For instance the two rules above would be related as $r_1 < r_2$, that is $r_2$ defeats $r_1$. In order to derive indisputable and defeasible knowledge, DDLV includes a set of inference rules, which besides deriving knowledge can detect conflicts. For instance, a *prima facie* conflict such as forgetting to supply the relation $r_1 < r_2$ will be detected, which is useful in the process of writing a conflict-free contract.

Besides defeasible rules, DDLV includes deontic modalities similar to its predecessor FCL (Section 1.3.6), which we will not replicate here. Unlike FCL however, DDLV supports contract normalisation, that is a transformation of any DDLV contract into a unique canonical form. The essence of the transformation is to rewrite implicit reparation clauses into explicit contrary-to-duty formulae (using the $\otimes$ connective of FCL), as well as to remove redundant clauses. As an example, the two rules:

$$\begin{aligned} &r_1 : \mathrm{O}_{\mathrm{Seller}}(\mathit{deliver}) \otimes \mathrm{O}_{\mathrm{Seller}}(\mathit{pay\_penalty}) \\ &r_2 : \neg \mathit{deliver}, \neg \mathit{pay\_penalty} \Rightarrow \mathrm{O}_{\mathrm{Seller}}(\mathit{pay\_large\_penalty}), \end{aligned}$$

are combined into a single rule:

$$r_{1+2} : \mathrm{O}_{\mathrm{Seller}}(\mathit{deliver}) \otimes \mathrm{O}_{\mathrm{Seller}}(\mathit{pay\_penalty}) \otimes \mathrm{O}_{\mathrm{Seller}}(\mathit{pay\_large\_penalty}).$$

A normal form is achieved by applying the two procedures of merging and removing redundancies until a fixed point is reached. (Such a fixed point always exists and it is unique [36].) Normal forms are beneficial for several reasons. First, contract equivalence is decidable, that is two contracts are equivalent if and only if their normal forms are identical. Second, contract analysis and run-time monitoring only need be defined on normal forms, which may ease construction of such algorithms.

In terms of our desiderata from Section 1.2, the extension of FCL to DDLV adds the analyses of conflict resolution and computation of normal forms (R16).

## 1.4 Contract Lifecycle Management

In this section we briefly list a set of commercial CLM systems, as well as the features that these systems (claim to) support. The list that we present here is an extension of the aspects identified by Tan et al. [106], but unlike Tan et al. we will not go into detail with the various aspects.

Our survey includes 14 software products, which we label for easy reference:

| | |
|---|---|
| Blueridge Software: *Contract Assistant*, http://www.blueridgesoftware.bz | (CA) |
| CobbleStone Systems: *ContractInsight*, http://www.cobblestonesystems.com | (CI) |
| Moai: *CompleteSource Contract Management*, http://www.moai.com | (CS) |
| Eceteon: *Contraxx*, http://www.ecteon.com | (CX) |
| Emptoris: *Contract Management Solutions*, http://www.emptoris.com | (EM) |
| Great Minds Software: *Contract Advantage*, http://www.greatminds-software.com | (GM) |
| IntelliSoft Group: *IntelliContract*, http://www.intellisoftgroup.com | (IC) |
| Ketera: *Contract Management*, http://www.ketera.com | (KE) |
| Open Text: *Contract Management*, http://www.opentext.com | (OT) |
| 8over8: *ProCon Contract Management*, http://www.8over8.com | (PC) |
| SAP: *SAP CLM*, http://www.sap.com | (SA) |
| StatsLog Software Corporation: *StatsLog*, http://www.statslog.com | (SL) |
| Procuri: *TotalContracts*, http://www.procuri.com | (TC) |
| Upside Software: *UpsideContract*, http://www.upsidesoft.com | (UC) |

Based on the descriptions found at the websites of the commercial products, we have gathered a list of CLM features below. We have not included technology related features, such as the database on which a system runs, or the particular technology used for the user interface—the list intentionally only includes CLM features. The CLM features are (in no particular order):

1. Centralised contract repository for storing pending, running, and finished contracts.

| | CA | CI | CS | CX | EM | GM | IC | KE | OT | PC | SA | SL | TC | UC |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | ✓ | | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ | |
| 2 | ✓ | | | | | | | ✓ | ✓ | ✓ | | | | |
| 3 | ✓ | ✓ | | | | ✓ | ✓ | ✓ | | ✓ | ✓ | | ✓ | ✓ |
| 4 | ✓ | | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | | ✓ | |
| 5 | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 6 | | ✓ | | | ✓ | ✓ | ✓ | | | ✓ | ✓ | ✓ | ✓ | ✓ |
| 7 | | ✓ | | | | | | | | | | | | ✓ |
| 8 | | ✓ | | | | ✓ | ✓ | | ✓ | ✓ | ✓ | | ✓ | ✓ |
| 9 | | | ✓ | ✓ | ✓ | | | ✓ | ✓ | | ✓ | | | ✓ |
| 10 | | | ✓ | ✓ | | | | | | | | | | |
| 11 | | | | | | | | | ✓ | | | | | ✓ |
| 12 | | ✓ | | | | | ✓ | | | | ✓ | | ✓ | ✓ |
| 13 | | | | | | | ✓ | ✓ | | ✓ | ✓ | | ✓ | |
| 14 | | | | | | | | | ✓ | | ✓ | | ✓ | |
| 15 | | | | | | | | | | ✓ | | | | |

Figure 1.13: Features comparison matrix (CLM products horizontally, features vertically).

2. Possibility of restricting access to contracts/data in the CLM system (access control).
3. E-mail notifications/alarms/alerts and run-time monitoring of contracts.
4. Reporting and analytics.
5. Search-capabilities.
6. Template-based contract creation.
7. Compositional construction of contracts from other subcontracts.
8. Workflows for contract approval/review.
9. Contract compliance and adherence to business standards.
10. Contract negotiation.
11. Task-list with outstanding obligations/rights.
12. Versioning system.
13. Full auditing trail.
14. Integration with enterprise resource planning (ERP) system(s).
15. Secure messaging in contract execution/collaboration.

Compared to the survey of Tan et al. [106], the features above represent a more fine-grained list of requirements—in addition, the list also contains features that are not covered by Tan et al. (2, 8, 13, and 14). The matrix in Figure 1.13 summarises which features are (supposedly) supported by the surveyed CLM products. Most interestingly, we have found no evidence that any of the CLM products use domain-specific languages for contracts—we conjecture that they instead implement contract templates ad hoc in a general purpose programming language.

## 1.5 Conclusion

Formal languages and models for contracts is a research topic that has drawn interest from different areas of computer science. Formalisation and automatic run-time monitoring of contracts is an interesting challenge, both from a business viewpoint as well as from a theoretical viewpoint. From a theoretical viewpoint, the challenge manifests itself in the wide variety of aspects found in contracts, some of which we have identified in this survey in the form of requirements:

*(R1) Contract model, contract language, and a formal semantics.*
*(R2) Contract participants.*
*(R3) (Conditional) commitments.*
*(R4) Absolute temporal constraints.*
*(R5) Relative temporal constraints.*
*(R6) Reparation clauses.*
*(R7) Instantaneous and continuous actions.*
*(R8) Potentially infinite and repetitive contracts.*
*(R9) Time-varying, external dependencies (observables).*
*(R10) History-sensitive commitments.*
*(R11) In-place expressions.*
*(R12) Parametrised contracts.*
*(R13) Isomorphic encoding.*
*(R14) Run-time monitoring.*
*(R15) Blame assignment.*
*(R16) Amenability to (compositional) analysis.*

In light of these requirements, existing well-established formalisms such as deontic logics, temporal logics, timed automata, and process calculi are inadequate for modelling all details of contracts. Consequently, several new models and languages for specifying contracts have been proposed. However, common to almost all existing approaches is the lack of detail and lack of formal semantics. With the exception of Andersen et al. (Section 1.3.7) and Prisacariu et al. (Section 1.3.8), existing approaches neglect formal mathematical underpinnings, or—at best—provide incomplete mathematical models and semantics. The languages of Andersen et al. and Prisacariu et al., on the other hand, lack important features in order to qualify as a *silver bullet*—most notably empirical evidence that the languages adequately capture contracts of the intended domains. We summarise our comparative analysis of Section 1.3 in Figure 1.14.

Besides the models and languages we have covered in this survey, more informal approaches exist [21, 80, 117, 122]. The contract expression language [21] is an XML based representation of contracts; Oren et al. [80] consider a contract model in which contracts are sets of norms, similar to the model of Goodchild et al. (Section 1.3.2); Weigand and Xu [117] consider a contract language based on dynamic logic à la Prisacariu et al. (Section 1.3.8); and Xu [122] investigates a graph theoretic representation of contracts. Common to all approaches is a too informal, or even inconsistent, presentation in order to include them in our comparative analysis.

| | Lee | Goo | Bou | Pey | Mol | Mil | And | Pri |
|---|---|---|---|---|---|---|---|---|
| R1 | | | | | | | ✓ | ✓ |
| R2 | ✓ | | ✓ | | | ✓ | ✓ | |
| R3 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| R4 | ✓ | ✓ | ✓ | ✓ | (✓) | ✓ | ✓ | |
| R5 | ✓ | | | ✓ | ✓ | ✓ | ✓ | ✓ |
| R6 | ✓ | | ✓ | | ✓ | ✓ | ✓ | ✓ |
| R7 | ✓ | | | | | | | |
| R8 | ✓ | | | ✓ | ✓ | | ✓ | ✓ |
| R9 | | | | ✓ | | | ✓ | |
| R10 | | | | | | | ✓ | |
| R11 | | ✓ | | ✓ | | | ✓ | |
| R12 | ✓ | | | ✓ | | | ✓ | |
| R13 | ✓ | | ✓ | (✓) | | ✓ | | |
| R14 | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ |
| R15 | (✓) | | (✓) | | | | | |
| R16 | ✓ | | | ✓ | ✓ | ✓ | ✓ | ✓ |

Figure 1.14: Requirements comparison matrix (contract formalisms horizontally, requirements vertically). *Lee* is the formalism of Section 1.3.1, *Goo* is the formalism of Section 1.3.2, *Bou* is the formalism of Section 1.3.3, *Pey* is the formalism of Section 1.3.4, *Mol* is the formalism of Section 1.3.5, *Mil* is the formalism of Section 1.3.6, *And* is the formalism of Section 1.3.7, and *Pri* is the formalism of Section 1.3.8.

As final remark, note that we have not given a formal comparison of the expressivity of each contract formalism. That is, we have not compared whether one formalism can be expressed in another, or whether two formalisms are mutually exclusive. We omit such an analysis both for simplicity, but also because it would require a formal semantics of each contract formalism.

# Chapter 2

# A Trace-Based Model for Multiparty Contracts[⋆]

**Abstract**

In this article we present a model for multiparty contracts in which contract conformance is defined abstractly as a property on traces. A key feature of our model is blame assignment, which means that for a given contract, every breach is attributed to a set of parties. We show that blame assignment is compositional by defining contract conjunction and contract disjunction. Moreover, to specify real-world contracts, we introduce the contract specification language CSL with an operational semantics. We show that each CSL contract has a counterpart in our trace-based model and from the operational semantics we derive a run-time monitor. CSL overcomes limitations of previously proposed formalisms for specifying contracts by supporting: (history sensitive and conditional) commitments, parametrised contract templates, relative and absolute temporal constraints, potentially infinite contracts, and in-place arithmetic expressions. Finally, we illustrate the general applicability of CSL by formalising in CSL various contracts from different domains.

## 2.1 Introduction

Contracts are legally binding agreements between parties and in e-business it is particularly crucial to automatically check conformance to them, for example for minimising financial penalties. The Aberdeen Group [84, 85] has recently identified *contract lifecycle management* (CLM) as a key methodology in e-business: CLM is a broad term used to cover the activities of systematically and efficiently managing contract creation, contract negotiation, contract approval, contract analysis, and contract execution. Monitoring the execution of contracts constitutes the primary incentive for enterprises to use CLM, since it enables qualified decision making and makes it possible to issue reminders for upcoming deadlines, which may lead to a significant decrease of financial loss due to noncompliance:

> “[...] the average savings of transactions that are compliant with contracts is 22%” [84, page 1].

[⋆] *Joint work with Felix Klaedtke and Eugen Zălinescu [52].*

Consequently, several systems that implement the CLM methodology have been deployed.[1] More traditional *enterprise resource planning* (ERP) systems such as Microsoft Dynamics NAV[2] and Microsoft Dynamics AX[3] are also used for managing business agreements. However, a shortcoming of existing CLM and ERP systems is that contracts are dealt with in an ad hoc manner rather than as first-class objects. In fact, the before mentioned studies by the Aberdeen Group [84, 85] suggest the use of a domain-specific language as the basis for automated CLM.

Although various authors have proposed domain-specific languages for representing contracts [6, 17, 33, 35, 58, 86, 93], constructing a widely applicable contract specification language remains a challenge [82]. One reason is that contracts involve many different aspects like absolute temporal constraints (as in deadlines), relative temporal constraints (for imposing an ordering on the occurrence of certain actions), reparation clauses, conditional commitments, different deontic modalities [111] (such as obligations and permissions), and repetitive patterns. In order to make some of these aspects concrete, consider the contract in Figure 2.1, which we will use as a running example in the remainder of this article. This sample contract involves both obligations (Paragraph 1), permissions (Paragraph 5), absolute deadlines (Paragraph 1), relative deadlines (Paragraph 3), and reparation activities (Paragraph 4). Additionally, it involves data dependencies between paragraphs, for example the payment amount in Paragraph 4 depends on the amount defined in Paragraph 3.

Besides being able to capture the various aspects found in contracts mentioned above, a contract specification language should also be amenable to automatic analysis. In particular, the language should support *run-time monitoring* [59] of contracts, that is reporting of (potential) contract breaches during execution—for instance as the result of passing a deadline or performing a forbidden action. Furthermore, in case of noncompliance the run-time monitor should be able to assign *blame* to one or more of the parties involved in the contract, rather than simply reporting noncompliance without specifying who is responsible for the breach of contract. Surprisingly, even though run-time monitoring of contracts has been studied extensively [6, 33, 35, 74, 93, 121], blame assignment has not been given much attention yet. To the best of our knowledge only Xu [121] investigates blame assignment though not from the viewpoint of run-time monitoring, but rather from an off-line

[1]Examples of such systems include (all URLs retrieved on May 18th 2011):

- Blueridge Software *Contract Assistant*, http://www.blueridgesoftware.bz.
- CobbleStone Systems *ContractInsight*, http://www.cobblestonesystems.com.
- Moai *CompleteSource Contract Management*, http://www.moai.com.
- Ecteon *Contraxx*, http://www.ecteon.com.
- Emptoris *Contract Management Solutions*, http://www.emptoris.com.
- Great Minds Software *Contract Advantage*, http://www.greatminds-software.com.
- IntelliSoft Group *IntelliContract*, http://www.intellisoftgroup.com.
- Ketera *Contract Management*, http://www.ketera.com.
- Open Text *Contract Management*, http://www.opentext.com.
- 8over8 *ProCon Contract Management*, http://www.8over8.com.
- SAP *SAP CLM*, http://www.sap.com.
- Procuri *TotalContracts*, http://www.procuri.com.
- Upside Software *UpsideContract*, http://www.upsidesoft.com.

[2]http://www.microsoft.com/en-us/dynamics/products/nav-overview.aspx.
[3]http://www.microsoft.com/en-us/dynamics/products/ax-overview.aspx.

**Paragraph 1.** Seller agrees to transfer and deliver to Buyer, on or before 2011-01-01, the goods: 1 laser printer.

**Paragraph 2.** Buyer agrees to accept the goods and to pay a total of €200 for them according to the terms further set out below.

**Paragraph 3.** Buyer agrees to pay for the goods half upon receipt, with the remainder due within 30 days of delivery.

**Paragraph 4.** If Buyer fails to pay the second half within 30 days, an additional fine of 10% has to be paid within 14 days.

**Paragraph 5.** Upon receipt, Buyer has 14 days to return the goods to Seller in original, unopened packaging. Within 7 days thereafter, Seller has to repay the total amount to Buyer.

Figure 2.1: A sales contract between a buyer and a seller.

viewpoint where blame has to be determined from a set of unfulfilled, dependent commitments.

In this article, we present a contract specification language that targets at naturally formalising and monitoring contracts. In particular, contracts formalised in our language can directly be monitored, and in case of noncompliance the monitor assigns blame to the responsible contract parties. Although our focus is on business contracts, our language is not essentially restricted to this particular application area.

### 2.1.1 Breach of Contract and Blame Assignment

A first question that arises when designing such a contract specification language is what constitutes a breach of contract? Returning to the example contract in Figure 2.1, one can think of several scenarios that arguably constitute breaches of contract:

(1) Seller fails to deliver to Buyer on time.

(2) Seller delivers on time, Buyer pays first half on delivery, but Buyer does not pay second half on time.

(3) Seller delivers on time, Buyer pays first half on delivery, Buyer does not pay second half on time, and Buyer does not pay the additional fine on time.

Clearly, the first scenario represents a breach of contract, and Seller is to be blamed for not delivering the goods to Buyer. In the second scenario, it is less clear, since Buyer has violated Paragraph 3, but depending on whether the extended deadline has passed, Buyer may or may not have breached the contract. Finally, in the last scenario it is clear that Buyer has breached the contract, but it is perhaps less clear whether violating Paragraph 3 or Paragraph 4 (or both) constitutes the breach of contract.

The approach we take is that of *fundamental breaches*: a breach of contract takes place only when a violation happens, from which the contract cannot recover, and from which it therefore does not make sense to continue executing the contract. In terms of run-time monitoring, a breach of contract hence takes place only when it is impossible to complete a conforming execution. With this rather informal definition of contract breach, we see that the first scenario constitutes indeed a breach of contract. Regarding the second scenario, it depends whether Buyer will pay the fine or not, as only neglecting to pay the fine constitutes a breach of contract. Thus scenario (2) does not yet represent a breach, in contrast to the last scenario (3).

We deliberately use the term *breach* rather than *violation* in order to distinguish our concept of (fundamental) breach from the more traditional notion of violation known from standard deontic logic (SDL) with contrary-to-duty obligations [92]. In the context of SDL, it is tempting to encode reparation clauses like the one in Paragraph 4 in the form of a contrary-to-duty obligation. Yet, with such an encoding there is an implicit agreement that the *primary* obligation (Paragraph 3) should be complied with first and foremost, and only complying with the reparation obligation constitutes a violation, even though—from a contractual point of view—the contract is fulfilled.

A classical example that illustrates the subtle, but important, difference is the “gentle murderer”: do not kill, but if you kill, kill gently [28]. The gentle murderer is an actual contrary-to-duty obligation, because there is an implicit agreement that you should not kill—only if you have no other options than killing, then at least you should do so gently.

We argue, however, that contracts should not contain implicit agreements, in particular because parties may have conflicting interests. Hence if one party wishes to impose that an obligation be primary, then the only way to do so is by making sure that there is an incentive for the responsible (counter) party to perform the primary obligation, for example by imposing a penalty for complying only with the reparation obligation. Hence the gentle murderer, as a contract, would be: do not kill, but if you kill, kill gently and go to jail. Attaching penalties to violations yields new obligations. Violating such an obligation might result in new obligations until either all obligations are fulfilled or eventually a breach of contract is reached. For the example, killing non-gently represents a breach of contract. Killing gently and not going to jail also represents a breach of contract. However, killing gently and going to jail is not a breach of contract. Note that the consequences of breaching the contract are not specified.

Ideally, blame assignment should be *deterministic*, that is it should uniquely determine the parties responsible for a breach. However, not all contracts allow for deterministic blame assignment, as illustrated by the following scenario: If one paragraph specifies that Alice has to fulfil an obligation by time $\tau$, and another paragraph that Bob has to fulfil another obligation by the same time $\tau$, and the contract only asks for conformance with one of the paragraphs, then we are in a delicate situation—who is to blame if neither Alice nor Bob has fulfilled her/his obligation?[4] Contracts involving disjunction, such as this one, lead to nondeterministic blame assignment. In other words, such contracts are ambiguous. For simplicity, we choose not to model them, except in the special cases when the same parties are blamed in both subcontracts. Our choice is also motivated by the fact that such scenarios rarely correspond to real-world contracts.

[4]We leave it to the reader to ponder whether blaming neither of the two, or blaming both of them is acceptable. Our view is that neither option is acceptable.

### 2.1.2 Contributions and Organisation

We see our main contributions as follows. First, we present an abstract, trace-based model for contracts that has blame assignment at its core. Furthermore, our model supports modular composition of contracts by contract conjunction and disjunction. Second, we introduce the contract specification language (CSL) that fits naturally—by means of a mapping—to our abstract model, and that overcomes many of the limitations of previous specification languages for contracts. Third, we describe a run-time monitoring algorithm for CSL specifications obtained as a by-product of the reduction semantics of CSL.

The remainder of this article is structured as follows. In Section 2.2 we present our abstract, trace-based model for contracts, relying on the informal notion of contract breach and blame assignment described above. We show how our model encodes various high-level aspects, such as obligations, permissions, and reparation clauses without relying on such notions. We also provide operators for composing contracts and show that they fulfil desirable algebraic properties. In Section 2.3 we introduce the contract specification language CSL, together with a formal semantics that maps CSL into our abstract, trace-based contract model. Furthermore, from the small-step, reduction-based semantics of CSL, we derive a run-time monitoring algorithm. We also demonstrate the applicability of CSL by means of several example contracts. We discuss related work in Section 2.4 and we draw conclusions in Section 2.5. Appendix B.1 contains additional proof details.

## 2.2 Trace-Based Contract Model

Trace-based contract models have been proposed before [6, 57], but unlike our model, those models partition traces into conforming and nonconforming traces, without taking blame assignment into account. A trace is a sequence of actions that represent the *complete* history of actions that have occurred during the execution of a contract. In order to capture real-time aspects, and not only relative temporality, actions of a trace are timestamped. In this article we ignore how actions are generated, and neither do we model how parties agree that actions have taken place—the latter would usually involve a hand-shaking protocol, which is outside the scope of our work. For the purpose of defining contracts, we hence assume a trace of timestamped actions is given.

### 2.2.1 Notation and Terminology

Before presenting our contract model, we fix the notation and terminology that we use in the remainder of the text. Throughout this article, $\mathsf{P}$ denotes the set of *parties*, $\mathsf{A}$ the set of *actions*, and $\mathsf{Ts}$ the set of *timestamps*. The sets $\mathsf{P}$ and $\mathsf{A}$ can be finite or infinite but we require that they are both non-empty. We require that $\mathsf{Ts}$ is totally ordered by the relation $\leq$, and that $\mathsf{Ts}$ has a least element and that no element in the set is an upper bound, that is for all $\tau \in \mathsf{Ts}$ there is some $\tau' \in \mathsf{Ts}$ such that $\tau \neq \tau'$ and $\tau \leq \tau'$. In the following, for representation issues, we assume that $\mathsf{Ts} = \mathbb{N}$.

We write a finite sequence $\sigma$ over an alphabet $\Sigma$ as $\langle \sigma[0], \sigma[1], \ldots, \sigma[n-1] \rangle$, where $\sigma[i] \in \Sigma$ denotes the $(i+1)$st letter of $\sigma$. Its length is $n$ and denoted by

$|\sigma|$. In particular, $\langle\rangle$ denotes the empty sequence which has length 0. Analogously, an infinite sequence $\sigma$ over $\Sigma$ is written as $\langle\sigma[0], \sigma[1], \sigma[2], \dots\rangle$ with $\sigma[i] \in \Sigma$, for every $i \in \mathbb{N}$. The length of an infinite sequence $\sigma$ is $|\sigma| = \infty$. We write $\sigma \sqsubset \sigma'$ if the sequence $\sigma$ is a finite prefix of the sequence $\sigma'$, that is if $\sigma$ is finite and there is a sequence $\sigma''$ such that $\sigma' = \sigma\sigma''$, where $\sigma\sigma''$ denotes the concatenation of the sequences $\sigma$ and $\sigma''$.

An *event* is a tuple $(\tau, \alpha)$, where $\tau \in \mathsf{Ts}$ is a timestamp and $\alpha \in \mathsf{A}$ is an action. We write $\mathrm{ts}(\epsilon)$ for the timestamp of an event $\epsilon = (\tau, \alpha)$, that is $\mathrm{ts}(\epsilon) = \tau$. A *trace* $\sigma$ is a finite or infinite sequence of events where the sequence of timestamps are:

(1) increasing, that is $\mathrm{ts}(\sigma[i]) \leq \mathrm{ts}(\sigma[j])$ for all $i, j \in \mathbb{N}$ with $i \leq j < |\sigma|$, and

(2) progressing for infinite traces, that is for all $\tau \in \mathsf{Ts}$ there is some $i \in \mathbb{N}$ such that $\mathrm{ts}(\sigma[i]) \geq \tau$ whenever $|\sigma| = \infty$.

We denote the set of all traces by $\mathsf{Tr}$, and the subset of finite traces by $\mathsf{Tr}_{\mathrm{fin}}$, that is $\mathsf{Tr}_{\mathrm{fin}} = \{\sigma \in \mathsf{Tr} \mid |\sigma| \neq \infty\}$. $\mathsf{Tr}^{\tau}$ denotes the subset of traces where all timestamps are at least $\tau$, and similarly for $\mathsf{Tr}^{\tau}_{\mathrm{fin}}$. For a finite non-empty trace $\sigma$, the timestamp of the last event in $\sigma$ is denoted by $\mathrm{end}(\sigma)$, and for the empty trace, we define $\mathrm{end}(\langle\rangle) = 0$.

For a trace $\sigma \in \mathsf{Tr}$ and a timestamp $\tau \in \mathsf{Ts}$, $\sigma_\tau$ denotes the longest prefix of $\sigma$ with $\mathrm{end}(\sigma_\tau) \leq \tau$. This prefix exists, since the properties (1) and (2) ensure that there are only finitely many prefixes $\sigma' \sqsubset \sigma$ with $\mathrm{end}(\sigma') \leq \tau$.

Finally, we denote the domain of a (partial) function $f$ by $\mathrm{dom}(f)$, that is $\mathrm{dom}(f)$ is the set of elements $a$ for which $f(a)$ is defined. For a function $f$ and a set $X \subseteq \mathrm{dom}(f)$, $f|_X$ denotes the restriction of $f$ to $X$.

### 2.2.2 Contracts

We capture blame assignment by generalising the outcome of a contract execution from a binary result (conformance or nonconformance) to *verdicts*, defined as elements of the set:

$$\mathsf{V} = \{\checkmark\} \cup \{(\tau, B) \mid \tau \in \mathsf{Ts} \text{ and } B \text{ is a non-empty finite subset of } \mathsf{P}\},$$

where $\checkmark$ represents *contract conformance*, that is no one is to be blamed, and $(\tau, B)$ represents a *breach of contract* at time $\tau$ by the parties in $B$. Whenever $|B| > 1$ then multiple parties have breached the contract simultaneously. For instance, both parties of a barter deal may breach the contract if neither hands over the agreed goods.

A contract is defined as a function that maps traces to verdicts:

**Definition 2.2.1.** Let $P$ be a non-empty and finite subset of $\mathsf{P}$. A *contract* between parties $P$, starting at time $\tau_0 \in \mathsf{Ts}$, is a function $\mathsf{c} : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ that satisfies the following conditions for all $\sigma \in \mathsf{Tr}^{\tau_0}$ and $(\tau, B) \in \mathsf{V}$:

$$\text{if } \mathsf{c}(\sigma) = (\tau, B) \text{ then } B \subseteq P \text{ and } \tau \geq \tau_0, \tag{2.1}$$

and

$$\text{if } \mathsf{c}(\sigma) = (\tau, B) \text{ then } \mathsf{c}(\sigma') = (\tau, B), \text{ for all } \sigma' \in \mathsf{Tr}^{\tau_0} \text{ with } \sigma_\tau = \sigma'_\tau. \tag{2.2}$$

The contract for which all traces are conforming is denoted $\mathsf{c}_{\checkmark}$, that is $\mathsf{c}_{\checkmark}$ is the function with $\mathsf{c}_{\checkmark}(\sigma) = \checkmark$, for all $\sigma \in \mathsf{Tr}^{\tau_0}$.

The definition entails that contracts are deterministic, as $\mathsf{c}$ is a function. Since traces are considered complete, condition (2.2) guarantees that a breach at time $\tau$ only depends on what has (and has not) happened up until time $\tau$. Moreover, the verdict of a contract can only depend on what has happened after the contract started.

**Example 2.2.2.** We illustrate our contract model by representing Paragraph 1 in Figure 2.1 as a contract $\mathsf{c}_1 : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$, for a suitable $\tau_0$. As the paragraph only defines an obligation on the party Seller, we define $\mathsf{c}_1$ as a contract "between" {Seller} with:

$$\mathsf{c}_1(\sigma) = \begin{cases} \checkmark & \text{if } \sigma[i] = (\tau, \text{delivery}), \text{ for some } i \in \mathbb{N} \text{ and } \tau \in \mathsf{Ts} \\ & \quad \text{with } i < |\sigma| \text{ and } \tau \leq \tau_d, \\ \big(\tau_d, \{\text{Seller}\}\big) & \text{otherwise.} \end{cases}$$

The action delivery represents the delivery of goods to the party Buyer and $\tau_d$ represents the deadline 2011-01-01. Note that dates like 2011-01-01 can be easily interpreted as non-negative integers by taking for instance the corresponding UNIX time. It is easy to check that $\mathsf{c}_1$ satisfies the properties of Definition 2.2.1.

### 2.2.3 Contract Conformance on Infinite Traces

The definition of contracts implicitly includes the crucial requirement that all breaches of contract are associated with a point in time. From this restriction it follows that contract conformance is not a *liveness property* [4], such as: Buyer must deliver the printer to Seller eventually. We see this as a natural restriction, since one of the purposes of formalising contracts is to run-time monitor their execution, and hence breaches of contract should be detected in finite time. In other words, every obligation must have a deadline.

The following lemma follows directly from the definition of contracts, because $\sigma_\tau$ is the longest prefix up to time $\tau$ of the trace $\sigma$.

**Lemma 2.2.3.** *Let* $\mathsf{c} : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ *be a contract and let* $\sigma$ *be a (finite or infinite) trace. Then* $\mathsf{c}(\sigma) = (\tau, B)$ *if and only if* $\mathsf{c}(\sigma_\tau) = (\tau, B)$*.*

The previous lemma entails that any nonconforming trace (in particular, any nonconforming infinite trace) has a nonconforming prefix. However, not all extensions of this prefix need be nonconforming too. Indeed, a nonconforming finite trace may be extended to a conforming trace (for instance, simply by performing an unfulfilled obligation), even if the time of the breach coincides with the timestamp of the last event: a contract $\mathsf{c}$ may satisfy, for example, $\mathsf{c}\big(\langle(\tau, \alpha)\rangle\big) = (\tau, B)$ and $\mathsf{c}\big(\langle(\tau, \alpha), (\tau, \alpha')\rangle\big) = \checkmark$, for some $\alpha, \alpha' \in \mathsf{A}$, $\tau \in \mathsf{Ts}$, and parties $B \subseteq \mathsf{P}$. Still, any extension of a nonconforming finite trace *after* the time of the breach is also nonconforming.

**Proposition 2.2.4.** *The set of infinite traces conforming with a contract is a safety property.*

*Proof.* Let $\mathsf{c} : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ be a contract and let

$$C = \{\sigma \in \mathsf{Tr}^{\tau_0} \mid \sigma \text{ is infinite and } \mathsf{c}(\sigma) = \checkmark\}.$$

We need to show that for any infinite trace $\sigma \notin C$, there is a prefix $\sigma'$ of $\sigma$ such that for any infinite trace $\sigma''$ with $\sigma' \sqsubset \sigma''$, it holds that $\sigma'' \notin C$.

Let $\sigma \notin C$ be an infinite trace. Then $\mathsf{c}(\sigma) = (\tau, B)$ for some $\tau$ and $B$. Let $\sigma'$ be an arbitrary prefix of $\sigma$ with $\mathrm{end}(\sigma') > \tau$, and consider an infinite trace $\sigma''$ with $\sigma' \sqsubset \sigma''$. Then, since $\mathrm{end}(\sigma') > \tau$, it follows that $\sigma''_\tau = \sigma_\tau$, and consequently condition (2.2) yields that $\mathsf{c}(\sigma'') = (\tau, B)$, hence $\sigma'' \notin C$, as required. □

The following lemma shows that "contracts" defined only on finite traces extend uniquely to contracts. In other words, contracts are uniquely determined by their verdicts on finite traces.

**Lemma 2.2.5.** *Let $P$ be a set of parties and $\mathsf{c} : \mathsf{Tr}^{\tau_0}_{\mathrm{fin}} \to \mathsf{V}$ be a function such that if $\mathsf{c}(\sigma) = (\tau, B)$ then $B \subseteq P$, $\tau \geq \tau_0$, and $\mathsf{c}(\sigma') = (\tau, B)$, for all $\sigma' \in \mathsf{Tr}^{\tau_0}_{\mathrm{fin}}$ with $\sigma_\tau = \sigma'_\tau$. Then there exists a unique extension $\mathsf{c}' : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ of $\mathsf{c}$, that is $\mathsf{c} = \mathsf{c}'|_{\mathsf{Tr}^{\tau_0}_{\mathrm{fin}}}$, such that $\mathsf{c}'$ is a contract.*

*Proof.* Let $\mathsf{c}' : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ be the function that extends $\mathsf{c}$ to infinite traces by:

$$\mathsf{c}'(\sigma) = \begin{cases} \checkmark & \text{if whenever } \mathsf{c}(\sigma') = (\tau, B) \text{ and } \sigma' \sqsubset \sigma \text{ then } \mathrm{end}(\sigma') \leq \tau, \\ \mathsf{c}(\sigma') & \text{otherwise, where } \sigma' \text{ is a prefix of } \sigma \text{ such that} \\ & \qquad\qquad \mathsf{c}(\sigma') = (\tau', B') \text{ and } \mathrm{end}(\sigma') > \tau', \end{cases}$$

for any infinite trace $\sigma$. We first show that $\mathsf{c}'$ is a contract between parties $P$ starting at time $\tau_0$.

First note that $\mathsf{c}'$ is well-defined, since if $\mathsf{c}(\sigma') = (\tau', B')$ and $\mathsf{c}(\sigma'') = (\tau'', B'')$ where $\sigma' \sqsubset \sigma$ and $\sigma'' \sqsubset \sigma$, then either $\sigma' \sqsubset \sigma''$ or $\sigma'' \sqsubset \sigma'$, and hence in both cases $(t', B') = (t'', B'')$ due to property (2.2). Next we note that $\mathsf{c}'(\sigma) = (\tau, B)$ if and only if there is $\sigma' \sqsubset \sigma$ with $\mathsf{c}(\sigma') = (\tau, B)$ and $\mathrm{end}(\sigma') > \tau$, hence property (2.1) follows immediately.

We show property (2.2), namely that if $\mathsf{c}'(\sigma) = (\tau, B)$ for some (finite or infinite) trace and some breach $(\tau, B)$, then $\mathsf{c}'(\sigma') = (\tau, B)$, for any (finite or infinite) trace $\sigma'$ with $\sigma'_\tau = \sigma_\tau$. We can have one of the following cases:

- $\sigma$ is finite and $\sigma'$ is finite. This case follows directly from the hypotheses of the lemma.
- $\sigma$ is finite and $\sigma'$ is infinite. Then $\mathsf{c}'(\sigma) = \mathsf{c}(\sigma) = \mathsf{c}(\sigma_\tau)$. Let $\epsilon$ be such that $\sigma'_\tau \epsilon \sqsubset \sigma'$. We have $\mathrm{ts}(\epsilon) > \tau$, hence $\mathrm{end}(\sigma'_\tau \epsilon) > \tau$. Moreover, $\mathsf{c}(\sigma'_\tau \epsilon) = (\tau, B)$ as $(\sigma'_\tau \epsilon)_\tau = \sigma_\tau$. Hence, by definition, $\mathsf{c}'(\sigma') = (\tau, B)$.
- $\sigma$ is infinite and $\sigma'$ is finite. By definition of $\mathsf{c}'$, there is $\sigma'' \sqsubset \sigma$ such that $\mathsf{c}(\sigma'') = (\tau, B)$ and $\mathrm{end}(\sigma'') > \tau$. Then $\mathsf{c}(\sigma''_\tau) = (\tau, B)$. As $\sigma'_\tau = \sigma''_\tau$, it follows that $\mathsf{c}(\sigma') = (\tau, B) = \mathsf{c}'(\sigma')$.
- $\sigma$ is infinite and $\sigma'$ is infinite. As in the previous case, there is $\sigma'' \sqsubset \sigma$ such that $\mathsf{c}(\sigma''_\tau) = (\tau, B)$ and $\mathrm{end}(\sigma'') > \tau$. Then $\sigma''_\tau = \sigma_\tau = \sigma'_\tau$. Let $\epsilon$ be such that $\sigma'_\tau \epsilon \sqsubset \sigma'$. As in the second case, we obtain that $\mathsf{c}'(\sigma') = \mathsf{c}(\sigma''_\tau) = (\tau, B)$.

This shows that $\mathsf{c}'$ is a contract between parties $P$ starting at time $\tau_0$. We now prove that this extension is unique. Let $\mathsf{c}''$ be a contract such that $\mathsf{c}''|_{\mathsf{Tr}^{\tau_0}_{\text{fin}}} = \mathsf{c}$. We show that $\mathsf{c}' = \mathsf{c}''$. The contracts $\mathsf{c}'$ and $\mathsf{c}''$ agree on all finite traces by construction, so assume for the sake of contradiction that $\mathsf{c}'(\sigma) \neq \mathsf{c}''(\sigma)$ for some infinite trace $\sigma$. Then either $\mathsf{c}'(\sigma) = (\tau, B)$ or $\mathsf{c}''(\sigma) = (\tau, B)$, for some $\tau$ and $B$, so assume that $\mathsf{c}'(\sigma) = (\tau, B)$. Then by Lemma 2.2.3 we have that $\mathsf{c}'(\sigma_\tau) = (\tau, B)$, and since $\sigma_\tau$ is finite, also $\mathsf{c}''(\sigma_\tau) = (\tau, B)$, and hence again by Lemma 2.2.3 we have that $\mathsf{c}''(\sigma) = (\tau, B)$, which is a contradiction. The case where $\mathsf{c}''(\sigma) = (\tau, B)$ is symmetric. □

### 2.2.4 Contract Composition

By composing contracts, through conjunction and disjunction, new contracts are obtained. Given that a contract assigns verdicts to traces, defining such compositions amounts to stating how verdicts are composed.

**Contract conjunction** This type of composition models the simultaneous commitment to several (sub)contracts. Conjunction is implicit in paper contracts: typically the involved parties have to conform with all the clauses therein. When some parties do not conform with some clauses, the resolution of blame assignment is given by the fundamental breach assumption: the earliest breach represents the overall verdict. When breaches of several clauses happen at the same time, then all breaching parties are to be blamed.

**Definition 2.2.6.** Let $\nu_1, \nu_2 \in \mathsf{V}$ be two verdicts. The *verdict conjunction* $\nu_1 \wedge \nu_2$ of $\nu_1$ and $\nu_2$ is given by:

$$\nu_1 \wedge \nu_2 = \begin{cases} \nu_1 & \text{if either } \nu_2 = \checkmark, \\ & \quad \text{or } \nu_1 = (\tau_1, B_1),\ \nu_2 = (\tau_2, B_2), \text{ and } \tau_1 < \tau_2, \\ \nu_2 & \text{if either } \nu_1 = \checkmark, \\ & \quad \text{or } \nu_1 = (\tau_1, B_1),\ \nu_2 = (\tau_2, B_2), \text{ and } \tau_1 > \tau_2, \\ (\tau, B) & \text{if } \nu_1 = (\tau, B_1),\ \nu_2 = (\tau, B_2), \text{ and } B = B_1 \cup B_2. \end{cases}$$

**Definition 2.2.7.** Let $\mathsf{c}_1 : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ and $\mathsf{c}_2 : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ be two contracts. The *conjunction* of contracts is defined by:

$$(\mathsf{c}_1 \wedge \mathsf{c}_2)(\sigma) = \mathsf{c}_1(\sigma) \wedge \mathsf{c}_2(\sigma).$$

Note that $(\mathsf{c}_1 \wedge \mathsf{c}_2)(\sigma) = \checkmark$ if and only if $\mathsf{c}_1(\sigma) = \mathsf{c}_2(\sigma) = \checkmark$, for any trace $\sigma$.

The following lemma confirms the intuition that the conjunction of two contracts is a contract.

**Lemma 2.2.8.** *Let* $\mathsf{c}_1 : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ *and* $\mathsf{c}_2 : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ *be two contracts between parties* $P_1$ *and* $P_2$*, respectively. Then the composition* $\mathsf{c}_1 \wedge \mathsf{c}_2 : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ *is a contract between parties* $P_1 \cup P_2$*.*

*Proof.* Property (2.1) follows immediately from the definition of verdict conjunction, so we need to prove property (2.2). Suppose that $(\mathsf{c}_1 \wedge \mathsf{c}_2)(\sigma) = (\tau, B)$ and $\sigma'$ is such that $\sigma'_\tau = \sigma_\tau$. We can have one of the following cases:

- $\mathsf{c}_1(\sigma) = \checkmark$. Then $\mathsf{c}_2(\sigma) = (\tau, B)$ and it follows that $\mathsf{c}_2(\sigma') = (\tau, B)$.

  If $\mathsf{c}_1(\sigma') = \checkmark$ then clearly $(\mathsf{c}_1 \wedge \mathsf{c}_2)(\sigma') = (\tau, B)$. Suppose that $\mathsf{c}_1(\sigma') = (\tau', B')$ for some $(\tau', B') \neq (\tau, B)$. If $\tau' \leq \tau$ then $\sigma'_{\tau'} \sqsubset \sigma'_{\tau} \sqsubset \sigma$ and hence $\mathsf{c}_1(\sigma) = (\tau', B')$—contradiction. Hence $\tau' > \tau$. Since $(\tau, B) \wedge (\tau', B') = (\tau, B)$ it follows that $(\mathsf{c}_1 \wedge \mathsf{c}_2)(\sigma) = (\tau, B)$.

- $\mathsf{c}_2(\sigma) = \checkmark$. This case is symmetric to the previous one.

- $\mathsf{c}_1(\sigma) = (\tau_1, B_1)$ and $\mathsf{c}_2(\sigma) = (\tau_2, B_2)$ such that $(\tau_1, B_1) \wedge (\tau_2, B_2) = (\tau, B)$. We then have $\mathsf{c}_1(\sigma') = (\tau_1, B_1)$ and $\mathsf{c}_1(\sigma') = (\tau_2, B_2)$. Hence $(\mathsf{c}_1 \wedge \mathsf{c}_2)(\sigma') = (\tau, B)$.

□

**Example 2.2.9.** Continuing Example 2.2.2, the first part of Paragraph 3 in Figure 2.1 (that is, "Buyer agrees to pay for the goods half upon receipt") can be represented by the contract $\mathsf{c}_3$ between {Buyer}, where:

$$\mathsf{c}_3(\sigma) = \begin{cases} \checkmark & \text{if } D = \emptyset \text{, or if } D \neq \emptyset \text{ and } \sigma[j] = (\tau_1, \text{payment}_1) \\ & \qquad\qquad\qquad \text{for some } j \text{ with } i_1 < j < |\sigma|, \\ (\tau_1, \{\text{Buyer}\}) & \text{otherwise,} \end{cases}$$

with $D = \{i \mid \sigma[i] = (\tau, \text{delivery}), 0 \leq i < |\sigma|, \tau \leq \tau_d\}$, $i_1 = \min(D)$, and $\tau_1 = \text{ts}(\sigma[i_1])$. Furthermore, the action payment$_1$ represents the first half payment to the Seller, and $i_1$ ($\tau_1$) is the index (timestamp) that represents the receipt time of the first delivery, assuming that delivery time and receipt time coincide.

The second part of Paragraph 3 (that is, "Buyer agrees to pay [...] the remainder within 30 days of delivery") can be encoded by the contract $\mathsf{c}'_3$ between {Buyer}, where:

$$\mathsf{c}'_3(\sigma) = \begin{cases} \checkmark & \text{if } D = \emptyset \text{, or if } D \neq \emptyset \text{ and } \sigma[j] = (\tau, \text{payment}_2) \\ & \qquad\qquad\qquad \text{for some } i_1 < j < |\sigma| \text{ and } \tau \leq \tau'_1, \\ (\tau'_1, \{\text{Buyer}\}) & \text{otherwise,} \end{cases}$$

with $\tau'_1 = \tau_1 + 30$ (we assume that the time unit is 1 day), and the action payment$_2$ represents the second half payment to the Seller.

Using the previous lemma, Paragraph 3 of Figure 2.1 is represented by the contract $\mathsf{c}_3 \wedge \mathsf{c}'_3$ between {Buyer}.

**Contract disjunction** This type of composition models the situation where fulfilling only one of the clauses of a contract is sufficient to fulfil the entire contract. Unlike conjunction, the case when all clauses are breached is problematic, as each of the clauses is individually an option. To be able to give an answer in this case, we take a global view: all involved parties are at any time aware of the contract execution status. Thus, those parties responsible for the latest breach are to blame for the overall failure, because they should have fulfilled their obligations after knowing that other options are not available anymore. Still, when breaches happen at the same time, there is no other way than to choose nondeterministically between the breaches. Note that blaming the parties altogether is not a better alternative, as then the nondeterminism would be hidden somewhere else: the cause of the overall failure could be any of the causes of the individual breaches.

It is not a surprise that the treatment of disjunction is more complicated, since disjunction is inherently nondeterministic. Nevertheless, in the special case where all clauses stipulate commitments on the same contract participant, disjunction corresponds to a *choice* that said participant has. In this case it is clear who is to blame when all clauses are breached.

**Definition 2.2.10.** Let $\nu_1, \nu_2 \in \mathsf{V}$ be two verdicts such that if $\nu_1 = (\tau, B_1)$ and $\nu_2 = (\tau, B_2)$ then $B_1 = B_2$. The *verdict disjunction* $\nu_1 \vee \nu_2$ of $\nu_1$ and $\nu_2$ is given by:

$$\nu_1 \vee \nu_2 = \begin{cases} \checkmark & \text{if } \nu_1 = \checkmark \text{ or } \nu_2 = \checkmark, \\ (\tau_1, B_1) & \text{if } \nu_1 = (\tau_1, B_1), \nu_2 = (\tau_2, B_2), \text{ and } \tau_1 > \tau_2, \\ (\tau_2, B_2) & \text{if } \nu_1 = (\tau_1, B_1), \nu_2 = (\tau_2, B_2), \text{ and } \tau_1 < \tau_2, \\ (\tau, B) & \text{if } \nu_1 = \nu_2 = (\tau, B). \end{cases}$$

Two contracts $\mathsf{c}_1$ and $\mathsf{c}_2$ have *unique blame assignment* if for all traces $\sigma$, whenever $\mathsf{c}_1(\sigma) = (\tau, B_1)$ and $\mathsf{c}_2(\sigma) = (\tau, B_2)$, then $B_1 = B_2$.

**Definition 2.2.11.** Let $\mathsf{c}_1 : \mathsf{Tr}^{\mathcal{T}_0} \to \mathsf{V}$ and $\mathsf{c}_2 : \mathsf{Tr}^{\mathcal{T}_0} \to \mathsf{V}$ be two contracts with unique blame assignment. The *disjunction* of contracts $\mathsf{c}_1$ and $\mathsf{c}_2$ is defined by:

$$(\mathsf{c}_1 \vee \mathsf{c}_2)(\sigma) = \mathsf{c}_1(\sigma) \vee \mathsf{c}_2(\sigma).$$

Note that $(\mathsf{c}_1 \vee \mathsf{c}_2)(\sigma) = \checkmark$ if and only if $\mathsf{c}_1(\sigma) = \checkmark$ or $\mathsf{c}_2(\sigma) = \checkmark$, for any $\sigma \in \mathsf{Tr}^{\mathcal{T}_0}$.

The following lemma confirms the intuition that the disjunction of two contracts is a contract.

**Lemma 2.2.12.** *Let* $\mathsf{c}_1 : \mathsf{Tr}^{\mathcal{T}_0} \to \mathsf{V}$ *and* $\mathsf{c}_2 : \mathsf{Tr}^{\mathcal{T}_0} \to \mathsf{V}$ *be two contracts with unique blame assignment, between parties* $P_1$ *and* $P_2$*, respectively. Then the composition* $\mathsf{c}_1 \vee \mathsf{c}_2 : \mathsf{Tr}^{\mathcal{T}_0} \to \mathsf{V}$ *is a contract between parties* $P_1 \cup P_2$*.*

*Proof.* Property (2.1) follows immediately from the definition of verdict disjunction, so we need to prove property (2.2). Suppose that $(\mathsf{c}_1 \vee \mathsf{c}_2)(\sigma) = (\tau, B)$ and $\sigma'$ is such that $\sigma'_\tau = \sigma_\tau$. We can have one of the following cases:

- $\mathsf{c}_1(\sigma) = (\tau, B)$ and $\mathsf{c}_2(\sigma) = (\tau_2, B_2)$ with $\tau_2 < \tau$. It follows that $\mathsf{c}_1(\sigma') = (\tau, B)$ and $\mathsf{c}_2(\sigma_{\tau_2}) = (\tau_2, B_2)$. As $\sigma_{\tau_2} \sqsubset \sigma_\tau \sqsubset \sigma'$, we have $\mathsf{c}_2(\sigma') = (\tau_2, B_2)$. Hence $(\mathsf{c}_1 \vee \mathsf{c}_2)(\sigma) = (\tau, B)$.

- $\mathsf{c}_2(\sigma) = (\tau, B)$ and $\mathsf{c}_1(\sigma) = (\tau_1, B_1)$ with $\tau_1 < \tau$. This case is symmetric to the previous one.

- $\mathsf{c}_1(\sigma) = (\tau, B)$ and $\mathsf{c}_2(\sigma) = (\tau, B)$. We then have $\mathsf{c}_1(\sigma') = (\tau, B)$ and $\mathsf{c}_2(\sigma') = (\tau, B)$. Hence $(\mathsf{c}_1 \vee \mathsf{c}_2)(\sigma') = (\tau, B)$.

□

**Example 2.2.13.** Continuing Example 2.2.9, the second part of Paragraph 4 in Figure 2.1 (that is, “an additional fine of 10% has to be paid within 14 days”) can be encoded by the contract $\mathsf{c}_4$ between {Buyer}:

$$\mathsf{c}_4(\sigma) = \begin{cases} \checkmark & \text{if } D = \emptyset\text{, or if } D \neq \emptyset \text{ and } \sigma[j] = (\tau, \text{payment}'_2) \\ & \qquad\qquad \text{for some } i_1 < j < |\sigma| \text{ and } \tau \leq \tau''_1, \\ (\tau''_1, \{\text{Buyer}\}) & \text{otherwise,} \end{cases}$$

where $\tau''_1 = \tau_1 + 44$ and the action payment$'_2$ represents the payment of the second half together with the 10% fine by Buyer. (Note that the confusion with regard to the reference for the 10% computation will have to be solved at a different level—when defining payment$'_2$ concretely.)

As, for all traces, the contracts $\mathsf{c}'_3$ and $\mathsf{c}_4$ only blame Buyer, the previous lemma ensures that $\mathsf{c}'_3 \vee \mathsf{c}_4$ is a well-defined contract. The first four paragraphs are thus represented by the contract $\mathsf{c}_1 \wedge (\mathsf{c}_3 \wedge (\mathsf{c}'_3 \vee \mathsf{c}_4))$ between {Buyer, Seller}. (We note that Paragraph 2 of Figure 2.1 is encoded implicitly in the encoding of the other paragraphs.)

**Algebraic properties of contract composition** The following lemma shows that the conjunction and disjunction operators on verdicts enjoy the expected algebraic properties, like commutativity, associativity, and distributivity.

**Lemma 2.2.14.** *Let $\nu, \nu_1, \nu_2, \nu_3, \nu'_1, \nu'_2, \nu'_3$ be verdicts such that if $\nu'_i = (\tau, B_i)$ and $\nu'_j = (\tau, B_j)$ then $B_i = B_j$, for any $i, j \in \{1, 2, 3\}$. Then the following equalities hold:*

$$\begin{aligned} \nu_1 \wedge \nu_2 &= \nu_2 \wedge \nu_1 && \text{(commutativity)} \\ \nu'_1 \vee \nu'_2 &= \nu'_2 \vee \nu'_1 && \text{(commutativity)} \\ \nu_1 \wedge (\nu_2 \wedge \nu_3) &= (\nu_1 \wedge \nu_2) \wedge \nu_3 && \text{(associativity)} \\ \nu'_1 \vee (\nu'_2 \vee \nu'_3) &= (\nu'_1 \vee \nu'_2) \vee \nu'_3 && \text{(associativity)} \\ \nu'_1 \vee (\nu'_1 \wedge \nu'_2) &= \nu'_1 && \text{(absorption)} \\ \nu'_1 \wedge (\nu'_1 \vee \nu'_2) &= \nu'_1 && \text{(absorption)} \\ \nu'_1 \vee (\nu'_2 \wedge \nu'_3) &= (\nu'_1 \vee \nu'_2) \wedge (\nu'_1 \vee \nu'_3) && \text{(distributivity)} \\ \nu_1 \wedge (\nu'_2 \vee \nu'_3) &= (\nu_1 \wedge \nu'_2) \vee (\nu_1 \wedge \nu'_3) && \text{(distributivity)} \\ \checkmark \wedge \nu &= \nu \wedge \checkmark = \nu && \text{(unit)} \\ \checkmark \vee \nu &= \nu \vee \checkmark = \checkmark && \text{(unit)} \end{aligned}$$

*Proof.* These equalities follow directly from Definitions 2.2.6 and 2.2.10. □

These algebraic properties are easily lifted from verdicts to contracts, which allows us to perform algebraic, meaning-preserving rewritings of contracts.

**Corollary 2.2.15.** *Let $C$ be a set of contracts that is closed under contract conjunction and disjunction, $\mathsf{c}_\checkmark \in C$, and for all $\mathsf{c}_1, \mathsf{c}_2 \in C$, the contracts $\mathsf{c}_1$ and $\mathsf{c}_2$ have unique blame assignment. Then $(C, \vee, \wedge)$ is a distributive lattice with unit element $\mathsf{c}_\checkmark$.*

We recall that the idempotency equalities $\mathsf{c} \wedge \mathsf{c} = \mathsf{c}$ and $\mathsf{c} \vee \mathsf{c} = \mathsf{c}$, that hold for any contract $\mathsf{c}$, follow from the absorption equalities. We also note that the equalities that only concern conjunction hold for arbitrary contracts.

### 2.2.5 Run-time Monitoring

The contract model presented above considers complete traces, which are either finite or infinite, and there is no restriction as to whether the verdict of a contract can be computed or not. For run-time monitoring, however, traces are always partial and finite, and it should be possible to compute verdicts at run-time. We consequently define, abstractly, what constitutes run-time monitoring for the contract model, using a conventional many-valued semantics [59].

The output of a run-time monitor is an element of the union of the sets $\mathsf{V}_\star = \{\nu_\star \mid \nu \in \mathsf{V}\}$ for $\star \in \{!, ?\}$, where $\nu_!$ is a *final verdict*, and $\nu_?$ is a *potential verdict*. Final verdicts are output when all extensions of the current partial trace have the same verdict. In other words, the verdict on the complete trace, whatever this would be, is uniquely determined by (the verdict on) the partial trace; there is hence no need to perform further monitoring. In contrast, potential verdicts are output when the verdicts on extensions of the current partial trace differ. Of course, if the current trace is a complete trace (in this case no more events occur), then the potential verdict is the actual verdict on this trace.

**Definition 2.2.16.** Let $\mathsf{c} : \mathsf{Tr}^{\mathcal{T}_0} \to \mathsf{V}$ be a contract between parties $P$. A *run-time monitor* for $\mathsf{c}$ is a *computable* function $\mathrm{mon} : \mathsf{Tr}^{\mathcal{T}_0}_{\mathrm{fin}} \to \mathsf{V}_! \cup \mathsf{V}_?$ that satisfies:

$$\mathrm{mon}(\sigma) = \begin{cases} \nu_! & \text{if } \mathsf{c}(\sigma') = \nu \text{ for all } \sigma' \text{ with } \sigma \sqsubset \sigma', \\ \nu_? & \text{if } \mathsf{c}(\sigma) = \nu \text{ and } \mathsf{c}(\sigma') \neq \nu \text{ for some } \sigma \sqsubset \sigma'. \end{cases}$$

Note that, in case of a potential breach, that is if $\mathrm{mon}(\sigma) = (\tau, B)_?$ then condition (2.2) of Definition 2.2.1 guarantees that $\mathrm{end}(\sigma) \leq \tau$, hence $(\tau, B)_?$ is always an indication of a future—but avoidable—breach.

The definition expresses both *impartiality* and *anticipation* [59]. Impartiality means that a final verdict is only output if the partial trace cannot be extended into a complete trace with a different verdict. Formally:

$$\text{if } \mathrm{mon}(\sigma) = \nu_! \text{ then } \mathsf{c}(\sigma') = \nu \text{ for all } \sigma' \text{ with } \sigma \sqsubset \sigma'.$$

Anticipation is the reverse of impartiality. It means that inevitable—possibly future—verdicts are output as early as possible, that is a potential verdict is only output if it is possible to reach a different verdict. Formally:

$$\text{if } \mathsf{c}(\sigma') = \nu \text{ for all } \sigma' \text{ with } \sigma \sqsubset \sigma' \text{ then } \mathrm{mon}(\sigma) = \nu_!.$$

Anticipation can be relaxed, for instance by allowing final breaches to be output only when the time of breach has been reached, but impartiality is a crucial requirement for run-time monitoring that cannot be relaxed.

**Example 2.2.17.** Consider the contract $\mathsf{c}_1 \wedge (\mathsf{c}_3 \wedge (\mathsf{c}_3' \vee \mathsf{c}_4))$ between $\{\text{Buyer}, \text{Seller}\}$ from Example 2.2.13, and the following events:

$$\begin{aligned} \epsilon_1 &= (\text{2011-01-01}, \text{delivery}), & \epsilon_4 &= (\text{2011-01-10}, \text{payment}_2), \\ \epsilon_2 &= (\text{2011-01-02}, \text{delivery}), & \epsilon_5 &= (\text{2011-02-10}, \text{payment}_2'). \\ \epsilon_3 &= (\text{2011-01-01}, \text{payment}_1), \end{aligned}$$

The outputs of an associated run-time monitor on the following sample traces are as follows:

$$\begin{aligned} \text{mon}\left(\langle\rangle\right) &= (\text{2011-01-01}, \{\text{Seller}\})_?, \\ \text{mon}\left(\langle\epsilon_2\rangle\right) &= (\text{2011-01-01}, \{\text{Seller}\})_!, \\ \text{mon}\left(\langle\epsilon_1\rangle\right) &= (\text{2011-01-01}, \{\text{Buyer}\})_?, \\ \text{mon}\left(\langle\epsilon_1, \epsilon_3\rangle\right) &= (\text{2011-02-14}, \{\text{Buyer}\})_?, \\ \text{mon}\left(\langle\epsilon_1, \epsilon_3, \epsilon_4\rangle\right) &= \text{mon}\left(\langle\epsilon_1, \epsilon_3, \epsilon_5\rangle\right) = \checkmark_!. \end{aligned}$$

## 2.3 A Contract Specification Language

The previous section provided a semantic account for compositional contracts. However, it is cumbersome to specify contracts directly in the abstract model, as we have seen in Examples 2.2.2–2.2.13. Thus we propose a contract specification language, CSL, which enables succinct, syntactic representation of real-world contracts in a human-readable form, and which has a formal semantics in terms of the abstract contract model. The primary target of CSL is business contracts, but rather than fixing the set of actions to for instance payments and deliveries, we parametrise the language with respect to a signature, which can be thought of as the vocabulary used in a contract.

Formally, a *signature* is a triple $S = (\mathcal{K}, \text{ar}, \mathcal{T})$, where $\mathcal{K}$ is a finite set of *action kinds* with associated *arities* and *types*, $\text{ar} : \mathcal{K} \to \mathcal{T}^*$, where $\mathcal{T}$ is a finite set of types. The *domain* of a type $t$ is denoted by $[\![t]\!]$, and we assume that $\mathcal{T}$ contains the basic types Bool, Int, Time, and Party, with the corresponding domains $[\![\text{Bool}]\!] = \{\textbf{false}, \textbf{true}\}$, $[\![\text{Int}]\!] = \mathbb{Z}$, $[\![\text{Time}]\!] = \mathsf{Ts}$, and $[\![\text{Party}]\!] = \mathsf{P}$, respectively. Signatures provide structure to actions, and we consequently redefine the set of actions, with respect to a given signature, as follows:

$$\mathsf{A} = \{k(\vec{v}) \mid k \in \mathcal{K}, \text{ar}(k) = \langle t_1, \ldots, t_n\rangle, \text{ and } \vec{v} \in [\![t_1]\!] \times \cdots \times [\![t_n]\!]\}.$$

Furthermore, we assume an infinite set of variables $\mathcal{V}$, ranged over by $x, y, z$, and an infinite set of template names $\mathcal{F}$, ranged over by $f$.

### 2.3.1 CSL Syntax

The grammar of CSL is presented in Figure 2.2. In what follows, we describe informally each construct of the language.

The atomic expressions of CSL are values $v \in [\![t]\!]$ of some type $t$ and variables. From integer values and variables, arithmetic and Boolean *expressions* are formed by using arithmetic operators, equalities, and inequalities. We note in particular that "/" denotes integer division and the specification needs to take into account the

$$
\begin{array}{lll}
s & ::= \textbf{letrec } \{f_i(\vec{x}_i)\langle\vec{y}_i\rangle = c_i\}_{i=1}^{n} \textbf{ in } c \textbf{ starting } \tau & \text{(CSL specification)} \\
c & ::= \textbf{fulfilment} & \text{(no obligations)} \\
 & \mid \langle e_1\rangle\ k(\vec{x}) \textbf{ where } e_2 \textbf{ due } d \textbf{ remaining } z \textbf{ then } c & \text{(obligation)} \\
 & \mid \textbf{if } k(\vec{x}) \textbf{ where } e \textbf{ due } d \textbf{ remaining } z \textbf{ then } c_1 \textbf{ else } c_2 & \text{(external choice)} \\
 & \mid \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 & \text{(internal choice)} \\
 & \mid c_1 \textbf{ and } c_2 & \text{(conjunction)} \\
 & \mid c_1 \textbf{ or } c_2 & \text{(disjunction)} \\
 & \mid f(\vec{e}_1)\langle\vec{e}_2\rangle & \text{(instantiation)} \\
e & ::= x \mid v \mid \neg e \mid e_1 \star e_2 \mid e_1 \prec e_2 & \text{(expression)} \\
d & ::= \textbf{after } e_1 \textbf{ within } e_2 & \text{(deadline expression)}
\end{array}
$$

Figure 2.2: The grammar of CSL. $f \in \mathcal{F}$ ranges over template names, $x, y, z \in \mathcal{V}$ range over variables, $k \in \mathcal{K}$ ranges over action kinds, and $v \in \bigcup_{t\in\mathcal{T}}[\![t]\!]$ ranges over values. Furthermore, $\star \in \{+, -, \times, /, \wedge\}$ and $\prec \in \{<, =\}$.

possible loss in precision with regard to real division. Abusing language, a *deadline expression* actually represents an interval of integers, as explained shortly.

A CSL *specification* $s$ is a set of template definitions together with a body $c$ and an absolute point in time $\tau$, which defines the starting time of the contract. Templates can be instantiated in the body of the specification. Mutual recursion is allowed and it enables potentially infinite contract executions. The parameters of a template are values $\vec{x}$ and parties $\vec{y}$. Value parameters are dynamic, that is they can be instantiated with values from earlier events, whereas party parameters are static, that is all parties are fixed before the contract is started, and they do not change over time.

*Clauses* describe the normative content of contracts. The bodies of CSL specifications and of template definitions are clauses. All deadlines that occur in clauses are relative to unspecified reference points that are given by the starting time of the specification and by the time of event occurrences. Thus, these relative deadlines are only lifted to absolute deadlines when the CSL specification is executed. The only atomic clause is **fulfilment**, which represents the clause that is always fulfilled.

Fully instantiated obligation clauses have the form:

$$\langle p\rangle\ k(\vec{x}) \textbf{ where } e \textbf{ due after } n_1 \textbf{ within } n_2 \textbf{ remaining } z \textbf{ then } c,$$

which should be read:

> Party $p$ is responsible that (but need not be in charge of) an action of kind $k$ satisfying condition $e$ takes place. This action should happen after $n_1$ time units, but within $n_2$ time units thereafter. If these requirements are satisfied, then the *continuation clause* $c$ determines any further obligations.

The variables of the vector $\vec{x}$ are bound to the parameters of the action, and their scope is $e$ and $c$. The variable $z$ is bound to the remainder of the deadline: if the deadline is for instance **after** 2 **within** 5 and the action takes place 4 time units after the reference point, then $z$ is bound to $(2 + 5) - 4 = 3$. The scope of $z$ is $c$ only. All deadlines in the continuation $c$ are relative to the time of the action.

```
letrec sale(deliveryDeadline, goods, payment)⟨buyer, seller⟩ =
⟨seller⟩ Delivery(s,r,g)
 where s = seller ∧ r = buyer ∧ g = goods due within deliveryDeadline
then
⟨buyer⟩ Payment(s,r,a)
 where s = buyer ∧ r = seller ∧ a = payment / 2 due immediately
then
(((⟨buyer⟩ Payment(s,r,a)
   where s = buyer ∧ r = seller ∧ a = payment / 2 due within 30D
  or
  ⟨buyer⟩ Payment(s,r,a)
   where s = buyer ∧ r = seller ∧ a = (payment × 110) / 200 due within 14D after 30D)
 and
 if Return(s,r,g)
     where s = buyer ∧ r = seller ∧ g = goods due within 14D
 then
   ⟨seller⟩ Payment(s,r,a) where s = seller ∧ r = buyer ∧ a = payment due within 7D)
in
sale(0, “Laser printer”, 200)⟨Buyer, Seller⟩ starting 2011-01-01
```

Figure 2.3: A CSL specification of a sales contract between a buyer and a seller.

External choices are similar to obligation clauses, but they contain an alternative continuation branch which becomes active if the deadline passes. For this reason, external choices have no responsible party parameter, since no one has to be blamed in case the deadline expires.

The clause **if** $e$ **then** $c_1$ **else** $c_2$ represents an internal choice, where the branching condition $e$ can be computed directly without having to wait for external input (that is, for events). The clauses $c_1$ **and** $c_2$ and $c_1$ **or** $c_2$ represent clause conjunction and disjunction, respectively. Finally, $f(\vec{e}_1)\langle\vec{e}_2\rangle$ is instantiation of template $f$, where $\vec{e}_1$ are value parameters and $\vec{e}_2$ are party parameters.

We use standard syntactic sugar such as $e_1 \vee e_2$ for $\neg(\neg e_1 \wedge \neg e_2)$, $e_1 \leq e_2$ for $(e_1 < e_2) \vee (e_1 = e_2)$, and $e_1 \neq e_2$ for $\neg(e_1 = e_2)$. Also, we omit continuations and **else** branches if they are **fulfilment**, we omit the **after** part of a deadline if it is 0, we write **immediately** for **within** 0, and we omit the **remaining** part if it is not used. Finally, we use abbreviations like 30D to denote the value representing an amount of time of 30 days, that is the integer $30 * 24 * 60 * 60$, assuming that the time unit is of one second.

In terms of deontic modalities [111], it may seem that CSL only supports obligations, and not permissions and prohibitions. However, permissions in a contractual context are only of interest if they entail new obligations (on counter parties). Hence we model permissions as external choices that trigger new obligations, as illustrated in the following example. Prohibitions can also be modelled as external choices, where the consequence is an unfulfillable obligation on the party who performed the prohibited action, as we shall see in Section 2.3.7, where we provide further examples.

**Example 2.3.1.** Figure 2.3 shows the specification in CSL of the sales contract in Figure 2.1. The formalisation assumes a signature that includes the action kinds $\{\text{Delivery}, \text{Payment}, \text{Return}\} \subseteq \mathcal{K}$, with types $\text{ar}(\text{Delivery}) = \text{ar}(\text{Return}) = \langle\text{Party},$

Party, String⟩ and ar(Payment) = ⟨Party, Party, Int⟩. The domain of String is the set of all strings, and the two parties of each action kind represent the sender and receiver, respectively. The example disambiguates the informal contract: the 10% fine is calculated with respect to half of the total price, and Buyer is only entitled to return the goods if the first half is paid upon delivery. A different disambiguation could be given by another CSL specification. Note also how we encode the permission to return the goods as an external choice that has the consequence that Seller has to pay the original amount back to Buyer.

### 2.3.2 CSL Type System

We equip CSL with a type system. For this purpose, we define different *typing judgements* over an implicit signature $S = (\mathcal{K}, \mathrm{ar}, \mathcal{T})$. Before presenting the typing judgements, we introduce some notation. We write $f : A \rightharpoonup_{\mathrm{fin}} B$ for a partial function $f$ from $A$ to $B$ with a finite domain. Furthermore, $f[a \mapsto b]$ denotes the function that maps $a$ to $b$ and behaves like $f$ on all other input. We write $f[\vec{a} \mapsto \vec{b}]$ for $f[a_1 \mapsto b_1] \cdots [a_n \mapsto b_n]$, for vectors $\vec{a} = (a_1, \dots, a_n)$ and $\vec{b} = (b_1, \dots, b_n)$. Finally, we write $A \subseteq_{\mathrm{fin}} B$ to say that $A \subseteq B$ and $A$ is finite.

Our typing judgements use the following typing environments:

$$\Lambda \subseteq_{\mathrm{fin}} \mathcal{V} \qquad \text{(party typing environment)}$$
$$\Gamma : \mathcal{V} \rightharpoonup_{\mathrm{fin}} \mathcal{T} \qquad \text{(variable typing environment)}$$
$$\Delta : \mathcal{F} \rightharpoonup_{\mathrm{fin}} \mathcal{T}^* \times \mathbb{N} \qquad \text{(template typing environment)}$$

The typing environment for parties $\Lambda$ keeps track of parametrised parties (such as the parameter *buyer* of the template *sale* in Figure 2.3), and the typing environment for values $\Gamma$ keeps track of parametrised values and their type (such as the parameter *goods* of the template *sale* in Figure 2.3). The typing environment for clause templates $\Delta$ associates with each template name the types of its parameters and the number of party parameters. Also, we use the meta-types Deadline, Clause⟨$P$⟩, and Contract⟨$P$⟩, parametrised by a finite set of parties $P \subseteq_{\mathrm{fin}} \mathsf{P}$, to represent the type of deadlines, clauses involving parties $P$, and contracts involving parties $P$, respectively.

The typing judgements for expressions $\Gamma \vdash e : t$, for party expressions (that is, the expressions determining responsibility in obligations) $\Lambda \vdash e' : P$, and for deadline expressions $\Gamma \vdash d : \text{Deadline}$ are presented in Figure 2.4. The typing rules for expressions are standard, but we require that the denominator of a division expression be known statically in order to avoid division by zero. The typing rules for party expressions $\Lambda \vdash e' : P$ are used to determine the parties that are involved in a given clause.

The typing rules for clauses $\Delta, \Lambda, \Gamma \vdash c : \text{Clause}\langle P \rangle$, for template definitions $\Delta \vdash D$, and for full CSL specifications $\vdash s : \text{Contract}\langle P \rangle$ are presented in Figure 2.5. A derivation $\Delta, \Lambda, \Gamma \vdash c : \text{Clause}\langle P \rangle$ intuitively means that in template environment $\Delta$ and variable environment $\Gamma$, $c$ is a clause in which only parties $P$ and parametrised parties $\Lambda$ can be blamed for a breach of contract. The typing rule for clause disjunction, $c_1$ **or** $c_2$, uses this invariant to check that at most one party can breach either $c_1$ or $c_2$, which guarantees that verdict disjunction is well-defined. The typing rules for obligations and external choices illustrate the scope of the bound variables $\vec{x}$ and $z$.

$$\boxed{\Gamma \vdash e : t} \qquad \frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{v \in [\![t]\!]}{\Gamma \vdash v : t}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{Int} \qquad \Gamma \vdash e_2 : \mathrm{Int}}{\Gamma \vdash e_1 \star e_2 : \mathrm{Int}} \ (\star \in \{+, -, *\}) \qquad \frac{\Gamma \vdash e_1 : \mathrm{Int} \qquad n_2 \in [\![\mathrm{Int}]\!]}{\Gamma \vdash e_1 / n_2 : \mathrm{Int}} \ (n_2 \neq 0)$$

$$\frac{\Gamma \vdash e : \mathrm{Bool}}{\Gamma \vdash \neg e : \mathrm{Bool}} \qquad \frac{\Gamma \vdash e_1 : \mathrm{Bool} \qquad \Gamma \vdash e_2 : \mathrm{Bool}}{\Gamma \vdash e_1 \wedge e_2 : \mathrm{Bool}}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{Int} \qquad \Gamma \vdash e_2 : \mathrm{Int}}{\Gamma \vdash e_1 < e_2 : \mathrm{Bool}} \qquad \frac{\Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 = e_2 : \mathrm{Bool}}$$

$$\boxed{\Lambda \vdash e' : P} \qquad \frac{x \in \mathcal{V}}{\{x\} \vdash x : \emptyset} \qquad \frac{p \in \mathsf{P}}{\emptyset \vdash p : \{p\}}$$

$$\boxed{\Gamma \vdash d : \mathrm{Deadline}} \qquad \frac{\Gamma \vdash e_1 : \mathrm{Int} \qquad \Gamma \vdash e_2 : \mathrm{Int}}{\Gamma \vdash \mathbf{after}\ e_1\ \mathbf{within}\ e_2 : \mathrm{Deadline}}$$

Figure 2.4: Typing judgements for expressions $e$, party expressions $e'$, and deadline expressions $d$.

The typing rule for template definitions $\Delta \vdash D$ requires that the body of each definition contains no “hard coded” parties, that is it must only contain variables, but not values of type Party. The restriction is strictly speaking not necessary, however we consider it best practice not to have hard coded parties inside template definitions, and we therefore rule out this possibility. We furthermore allow party parameters to be used in the scope of ordinary expressions; see the definition of $\Gamma_i$, and the body of the template *sale* in Figure 2.3 for an example.

An expression $e$ is *well-typed* in the variable typing environment $\Gamma$, if there is a type $t$ such that $\Gamma \vdash e : t$. Similarly, a deadline expression $d$ is well-typed in the variable typing environment $\Gamma$, if $\Gamma \vdash d$ : Deadline. A clause $c$ involving parties $P$ is well-typed in the variable environment $\Gamma$, party environment $\Lambda$, and template environment $\Delta$, if $\Delta, \Lambda, \Gamma \vdash c : \mathrm{Clause}\langle P \rangle$. A specification $s$ involving parties $P$ is well-typed, if $\vdash s : \mathrm{Contract}\langle P \rangle$. We say simply that a CSL construct is well-typed, if there are appropriate environments and involved parties within which the construct is well-typed.

Lastly, we remark that the type system presented here is declarative, that is checking whether CSL specifications are well-typed cannot be directly implemented based on the given typing rules. This is because of the rule for template definitions, for which we have to guess the types of value parameters. An actual implementation will either rely on explicit type annotations of template parameters or perform type inference. While we treat neither approaches formally here, we note that explicit type annotations will immediately give rise to an algorithmic type system.

### 2.3.3 Well-formed Specifications

Unfolding of template definitions need not always terminate—even for well-typed specifications—as illustrated in the following example:

$$\boxed{\Delta, \Lambda, \Gamma \vdash c : \text{Clause}\langle P\rangle} \qquad \frac{}{\Delta, \emptyset, \Gamma \vdash \textbf{fulfilment} : \text{Clause}\langle \emptyset\rangle}$$

$$\frac{\begin{array}{ccc} & \Lambda_1 \vdash e_1 : P_1 & \\ \Gamma' = \Gamma[\vec{x} \mapsto \text{ar}(k)] & \Gamma' \vdash e_2 : \text{Bool} & \\ \Gamma_2 = \Gamma'[z \mapsto \text{Int}] & \Gamma \vdash d : \text{Deadline} & \Delta, \Lambda_2, \Gamma_2 \vdash c : \text{Clause}\langle P_2\rangle \end{array}}{\Delta, \Lambda_1 \cup \Lambda_2, \Gamma \vdash \langle e_1\rangle\ k(\vec{x})\ \textbf{where}\ e_2\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c : \text{Clause}\langle P_1 \cup P_2\rangle}$$

$$\frac{\begin{array}{ccc} \Gamma' = \Gamma[\vec{x} \mapsto \text{ar}(k)] & \Gamma' \vdash e : \text{Bool} & \Delta, \Lambda_1, \Gamma_1 \vdash c_1 : \text{Clause}\langle P_1\rangle \\ \Gamma_1 = \Gamma'[z \mapsto \text{Int}] & \Gamma \vdash d : \text{Deadline} & \Delta, \Lambda_2, \Gamma \vdash c_2 : \text{Clause}\langle P_2\rangle \end{array}}{\Delta, \Lambda_1 \cup \Lambda_2, \Gamma \vdash \textbf{if}\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \text{Clause}\langle P_1 \cup P_2\rangle}$$

$$\frac{\Gamma \vdash e : \text{Bool} \qquad \Delta, \Lambda_1, \Gamma \vdash c_1 : \text{Clause}\langle P_1\rangle \qquad \Delta, \Lambda_2, \Gamma \vdash c_2 : \text{Clause}\langle P_2\rangle}{\Delta, \Lambda_1 \cup \Lambda_2, \Gamma \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \text{Clause}\langle P_1 \cup P_2\rangle}$$

$$\frac{\Delta, \Lambda_1, \Gamma \vdash c_1 : \text{Clause}\langle P_1\rangle \qquad \Delta, \Lambda_2, \Gamma \vdash c_2 : \text{Clause}\langle P_2\rangle}{\Delta, \Lambda_1 \cup \Lambda_2, \Gamma \vdash c_1\ \textbf{and}\ c_2 : \text{Clause}\langle P_1 \cup P_2\rangle}$$

$$\frac{|\Lambda_1 \cup \Lambda_2| + |P_1 \cup P_2| \leq 1 \qquad \Delta, \Lambda_1, \Gamma \vdash c_1 : \text{Clause}\langle P_1\rangle \qquad \Delta, \Lambda_2, \Gamma \vdash c_2 : \text{Clause}\langle P_2\rangle}{\Delta, \Lambda_1 \cup \Lambda_2, \Gamma \vdash c_1\ \textbf{or}\ c_2 : \text{Clause}\langle P_1 \cup P_2\rangle}$$

$$\frac{\Delta(f) = (\langle t_1, \ldots, t_m\rangle, n) \qquad \forall i \in \{1, \ldots, m\}.\, \Gamma \vdash e_i : t_i \qquad \forall i \in \{1, \ldots, n\}.\, \Lambda_i \vdash e'_i : P_i}{\Delta, \bigcup_{i=1}^{n} \Lambda_i, \Gamma \vdash f(e_1, \ldots, e_m)\langle e'_1, \ldots, e'_n\rangle : \text{Clause}\langle \bigcup_{i=1}^{n} P_i\rangle}$$

$$\boxed{\Delta \vdash D} \quad \frac{\begin{array}{l} \Gamma_i = \left[\vec{x}_i \mapsto \vec{t}_i, \vec{y}_i \mapsto \overrightarrow{\text{Party}}\right] \\ \forall i, j \in \{1, \ldots, n\}. i \neq j \Rightarrow f_i \neq f_j \\ \Delta = \left[f_1 \mapsto (\vec{t}_1, |\vec{y}_1|), \ldots, f_n \mapsto (\vec{t}_n, |\vec{y}_n|)\right] \quad \forall i \in \{1, \ldots, n\}.\, \Delta, \vec{y}_i, \Gamma_i \vdash c_i : \text{Clause}\langle\emptyset\rangle \end{array}}{\Delta \vdash \{f_i(\vec{x}_i)\langle\vec{y}_i\rangle = c_i\}_{i=1}^{n}}$$

$$\boxed{\vdash s : \text{Contract}\langle P\rangle} \qquad \frac{\Delta \vdash D \qquad \Delta, \emptyset, \emptyset \vdash c : \text{Clause}\langle P\rangle}{\vdash \textbf{letrec}\ D\ \textbf{in}\ c\ \textbf{starting}\ \tau : \text{Contract}\langle P\rangle}$$

Figure 2.5: Typing judgements for CSL clauses $c$, template definitions $D$, and specifications $s$.

$$s_\Omega = \textbf{letrec}\ f()\langle\rangle = f()\langle\rangle\ \textbf{in}\ f()\langle\rangle\ \textbf{starting}\ \text{2011-01-01}$$

We avoid such ill-formed specifications by considering only specifications that satisfy a certain syntactic criterion that we introduce next.

Given a clause $c$, we recursively define the *immediate subclauses* of $c$ as follows:

$$\text{Sub}(c) = \{c\} \cup \begin{cases} \text{Sub}(c_2) & \text{if } c = \textbf{if}\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d \\ & \qquad\quad \textbf{remaining}\ z\ \textbf{then}\ c_1\ \textbf{else}\ c_2, \\ \text{Sub}(c_1) \cup \text{Sub}(c_2) & \text{if } c = c_1\ \textbf{and}\ c_2, \\ \text{Sub}(c_1) \cup \text{Sub}(c_2) & \text{if } c = c_1\ \textbf{or}\ c_2, \\ \text{Sub}(c_1) \cup \text{Sub}(c_2) & \text{if } c = \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2, \\ \emptyset & \text{otherwise.} \end{cases}$$

$$\boxed{e \Downarrow v} \quad \frac{}{v \Downarrow v} \quad \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{e_1 \star e_2 \Downarrow n_1 \star n_2}\ (\star \in \{+,-,*,/\}) \quad \frac{e \Downarrow \mathbf{true}}{\neg e \Downarrow \mathbf{false}} \quad \frac{e \Downarrow \mathbf{false}}{\neg e \Downarrow \mathbf{true}}$$

$$\frac{e_1 \Downarrow \mathbf{true} \qquad e_2 \Downarrow \mathbf{true}}{e_1 \wedge e_2 \Downarrow \mathbf{true}} \qquad \frac{e_1 \Downarrow \mathbf{false}}{e_1 \wedge e_2 \Downarrow \mathbf{false}} \qquad \frac{e_2 \Downarrow \mathbf{false}}{e_1 \wedge e_2 \Downarrow \mathbf{false}}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 \prec e_2 \Downarrow b} \left( \prec \in \{<,=\}, b = \begin{cases} \mathbf{true}, & \text{if } v_1 \prec v_2 \\ \mathbf{false}, & \text{if } v_1 \not\prec v_2 \end{cases} \right)$$

$$\boxed{d \Downarrow^{\tau} (\tau_1, \tau_2)} \qquad \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{\mathbf{after}\ e_1\ \mathbf{within}\ e_2 \Downarrow^{\tau} (\tau + n_1, \tau + n_1 + n_2)}$$

Figure 2.6: Evaluation of expressions and deadline expressions.

Given a set of template definitions $D$, we let $\mathcal{F}_D$ denote the names of the templates defined in $D$. The *immediate unfolding relation* $\Rightarrow_D$ on $\mathcal{F}_D$ is defined as follows: $f \Rightarrow_D g$ if and only if there is a subclause $g(\vec{e}_1)\langle\vec{e}_2\rangle \in \mathrm{Sub}(c_f)$ where $c_f$ is such that $(f(\vec{x})\langle\vec{y}\rangle = c_f) \in D$. Intuitively, $\Rightarrow_D$ represents a dependency relation between templates, where $f \Rightarrow_D g$ means that the unfolding of $f$ requires an immediate unfolding of $g$. The definition of immediate subclauses reflects this intuition. For instance, in the continuation clause $c$ of an obligation, the templates in $c$ are not immediately instantiated—they are instantiated only after the obligation is fulfilled.

We say that a specification $s$ is *well-formed* with parties $P$, if $s$ involving parties $P$ is well-typed and the immediate unfolding relation on the template names of $s$ is acyclic. By requiring that the unfolding relation be acyclic, we avoid exactly those cases where the unfolding of a template $f$ requires a series of immediate unfoldings leading to an unfolding of $f$ itself. Note that the specification given in Figure 2.3 is well-formed, while the specification $s_\Omega$ above is not.

### 2.3.4 CSL Semantics

We now present the operational semantics for CSL, which is used to define the mapping of CSL specifications to abstract contracts, and which gives rise to a run-time monitoring algorithm as well. Inspired by Andersen et al. [6], we define a reduction semantics, which has the advantage that residual obligations, after an event has taken place, can be seen directly by inspecting the reduced term. More generally it follows that any analysis applicable to initial CSL specifications will also be applicable at any given point in time, since running CSL specifications are conceptually no different from initial specifications.

We first define the evaluation of well-typed expressions $e \Downarrow v$ and well-typed deadline expressions $d \Downarrow^{\tau} (\tau_1, \tau_2)$ in Figure 2.6, using standard derivation rules. The timestamp $\tau$ in the rule for deadlines is the time with respect to which relative deadlines are calculated. It represents the starting time of the specification or the time of its last update, which equals the time of the last event occurrence. The following lemma shows the expected correspondence between the typing rules and the evaluation rules for (deadline) expressions.

**Lemma 2.3.2.** *Let $e$ be an expression, $d$ be a deadline expression, and $t$ be a type. If $\emptyset \vdash e : t$, then there is a unique $v \in [\![t]\!]$ such that $e \Downarrow v$. If $\emptyset \vdash d : \text{Deadline}$, then for any $\tau \in \mathsf{Ts}$, there are unique $\tau_1, \tau_2 \in \mathbb{Z}$ with $d \Downarrow^{\tau} (\tau_1, \tau_2)$.*

*Proof.* For the first claim, existence follows by induction on the derivation of $\emptyset \vdash e : t$, while uniqueness follows by structural induction on $e$. The last claim follows immediately from the first one. $\square$

During reductions, variables are instantiated with values in expressions and clauses. Since party parameters do not depend on event data, we use two kinds of (applications of) substitutions, namely substitutions of value parameters and substitutions of party parameters. Formally, a *(value) substitution* is an element of the set $\mathcal{V} \rightharpoonup_{\text{fin}} \bigcup_{t \in \mathcal{T}} [\![t]\!]$. A *party substitution* is a substitution having $\mathsf{P}$ as the codomain. Hence, party substitutions are special cases of value substitutions.

In Figure 2.7, we define two types of applications of substitutions to CSL constructs: substitutions of value parameters in (deadline) expressions and clauses, denoted $e[\theta]$, $d[\theta]$, and $c[\theta]$, respectively, where $\theta$ is a substitution; and substitution of party parameters in clauses, denoted $c\langle\theta\rangle$, where $\theta$ is a party substitution. We write $c[v/x]$ for the application on clause $c$ of the substitution that maps $x$ to $v$. Also, $c[\vec{v}/\vec{x}] = c[v_1/x_1] \dots [v_n/x_n]$ for vectors $\vec{v} = (v_1, \dots, v_n)$ and $\vec{x} = (x_1, \dots, x_n)$. Finally, we abuse notation by interpreting vectors of variables as sets in Figure 2.7.

The following lemma shows that the substitutions defined in Figure 2.7 fulfil the expected properties with respect to the type system. Moreover, party parameters are typed using *relevance typing* [90], that is parametrised parties are used at least once in the body of a template definition.

**Lemma 2.3.3.** *Consider a well-typed expression $\Gamma \vdash e : t$, a well-typed deadline expression $\Gamma \vdash d : \text{Deadline}$, and a well-typed clause $\Delta, \Lambda, \Gamma \vdash c : \text{Clause}\langle P\rangle$. For any substitution $\theta$ such that $\theta(x) \in [\![\Gamma(x)]\!]$ for all $x \in \text{dom}(\theta) \cap \text{dom}(\Gamma)$, it holds that:*

$$\begin{aligned} \Gamma' &\vdash e[\theta] : t, \\ \Gamma' &\vdash d[\theta] : \text{Deadline}, \\ \Delta, \Lambda, \Gamma' &\vdash c[\theta] : \text{Clause}\langle P\rangle, \end{aligned}$$

*where $\Gamma' = \Gamma|_{\text{dom}(\Gamma)\setminus\text{dom}(\theta)}$. Moreover, for any party substitution $\theta$, it holds that:*

$$\Delta, \Lambda \setminus \text{dom}(\theta), \Gamma \vdash c\langle\theta\rangle : \text{Clause}\langle P \cup \{p \mid \theta(x) = p, x \in \text{dom}(\Lambda) \cap \text{dom}(\theta)\}\rangle.$$

*Proof.* The first typing judgement (that is, $\Gamma' \vdash e[\theta] : t$) follows easily by induction on the typing derivation $\Gamma \vdash e : t$, and the second judgement then follows immediately. The third judgement follows by induction on the typing derivation $\Delta, \Lambda, \Gamma \vdash c : \text{Clause}\langle P\rangle$, and the same goes for the fourth judgement. $\square$

The reduction semantics for well-formed specifications is presented in Figure 2.8. The reduction relation for clauses has the form $D, \tau \vdash c \xrightarrow{\epsilon} \mathbf{c}$, where $D$ is a set of template definitions, $\tau$ is the time of the last update to the contract (initially the starting time), $c$ is the clause to reduce, $\epsilon$ is the event that takes place, and $\mathbf{c}$ is the *residue*. A residue $\mathbf{c}$ is either a clause, representing the remaining obligations, or a breach of contract.

$\boxed{e[\theta]}$

$$\begin{aligned}
x[\theta] &= \begin{cases} \theta(x) & \text{if } x \in \mathrm{dom}(\theta) \\ x & \text{otherwise} \end{cases} \\
v[\theta] &= v \\
(\neg e)[\theta] &= \neg e[\theta] \\
(e_1 \star e_2)[\theta] &= e_1[\theta] \star e_2[\theta] \\
(e_1 \prec e_2)[\theta] &= e_1[\theta] \prec e_2[\theta]
\end{aligned}$$

$\boxed{d[\theta]}$

$$(\textbf{after } e_1 \textbf{ within } e_2)[\theta] = \textbf{after } e_1[\theta] \textbf{ within } e_2[\theta]$$

$\boxed{c[\theta]}$

$$\begin{aligned}
\textbf{fulfilment}[\theta] &= \textbf{fulfilment} \\
\begin{pmatrix} \langle e_1 \rangle\ k(\vec{x}) \textbf{ where } e_2 \textbf{ due } d \\ \textbf{remaining } z \textbf{ then } c \end{pmatrix}[\theta] &= \begin{array}{l} \langle e_1 \rangle\ k(\vec{x}) \textbf{ where } e_2[\theta|_{\mathcal{V}\setminus\vec{x}}] \textbf{ due } d[\theta] \\ \textbf{remaining } z \textbf{ then } c[\theta|_{\mathcal{V}\setminus(\vec{x}\cup\{z\})}] \end{array} \\
\begin{pmatrix} \textbf{if } k(\vec{x}) \textbf{ where } e \textbf{ due } d \\ \textbf{remaining } z \textbf{ then } c_1 \textbf{ else } c_2 \end{pmatrix}[\theta] &= \begin{array}{l} \textbf{if } k(\vec{x}) \textbf{ where } e[\theta|_{\mathcal{V}\setminus\vec{x}}] \textbf{ due } d[\theta] \\ \textbf{remaining } z \textbf{ then } c_1[\theta|_{\mathcal{V}\setminus(\vec{x}\cup\{z\})}] \textbf{ else } c_2[\theta] \end{array} \\
(c_1 \textbf{ and } c_2)[\theta] &= c_1[\theta] \textbf{ and } c_2[\theta] \\
(c_1 \textbf{ or } c_2)[\theta] &= c_1[\theta] \textbf{ or } c_2[\theta] \\
(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2)[\theta] &= \textbf{if } e[\theta] \textbf{ then } c_1[\theta] \textbf{ else } c_2[\theta] \\
f(e_1, \dots, e_n)\langle \vec{e'} \rangle[\theta] &= f(e_1[\theta], \dots, e_n[\theta])\langle \vec{e'} \rangle
\end{aligned}$$

$\boxed{c\langle\theta\rangle}$

$$\begin{aligned}
\textbf{fulfilment}\langle\theta\rangle &= \textbf{fulfilment} \\
\begin{pmatrix} \langle e_1 \rangle\ k(\vec{x}) \textbf{ where } e_2 \textbf{ due } d \\ \textbf{remaining } z \textbf{ then } c \end{pmatrix}\langle\theta\rangle &= \begin{array}{l} \langle e_1[\theta] \rangle\ k(\vec{x}) \textbf{ where } e_2 \textbf{ due } d \\ \textbf{remaining } z \textbf{ then } c\langle\theta\rangle \end{array} \\
\begin{pmatrix} \textbf{if } k(\vec{x}) \textbf{ where } e \textbf{ due } d \\ \textbf{remaining } z \textbf{ then } c_1 \textbf{ else } c_2 \end{pmatrix}\langle\theta\rangle &= \begin{array}{l} \textbf{if } k(\vec{x}) \textbf{ where } e \textbf{ due } d \\ \textbf{remaining } z \textbf{ then } c_1\langle\theta\rangle \textbf{ else } c_2\langle\theta\rangle \end{array} \\
(c_1 \textbf{ and } c_2)\langle\theta\rangle &= c_1\langle\theta\rangle \textbf{ and } c_2\langle\theta\rangle \\
(c_1 \textbf{ or } c_2)\langle\theta\rangle &= c_1\langle\theta\rangle \textbf{ or } c_2\langle\theta\rangle \\
(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2)\langle\theta\rangle &= \textbf{if } e \textbf{ then } c_1\langle\theta\rangle \textbf{ else } c_2\langle\theta\rangle \\
f(\vec{e})\langle e'_1, \dots, e'_n \rangle\langle\theta\rangle &= f(\vec{e})\langle e'_1[\theta], \dots, e'_n[\theta] \rangle
\end{aligned}$$

Figure 2.7: Substitution of value parameters into expressions $e[\theta]$, deadline expressions $d[\theta]$, and clauses $c[\theta]$; and substitution of party parameters into clauses $c\langle\theta\rangle$.

The second, third, and fourth rules describe the three different situations for obligations: (1) either the event fulfils the obligation, and the residue is determined by the continuation clause; or (2) the event does not fulfil the obligation by missing the deadline, in which case a breach of contract takes place; or (3) the event does not fulfil the obligation, but nor does it violate the deadline, so the obligation—with updated deadlines—remains the residue. The three rules for external choice are similar, except that in the second case the residue is determined by the alternative branch of the choice, rather than a breach of contract.

It follows from the operational semantics that a clause can only be breached by missing a deadline, and the time of breach is determined by the deadline itself. However, we need to take into account that deadlines may be negative, in which case

$\boxed{D, \tau \vdash c \xrightarrow{\epsilon} \mathbf{c}}$

$$\frac{}{D, \tau \vdash \textbf{fulfilment} \xrightarrow{\epsilon} \textbf{fulfilment}}$$

$$\frac{e[\vec{v}/\vec{x}] \Downarrow \textbf{true} \qquad d \Downarrow^{\tau} (\tau_1, \tau_2) \qquad \tau_1 \leq \tau' \leq \tau_2}{D, \tau \vdash \langle p \rangle\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c \xrightarrow{(\tau', k(\vec{v}))} c[\vec{v}/\vec{x}, \tau_2 - \tau'/z]}$$

$$\frac{d \Downarrow^{\tau} (\tau_1, \tau_2) \qquad \tau' > \tau_2}{D, \tau \vdash \langle p \rangle\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c \xrightarrow{(\tau', k'(\vec{v}))} (\max(\tau, \tau_2), \{p\})}$$

$$\frac{\begin{array}{l} d \Downarrow^{\tau} (\tau_1, \tau_2) \\ \quad \tau' \leq \tau_2 \qquad \tau' < \tau_1 \text{ or } k' \neq k \text{ or } e[\vec{v}/\vec{x}] \Downarrow \textbf{false} \qquad d' = \textbf{after}\ \tau_1 - \tau'\ \textbf{within}\ \tau_2 - \tau_1 \end{array}}{D, \tau \vdash \begin{array}{l} \langle p \rangle\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c \xrightarrow{(\tau', k'(\vec{v}))} \\ \langle p \rangle\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d'\ \textbf{remaining}\ z\ \textbf{then}\ c \end{array}}$$

$$\frac{e[\vec{v}/\vec{x}] \Downarrow \textbf{true} \qquad d \Downarrow^{\tau} (\tau_1, \tau_2) \qquad \tau_1 \leq \tau' \leq \tau_2}{D, \tau \vdash \textbf{if}\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \xrightarrow{(\tau', k(\vec{v}))} c_1[\vec{v}/\vec{x}, \tau_2 - \tau'/z]}$$

$$\frac{d \Downarrow^{\tau} (\tau_1, \tau_2) \qquad \tau' > \tau_2 \qquad D, \max(\tau, \tau_2) \vdash c_2 \xrightarrow{(\tau', k'(\vec{v}))} \mathbf{c}}{D, \tau \vdash \textbf{if}\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \xrightarrow{(\tau', k'(\vec{v}))} \mathbf{c}}$$

$$\frac{\begin{array}{l} d \Downarrow^{\tau} (\tau_1, \tau_2) \\ \quad \tau' \leq \tau_2 \qquad \tau' < \tau_1 \text{ or } k' \neq k \text{ or } e[\vec{v}/\vec{x}] \Downarrow \textbf{false} \qquad d' = \textbf{after}\ \tau_1 - \tau'\ \textbf{within}\ \tau_2 - \tau_1 \end{array}}{D, \tau \vdash \begin{array}{l} \textbf{if}\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \xrightarrow{(\tau', k'(\vec{v}))} \\ \textbf{if}\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d'\ \textbf{remaining}\ z\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \end{array}}$$

$$\frac{D, \tau \vdash c_1 \xrightarrow{\epsilon} \mathbf{c}_1 \qquad D, \tau \vdash c_2 \xrightarrow{\epsilon} \mathbf{c}_2}{D, \tau \vdash c_1\ \textbf{and}\ c_2 \xrightarrow{\epsilon} \mathbf{c}_1 \owedge \mathbf{c}_2} \qquad \frac{D, \tau \vdash c_1 \xrightarrow{\epsilon} \mathbf{c}_1 \qquad D, \tau \vdash c_2 \xrightarrow{\epsilon} \mathbf{c}_2}{D, \tau \vdash c_1\ \textbf{or}\ c_2 \xrightarrow{\epsilon} \mathbf{c}_1 \ovee \mathbf{c}_2}$$

$$\frac{e \Downarrow \textbf{true} \qquad D, \tau \vdash c_1 \xrightarrow{\epsilon} \mathbf{c}_1}{D, \tau \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \xrightarrow{\epsilon} \mathbf{c}_1} \qquad \frac{e \Downarrow \textbf{false} \qquad D, \tau \vdash c_2 \xrightarrow{\epsilon} \mathbf{c}_2}{D, \tau \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \xrightarrow{\epsilon} \mathbf{c}_2}$$

$$\frac{\vec{e} \Downarrow \vec{v} \qquad (f(\vec{x})\langle\vec{y}\rangle = c) \in D \qquad D, \tau \vdash c[\vec{v}/\vec{x}, \vec{p}/\vec{y}]\langle\vec{p}/\vec{y}\rangle \xrightarrow{\epsilon} \mathbf{c}}{D, \tau \vdash f(\vec{e})\langle\vec{p}\rangle \xrightarrow{\epsilon} \mathbf{c}}$$

$\boxed{s \xrightarrow{\epsilon} \mathbf{s}}$

$$\frac{D, \tau \vdash c \xrightarrow{\epsilon} (\tau', B)}{\textbf{letrec}\ D\ \textbf{in}\ c\ \textbf{starting}\ \tau \xrightarrow{\epsilon} (\tau', B)}$$

$$\frac{D, \tau \vdash c \xrightarrow{\epsilon} c' \qquad \mathrm{ts}(\epsilon) = \tau'}{\textbf{letrec}\ D\ \textbf{in}\ c\ \textbf{starting}\ \tau \xrightarrow{\epsilon} \textbf{letrec}\ D\ \textbf{in}\ c'\ \textbf{starting}\ \tau'}$$

Figure 2.8: Reduction semantics for CSL clauses $c$ and specifications $s$.

we define the time of breach as the time of the last update. Similarly, we need to take negative deadlines into account for external choices. Note that in the rules, clauses are fully instantiated, that is they have no free variables (for the straightforward definition of free variables): the type system guarantees that well-typed clauses are fully instantiated, as we shall see shortly.

The semantics of clause conjunction and clause disjunction use lifted versions of the corresponding verdict compositions, which are defined by:

$$\mathbf{c}_1 \owedge \mathbf{c}_2 = \begin{cases} c_1 \textbf{ and } c_2 & \text{if } \mathbf{c}_1 = c_1 \text{ and } \mathbf{c}_2 = c_2, \\ (\tau_1, B_1) & \text{if } \mathbf{c}_1 = (\tau_1, B_1) \text{ and } \mathbf{c}_2 = c_2, \\ (\tau_2, B_2) & \text{if } \mathbf{c}_2 = (\tau_2, B_2) \text{ and } \mathbf{c}_1 = c_1, \\ (\tau_1, B_1) \wedge (\tau_2, B_2) & \text{if } \mathbf{c}_1 = (\tau_1, B_1) \text{ and } \mathbf{c}_2 = (\tau_2, B_2), \end{cases}$$

and

$$\mathbf{c}_1 \ovee \mathbf{c}_2 = \begin{cases} c_1 \textbf{ or } c_2 & \text{if } \mathbf{c}_1 = c_1 \text{ and } \mathbf{c}_2 = c_2, \\ c_1 & \text{if } \mathbf{c}_1 = c_1 \text{ and } \mathbf{c}_2 = (\tau_2, B_2), \\ c_2 & \text{if } \mathbf{c}_2 = c_2 \text{ and } \mathbf{c}_1 = (\tau_1, B_1), \\ (\tau_1, B_1) \vee (\tau_2, B_2) & \text{if } \mathbf{c}_1 = (\tau_1, B_1) \text{ and } \mathbf{c}_2 = (\tau_2, B_2). \end{cases}$$

The reduction semantics is lifted to specifications $s \xrightarrow{\epsilon} \mathbf{s}$, where the residue $\mathbf{s}$ is either a residual specification or a breach of contract. Note that the time of the last update (that is, event) is recorded in the residual specification.

The following theorem shows that the semantics satisfies *type preservation* [89]. Moreover, the set of parties in the typing of the residual specification may decrease, matching the intuition that parties may become free of obligations during the execution of a contract.

**Theorem 2.3.4.** *Let $s$ be a well-formed specification involving parties $P$ and $s'$ be a specification. If $s \xrightarrow{\epsilon} s'$ then $s'$ is a well-formed specification involving parties $P'$, for some $P' \subseteq P$.*

*Proof.* The proof is presented in Appendix B.1, page 189. The proof is by induction on the typing derivation. □

The following theorem shows that the semantics also satisfies the *progress property* [89], that is well-formed specifications never get stuck.

**Theorem 2.3.5.** *Let $s$ be a well-formed specification with parties $P$ and starting time $\tau_0$. Then for any event $\epsilon$ with $\mathrm{ts}(\epsilon) \geq \tau_0$ there is a unique residue $\mathbf{s}$ such that $s \xrightarrow{\epsilon} \mathbf{s}$. Furthermore, whenever $\mathbf{s} = (\tau, B)$ then $\tau_0 \leq \tau \leq \mathrm{ts}(\epsilon)$ and $B \subseteq P$.*

*Proof.* The proof is presented in Appendix B.1, page 192. The proof is by nested induction on the structure of the immediate unfolding relation and the step derivation. □

### 2.3.5 Mapping CSL Specifications to Contracts

The reduction semantics presented in Section 2.3.4 is event-driven: at the occurrence of an event, a specification reduces to either a breach of contract or a residual

$\boxed{D, \tau \vdash c \downarrow \nu}$

$$\frac{}{D, \tau \vdash \textbf{fulfilment} \downarrow \checkmark}$$

$$\frac{d \Downarrow^{\tau} (\tau_1, \tau_2)}{D, \tau \vdash \langle p \rangle\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c \downarrow (\max(\tau, \tau_2), \{p\})}$$

$$\frac{d \Downarrow^{\tau} (\tau_1, \tau_2) \qquad D, \max(\tau, \tau_2) \vdash c_2 \downarrow \nu_2}{D, \tau \vdash \textbf{if}\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \downarrow \nu_2}$$

$$\frac{e \Downarrow \textbf{true} \qquad D, \tau \vdash c_1 \downarrow \nu_1}{D, \tau \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \downarrow \nu_1} \qquad \frac{e \Downarrow \textbf{false} \qquad D, \tau \vdash c_2 \downarrow \nu_2}{D, \tau \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \downarrow \nu_2}$$

$$\frac{D, \tau \vdash c_1 \downarrow \nu_1 \qquad D, \tau \vdash c_2 \downarrow \nu_2}{D, \tau \vdash c_1\ \textbf{and}\ c_2 \downarrow \nu_1 \wedge \nu_2} \qquad \frac{D, \tau \vdash c_1 \downarrow \nu_1 \qquad D, \tau \vdash c_2 \downarrow \nu_2}{D, \tau \vdash c_1\ \textbf{or}\ c_2 \downarrow \nu_1 \vee \nu_2}$$

$$\frac{\vec{e} \Downarrow \vec{v} \qquad f(\vec{x})\langle \vec{y} \rangle = c \in D \qquad D, \tau \vdash c[\vec{v}/\vec{x}, \vec{p}/\vec{y}]\langle \vec{p}/\vec{y} \rangle \downarrow \nu}{D, \tau \vdash f(\vec{e})\langle \vec{p} \rangle \downarrow \nu}$$

$\boxed{\vdash s \downarrow \nu}$

$$\frac{D, \tau \vdash c \downarrow \nu}{\vdash \textbf{letrec}\ D\ \textbf{in}\ c\ \textbf{starting}\ \tau \downarrow \nu}$$

Figure 2.9: Verdict $\nu$ associated with specification $s$.

specification. However, the absence of events is also significant, because it may imply that the contract execution is considered finished and no more events are produced. In this case a verdict needs to be associated with the residual specification. Formally, we associate the verdict $\nu$ with a specification $s$ if $\vdash s \downarrow \nu$ can be derived using the derivation rules of Figure 2.9. For any well-formed specification $s$, there exists a unique verdict $\nu$ associated with $s$.

We can now associate a verdict with a specification and an event trace by running the specification on the trace: at each step the specification is reduced on the current event, until either a breach occurs or there are no more events, in which case we check if the residual specification is fulfilled according to the relation in Figure 2.9. Formally, the function $[\![s]\!] : \mathsf{Tr}^{\tau_0} \to \mathsf{V}$ where $\tau_0$ is the start time of $s$, is defined on finite traces inductively by:

$$[\![s]\!](\sigma) = \begin{cases} \nu & \text{if } \sigma = \langle\rangle \text{ and } \vdash s \downarrow \nu, \\ (\tau, B) & \text{if } \sigma = \epsilon\sigma' \text{ and } s \xrightarrow{\epsilon} (\tau, B), \\ [\![s']\!](\sigma') & \text{if } \sigma = \epsilon\sigma' \text{ and } s \xrightarrow{\epsilon} s', \end{cases}$$

and on infinite traces by the (unique) extension in Lemma 2.2.5.

The following theorem shows that CSL specifications indeed represent contracts in the sense of Definition 2.2.1.

**Theorem 2.3.6.** *Let $s$ be a well-formed specification with parties $P$ and start time $\tau_0$. Then $[\![s]\!]$ is a contract between parties $P$ starting at time $\tau_0$.*

*Proof.* The proof is presented in Appendix B.1, page 194. The proof follows by induction on the length of the trace using Theorems 2.3.4 and 2.3.5. □

**Corollary 2.3.7.** *Let $s =$ **letrec** $D$ **in** $c$ **starting** $\tau$ be a well-formed specification. Then:*

$$[\![s]\!] = \begin{cases} \mathsf{c}_{\checkmark} & \text{if } c = \textbf{fulfilment}, \\ [\![s_1]\!] \wedge [\![s_2]\!] & \text{if } c = c_1 \textbf{ and } c_2, \\ [\![s_1]\!] \vee [\![s_2]\!] & \text{if } c = c_1 \textbf{ or } c_2, \end{cases}$$

*where $s_i =$ **letrec** $D$ **in** $c_i$ **starting** $\tau$.*

*Proof.* For finite traces the proofs follow by induction on the trace length, similar to the proof of Theorem 2.3.6. For infinite traces the results then follow from the uniqueness result of Lemma 2.2.5. □

The theorem and its corollary show that CSL enjoys the principles underpinning the contract model defined in Section 2.2, that is deterministic blame assignment and compositionality. Moreover, the algebraic properties stated in Corollary 2.2.15 carry over to CSL.

### 2.3.6 Monitoring CSL Specifications

The reduction semantics presented above gives rise to an incremental run-time monitoring algorithm for CSL specifications. The main ingredient of the monitor is the function $\text{mon} : \mathsf{S} \times \mathsf{Tr}^{\tau_0}_{\text{fin}} \to (\mathsf{V}_! \cup \mathsf{V}_?) \times \mathsf{S}$ defined by:

$$\text{mon}(s, \sigma) = \begin{cases} (\nu_?, s) & \text{if } \sigma = \langle\rangle \text{ and } \vdash s \downarrow \nu, \\ (\nu_!, s') & \text{if } \sigma = \sigma'\epsilon \text{ and } \text{mon}(s, \sigma') = (\nu_!, s'), \\ ((\tau, B)_!, s') & \text{if } \sigma = \sigma'\epsilon \text{ and } \text{mon}(s, \sigma') = (\nu_?, s') \\ & \qquad\qquad \text{and } s' \xrightarrow{\epsilon} (\tau, B), \\ (\nu_?, s'') & \text{if } \sigma = \sigma'\epsilon \text{ and } \text{mon}(s, \sigma') = (\nu'_?, s') \\ & \qquad\qquad \text{and } s' \xrightarrow{\epsilon} s'' \text{ and } \vdash s'' \downarrow \nu, \end{cases}$$

where $\mathsf{S}$ is the set of all well-formed CSL specifications.

The monitor is invoked whenever an event occurs, provided that the monitor has not already output a final verdict. Between invocations, it only needs to remember the previous result, that is in order to process the event $\epsilon$, after the events $\sigma$ have happened, we only need the previous result $\text{mon}(s, \sigma)$ in order to compute the new result $\text{mon}(s, \sigma\epsilon)$.

The function mon is not a run-time monitor in the sense of Definition 2.2.16. However, it is very close to one, as shown by the following theorem, which follows directly from Theorem 2.3.6.

**Theorem 2.3.8.** *Let $s$ be a specification with starting time $\tau_0$. The function* mon *is computable and for any trace $\sigma \in \mathsf{Tr}^{\tau_0}_{\text{fin}}$, verdict $\nu_\star$, and residual specification $s'$, with $\text{mon}(s, \sigma) = (\nu_\star, s')$, it holds that*

*(1) if $\nu_\star = (\tau, B)_!$ then $[\![s]\!](\sigma') = (\tau, B)$ for all $\sigma'$ with $\sigma \sqsubset \sigma'$,*

*(2) if $\nu_\star = \checkmark_?$ then $[\![s]\!](\sigma) = \checkmark$, and*

*(3) if $\nu_\star = (\tau, B)_?$ then $[\![s]\!](\sigma) = (\tau, B)$ and $\tau \geq \text{end}(\sigma)$.*

The result above shows that our run-time monitor satisfies impartiality (1), however it does not always satisfy anticipation. For instance, if the body of a specification is **fulfilment**, then our monitor always outputs $\checkmark_?$, even if anticipation requires that it outputs $\checkmark_!$. Building a run-time monitor that guarantees anticipation is hard, because the expression language can "hide" anticipated verdicts. Consider for instance the clauses:

$$\begin{aligned}
c_1 &= \langle p\rangle\ k(x)\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c,\\
c_2 &= \textbf{if}\ k(x)\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c\ \textbf{else fulfilment},
\end{aligned}$$

where $e$ is some expression for which $e[v/x] \Downarrow \textbf{false}$ for all values $v$, for instance $e = x > 0 \land x < 0$. The contract represented by $c_1$ is always breached, while the one represented by $c_2$ is never breached. Hence, in order to guarantee anticipation, we first need to decide satisfiability for the expression language.

**Example 2.3.9.** We demonstrate the reduction semantics and run-time monitor using the CSL specification in Figure 2.3. As in Example 2.2.17, we consider the trace $\langle \epsilon_1, \epsilon_3, \epsilon_4\rangle$, where the events are as in the example, except that they use concrete actions instead of abstract actions:

$$\begin{aligned}
\epsilon_1 &= (\text{2011-01-01}, \text{Delivery(Seller, Buyer, ``Laser printer'')}),\\
\epsilon_3 &= (\text{2011-01-01}, \text{Payment(Buyer, Seller, 100)}),\\
\epsilon_4 &= (\text{2011-01-10}, \text{Payment(Buyer, Seller, 100)}).
\end{aligned}$$

We first define the specifications $s_i$, with $i \in \{0, 1, 2, 3\}$:

$$\begin{aligned}
s_i \ = \ & \textbf{letrec}\ \mathit{sale}(\mathit{deliveryDeadline}, \mathit{goods}, \mathit{payment})\langle \mathit{buyer}, \mathit{seller}\rangle = c\\
& \textbf{in}\ c_i[\theta]\ \textbf{starting}\ \text{2011-01-01}
\end{aligned}$$

where $\theta(\mathit{deliveryDeadline}) = 0$, $\theta(\mathit{goods}) =$ "Laser printer", $\theta(\mathit{payment}) = 200$, $\theta(\mathit{buyer}) = \text{Buyer}$, $\theta(\mathit{seller}) = \text{Seller}$, and

$$\begin{aligned}
c_0 = {} & \mathit{sale}(0, \text{``Laser printer''}, 200)\langle \text{Buyer, Seller}\rangle\\
c \ = {} & \langle \mathit{seller}\rangle\ \text{Delivery}(s,r,g)\\
& \quad \textbf{where}\ s = \mathit{seller} \land r = \mathit{buyer} \land g = \mathit{goods}\ \textbf{due within}\ \mathit{deliveryDeadline}\\
& \quad \textbf{then}\ c_1\\
c_1 = {} & \langle \mathit{buyer}\rangle\ \text{Payment}(s,r,a)\\
& \quad \textbf{where}\ s = \mathit{buyer} \land r = \mathit{seller} \land a = \mathit{payment}\ /\ 2\ \textbf{due immediately}\\
& \quad \textbf{then}\ c_2\\
c_2 = {} & (\langle \mathit{buyer}\rangle\ \text{Payment}(s,r,a)\\
& \quad \textbf{where}\ s = \mathit{buyer} \land r = \mathit{seller} \land a = \mathit{payment}\ /\ 2\ \textbf{due within}\ \text{30D}\\
& \quad \textbf{or}\\
& \quad \langle \mathit{buyer}\rangle\ \text{Payment}(s,r,a)\\
& \quad \textbf{where}\ s = \mathit{buyer} \land r = \mathit{seller} \land a = (\mathit{payment} \times 110)\ /\ 200\\
& \quad \textbf{due within}\ \text{14D}\ \textbf{after}\ \text{30D})\\
& \quad \textbf{and}
\end{aligned}$$

```
        if Return($s$,$r$,$g$)
            where $s = \mathit{buyer} \wedge r = \mathit{seller} \wedge g = \mathit{goods}$ due within 14D
        then
        $\langle \mathit{seller} \rangle$ Payment($s$,$r$,$a$)
         where $s = \mathit{seller} \wedge r = \mathit{buyer} \wedge a = \mathit{payment}$ due within 7D
$c_3$ = if Return($s$,$r$,$g$)
         where $s = \mathit{buyer} \wedge r = \mathit{seller} \wedge g = \mathit{goods}$ due after $-9$D within 14D
        then
        $\langle \mathit{seller} \rangle$ Payment($s$,$r$,$a$)
         where $s = \mathit{seller} \wedge r = \mathit{buyer} \wedge a = \mathit{payment}$ due within 7D
```

The specification in Figure 2.3 equals $s_0$. We have $s_0 \xrightarrow{\epsilon_1} s_1 \xrightarrow{\epsilon_3} s_2 \xrightarrow{\epsilon_4} s_3$. Note that the relative deadline in $c_3$ for returning the goods is shifted with regard to the corresponding relative deadline in $c_2$, due to the passing of time. The incremental output of the monitor on the trace $\langle \epsilon_1, \epsilon_3, \epsilon_4 \rangle$ is as follows:

$$\begin{aligned}
\mathrm{mon}(s_0, \langle\rangle) &= ((\text{2011-01-01}, \{\text{Seller}\})_?, s_0),\\
\mathrm{mon}(s_0, \langle \epsilon_1 \rangle) &= ((\text{2011-01-01}, \{\text{Buyer}\})_?, s_1),\\
\mathrm{mon}(s_0, \langle \epsilon_1, \epsilon_3 \rangle) &= ((\text{2011-02-14}, \{\text{Buyer}\})_?, s_2),\\
\mathrm{mon}(s_0, \langle \epsilon_1, \epsilon_3, \epsilon_4 \rangle) &= (\checkmark_?, s_3).
\end{aligned}$$

Finally, remark that on all traces, except the last one, the value of mon coincides with the value of the run-time monitor of Example 2.2.17.

### 2.3.7 Contract Examples

We have seen one example of a realistic contract specified in CSL, namely the sales contract in Figure 2.1. The example illustrates how dependencies between paragraphs are realised as continuation clauses, how obligations and permissions are represented, and how contract disjunction enables choices. In this section we provide further specification examples, which illustrate prohibitions, potentially infinite contracts, linear treatment of events (as in linear logic [32]), and a more involved application of arithmetic expressions.

**Example 2.3.10.** Prohibitions are not built-in to CSL, yet it is possible to express prohibitions using external choices and obligations. Consider the *non-disclosure agreement* in Figure 2.10 (top). The agreement is formalised in Figure 2.10 (bottom), using a signature that includes the action kinds {Disclosure, Unfulfillable} $\subseteq \mathcal{K}$, with types ar(Disclosure) = ⟨Party⟩ and ar(Unfulfillable) = ⟨⟩. We use the action kind Unfulfillable to point out that the corresponding obligation cannot be fulfilled.

Besides the technique for encoding prohibitions, the example illustrates an important point, namely that we do not model how parties agree that events have taken place. In the agreement above, a dispute is more likely to involve proving (or disproving) disclosure of information, rather than interpreting whether disclosing information is allowed or not.

**Example 2.3.11.** The next example is a lease agreement presented in Figure 2.11 (top). The contract is formalised in Figure 2.11 (bottom), using a signature that

**Paragraph 1.** The following agreement is enacted on 2011-01-01, and is valid for 5 years.

**Paragraph 2.** The Employee agrees not to disclose any information regarding the work carried out under the Employer, as stipulated in Paragraph 3.

**Paragraph 3.** (Omitted.)

– ◇ –

```
letrec $nda()\langle employee\rangle$ =
 if Disclosure($e$)
     where $e = employee$ due within 5Y
 then
 $\langle employee\rangle$ Unfulfillable where false due immediately
in
$nda()\langle$Employee$\rangle$ starting 2011-01-01
```

Figure 2.10: A non-disclosure agreement (paper version top, CSL version bottom).

**Paragraph 1.** The term of this lease is for 6 months, beginning on 2011-01-01. At the expiration of said term, the lease will automatically be renewed for a period of one month unless either party (Landlord or Tenant) notifies the other of its intention to terminate the lease at least one month before its expiration date.

**Paragraph 2.** The lease is for 1 apartment, which is provided by Landlord throughout the term.

**Paragraph 3.** Tenant agrees to pay the amount of €1000 per month, each payment due on the 7th day of each month.

– ◇ –

```
letrec $lease(property, leaseStart, leasePeriod, leasePeriods, payment, payDeadline,$
              $terminationRequested)\langle lessor, lessee\rangle$ =
 if $leasePeriods \leq 0 \wedge terminationRequested$ then
   fulfilment
 else
   $\langle lessee\rangle$ Payment($s$,$r$,$a$)
    where $s = lessee \wedge r = lessor \wedge a = payment$
    due immediately after $leaseStart + payDeadline$
   and
   $\langle lessor\rangle$ Provision($s$,$r$,$p$,$l$)
    where $s = lessor \wedge r = lessee \wedge p = property \wedge l = leasePeriod$
    due immediately after $leaseStart$
   then
   if $terminationRequested$ then
     $lease(property, leasePeriod, leasePeriod, leasePeriods - 1, payment,$
           $payDeadline,$ true$)\langle lessor, lessee\rangle$
   else
   if ReqTermination($s$)
       where $s = lessor \vee s = lessee$ due within $leasePeriod$ remaining $z$
   then
     $lease(property, z, leasePeriod, min(1, leasePeriods - 1), payment,$
           $payDeadline,$ true$)\langle lessor, lessee\rangle$
   else
     $lease(property, 0, leasePeriod, leasePeriods - 1, payment, payDeadline,$ false$)\langle lessor, lessee\rangle$
in
$lease($"Apartment", 0, 1M, 6, 1000, 7D, false$)\langle$Landlord, Tenant$\rangle$ starting 2011-01-01
```

Figure 2.11: A lease agreement (paper version top, CSL version bottom).

**Paragraph 1.** The master agreement between Vendor and Customer is for 1000 printers, with a unit price of €100. The agreement is valid for one year, starting 2011-01-01.

**Paragraph 2.** The customer may at any time order an amount of printers (with the total not exceeding the threshold of 1000), after which the Vendor must deliver the goods before the maximum of (i) 14 days, or (ii) the number of ordered goods divided by ten days.

**Paragraph 3.** After delivering the goods, Vendor may invoice the Customer within 1 month, after which the goods must be paid for by Customer within 14 days.

– ◇ –

```
letrec master(goods, amount, terminationDeadline, payment, invoiceDeadline,
              paymentDeadline, id)⟨vendor, customer⟩ =
 if amount = 0 then
   fulfilment
 else
 if Request(s,r,n,g)
     where s = customer ∧ r = vendor ∧ n ≤ amount ∧ n > 0 ∧ g = goods
     due within terminationDeadline remaining z
 then
   sale(n, g, n × payment, max(14D, n × 24 × 60 × 6),
        invoiceDeadline, paymentDeadline, id)⟨vendor, customer⟩
   and
   master(goods, amount − n, z, payment,
          invoiceDeadline, paymentDeadline, id + 1)⟨vendor, customer⟩

sale(number, goods, payment, deliveryDeadline, invoiceDeadline, paymentDeadline, id)
    ⟨seller, buyer⟩ =
⟨seller⟩ Delivery(s,r,n,g,i)
  where s = seller ∧ r = buyer ∧ n = number ∧ g = goods ∧ i = id due within deliveryDeadline
 then
 if IssueInvoice(s,r,i)
     where s = seller ∧ r = buyer ∧ i = id due within invoiceDeadline
 then
   ⟨buyer⟩ Payment(s,r,a,i)
    where s = buyer ∧ r = seller ∧ a = payment ∧ i = id due within paymentDeadline
in
master("Printer", 1000, 1Y, 100, 1M, 14D, 0)⟨Vendor, Customer⟩ starting 2011-01-01
```

Figure 2.12: Master sales agreement (paper version top, CSL version bottom).

includes the action kinds $\{\text{Payment}, \text{ReqTermination}, \text{Provision}\} \subseteq \mathcal{K}$, with associated types $\text{ar}(\text{Payment}) = \langle \text{Party}, \text{Party}, \text{Int} \rangle$, $\text{ar}(\text{ReqTermination}) = \langle \text{Party} \rangle$, and $\text{ar}(\text{Provision}) = \langle \text{Party}, \text{Party}, \text{String}, \text{Int} \rangle$. We assume that the expression language has been extended with a function for calculating the minimum of two integers.

The example demonstrates how recursive template definitions enable potentially infinite contracts: each lease period is guaranteed to be executed at least 6 times, but there is no a priori upper bound on the number of iterations. The example also illustrates the usage of the **remaining** construct, which is needed in order to determine the start of the next lease period, when a party requests termination.

**Example 2.3.12.** Next we consider a master sales agreement in Figure 2.12 (top). The contract is formalised in Figure 2.12 (bottom), using a signature that includes the action kinds $\{\text{Request}, \text{IssueInvoice}, \text{Delivery}, \text{Payment}\} \subseteq \mathcal{K}$, with types

**Paragraph 1.** Buyer agrees to pay to Seller the total sum €10000, in the manner following:

**Paragraph 2.** €500 is to be paid at closing, and the remaining balance of €9500 shall be paid as follows:

**Paragraph 3.** €500 or more per month on the first day of each and every month, and continuing until the entire balance, including both principal and interest, shall be paid in full; provided, however, that the entire balance due plus accrued interest and any other amounts due here-under shall be paid in full on or before 24 months.

**Paragraph 4.** Monthly payments shall include both principal and interest with interest at the rate of 10%, computed monthly on the remaining balance from time to time unpaid.

– ◇ –

```
letrec instalments(balance, instalment, payDeadline, start, end, frequency,
                   rate, closingPayment, seller)⟨buyer⟩ =
 if balance ≤ 0 then
   ⟨buyer⟩ Payment(s,r,a)
    where s = buyer ∧ r = seller ∧ a = closingPayment due within end
 else
 if end ≤ start then
   ⟨buyer⟩ Payment(s,r,a)
    where s = buyer ∧ r = seller ∧ a = balance + closingPayment due within end
 else
   ⟨buyer⟩ Payment(s,r,a)
    where s = buyer ∧ r = seller ∧ a ≥ min(balance,instalment) ∧ a ≤ balance
    due within payDeadline after start remaining z
   then
   instalments(((100 + rate) × (balance − a)) / 100, instalment, payDeadline,
               frequency − payDeadline + z, end − start − payDeadline + z,
               frequency, rate, closingPayment, seller)⟨buyer⟩
in
instalments(10000, 500, 1D, 0, 24M, 1M, 10, 500, Seller)⟨Buyer⟩ starting 2011-01-01
```

Figure 2.13: Instalment sale (paper version top, CSL version bottom).

ar(Request) = ⟨Party, Party, Int, String⟩, ar(IssueInvoice) = ⟨Party, Party, Int⟩, ar(Delivery) = ⟨Party, Party, Int, String, Int⟩, and ar(Payment) = ⟨Party, Party, Int, Int⟩. We assume that the expression language has been extended with a function for calculating the maximum of two integers.

The encoding illustrates the usage of multiple template definitions and that deadlines can be calculated dynamically based on previous events. Moreover, the action kinds pertaining to each individual sale contain identifiers that are needed in order to distinguish potentially identical payments, deliveries, or invoices when there are simultaneous orders.

**Example 2.3.13.** The last contract we consider is an instalment sale in Figure 2.13 (top). For simplicity, we have only included the payment part of the contract, and not Seller's obligation to deliver goods. The CSL formalisation is presented in Figure 2.13 (bottom), and it shows a more involved application of in-place arithmetic expressions, namely calculation of the remaining balance after each instalment has been payed. Note that contract termination not only depends on the initial 24 months period, but that the contract may end earlier, in case the remaining balance is fully payed.

## 2.4 Related Work

Formal specification of contracts and automatic reasoning about contracts has drawn interest from a wide variety of research areas within computer science, going back to the late eighties with the pioneering work by Lee [58]. Contract formalisms typically fall into three categories: (deontic) logic based formalisms [35, 58, 93], event-condition-action based formalisms [33, 60], and trace based formalisms [6, 57]. The logic based approaches mainly focus on declarative specification of contracts, and on (meta) reasoning, such as decidability of the logic. On the other hand, the event-condition-action and trace based models focus mainly on contract execution. The latter approach takes a more extensional view of contracts, that is contracts are denoted by the set of traces they accept. Other approaches to contract modelling include combinator libraries [86], defeasible reasoning [34, 36, 105], commitment graphs, that is graph theoretic representations of responsibility between parties [121, 122], finite state machines [74], and more informal frameworks [17, 21, 80, 117]. Common to all approaches is the goal of modelling (electronic) contracts in general, except for Peyton-Jones and Eber [86], Andersen et al. [6], and Tan and Thoen [105] who specifically consider financial contracts, commercial contracts, and trade contracts, respectively.

Existing contract frameworks tend to focus either on contract execution models [74, 80, 121, 122], or on concrete specification languages [6, 17, 21, 33, 35, 36, 58, 86, 93, 105], rather than considering both an abstract semantic model and a specification language. Consequently, these frameworks either lack a language for specifying contracts, or they lack an operational interpretation—with the exception of [6, 93], who however do not characterise contracts abstractly in terms of their semantic models. In contrast, we consider both an abstract execution model and a specification language. Besides giving a formal operational interpretation to specifications, this makes it possible to consider different specification languages for different contract domains, and still compare their semantics in terms of the abstract model. Moreover, by mapping a specification language into our model, deterministic blame assignment is guaranteed, algebraic properties of conjunction and disjunction follow automatically, and run-time monitoring has a well-defined meaning.

Compared with the previous contract execution models [74, 80, 121, 122], our abstract contract model relies on fewer high-level concepts. For instance, the existing models rely on concepts such as deadlines [121, 122], deontic modalities [74] and logical formulae [80], which are all definable within our model.

Compared with the previous contract specification languages [6, 17, 21, 33, 35, 36, 58, 86, 93, 105], ours mainly distinguishes itself by incorporating deterministic blame assignment. Besides, existing languages all fall short of other important features. History sensitive commitments, that is commitments which depend on what has happened in the past, are only supported in few languages [6, 35]. History sensitivity is typically not supported because actions are modelled as propositional variables, hence actions cannot carry values. Only the language of Andersen et al. [6] has support for (recursive) contract templates; we have adapted their construction to CSL. Furthermore, potentially infinite contracts are only supported in few languages [6, 58, 93]. Finally, some languages lack absolute temporal constraints [34, 36, 93], and instead consider only relative temporal constraints.

The importance of monitoring contracts is widely recognised [6, 33, 35, 74, 93, 121, 122], yet few authors provide a formal, operational semantics for contract execution [6, 93]. Such a semantics is a prerequisite for showing that a monitor achieves its goals. Furthermore, deterministic blame assignment is crucial for run-time monitoring, a feature which—to the best of our knowledge—has only previously been recognised by Xu and Jeusfeld [122]. However, Xu and Jeusfeld only consider monitoring and blame assignment for their particular specification language, while we also define these notions in a general and abstract setting.

Compositional specification of contracts is traditionally obtained by means of conjunction and disjunction [6, 35, 86, 93]. Besides, Andersen et al. [6] present a language that supports linear conjunction [32]. Despite the fact that compositionality of contracts has previously been considered, there has been no previous treatment of the effect of compositionality on blame assignment, and in particular on how disjunctions involving different parties may give rise to nondeterminism.

Standard deontic logic (SDL) [111]—the logic of obligations, permissions, and prohibitions—has inspired existing contract formalisms [35, 58, 93] due to the appealing similarities with concepts from contracts. Yet the *possible worlds* semantics [120] of deontic logic lacks an operational interpretation, which in our view makes SDL inappropriate as a basis for formalising contracts. To alleviate this weakness, Prisacariu and Schneider [93] consider a restricted form of deontic modalities with *ought-to-do* rather than *ought-to-be*, meaning that deontic modalities are only to specify what should happen ("Seller ought to deliver"), and not what should be the general state of affairs ("it ought to be the case that Seller delivers"). The restriction to ought-to-do statements gives rise to an alternative $\mu$-calculus semantics based on actions. We also restrict contracts to ought-to-do statements.

It has been argued that contrary-to-duty obligations [92]—also a SDL related concept—are crucial for contracts as well [17, 35, 80, 93]. Although we recognise the importance of reparation activities in contracts, we instead consider them ordinary choices, rather than choices with an implicit agreement to conform first and foremost with primary objectives. In consideration hereof, we avoid the philosophical considerations of contrary-to-duty [35, 92], and the treatment of intermediate violations generated by failing to comply with primary objectives.

## 2.5 Conclusion

In this article we have presented a novel, trace-based model for multiparty contracts with blame assignment. We have illustrated that high-level contract concepts such as obligations, deadlines, and reparation clauses are representable within our model. This shows that our model is well-suited for representing real-world contracts. For the purpose of writing contracts, we have given a contract specification language, which enjoys the principle of blame assignment by inheritance from the abstract model, and which is amenable to incremental run-time monitoring.

We plan to use CSL in case studies to further evaluate its applicability for formalising contracts and monitoring their executions. Here, we expect that the expression language of CSL needs to be extended, while hopefully the clause language does not require additions. The extensions to the expression language should be straightforward.

A restriction in our model is that blame is deterministically assigned to contract parties in case of breach of contract. Although deterministic blame assignment is a desired feature, not all real-world contracts have this feature. In future work, we plan to extend our model such that verdicts can be nondeterministically associated with traces. Such an extension is also motivated by the objective for obtaining less restrictive operators for composing contracts.

Future work also includes contract analysis. Such an analysis can be based on our abstract contract model or on the reduction semantics of CSL. For instance, an immediately implementable online analysis based on the reduction semantics is to simulate the outcome of possible future events. Together with the information on who is responsible for an event, this is useful to avoid a breach of contract and to issue reminders of deadlines. The monitoring algorithm partly does this already by outputting potential breaches that represent upcoming deadlines. A further goal of such an online analysis is to monitor contract execution with full anticipation. However, in order to effectively perform such monitoring of CSL specifications, it may be necessary to restrict oneself to fragments of CSL. Other contract analyses are (1) satisfiability, that is whether a contract can be fulfilled at all, (2) satisfiability with respect to a particular party, that is whether a party can avoid breaching a contract in which it is involved, (3) contract valuation, that is what is the expected value of a contract for a given party, and (4) contract entailment, that is whether fulfilling a contract entails the fulfilment of another contract. The last analysis has applications for instance in checking contract conformance with regulations, when regulations are themselves formalised as contracts.

**Acknowledgements** The authors thank the participants of the FLACOS 2010 workshop for providing valuable input to an early state of this work. The first author would also like to thank David Basin for the invitation to visit his research group at ETH Zürich in the first half of 2010, during which foundations of the work presented in this article were laid.

# Chapter 3

# Foundations for Distributed Programming-by-Contract[⋆]

**Abstract**

Programming-by-contract (PBC) is a well-established paradigm for specifying and verifying sequential, one-machine programs in a modular fashion. Traditional PBC is characterised by absolute conformance of code to its specification, propagating blame in case of failures, and a hierarchical, cooperative decomposition model—none of which extend naturally to a distributed environment with multiple administrative peers. We consequently propose a fundamentally new theory of PBC for concurrent and distributed environments. Our theory is based on quantifiable performance of implementations; assuming responsibility for success; and an adversarial model of system integration, where each component provider is optimising its behaviour locally, with respect to potentially conflicting demands. Our model gives rise to a game-theoretic formulation of contract-governed process interaction, and contract conformance is defined with respect to a set of contracts—a contract portfolio—in order to properly account for the possibility of delegation to subcontractors. We show that our definition of contract conformance permits compositional reasoning in the vein of traditional programming-by-contract.

## 3.1 Introduction

Programming-by-contract (PBC)—or, design-by-contract [69]—is a paradigm for specifying and verifying computer programs, typically by means of formal preconditions and postconditions for code fragments [43]. Given a program component $c$, a precondition $A$ is a predicate over $c$'s inputs specifying the requirements or assumptions made by $c$. If, for instance, $c$ computes a function on numbers, $A$ may be the requirement that input $x$ satisfies $x \geq 0$. Likewise, a postcondition $B$ is a predicate over both inputs and outputs of $c$, specifying $c$'s guarantees about its outputs, for the given inputs. In the example, $B$ may for instance specify that the output number $r$ must satisfy $r^2 = x$. A piece of code that satisfies its contract,

[⋆] *Revised version of the chapter "Foundations for Programming By Contract in a Concurrent and Distributed Environment" [49], which in turn is based on joint work with Anders Starcke Henriksen and Andrzej Filinski [42].*

even in an inefficient or unexpected way such as returning $r = -\sqrt{x}$, is then said to be *correct*.

The purpose of PBC is to enable modular design and implementation of programs, by establishing detailed contracts for all module interfaces, in such a way that correctness of the whole program (that is, top-level module) follows from the correctness of all the component modules. In particular, any failure of the whole program to satisfy its contract can ultimately be attributed to a violation of a specific precondition or postcondition. The former occurs when a caller fails to satisfy the input requirements for invoking a submodule, while the latter indicates the failure of the callee to satisfy its output guarantee. In both cases, the implementor of the faulty module is the one who is blamed, and the module has to be corrected. For this reason, PBC has also been dubbed "The Blame Game" [115], due to the (somewhat degenerate) game-theoretic nature of each implementor's incentive being to avoid getting blamed.

The compositional nature of PBC means that the implementor of a module need not be aware of the entire context in which the module is used. Preconditions and postconditions define exactly what the module and its context can expect from each other, hence when implementing the module nothing else can—or should—be assumed about the environment, and vice versa. For instance, in the numeric example above, it would be wrong of the environment to use the output of $c$ directly as the input for a second invocation of $c$, even though this latent bug would go undetected if $c$ simply returned the positive square root $r = \sqrt{x}$.

The goal of our work is to extend PBC from a classical one-machine setup to a setting where programs run concurrently on potentially different machines, owned by different administrative peers. Compositionality is a crucial feature in PBC, and it becomes ever more important in a distributed computation model, in which knowledge of the entire context is not realistic. Existing work on extending precondition- and postcondition-style contracts to a concurrent setting have been proposed [47, 81], but to our knowledge there exist no extensions of the PBC paradigm to a distributed environment.

### 3.1.1 Distributed Programming-by-Contract

Extending PBC to a concurrent, message-passing setting is, in principle, relatively straightforward. The evident difference from a sequential setting is that preconditions and postconditions must be generalised from one-shot input–output contracts to communication-protocol contracts. That is, input requirements now specify what may legitimately be sent to a concurrent module or process, while output guarantees capture what must in turn be sent by that process. Moreover, both input contracts and output contracts may now in general refer to the entire communication history. Still, no fundamental, conceptual changes to the PBC paradigm seem necessary. The notion of session types [16, 45] is an example of such communication-protocol contracts.

On the other hand, a proper account of realistic distributed systems does seem to require a complete reassessment of some basic assumptions of PBC. The problematic characteristics here are the notions of *absolute conformance*, *blame propagation*, and the ultimately *cooperative* model of system development.

**Absolute conformance** By absolute conformance we mean that once a module violates its contract, the "world stops" and no formal guarantees can be given about possible continued execution. The erroneous module must be repaired before the program can be reliably resumed or restarted. In a large-scale distributed environment, however, failures are a fact of life, and though we do need to assign blame for them, the model must also include a robust specification of the relevant recovery procedures, and how any further failures by both parties are to be accounted for.

Moreover, not all failures are equally serious, and some might even be expected to occur in the course of a typical interaction sequence. We thus prefer a contract conformance model based not on a binary outcome, but on a quantitative measure, making it possible to also uniformly express performance characteristics, such as responsiveness, and relative importance of potentially conflicting contracts. Using an economic metaphor, a module (or rather, the module's implementor) is rewarded by its environment for "good" behaviour, and penalised for "bad" behaviour. We can recover the usual absolute notion of contract satisfaction as a requirement that a *correct* module's accumulated balance is always non-negative. Thus, in particular, such a module will never be the first to break a communication contract. But the key motivation for quantifiable performance contracts is that they support compositional reasoning about module correctness in systems with more than two module producers, as sketched next.

**Blame propagation** The second problem with traditional PBC is that simple blame propagation is not by itself a proper foundation for composing distributed modules. As an example, consider a service provider $\mathsf{P}$ that implements a web service $W_1$ using a gateway service $W_2$ provided by a subcontractor $\mathsf{S}$. If $\mathsf{P}$ provides $W_1$ as a service to some client $\mathsf{C}$, then $\mathsf{P}$ cannot rely on propagating blame to $\mathsf{S}$ if $W_1$ fails as a result of $W_2$ failing first. In other words, $\mathsf{P}$ is still *responsible* to $\mathsf{C}$ for the correct behaviour of $W_1$ and cannot be excused by $W_2$ failing. However, $\mathsf{S}$ is in turn responsible to $\mathsf{P}$ for $W_2$, which may imply that $\mathsf{S}$ has to pay a fine to $\mathsf{P}$ each time $W_2$ fails. If $\mathsf{P}$ is properly organised, this fine will be sufficient to cover the fine that $\mathsf{P}$ has to pay to $\mathsf{C}$.

Again, traditional PBC blame propagation can be seen as a degenerate instance of the responsibility model. The difference is that assuming responsibility for satisfying a contract in general involves more than merely being able to deflect all blame for failure. It is an inherently more robust notion, because it requires a module offering a service to explicitly plan for any or all of its subcontractors not fulfilling their nominal ("happy-path") contracts, and ensuring that any penalties imposed by the service's client can ultimately be recovered from the subservices. In particular, a correct module will never make a high-assurance service (that is, with a high penalty for failure) rely on a low-assurance one.

The final problem with blame propagation is that it is not compositional: in blame propagation all modules of the program must be known, since blame can be propagated to any of the modules. In the example above this implies that $\mathsf{C}$ has to know $\mathsf{S}$, since $\mathsf{P}$ may propagate blame onto $\mathsf{S}$ in case of failure. And $\mathsf{S}$ may in turn propagate blame onto its (sub)subcontractors, which means that the entire network must be known. This problem is avoided by requiring that $\mathsf{P}$ must assume responsibility for $\mathsf{S}$ when servicing $\mathsf{C}$.

**Cooperation** By a cooperative decomposition model in traditional PBC we mean that all the module implementors are—to some degree—ultimately working towards a single goal of building a correct system. Thus, if a module contract is ambiguous or incomplete, the implementor will typically still opt to implement the module in the "intended" way, even if he does not explicitly stand to gain anything from it. In particular, he would normally not choose a deliberately suboptimal algorithm for solving a problem, nor intentionally cause minor failures—even if such failures are nominally allowed by the performance contract. For instance, a contract may stipulate that providing a wrong answer to a question is unacceptable, but explicitly declining to answer is a legitimate response—yet simply uniformly refusing to answer would be considered a "bad-faith" implementation.

In a distributed setting, the implicit assumption of cooperation is not necessarily justified. In the web-service example above it may be *locally optimal* for the subcontractor S to sometimes have $W_2$ failing (and taking the penalty), if that for instance enables S to respond to a request from some other (higher-payoff) company O that neither the main contractor P nor the ultimate client C know or care about. Thus, all parties in a distributed system need to recognise that their PBC communication peers may occasionally—or even consistently—violate contracts, not due to coding errors or legitimate misinterpretation of the contract, but as a deliberate design choice. The adversarial nature also prompts the need for absolute guarantees in contracts rather than unenforceable promises. That is, commitments must be specified with absolute deadlines such as "answer within 10 seconds", rather than "answer eventually".

**Game-theoretic model** In summary of the observations above, we propose an explicitly game-theoretic [68] extension of the PBC paradigm. Companies P, S and C above are modelled as players (or, principals). Principals are the responsible actors in the distributed environment, and responsibility is codified in contracts, which generalise the precondition- and postcondition-style contracts of PBC. Each contract is a game between two principals, which specifies *what* should be communicated between them and *when*. This notion respects compositionality, as bilateral contracts only mention the exact part of the context to whom the principal has commitments. It is, of course, possible to generalise to multiparty contracts, but we restrict ourselves to bilateral contracts both for simplicity, and because it forces compositionality and eliminates unintended blame propagation.

The *moves* of principals in a game are what they communicate to each other in each round. Guarantees are quantified by assigning to each transition of the game state a *payoff*, which can be thought of as the incremental payment to the first player from the second, resulting from the transition. Since communication games may go on indefinitely, we assign payoffs also to non-terminal states of the games. A payoff may represent either a payment for services properly rendered, or a fine for unsatisfactory performance. The game-theoretic formulation of a distributed PBC contract is therefore an *infinite, simultaneous, zero-sum, two-person game*. In general, each player participates in multiple, concurrent games, and aims to maximise his total payoff, rather than to do well in any particular game.

Even though each contract specifies a zero-sum game, and all principals may be assumed to be rational and enter contracts in expectation of a positive payoff, it does not necessarily mean that at least one principal has to lose. The reason is that the system is considered *open*, hence the "losing" principal may actually have an overall positive payoff, as a result of some unknown contracts with an unspecified environment, or ultimately with "nature".

A contract describes a logical commitment between two principals, but not how communication is enacted physically. In order to fulfil its contractual obligations, each principal implements an overall *strategy* for playing all of its communication games. An implementation consists of a set of *processes* for performing actual computation and communication, and a means of *delegation* to other principals. The latter makes it possible to satisfy a logical commitment without doing the actual communication oneself—in some cases it may not even be possible to be in charge of the communication oneself.

**Communication model** Having described how contracts are extended from one-shot input–output contracts, we need to generalise sequential programs to concurrent processes. We aim for a simple model of communication that assumes no common computational model at the peers in the network. Our model of communication is inspired by the Input Output Timed Automaton (IOTA) model [14], in which messages are sent asynchronously between automata. Unlike process calculi such as CSP [44], CCS [70], and the $\pi$-calculus [71], we assume no advanced synchronisation primitives, and the definition of processes is *extensional* (black box), rather than *intensional*, in order to reflect that the internal structure of a process may not be known. Furthermore, there is no possibility of refusing input as in for instance CSP, which means that contracts can be defined on traces of actual communication rather than traces of input–output requests.

The link between implementations and contracts is established via a notion of *contract portfolio conformance*, which generalises the definition of Hoare triple validity [119]. Contract conformance is defined as a safety property [4], meaning that all violations happen in finite time. This is due to the fact that implementations are black boxes, hence their internal organisation cannot be inspected, and monitoring of contracts should be possible only by inspecting the observational behaviour of implementations. However, the restriction to safety properties does not imply that contract conformance corresponds to partial correctness of Hoare logic, in which a program stuck in an infinite loop satisfies any contract. Rather, it can be seen as a cross between partial correctness and total correctness, which we may call *timed total correctness*.

### 3.1.2 Outline

The remainder of the chapter is structured as follows. In Section 3.2 we introduce an abstract model of communication, in which processes are described only by their observational behaviour, and we define what it means to combine processes. In Section 3.3 we introduce a model of I/O automata that is equivalent to the process model, but in some cases easier to reason with. We introduce principals and contracts in Section 3.4, and Section 3.5 provides the link between processes and contracts by means of contract conformance. We show that contract conformance permits compositional reasoning.

## 3.2 Process Model

In order to extend programming-by-contract to a distributed setting, we must first define what kind of peers we are interested in specifying. This section and the following section target that question. In Section 3.4 we return to the question of what a contract is, and who the responsible actors are. The reader may skip to Section 3.3, in which we define a model of communication that is equivalent to the process model of this section. However, the notions of channels and moves are also used in that model. The process model is included to illustrate the choices that underlie our communication model.

As in IOTA [14], CSP [44], CCS [70], and the $\pi$-calculus [71], we use *channels* as an abstraction for an ideal communication medium. We denote the set of all channels by $\mathcal{C}$, and we write $\alpha, \beta, \gamma, \ldots$ for channels. Unlike CCS and the $\pi$-calculus, channels are directed, and they have exactly one sender and one receiver. The reason for this low-level approach is to have as few assumptions about the communication medium as possible. The restriction means, for instance, that it is not possible for a process to broadcast to multiple processes on a single channel—instead the fan out to the receiving processes has to implemented explicitly via individual channels.

**Definition 3.2.1.** Given a finite set of channels $C \subseteq_{\text{fin}} \mathcal{C}$ and an alphabet $\mathcal{A}_c$ for each $c \in C$, a *move* $m$ is an element of the set $\mathcal{M}_C = \Pi_{c \in C} \mathcal{A}_c$. We write $m_1, m_2, m_3, \ldots$ for moves. A *log* $l$ over $C \subseteq_{\text{fin}} \mathcal{C}$ with end time $t \in \mathbb{N}$ is an element of the set $\mathcal{L}_C^t = [0; t) \to \mathcal{M}_C$. The set of all logs over $C \subseteq \mathcal{C}$ is defined by $\mathcal{L}_C = \biguplus_{t \in \mathbb{N}} \mathcal{L}_C^t$, where $\uplus$ denotes disjoint union. We write $l_1, l_2, l_3, \ldots$ for logs.

Moves and logs are introduced in order to represent processes by their observational behaviour. A move over a finite set of channels is a snapshot of what gets communicated on the channels at a particular point in time. Each $\mathcal{A}_c$ must contain the special silent action $\varepsilon$. A log over $C$ with end time $t$ is a description of all that has happened on the channels of $C$ before time $t$. We note that all timestamps are in $\mathbb{N}$, which means that we consider a discrete model of time. The restriction to a discrete time model is for simplicity—we discuss briefly in Section 3.2.1 why a dense time model is not straightforward.

In order to represent logs, we use a list-like notation, in which silent actions are omitted. For instance:

$$[t_1 : (\alpha \mapsto 10), t_2 : (\beta \mapsto 3), t_3 : (\alpha \mapsto 2, \beta \mapsto 5); t_4], \tag{3.1}$$

with $t_1 < t_2 < t_3 < t_4$ represents the log $l \in \mathcal{L}_{\{\alpha,\beta\}}^{t_4}$ defined by:

$$l(t)(c) = \begin{cases} 10 & \text{if } t = t_1 \text{ and } c = \alpha, \\ 3 & \text{if } t = t_2 \text{ and } c = \beta, \\ 2 & \text{if } t = t_3 \text{ and } c = \alpha, \\ 5 & \text{if } t = t_3 \text{ and } c = \beta, \\ \varepsilon & \text{otherwise.} \end{cases}$$

Given a log $l \in \mathcal{L}_C$ we denote by $\operatorname{eol}(l)$ the end of the log, that is $\operatorname{eol}(l) = t$ whenever $l \in \mathcal{L}_C^t$. Given two logs $l_1 \in \mathcal{L}_{C_1}^t$ and $l_2 \in \mathcal{L}_{C_2}^t$ with $C_1 \cap C_2 = \emptyset$ we define the combined log $l_1 \cup l_2 \in \mathcal{L}^t_{C_1 \cup C_2}$ by:

$$(l_1 \cup l_2)(t')(c) = \begin{cases} l_1(t')(c) & \text{if } c \in C_1, \\ l_2(t')(c) & \text{if } c \in C_2. \end{cases}$$

Given a log $l \in \mathcal{L}_C$ it can be restricted to channels $C' \subseteq C$, written $l_{|C'} \in \mathcal{L}_{C'}$, and restricted to time $t \leq \mathrm{eol}(l)$, written $l_{|t} \in \mathcal{L}^t_C$, where:

$$l_{|C'}(t) = l(t)_{|C'} \qquad\qquad l_{|t} = l_{|[0;t)}.$$

(We use ordinary function restriction on the right-hand side of the equations above.) For a given log set $\mathcal{L}_C$ we define the partial order $(\cdot \sqsubseteq \cdot) \subseteq \mathcal{L}_C \times \mathcal{L}_C$ by:

$$l_1 \sqsubseteq l_2 \text{ iff } \mathrm{eol}(l_1) \leq \mathrm{eol}(l_2) \text{ and } l_1 = l_{2|\mathrm{eol}(l_1)}.$$

Given the definition of logs we can now concisely capture processes as transformations on logs. However, in order to faithfully capture the intuition of "communicating black boxes", we require two conditions:

**Definition 3.2.2.** A *process* $\mathsf{p} \in \mathfrak{P}$ is a triple $\mathsf{p} = (C_I, C_O, f)$, where the *input channels* $C_I$ and *output channels* $C_O$ are disjoint and finite, and $f : \mathcal{L}_{C_I} \to \mathcal{L}_{C_O}$ is a *log transformer*. The log transformer $f$ must satisfy the following conditions for all logs $l, l_1, l_2 \in \mathcal{L}_{C_I}$ and timestamps $t \in \mathbb{N}$ with $t < \min(\mathrm{eol}(l_1), \mathrm{eol}(l_2))$:

(1) $\mathrm{eol}(l) = \mathrm{eol}(f(l))$, and

(2) if $l_{1|t} = l_{2|t}$ then $f(l_1)_{|t+1} = f(l_2)_{|t+1}$.

The definition of a process requires a fixed set of finite, disjoint input channels and output channels. Input channels are the source of stimuli to a process, and output channels are the reactions. Channels are fixed for simplicity—unlike for instance the $\pi$-calculus, channels cannot be created dynamically, hence the network topology is static. The definition requires that the relation between stimuli and reaction be deterministic, that is we do not model internal nondeterminism. It is possible to generalise the definition to powersets in order to model internal nondeterminism as well, but we omit it here for simplicity. The two additional requirements on log transformers are motivated below:

(1) states that a log transformer must preserve the end time for logs. Intuitively this means that the observation of input and output ends at the same time.

(2) is called *strict monotonicity*[1], and it guarantees two properties. First, processes cannot "change the past". That is, if output is known for a log $l$ and $l'$ is an extension of $l$, then the output for $l'$ is an extension of the output for $l$. Second, processes cannot respond instantly. That is, if two input logs are equal before time $t$—and possibly differing at time $t$—then output is equal before and at time $t$. This restriction is imposed to reflect the intuition that reaction takes time. That is, we will not allow processes to respond infinitely fast.

The following lemma shows that strict monotonicity implies monotonicity:

[1]We remark that the later notation of *guardedness* due to Krishnaswami and Benton [56] is very similar to our notion of strict monotonicity.

**Lemma 3.2.3.** *Let* $\mathsf{p} = (C_I, C_O, f)$ *be a process. Then* $f : \mathcal{L}_{C_I} \to \mathcal{L}_{C_O}$ *is monotone with respect to* $\sqsubseteq$*.*

*Proof.* Assume $l_1 \sqsubseteq l_2$. Then $\mathrm{eol}(f(l_1)) = \mathrm{eol}(l_1) \leq \mathrm{eol}(l_2) = \mathrm{eol}(f(l_2))$ so if $f(l_1) \not\sqsubseteq f(l_2)$ then there exists some $t < \mathrm{eol}(f(l_1))$ such that $f(l_1)(t) \neq f(l_2)(t)$. But since $l_{1|t} = l_{2|t}$ it follows by strict monotonicity that $f(l_1)(t) = f(l_2)(t)$, which is a contradiction. Hence we must have that $f(l_1) \sqsubseteq f(l_2)$ as required. □

It follows from the lemma that an extension of the process model to infinite logs will not add extra expressivity. That is, the infinite output associated with an infinite input will be uniquely determined by all finite prefixes of input and output, similar to the notion of continuity in denotational semantics [119].

In order to reason about the behaviour of multiple processes we next introduce process composition:

**Definition 3.2.4.** Let $\mathsf{p}_1 = (C_I^1, C_O^1, f^1)$ and $\mathsf{p}_2 = (C_I^2.C_O^2, f^2)$ be processes with $C_I^1 \cap C_I^2 = C_O^1 \cap C_O^2 = \emptyset$. The *parallel composition* $\mathsf{p}_1 \parallel \mathsf{p}_2 = (C_I, C_O, f)$ of $\mathsf{p}_1$ and $\mathsf{p}_2$ is defined by:

$$C_I = (C_I^1 \cup C_I^2) \setminus C_{\mathrm{int}} \qquad \text{(input channels)}$$
$$C_O = (C_O^1 \cup C_O^2) \setminus C_{\mathrm{int}}, \qquad \text{(output channels)}$$

where $C_{\mathrm{int}} = (C_I^1 \cap C_O^2) \cup (C_I^2 \cap C_O^1)$ are *internal channels.* For $l \in \mathcal{L}_{C_I}^t$ we first define $\mathcal{I}_n^1 \in \mathcal{L}_{C_O^1}^t$ and $\mathcal{I}_n^2 \in \mathcal{L}_{C_O^2}^t$ inductively as follows:

$$\forall t' \in [0;t).\ \mathcal{I}_0^1(t')(c) = \varepsilon \qquad \mathcal{I}_{n+1}^1 = f^1((\mathcal{I}_n^2 \cup l)_{|C_I^1})$$
$$\forall t' \in [0;t).\ \mathcal{I}_0^2(t')(c) = \varepsilon \qquad \mathcal{I}_{n+1}^2 = f^2((\mathcal{I}_n^1 \cup l)_{|C_I^2}).$$

Then $f(l) = (\mathcal{I}_N^1 \cup \mathcal{I}_N^2)_{|C_O}$, where $N$ is such that $\mathcal{I}_N^1 = \mathcal{I}_{N+1}^1$ and $\mathcal{I}_N^2 = \mathcal{I}_{N+1}^2$.

At first sight, the definition of parallel composition may seem rather involved. The reason for the iterative definition is that internal communication between the two processes needs to be "fed back" as input. In the first iteration we calculate the output for both processes with respect to the external input. Some of the output then needs to be routed internally, which is what happens in the second iteration. But this may in turn result in new internal messages that are included in the next iteration. This iteration is continued until a fixed point is reached. The example below illustrates parallel composition with two concrete processes.

**Example 3.2.5.** Let $\mathsf{p}_1 = (\{\alpha\}, \{\beta\}, f_1)$ and $\mathsf{p}_2 = (\{\beta\}, \{\alpha, \gamma\}, f_2)$ be two processes defined by:

$$f_1(l)(t)(\beta) = \begin{cases} 1 & \text{if } t = 0, \\ 2x & \text{if } l(t-1)(\alpha) = x, \\ \varepsilon & \text{otherwise,} \end{cases} \quad f_2(l)(t)(c) = \begin{cases} x+1 & \text{if } l(t-1)(\beta) = x, \\ \varepsilon & \text{otherwise.} \end{cases}$$

The two processes are depicted in Figure 3.1. $\mathsf{p}_1$ initially outputs 1 on $\beta$ and continuously doubles input from $\alpha$ on $\beta$. $\mathsf{p}_2$ continuously increments its input from $\beta$ on both $\alpha$ and $\gamma$. The parallel composition $\mathsf{p}_1 \parallel \mathsf{p}_2$ has no input channels and

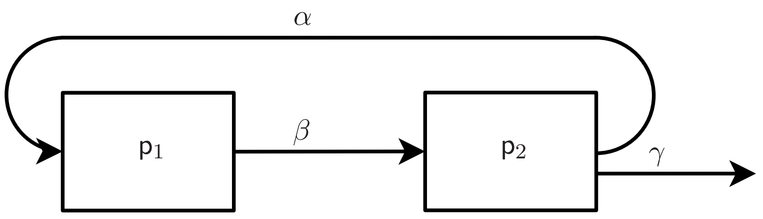


Figure 3.1: Parallel composition of two processes.

$\gamma$ as output channel. In order to calculate the output after, say, 6 time units, we must apply the composed log transformer to the log $[\cdot; 6]$. The iterative procedure of Definition 3.2.4 produces the following approximations:

$$\begin{aligned}
&\mathcal{I}_0^1 = [\cdot; 6] && \mathcal{I}_0^2 = [\cdot; 6] \\
&\mathcal{I}_1^1 = [0 : (\beta \mapsto 1); 6] && \mathcal{I}_1^2 = [\cdot; 6] \\
&\mathcal{I}_2^1 = [0 : (\beta \mapsto 1); 6] && \mathcal{I}_2^2 = [1 : (\alpha, \gamma \mapsto 2); 6] \\
&\mathcal{I}_3^1 = [0 : (\beta \mapsto 1), 2 : (\beta \mapsto 4); 6] && \mathcal{I}_3^2 = [1 : (\alpha, \gamma \mapsto 2); 6] \\
&\mathcal{I}_4^1 = [0 : (\beta \mapsto 1), 2 : (\beta \mapsto 4); 6] && \mathcal{I}_4^2 = [1 : (\alpha, \gamma \mapsto 2), 3 : (\alpha, \gamma \mapsto 5); 6] \\
&\mathcal{I}_5^1 = [0 : (\beta \mapsto 1), 2 : (\beta \mapsto 4), && \\
&\qquad 4 : (\beta \mapsto 10); 6] && \mathcal{I}_5^2 = [1 : (\alpha, \gamma \mapsto 2), 3 : (\alpha, \gamma \mapsto 5); 6] \\
&\mathcal{I}_6^1 = \mathcal{I}_5^1 && \mathcal{I}_6^2 = [1 : (\alpha, \gamma \mapsto 2), 3 : (\alpha, \gamma \mapsto 5), \\
& && \qquad 5 : (\alpha, \gamma \mapsto 11); 6] \\
&\mathcal{I}_7^1 = \mathcal{I}_6^1 && \mathcal{I}_7^2 = \mathcal{I}_6^2
\end{aligned}$$

Hence the output is $(\mathcal{I}_6^1 \cup \mathcal{I}_6^2)_{|\{\gamma\}} = [1 : (\gamma \mapsto 2), 3 : (\gamma \mapsto 5), 5 : (\gamma \mapsto 11); 6]$.

It does not follow immediately that the fixed point of Definition 3.2.4 always exists, nor that the composition of two processes is itself a process. However, the requirement of strict monotonicity ensures the desired result:

**Lemma 3.2.6.** *Let* $\mathsf{p}_1 = (C_I^1, C_O^1, f^1)$ *and* $\mathsf{p}_2 = (C_I^2.C_O^2, f^2)$ *be two processes with* $C_I^1 \cap C_I^2 = C_O^1 \cap C_O^2 = \emptyset$. *Then the parallel composition* $\mathsf{p}_1 \parallel \mathsf{p}_2$ *exists and it is a process.*

*Proof.* The proof is presented in Appendix C.1, page 195. □

The model of processes and their composition is—we believe—an intuitive model of communicating black boxes that may be put together to form new boxes. The weakness of the model, however, is that it is cumbersome to reason about process composition due to the iterative definition. For instance, we want to prove that process composition is commutative and associative. Even though this is indeed the case, the latter is easier to show in an equivalent automaton model, which we define in Section 3.3. We therefore postpone the associativity result.

**Lemma 3.2.7.** *Let* $\mathsf{p}_1 = (C_I^1, C_O^1, f^1)$ *and* $\mathsf{p}_2 = (C_I^2.C_O^2, f^2)$ *be two processes with* $C_I^1 \cap C_I^2 = C_O^1 \cap C_O^2 = \emptyset$. *Then* $\mathsf{p}_1 \parallel \mathsf{p}_2 = \mathsf{p}_2 \parallel \mathsf{p}_1$.

*Proof.* The result follows directly from the corresponding commutativity of the operators $\cup$ and $\cap$. □

### 3.2.1 A Digression on Time

In this subsection we briefly discuss why it is not straightforward to extend our existing model with a dense time model. The reader may skip this section as it is not a prerequisite for the remaining chapter.

Consider $\mathbb{Q}_+$ as a model of time rather than $\mathbb{N}$, similar to Alur and Dill [5]. Then a log with end time $q \in \mathbb{Q}_+$ is a function of the type $[0; q) \to \mathcal{M}_C$. Since this definition allows for infinitely many messages in finite time, we follow Alur and Dill and require logs to have progress:

**Definition 3.2.8.** A log $l \in \mathcal{L}_C$ is said to have *progress* whenever the set $\{q \in \mathbb{Q}_+ \mid l(q)(\alpha) \neq \varepsilon \text{ for some } \alpha \in C\}$ is finite.

The crucial point in extending our model to a dense time domain is to redefine strict monotonicity of Definition 3.2.2 (2). The definition should still imply ordinary monotonicity and reflect the intuition that "reaction takes time". One possible generalisation is to have a fixed $\delta \in \mathbb{Q}_+$ and redefine strict monotonicity:

$$\text{if } l_{1|t} = l_{2|t} \text{ then } f(l_1)_{|t+\delta} = f(l_2)_{|t+\delta}. \tag{3.2}$$

However, this definition is equivalent to a discrete time model where $\mathbb{Q}_+$ is partitioned into $\delta$-intervals. Another definition that utilises the density of $\mathbb{Q}_+$ is:

$$\text{if } l_{1|t} = l_{2|t} \text{ then } f(l_1)(t') = f(l_2)(t') \text{ for all } t' \leq t. \tag{3.3}$$

That is, a process can be *arbitrarily* fast—but still not infinitely fast. Unfortunately, this definition is problematic as we shall see in the following example.

**Example 3.2.9.** Let $\mathsf{p}_1 = (\{\alpha\}, \{\beta\}, f_1)$ and $\mathsf{p}_2 = (\{\beta\}, \{\alpha, \gamma\}, f_2)$ be two processes defined by:

$$f_1(l)(t)(\beta) = \begin{cases} 42 & \text{if } t = 0, \\ x & \text{if } l(1 - \frac{1}{n})(\alpha) = x \text{ and } t = 1 - \frac{1}{n+1}, \\ \varepsilon & \text{otherwise,} \end{cases}$$

$$f_2(l)(t)(c) = \begin{cases} x & \text{if } l(1 - \frac{1}{n})(\beta) = x \text{ and } t = 1 - \frac{1}{n+1}, \\ \varepsilon & \text{otherwise.} \end{cases}$$

Both $\mathsf{p}_1$ and $\mathsf{p}_2$ are valid processes, that is if the input log has progress then so has the output log, since both processes duplicate some of their input with decreasing delay. For example, $\mathsf{p}_1$ transforms the log

$$\left[\frac{2}{3} : (\alpha \mapsto 1), \frac{9}{10} : (\alpha \mapsto 2), 33 : (\alpha \mapsto 3); 40\right]$$

to

$$\left[0 : (\beta \mapsto 42), \frac{3}{4} : (\beta \mapsto 1), \frac{10}{11} : (\beta \mapsto 2); 40\right].$$

However, when composed in parallel we get a process that outputs the following infinite sequence of 42's:

$$\left[\frac{1}{2} : (\gamma \mapsto 42), \frac{3}{4} : (\gamma \mapsto 42), \frac{5}{6} : (\gamma \mapsto 42), \ldots, \frac{2i-1}{2i} : (\gamma \mapsto 42), \ldots\right].$$

Hence when applied to the empty log with end time 1, $\mathsf{p}_1$ and $\mathsf{p}_2$ will produce an output log without progress, as the sequence $\{1 - \frac{1}{2i}\}_{i\in\mathbb{N}}$ converges towards 1.

The example above illustrates why condition (3.3) is too general: two processes that in isolation preserve progress and satisfy condition (3.3) may speed each other up to infinity. Therefore, in order to utilise a dense time model we will need a definition of strict monotonicity that is more general than (3.2) but more restrictive than (3.3). One possibility is to have a fixed delay $\delta_c$ per channel $c$, rather than a uniform $\delta$ as in (3.2). We leave this investigation as future work.

## 3.3 Automaton Model

In this section we introduce a communication model that is equivalent to the process model of the previous section. We focus on the model of this section in the remainder of the chapter.

**Definition 3.3.1.** An *I/O automaton* $\mathsf{a} \in \mathfrak{A}$ is a 6-tuple $(C_I, C_O, S, s_0, \delta_{\mathrm{o}}, \delta_{\mathrm{t}})$. *Input channels* $C_I$ *and output channels* $C_O$ *are disjoint and finite,* $S$ *is the (potentially infinite) set of automaton states, and* $s_0 \in S$ *is the start state.* $\delta_{\mathrm{o}} : S \to \mathcal{M}_{C_O}$ *is the output function and* $\delta_{\mathrm{t}} : S \times \mathcal{M}_{C_I} \to S$ *is the transition function.*

The I/O automaton's output in the current time unit is determined by its internal state, while its next state depends on the current one, and the input received. Hence the output $m_o$ as a reaction to input $m_i$ is only observed in the next time unit, compare the intuition of the process model in which reaction takes time. We do not impose any structure on the set of states $S$, and the automaton does not specify how the transition functions are computed, only what they compute. The definition of parallel composition is simple:

**Definition 3.3.2.** Let $\mathsf{a}_1 = (C_I^1, C_O^1, S^1, s_0^1, \delta_{\mathrm{o}}^1, \delta_{\mathrm{t}}^1)$ and $\mathsf{a}_2 = (C_I^2.C_O^2, S^2, s_0^2, \delta_{\mathrm{o}}^2, \delta_{\mathrm{t}}^2)$ be two automata with $C_I^1 \cap C_I^2 = C_O^1 \cap C_O^2 = \emptyset$. The *parallel composition* $\mathsf{a}_1 \parallel \mathsf{a}_2 = (C_I, C_O, S^1 \times S^2, \langle s_0^1, s_0^2\rangle, \delta_{\mathrm{o}}, \delta_{\mathrm{t}})$ is defined by:

$$\delta_{\mathrm{o}}(\langle s_1, s_2\rangle) = (\delta_{\mathrm{o}}^1(s_1) \cup \delta_{\mathrm{o}}^2(s_2))_{|C_O}$$
$$\delta_{\mathrm{t}}(\langle s_1, s_2\rangle, m) = \langle \delta_{\mathrm{t}}^1(s_1, (m \cup \delta_{\mathrm{o}}^2(s_2))_{|C_I^1}), \delta_{\mathrm{t}}^2(s_2, (m \cup \delta_{\mathrm{o}}^1(s_1))_{|C_I^2})\rangle,$$

where $C_I$ and $C_O$ are as in Definition 3.2.4. For a move $m \in \mathcal{M}_C$, $m_{|C'}$ denotes the domain restriction of $m$ to $C'$, and moves $m_1 \in \mathcal{M}_{C_1}$ and $m_2 \in \mathcal{M}_{C_2}$ over disjoint channel sets $C_1$ and $C_2$ are combined as $m_1 \cup m_2 \in \mathcal{M}_{C_1 \cup C_2}$.

Unlike processes, automata have a notion of internal state. This means that two automata that "behave the same way" need not be the same. However, we do not wish to distinguish such automata, which motivates the following definition:

**Definition 3.3.3.** Let $\mathsf{a}_1 = (C_I, C_O, S^1, s_0^1, \delta_{\mathrm{o}}^1, \delta_{\mathrm{t}}^1)$ and $\mathsf{a}_2 = (C_I, C_O, S^2, s_0^2, \delta_{\mathrm{o}}^2, \delta_{\mathrm{t}}^2)$ be two automata. A relation $R \subseteq S^1 \times S^2$ is said to be a *bisimulation* for $\mathsf{a}_1$ and $\mathsf{a}_2$ iff the following holds for all states $s_1 \in S_1$ and $s_2 \in S_2$ and moves $m \in \mathcal{M}_{C_I}$:

$$\text{if } (s_1, s_2) \in R \text{ then } \delta_{\mathrm{o}}^1(s_1) = \delta_{\mathrm{o}}^2(s_2) \text{ and } (\delta_{\mathrm{t}}^1(s_1, m), \delta_{\mathrm{t}}^2(s_2, m)) \in R.$$

The two automata are said to be *bisimilar*, written $\mathsf{a}_1 \equiv \mathsf{a}_2$, whenever $(s_0^1, s_0^2) \in R$ for some bisimulation $R$.

Bisimilarity satisfies the expected property of being an equivalence relation on automata. Moreover, bisimilarity is a congruence relation with respect to parallel composition:

**Lemma 3.3.4.** *Bisimilarity is a congruence relation on automata. That is, it is an equivalence relation (reflexive, transitive and symmetric) and the following holds for all automata* $\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3, \mathsf{a}_4 \in \mathfrak{A}$*:*

$$\textit{if } \mathsf{a}_1 \equiv \mathsf{a}_2 \textit{ and } \mathsf{a}_3 \equiv \mathsf{a}_4 \textit{ then } \mathsf{a}_1 \parallel \mathsf{a}_3 \equiv \mathsf{a}_2 \parallel \mathsf{a}_4.$$

*Proof.* The proof is presented in Appendix C.1, page 197. □

We can now show that parallel composition of automata is associative modulo bisimilarity. Note that we need to require pairwise disjointedness of input channels and output channels for all three automata—otherwise parallel composition is in fact not associative, even though it may be well-defined!

**Lemma 3.3.5.** *Let* $\mathsf{a}_i = (C_I^i, C_O^i, S^i, s_0^i, \delta_{\mathrm{o}}^i, \delta_{\mathrm{t}}^i)$ *be automata for* $i = 1, 2, 3$ *with:*

$$C_I^1 \cap C_I^2 = C_I^1 \cap C_I^3 = C_I^2 \cap C_I^3 = \emptyset,$$
$$C_O^1 \cap C_O^2 = C_O^1 \cap C_O^3 = C_O^2 \cap C_O^3 = \emptyset.$$

*Then* $\mathsf{a}_1 \parallel (\mathsf{a}_2 \parallel \mathsf{a}_3) \equiv (\mathsf{a}_1 \parallel \mathsf{a}_2) \parallel \mathsf{a}_3$.

*Proof.* The proof is presented in Appendix C.1, page 198. The proof consists of two parts: first we show that the two automata have the same input channels and output channels, and second we construct a bisimulation. □

## 3.3.1 Equivalence of Models

In this subsection we give a brief account of the proof that the models of communication are in fact equivalent. The result is not surprising, and we therefore refer to Appendix C.3 for the full details.

In order to establish the equivalence result, we must first make it clear what it means for the two models to be equivalent. Clearly, we need two maps $\ulcorner\cdot\urcorner : \mathfrak{A} \to \mathfrak{P}$ and $\llcorner\cdot\lrcorner : \mathfrak{P} \to \mathfrak{A}$, and the two maps must somehow be mutually inverse and both be bijections. However, as we have seen, the automaton model allows for automata that are bisimilar but not structurally equal. We therefore instead consider two maps:

$$\ulcorner\cdot\urcorner : (\mathfrak{A}/{\equiv}) \to \mathfrak{P} \quad \text{and} \quad \llcorner\cdot\lrcorner : \mathfrak{P} \to (\mathfrak{A}/{\equiv}).$$

That is, we only consider automata modulo bisimilarity. The map $\ulcorner\cdot\urcorner$ is induced by a mapping of automata to process that runs the automaton incrementally on the input log—this mapping, of course, must respect bisimilarity in order to induce a mapping on the quotient space. The map $\llcorner\cdot\lrcorner$ is constructed by building an automaton that maintains in its state the log of previous messages, and appends new messages to the state when they are received. Note that this construction relies on the fact that I/O automata may have infinite state space—if restricted to finite automata, the two models are in fact not equivalent!

We show in Appendix C.3 that the two maps constitute an isomorphism between the two models (Corollary C.3.13). That is, they are both bijections, they are mutually inverse, and they are homomorphic with respect to parallel composition. Moreover, we obtain that bisimilarity is equivalent to equality of the processes that automata denote, that is:

$$\mathsf{a}_1 \equiv \mathsf{a}_2 \text{ iff } \ulcorner \mathsf{a}_1 \urcorner = \ulcorner \mathsf{a}_1 \urcorner.$$

Hence bisimilarity and trace equivalence coincide.

The isomorphism result allows us to transfer results from one model to the other and vice versa. For instance, it follows from Lemma 3.2.7 that parallel composition of automata is commutative modulo bisimilarity (Corollary C.3.15) and it follows from Lemma 3.3.5 that parallel composition of processes is associative (Corollary C.3.14).

## 3.4 Principals and Contracts

We now pursue the extension of programming-by-contract (PBC) to a distributed environment. *Principals* refer to the administrative parties in the distributed environment, for instance a person or an organisation. We write $\mathsf{P}_1, \mathsf{P}_2, \mathsf{P}_3, \dots$ for principals. Two principals can negotiate a *contract*, which is an abstraction for communication obligations.

In order to capture different kinds of communication, we define *logical communication links*. A logical communication link specifies the type of messages to be communicated, and it is similar to channels. A logical communication link is written $\lambda$, sets of links are denoted $\Lambda$, and a move $m \in \mathcal{M}_\Lambda$ over a finite set of links $\Lambda$ is defined as for channels. A logical communication link is directed and always between two principals. For each logical communication link $\lambda$, there is a corresponding set of actions communicated on that link, denoted $\mathcal{A}_\lambda$. Each action set has a distinguished silent action $\varepsilon$. In the context of distributed computing, a logical link will typically have as action set the set of all IP packets between two predefined IP addresses. But logical links can also be used for modelling real-world events.

The reason why links are *logical* is that the principals at each end of a link need not be the actual physical sender or receiver for that link. The sender (and receiver respectively) of a logical link has the opportunity of being the physical sender (receiver), but it may pass on this opportunity to another principal. The term logical hence means that the principals have committed to some actions on the links in the contract, but they may not be in control of the underlying physical communication. This is what differentiates logical links from channels, compare Section 3.2. Logical links are always associated with exactly one contract, which we define next:

**Definition 3.4.1.** A *contract* between principals $\mathsf{P}$ (player) and $\mathsf{A}$ (adversary) is a 5-tuple $c = (\Lambda_{\mathsf{PA}}, \Lambda_{\mathsf{AP}}, G, g_0, \rho)$. $\Lambda_{\mathsf{PA}}$ and $\Lambda_{\mathsf{AP}}$ are finite sets of logical links from principal $\mathsf{P}$ to principal $\mathsf{A}$ and vice versa. $G$ is the (potentially infinite) set of *contract states*, and $g_0 \in G$ is the *start state*. $\rho : G \times \mathcal{M}_{\Lambda_{\mathsf{PA}}} \times \mathcal{M}_{\Lambda_{\mathsf{AP}}} \to G \times \mathbb{Q}$ is the *rule function* for the game.

A contract evolves in each time unit, based on the chosen moves $m_{\mathsf{P}}$ and $m_{\mathsf{A}}$ of the two players. Consider $(g', k) = \rho(g, m_{\mathsf{P}}, m_{\mathsf{A}})$: when the contract is in a state $g$, and the moves on $\Lambda_{\mathsf{PA}}$ and $\Lambda_{\mathsf{AP}}$ are $m_{\mathsf{P}}$ and $m_{\mathsf{A}}$ respectively then $g'$ is the new contract state, and $k$ is the—possibly negative—incremental payoff to $\mathsf{P}$ from

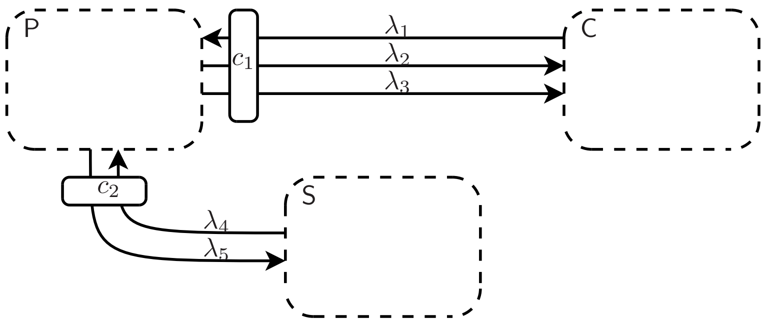


Figure 3.2: Graphical representation of three principals with bilateral contracts.

A. Time units are assumed to be small and fixed. General timing constraints are expressed by means of explicit counters in the game state, which we will see an example of soon. Note that there are no "illegal" moves per se: moves in violation of the nominal game rules will typically be assigned a large negative payoff, but the contract remains in a well-defined state, to guide an orderly recovery.

**Observation 3.4.2.** A contract $(\Lambda_{\mathsf{PA}}, \Lambda_{\mathsf{AP}}, G, g_0, \rho)$ describes an *infinite, simultaneous, zero-sum, two-person game* between principals P and A. Finite contracts can be modelled by introducing a terminal state $g_t$ and extend $\rho$ such that $\rho(g_t, m_{\mathsf{P}}, m_{\mathsf{A}}) = (g_t, 0)$ for all $(m_{\mathsf{P}}, m_{\mathsf{A}}) \in \mathcal{M}_{\Lambda_{\mathsf{PA}}} \times \mathcal{M}_{\Lambda_{\mathsf{AP}}}$.

In order to model situations with more than two principals, principals can in general negotiate a (finite) set of contracts:

**Definition 3.4.3.** A *contract portfolio* for a principal P is a finite set of contracts $C = \{c_1, \ldots, c_n\}$ where $c_i = (\Lambda_{\mathsf{PA}_i}, \Lambda_{\mathsf{A}_i\mathsf{P}}, G_i, g_i, \rho_i)$ for $i = 1, \ldots, n$.

Contract portfolios make it possible to model multiparty scenarios, by means of bilateral contracts only. This approach is different from the commonly used global approach to multiparty scenarios, represented typically as sequence diagrams [102]. For instance, in the context of web services the global approach is manifested in the web services choreography description language (WSCDL)[2]. The difference is illustrated in the following example.

**Example 3.4.4.** Consider the three principals in Figure 3.2: a service provider P, from the viewpoint of whom we are considering, a subcontractor S, and a client C. P offers a cellphone greeting service, which enables client C to send a greeting card MMS to a specified cellphone number. To send the actual MMS, the service provider has subcontracted with an MMS gateway provider S, who provides the service of sending MMSs with arbitrary content. The traditional way to describe such a scenario is by means of a global choreography as in Figure 3.3.

The global choreography may provide good intuition. However, as mentioned previously, there are shortcomings to this approach. A more faithful model is to consider bilateral agreements between P and C, and P and S, respectively, in which C and S have no awareness of each other. Informally, P's contract with C says:

[2]http://www.w3.org/TR/wsdl20/.

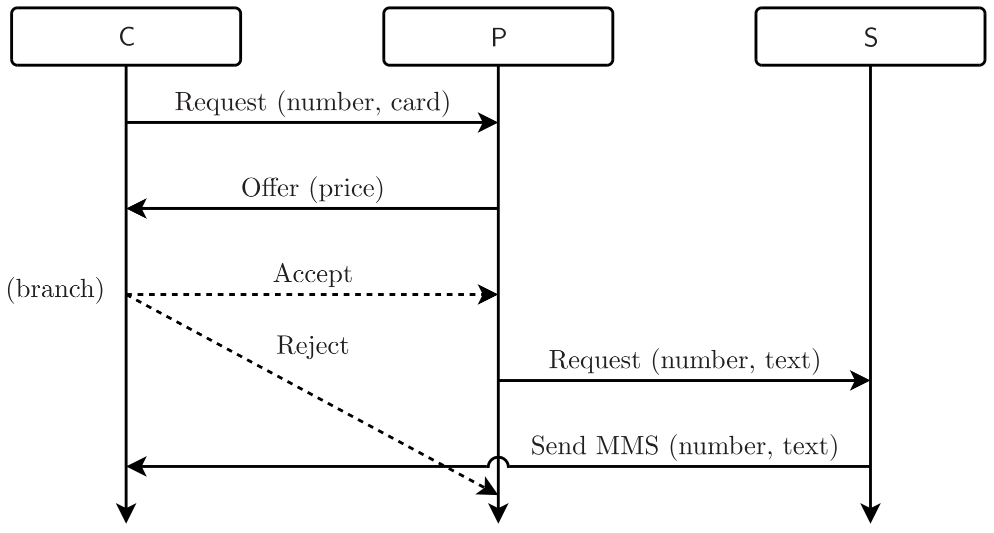


Figure 3.3: A global choreography for principals P, C, and S (time flows downwards).

> C can request the price of sending greeting card $g$ to cellphone number $n$ and P can reply with a price $p$. Subsequently C can accept or reject. If C accepts, P has to send the MMS before at most $t$ time units. If P fails to do so, P is assigned a penalty of 1.

P is the one responsible for sending the MMS, and the contract has no mention of S. Payoffs model what *should* be paid from C to P, not what has been paid. Actions are used to model real-world events, for instance sending of the MMS, communication between P and C, etc. The contract with C contains three logical links:

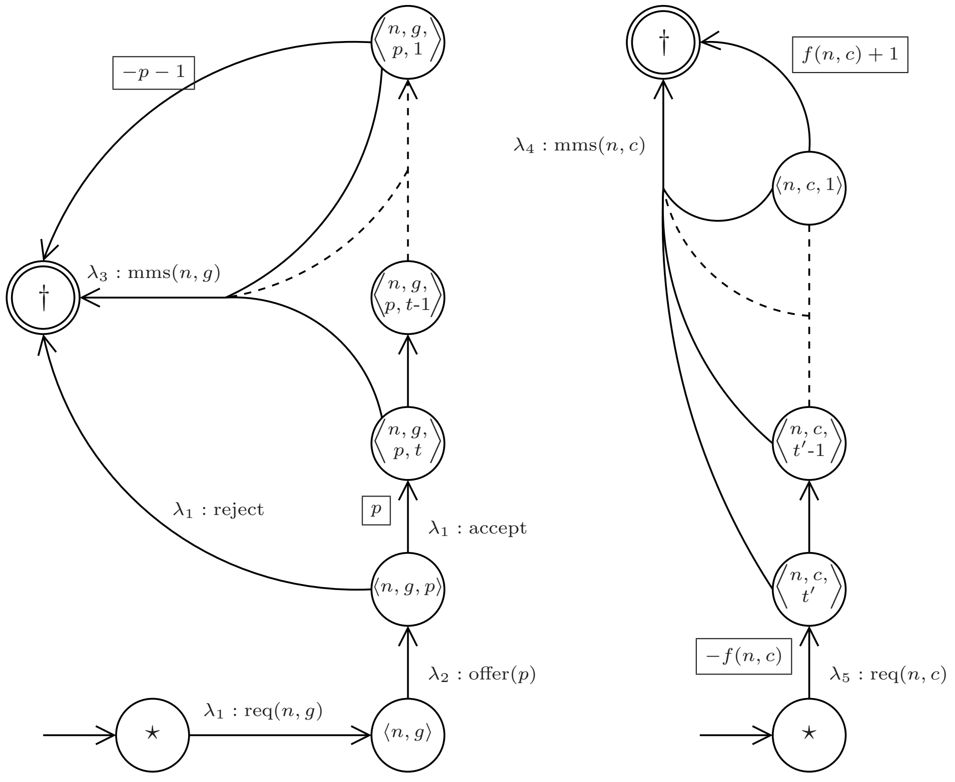
$$c_1 = (\{\lambda_2, \lambda_3\}, \{\lambda_1\}, G_1, g_1, \rho_1).$$

$\lambda_1$ and $\lambda_2$ are used for communication between P and C, and $\lambda_3$ is used for the special communication of sending an MMS. The fact that $\lambda_3$ is directed from P to C should not be interpreted that an MMS has to be sent from P to C—it means that P is responsible to C for sending an MMS.

Formally we therefore have (omitting the silent actions, letting $\mathcal{G}$ denote the set of all greeting cards, and letting $\mathcal{N}$ denote the set of phone numbers):

$$\begin{aligned}\mathcal{A}_{\lambda_1} &= \{\text{accept}, \text{reject}\} \cup \{\text{req}(n, g) \mid n \in \mathcal{N} \text{ and } g \in \mathcal{G}\},\\ \mathcal{A}_{\lambda_2} &= \{\text{offer}(p) \mid p \in \mathbb{Q}\},\\ \mathcal{A}_{\lambda_3} &= \{\text{mms}(n, g) \mid n \in \mathcal{N} \text{ and } g \in \mathcal{G}\}.\end{aligned}$$

The formalised contract is presented graphically in Figure 3.4 (left). Contract states are depicted as circles, and the double-circled state is a terminal state, compare Observation 3.4.2. An arrow from $g_1$ to $g_2$ with label $\lambda : a$ and a boxed $\boxed{k}$ means that $\rho(g_1, m_1, m_2) = (g_2, k)$, whenever $m_1(\lambda) = a$ or $m_2(\lambda) = a$ (depending on who of the principals is responsible for $\lambda$). When no box is present on a transition, it means an implicit $\boxed{0}$ (that is, no payoff). An arrow from $g_1$ to $g_2$ with no label and (implicit) $\boxed{k}$ means that $\rho(g_1, m_1, m_2) = (g_2, k)$, whenever no other label



$-p-1$
$\langle n, g, p, 1\rangle$
$\lambda_3$ : mms$(n, g)$
†
$\langle n, g, p, t\text{-}1\rangle$
$\langle n, g, p, t\rangle$
$\lambda_1$ : reject
$p$
$\lambda_1$ : accept
$\langle n, g, p\rangle$
$\lambda_2$ : offer$(p)$
⋆
$\lambda_1$ : req$(n, g)$
$\langle n, g\rangle$
†
$f(n, c) + 1$
$\lambda_4$ : mms$(n, c)$
$\langle n, c, 1\rangle$
$\langle n, c, t'\text{-}1\rangle$
$\langle n, c, t'\rangle$
$-f(n, c)$
$\lambda_5$ : req$(n, c)$
⋆

Figure 3.4: The contracts negotiated by P with C (left) and S (right).

matches $m_1$ and $m_2$. If there is no explicit unlabelled arrow from a state $g$, there is an implicit unlabelled arrow from $g$ to itself.

Some of the states in the diagram have internal state, for example $\langle n, g\rangle$. This means that the node actually represents a *class* of states—potentially one for each combination of $n$ and $g$. We therefore have:

$$\begin{aligned}
G_1 = \{\star, \dagger\} &\cup \{\langle n, g\rangle \mid n \in \mathcal{N} \text{ and } g \in \mathcal{G}\}\\
&\cup \{\langle n, g, p\rangle \mid n \in \mathcal{N} \text{ and } g \in \mathcal{G} \text{ and } p \in \mathbb{Q}\}\\
&\cup \{\langle n, g, p, \tau\rangle \mid n \in \mathcal{N} \text{ and } g \in \mathcal{G} \text{ and } p \in \mathbb{Q} \text{ and } 1 \leq \tau \leq t\},\\
g_1 = \star.
\end{aligned}$$

The rule function can be read from the diagram (according to the description above) and be presented as the tabular in Figure 3.5 (top). The tabular should be read from top to bottom, that is given a state $g$ and moves $m_\mathsf{P}$ and $m_\mathsf{C}$, then $\rho_1(g, m_\mathsf{P}, m_\mathsf{C})$ is determined by the first line matching $g$, $m_\mathsf{P}$, and $m_\mathsf{C}$ (that is, pattern matching where "–" matches any move).

The service provider P's contract with the MMS gateway S says:

> P can request an MMS to cellphone number $n$ with content $c$ at a price $f(n, c)$, for some predefined rate function $f$. Subsequently S must send the MMS before $t'$ time units. If S fails to do so, S is assigned a penalty of 1.

| $g$ | $m_{\mathsf{P}}$ | $m_{\mathsf{C}}$ | $\rho_1(g, m_{\mathsf{P}}, m_{\mathsf{C}})$ |
|---|---|---|---|
| $\star$ | – | $\lambda_1 \mapsto \text{req}(n, g)$ | $(\langle n, g\rangle, 0)$ |
| $\star$ | – | – | $(\star, 0)$ |
| $\langle n, g\rangle$ | $\lambda_2 \mapsto \text{offer}(p)$ | – | $(\langle n, g, p\rangle, 0)$ |
| $\langle n, g\rangle$ | – | – | $(\langle n, g\rangle, 0)$ |
| $\langle n, g, p\rangle$ | – | $\lambda_1 \mapsto \text{reject}$ | $(\dagger, 0)$ |
| $\langle n, g, p\rangle$ | – | $\lambda_1 \mapsto \text{accept}$ | $(\langle n, g, p, t\rangle, p)$ |
| $\langle n, g, p, \tau\rangle$ | $\lambda_3 \mapsto \text{mms}(n, g)$ | – | $(\dagger, 0)$ |
| $\langle n, g, p, 1\rangle$ | – | – | $(\dagger, -p - 1)$ |
| $\langle n, g, p, \tau + 1\rangle$ | – | – | $(\langle n, g, p, \tau\rangle, 0)$ |
| $\dagger$ | – | – | $(\dagger, 0)$ |

| $g$ | $m_{\mathsf{P}}$ | $m_{\mathsf{S}}$ | $\rho_2(g, m_{\mathsf{P}}, m_{\mathsf{S}})$ |
|---|---|---|---|
| $\star$ | $\lambda_5 \mapsto \text{req}(n, c)$ | – | $(\langle n, c, t'\rangle, -f(n, c))$ |
| $\star$ | – | – | $(\star, 0)$ |
| $\langle n, c, \tau\rangle$ | – | $\lambda_4 \mapsto \text{mms}(n, c)$ | $(\dagger, 0)$ |
| $\langle n, c, 1\rangle$ | – | – | $(\dagger, f(n, c) + 1)$ |
| $\langle n, c, \tau + 1\rangle$ | – | – | $(\langle n, c, \tau\rangle, 0)$ |
| $\dagger$ | – | – | $(\dagger, 0)$ |

Figure 3.5: The contracts negotiated by P with C (top) and S (bottom).

The formalised contract $c_2$ is presented in Figure 3.4 (right), where:

$$\begin{aligned}
c_2 &= (\{\lambda_5\}, \{\lambda_4\}, G_2, g_2, \rho_2),\\
G_2 &= \{\star, \dagger\} \cup \{\langle n, c, \tau\rangle \mid n \in \mathcal{N} \text{ and } c \in \mathcal{C} \text{ and } 1 \leq \tau \leq t'\},\\
g_2 &= \star,\\
\mathcal{A}_{\lambda_4} &= \{\text{mms}(n, c) \mid n \in \mathcal{N} \text{ and } c \in \mathcal{C}\},\\
\mathcal{A}_{\lambda_5} &= \{\text{req}(n, c) \mid n \in \mathcal{N} \text{ and } c \in \mathcal{C}\}.
\end{aligned}$$

$\mathcal{C}$ represents the set of all possible MMS content (which in particular contains the predefined greeting cards provided by P, that is $\mathcal{G} \subseteq \mathcal{C}$). The rule function can again be read from the diagram, and it is presented in Figure 3.5 (bottom).

Now P has negotiated contracts with principals C and S. But the contracts are not "active" yet, since a physical realisation of the logical links has to be established. Hence, P has to construct a physical implementation for fulfilling its contract portfolio $\{c_1, c_2\}$. The contract with S bears no obligations, hence this contract is easy to fulfil, but in the contract $c_1$ with C, P has the obligation to send the greeting card MMS (at least if P intends to make money—otherwise denying to offer a price will do). Implementations is the subject of the next section, where we return to this example.

## 3.5 Implementations and Conformance

In this section we describe how contracts between principals are realised physically by means of *implementations*. An implementation defines a strategy for playing the games specified by a contract portfolio. A strategy consists of a set of automata together with a mapping of logical links in the contract portfolio to channels in the automata. However, a strategy will also consist of an ability to *delegate* a logical obligation to another principal of the contract portfolio. In Example 3.4.4 for instance, such a delegation will be used for fulfilling the commitment to send an MMS, as the service provider $\mathsf{P}$ is not able implement an automaton with alphabet $\mathcal{A}_{\lambda_3}$. That is, $\mathsf{P}$ does not have the hardware for sending the MMS.

We first define the mapping of logical links to channels, which also comprises delegation:

**Definition 3.5.1.** Let $\{c_1, \dots, c_n\}$ be a contract portfolio for a principal $\mathsf{P}$ with $c_i = (\Lambda_{\mathsf{PA}_i}, \Lambda_{\mathsf{A}_i\mathsf{P}}, G_i, g_i, \rho_i)$ for $i = 1, \dots, n$. A *routing* $r = (r_\mathrm{i}, r_\mathrm{o}, r_\mathrm{d})$ for $\{c_1, \dots, c_n\}$ and input–output channels $C_I/C_O$ consists of three functions:

$$\begin{aligned} r_\mathrm{i} &: C_I \to \Lambda_I, \\ r_\mathrm{o} &: C_O \to \Lambda_O, \\ r_\mathrm{d} &: \Lambda_I \setminus r_\mathrm{i}(C_I) \to \Lambda_O \setminus r_\mathrm{o}(C_O), \end{aligned}$$

where $\Lambda_I = \bigcup_{i=1}^{n} \Lambda_{\mathsf{A}_i\mathsf{P}}$ and $\Lambda_O = \bigcup_{i=1}^{n} \Lambda_{\mathsf{PA}_i}$. $r_\mathrm{i}$ and $r_\mathrm{o}$ realise physical communication by using the input–output channels $C_I/C_O$, while $r_\mathrm{d}$ realises communication by means of delegation to other principals. The routing must satisfy the following conditions:

(1) $r_\mathrm{i}$ and $r_\mathrm{o}$ are injective,

(2) $r_\mathrm{d}$ is bijective, and

(3) $\mathcal{A}_x = \mathcal{A}_{r_\mathrm{i}(x)}$ and $\mathcal{A}_y = \mathcal{A}_{r_\mathrm{o}(y)}$ for all channels $x \in C_I$ and $y \in C_O$, and $\mathcal{A}_z = \mathcal{A}_{r_\mathrm{d}(z)}$ for all links $z \in \Lambda_I \setminus r_\mathrm{i}(C_I)$.

(1) and (2) state that input channels and output channels must be mapped to exactly one logical link in the portfolio. (3) guarantees that only compatible channels and links are connected, that is the link alphabet must match the channel alphabet.

Rather than defining an implementation as a set of automata and a suitable routing, we only consider the case in which an implementation consists of a single automaton and a routing. This simplification is justified by the fact that multiple automata, which may interact with each other internally, can be described by a single automaton, compare Definition 3.3.2.

**Definition 3.5.2.** Let $C = \{c_1, \dots, c_n\}$ be a contract portfolio for a principal $\mathsf{P}$. An *implementation* $i = (\mathsf{a}, r)$ consists of an automaton $\mathsf{a} = (C_I, C_O, S, s_o, \delta_\mathrm{o}, \delta_\mathrm{t})$ and a routing $r$ for $C$ and $C_I/C_O$.

**Example 3.5.3** (Continuing Example 3.4.4)**.** An example of a legal implementation for the service provider $\mathsf{P}$ in Example 3.4.4 is $i = (\mathsf{a}, r)$, where:

$$\mathsf{a} = (\{\alpha\}, \{\beta, \gamma\}, S, s_0, \delta_\mathrm{o}, \delta_\mathrm{t}),$$

and $r = (r_\mathrm{i}, r_\mathrm{o}, r_\mathrm{d})$ is defined by:

$$r_\mathrm{i}(\alpha) = \lambda_1 \qquad r_\mathrm{o}(\gamma) = \lambda_2 \qquad r_\mathrm{o}(\beta) = \lambda_5 \qquad r_\mathrm{d}(\lambda_4) = \lambda_3.$$

The implementation is depicted in Figure 3.6.

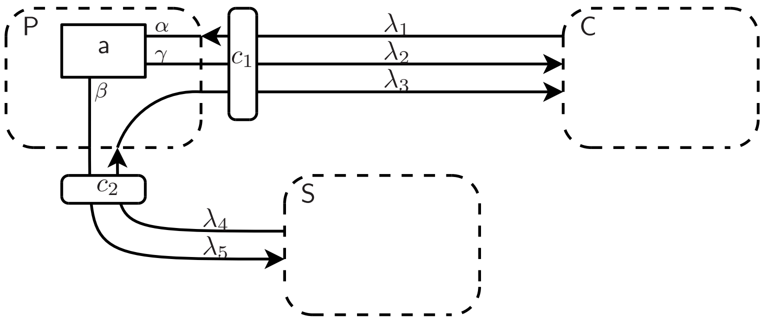


Figure 3.6: An implementation at principal P.

### 3.5.1 Contract Conformance

We are now in a position to define correctness, that is what it means for an implementation to satisfy a contract portfolio. Since contracts are generalised from a binary outcome, we define contract conformance as a guarantee related to the payoffs in the contract portfolio. Namely, we defined contract conformance as a guarantee of an all-time, non-negative, accumulated payoff.

This definition of contract conformance does not prevent us from verifying that an implementation will provide a certain positive profit $p$: in order to verify a profit of $p$ after $t$ time units, we simply add a "pseudo contract" to the portfolio that yields a payoff of $-p$ after $t$ time units. However, we cannot provide guarantees such as "the implementation will provide a profit of $p$ eventually", since contract conformance is a safety property.

Before we define contract conformance, we need the following auxiliary definition:

**Definition 3.5.4.** Let $r = (r_\mathrm{i}, r_\mathrm{o}, r_\mathrm{d})$ be a routing for input–output channels $C_I/C_O$ and contract portfolio $C = \{c_1, \dots, c_n\}$, with $c_i = (\Lambda_{\mathsf{PA}_i}, \Lambda_{\mathsf{A}_i\mathsf{P}}, G_i, g_i, \rho_i)$ for $i = 1, \dots, n$. Furthermore, let $\Lambda_I = \bigcup_{i=1}^{n} \Lambda_{\mathsf{A}_i\mathsf{P}}$ be the set of all incoming links from the contracts. We then define for each $i = 1, \dots, n$ the function $\overline{r}_i : \mathcal{M}_{C_O} \times \mathcal{M}_{\Lambda_I} \to \mathcal{M}_{\Lambda_{\mathsf{PA}_i}}$ by:

$$\overline{r}_i(m_{C_O}, m_{\Lambda_I})(\lambda) = \begin{cases} m_{C_O}(\alpha) & \text{if } r_\mathrm{o}(\alpha) = \lambda \text{ for some } \alpha \in C_O, \\ m_{\Lambda_I}(\lambda') & \text{if } r_\mathrm{d}(\lambda') = \lambda \text{ for some } \lambda' \in \Lambda_I \setminus r_\mathrm{o}(C_O). \end{cases}$$

The definition captures the intuition that some contractual output obligations may be handled by the automaton (the first case), while others may be delegated (the second case). Hence if opponent $\mathsf{A}_i$ has moved $m_i$ and the automaton has produced output $m$, then P's move in contract $c_i$ is $\overline{r}_i(\bigcup_{j=1}^{n} m_j, m)$. We can now define contract conformance:

**Definition 3.5.5.** Let $i = (\mathsf{a}, r)$ be an implementation for the contract portfolio $C = \{c_1, \dots, c_n\}$ with $\mathsf{a} = (C_I, C_O, S, s_0, \delta_\mathrm{o}, \delta_\mathrm{t})$, $r = (r_\mathrm{i}, r_\mathrm{o}, r_\mathrm{d})$, and $c_i = (\Lambda_{\mathsf{PA}_i}, \Lambda_{\mathsf{A}_i\mathsf{P}}, G_i, g_i, \rho_i)$ for $i = 1, \dots, n$. Furthermore, let $\Lambda_I = \bigcup_{i=1}^{n} \Lambda_{\mathsf{A}_i\mathsf{P}}$ be the set of all incoming contract links.

A relation $R \subseteq \mathbb{Q} \times S \times G_1 \times \ldots \times G_n$ is said to be a *conformance relation* for $i$ and $C$ iff the following holds for all payoffs $k \in \mathbb{Q}$, states $s \in S$, game states

$(g_1, \ldots, g_n) \in G_1 \times \cdots \times G_n$, and moves $m \in \mathcal{M}_{\Lambda_I}$:

$$\begin{aligned}&\text{if } (k, s, g_1, \ldots, g_n) \in R \text{ then } k' \geq k \text{ and } (k - k', \delta_{\mathrm{t}}(s, m \circ r_{\mathrm{i}}), g'_1, \ldots, g'_n) \in R,\\ &\qquad\qquad\text{where } (g'_i, k_i) = \rho_i(g_i, \overline{r}_i(\delta_{\mathrm{o}}(s), m), m_{|\Lambda_{\mathsf{A}_i\mathsf{P}}}) \text{ for } i = 1, \ldots, n,\\ &\qquad\qquad\text{and } k' = \sum_{i=1}^{n} k_i.\end{aligned}$$

The implementation $i$ is said to *conform* with contract portfolio $C$, written $\models i : C$, if $(0, s_0, g_1, \ldots, g_n) \in R$ for some conformance relation $R$.

The definition of $\models i : C$ formalises the guarantee of a consistently, non-negative, accumulated payoff. More generally, whenever $\mathsf{a}$ is in state $s$, contract $c_i$ is in state $g_i$, and $(k, s, g_1, \ldots, g_n) \in R$, where $R$ is a conformance relation for $i$ and $C$, then the accumulated payoff will remain at least $k$ throughout the remainder of the games. Note also how delegation is handled by the auxiliary functions $\overline{r}_i$ from Definition 3.5.4.

### 3.5.2 Automaton Contracts

The definition of contract conformance enables us to reason about implementations, when we know all the logical contracts that have been negotiated with other principals. However, we want to be able to reason about automata without having to worry about delegation and principals directly.

**Definition 3.5.6.** An *automaton contract* $\mathsf{c}$ is a 4-tuple $\mathsf{c} = (C, G, g_0, \rho)$, where $C$ is a finite set of channels, $G$ is a (potentially infinite) set of contract states, and $g_0 \in G$ is the start state. $\rho : G \times \mathcal{M}_C \to G \times \mathbb{Q}$ is the rule function for the game: when the contract is in a state $g$, and the move on $C$ is $m$, let $(g', k) = \rho(g, m)$; then $g'$ is the new contract state, and $k$ is the—possibly negative—incremental payoff.

The definition of automaton contracts is similar to the original definition of logical contracts, compare Definition 3.4.1. Automaton contracts can be seen as instances of logical contracts, in which logical links are renamed to physical channels. This means that the physical channels of the game need not be contained in the channels of the automaton, which is the reason why automaton contracts are not defined with respect to a particular automaton. Automaton contract conformance can now be defined in a similar manner as for contracts:

**Definition 3.5.7.** Let $\mathsf{a} = (C_I, C_O, S, s_0, \delta_{\mathrm{o}}, \delta_{\mathrm{t}})$ be an automaton and let $\mathsf{C} = \{\mathsf{c}_1, \ldots, \mathsf{c}_n\}$ be a set of automaton contracts with $\mathsf{c}_i = (C_i, G_i, g_i, \rho_i)$ for $i = 1, \ldots, n$. Furthermore, let $C = (C_I \cup \bigcup_{i=1}^{n} C_i) \setminus C_O$ be the set of all incoming channels and external channels.

A relation $R \subseteq \mathbb{Q} \times S \times G_1 \times \ldots \times G_n$ is said to be a *conformance relation* for $\mathsf{a}$ and $\mathsf{C}$ iff the following holds for all payoffs $k \in \mathbb{Q}$, states $s \in S$, game states $(g_1, \ldots, g_n) \in G_1 \times \cdots \times G_n$, and moves $m \in \mathcal{M}_C$:

$$\begin{aligned}&\text{if } (k, s, g_1, \ldots, g_n) \in R \text{ then } k' \geq k \text{ and } (k - k', \delta_{\mathrm{t}}(s, m_{|C_I}), g'_1, \ldots, g'_n) \in R,\\ &\qquad\qquad\text{where } (g'_i, k_i) = \rho_i(g_i, (m \cup \delta_{\mathrm{o}}(s))_{|C_i}) \text{ for } i = 1, \ldots, n,\\ &\qquad\qquad\text{and } k' = \sum_{i=1}^{n} k_i.\end{aligned}$$

| $s$ | $m$ | $\delta_t(s,m)$ | $\delta_o(s)$ |
|---|---|---|---|
| **start** | $\alpha \mapsto \text{req}(n,g)$ | **req_received**$(n,g)$ | |
| **start** | $-$ | **start** | |
| **req_received**$(n,g)$ | $-$ | **price_offered**$(n,g)$ | $\gamma \mapsto \text{offer}(1.5 * f(n,g))$ |
| **price_offered**$(n,g)$ | $\alpha \mapsto \text{reject}$ | **end** | |
| **price_offered**$(n,g)$ | $\alpha \mapsto \text{accept}$ | **accepted**$(n,g)$ | |
| **price_offered**$(n,g)$ | $-$ | **price_offered**$(n,g)$ | |
| **accepted**$(n,g)$ | $-$ | **end** | $\beta \mapsto \text{req}(n,g)$ |
| **end** | $-$ | **end** | |

Figure 3.7: Automaton transition functions.

The automaton $\mathsf{a}$ is said to *conform* with contract portfolio $\mathsf{C}$, written $\models \mathsf{a} : \mathsf{C}$, if $(0, s_0, g_1, \ldots, g_n) \in R$ for some conformance relation $R$.

Before we show the link between logical contracts and automaton contracts, we provide an example that illustrates automaton contracts and automaton contract conformance:

**Example 3.5.8** (Continuing Example 3.5.3)**.** Consider the two automaton contracts $\mathsf{c}_1 = (\{\alpha, \gamma, \delta\}, G_1, g_1, \rho'_1)$ and $\mathsf{c}_2 = (\{\beta, \delta\}, G_2, g_2, \rho'_2)$ obtained by replacing the logical links of the contracts in Example 3.5.3 with the following channels:

$$\lambda_1 \mapsto \alpha \qquad \lambda_3, \lambda_4 \mapsto \delta$$
$$\lambda_2 \mapsto \gamma \qquad \lambda_5 \mapsto \beta.$$

The definitions of $G_1$, $g_1$, $G_2$, and $g_2$ are identical to those of Example 3.4.4, and $\rho'_1$ is obtained by replacing the links of $\rho_1$ with the channels above, that is $\rho'_1(g, m) = \rho_1(g, m \circ \theta_{|\{\lambda_1\}}, m \circ \theta_{|\{\lambda_2,\lambda_3\}})$, where $\theta$ is the substitution above (and similar for $\rho'_2$).

The automaton $\mathsf{a} = (\{\alpha\}, \{\beta, \gamma\}, S, s_o, \delta_o, \delta_t)$ is defined by:

$$\begin{aligned} S = {} & \{\textbf{start}, \textbf{end}\} \\ & \cup \{\textbf{req\_received}(n,g) \mid n \in \mathcal{N} \text{ and } g \in \mathcal{G}\} \\ & \cup \{\textbf{price\_offered}(n,g) \mid n \in \mathcal{N} \text{ and } g \in \mathcal{G}\} \\ & \cup \{\textbf{accepted}(n,g) \mid n \in \mathcal{N} \text{ and } g \in \mathcal{G}\}, \\ s_0 = {} & \textbf{start}. \end{aligned}$$

The transition functions of $\mathsf{a}$ are represented in the tabular in Figure 3.7. For instance, if the automaton is in the state **start** and the input on $\alpha$ is $\text{req}(n,g)$ for some $n$ and $g$, then the next state is **req_received**$(n,g)$ with no output.

Note that $\mathsf{a}$ has nothing to do with the actual sending of MMSs—the automaton instead contacts $\mathsf{S}$ on channel $\beta$ to request the MMS, and directly thereafter stops by entering the **end** state. This delegation to $\mathsf{S}$ is represented by the channel $\delta$, which is not connected to $\mathsf{a}$.

We now wish to show that the automaton conforms with the portfolio $\{\mathsf{c}_1, \mathsf{c}_2\}$. In order to do so, we need to construct a conformance relation $R \subseteq \mathbb{Q} \times S \times G_1 \times G_2$ that contains the initial states $(0, \textbf{start}, \star, \star)$. But, in fact, such a conformance relation does not always exist—only if we assume that $t' < t - 1$. That is, we need that $\mathsf{S}$ guarantees to send the MMS before $t - 1$ time units, where $t$ is the guarantee negotiated with the client $\mathsf{C}$. Having this assumption we can now build a conformance relation (we write $p_{n,g}$ for $1.5 * f(n,g)$, we write $r_{n,g}$ for $0.5 * f(n,g)$, and in each set we have and implicit condition that $n \in \mathcal{N}$ and $g \in \mathcal{G}$):

$$
\begin{aligned}
R = {} & \{(0, \mathbf{start}, \star, \star)\} && (3.4)\\
\cup {} & \{(0, \mathbf{req\_received}(n,g), \langle n,g\rangle, \star)\} && (3.5)\\
\cup {} & \{(0, \mathbf{price\_offered}(n,g), \langle n,g,p_{n,g}\rangle, \star)\} && (3.6)\\
\cup {} & \{(0, \mathbf{end}, \dagger, \star)\} && (3.7)\\
\cup {} & \{(-p_{n,g}, \mathbf{accepted}(n,g), \langle n,g,p_{n,g},t\rangle, \star)\} && (3.8)\\
\cup {} & \{(-r_{n,g}, \mathbf{end}, \langle n,g,p_{n,g},t-k-1\rangle, \langle n,g,t'-k\rangle) \mid 0 \le k < t'\} && (3.9)\\
\cup {} & \{(-p_{n,g}-1, \mathbf{end}, \langle n,g,p_{n,g},t-t'-k-1\rangle, \dagger) \mid 0 \le k < t-t'-1\} && (3.10)\\
\cup {} & \{(-r_{n,g}, \mathbf{end}, \dagger, \langle n,g,t'-k\rangle) \mid 0 \le k < t'\} && (3.11)\\
\cup {} & \{(-r_{n,g}, \mathbf{end}, \dagger, \dagger)\} && (3.12)\\
\cup {} & \{(0, \mathbf{end}, \dagger, \dagger)\}. && (3.13)
\end{aligned}
$$

We will not show in detail that $R$ does indeed define a conformance relation, rather we show the dependencies that make $R$ a conformance relation below. An arrow $\boxed{A} \longrightarrow \boxed{B}$ means that $A \in R$ is a prerequisite for $B \in R$.

$$\begin{tikzcd}
\boxed{3.4} \arrow[loop left] & \boxed{3.5} \arrow[l] & \boxed{3.6} \arrow[l] \arrow[loop above] & \boxed{3.8} \arrow[l] & \boxed{3.9} \arrow[l] \arrow[loop above] & \boxed{3.10} \arrow[l] \arrow[loop right] \\
& & \boxed{3.7} \arrow[u] \arrow[loop below] & \boxed{3.11} \arrow[u] \arrow[loop below] & \boxed{3.12} \arrow[u] \arrow[l] \arrow[loop below] & \boxed{3.13} \arrow[u] \arrow[loop right]
\end{tikzcd}$$

When constructing a conformance relation such as the one above, situations that may normally be overseen are identified. For instance, the—very unlikely—event that $\mathsf{S}$ sends the MMS requested by $\mathsf{C}$ right after $\mathsf{C}$ has accepted the offer, but before $\mathsf{P}$ has requested $\mathsf{S}$ to do so, is handled by the conformance relation in (3.11). Notice also how the set (3.7) represents the case where $\mathsf{C}$ rejects $\mathsf{P}$'s offer, (3.12) represents the case of successful sending, and (3.13) represents the case of failure to send the MMS.

### 3.5.3 From Contracts to Automaton Contracts

In this section we show how to transform a portfolio of contracts to a portfolio of automaton contracts, in such a way that automaton conformance (Definition 3.5.7) with respect to the transformed portfolio implies conformance (Definition 3.5.5) with respect to the original portfolio. In particular, we get that the automaton conformance relation of Example 3.5.8 yields a conformance relation for the implementation in Example 3.5.3 with respect to the portfolio in Example 3.4.4.

The transformation from logical contracts to automaton contracts is straightforward: for each logical contract we define one automaton contract, the links that are routed to automaton channels are transformed directly to the corresponding channels, and logical delegations are transformed into new channels, similar to $\delta$ in Example 3.5.8.

**Definition 3.5.9.** Let $C = \{c_1, \dots, c_n\}$ be a contract portfolio for a principal $\mathsf{P}$, where $c_i = (\Lambda_{\mathsf{PA}_i}, \Lambda_{\mathsf{A}_i\mathsf{P}}, G_i, g_i, \rho_i)$ for $i = 1, \dots, n$, and let $\Lambda = \bigcup_{i=1}^{n} \Lambda_{\mathsf{PA}_i} \cup \bigcup \Lambda_{\mathsf{A}_i\mathsf{P}}$. Given a routing $r = (r_\mathrm{i}, r_\mathrm{o}, r_\mathrm{d})$ for $C$ and input–output channels $C_I/C_O$, we define a *renaming map* $\theta : \Lambda \to \mathcal{C}$ to be a function that satisfies the following for all channels $\alpha \in C_I$, $\beta \in C_O$ and links $\lambda_1, \lambda_2 \in \Lambda$:

(1) $\theta(r_\mathrm{i}(\alpha)) = \alpha$,

(2) $\theta(r_\mathrm{o}(\beta)) = \beta$,

(3) if $r_\mathrm{d}(\lambda_1) = \lambda_2$ then $\theta(\lambda_1) = \theta(\lambda_2)$, and

(4) if $\theta(\lambda_1) = \theta(\lambda_2)$ then either $\lambda_1 = \lambda_2$, $r_\mathrm{d}(\lambda_1) = \lambda_2$, or $r_\mathrm{d}(\lambda_2) = \lambda_1$.

Conditions (1–3) state that renaming must agree with the routing map, and condition (4) states that the renaming map must be injective modulo delegation.

**Definition 3.5.10.** Let $C = \{c_1, \dots, c_n\}$ be a contract portfolio for a principal $\mathsf{P}$ with $c_i = (\Lambda_{\mathsf{PA}_i}, \Lambda_{\mathsf{A}_i\mathsf{P}}, G_i, g_i, \rho_i)$ for $i = 1, \dots, n$. Given a routing $r = (r_\mathrm{i}, r_\mathrm{o}, r_\mathrm{d})$ for $C$ and input–output channels $C_I/C_O$, and a renaming map $\theta$ respecting $C$ and $r$, we define the *contract projection* to be $\{\mathsf{c}_1, \dots, \mathsf{c}_n\}$, where:

$$\mathsf{c}_i = (\theta(\Lambda_{\mathsf{PA}_i} \cup \Lambda_{\mathsf{A}_i\mathsf{P}}), G_i, g_i, \rho_i') \text{ for } i = 1, \dots, n,$$

with:

$$\rho_i'(g, m) = \rho_i(g, m \circ \theta_{|\Lambda_{\mathsf{PA}_i}}, m \circ \theta_{|\Lambda_{\mathsf{A}_i\mathsf{P}}}) \text{ for } i = 1, \dots, n.$$

We write $\{c_1, \dots, c_n\} \rightsquigarrow^{\theta} \{\mathsf{c}_1, \dots, \mathsf{c}_n\}$ for this projection.

**Observation 3.5.11.** Whenever $C \rightsquigarrow^{\theta} \mathsf{C}$ and $\theta$ is a renaming map for $r$, which is a routing for channels $C_I/C_O$, then each channel $\alpha \in C_I \cup C_O$ is mentioned in at least one contract in $\mathsf{C}$.

**Example 3.5.12** (Continuing Example 3.5.8)**.** We have, in fact, already seen an example of a logical contract portfolio being mapped to a set of automaton contracts. In Example 3.5.8 the rename map $\theta$ is defined by:

$$\theta(l) = \begin{cases} \alpha & \text{if } l = \lambda_1, \\ \gamma & \text{if } l = \lambda_2, \\ \delta & \text{if } l = \lambda_3 \text{ or } l = \lambda_4, \\ \beta & \text{if } l = \lambda_5. \end{cases}$$

The reader may check that $\theta$ does indeed define a legal renaming map with respect to the routing of Example 3.5.3. The channel $\delta$ is an example of a logical route that is mapped to a "fresh" channel that does not occur in the automaton of the implementation. This corresponds to labelling the "wire" of Figure 3.6 that is not connected to the automaton with $\delta$. With the notion of Definition 3.5.10 we therefore have $\{c_1, c_2\} \rightsquigarrow^{\theta} \{\mathsf{c}_1, \mathsf{c}_2\}$.

With the definition of contract projection we are able to state and prove soundness of the projection:

**Theorem 3.5.13.** *Let $C = \{c_1, \dots, c_n\}$ be a contract portfolio for principal $\mathsf{P}$ and let $r = (r_\mathrm{i}, r_\mathrm{o}, r_\mathrm{d})$ be a routing for $C$ and input–output channels $C_I/C_O$. Assume furthermore that $C \rightsquigarrow^\theta \mathsf{C}$, where $\theta$ is a renaming map for $r$. If $\models \mathsf{a} : \mathsf{C}$, for an automaton $\mathsf{a} = (C_I, C_O, S, s_0, \delta_\mathrm{o}, \delta_\mathrm{t})$, then $\models (\mathsf{a}, r) : C$.*

*Proof.* The proof is presented in Appendix C.1, page 200. □

The soundness result above together with the conformance relation built in Example 3.5.8, and the observation in Example 3.5.12, gives us that the implementation of the MMS greeting card service is indeed "good". That is, $\mathsf{P}$ is guaranteed never to lose money on the service.

## 3.5.4 Compositionality

We conclude this section with the main result about contract conformance, namely that it is compositional. The theorem enables us to prove conformance for a composed automaton by reasoning about the subautomata in isolation. Note that this form of compositionality is different from compositionality at the principal level, which is forced by the restriction to bilateral contracts.

**Definition 3.5.14.** For an automaton contract $\mathsf{c} = (C, G, g_o, \rho)$ the *dual automaton contract* $\overline{\mathsf{c}} = (C, G, g_0, \overline{\rho})$ is defined by $\overline{\rho}(g, m) = (g', -k)$, where $\rho(g, m) = (g', k)$.

**Theorem 3.5.15.** *Let $\mathsf{a}_1 = (C_I^1, C_O^1, S_1, s_0^1, \delta_\mathrm{o}^1, \delta_\mathrm{t}^1)$ and $\mathsf{a}_2 = (C_I^2, C_O^2, S_2, s_0^2, \delta_\mathrm{o}^2, \delta_\mathrm{t}^2)$ be two automata with parallel composition $\mathsf{a}_1 \parallel \mathsf{a}_2 = (C_I, C_O, S_1 \times S_2, \langle s_0^1, s_0^2 \rangle, \delta_\mathrm{o}, \delta_\mathrm{t})$ and internal channels $C_\mathrm{int}$ (compare Definition 3.3.2). If:*

$$\models \mathsf{a}_1 : \mathsf{c}_1, \dots, \mathsf{c}_n, \mathsf{c}_1', \dots, \mathsf{c}_{n_1}'$$
$$\models \mathsf{a}_2 : \overline{\mathsf{c}_1}, \dots, \overline{\mathsf{c}_n}, \mathsf{c}_1'', \dots, \mathsf{c}_{n_2}'',$$

*and the internal channels in $C_\mathrm{int}$ are not mentioned in the automaton contracts $\mathsf{c}_1', \dots, \mathsf{c}_{n_1}'$ and $\mathsf{c}_1'', \dots, \mathsf{c}_{n_2}$, then:*

$$\models \mathsf{a}_1 \parallel \mathsf{a}_2 : \mathsf{c}_1', \dots, \mathsf{c}_{n_1}', \mathsf{c}_1'', \dots, \mathsf{c}_{n_2}''.$$

*Proof.* The proof is presented in Appendix C.1, page 201. □

The theorem shows how to fulfil a set of contracts by splitting the obligations between two *contractually compatible* automata. If the set of internal contracts—in the theorem they are written $\mathsf{c}_1, \dots, \mathsf{c}_n$—is empty, then the theorem simply says that two disjoint automata can fulfil a set of contracts by partitioning the set between them. If the set of internal contracts is non-empty, then the contracts express how the automata can communicate internally to fulfil the external obligations. The duality expresses that they must play opposite roles in the internal contracts.

A special and important case is the one where we seek to fulfil one contract $\mathsf{c}$, but we subdivide it into smaller contracts $\mathsf{c}_1, \dots, \mathsf{c}_n$. We can then write automata for each of the smaller contracts, and combine them via an "orchestrator" automaton that communicates with the subautomata (the orchestrator must then conform with $\mathsf{c}, \overline{\mathsf{c}_1}, \dots, \overline{\mathsf{c}_n}$). Parallel composition of the subautomata and the orchestrator is then guaranteed to conform with the original contract.

The theorem can be seen as a generalisation of two rules of composition for sequential Hoare triples (we use the same notation as Winskel [119]):

$$\text{if } \models \{A\}\, c_1\, \{C\} \text{ and } \models \{C\}\, c_2\, \{B\} \text{ then } \models \{A\}\, c_1; c_2\, \{B\}, \tag{3.14}$$

$$\begin{aligned}&\text{if } \models \{A_1\}\, c_1\, \{B_1\} \text{ and } \models \{A_2\}\, c_2\, \{B_2\} \text{ and}\\ &\quad \models \{A_2\}\, c_1\, \{A_2\} \text{ and } \models \{B_1\}\, c_2\, \{B_1\}\\ &\text{then } \models \{A_1 \wedge A_2\}\, c_1; c_2\, \{B_1 \wedge B_2\}.\end{aligned} \tag{3.15}$$

The first rule (3.14) corresponds to the second case explained above, where internal contracts are utilised to fulfil a contract—the assertion $C$ can be interpreted as the internal contract. The second rule (3.15) corresponds to the first case explained above, where the two automata have no internal contracts, which means that the two commands above do not interfere.

**Example 3.5.16.** We conclude this subsection with an example that illustrates the compositionality theorem. Consider two automata $\mathsf{a}_1$ (a "doubler") and $\mathsf{a}_2$ (an "incrementer"), which are constructed via the translation $\llcorner\cdot\lrcorner$ in Definition C.3.10 from the processes in Example 3.2.5. We want to show that the two automata in parallel produce an ever growing list of integers on the form $f^0(1), f^1(1), f^2(1), \ldots$, where $f(x) = 2x + 1$. We require that the integers must be sent with a delay of at most 10 time units after the previous result. This requirement can be captured by the following contract:

$$\begin{aligned}
\mathsf{c} &= (\{\gamma\}, G, g_0, \rho),\\
\mathcal{A}_\gamma &= \mathbb{N} \cup \{\varepsilon\},\\
G &= \{\langle n, t\rangle \mid n \in \mathbb{N} \text{ and } t \in \mathbb{N}\} \cup \{\mathtt{stop}\},\\
g_0 &= \langle 1, 10\rangle,\\
\rho(\langle n, 1\rangle, m) &= \begin{cases} (\langle 2n+1, 10\rangle, 0) & \text{if } m(\gamma) = n,\\ (\mathtt{stop}, -1) & \text{otherwise,}\end{cases}\\
\rho(\langle n, t+1\rangle, m) &= \begin{cases} (\langle n, t\rangle, 0) & \text{if } m(\gamma) = \varepsilon,\\ (\langle 2n+1, 10\rangle, 0) & \text{if } m(\gamma) = n,\\ (\mathtt{stop}, -1) & \text{if } m(\gamma) \neq n,\end{cases}\\
\rho(\mathtt{stop}, m) &= \ (\mathtt{stop}, 0).
\end{aligned}$$

In order to show that $\mathsf{a}_1 \parallel \mathsf{a}_2$ fulfils this contract, we write an "incrementer" contract for $\mathsf{a}_2$, where $\mathsf{a}_2$ is given 5 time units to produce its output after receiving an input. This means that $\mathsf{a}_1$ will have time to calculate the "doubling" function as well:

$$\begin{aligned}
\mathsf{c}_{\text{inc}} &= (\{\alpha, \beta, \gamma\}, G, g_0, \rho),\\
\mathcal{A}_c &= \mathbb{N} \cup \{\varepsilon\},\\
G &= \{\mathtt{wait}\} \cup \{\langle n, t\rangle \mid n \in \mathbb{N} \text{ and } t \in \mathbb{N}\} \cup \{\mathtt{stop}\},\\
g_0 &= \mathtt{wait},\\
\rho(\mathtt{wait}, m) &= \begin{cases} (\mathtt{stop}, -1) & \text{if } m(\alpha) \neq \varepsilon \text{ or } m(\gamma) \neq \varepsilon,\\ (\mathtt{wait}, 0) & \text{if } m(\beta) = \varepsilon,\\ (\langle n+1, 5\rangle, 0) & \text{if } m(\beta) = n,\end{cases}
\end{aligned}$$

$$\rho(\langle n,1\rangle,m)=\begin{cases}(\texttt{stop},1) & \text{if } m(\beta)\neq\varepsilon,\\(\texttt{wait},0) & \text{if } m(\alpha)=m(\gamma)=n,\\(\texttt{stop},-1) & \text{otherwise,}\end{cases}$$
$$\rho(\langle n,t+1\rangle,m)=\begin{cases}(\texttt{stop},1) & m(\beta)\neq\varepsilon,\\(\langle n,t\rangle,0) & \text{if } m(\alpha)=m(\gamma)=\varepsilon,\\(\texttt{wait},0) & \text{if } m(\alpha)=m(\gamma)=n,\\(\texttt{stop},-1) & \text{otherwise,}\end{cases}$$
$$\rho(\texttt{stop})=(\texttt{stop},0).$$

It is fairly straightforward, but tedious, to show that $\models \mathsf{a}_2 : \mathsf{c}_{\text{inc}}$. The remaining obligation is to show that $\models \mathsf{a}_1 : \overline{\mathsf{c}_{\text{inc}}}, \mathsf{c}$ which we also omit here. By Theorem 3.5.15 with $C_{\text{int}} = \{\alpha, \beta\}$ it then follows that $\models \mathsf{a}_1 \parallel \mathsf{a}_2 : \mathsf{c}$.

## 3.6 Conclusion

We have developed and described a theory for extending programming-by-contract (PBC) to a distributed and concurrent environment. The main contribution of our work is a shift from cooperative, intracompany decomposition of a contract to an adversarial model of composition with different parties. This shift has sparked a game-theoretic view of contracts, and by means of a generalised payoff measure we are able to model for instance quality of service, degrees of fulfilment, local optimisation, etc. (see the progress report [49] for concrete examples).

Our work is, admittedly, very foundational: we use abstract automata to model both communication and contracts. In order to be applied in practice, our ideas need to be transferred into a setting in which communication is described at a higher level (programming language), and a suitable abstraction for contracts must be developed as well (a contract language). The progress report [49] contains additional material that targets this question: we show, for instance, how to construct an ad hoc, preliminary contract language to more concisely describe PBC contracts, and we show how a restricted form of session types [45] extended with time can be captured in our model. We omit the details here, because we believe our model is not realistic as a practical communication model and contract model—we see instead our model as a means of presenting our ideas concretely.

### 3.6.1 Related Work

Much work has been done previously in the context of design-by-contract, concurrency, and distributed systems. In this section we briefly describe some of this work, and how it relates to our approach.

The original inspiration for our work is traditional programming-by-contract for sequential programs, or more specifically precondition- and postcondition-style contracts:

$$\{A\}\, c\, \{B\}.$$

In fact, Hoare triple validity is not too far from our interpretation of a contract as a two-person game with payoffs. In order to “win” the game $\{A\}\, c\, \{B\}$, the player (implementor) must construct a program fragment $c$ (strategy) such that the store produced by $c$ satisfies $B$ whenever the initial store (the move of the opponent) satisfies $A$. If $c$ is a winning strategy, then the triple is valid. A similar observation is made by Wadler and Findler [115], who introduce the terminology "The Blame Game" (although Wadler and Findler do not consider general programming-by-contract, but rather types as contracts).

This game-theoretic interpretation of contracts is also closely related to the theory of game semantics [22] introduced by Abramsky and Jagadeesan [2]. In game semantics, a program $c$ again denotes a strategy for playing a game against an opponent. The game then specifies a type, such as **Int** or **Int** $\rightarrow$ **Int**, and $c$ is a winning strategy exactly when it is well-typed. Hence the type is the contract, and type checking corresponds to our notion of conformance.

In object-oriented programming, specification of distributed programs has been investigated. Exton and Chen [23] consider methods for specifying interfaces for remote method invocations. As in our model, the internal state must be hidden in the interface. However, this is from a code-encapsulation viewpoint rather from the viewpoint of different administrative principals. Helm et al. introduce the contract language Contracts [40] in order to specify the behaviour of compositions of objects. This includes interface contracts (variables, methods) as well as causal obligations. The latter makes it possible to specify that a series of actions must be taken in response to some event. Hence, a concrete class implementation can only conform with the contract if it implements the interface and respects the causal obligations.

Another inspiration for our work is session types [45]. Like our model, session types are concerned with distributed communication. There are, however, several important differences. Being based on the $\pi$-calculus, session-types rely on a higher level of communication than our model. Moreover, with session types it is not possible to specify absolute timing guarantees, which we argued is necessary due to the adversarial nature of distributed computing. On the other hand, session types support dynamically created channels, which we have postponed to future work.

In the more general area of concurrent programming and compositional reasoning, Jones [54] provides an overview of what has been done, and—perhaps more interestingly—what remains to be done in order to be applicable in practice. The focus in this area is more on capturing the behaviour of concurrently running processes than on distribution. An example in this area is the work by Hooman [47], in which Hoare logic is extended to real-time systems.

### 3.6.2 Future Work

The model of distributed programming-by-contract that we have presented in this work is far from complete. However, we hope that with our model we have shed light on some problematic aspects that, to our knowledge, have not been presented previously. In order for our approach to be useful in practice, several directions for future work are needed. These include: extending the model to a dynamic network topology, considering a refined model of time in which all peers need not be in sync, and investigate better abstractions (languages) for writing contracts and implementations.

Other possible directions for future work include: pursue a definition of certified code for distributed systems in the style of proof-carrying-code [76]; a further investigation of the relation between I/O automata and contract automata; and contract refinement in the style of Back and von Wright [9], that is a refinement relation on contracts. Finally, as mentioned in the introduction, it should be possible to generalise our model of contracts from bilateral contract to multiparty contracts. However, one reason for restricting ourselves to bilateral contracts is the forced prevention of unintended blame propagation: it would be wrong, for instance, to encode Example 3.4.4 as a three-party contract!

## Part II

# Modular Implementation of Domain-Specific Languages

# Chapter 4

# Compositional Data Types[⋆]

**Abstract**

Building on Wouter Swierstra's *data types à la carte*, we present a comprehensive Haskell library of *compositional data types* suitable for practical applications. In this framework, data types and functions on them can be defined in a modular fashion. We extend the existing work by implementing a wide array of recursion schemes including monadic computations. Above all, we generalise recursive data types to *contexts*, which allow us to characterise a special yet frequent kind of catamorphisms. The thus established notion of *term homomorphisms* allows for flexible reuse and enables fusion-style deforestation which yields considerable speedups. We demonstrate our framework in the setting of compiler construction, and moreover, we compare compositional data types with generic programming techniques and show that both are comparable in run-time performance and expressivity, while our approach allows for stricter types. We substantiate this conclusion by lifting compositional data types to mutually recursive data types and generalised algebraic data types. Lastly, we compare the run-time performance of our techniques with traditional implementations over algebraic data types. The results are surprisingly good.

## 4.1 Introduction

Static typing provides a valuable tool for expressing invariants of a program. Yet, all too often, this tool is not leveraged to its full extent because it is simply not practical. Vice versa, if we want to use the full power of a type system, we often find ourselves writing large chunks of boilerplate code or—even worse—duplicating code. For example, consider the type of non-empty lists. Even though having such a type at your disposal is quite useful, you would rarely find it in use since—in a practical type system such as Haskell's—it would require the duplication of functions that work both on general and non-empty lists.

The situation illustrated above is an ubiquitous issue in compiler construction. In a compiler, an abstract syntax tree (AST) is produced from a source file, which then goes through different transformation and analysis phases, and is finally transformed into the target code. As functional programmers, we want to reflect the changes of each transformation step in the type of the AST. For example, consider the desugaring phase of a compiler that reduces syntactic sugar to the core syntax of

[⋆] *Joint work with Patrick Bahr [11].*

the object language. To properly reflect this structural change also in the types, we have to create and maintain a variant of the data type defining the AST for the core syntax. Then, however, functions have to be defined for both types independently, that is code cannot be readily reused for both types! If we add annotations in an analysis step of the compiler, the type of the AST has to be changed again. But some functions should ignore certain annotations while being aware of others. And it gets even worse if we allow extensions to the object language that can be turned on and off independently, or if we want to implement several domain-specific languages that share a common core. This quickly becomes a nightmare with the choice of either duplicating lots of code or giving up static type safety by using a huge AST data type that covers all cases.

The essence of this problem can be summarised as the *Expression Problem*: "the goal [...] to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety" [114]. Wouter Swierstra [104] elegantly addressed this problem using Haskell and its type classes machinery. While Swierstra's approach exhibits invaluable simplicity and clarity, it lacks abilities necessary to apply it in a practical setting beyond the confined simplicity of the expression problem.

The goal of this paper is to extend Swierstra's work in order to enhance its flexibility, improve its performance, and broaden its scope of applications. In concrete terms, our contributions are:

- We implement recursion schemes other than catamorphisms (Section 4.4.5) and also recursion schemes over *monadic computations* (Section 4.3.2).
- We show how *generic programming* techniques can be efficiently implemented on top of the compositional data types framework (Section 4.3.1), providing a performance competitive with top-performing dedicated generic programming libraries.
- By generalising terms—that is, recursive data types—to contexts—that is, recursive data types with holes—we are able to capture the notion of term homomorphisms (Section 4.4.4), a special but common case of term algebras. In contrast to general algebras, term homomorphisms can easily be lifted to different data types, readily reused, and composed (also with algebras). The latter allows us to perform optimisation via fusion rules that provide considerable speedups (Section 4.6.2).
- We further extend the scope of applications by capturing compositional mutually recursive data types and GADTs via the construction of Johann and Ghani [53] (Section 4.5).
- Finally, we show the practical competitiveness of compositional data types by reducing their syntactic overhead using Template Haskell [99] (Section 4.6.1), and by comparing the run-time of typical functions with corresponding implementations over ordinary recursive data types (Section 4.6.2).

The framework of compositional data types that we present here is available from Hackage[1]. It contains the complete source code, numerous examples, and the

[1]See http://hackage.haskell.org/package/compdata.

benchmarks whose results we present in this paper. All code fragments presented throughout the paper are written in Haskell [62].

## 4.2 Data Types à la Carte

This section serves as an introduction to Swierstra's *data types à la carte* [104] (from here on, *compositional data types*), using our slightly revised notation and terminology. We demonstrate the application of compositional data types to a setting consisting of a family of expression languages that pairwise share some sublanguage, and operations that provide transformations between some of them. We illustrate the merits of this method on two examples: expression evaluation and desugaring.

### 4.2.1 Evaluating Expressions

Consider a simple language of expressions over integers and pairs, together with an evaluation function:

```
data Exp   = Const Int | Mult Exp Exp | Pair Exp Exp | Fst Exp | Snd Exp
data Value = VConst Int | VPair Value Value
eval :: Exp → Value
eval (Const n)  = VConst n
eval (Mult x y) = let (VConst m, VConst n) = (eval x, eval y)
                  in VConst (m ∗ n)
eval (Pair x y) = VPair (eval x) (eval y)
eval (Fst x)    = let VPair v _ = eval x in v
eval (Snd x)    = let VPair _ v = eval x in v
```

In order to statically guarantee that the evaluation function produces values—a sublanguage of the expression language—we are forced to replicate parts of the expression structure in order to represent values. Consequently, we are also forced to duplicate common functionality such as pretty printing. Compositional data types provide a solution to this problem by relying on the well-known technique [67] of separating the recursive structure of terms from their signatures (functors). Recursive functions, in the form of catamorphisms, can then be specified by algebras on these signatures.

For our example, it suffices to define the following two signatures in order to separate values from general expressions:

```
data Val a = Const Int | Pair a a
data Op a  = Mult a a | Fst a | Snd a
```

The novelty of compositional data types then is to combine signatures—and algebras defined on them—in a modular fashion, by means of a formal sum of functors:

```
data (f :+: g) a = Inl (f a) | Inr (g a)
```

It is easy to show that $f$ :+: $g$ is a functor whenever $f$ and $g$ are functors. We thus obtain the combined signature for expressions:

```
type Sig = Op :+: Val
```

Finally, the type of terms over a (potentially compound) signature $f$ can be constructed as the fixed point of the signature $f$:

```
data Term f = In { out :: (f (Term f)) }
```

We then have that *Term Sig* $\cong$ *Exp* and *Term Val* $\cong$ *Value*.[2]

However, using compound signatures constructed by formal sums means that we have to explicitly tag constructors with the right injections. For instance, the term $1 * 2$ must be written:

```
e :: Term Sig
e = In $ Inl $ Mult (In $ Inr $ Const 1) (In $ Inr $ Const 2)
```

Even worse, if we want to embed the term $e$ into a type over an extended signature, say with syntactic sugar, then we have to add another level of injections *throughout* its definition. To overcome this problem, injections are derived using a type class:

```
class sub :≺: sup where
  inj  :: sub a → sup a
  proj :: sup a → Maybe (sub a)
```

Using *overlapping instance* declarations, the sub-signature relation $:\prec:$ can be constructively defined. However, due to restrictions of the type class system, we have to restrict ourselves to instances of the form $f :\prec: g$ where $f$ is atomic, that is not a sum, and $g$ is a right-associated sum, for instance $g_1 :+: (g_2 :+: g_3)$ but not $(g_1 :+: g_2) :+: g_3$.[3] Using the carefully defined instances for $:\prec:$, we can then define injection and projection functions:

```
inject :: (g :≺: f) ⇒ g (Term f) → Term f
inject = In . inj
project :: (g :≺: f) ⇒ Term f → Maybe (g (Term f))
project = proj . out
```

Additionally, in order to reduce the syntactic overhead, we use smart constructors—which can be derived automatically, see Section 4.6.1—that already comprise the injections:

```
iMult :: (Op :≺: f) ⇒ Term f → Term f → Term f
iMult x y = inject (Mult x y)
```

The term $1 * 2$ can now be written without syntactic overhead:

```
e :: Term Sig
e = iConst 1 `iMult` iConst 2
```

[2]For clarity, we have omitted the strictness annotation to the constructor *In* which is necessary in order to obtain the indicated isomorphisms.

[3]We encourage the reader to consult Swierstra's original paper [104] for the proper definition of the $:\prec:$ relation.

We can even give $e$ the *open* type $(Val :\prec: f, Op :\prec: f) \Rightarrow Term\ f$. That is, $e$ can be used as a term over any signature containing at least values and operators.

Next, we want to define the evaluation function, that is a function of the type $Term\ Sig \rightarrow Term\ Val$. To this end, we define the following *algebra class Eval*:

```
type Alg f a = f a → a
class Eval f v where
  evalAlg :: Alg f (Term v)
instance (Eval f v, Eval g v) ⇒ Eval (f :+: g) v where
  evalAlg (Inl x) = evalAlg x
  evalAlg (Inr x) = evalAlg x
```

The instance declaration for sums is crucial, as it defines how to combine instances for the different signatures—yet the structure of its declaration is independent from the particular algebra class, and it can be automatically derived for any algebra. Thus, we will omit the instance declarations lifting algebras to sums from now on. The actual evaluation function can then be obtained from instances of this algebra class as a *catamorphism*. In order to perform the necessary recursion, we require the signature $f$ to be an instance of *Functor*, providing the method $fmap :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$:

```
cata :: Functor f ⇒ Alg f a → Term f → a
cata φ = φ . fmap (cata φ) . out
eval :: (Functor f, Eval f v) ⇒ Term f → Term v
eval = cata evalAlg
```

What remains is to define the algebra instances for *Val* and *Op*. One approach is to define instances *Eval Val Val* and *Eval Op Val*. However, such definitions are problematic if we later want to add a signature to the language that also extends the signature for values, say with Boolean values. We could hope to achieve such extensibility by defining an instance:

```
instance (Eval f v, v :≺: v') ⇒ Eval f v'
```

But this is problematic for two reasons. First, the relation $:\prec:$ only works for atomic left-hand sides, and second, we can in fact not define this instance because the function $evalAlg :: f\ (Term\ v) \rightarrow Term\ v$ cannot be lifted to the type $f\ (Term\ v') \rightarrow Term\ v'$, as the type of the domain also changes. Instead, the correct approach is to leave the instance declarations open in the target signature:

```
instance (Val :≺: v) ⇒ Eval Val v where
  evalAlg = inject
instance (Val :≺: v) ⇒ Eval Op v where
  evalAlg (Mult x y) = iConst (projC x ∗ projC y)
  evalAlg (Fst x)    = fst (projP x)
  evalAlg (Snd x)    = snd (projP x)
projC :: (Val :≺: v) ⇒ Term v → Int
projC v = case project v of Just (Const n) → n
```

```
projP :: (Val :≺: v) ⇒ Term v → (Term v, Term v)
projP v = case project v of Just (Pair x y) → (x, y)
```

Notice how the constructors *Const* and *Pair* are treated with a single *inject*, as these are already part of the value signature.

### 4.2.2 Adding Sugar on Top

We now consider an extension of the expression language with *syntactic sugar*, exemplified via negation and swapping of pairs:

```
data Sug a = Neg a | Swap a
type Sig'   = Sug :+: Sig
```

Defining a desugaring function $Term\ Sig' \to Term\ Sig$ then amounts to instantiating the following algebra class:

```
class Desug f g where
  desugAlg :: Alg f (Term g)
desug :: (Functor f, Desug f g) ⇒ Term f → Term g
desug = cata desugAlg
```

Using overlapping instances, we can define a default translation for *Val* and *Op*, so we only have to write the "interesting" cases:

```
instance (f :≺: g) ⇒ Desug f g where
  desugAlg = inject
instance (Val :≺: f, Op :≺: f) ⇒ Desug Sug f where
  desugAlg (Neg x)  = iConst (−1) `iMult` x
  desugAlg (Swap x) = iSnd x `iPair` iFst x
```

Note how the context of the last instance reveals that desugaring of the extended syntax requires a target signature with at least base values, *Val* :≺: *f*, and operators, *Op*:≺:*f*. By composing *desug* and *eval*, we get an evaluation function for the extended language:

```
eval' :: Term Sig' → Term Val
eval' = eval . (desug :: Term Sig' → Term Sig)
```

The definition above shows that there is a small price to pay for leaving the algebra instances open: we have to annotate the desugaring function in order to pin down the intermediate signature *Sig*.

## 4.3 Extensions

In this section, we introduce some rather straightforward extensions to the compositional data types framework: generic programming combinators, monadic computations, and annotations.

### 4.3.1 Generic Programming

Most of the functions that are definable in the common generic programming frameworks [94] can be categorised as either query functions $d \to r$, which analyse a data structure of type $d$ by extracting some relevant information of type $r$ from parts of the input and compose them, or as transformation functions $d \to d$, which recursively apply some type preserving functions to parts of the input. The benefit that generic programming frameworks offer is that programmers only need to specify the "interesting" parts of the computation. We will show how we can easily reproduce this experience on top of compositional data types.

Applying a type-preserving function recursively throughout a term can be implemented easily. The function below applies a given function in a bottom-up manner:

```
trans :: Functor f ⇒ (Term f → Term f) → Term f → Term f
trans f = cata (f . In)
```

Other recursion schemes can be implemented just as easily.

In order to implement generic querying functions, we need a means to combine the result of querying a functorial value. The standard type class *Foldable* generalises folds over lists and thus provides us with exactly the interface we need:[4]

```
class Foldable f where
  foldl :: (a → b → a) → a → f b → a
```

For example, an appropriate instance for the functor *Val* can be defined like this:

```
instance Foldable Val where
  foldl _ a (Const _) = a
  foldl f a (Pair x y) = (a ‘f‘ x) ‘f‘ y
```

With *Foldable*, a generic querying function can be implemented easily. It takes a function $q :: Term\ f \to r$ to query a single node of the term and a function $c :: r \to r \to r$ to combine two results:

```
query :: Foldable f ⇒ (Term f → r) → (r → r → r) → Term f → r
query q c t = foldl (λr x → r ‘c‘ query q c x) (q t) (out t)
```

We can instantiate this scheme, for example, to implement a generic size function:

```
gsize :: Foldable f ⇒ Term f → Int
gsize = query (const 1) (+)
```

A very convenient scheme of query functions introduced by Mitchell and Runciman [73], in the form of the *universe* combinator, simply returns a list of all subterms. Specific queries can then be written rather succinctly using list comprehensions. Such a combinator can be implemented easily via *query*:

```
subs :: Foldable f ⇒ Term f → [Term f]
subs = query (λx → [x]) (++)
```

[4] *Foldable* also has other fold functions, but they are derivable from *foldl* and are not relevant for our purposes.

However, in order to make the pattern matching in list comprehensions work, we need to project the terms to the functor that contains the constructor we want to match against:

$$\begin{aligned}&\mathit{subs}' :: (\mathit{Foldable}\ f, g :\prec: f) \Rightarrow \mathit{Term}\ f \rightarrow [g\ (\mathit{Term}\ f)]\\ &\mathit{subs}' = \mathit{mapMaybe}\ \mathit{project}\ .\ \mathit{subs}\end{aligned}$$

With this in place we can for example easily sum up all integer literals in an expression:

$$\begin{aligned}&\mathit{sumInts} :: (\mathit{Val} :\prec: f) \Rightarrow \mathit{Term}\ f \rightarrow \mathit{Int}\\ &\mathit{sumInts}\ t = \mathit{sum}\ [i \mid \mathit{Const}\ i \leftarrow \mathit{subs}'\ t]\end{aligned}$$

This shows that we can obtain functionality similar to what dedicated generic programming frameworks offer. In contrast to generic programming, however, the compositional data type approach provides additional tools that allow us to define functions with a stricter type that reflects the underlying transformation. For example, we could have defined the desugaring function in terms of *trans*, but that would have resulted in the "weaker" type $\mathit{Term}\ \mathit{Sig}' \rightarrow \mathit{Term}\ \mathit{Sig}'$ instead of $\mathit{Term}\ \mathit{Sig}' \rightarrow \mathit{Term}\ \mathit{Sig}$. The latter type witnesses that indeed all syntactic sugar is removed!

Nevertheless, the examples show that at least the querying combinators *query* and $\mathit{subs}'$ provide an added value to our framework. Moreover, by applying standard optimisation techniques we can obtain run-time performance comparable with top-performing generic programming libraries (compare Section 4.6.2). In contrast to common generic programming libraries [94], we only considered combinators that work on a single recursive data type. This restriction is lifted in Section 4.5 when we move to mutually recursive data types.

### 4.3.2 Monadic Computations

We saw in Section 4.2 how to realise a modular evaluation function for a small expression language in terms of catamorphisms defined by algebras. In order to deal with type mismatches, we employed non-exhaustive case expressions. Clearly, it would be better to use a monad instead. However, a monadic carrier type $m\ a$ would yield an algebra $f\ (m\ a) \rightarrow m\ a$, which means that we have to explicitly sequence the nested monadic values of the argument. What we would rather like to do is to write a *monadic algebra* [27]:

$$\textbf{type}\ \mathit{AlgM}\ m\ f\ a = f\ a \rightarrow m\ a$$

Here the nested sequencing is done automatically and thus the monadic type only occurs in the codomain. Again we are looking for a function that we already know from lists:

$$\mathit{sequence} :: \mathit{Monad}\ m \Rightarrow [m\ a] \rightarrow m\ [a]$$

The standard type class *Traversable* [63] provides the appropriate generalisation to functors:

```
class (Functor f, Foldable f) ⇒ Traversable f where
  sequence :: Monad m ⇒ f (m a) → m (f a)
  mapM     :: Monad m ⇒ (a → m b) → f a → m (f b)
```

Here, *mapM* is simply the composition of *sequence* and *fmap*.

The definition of a monadic variant of catamorphisms can then be derived by replacing *fmap* with *mapM* and function composition with monadic function composition ⋘:

```
cataM :: (Traversable f, Monad m) ⇒ AlgM m f a → Term f → m a
cataM φ = φ ⋘ mapM (cataM φ) . out
```

The following definitions illustrate how monadic catamorphisms can be used to define a *safe* version of the evaluation function from Section 4.2, which properly handles errors when applied to a *bad term* (using the *Maybe* monad for simplicity):

```
class EvalM f v where
  evalAlgM :: AlgM Maybe f (Term v)
evalM :: (Traversable f, EvalM f v) ⇒ Term f → Maybe (Term v)
evalM = cataM evalAlgM
instance (Val :≺: v) ⇒ EvalM Val v where
  evalAlgM = return . inject
instance (Val :≺: v) ⇒ EvalM Op v where
  evalAlgM (Mult x y) = liftM iConst (liftM2 (∗) (projCM x) (projCM y))
  evalAlgM (Fst x)    = liftM fst (projPM x)
  evalAlgM (Snd x)    = liftM snd (projPM x)
projCM :: (Val :≺: v) ⇒ Term v → Maybe Int
projCM v = case project v of Just (Const n) → return n
                             _              → Nothing
projPM :: (Val :≺: v) ⇒ Term v → Maybe (Term v, Term v)
projPM v = case project v of Just (Pair x y) → return (x, y)
                             _               → Nothing
```

### 4.3.3 Products and Annotations

We have seen in Section 4.2 how the sum :+: can be used to combine signatures. This inevitably leads to the dual construction:

```
data (f :∗: g) a = f a :∗: g a
```

In its general form, the product :∗: seems of little use: each constructor of $f$ can be paired with each constructor of $g$. The special case, however, where $g$ is a constant functor, is easy to comprehend yet immensely useful:

```
data (f :&: c) a = f a :&: c
```

Now, every value of type $(f$ :&: $c)$ $a$ is value from $f$ $a$ annotated with a value in $c$. On the term level, this means that a term over $f$ :&: $c$ is a term over $f$ in which each subterm is annotated with a value in $c$.

This addresses a common problem in compiler implementations: how to deal with annotations of AST nodes such as source positions or type information, which have only a limited lifespan or are only of interest for some parts of the compiler?

Given the signature *Sig* for our simple expression language and a type *Pos* that represents source position information such as a file name and a line number, we can represent ASTs with source position annotations as *Term* (*Sig* :&: *Pos*) and write a parser that provides such annotations [108].

The resulting representation yields a clean separation between the actual data—the AST—and the annotation data—the source positions—which is purely supplemental for supplying better error messages. The separation allows us to write a generic function that strips off annotations when they are not needed:

```
remA :: (f :&: c) a → f a
remA (v :&: _) = v
stripA :: Functor f ⇒ Term (f :&: c) → Term f
stripA = cata (In . remA)
```

With this in place, we can provide a generic combinator that lifts a function on terms to a function on terms with annotations:

```
liftA :: Functor f ⇒ (Term f → t) → Term (f :&: c) → t
liftA f = f . stripA
```

This works for instance for the evaluation function:

```
liftA eval :: Term (Sig :&: Pos) → Term Val
```

But how do we actually define an algebra that uses the position annotations? We are faced with the problem that the product :&: is applied to a *sum*, viz. *Sig* = *Op* :+: *Val*. When defining the algebra for one of the summands, say *Val*, we do not have immediate access to the factor *Pos*, which is outside of the sum.

We can solve this issue in two ways: (a) propagating the annotation using a *Reader* monad or (b) providing operations that allow us to make use of the right-distributivity of :&: over :+:. For the first approach, we only need to move from algebras *Alg f a* to monadic algebras *AlgM* (*Reader c*) *f a*, for *c* the type of the annotations. Given an algebra class, for instance for type inference:

```
class Infer f where
  inferAlg :: AlgM (Reader Pos) f Type
```

we can lift it to annotated signatures:[5]

```
instance Infer f ⇒ Infer (f :&: Pos) where
  inferAlg (v :&: p) = local (const p) (inferAlg v)
```

When defining the other instances of the class, we can use the monadic function *ask* :: *Reader Pos Pos* to query the annotation of the current subterm. This provides

[5]The standard function *local* :: $(r \to r) \to$ *Reader r a* $\to$ *Reader r a* updates the environment by the function given as first argument.

a clean interface to the annotations. It requires, however, that we define a monadic algebra.

The alternative approach is to distribute the annotations over the sum, that is instead of *Sig* :&: *Pos* we use the type:

```
type SigP = Op :&: Pos :+: Val :&: Pos
```

Now we are able to define an instance where we have direct access to the annotation:

```
instance Infer (Val :&: Pos) where
  inferAlg (v :&: p) = ...
```

However, now we have the dual problem: we do not have immediate access to the annotation at the outermost level of the sum. Hence, we cannot use the function *liftA* to lift functions to annotated terms. Yet, this direction—propagating annotations outwards—is easier to deal with. We have to generalise the function *remA* to also deal with annotations distributed over sums. This is an easy exercise:

```
class RemA f g | f → g where
  remA :: f a → g a
instance RemA (f :&: c) f where
  remA (v :&: _) = v
instance RemA f f′ ⇒ RemA (g :&: c :+: f) (g :+: f′) where
  remA (Inl (v :&: _)) = Inl v
  remA (Inr v)         = Inr (remA v)
```

Now the function *remA* works as before, but it can also deal with signatures such as *SigP*, and the type of *liftA* becomes:

$$(\mathit{Functor}\ f, \mathit{RemA}\ f\ g) \Rightarrow (\mathit{Term}\ g \to t) \to \mathit{Term}\ f \to t$$

Both approaches have their share of benefits and drawbacks. The monadic approach provides a cleaner interface but necessitates a monadic style. The explicit distribution is more flexible as it both allows us to access the annotations directly by pattern matching or to thread them through a monad if that is more convenient. On the other hand, it means that adding annotations is not straightforwardly compositional anymore. The annotation :&:$c$ has to be added to each summand—just like compound signatures are not straightforwardly compositional, for instance we have to write the sum $f$ :+: $g$, for a signature $f = f_1$ :+: $f_2$, explicitly as $f_1$ :+: $f_2$ :+: $g$.

## 4.4 Context Matters

In this section, we will discuss two problems that arise when defining term algebras, that is algebras with a carrier of the form *Term f*. These problems occur when we want to lift term algebras to algebras on annotated terms, and when trying to compose term algebras. We will show how these problems can be addressed by *term homomorphisms*, a quite common special case of term algebras. In order to make this work, we shall generalise terms to contexts by using generalised algebraic data types (GADTs) [97].

### 4.4.1 Propagating Annotations

As we have seen in Section 4.3.3, it is easy to lift functions on terms to functions on annotated terms. It only amounts to removing all annotations before passing the term to the original function.

But what if we do not want to completely ignore the annotation but propagate it in a meaningful way to the output? Take for example the desugaring function *desug* we have defined in Section 4.2, which transforms terms over *Sig′* to terms over *Sig*. How do we lift this function easily to a function of the type:

$$\mathit{Term}\ (\mathit{Sig}'\ {:}\&{:}\ \mathit{Pos}) \to \mathit{Term}\ (\mathit{Sig}\ {:}\&{:}\ \mathit{Pos})$$

that propagates the annotations such that each annotation of a subterm in the result is taken from the subterm it originated? For example, in the desugaring of a term *iSwap x* to the term *iSnd x* ‘*iPair*‘ *iFst x*, the top-most *Pair*-term, as well as the two terms *Snd x* and *Fst x* should get the same annotation as the original subterm *iSwap x*.

This propagation is independent of the transformation function. The same scheme can also be used for the type inference function in order to annotate the inferred type terms with the positions of the code that is responsible for each part of the type terms.

It is clear that we will not be able provide a combinator of the type:

$$(\mathit{Term}\ f \to \mathit{Term}\ g) \to \mathit{Term}\ (f\ {:}\&{:}\ c) \to \mathit{Term}\ (g\ {:}\&{:}\ c)$$

that lifts any function to one that propagates annotations meaningfully. We cannot tell from a plain function of the type *Term f* → *Term g* where the subterms of the result term are originated in the input term. However, restricting ourselves to term algebras will not be sufficient either. That is, also a combinator of the type:

$$\mathit{Alg}\ f\ (\mathit{Term}\ g) \to \mathit{Alg}\ (f\ {:}\&{:}\ c)\ (\mathit{Term}\ (g\ {:}\&{:}\ c))$$

is out of reach. While we can tell from a term algebra—that is, a function of the type $f\ (\mathit{Term}\ g) \to \mathit{Term}\ g$—that some initial parts of the result term originate from the $f$-constructor at the root of the input, we do not know which parts. The term algebra only returns a *uniform* term of the type *Term g* that provides no information as to which parts were constructed from the $f$-part of the $f\ (\mathit{Term}\ g)$ argument and which were copied from the (*Term g*)-part.

Term algebras are still too general! We need to move to a function type that clearly states which parts are constructed from the “current” top-level symbol in $f$ and which are copied from its arguments in *Term g*. In order to express that certain parts are just copied, we can make use of parametric polymorphism.

Instead of an algebra, we can define a function on terms also by a *natural transformation*, a function of the type $\forall\ a\ .\ f\ a \to g\ a$. Such a function can only transform an $f$-constructor into a $g$-constructor and copy its arguments around. Since the copying is made explicit in the type, defining a function that propagates annotations through natural transformations is straightforward:

$$\begin{aligned}&\mathit{prop} :: (f\ a \to g\ a) \to (f\ {:}\&{:}\ c)\ a \to (g\ {:}\&{:}\ c)\ a\\&\mathit{prop}\ \eta\ (v\ {:}\&{:}\ c) = \eta\ v\ {:}\&{:}\ c\end{aligned}$$

Unfortunately, natural transformations are also quite limited. They only allow us to transform each constructor of the original term to exactly one constructor in the target term. This is for example not sufficient for the desugaring function, which translates a constructor application $iSwap\ x$ into three constructor applications $iSnd\ x`iPair`iFst\ x$. In order to lift this restriction, we need to be able to define a function of the type $\forall\ a.f\ a \to Context\ g\ a$ that transforms an $f$-constructor application to a $g$-*context* application, that is several nested applications of $g$-constructors potentially with some "holes" filled by values of type $a$.

We shall return to this idea in Section 4.4.4.

### 4.4.2 Composing Term Algebras

The benefit of having a desugaring function $desug :: Term\ Sig' \to Term\ Sig$, which is able to reduce terms over the richer signature $Sig'$ to terms over the core signature $Sig$, is that it allows us to easily lift functions that are defined on terms over $Sig$—such as evaluation and type inference—to terms over $Sig'$:

$$\begin{aligned}&eval' :: Term\ Sig' \to Term\ Val\\&eval' = eval\ .\ (desug :: Term\ Sig' \to Term\ Sig)\end{aligned}$$

However, looking at how $eval$ and $desug$ are defined, viz. as catamorphisms, we notice a familiar pattern:

$$eval' = cata\ evalAlg\ .\ cata\ desugAlg$$

This looks quite similar to the classic example of *deforestation* [31]:

$$map\ f\ .\ map\ g \quad\rightsquigarrow\quad map\ (f\ .\ g)$$

An expression that traverses a data structure twice is transformed into one that only does this once.

To replicate this on terms, we need an appropriately defined composition operator $\circledcirc$ on term algebras that allows us to perform a similar semantics-preserving transformation:

$$cata\ \phi_1\ .\ cata\ \phi_2 \quad\rightsquigarrow\quad cata\ (\phi_1 \circledcirc \phi_2)$$

As a result, the input term only needs to be traversed once instead of twice and the composition and decomposition of an intermediate term is avoided. The type of $\circledcirc$ should be:

$$Alg\ g\ (Term\ h) \to Alg\ f\ (Term\ g) \to Alg\ f\ (Term\ h)$$

Since term algebras are functions, the only way to compose them is by first making them compatible and then performing function composition. Given two term algebras $\phi_1 :: Alg\ g\ (Term\ h)$ and $\phi_2 :: Alg\ f\ (Term\ g)$, we can turn them into compatible functions by lifting $\phi_1$ to terms via $cata$. The problem now is that the composition $cata\ \phi_1\ .\ \phi_2$ has the type $f\ (Term\ g) \to Term\ h$, which is only an algebra if $g = h$. This issue arises due to the simple fact that the carrier of an algebra occurs in both the domain and the codomain of the function! Instead of a term algebra of type $f\ (Term\ g) \to Term\ g$, we need a function type in which the domain is more independent from the codomain in order to allow composition. Again, a type of the form $\forall\ a\ .\ f\ a \to Context\ g\ a$ provides a solution.

### 4.4.3 From Terms to Contexts and back

We have seen in the two preceding sections that we need an appropriate notion of *contexts*, that is a term that can also contain "holes" filled with values of a certain type. Starting from the definition of terms, we can easily generalise it to contexts by simply adding an additional case:

$$\mathbf{data}\ \mathit{Context}\ f\ a = \mathit{In}\ (f\ (\mathit{Context}\ f\ a))\mid \mathit{Hole}\ a$$

Note that we can obtain a type isomorphic to the one above using summation; $\mathit{Context}\ f\ a \cong \mathit{Term}\ (f \mathbin{:+:} K\ a)$ for a type:

$$\mathbf{data}\ K\ a\ b = K\ a$$

Since we will use contexts quite often, we will use the direct representation. Moreover, this allows us to tightly integrate contexts into our framework. Since contexts are terms with holes, we also want to go the other way around by defining terms as contexts *without holes*! This will allow us to lift functions defined on terms—catamorphisms, injections etc.—to functions on contexts that provide the original term-valued function as a special case.

The idea of defining terms as contexts without holes can be encoded in Haskell quite easily as a *generalised algebraic data type (GADT)* [97] with a *phantom type Hole*:

```
data Cxt :: ∗ → (∗ → ∗) → ∗ → ∗ where
  In   :: f (Cxt h f a) → Cxt h f a
  Hole :: a             → Cxt Hole f a
data Hole
```

In this representation we add an additional type argument that indicates whether the context might contain holes or not. A context that does have a hole must have a type of the form *Cxt Hole f a*. Our initial definition of contexts can thus be recovered by defining:

$$\mathbf{type}\ \mathit{Context} = \mathit{Cxt}\ \mathit{Hole}$$

That is, contexts *may* contain holes. On the other hand, terms must not contain holes. This can be defined by:

$$\mathbf{type}\ \mathit{Term}\ f = \forall\ h\ a\ .\ \mathit{Cxt}\ h\ f\ a$$

While this is a natural representation of terms as a special case of the more general concept of contexts, this usually causes some difficulties because of the impredicative polymorphism. We therefore prefer an approximation of this type that will do fine in almost any relevant case. Instead of universal quantification, we use empty data types *NoHole* and *Nothing*:

$$\mathbf{type}\ \mathit{Term}\ f = \mathit{Cxt}\ \mathit{NoHole}\ f\ \mathit{Nothing}$$

In practice, this does not pose any restriction whatsoever. Both *NoHole* and *Nothing* are phantom types and do not contribute to the internal representation of values. For the former this is obvious, for the latter this follows from the fact that the phantom type *NoHole* witnesses that the context has indeed no holes which would otherwise enforce the type *Nothing*. Hence, we can transform a term to any context type over any type of holes:

$$\begin{aligned}&\mathit{toCxt} :: \mathit{Functor}\ f \Rightarrow \mathit{Term}\ f \to \forall\ h\ a\ .\ \mathit{Cxt}\ h\ f\ a\\ &\mathit{toCxt}\ (\mathit{In}\ t) = \mathit{In}\ (\mathit{fmap}\ \mathit{toCxt}\ t)\end{aligned}$$

In fact, *toCxt* does not change the representation of the input term. Looking at its definition, *toCxt* is operationally equivalent to the identity. Thus, we can safely use the function $\mathit{unsafeCoerce} :: a \to b$ in order to avoid run-time overhead:

$$\begin{aligned}&\mathit{toCxt} :: \mathit{Functor}\ f \Rightarrow \mathit{Term}\ f \to \forall\ h\ a\ .\ \mathit{Cxt}\ h\ f\ a\\ &\mathit{toCxt} = \mathit{unsafeCoerce}\end{aligned}$$

This representation of contexts and terms allows us to uniformly define functions that work on both types. The function *inject* can be defined as before, but now has the type:

$$\mathit{inject} :: (g :\prec: f) \Rightarrow g\ (\mathit{Cxt}\ h\ f\ a) \to \mathit{Cxt}\ h\ f\ a$$

It thus works for both terms and proper contexts. The projection function has to be extended slightly to accommodate for holes:

$$\begin{aligned}&\mathit{project} :: (g :\prec: f) \Rightarrow \mathit{Cxt}\ h\ f\ a \to \mathit{Maybe}\ (g\ (\mathit{Cxt}\ h\ f\ a))\\ &\mathit{project}\ (\mathit{In}\ t) \quad\ \ = \mathit{proj}\ t\\ &\mathit{project}\ (\mathit{Hole}\ \_) = \mathit{Nothing}\end{aligned}$$

The relation between terms and contexts can also be illustrated algebraically. If we ignore for a moment the ability to define infinite terms due to Haskell's non-strict semantics, the type *Term* $\mathcal{F}$ represents the initial $\mathcal{F}$-algebra that has the carrier $\mathcal{T}(\mathcal{F})$, the terms over signature $\mathcal{F}$. The type of contexts *Context* $\mathcal{F}$ $\mathcal{X}$ on the other hand represents the free $\mathcal{F}$-algebra generated by $\mathcal{X}$ that has the carrier $\mathcal{T}(\mathcal{F},\mathcal{X})$, the terms over signature $\mathcal{F}$ and variables $\mathcal{X}$.

Thus, for recursion schemes, we can move naturally from catamorphisms, that is initial algebra semantics, to free algebra semantics:

$$\begin{aligned}&\mathit{free} :: \mathit{Functor}\ f \Rightarrow \mathit{Alg}\ f\ b \to (a \to b) \to \mathit{Cxt}\ h\ f\ a \to b\\ &\mathit{free}\ \phi\ f\ (\mathit{In}\ t) \quad\ = \phi\ (\mathit{fmap}\ (\mathit{free}\ \phi\ f)\ t)\\ &\mathit{free}\ \_\ f\ (\mathit{Hole}\ h) = f\ h\end{aligned}$$

$$\begin{aligned}&\mathit{freeM} :: (\mathit{Traversable}\ f, \mathit{Monad}\ m)\\ &\qquad\qquad \Rightarrow \mathit{AlgM}\ m\ f\ b \to (a \to m\ b) \to \mathit{Cxt}\ h\ f\ a \to m\ b\\ &\mathit{freeM}\ \phi\ f\ (\mathit{In}\ t) \quad\ = \phi \mathrel{=\!\!\ll} \mathit{mapM}\ (\mathit{freeM}\ \phi\ f)\ t\\ &\mathit{freeM}\ \_\ f\ (\mathit{Hole}\ h) = f\ h\end{aligned}$$

This yields the central function for working with contexts:

$$\begin{aligned}&\mathit{appCxt} :: \mathit{Functor}\ f \Rightarrow \mathit{Context}\ f\ (\mathit{Cxt}\ h\ f\ a) \to \mathit{Cxt}\ h\ f\ a\\ &\mathit{appCxt} = \mathit{free}\ \mathit{In}\ \mathit{id}\end{aligned}$$

This function takes a context whose holes are terms (or contexts) and returns the term (respectively context) that is obtained by merging the two—essentially by removing each constructor *Hole*. Notice how the type variables $h$ and $a$ are propagated from the input context's holes to the return type. In this way, we can uniformly treat both terms and contexts.

### 4.4.4 Term Homomorphisms

The examples from Sections 4.4.1 and 4.4.2 have illustrated the need for defining functions on terms by functions of the form $\forall\, a \,.\, f\ a \to \mathit{Context}\ g\ a$. Such functions can then be transformed to term algebras via *appCxt* and, thus, be lifted to terms:

```
appHom :: (Functor f, Functor g)
            ⇒ (∀ a . f a → Context g a) → Term f → Term g
appHom ρ = cata (appCxt . ρ)
```

In fact, the polymorphism in the type $\forall\, a \,.\, f\ a \to \mathit{Context}\ g\ a$ guarantees that arguments of the functor $f$ can only be copied—not inspected or modified. This restriction captures a well-known concept from tree automata theory:

**Definition 4.4.1** (term homomorphisms[6] [20, 107])**.** Let $\mathcal{F}$ and $\mathcal{G}$ be two sets of function symbols, possibly not disjoint. For each $n > 0$, let $\mathcal{X}_n = \{x_1, \dots, x_n\}$ be a set of variables disjoint from $\mathcal{F}$ and $\mathcal{G}$. Let $\rho_{\mathcal{F}}$ be a mapping that, with $f \in \mathcal{F}$ of arity $n$, associates a context $t_f \in \mathcal{T}(\mathcal{G}, \mathcal{X}_n)$. The *term homomorphism* $\rho\colon \mathcal{T}(\mathcal{F}) \to \mathcal{T}(\mathcal{G})$ determined by $\rho_{\mathcal{F}}$ is defined as follows:

$$\rho(f(t_1, \dots, t_n)) = t_f\,\{x_1 \mapsto \rho(t_1), \dots, x_n \mapsto \rho(t_n)\}$$

The term homomorphism $\rho$ is called *symbol-to-symbol* if, for each $f \in \mathcal{F}$, $t_f = g(y_1, \dots, y_m)$ with $g \in \mathcal{G}, y_1, \dots, y_m \in \mathcal{X}_n$, that is each $t_f$ is a context of height 1. It is called *$\varepsilon$-free* if, for each $f \in \mathcal{F}$, $t_f \notin \mathcal{X}_n$, that is each $t_f$ is a context of height at least 1.

Applying the *placeholders-via-naturality* principle of Hasuo et al. [39], term homomorphisms are captured by the following type:

```
type Hom f g = ∀ a . f a → Context g a
```

As we did for other functions on terms, we can generalise the application of term homomorphism uniformly to contexts:

```
appHom :: (Functor f, Functor g) ⇒ Hom f g → Cxt h f a → Cxt h g a
appHom ρ (In t)     = appCxt (ρ (fmap (appHom ρ) t))
appHom _ (Hole h) = Hole h
```

The use of explicit pattern matching in lieu of defining the function as a free algebra homomorphism *free* (*appCxt* . $\rho$) *Hole* is essential in order to obtain this general type. In particular, the use of the proper GADT constructor *Hole*, which has result type *Context* $g$ $a$, makes this necessary.

Of course, the polymorphic type of term homomorphisms restricts the class of functions that can be defined in this way. It can be considered as a special form of term algebra: *appCxt* . $\rho$ is the term algebra corresponding to the term homomorphism $\rho$. But not every catamorphism is also a term homomorphism. For certain term algebras we actually need to inspect the arguments of the functor

[6]Actually, Thatcher [107] calls them "tree homomorphisms". But we prefer the notion "term" over "tree" in our context.

instead of only shuffling them around. For example, we cannot hope to define the evaluation function *eval* as a term homomorphism.

Some catamorphisms, however, can be represented as term homomorphisms, for instance the desugaring function *desug*:

```
class (Functor f, Functor g) ⇒ Desug f g where
  desugHom :: Hom f g
```

Lifting term homomorphisms to sums is standard. The instances for the functors that do not need to be desugared can be implemented by turning a single functor application to a context of height 1, and using overlapping instances:

```
simpCxt :: Functor f ⇒ f a → Context f a
simpCxt = In . fmap Hole
instance (f :≺: g, Functor g) ⇒ Desug f g where
  desugHom = simpCxt . inj
```

Turning to the instance for *Sug*, we can see why a term homomorphism suffices for implementing *desug*. In the original catamorphic definition, we had for example:

```
desugAlg (Neg x) = iConst (−1) `iMult` x
```

Here we only need to copy the argument $x$ of the constructor *Neg* and define the appropriate context around it. This definition can be copied almost verbatim for the term homomorphism:

```
desugHom (Neg x) = iConst (−1) `iMult` Hole x
```

We only need to embed the $x$ as a hole. The same also applies to the other defining equations. In order to make the definitions more readable, we add a convenience function to the class *Desug*, which makes it possible to copy the catamorphic definition one-to-one:

```
class (Functor f, Functor g) ⇒ Desug f g where
  desugHom  :: Hom f g
  desugHom  = desugHom′ . fmap Hole
  desugHom′ :: Alg f (Context g a)
  desugHom′ = appCxt . desugHom
instance (Op :≺: f, Val :≺: f, Functor f) ⇒ Desug Sug f where
  desugHom′ (Neg x)  = iConst (−1) `iMult` x
  desugHom′ (Swap x) = iSnd x `iPair` iFst x
```

In the next two sections, we will show what we actually gain by adopting the term homomorphism approach. We will reconsider and address the issues that we identified in Sections 4.4.1 and 4.4.2.

#### 4.4.4.1 Propagating Annotations through Term Homomorphisms

The goal is now to take advantage of the structure of term homomorphisms in order to automatically propagate annotations. This boils down to transforming a function of type *Hom f g* to a function of type *Hom* (*f* :&: *c*) (*g* :&: *c*). In order to do this, we need a function that is able to annotate a context with a fixed annotation. Such a function is in fact itself a term homomorphism:

```
ann :: Functor f ⇒ c → Cxt h f a → Cxt h (f :&: c) a
ann c = appHom (simpCxt . (:&:c))
```

To be more precise, this function is a *symbol-to-symbol* term homomorphism—(:&:*c*) is of type ∀ *a* . *f a* → (*f* :&: *c*) *a*—that maps each constructor to exactly one constructor. The composition with *simpCxt* lifts it to the type of general term homomorphisms. The propagation of annotations is now simple:

```
propAnn :: Functor g ⇒ Hom f g → Hom (f :&: c) (g :&: c)
propAnn ρ (t :&: c) = ann c (ρ t)
```

The annotation of the current subterm is propagated to the context created by the original term homomorphism.

This definition can now be generalised—as we did in Section 4.3.3—such that it can also deal with annotations that have been distributed over a sum of signatures. Unfortunately, the type class *RemA* that we introduced for dealing with such distributed annotations is not enough for this setting as we need to extract and inject annotations now:

```
class DistAnn f c f' | f' → f, f' → c where
  injectA  :: c → f a → f' a
  projectA :: f' a → (f a, c)
```

An instance of *DistAnn f c f'* indicates that signature *f'* is a variant of *f* annotated with values of type *c*. The relevant instances are straightforward:

```
instance DistAnn f c (f :&: c) where
  injectA c v = v :&: c
  projectA (v :&: c) = (v, c)
instance DistAnn f c f' ⇒ DistAnn (g :+: f) c ((g :&: c) :+: f') where
  injectA c (Inl v) = Inl (v :&: c)
  injectA c (Inr v) = Inr (injectA c v)
  projectA (Inl (v :&: c)) = (Inl v, c)
  projectA (Inr v)         = let (v', c) = projectA v in (Inr v', c)
```

We can then make use of this infrastructure in the definition of *ann* and *propAnn*:

```
ann :: (DistAnn f c g, Functor f, Functor g) ⇒ c → Cxt h f a → Cxt h g a
ann c = appHom (simpCxt . injectA c)
propAnn :: (DistAnn f c f', DistAnn g c g', Functor g, Functor g')
           ⇒ Hom f g → Hom f' g'
propAnn f t' = let (t, c) = projectA t' in ann c (f t)
```

We can now use *propAnn* to propagate source position information from a full AST to its desugared version:

```
type SigP' = Sug :&: Pos :+: SigP
desugHom' :: Hom SigP' SigP
desugHom' = propAnn desugHom
```

#### 4.4.4.2 Composing Term Homomorphisms

Another benefit of the function type of term homomorphisms over term algebras is the simple fact that its domain $f\ a$ is independent of the target signature $g$:

$$\textbf{type}\ \mathit{Hom}\ f\ g = \forall\ a\ .\ f\ a \to \mathit{Context}\ g\ a$$

This enables us to compose term homomorphisms:

$$(\circledcirc) :: (\mathit{Functor}\ g, \mathit{Functor}\ h) \Rightarrow \mathit{Hom}\ g\ h \to \mathit{Hom}\ f\ g \to \mathit{Hom}\ f\ h$$
$$\rho_1 \circledcirc \rho_2 = \mathit{appHom}\ \rho_1\ .\ \rho_2$$

Here we make use of the fact that *appHom* also allows us to apply a term homomorphism to a proper context—*appHom* $\rho_1$ has type $\forall\ a\ .\ \mathit{Context}\ g\ a \to \mathit{Context}\ h\ a$.

Although the occurrence of the target signature in the domain of term algebras prevents them from being composed with each other, the composition with a term homomorphism is still possible:

$$(\boxdot) :: \mathit{Functor}\ g \Rightarrow \mathit{Alg}\ g\ a \to \mathit{Hom}\ f\ g \to \mathit{Alg}\ f\ a$$
$$\phi \boxdot \rho = \mathit{free}\ \phi\ \mathit{id}\ .\ \rho$$

The ability to compose term homomorphisms with term algebras or other term homomorphisms allows us to perform program transformations in the vein of deforestation [31]. For an example, recall that we have extended the evaluation to terms over $\mathit{Sig}'$ by precomposing the evaluation function with the desugaring function:

$$\mathit{eval}' :: \mathit{Term}\ \mathit{Sig}' \to \mathit{Term}\ \mathit{Val}$$
$$\mathit{eval}' = \mathit{eval}\ .\ \mathit{desug}$$

The same can be achieved by composing on the level of algebras respectively term homomorphisms instead of the level of functions:

$$\mathit{eval}' :: \mathit{Term}\ \mathit{Sig}' \to \mathit{Term}\ \mathit{Val}$$
$$\mathit{eval}' = \mathit{cata}\ (\mathit{evalAlg} \boxdot \mathit{desugHom})$$

Using the rewrite mechanism of GHC [87], we can make this optimisation automatic, by including the following rewrite rule:

$$\texttt{"cata/appHom"}\ \forall\ (\phi :: \mathit{Alg}\ g\ a)\ (\rho :: \mathit{Hom}\ f\ g)\ x\ .$$
$$\quad \mathit{cata}\ \phi\ (\mathit{appHom}\ \rho\ x) = \mathit{cata}\ (\phi \boxdot \rho)\ x$$

One can easily show that this transformation is sound. Moreover, a similar rule can be devised for composing two term homomorphisms. The run-time benefits of these optimisation rules are considerable as we will see in Section 4.6.2.

#### 4.4.4.3 Monadic Term Homomorphisms

Like catamorphisms, we can also easily lift term homomorphisms to monadic computations. We only need to lift the computations to a monadic type and use *mapM* instead of *fmap* for the recursion respectively use monadic function composition $\lll$ instead of pure function composition:

$$\begin{aligned}
&\textbf{type}\ \mathit{HomM}\ m\ f\ g = \forall\ a\ .\ f\ a \to m\ (\mathit{Context}\ g\ a)\\
&\mathit{appHomM} :: (\mathit{Traversable}\ f, \mathit{Functor}\ g, \mathit{Monad}\ m)\\
&\qquad\qquad \Rightarrow \mathit{HomM}\ m\ f\ g \to \mathit{Cxt}\ h\ f\ a \to m\ (\mathit{Cxt}\ h\ g\ a)\\
&\mathit{appHomM}\ \rho\ (\mathit{In}\ t) \quad\ = \mathit{liftM}\ \mathit{appCxt}\ .\ \rho \lll \mathit{mapM}\ (\mathit{appHomM}\ \rho)\ t\\
&\mathit{appHomM}\ \_\ (\mathit{Hole}\ h) = \mathit{return}\ (\mathit{Hole}\ h)
\end{aligned}$$

The same strategy yields monadic variants of $\circledcirc$ and $\boxdot$:

$$\begin{aligned}
&(\hat{\circledcirc}) :: (\mathit{Traversable}\ g, \mathit{Functor}\ h, \mathit{Monad}\ m)\\
&\qquad \Rightarrow \mathit{HomM}\ m\ g\ h \to \mathit{HomM}\ m\ f\ g \to \mathit{HomM}\ m\ f\ h\\
&\rho_1 \mathbin{\hat{\circledcirc}} \rho_2 = \mathit{appHomM}\ \rho_1 \lll \rho_2\\
&(\hat{\boxdot}) :: (\mathit{Traversable}\ g, \mathit{Monad}\ m)\\
&\qquad \Rightarrow \mathit{AlgM}\ m\ g\ a \to \mathit{HomM}\ m\ f\ g \to \mathit{AlgM}\ m\ f\ a\\
&\phi \mathbin{\hat{\boxdot}} \rho = \mathit{freeM}\ \phi\ \mathit{return} \lll \rho
\end{aligned}$$

In contrast to pure term homomorphisms, one has to be careful when applying these composition operators. The fusion equation:

$$\mathit{appHomM}\ (\rho_1 \mathbin{\hat{\circledcirc}} \rho_2) = \mathit{appHomM}\ \rho_1 \lll \mathit{appHomM}\ \rho_2$$

does not hold in general! However, Fokkinga [27] showed that for monads satisfying a certain distributivity law, the above equation indeed holds. An example of such a monad is the *Maybe* monad. Furthermore, the equation is also true whenever one of the term homomorphisms is in fact pure, that is of the form *return* . $\rho$ for a non-monadic term homomorphism $\rho$. The same also applies to the fusion equation for $\hat{\boxdot}$. Nevertheless, it is still possible to devise rewrite rules that perform deforestation under these restrictions.

An example of a monadic term homomorphism is the following function that recursively coerces a term to a sub-signature:

$$\begin{aligned}
&\mathit{deepProject} :: (\mathit{Functor}\ g, \mathit{Traversable}\ f, g :\prec: f) \Rightarrow \mathit{Term}\ f \to \mathit{Maybe}\ (\mathit{Term}\ g)\\
&\mathit{deepProject} = \mathit{appHomM}\ (\mathit{liftM}\ \mathit{simpCxt}\ .\ \mathit{proj})
\end{aligned}$$

As *proj* is, in fact, a monadic *symbol-to-symbol* term homomorphism we have to compose it with *simpCxt* to obtain a general monadic term homomorphism.

### 4.4.5 Beyond Catamorphisms

So far we have only considered (monadic) algebras and their (monadic) catamorphisms. It is straightforward to implement the machinery for programming in coalgebras and their anamorphisms:

$$\begin{aligned}
&\textbf{type}\ \mathit{Coalg}\ f\ a = a \to f\ a\\
&\mathit{ana} :: \mathit{Functor}\ f \Rightarrow \mathit{Coalg}\ f\ a \to a \to \mathit{Term}\ f\\
&\mathit{ana}\ \psi\ x = \mathit{In}\ (\mathit{fmap}\ (\mathit{ana}\ \psi)\ (\psi\ x))
\end{aligned}$$

In fact, also more advanced recursion schemes can be accounted for in our framework. This includes paramorphisms and histomorphisms as well as their dual notions of apomorphisms and futumorphisms [109]. Similarly, monadic variants of these recursion schemes can be derived using the type class *Traversable*.

As an example of the above-mentioned recursion schemes, we want to single out *futumorphisms*, as they can be represented conveniently using contexts and in fact are more natural to program than run-of-the-mill anamorphisms. The algebraic counterpart of futumorphisms are *cv-coalgebras* [109]. In their original algebraic definition they look rather cumbersome [109, Ch. 4.3]. If we implement cv-coalgebras in Haskell using contexts, the computation they denote becomes clear immediately:

$$\textbf{type}\ \mathit{CVCoalg}\ f\ a = a \to f\ (\mathit{Context}\ f\ a)$$

Anamorphisms only allow us to construct the target term one layer at a time. This can be plainly seen from the type $a \to f\ a$ of coalgebras. Futumorphisms on the other hand allow us to construct an arbitrary large part of the target term. Instead of only producing a single application of a constructor, cv-coalgebras produce a *non-empty* context, that is a context of height at least 1. The non-emptiness of the produced contexts guarantees that the resulting futumorphism is *productive*.

For the sake of brevity, we lift this restriction to non-empty contexts and consider *generalised cv-coalgebras*:

$$\textbf{type}\ \mathit{CVCoalg}\ f\ a = a \to \mathit{Context}\ f\ a$$

Constructing the corresponding futumorphism is simple and almost the same as for anamorphisms:

$$\begin{aligned}&\mathit{futu} :: \mathit{Functor}\ f \Rightarrow \mathit{CVCoalg}\ f\ a \to a \to \mathit{Term}\ f\\ &\mathit{futu}\ \psi\ x = \mathit{appCxt}\ (\mathit{fmap}\ (\mathit{futu}\ \psi)\ (\psi\ x))\end{aligned}$$

Generalised cv-coalgebras also occur when composing a coalgebra and a term homomorphism, which can be implemented by plain function composition:

$$\begin{aligned}&\mathit{compCoa} :: \mathit{Hom}\ f\ g \to \mathit{Coalg}\ f\ a \to \mathit{CVCoalg}\ g\ a\\ &\mathit{compCoa}\ \rho\ \psi = \rho\ .\ \psi\end{aligned}$$

This can then be lifted to the composition of a generalised cv-coalgebra and a term homomorphism, by running the term homomorphism:

$$\begin{aligned}&\mathit{compCVCoalg} :: (\mathit{Functor}\ f, \mathit{Functor}\ g)\\ &\qquad\qquad\qquad \Rightarrow \mathit{Hom}\ f\ g \to \mathit{CVCoalg}\ f\ a \to \mathit{CVCoalg}\ g\ a\\ &\mathit{compCVCoalg}\ \rho\ \psi = \mathit{appHom}\ \rho\ .\ \psi\end{aligned}$$

With generalised cv-coalgebras one has to be careful, though, as they might not be productive. However, the above constructions can be replicated with ordinary cv-coalgebras. Instead of general term homomorphisms, we have to restrict ourselves to $\varepsilon$*-free* term homomorphisms [20], which are captured by the type:

$$\textbf{type}\ \mathit{Hom}'\ f\ g = \forall\ a\ .\ f\ a \to g\ (\mathit{Context}\ g\ a)$$

This illustrates that with the help of contexts, (generalised) futumorphisms provide a much more natural coalgebraic programming model than anamorphisms.

## 4.5 Mutually Recursive Data Types and GADTs

Up to this point we have only considered the setting of a single recursively defined data type. We argue that this is the most common setting in the area we are targeting, viz. processing and analysing abstract syntax trees. Sometimes it is, however, convenient to encode certain invariants of the data structure, for instance well-typing of ASTs, as mutually recursive data types or GADTs. In this section, we will show how this can be encoded as a family of compositional data types by transferring the construction of Johann and Ghani [53] to compositional data types.

Recall that the idea of representing recursive data types as fixed points of functors is to abstract from the recursive reference to the data type that should be defined. Instead of a recursive data type:

```
data Exp   = ··· | Mult Exp Exp | Fst Exp
```

we define a functor:

```
data Sig a = ··· | Mult a   a   | Fst a
```

The trick for defining mutually recursive data types is to use phantom types as labels that indicate which data type we are currently in. As an example, reconsider our simple expression language over integers and pairs. But now we define them in a family of two mutually recursive data types in order to encode the expected invariants of the expression language, for instance the sum of two integers yields an integer:

```
data IExp  = Const Int | Mult IExp IExp | Fst PExp | Snd PExp
data PExp = Pair IExp IExp
```

We can encode this on signatures by adding an additional type argument that indicates the data types we are expecting as arguments to the constructors:

```
data Pair
data ISig a i = Const Int | Mult (a Int) (a Int) | Fst (a Pair) | Snd (a Pair)
data PSig a i = Pair (a Int) (a Int)
```

Notice that the type variable $a$ that is inserted in lieu of recursion is now of kind $* \to *$ as we consider a family of types. The “label type”—*Int* respectively *Pair*—then selects the desired type from this family. The definitions above, however, only indicate which data type we are expecting, for instance *Mult* expects two integer expressions and *Swap* a pair expression. In order to also label the result type accordingly, we rather want to define *ISig* and *PSig* as:

```
data ISig a Int   = ...
data PSig a Pair = ...
```

Using GADTs we can do this, although in a syntactically more verbose way:

```
data ISig a i where
  Const  ::             Int    → ISig a Int
  Mult   :: a Int → a Int   → ISig a Int
  Fst, Snd ::          a Pair → ISig a Int
data PSig a i where
  Pair :: a Int → a Int → PSig a Pair
```

Notice that signatures are not functors of kind $* \to *$ anymore. Instead, they have the kind $(* \to *) \to * \to *$, thus adding one level of indirection.

Following previous work [53, 123], we can formulate the actual recursive definition of terms as follows:

```
data Term f i = In {out :: f (Term f) i}
```

The first argument $f$ is a signature, that is it has the kind $(* \to *) \to * \to *$. The type constructor *In* recursively applies the signature $f$ while propagating the index $i$ according to the signature. Note that *Term* $f$ is of kind $* \to *$. A value of type *Term* $f$ $i$ is a mutually recursive data structure with topmost label $i$. In the recursive definition, *Term* $f$ is applied to a signature $f$, that is in the case of $f$ being *ISig* or *PSig* it instantiates the type variable $a$ in their respective definitions. The type signatures of *ISig* and *PSig* can thus be read as propagation rules for the labels. For example, *Fst* takes a term with top-level labeling *Pair* and returns a term with top-level labeling *Int*.

## 4.5.1 Higher-Order Functors

It is important to realise that the transition to a family of mutually recursive data types amounts to nothing more than adding a layer of indirection. A signature, which has previously been a functor, is now a (generalised) *higher-order functor* [53]:

```
type a →̇ b = ∀ i . a i → b i
class HFunctor f where
  hfmap :: a →̇ b → f a →̇ f b
instance HFunctor ISig where
  hfmap _ (Const i)  = Const i
  hfmap f (Mult x y) = Mult (f x) (f y)
  hfmap f (Fst x)    = Fst (f x)
```

The function *hfmap* witnesses that a natural transformation $a \overset{\cdot}{\to} b$ from functor $a$ to functor $b$ is mapped to a natural transformation $f\ a \overset{\cdot}{\to} f\ b$.

Observe the simplicity of the pattern that we used to lift our representation of compositional data types to mutually recursive types: replace functors with higher-order functors, and instead of the function space $\to$ consider the natural transformation space $\overset{\cdot}{\to}$. This simple pattern will turn out to be sufficient in order to lift most of the concepts of compositional data types to mutually recursive data types. Sums and injections can thus be represented as follows:

```
data (f :+: g) (a :: * → *) i = Inl (f a i) | Inr (g a i)
type NatM m f g = ∀ i . f i → m (g i)
class (sub :: (* → *) → * → *) :≺: sup where
  inj  :: sub a →̇ sup a
  proj :: NatM Maybe (sup a) (sub a)
```

Lifting *HFunctor* instances to sums works in the same way as we have seen for *Functor*. The same goes for instances of :≺:.

With the summation :+: in place we can define the family of data types that defines integer and pair expressions:

```
type Expr = Term (ISig :+: PSig)
```

This is indeed a family of types. We obtain the type of integer expressions with *Expr Int* and the type of pair expressions as *Expr Pair*.

### 4.5.2 Representing GADTs

Before we continue with lifting recursion schemes such as catamorphisms to the higher-order setting, we reconsider our example of mutually recursive data types. In contrast to the representation using a single recursive data type, the definition of *IExp* and *PExp* does not allow nested pairs—pairs are always built from integer expressions. The same goes for *Expr Int* and *Expr Pair*, respectively. This restriction is easily lifted by using a GADT instead:

```
data SExp i where
  Const ::                Int        → SExp Int
  Mult  :: SExp Int → SExp Int   → SExp Int
  Fst   ::                SExp (i,j) → SExp i
  Snd   ::                SExp (i,j) → SExp j
  Pair  :: SExp i   → SExp j     → SExp (i,j)
```

This standard GADT representation can be mapped directly to our signature definitions. However, instead of defining a single GADT, we proceed as we did with non-mutually recursive compositional data types. We split the signature into values and operations:

```
data Val a i where
  Const ::          Int → Val a Int
  Pair  :: a i → a j → Val a (i,j)
data Op a i where
  Mult :: a Int → a Int   → Op a Int
  Fst  ::          a (i,j) → Op a i
  Snd  ::          a (i,j) → Op a j
type Sig = Op :+: Val
```

Combining the two signatures above then yields the desired family of mutually recursive data types $\mathit{Term\ Sig} \cong \mathit{SExp}$.

This shows that the transition to higher-order functors also allows us to naturally represent GADTs in a modular fashion.

### 4.5.3 Recursion Schemes

We shall continue to apply the pattern for shifting to mutually recursive data types: replace *Functor* with *HFunctor* and function space $\rightarrow$ with the space of natural transformations $\stackrel{.}{\rightarrow}$. Take, for example, algebras and catamorphisms:

```
type Alg f a = f a ⥅ a
cata :: HFunctor f ⇒ Alg f a → Term f ⥅ a
cata φ = φ . hfmap (cata φ) . out
```

Now, an algebra has a family of types $a :: * \to *$ as carrier. That is, we have to move from algebras to *many-sorted* algebras. Representing many-sorted algebras comes quite natural in most cases. For example, the evaluation algebra class can be recast as a many-sorted algebra class as follows:

```
class Eval e v where
  evalAlg :: Alg e (Term v)
eval :: (HFunctor e, Eval e v) ⇒ Term e ⥅ Term v
eval = cata evalAlg
```

Here, we can make use of the fact that *Term v* is in fact a family of types and can thus be used as a carrier of a many-sorted algebra.

Except for the slightly more precise type of *projC* and *projP*, the definition of *Eval* is syntactically equal to its non-mutually recursive original from Section 4.2.1:

```
instance (Val :≺: v) ⇒ Eval Val v where
  evalAlg = inject
instance (Val :≺: v) ⇒ Eval Op v where
  evalAlg (Mult x y) = iConst (projC x ∗ projC y)
  evalAlg (Fst x)    = fst (projP x)
  evalAlg (Snd x)    = snd (projP x)
projC :: (Val :≺: v) ⇒ Term v Int → Int
projC v = case project v of Just (Const n) → n
projP :: (Val :≺: v) ⇒ Term v (i, j) → (Term v i, Term v j)
projP v = case project v of Just (Pair x y) → (x, y)
```

In some cases, it might be a bit more cumbersome to define and use the carrier of a many-sorted algebra. However, most cases are well-behaved and we can use the family of terms *Term f* as above or alternatively the identity respectively the constant functor:

```
data I a   = I {unI :: a}
data K a b = K {unK :: a}
```

For example, a many-sorted algebra class to evaluate expressions directly into Haskell values of the corresponding types can be defined as follows:

```
class EvalI f where
  evalAlgI :: Alg f I
evalI :: (EvalI f, HFunctor f) ⇒ Term f i → i
evalI = unI . cata evalAlgI
```

The lifting of other recursion schemes whether algebraic or coalgebraic can be achieved in the same way as illustrated for catamorphisms above. The necessary changes are again quite simple. Similarly to the type class *HFunctor*, we can obtain lifted versions of *Foldable* and *Traversable* that can then be used to implement generic programming techniques and to perform monadic computations, respectively. The generalisation of terms to contexts and the corresponding notion of term homomorphisms is also straightforward. The same fusion rules that we have considered for simple compositional data types can be implemented without any surprises as well.

The only real issue worth mentioning is that the generic querying combinator *query* needs to produce result values of a fixed type as opposed to a family of types. The propagation of types defined by GADTs cannot be captured by the simple pattern of the querying combinator. Thus, the querying combinator is typed as follows:

$$\mathit{query} :: \mathit{HFoldable}\ f \Rightarrow (\forall\ i\ .\ \mathit{Term}\ f\ i \to r) \to (r \to r \to r) \to \mathit{Term}\ f\ i \to r$$

For the *subs* combinator, which produces a list of all subterms, the issue is similar: *Term f* is a type family, thus [*Term f*] is not a valid type. However, we can obtain the desired type of list of terms by existentially quantifying over the index type using the GADT:

$$\mathbf{data}\ A\ f = \forall\ i\ .\ A\ (f\ i)$$

The type of *subs* can now be stated as follows:

$$\mathit{subs} :: \mathit{HFoldable}\ f \Rightarrow \mathit{Term}\ f\ i \to [A\ (\mathit{Term}\ f)]$$

## 4.6 Practical Considerations

Besides showing the expressiveness and usefulness of the framework of compositional data types, we also want to showcase its practical applicability as a software development tool. To this end, we consider aspects of *usability* and *performance impacts* as well.

### 4.6.1 Generating Boilerplate Code

The implementation of recursion schemes depends on the signatures being instances of the type class *Functor*. For generic programming techniques and monadic computations, we rely on the type classes *Foldable* and *Traversable*, respectively. Additionally, higher-order functors necessitate a set of lifted variants of the above-mentioned type classes. That is a lot of boilerplate code! Writing and maintaining this code would almost entirely defeat the advantage of using compositional data types in the first place.

Luckily, by leveraging Template Haskell [99], instance declarations of all generic type classes that we have mentioned in this paper can be generated automatically at compile time similar to Haskell's **deriving** mechanism. Even though some Haskell packages such as *derive* already provide automatically derived instances for some of the standard classes like *Functor*, *Foldable*, and *Traversable*, we chose to implement the instance generators for these as well. The heavy use of the methods of these classes for implementing recursion schemes means that they contribute considerably to the computational overhead! Automatically deriving instance declarations with carefully optimised implementations of each of the class methods has proven to yield substantial run-time improvements, especially for monadic computations.

We already mentioned that we assume with each constructor:

$$\mathit{Constr} :: t_1 \to \cdots \to t_n \to f\ a$$

of a signature $f$, a smart constructor defined by:

$$\begin{aligned}&\mathit{iConstr} :: f :\prec: g \Rightarrow s_1 \to \cdots \to s_n \to \mathit{Term}\ g\\&\mathit{iConstr}\ x_1 \ldots x_n = \mathit{inject}\ (\mathit{Constr}\ x_1 \ldots x_n)\end{aligned}$$

where the types $s_i$ are the same as $t_i$ except with occurrences of the type variable $a$ replaced by $\mathit{Term}\ g$. These smart constructors can be easily generated automatically using Template Haskell.

Another issue is the declaration of instances of type classes *Eq*, *Ord*, and *Show* for types of the form $\mathit{Term}\ f$. This can be achieved by lifting these type classes to functors, for instance for *Eq*:

$$\begin{aligned}&\textbf{class}\ \mathit{EqF}\ f\ \textbf{where}\\&\quad \mathit{eqF} :: \mathit{Eq}\ a \Rightarrow f\ a \to f\ a \to \mathit{Bool}\end{aligned}$$

From instances of this class, corresponding instances of *Eq* for terms and contexts can be derived:

$$\begin{aligned}&\textbf{instance}\ (\mathit{EqF}\ f, \mathit{Eq}\ a) \Rightarrow \mathit{Eq}\ (\mathit{Cxt}\ h\ f\ a)\ \textbf{where}\\&\quad (\equiv)\ (\mathit{In}\ t_1)\quad (\mathit{In}\ t_2) \quad = t_1\ \text{`}\mathit{eqF}\text{`}\ t_2\\&\quad (\equiv)\ (\mathit{Hole}\ h_1)\ (\mathit{Hole}\ h_2) = h_1 \equiv h_2\\&\quad (\equiv)\ \_ \qquad\quad \_ \qquad\qquad = \mathit{False}\end{aligned}$$

Instances of *EqF*, *OrdF*, and *ShowF* can be derived straightforwardly using Template Haskell, which then yield corresponding instances of *Eq*, *Ord*, and *Show* for terms and contexts. The thus obtained instances are equivalent to the ones obtained from Haskell's **deriving** mechanism on corresponding recursive data types.

Figure 4.1 demonstrates the *complete* source code needed in order to implement some of the earlier examples in our library.

### 4.6.2 Performance Impact

In order to minimise the overhead of the recursion schemes, we applied some simple optimisations to the implementation of the recursion schemes themselves. For example, *cata* is defined as:

$$\begin{aligned}&\mathit{cata} :: \forall f\ a\ .\ \mathit{Functor}\ f \Rightarrow \mathit{Alg}\ f\ a \to \mathit{Term}\ f \to a\\&\mathit{cata}\ \phi = \mathit{run}\\&\quad \textbf{where}\ \mathit{run} :: \mathit{Term}\ f \to a\\&\qquad\qquad \mathit{run}\ (\mathit{In}\ t) = \phi\ (\mathit{fmap}\ \mathit{run}\ t)\end{aligned}$$

The biggest speedup, however, can be obtained by providing automatically generated, carefully optimised implementations for each method of the type classes *Foldable* and *Traversable*.

```
import Data.Comp
import Data.Comp.Derive
import Data.Comp.Show ()

data Val a = Const Int | Pair a a
data Op a  = Mult a a | Fst a | Snd a
data Sug a = Neg a | Swap a
type Sig   = Op :+: Val
type Sig'  = Sug :+: Sig

$(derive [makeFunctor, makeFoldable, makeTraversable, makeShowF, smartConstructors]
         [''Val, ''Op, ''Sug])

-- * Term Evaluation
class Eval f v where evalAlg :: Alg f (Term v)

$(derive [liftSum] [''Eval]) -- lift Eval to coproducts

eval :: (Functor f, Eval f v) ⇒ Term f → Term v
eval = cata evalAlg

instance (Val :<: v) ⇒ Eval Val v where
  evalAlg = inject

instance (Val :<: v) ⇒ Eval Op v where
  evalAlg (Mult x y) = let (Just (Const n), Just (Const m)) = (project x, project y)
                       in iConst (n ∗ m)
  evalAlg (Fst x)    = let Just (Pair v _) = project x in v
  evalAlg (Snd x)    = let Just (Pair _ v) = project x in v

-- * Desugaring
class (Functor f, Functor g) ⇒ Desug f g where
  desugHom :: Hom f g
  desugHom = desugHom' . fmap Hole
  desugHom' :: Alg f (Context g a)
  desugHom' x = appCxt (desugHom x)

$(derive [liftSum] [''Desug]) -- lift Desug to coproducts

desug :: Desug f g ⇒ Term f → Term g
desug = appHom desugHom

instance (Functor f, Functor g, f :<: g) ⇒ Desug f g where
  desugHom = simpCxt . inj

instance (Op :<: f, Val :<: f, Functor f) ⇒ Desug Sug f where
  desugHom' (Neg x)  = iConst (-1) `iMult` x
  desugHom' (Swap x) = iSnd x `iPair` iFst x

eval' :: Term Sig' → Term Val
eval' = eval . (desug :: Term Sig' → Term Sig)
```

Figure 4.1: Example usage of the compositional data types library.

In order to gain speedup in the implementation of generic programming combinators, we applied the same techniques as Mitchell and Runciman [73] by leveraging deforestation [31] via *build*. The *subs* combinator is thus defined as:

$$
\begin{aligned}
&\mathit{subs} :: \forall f\ .\ \mathit{Foldable}\ f \Rightarrow \mathit{Term}\ f \rightarrow [\mathit{Term}\ f]\\
&\mathit{subs}\ t = \mathit{build}\ (f\ t)\ \textbf{where}\\
&\quad f :: \mathit{Term}\ f \rightarrow (\mathit{Term}\ f \rightarrow b \rightarrow b) \rightarrow b \rightarrow b\\
&\quad f\ t\ \mathit{cons}\ \mathit{nil} = t\ \text{`}\mathit{cons}\text{`}\ \mathit{foldl}\ (\lambda u\ s \rightarrow f\ s\ \mathit{cons}\ u)\ \mathit{nil}\ (\mathit{out}\ t)
\end{aligned}
$$

| **Function** | hand-written | random (10) | random (20) |
|---|---|---|---|
| *desugHom* | $3.6 \cdot 10^{-1}$ | $5.0 \cdot 10^{-3}$ | $6.1 \cdot 10^{-6}$ |
| *desugCata* | $1.8 \cdot 10^{-1}$ | $4.41 \cdot 10^{-3}$ | $5.3 \cdot 10^{-6}$ |
| *inferDesug* | (3.38) 1.11 | (3.45) 1.52 | (3.14) 0.82 |
| *inferDesugM* | (2.68) 1.38 | (2.87) 1.61 | (2.79) 0.84 |
| *infer* | 2.39 | 2.29 | 2.65 |
| *inferM* | 1.06 | 1.30 | 1.68 |
| *evalDesug* | (6.40) 2.64 | (3.13) 1.79 | (4.74) 0.89 |
| *evalDesugM* | (7.32) 4.34 | (6.22) 3.47 | (9.69) 2.98 |
| *eval* | 2.58 | 1.84 | 1.64 |
| *evalDirect* | 6.10 | 3.96 | 3.62 |
| *evalM* | 3.41 | 4.78 | 7.52 |
| *evalDirectM* | 5.72 | 4.90 | 4.56 |
| *contVar* | 1.92 | 1.97 | 3.22 |
| *freeVars* | 1.23 | 1.26 | 1.41 |
| *contVarC* | 10.05 | 7.01 | 11.68 |
| *contVarU* | 8.24 | 5.64 | 11.21 |
| *freeVarsC* | 2.34 | 2.04 | 1.68 |
| *freeVarsU* | 2.03 | 1.75 | 1.58 |

Table 4.1: Run-time of functions on compositional data types (as multiples of the run-time of an implementation using ordinary algebraic data types).

Instead of building the result list directly, we use the *build* combinator, which then can be eliminated if combined with a consumer such as a fold or a list comprehension.

Table 4.1 shows the run-time performance of our framework for various functions dealing with ASTs: desugaring (*desug*), type inference (*infer*), expression evaluation (*eval*), and listing respectively searching for free variables (*freeVars*, *contVar*). The *Hom* and *Cata* version of *desug* differ in that the former is defined as a term homomorphism, the latter as a catamorphism. For *eval* and *infer*, the suffix *Desug* indicates that the computation is prefixed by a desugaring phase (using *desugHom*), the suffix *M* indicates monadic variants (for error handling), and *Direct* indicates that the function was implemented not as a catamorphism but using explicit recursion. The numbers in the table are multiples of the run-time of an implementation using ordinary algebraic data types and recursion. The numbers in parentheses indicate the run-time factor if the automatic fusion described in Section 4.4.4.2 is disabled. Each function is tested on three different inputs of increasing size. The first is a hand-written "natural" expression consisting of 16 nodes. The other two expressions are randomly generated expressions of depth 10 and 20, respectively, which corresponds to approximately 800 respectively 200,000 nodes. This should reveal how the overhead of our framework scales. The benchmarks were performed with the *criterion* framework using GHC 7.0.2 with optimisation flag `-O2`.

As a pleasant surprise, we observe that the penalty of using compositional data types is comparatively low. It is in the same ballpark as for generic programming libraries [73, 95]. For some functions we even obtain a speedup! The biggest surprise is, however, the massive speedup gained by the desugaring function. In both its catamorphic and term-homomorphic version, it seems to perform asymptotically better than the classic implementation, yielding a speedup of over five orders of magnitude. We were also surprised to see that (except for one case) functions programmed as catamorphisms outperformed functions using explicit recursion! In fact, with GHC 6.12, the situation was reversed.

Moreover, we observe that the fusion rules implemented in our framework uniformly yield a considerable speedup of up to factor five. As a setback, however, we have to recognise that implementing desugaring as a term homomorphism yields a slowdown of factor up to two compared to its catamorphic version.

Finally, we compared our implementation of generic programming techniques with Uniplate [73], one of the top-performing generic programming libraries. In particular, we looked at its *universe* combinator that computes the list of all subexpressions. We have implemented this combinator in our framework as *subs*. In Table 4.1, our implementation is indicated by the suffix $C$, the Uniplate implementation, working on ordinary algebraic data types, is indicated by $U$. We can see that we are able to obtain comparable performance in all cases.

## 4.7 Discussion

Starting from Swierstra's *data types à la carte* [104], we have constructed a framework for representing data types in a compositional fashion that is readily usable for practical applications. Our biggest contribution is the generalisation of terms to contexts which allow us to capture the notion of term homomorphisms. Term homomorphisms provide a rich structure that allows flexible reuse and enables simple but effective optimisation techniques. Moreover, term homomorphisms can be easily extended with a state. Depending on how the state is propagated, this yields bottom-up respectively top-down tree transducers [20]. The techniques for fusion and propagation of annotations can be easily adapted.

### 4.7.1 Related Work

The definition of monadic catamorphisms that we use goes back to Fokkinga [27]. He only considers monads satisfying a certain distributivity law. However, this distributivity is only needed for the fusion rules of Section 4.4.4.3 to be valid. Steenbergen et al. [108] use the same approach to implement catamorphisms with errors. In contrast, Visser and Löh [110] consider monadic catamorphism for which the monadic effect is part of the term structure.

The construction to add annotations to functors is also employed by Steenbergen et al. [108] to add detailed source position annotations to ASTs. However, since they are considering general catamorphisms, they are not able to provide a means to propagate annotations. Moreover, since Steenbergen et al. do not account for sums of functors, the distribution of annotations over sums is not an issue for them. Visser and Löh [110] consider a more general form of annotations via arbitrary *functor transformations*. Unfortunately, this generality prohibits the automatic propagation of annotations as well as their distribution over sums.

Methods to represent mutually recursive data types as fixed points of (regular) functors have been explored to some extent [15, 61, 103, 123]. All of these techniques are limited to mutually recursive data types in which the number of nested data types is limited up front and are thus not compositional. However, in the representation of

Yakushev et al. [123], the restriction to mutually recursive data types with a closed set of constituent data types was implemented intentionally. Our representation simply removes these restrictions which would in fact add no benefit in our setting. The resulting notion of higher-order functors that we considered was also used by Johann and Ghani [53] in order to represent GADTs.

### 4.7.2 Future Work

There are a number of aspects that are still missing which should be the subject of future work. As we have indicated, the restriction of the subtyping class :≺: hinders full compositionality of signature summation :+:. A remedy could be provided with a richer type system as proposed by Yorgey [124]. This would also allow us to define the right-distributivity of annotations :&: over sums :+: more directly by a type family. Alternatively, this issue can be addressed with type instance-chains as proposed by Morris and Jones [75]. Another issue of Swierstra's original work is the *project* function that allows us to inspect terms ad-hoc. Unfortunately, it does not allow us to give a complete case analysis. In order to provide this, we need a function of the type:

$$(f :\prec: g) \Rightarrow \mathit{Term}\ g \rightarrow \mathit{Either}\ (f\ (\mathit{Term}\ g))\ ((g :-: f)\ (\mathit{Term}\ g))$$

which allows us to match against the "remainder signature" $g :-: f$.

# Chapter 5

# Parametric Compositional Data Types[⋆]

**Abstract**

In previous work we have illustrated the benefits that *compositional data types* (CDTs) offer for implementing languages and in general for dealing with abstract syntax trees (ASTs). Based on Swierstra's *data types à la carte*, CDTs are implemented as a Haskell library that enables the definition of recursive data types and functions on them in a modular and extendable fashion. Although CDTs provide a powerful tool for analysing and manipulating ASTs, they lack a convenient representation of variable binders. In this paper we remedy this deficiency by combining the framework of CDTs with Chlipala's parametric higher-order abstract syntax (PHOAS). We show how a generalisation from functors to difunctors enables us to capture PHOAS while still maintaining the features of the original implementation of CDTs, in particular its modularity. Unlike previous approaches, we avoid so-called *exotic terms* without resorting to abstract types: this is crucial when we want to perform *transformations* on CDTs that inspect the recursively computed CDTs, such as constant folding.

## 5.1 Introduction

When implementing domain-specific languages (DSLs)—either as embedded languages or stand-alone languages—the abstract syntax trees (ASTs) of programs are usually represented as elements of a recursive algebraic data type. These ASTs typically undergo various transformation steps, such as desugaring from a full language to a core language. But reflecting the invariants of these transformations in the type system of the host language can be problematic. For instance, in order to reflect a desugaring transformation in the type system, we must define a separate data type for ASTs of the core language. Unfortunately, this has the side effect that common functionality, such as pretty printing, has to be duplicated.

Wadler identified the essence of this issue as the *Expression Problem*: "the goal [...] to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety" [114]. Swierstra [104] elegantly addressed this problem using Haskell and its type classes machinery. While Swierstra's approach exhibits

[⋆] *Joint work with Patrick Bahr [10].*

invaluable simplicity and clarity, it lacks features necessary to apply it in a practical setting beyond the confined simplicity of the expression problem. To this end, the framework of *compositional data types* (CDTs) [11] provides a rich library for implementing practical functionality on highly modular data types. This includes support of a wide array of recursion schemes in both pure and monadic forms, as well as mutually recursive data types and generalised algebraic data types (GADTs) [97].

What CDTs fail to address, however, is a transparent representation of variable binders that frees the programmer's mind from common issues like computations modulo $\alpha$-equivalence and capture-avoiding substitutions. The work we present in this paper fills that gap by adopting (a restricted form of) higher-order abstract syntax (HOAS) [88], which uses the host language's variable binding mechanism to represent binders in the object language. Since implementing efficient recursion schemes in the presence of HOAS is challenging [25, 66, 98, 116], integrating this technique with CDTs is a non-trivial task.

Following a brief introduction to CDTs in Section 5.2, we describe how to achieve this integration as follows:

- We adopt parametric higher-order abstract syntax (PHOAS) [19], and we show how to capture this restricted form of HOAS via difunctors. The thus obtained *parametric compositional data types* (PCDTs) allow for the definition of modular catamorphisms à la Fegaras and Sheard [25] in the presence of binders. Unlike previous approaches, our technique does not rely on abstract types, which is crucial for modular computations that are also modular in their result type (Section 5.3).
- We illustrate why monadic computations constitute a challenge in the parametric setting and we show how monadic catamorphisms can still be defined for a restricted class of PCDTs (Section 5.4).
- We show how to transfer the restricted recursion scheme of *term homomorphisms* [11] to PCDTs. Term homomorphisms enable the same flexibility for reuse and opportunity for deforestation [112] that we know from CDTs.
- We show how to represent mutually recursive data types and GADTs by generalising PCDTs in the style of Johann and Ghani [53] (Section 5.6).
- We illustrate the practical applicability of our framework by means of a complete library example, and we show how to automatically derive functionality for deciding equality (Section 5.7).

Parametric compositional data types are available as a Haskell library[1], including numerous examples. We have included two of these examples in Appendix D.1. All code fragments presented throughout the paper are written in (literate) Haskell [62], and the library relies on several language extensions that are currently only known to be supported by the Glasgow Haskell Compiler (GHC).

## 5.2 Compositional Data Types

Based on Swierstra's *data types à la carte* [104], compositional data types (CDTs) [11] provide a framework for manipulating recursive data structures in a type-safe,

[1]See http://hackage.haskell.org/package/compdata.

modular manner. The prime application of CDTs is within language implementation and AST manipulation, and we present the basic concepts of CDTs in this section. More advanced concepts are introduced in Sections 5.4, 5.5, and 5.6.

### 5.2.1 Motivating Example

Consider an extension of the lambda calculus with integers, addition, let expressions, and error signalling:

$$e ::= \lambda x.e \mid x \mid e_1\, e_2 \mid n \mid e_1 + e_2 \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid \mathbf{error}$$

Our goal is to implement a pretty printer, a desugaring transformation, constant folding, and a call-by-value interpreter for the simple language above. The desugaring transformation will turn let expressions **let** $x = e_1$ **in** $e_2$ into $(\lambda x.e_2)\ e_1$. Constant folding and evaluation will take place after desugaring, that is both computations are only defined for the core language without let expressions.

The standard approach to representing the language above is in terms of an algebraic data type:

$$\begin{aligned}
&\mathbf{type}\ \mathit{Var} = \mathit{String}\\
&\mathbf{data}\ \mathit{Exp} = \mathit{Lam}\ \mathit{Var}\ \mathit{Exp} \mid \mathit{Var}\ \mathit{Var} \mid \mathit{App}\ \mathit{Exp}\ \mathit{Exp}\\
&\qquad\qquad \mid\ \mathit{Lit}\ \mathit{Int} \mid \mathit{Plus}\ \mathit{Exp}\ \mathit{Exp} \mid \mathit{Let}\ \mathit{Var}\ \mathit{Exp}\ \mathit{Exp} \mid \mathit{Err}
\end{aligned}$$

We may then straightforwardly define the pretty printer $\mathit{pretty} :: \mathit{Exp} \to \mathit{String}$. However, when we want to implement the desugaring transformation, we need a new algebraic data type:

$$\begin{aligned}
&\mathbf{data}\ \mathit{Exp}' = \mathit{Lam}'\ \mathit{Var}\ \mathit{Exp}' \mid \mathit{Var}'\ \mathit{Var} \mid \mathit{App}'\ \mathit{Exp}'\ \mathit{Exp}'\\
&\qquad\qquad \mid\ \mathit{Lit}'\ \mathit{Int} \mid \mathit{Plus}'\ \mathit{Exp}'\ \mathit{Exp}' \mid \mathit{Err}'
\end{aligned}$$

That is, we need to replicate all constructors of *Exp*—except *Let*—into a new type $\mathit{Exp}'$ of core expressions, in order to obtain a properly typed desugaring function $\mathit{desug} :: \mathit{Exp} \to \mathit{Exp}'$. Not only does this mean that we have to replicate the constructors, we also need to replicate common functionality, for instance in order to obtain a pretty printer for $\mathit{Exp}'$ we must either write a new function, or write an injection function $\mathit{Exp}' \to \mathit{Exp}$.

CDTs provide a solution that allows us to define the ASTs for (core) expressions without having to duplicate common constructors, and without having to give up on statically guaranteed invariants about the structure of the ASTs. CDTs take the viewpoint of data types as fixed points of functors [67], that is the definition of the AST data type is separated into non-recursive signatures (functors) on the one hand and the recursive structure on the other hand. For our example, we define the following signatures (omitting the straightforward *Functor* instance declarations):

$$\begin{array}{ll}
\mathbf{data}\ \mathit{Lam}\ a = \mathit{Lam}\ \mathit{Var}\ a & \mathbf{data}\ \mathit{Plus}\ a = \mathit{Plus}\ a\ a\\
\mathbf{data}\ \mathit{Var}\ a\ = \mathit{Var}\ \mathit{Var} & \mathbf{data}\ \mathit{Let}\ a\ = \mathit{Let}\ \mathit{Var}\ a\ a\\
\mathbf{data}\ \mathit{App}\ a = \mathit{App}\ a\ a & \mathbf{data}\ \mathit{Err}\ a\ = \mathit{Err}\\
\mathbf{data}\ \mathit{Lit}\ a\ = \mathit{Lit}\ \mathit{Int} &
\end{array}$$

Signatures can then be combined in a modular fashion by means of a formal sum of functors:

```
data (f :+: g) a = Inl (f a) | Inr (g a)
instance (Functor f, Functor g) ⇒ Functor (f :+: g) where
  fmap f (Inl x) = Inl (fmap f x)
  fmap f (Inr x) = Inr (fmap f x)
type Sig  = Lam :+: Var :+: App :+: Lit :+: Plus :+: Err :+: Let
type Sig′ = Lam :+: Var :+: App :+: Lit :+: Plus :+: Err
```

Finally, the type of terms over a (potentially compound) signature $f$ can be constructed as the (least) fixed point of the signature $f$:

```
data Term f = In {out :: f (Term f)}
```

Modulo strictness, *Term Sig* is isomorphic to *Exp*, and *Term Sig′* is isomorphic to *Exp′*.

The use of formal sums entails that each (sub)term has to be explicitly tagged with zero or more *Inl* or *Inr* tags. In order to add the right tags automatically, injections are derived using a type class:

```
class sub :≺: sup where
  inj  :: sub a → sup a
  proj :: sup a → Maybe (sub a)
```

Using *overlapping instance* declarations, the subsignature relation :≺: can be constructively defined [104]. However, due to the limitations of Haskell's type class system, instances are restricted to the form $f$ :≺: $g$ where $f$ is atomic, that is not a sum, and $g$ is a right-associated sum, for instance $g_1$ :+: ($g_2$ :+: $g_3$) but not ($g_1$ :+: $g_2$) :+: $g_3$. With the carefully defined instances for :≺:, injection and projection functions for terms can then be defined as follows:

```
inject :: (g :≺: f) ⇒ g (Term f) → Term f
inject = In . inj
project :: (g :≺: f) ⇒ Term f → Maybe (g (Term f))
project = proj . out
```

Additionally, in order to reduce the syntactic overhead, the CDTs library can automatically derive smart constructors that comprise the injections [11], for instance:

```
iPlus :: (Plus :≺: f) ⇒ Term f → Term f → Term f
iPlus x y = inject (Plus x y)
```

Using the derived smart constructors, we can then write expressions such as **let** $x = 2$ **in** $(\lambda y.y + x)\ 3$ without syntactic overhead:

```
e :: Term Sig
e = iLet "x" (iLit 2) ((iLam "y" (Var "y" `iPlus` Var "x")) `iApp` iLit 3)
```

In fact, the principal type of $e$ is the *open* type:

$$(\mathit{Lam} :\prec: f, \mathit{Var} :\prec: f, \mathit{App} :\prec: f, \mathit{Lit} :\prec: f, \mathit{Plus} :\prec: f, \mathit{Let} :\prec: f) \Rightarrow \mathit{Term}\ f$$

which means that $e$ can be used as a term over any signature containing at least these six signatures!

Next, we want to define the pretty printer, that is a function of the type $\mathit{Term}\ \mathit{Sig} \rightarrow \mathit{String}$. In order to make a recursive function definition modular too, it is defined as the catamorphism of an algebra [67]:

```
type Alg f a = f a → a
cata :: Functor f ⇒ Alg f a → Term f → a
cata φ = φ . fmap (cata φ) . out
```

The advantage of this approach is that algebras can be easily combined over formal sums. A modular algebra definition is obtained by an open family of algebras indexed by the signature and closed under forming formal sums. This is achieved as a type class:

```
class Pretty f where
  prettyAlg :: Alg f String
instance (Pretty f, Pretty g) ⇒ Pretty (f :+: g) where
  prettyAlg (Inl x) = prettyAlg x
  prettyAlg (Inr x) = prettyAlg x
pretty :: (Functor f, Pretty f) ⇒ Term f → String
pretty = cata prettyAlg
```

The instance declaration that lifts *Pretty* instances to sums is crucial. Yet, the structure of its declaration is independent from the particular algebra class, and the CDTs library provides a mechanism for automatically deriving such instances [11]. What remains in order to implement the pretty printer is to define instances of the *Pretty* algebra class for the six signatures:

```
instance Pretty Lam where
  prettyAlg (Lam x e) = "(\\" ++ x ++ ". " ++ e ++ ")"
instance Pretty Var where
  prettyAlg (Var x) = x
instance Pretty App where
  prettyAlg (App e1 e2) = "(" ++ e1 ++ " " ++ e2 ++ ")"
instance Pretty Lit where
  prettyAlg (Lit n) = show n
instance Pretty Plus where
  prettyAlg (Plus e1 e2) = "(" ++ e1 ++ " + " ++ e2 ++ ")"
instance Pretty Let where
  prettyAlg (Let x e1 e2) = "let " ++ x ++ " = " ++ e1 ++ " in " ++ e2
instance Pretty Err where
  prettyAlg Err = "error"
```

With these definitions we then have that *pretty e* evaluates to the string `let x = 2 in ((\y. (y + x)) 3)`. Moreover, we automatically obtain a pretty printer for the core language as well, compare the type of *pretty*.

## 5.3 Parametric Compositional Data Types

In the previous section we considered a first-order encoding of the language, which means that we have to be careful to ensure that computations are invariant under $\alpha$-equivalence, for instance when implementing capture-avoiding substitutions. *Higher-order abstract syntax* (HOAS) [88] remedies this issue, by representing variables and binders of the object language in terms of those of the meta language.

### 5.3.1 Higher-Order Abstract Syntax

In a standard Haskell HOAS encoding we replace the signatures *Var* and *Lam* by a revised *Lam* signature:

$$\textbf{data}\ \mathit{Lam}\ a = \mathit{Lam}\ (a \to a)$$

Now, however, *Lam* is no longer an instance of *Functor*, because $a$ occurs both as a contravariant argument and a covariant argument. We therefore need to generalise functors in order to allow for negative occurrences of the recursive parameter. *Difunctors* [66] provide such a generalisation:

$$\begin{aligned}
&\textbf{class}\ \mathit{Difunctor}\ f\ \textbf{where}\\
&\quad \mathit{dimap} :: (a \to b) \to (c \to d) \to f\ b\ c \to f\ a\ d\\
&\textbf{instance}\ \mathit{Difunctor}\ (\to)\ \textbf{where}\\
&\quad \mathit{dimap}\ f\ g\ h = g\ .\ h\ .\ f\\
&\textbf{instance}\ \mathit{Difunctor}\ f \Rightarrow \mathit{Functor}\ (f\ a)\ \textbf{where}\\
&\quad \mathit{fmap} = \mathit{dimap}\ \mathit{id}
\end{aligned}$$

A difunctor must preserve the identity function and distribute over function composition:

$$\mathit{dimap}\ \mathit{id}\ \mathit{id} = \mathit{id} \qquad \text{and} \qquad \mathit{dimap}\ (f\ .\ g)\ (h\ .\ i) = \mathit{dimap}\ g\ h\ .\ \mathit{dimap}\ f\ i$$

The derived *Functor* instance obtained by fixing the contravariant argument will hence satisfy the functor laws, provided that the difunctor laws are satisfied.

Meijer and Hutton [66] showed that it is possible to perform recursion over difunctor terms:

$$\begin{aligned}
&\textbf{data}\ \mathit{Term}_{\mathrm{MH}}\ f = \mathit{In}_{\mathrm{MH}}\ \{\mathit{out}_{\mathrm{MH}} :: f\ (\mathit{Term}_{\mathrm{MH}}\ f)\ (\mathit{Term}_{\mathrm{MH}}\ f)\}\\
&\mathit{cata}_{\mathrm{MH}} :: \mathit{Difunctor}\ f \Rightarrow (f\ b\ a \to a) \to (b \to f\ a\ b) \to \mathit{Term}_{\mathrm{MH}}\ f \to a\\
&\mathit{cata}_{\mathrm{MH}}\ \phi\ \psi = \phi\ .\ \mathit{dimap}\ (\mathit{ana}_{\mathrm{MH}}\ \phi\ \psi)\ (\mathit{cata}_{\mathrm{MH}}\ \phi\ \psi)\ .\ \mathit{out}_{\mathrm{MH}}\\
&\mathit{ana}_{\mathrm{MH}} :: \mathit{Difunctor}\ f \Rightarrow (f\ b\ a \to a) \to (b \to f\ a\ b) \to b \to \mathit{Term}_{\mathrm{MH}}\ f\\
&\mathit{ana}_{\mathrm{MH}}\ \phi\ \psi = \mathit{In}_{\mathrm{MH}}\ .\ \mathit{dimap}\ (\mathit{cata}_{\mathrm{MH}}\ \phi\ \psi)\ (\mathit{ana}_{\mathrm{MH}}\ \phi\ \psi)\ .\ \psi
\end{aligned}$$

With Meijer and Hutton's approach, however, in order to lift an algebra $\phi :: f\ b\ a \to a$ to a catamorphism, we also need to supply the *inverse coalgebra* $\psi :: b \to f\ b\ a$.

That is, in order to write a pretty printer we must supply a parser, which is not feasible—or perhaps even possible—in practice.

Fortunately, Fegaras and Sheard [25] realised that if the embedded functions within terms are *parametric*, then the inverse coalgebra is only used in order to *undo* computations performed by the algebra, since parametric functions can only "push around their arguments" without examining them. The solution proposed by Fegaras and Sheard is to add a *placeholder* to the structure of terms, which acts as a right-inverse of the catamorphism:[2]

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Term}_{\mathrm{FS}}\ f\ a = \mathit{In}_{\mathrm{FS}}\ (f\ (\mathit{Term}_{\mathrm{FS}}\ f\ a)\ (\mathit{Term}_{\mathrm{FS}}\ f\ a)) \mid \mathit{Place}\ a\\
&\mathit{cata}_{\mathrm{FS}} :: \mathit{Difunctor}\ f \Rightarrow (f\ a\ a \to a) \to \mathit{Term}_{\mathrm{FS}}\ f\ a \to a\\
&\mathit{cata}_{\mathrm{FS}}\ \phi\ (\mathit{In}_{\mathrm{FS}}\ t)\ \ = \phi\ (\mathit{dimap}\ \mathit{Place}\ (\mathit{cata}_{\mathrm{FS}}\ \phi)\ t)\\
&\mathit{cata}_{\mathrm{FS}}\ \phi\ (\mathit{Place}\ x) = x
\end{aligned}
$$

We can then for instance define a signature for lambda terms, and a function that calculates the number of bound variables occurring in a term, as follows (the example is adopted from Washburn and Weirich [116]):

$$
\begin{aligned}
&\mathbf{data}\ T\ a\ b = \mathit{Lam}\ (a \to b) \mid \mathit{App}\ b\ b\\
&\quad \text{-- } T \text{ is a difunctor, we omit the instance declaration}\\
&\phi :: T\ \mathit{Int}\ \mathit{Int} \to \mathit{Int}\\
&\phi\ (\mathit{Lam}\ f)\ \ \ \ = f\ 1\\
&\phi\ (\mathit{App}\ x\ y) = x + y\\
&\mathit{countVar} :: \mathit{Term}_{\mathrm{FS}}\ T\ \mathit{Int} \to \mathit{Int}\\
&\mathit{countVar} = \mathit{cata}_{\mathrm{FS}}\ \phi
\end{aligned}
$$

In the $\mathit{Term}_{\mathrm{FS}}$ encoding above, however, parametricity of the embedded functions is not guaranteed. More specifically, the type allows for three kinds of *exotic terms* [116], that is values in the meta language that do not correspond to terms in the object language:

$$
\begin{aligned}
&\mathit{badPlace} :: \mathit{Term}_{\mathrm{FS}}\ T\ \mathit{Bool}\\
&\mathit{badPlace} = \mathit{In}_{\mathrm{FS}}\ (\mathit{Place}\ \mathit{True})\\
&\mathit{badCata} :: \mathit{Term}_{\mathrm{FS}}\ T\ \mathit{Int}\\
&\mathit{badCata} = \mathit{In}_{\mathrm{FS}}\ (\mathit{Lam}\ (\lambda x \to \mathbf{if}\ \mathit{countVar}\ x \equiv 0\ \mathbf{then}\ x\ \mathbf{else}\ \mathit{Place}\ 0))\\
&\mathit{badCase} :: \mathit{Term}_{\mathrm{FS}}\ T\ a\\
&\mathit{badCase} = \mathit{In}_{\mathrm{FS}}\ (\mathit{Lam}\ (\lambda x \to \mathbf{case}\ x\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad\qquad \mathit{Term}_{\mathrm{FS}}\ (\mathit{App}\ \_\ \_) \to \mathit{Term}_{\mathrm{FS}}\ (\mathit{App}\ x\ x)\\
&\qquad\qquad\qquad\qquad\qquad \_ \qquad\qquad\qquad\quad\ \to x))
\end{aligned}
$$

Fegaras and Sheard showed how to avoid exotic terms by means of a custom type system. Washburn and Weirich [116] later showed that exotic terms can be avoided in a Haskell encoding via type parametricity and an abstract type of terms: terms are restricted to the type $\forall\ a\ .\ \mathit{Term}_{\mathrm{FS}}\ f\ a$, and the constructors of $\mathit{Term}_{\mathrm{FS}}$ are hidden. Parametricity rules out *badPlace* and *badCata*, while the use of an abstract type rules out *badCase*.

[2]Actually, Fegaras and Sheard do not use difunctors, but the given definition corresponds to their encoding.

### 5.3.2 Parametric Higher-Order Abstract Syntax

While the approach of Washburn and Weirich effectively rules out exotic terms in Haskell, we prefer a different encoding that relies on type parametricity only, and not an abstract type of terms. Our solution is inspired by Chlipala's *parametric higher-order abstract syntax* (PHOAS) [19]. PHOAS is similar to the restricted form of HOAS that we saw above; however, Chlipala makes the parametricity explicit in the definition of terms by distinguishing between the type of bound variables and the type of recursive terms. In Chlipala's approach, an algebraic data type encoding of lambda terms *LTerm* can effectively be defined via an auxiliary data type *LTrm* of "preterms" as follows:

$$\textbf{type}\ \mathit{LTerm}\ = \forall\ a\ .\ \mathit{LTrm}\ a$$
$$\textbf{data}\ \mathit{LTrm}\ a = \mathit{Lam}\ (a \to \mathit{LTrm}\ a) \mid \mathit{Var}\ a \mid \mathit{App}\ (\mathit{LTrm}\ a)\ (\mathit{LTrm}\ a)$$

The definition of *LTerm* guarantees that all functions embedded via *Lam* are parametric, and likewise that *Var*—Fegaras and Sheard's *Place*—can only be applied to variables bound by an embedded function. Atkey [7] showed that the encoding above adequately captures closed lambda terms modulo $\alpha$-equivalence, assuming that there is no infinite data and that all embedded functions are total.

#### 5.3.2.1 Parametric Terms

In order to transfer Chlipala's idea to non-recursive signatures and catamorphisms, we need to distinguish between covariant and contravariant uses of the recursive parameter. But this is exactly what difunctors do! We therefore arrive at the following definition of terms over difunctors:

$$\textbf{newtype}\ \mathit{Term}\ f = \mathit{Term}\ \{\mathit{unTerm} :: \forall\ a\ .\ \mathit{Trm}\ f\ a\}$$
$$\textbf{data}\ \mathit{Trm}\ f\ a \quad = \mathit{In}\ (f\ a\ (\mathit{Trm}\ f\ a)) \mid \mathit{Var}\ a \quad \text{-- ``preterm''}$$

Note the difference in *Trm* compared to $Term_{\mathrm{FS}}$ (besides using the name *Var* rather than *Place*): the contravariant argument to the difunctor $f$ is not the type of terms *Trm* $f$ $a$, but rather a parametrised type $a$, which we quantify over at top-level to ensure parametricity. Hence, the only way to use a bound variable is to wrap it in a *Var* constructor—it is not possible to inspect the parameter. This representation more faithfully captures—we believe—the restricted form of HOAS than the representation of Washburn and Weirich: in our encoding it is explicit that bound variables are merely placeholders, and not the same as terms. Moreover, in some cases we actually *need* to inspect the structure of terms in order to define term transformations—we will see such an example in Section 5.3.2.3. With an abstract type of terms, this is not possible as Washburn and Weirich note [116].

Before we define algebras and catamorphisms, we lift the ideas underlying CDTs to *parametric compositional data types* (PCDTs), namely coproducts and implicit injections. Fortunately, the constructions of Section 5.2 are straightforwardly generalised (the instances for $:\prec:$ are exactly as in *data types à la carte* [104], so we omit them here):

$$\textbf{data}\ (f :+: g)\ a\ b = \mathit{Inl}\ (f\ a\ b) \mid \mathit{Inr}\ (g\ a\ b)$$

```
instance (Difunctor f, Difunctor g) ⇒ Difunctor (f :+: g) where
  dimap φ ψ (Inl x) = Inl (dimap φ ψ x)
  dimap φ ψ (Inr x) = Inr (dimap φ ψ x)

class sub :≺: sup where
  inj  :: sub a b → sup a b
  proj :: sup a b → Maybe (sub a b)

inject :: (g :≺: f) ⇒ g a (Trm f a) → Trm f a
inject = In . inj

project :: (g :≺: f) ⇒ Trm f a → Maybe (g a (Trm f a))
project (Term t) = proj t
project (Var _)  = Nothing
```

We can then recast our previous signatures as difunctors, but using PHOAS rather than explicit representations of variable names:

```
data Lam a b = Lam (a → b)        data Plus a b = Plus b b
data App a b = App b b            data Let a b  = Let b (a → b)
data Lit a b = Lit Int            data Err a b  = Err
type Sig     = Lam :+: App :+: Lit :+: Plus :+: Err :+: Let
type Sig′    = Lam :+: App :+: Lit :+: Plus :+: Err
```

Finally, we can automatically derive instance declarations for *Difunctor* as well as smart constructor definitions that comprise the injections as for CDTs [11]. However, in order to avoid the explicit *Var* constructor, we insert *dimap Var id* into the declarations, for instance:

```
iLam :: (Lam :≺: f) ⇒ (Trm f a → Trm f a) → Trm f a
iLam f = inject (dimap Var id (Lam f))   -- (= inject (Lam (f . Var)))
```

Using *iLam* we then need to be aware, though, that even if it takes a function $Trm\ f\ a \to Trm\ f\ a$ as argument, the input to that function will always be of the form *Var x* by *construction*. We can now again represent terms such as **let** $x = 2$ **in** $(\lambda y.y + x)\ 3$ compactly as follows:

```
e :: Term Sig
e = Term (iLet (iLit 2) (λx → (iLam (λy → y ‘iPlus‘ x) ‘iApp‘ iLit 3)))
```

### 5.3.2.2 Algebras and Catamorphisms

Given the representation of terms as fixed points of difunctors, we can now define algebras and catamorphisms:

```
type Alg f a = f a a → a
cata :: Difunctor f ⇒ Alg f a → Term f → a
cata φ (Term t) = cat t
  where cat (In t)  = φ (fmap cat t)   -- recall: fmap = dimap id
        cat (Var x) = x
```

The definition of *cata* above is essentially the same as $cata_{\mathrm{FS}}$. The only difference is that bound variables within terms are already wrapped in a *Var* constructor. Therefore, the contravariant argument to *dimap* is the identity function, and we consequently use the derived function *fmap* instead.

With these definitions we can now recast the modular pretty printer from Section 5.2.1 to the new difunctor signatures. However, since we now use a higher-order encoding, we need to generate variable names for printing. We therefore arrive at the following definition (the example is adopted from Washburn and Weirich [116], but we use streams rather than lists to represent the sequence of available variable names):

```
data Stream a = Cons a (Stream a)

class Pretty f where
  prettyAlg :: Alg f (Stream String → String)

  -- instance that lifts Pretty to coproducts omitted

pretty :: (Difunctor f, Pretty f) ⇒ Term f → String
pretty t = cata prettyAlg t (names 1)
  where names n = Cons ('x' : show n) (names (n + 1))

instance Pretty Lam where
  prettyAlg (Lam f) (Cons x xs) = "(\\" ++ x ++ ". " ++
                                    f (const x) xs ++ ")"

instance Pretty App where
  prettyAlg (App e1 e2) xs = "(" ++ e1 xs ++ " " ++ e2 xs ++ ")"

instance Pretty Lit where
  prettyAlg (Lit n) _ = show n

instance Pretty Plus where
  prettyAlg (Plus e1 e2) xs = "(" ++ e1 xs ++ " + " ++ e2 xs ++ ")"

instance Pretty Let where
  prettyAlg (Let e1 e2) (Cons x xs) = "let " ++ x ++ " = " ++ e1 xs ++
                                       " in " ++ e2 (const x) xs

instance Pretty Err where
  prettyAlg Err _ = "error"
```

With these definitions we then have that *pretty e* evaluates to the string `let x1 = 2 in ((\x2. (x2 + x1)) 3)`.

### 5.3.2.3 Term Transformations

The pretty printer is an example of a modular computation over a PCDT. However, we also want to define computations over PCDTs that *construct* PCDTs, such as the desugaring transformation. That is, we want to construct functions of the form $Term\ f \to Term\ g$, which means that we must construct functions of the form $(\forall\ a\ .\ Trm\ f\ a) \to (\forall\ a\ .\ Trm\ g\ a)$. But such a function can be obtained from a function of the type $\forall\ a\ .\ (Trm\ f\ a \to Trm\ g\ a)$, which motivates the following definition of the desugaring algebra type class:

```
class Desug f g where
  desugAlg :: ∀ a . Alg f (Trm g a)   -- not Alg f (Term g) !
  -- instance that lifts Desug to coproducts omitted
desug :: (Difunctor f, Desug f g) ⇒ Term f → Term g
desug t = Term (cata desugAlg t)
```

The algebra type class above is a *multi-parameter type class*. That is, it is parametrised both by the domain signature $f$ and the codomain signature $g$. We do this in order to obtain a desugaring function that is also modular in the codomain, similar to the evaluation function for vanilla CDTs [11].

We can now define the instances of *Desug* for the six signatures in order to obtain the desugaring function. However, by utilising overlapping instances we can make do with just two instances:

```
instance (Difunctor f, f :≺: g) ⇒ Desug f g where
  desugAlg = inject . dimap Var id   -- default instance for core signatures
instance (App :≺: f, Lam :≺: f) ⇒ Desug Let f where
  desugAlg (Let e₁ e₂) = iLam e₂ `iApp` e₁
```

Given a term $e$ :: *Term Sig*, we then have that *desug e* :: *Term Sig′*, that is the type shows that indeed all syntactic sugar has been removed.

Whereas the desugaring transformation shows that we can construct PCDTs from PCDTs in a modular fashion, we did not make use of the fact that PCDTs can be inspected. That is, the desugaring transformation does not inspect the recursively computed values, compare the instance for *Let*. However, in order to implement the constant folding transformation, we actually need to inspect recursively computed PCDTs. We again utilise overlapping instances:

```
class Constf f g where
  constfAlg :: ∀ a . Alg f (Trm g a)
  -- instance that lifts Constf to coproducts omitted
constf :: (Difunctor f, Constf f g) ⇒ Term f → Term g
constf t = Term (cata constfAlg t)
instance (Difunctor f, f :≺: g) ⇒ Constf f g where
  constfAlg = inject . dimap Var id   -- default instance
instance (Plus :≺: f, Lit :≺: f) ⇒ Constf Plus f where
  constfAlg (Plus e₁ e₂) = case (project e₁, project e₂) of
    (Just (Lit n), Just (Lit m)) → iLit (n + m); _ → e₁ `iPlus` e₂
```

Note that with the default instance we not only have constant folding for the core language, but also for the full language, that is *constf* has both the types *Term Sig′* → *Term Sig′* and *Term Sig* → *Term Sig*.

## 5.4 Monadic Computations

In the last section, we demonstrated how to extend CDTs with parametric higher-order abstract syntax, and how to perform modular, recursive computations over terms containing binders. In this section we investigate monadic computations over PCDTs, and why they are problematic compared to CDTs.

### 5.4.1 Monadic Computations over CDTs

We have previously shown how to perform monadic computations over CDTs [11] by utilising the standard type class *Traversable*[3]:

```
type AlgM m f a = f a → m a
class Functor f ⇒ Traversable f where
  sequence :: Monad m ⇒ f (m a) → m (f a)
cataM :: (Traversable f, Monad m) ⇒ AlgM m f a → Term f → m a
cataM ϕ = ϕ ⋘ sequence . fmap (cataM ϕ) . out
```

*AlgM m f a* represents the type of monadic algebras [27] over $f$ and $m$, with carrier $a$, which is different from *Alg f (m a)* since the monad only occurs in the codomain of the monadic algebra. *cataM* is obtained from *cata* in Section 5.2 by performing *sequence* after applying *fmap* and replacing function composition with monadic function composition ⋘. Monadic algebras are useful for instance if we want to recursively project a term over a compound signature to a smaller signature:

```
deepProject :: (Traversable g, f :≺: g) ⇒ Term f → Maybe (Term g)
deepProject = cataM (liftM In . proj)
```

### 5.4.2 Monadic Computations over PCDTs

Turning back to parametric terms, we can apply the same idea to difunctors yielding the following definition of monadic algebras:

```
type AlgM m f a = f a a → m a
```

Similarly, we can easily generalise *Traversable* and *cataM* to difunctors:

```
class Difunctor f ⇒ Ditraversable f where
  disequence :: Monad m ⇒ f a (m b) → m (f a b)
cataM :: (Ditraversable f, Monad m) ⇒ AlgM m f a → Term f → m a
cataM ϕ (Term t) = cat t
  where cat (In t)    = disequence (fmap cat t) ≫= ϕ
        cat (Var x) = return x
```

Unfortunately, *cataM* only works for difunctors that do not use the contravariant argument. To see why this is the case, reconsider the *Lam* constructor; in order to define an instance of *Ditraversable* for *Lam* we must write a function of the type:

```
disequence :: Monad m ⇒ Lam a (m b) → m (Lam a b)
```

Since *Lam* is isomorphic to the function type constructor →, this is equivalent to a function of the type:

```
∀ a b m . Monad m ⇒ (a → m b) → m (a → b)
```

[3]We have omitted methods from the definition of *Traversable* that are not necessary for our purposes.

We cannot hope to be able to construct a meaningful combinator of that type. Intuitively, in a function of type $a \rightarrow m\ b$, the monadic effect of the result can depend on the input of type $a$. The monadic effect of a monadic value of type $m\ (a \rightarrow b)$ is not dependent on such input. For example, think of a state transformer monad $ST$ with state $S$ and its put function $put :: S \rightarrow ST\ ()$. What would be the corresponding transformation to a monadic value of type $ST\ (S \rightarrow ())$? Hence, *cataM* does not extend to terms with binders, but it still works for terms without binders as in vanilla CDTs [11].

#### 5.4.2.1 Monadic Interpretation

While integrating the sequencing of monadic effects into the catamorphic recursion scheme is difficult, we can nevertheless perform monadic computations over PCDTs when we sequence the monad explicitly in the algebra definition. That is, if we instead consider an ordinary algebra with a monadic carrier *Alg f* $(m\ a)$.

The semantic domain of our interpreter can be described by the following algebraic data type (we could also use a PCDT, but we use an ordinary algebraic data type for simplicity):

```
data Sem m = Fun (Sem m → m (Sem m)) | Int Int
```

We parametrise the domain by a monad $m$ in order to separate computations from pure values. Note that the monad only occurs in the codomain of *Fun*—if we want call-by-name semantics rather than call-by-value semantics we simply add $m$ also to the domain. Additionally, the monad is also used to signal errors as well as indicating that the interpreter gets stuck. We can now define our modular call-by-value interpreter:

```
class Monad m ⇒ Eval m f where
  evalAlg :: Alg f (m (Sem m))
  -- instance that lifts Eval to coproducts omitted
eval :: (Difunctor f, Eval m f) ⇒ Term f → m (Sem m)
eval = cata evalAlg
instance Monad m ⇒ Eval m Lam where
  evalAlg (Lam f) = return (Fun (f . return))
instance MonadError String m ⇒ Eval m App where
  evalAlg (App mx my) = do x ← mx
                           case x of Fun f → my >>= f
                                     _     → throwError "stuck"
instance Monad m ⇒ Eval m Lit where
  evalAlg (Lit n) = return (Int n)
instance MonadError String m ⇒ Eval m Plus where
  evalAlg (Plus mx my) = do x ← mx; y ← my;
                            case (x, y) of
                              (Int n, Int m) → return (Int (n + m))
                              _              → throwError "stuck"
instance MonadError String m ⇒ Eval m Err where
  evalAlg Err = throwError "error"
```

In order to indicate errors in the course of the evaluation, we require the monad to provide a method to throw an error. To this end, we use the type class *MonadError*. Note how the modular design allows us to require the stricter constraint *MonadError String m* only for the cases where it is needed. This modularity of effects will become quite useful when we will rule out `"stuck"` errors in Section 5.6.

With the definition of the interpreter above, we have that *eval* (*desug e*) evaluates to the value *Right* (*Int* 5) as expected, where *e* is as of page 131 and *m* is the *Either String* monad. Moreover, we also have that $0 + \mathbf{error}$ and $0 + \lambda x.x$ evaluate to *Left* `"error"` and *Left* `"stuck"` respectively.

Note that if we want to implement a call-by-name interpreter instead, it is in fact crucial that we do not use a recursion scheme that performs the sequencing of monadic effects like *cataM* does. In order to implement call-by-name semantics, we must be able to avoid monadic effects that are not needed.

## 5.5 Contexts and Term Homomorphisms

While the generality of catamorphisms makes them a powerful tool for modular function definitions, their generality at the same time inhibits flexibility and reusability. However, the full generality of catamorphisms is not always needed in the case of term algebras, that is algebras with carrier types of the form *Term g*. To this end, we have previously studied term homomorphisms [11] as a restricted form of term algebras. In this section, we redevelop term homomorphisms for PCDTs.

### 5.5.1 From Terms to Contexts and back

The crucial idea behind term homomorphisms is to generalise terms to *contexts*, that is terms with *holes*. Following previous work [11], we define the generalisation of terms with holes as a *generalised algebraic data type (GADT)* [97] with *phantom types Hole* and *NoHole*:

```
data Hole
data NoHole
data Cxt :: * → (* → * → *) → * → * → * where
  In   :: f a (Cxt h f a b) → Cxt h    f a b
  Var  :: a                 → Cxt h    f a b
  Hole :: b                 → Cxt Hole f a b
```

The first argument to *Cxt* is a phantom type indicating whether the term contains holes or not. A context can thus be defined as:

```
type Context = Cxt Hole
```

That is, contexts *may* contain holes. On the other hand, terms must not contain holes, so we can recover our previous definition of preterms *Trm* as follows:

```
type Trm f a = Cxt NoHole f a ()
```

The definition of *Term* remains unchanged. This representation of contexts and preterms allows us to uniformly define functions that work on both types. For example, the function *inject* now has the type:

```
inject :: (g :≺: f) ⇒ g a (Cxt h f a b) → Cxt h f a b
```

### 5.5.2 Term Homomorphisms

The type *Context* allows us to realise term homomorphisms as functions of the type:

```
type Hom f g = ∀ a b . f a b → Context g a b
```

A function $\rho$::*Hom f g* is a transformation of constructors from $f$ applied to arbitrary arguments into a context over $g$, that is a term over $g$ that may embed values taken from the arguments of the $f$-constructor. The parametric polymorphism of the type guarantees that the arguments to the $f$-constructor cannot be inspected but only embedded into the result context. In order to apply term homomorphisms to terms, we need an auxiliary function that merges nested contexts:

```
appCxt :: Difunctor f ⇒ Context f a (Cxt h f a b) → Cxt h f a b
appCxt (In t)     = In (fmap appCxt t)
appCxt (Var x)  = Var x
appCxt (Hole h) = h
```

Given a context that has terms embedded in its holes, we obtain a term as a result; given a context with embedded contexts, the result is again a context.

Using the combinator above, we can now apply a term homomorphism to a preterm—or more generally, to a context:

```
appHom :: (Difunctor f, Difunctor g)
              ⇒ Hom f g → Cxt h f a b → Cxt h g a b
appHom ρ (In t)     = appCxt (ρ (fmap (appHom ρ) t))
appHom ρ (Var x)  = Var x
appHom ρ (Hole h) = Hole h
```

From *appHom* we can then obtain the actual transformation on terms as follows:

```
appTHom :: (Difunctor f, Difunctor g) ⇒ Hom f g → Term f → Term g
appTHom ρ (Term t) = Term (appHom ρ t)
```

Before we describe the benefits of term homomorphisms over term algebras, we reconsider the desugaring transformation from Section 5.3.2.3, but as a term homomorphism rather than a term algebra:

```
class Desug f g where
  desugHom :: Hom f g
  -- instance that lifts Desug to coproducts omitted
desug :: (Difunctor f, Difunctor g, Desug f g) ⇒ Term f → Term g
desug = appTHom desugHom
instance (Difunctor f, Difunctor g, f :≺: g) ⇒ Desug f g where
  desugHom = In . fmap Hole . inj   -- default instance for core signatures
instance (App :≺: f, Lam :≺: f) ⇒ Desug Let f where
  desugHom (Let e₁ e₂) = inject (Lam (Hole . e₂)) `iApp` Hole e₁
```

Note how, in the instance for *Let*, the constructor *Hole* is used to embed arguments of the constructor *Let*, viz. $e_1$ and $e_2$, into the context that is constructed as the result.

As for the desugaring function in Section 5.3.2.3, we utilise overlapping instances to provide a default translation for the signatures that need not be translated. The definitions above yield the desired desugaring function $desug :: Term\ Sig \rightarrow Term\ Sig'$.

### 5.5.3 Transforming and Combining Term Homomorphisms

In the following we shall shortly describe what we actually gain by adopting the term homomorphism approach. First, term homomorphisms enable automatic propagation of annotations, where annotations are added via a restricted difunctor product, namely a product of a difunctor $f$ and a constant $c$:

$$\textbf{data}\ (f :\&: c)\ a\ b = f\ a\ b :\&: c$$

For instance, the type of ASTs of our language where each node is annotated with source positions is captured by the type $Term\ (Sig :\&: SrcPos)$. With a term homomorphism $Hom\ f\ g$ we automatically get a lifted version $Hom\ (f :\&: c)\ (g :\&: c)$, which propagates annotations from the input to the output. Hence, from our desugaring function in the previous section we automatically get a lifted function on parse trees $Term\ (Sig :\&: SrcPos) \rightarrow Term\ (Sig' :\&: SrcPos)$, which propagates source positions from the syntactic sugar to the core constructs. We omit the details here, but note that the constructions for CDTs [11] carry over straightforwardly to PCDTs.

The second motivation for introducing term homomorphisms is deforestation [112]. As we have shown previously [11], it is not possible to fuse two term algebras in order to only traverse the term once. That is, we cannot perform a transformation:

$$cata\ \phi_1\ .\ cata\ \phi_2 \rightsquigarrow cata\ (\phi_1 \circledcirc \phi_2)$$

when $\phi_1$ and $\phi_2$ are term algebras and $\circledcirc$ is the desired term algebra composition. However, with term homomorphism we can:

$$\begin{aligned}
&(\circledcirc) :: (Difunctor\ g, Difunctor\ h) \Rightarrow Hom\ g\ h \rightarrow Hom\ f\ g \rightarrow Hom\ f\ h\\
&\rho_1 \circledcirc \rho_2 = appHom\ \rho_1\ .\ \rho_2
\end{aligned}$$

In fact, we can compose an arbitrary algebra with a term homomorphism:

$$\begin{aligned}
&(\boxdot) :: Difunctor\ g \Rightarrow Alg\ g\ a \rightarrow Hom\ f\ g \rightarrow Alg\ f\ a\\
&\phi \boxdot \rho = free\ \phi\ .\ \rho\\
&\quad \textbf{where}\ free\ \phi\ (In\ t) \quad = \phi\ (fmap\ (free\ \phi)\ t)\\
&\qquad\qquad free\ \_\ (Var\ x)\ = x\\
&\qquad\qquad free\ \_\ (Hole\ h) = h
\end{aligned}$$

Hence in order to evaluate a term with syntactic sugar, rather than composing *eval* and *desug*, we can use $cata\ (evalAlg \boxdot desugHom)$ that only traverses the term once. This transformation can be automated and our experimental results for CDTs show that the thus obtained speedup is significant [11].

## 5.6 Generalised Parametric Compositional Data Types

In this section we briefly describe how to lift the construction of mutually recursive data types and—more generally—GADTs from CDTs to PCDTs. The construction is based on the work of Johann and Ghani [53]. For CDTs the generalisation, roughly speaking, amounts to lifting functors to (generalised) *higher-order functors* [53], and functions on terms to *natural transformations*, as shown earlier [11]:

```
type a →̇ b = ∀ i . a i → b i
class HFunctor f where
  hfmap :: a →̇ b → f a →̇ f b
```

Now, signatures are of the kind $(* \to *) \to * \to *$, rather than $* \to *$, which reflects the fact that signatures are now *type families*, and so are terms (or contexts in general). At the algebra level, carriers are of the kind $* \to *$. Since the signatures will be defined as GADTs, we effectively deal with *many-sorted algebras*:

```
type Alg f a = f a →̇ a
```

If a subterm has the type index $i$, then the value computed recursively by the catamorphism will have the type $a\ i$. The coproduct :+: and the automatic injections :≺: carry over straightforwardly from functors to higher-order functors [11].

In order to lift the ideas from CDTs to PCDTs, we need to consider indexed difunctors. This prompts the notion of *higher-order difunctors*:

```
class HDifunctor f where
  hdimap :: (a →̇ b) → (c →̇ d) → f b c →̇ f a d
instance HDifunctor f ⇒ HFunctor (f a) where
  hfmap = hdimap id
```

Note the familiar pattern from ordinary PCDTs: a higher-order difunctor gives rise to a higher-order functor when the contravariant argument is fixed.

To illustrate higher-order difunctors, consider a modular GADT encoding of our core language:

```
data TArrow i j
data TInt
data Lam :: (∗ → ∗) → (∗ → ∗) → ∗ → ∗ where
  Lam :: (a i → b j) → Lam a b (i ‘TArrow‘ j)
data App :: (∗ → ∗) → (∗ → ∗) → ∗ → ∗ where
  App :: b (i ‘TArrow‘ j) → b i → App a b j
data Lit :: (∗ → ∗) → (∗ → ∗) → ∗ → ∗ where
  Lit :: Int → Lit a b TInt
data Plus :: (∗ → ∗) → (∗ → ∗) → ∗ → ∗ where
  Plus :: b TInt → b TInt → Plus a b TInt
data Err :: (∗ → ∗) → (∗ → ∗) → ∗ → ∗ where
  Err :: Err a b i
type Sig′ = Lam :+: App :+: Lit :+: Plus :+: Err
```

Note, in particular, the type of *Lam*: now the bound variable is typed!

We use *TArrow* and *TInt* as label types of the GADT definitions above. The preference of these fresh types over Haskell's → and *Int* is meant to emphasise that these phantom types are only labels that represent the type constructors of our object language.

We use the coproduct :+: of higher-order difunctors above to combine signatures, which is easily defined, and as for CDTs it is straightforward to lift instances of *HDifunctor* for $f$ and $g$ to an instance for $f$ :+: $g$. Similarly, we can generalise the relation :≺: from difunctors to higher-order difunctors, so we omit its definition here.

The type of generalised parametric (pre)terms can now be constructed as an indexed type:

```
newtype Term f i = Term {unTerm :: ∀ a . Trm f a i}
data Trm f a i     = In (f a (Trm f a) i) | Var (a i)
```

Moreover, we use smart constructors as for PCDTs to compactly represent terms, for instance:

```
e :: Term Sig′ TInt
e = Term (iLam (λx → x ‘iPlus‘ x) ‘iApp‘ iLit 2)
```

Finally, we can lift algebras and their induced catamorphisms, by lifting the definitions in Section 5.3.2.2 via natural transformations and higher-order difunctors:

```
type Alg f a = f a a ⇀̇ a
cata :: HDifunctor f ⇒ Alg f a → Term f ⇀̇ a
cata φ (Term t) = cat t
  where cat (In t)   = φ (hfmap cat t)   -- recall: hfmap = hdimap id
        cat (Var x) = x
```

With the definitions above we can now define a call-by-value interpreter for our typed example language. To this end, we have to provide a type-level function that, for a given object language type constructed from *TArrow* and *TInt*, selects the corresponding subset of the semantic domain *Sem m* from Section 5.4.2.1. This can be achieved via *type families* [96]:

```
type family Sem (m :: ∗ → ∗) i
type instance Sem m (i ‘TArrow‘ j) = Sem m i → m (Sem m j)
type instance Sem m TInt           = Int
```

The type *Sem m t* is obtained from an object language type $t$ by replacing each function type $t_1$ ‘*TArrow*‘ $t_2$ occurring in $t$ with *Sem m* $t_1$ → $m$ (*Sem m* $t_2$) and each *TInt* with *Int*.

In order to make this into a proper type function and simultaneously add the monad $m$ at the top level, we define a **newtype** $M$:

```
newtype M m i = M {unM :: m (Sem m i)}
class Monad m ⇒ Eval m f where
  evalAlg :: f (M m) (M m) i → m (Sem m i)
```

```
    -- M . evalAlg :: Alg f (M m) is the actual algebra
eval :: (Monad m, HDifunctor f, Eval m f) ⇒ Term f i → m (Sem m i)
eval = unM . cata (M . evalAlg)
```

We can then provide the instance declarations for the signatures of the core language, and effectively obtain a tagless, modular, and extendable monadic interpreter:

```
instance Monad m ⇒ Eval m Lam where
  evalAlg (Lam f) = return (unM . f . M . return)
instance Monad m ⇒ Eval m App where
  evalAlg (App (M mf) (M mx)) = do f ← mf; x ← mx; f x
instance Monad m ⇒ Eval m Lit where
  evalAlg (Lit n) = return n
instance Monad m ⇒ Eval m Plus where
  evalAlg (Plus (M mx) (M my)) = do x ← mx; y ← my; return (x + y)
instance MonadError String m ⇒ Eval m Err where
  evalAlg Err = throwError "error"
```

With these definitions we then have, for instance, that *eval e* :: *Either String Int* evaluates to the value *Right* 4. Due to the fact that we now have a typed language, the *Err* constructor is the only source of an erroneous computation—the interpreter cannot get stuck. Moreover, since the modular specification of the interpreter only enforces the constraint *MonadError String m* for the signature *Err*, then the term *e* can in fact be interpreted in the identity monad, rather than the *Either String* monad. Consequently, we know statically that the evaluation of *e* cannot fail!

Note that computations over generalised PCDTs are not limited to the tagless approach that we have illustrated above. We could have easily reformulated the semantic domain *Sem m* from Section 5.4.2.1 as a GADT to use it as the carrier of a many-sorted algebra. Other natural carriers for many-sorted algebras are the type families of terms *Term f*, of course.

Other concepts that we have introduced for vanilla PCDTs before can be transferred straightforwardly to generalised PCDTs in the same fashion. This includes contexts and term homomorphisms.

## 5.7 Practical Considerations

The motivation for introducing CDTs was to make Swierstra's *data types à la carte* [104] readily useful in practice. Besides extending *data types à la carte* with various aspects, such as monadic computations and term homomorphisms, the CDTs library provides all the generic functionality as well as automatic derivation of boilerplate code. With (generalised) PCDTs we have followed that path. Our library provides Template Haskell [99] code to automatically derive instances of the required type classes, such as *Difunctor* and *Ditraversable*, as well as smart constructors and lifting of algebra type classes to coproducts. Moreover, our library supports automatic derivation of standard type classes *Show*, *Eq*, and *Ord* for terms, similar to Haskell's **deriving** mechanism. We show how to derive instances of *Eq* in the following subsection. *Ord* follows the same approach, and *Show* follows an approach similar to the pretty printer in Section 5.3.2.2, but using the monad *FreshM* that is also used to determine equality, as we shall see below.

Figure 5.1 provides the complete source code needed to implement our example language from Section 5.2.1. Note that we have derived *Eq* and *Show* instances for terms of the language—in particular the term $e$ is printed as `Let (Lit 2) (\a -> App (Lam (\b -> Plus b a)) (Lit 3))`.

## 5.7.1 Equality

A common pattern when developing in Haskell is to derive instances of the type class *Eq*, for instance in order to test the desugaring transformation in Section 5.3.2.3. While the use of PHOAS ensures that all functions are invariant under $\alpha$-renaming, we still have to devise an algorithm that decides $\alpha$-equivalence. To this end, we will turn the rather elusive representation of bound variables via functions into a concrete form.

In order to obtain concrete representations of bound variables, we provide a method for generating fresh variable names. This is achieved via a monad *FreshM* offering the following operations:

$$\begin{aligned}
&\mathit{withName} :: (\mathit{Name} \to \mathit{FreshM}\ a) \to \mathit{FreshM}\ a\\
&\mathit{evalFreshM} :: \mathit{FreshM}\ a \to a
\end{aligned}$$

*FreshM* is an abstraction of an infinite sequence of fresh names. The function *withName* provides a fresh name. Names are represented by the abstract type *Name*, which implements instances of *Show*, *Eq*, and *Ord*.

We first introduce a variant of the type class *Eq* that uses the *FreshM* monad:

$$\begin{aligned}
&\textbf{class}\ \mathit{PEq}\ a\ \textbf{where}\\
&\quad \mathit{peq} :: a \to a \to \mathit{FreshM}\ \mathit{Bool}
\end{aligned}$$

This type class is used to define the type class *EqD* of equatable difunctors, which lifts to coproducts:

$$\begin{aligned}
&\textbf{class}\ \mathit{EqD}\ f\ \textbf{where}\\
&\quad \mathit{eqD} :: \mathit{PEq}\ a \Rightarrow f\ \mathit{Name}\ a \to f\ \mathit{Name}\ a \to \mathit{FreshM}\ \mathit{Bool}\\
&\textbf{instance}\ (\mathit{EqD}\ f, \mathit{EqD}\ g) \Rightarrow \mathit{EqD}\ (f :+: g)\ \textbf{where}\\
&\quad \mathit{eqD}\ (\mathit{Inl}\ x)\ (\mathit{Inl}\ y) = x\ \text{`}\mathit{eqD}\text{`}\ y\\
&\quad \mathit{eqD}\ (\mathit{Inr}\ x)\ (\mathit{Inr}\ y) = x\ \text{`}\mathit{eqD}\text{`}\ y\\
&\quad \mathit{eqD}\ \_\qquad\ \ \_\qquad\quad\ = \mathit{return}\ \mathit{False}
\end{aligned}$$

We then obtain equality of terms as follows (we do not consider contexts here for simplicity):

$$\begin{aligned}
&\textbf{instance}\ \mathit{EqD}\ f \Rightarrow \mathit{PEq}\ (\mathit{Trm}\ f\ \mathit{Name})\ \textbf{where}\\
&\quad \mathit{peq}\ (\mathit{In}\ t_1)\quad (\mathit{In}\ t_2)\quad = t_1\ \text{`}\mathit{eqD}\text{`}\ t_2\\
&\quad \mathit{peq}\ (\mathit{Var}\ x_1)\ (\mathit{Var}\ x_2) = \mathit{return}\ (x_1 \equiv x_2)\\
&\quad \mathit{peq}\ \_\qquad\quad \_\qquad\quad\ = \mathit{return}\ \mathit{False}\\
&\textbf{instance}\ (\mathit{Difunctor}\ f, \mathit{EqD}\ f) \Rightarrow \mathit{Eq}\ (\mathit{Term}\ f)\ \textbf{where}\\
&\quad (\equiv)\ (\mathit{Term}\ x)\ (\mathit{Term}\ y) = \mathit{evalFreshM}\ ((x :: \mathit{Trm}\ f\ \mathit{Name})\ \text{`}\mathit{peq}\text{`}\ y)
\end{aligned}$$

Note that we need to explicitly instantiate the parametric type in $x$ to *Name* in the last instance, in order to trigger the instance for *Trm f Name* defined above.

Equality of terms, that is $\alpha$-equivalence, has thus been reduced to providing instances of *EqD* for the difunctors comprising the signature of the term, which for *Lam* can be defined as follows:

```
instance EqD Lam where
  eqD (Lam f) (Lam g) = withName (λx → f x ‘peq‘ g x)
```

That is, $f$ and $g$ are considered equal if they are equal when applied to the same fresh name $x$.

## 5.8 Discussion and Related Work

Implementing languages with binders can be a difficult task. Using explicit variable names, we have to be careful in order to make sure that functions on ASTs are invariant under $\alpha$-renaming. HOAS [88] is one way of tackling this problem, by reusing the binding mechanisms of the implementation language to define those of the object language. The challenge with HOAS, however, is that it is difficult to perform recursive computations over ASTs with binders [25, 66, 116].

*Nominal sets* [91] is another approach for dealing with binders, in which variables are explicit, but recursively defined functions are guaranteed to be invariant with respect to $\alpha$-equivalence of terms. Implementations of this approach, however, require extensions of the metalanguage [100].

Our approach of using PHOAS [19] amounts to the same restriction on embedded functions as Fegeras and Sheard [25], and Washburn and Weirich [116]. However, unlike Washburn and Weirich's Haskell implementation, our approach does not rely on making the type of terms abstract. Not only is it interesting to see that we can do without type abstraction, in fact we sometimes need to inspect terms in order to write functions that produce terms, such as our constant folding algorithm. With Washburn and Weirich's encoding this is not possible.

Ahn and Sheard [3] recently showed how to generalise the recursion schemes of Washburn and Weirich to Mendler-style recursion schemes, using the same representation for terms as Washburn and Weirich. Hence their approach also suffers from the inability to inspect terms. Although we could easily adopt Mendler-style recursion schemes in our setting, their generality does not make a difference in a non-strict language such as Haskell.

The *finally tagless* approach of Carette et al. [18] has been proposed as an alternative solution to the expression problem [114]. While the approach is very simple and elegant, and also supports higher-order encodings, the approach falls short when we want to define recursive, modular computations that construct modular terms too. Atkey et al. [8], for instance, use the finally tagless approach to build a modular interpreter. However, the interpreter cannot be made modular in the return type, that is the language defining values. Hence, when Atkey et al. extend their expression language they need to also change the data type that represents values, which means that the approach is not fully modular.

Besides what is documented in this paper, we have also lifted (generalised) parametric compositional data types to other (co)recursion schemes, such as anamorphisms. Moreover, term homomorphisms can be straightforwardly extended with a state space: depending on how the state is propagated, this yields bottom-up respectively top-down tree transducers [20].

## Acknowledgement

The authors would like to thank Andrzej Filinski for his insightful comments on an earlier version of this paper.

```
import Data.Comp.Param
import Data.Comp.Param.Show ()
import Data.Comp.Param.Equality ()
import Data.Comp.Param.Derive
import Control.Monad.Error (MonadError, throwError)

data Lam a b  = Lam (a → b)
data App a b  = App b b
data Lit a b  = Lit Int
data Plus a b = Plus b b
data Let a b  = Let b (a → b)
data Err a b  = Err

$(derive [smartConstructors, makeDifunctor, makeShowD, makeEqD]
         [''Lam, ''App, ''Lit, ''Plus, ''Let, ''Err])

e :: Term (Lam :+: App :+: Lit :+: Plus :+: Let :+: Err)
e = Term (iLet (iLit 2) (λx → (iLam (λy → y `iPlus` x) `iApp` iLit 3)))

-- * Desugaring
class Desug f g where desugHom :: Hom f g

$(derive [liftSum] [''Desug]) -- lift Desug to coproducts

desug :: (Difunctor f, Difunctor g, Desug f g) ⇒ Term f → Term g
desug (Term t) = Term (appHom desugHom t)

instance (Difunctor f, Difunctor g, f :<: g) ⇒ Desug f g where
  desugHom = In . fmap Hole . inj -- default instance for core signatures

instance (App :<: f, Lam :<: f) ⇒ Desug Let f where
  desugHom (Let e1 e2) = inject (Lam (Hole . e2)) `iApp` Hole e1

-- * Constant folding
class Constf f g where constfAlg :: forall a. Alg f (Trm g a)

$(derive [liftSum] [''Constf]) -- lift Constf to coproducts

constf :: (Difunctor f, Constf f g) ⇒ Term f → Term g
constf t = Term (cata constfAlg t)

instance (Difunctor f, f :<: g) ⇒ Constf f g where
  constfAlg = inject . dimap Var id -- default instance

instance (Plus :<: f, Lit :<: f) ⇒ Constf Plus f where
  constfAlg (Plus e1 e2) = case (project e1, project e2) of
                             (Just (Lit n),Just (Lit m)) → iLit (n + m)
                             _                           → e1 `iPlus` e2

-- * Call-by-value evaluation
data Sem m = Fun (Sem m → m (Sem m)) | Int Int

class Monad m ⇒ Eval m f where evalAlg :: Alg f (m (Sem m))

$(derive [liftSum] [''Eval]) -- lift Eval to coproducts

eval :: (Difunctor f, Eval m f) ⇒ Term f → m (Sem m)
eval = cata evalAlg

instance Monad m ⇒ Eval m Lam where
  evalAlg (Lam f) = return (Fun (f . return))

instance MonadError String m ⇒ Eval m App where
  evalAlg (App mx my) = do x ← mx
                           case x of Fun f → my >>= f
                                     _     → throwError "stuck"

instance Monad m ⇒ Eval m Lit where
  evalAlg (Lit n) = return (Int n)

instance MonadError String m ⇒ Eval m Plus where
  evalAlg (Plus mx my) = do x ← mx; y ← my
                            case (x,y) of (Int n,Int m) → return (Int (n + m))
                                          _             → throwError "stuck"

instance MonadError String m ⇒ Eval m Err where
  evalAlg Err = throwError "error"
```

Figure 5.1: Complete example using the parametric compositional data types library.

# Chapter 6

# Domain-Specific Languages for Enterprise Systems[⋆]

**Abstract**

The process-oriented event-driven transaction systems (POETS) architecture introduced by Henglein et al. is a novel software architecture for enterprise resource planning (ERP) systems. POETS employs a pragmatic separation between (i) transactional data, that is what has happened; (ii) reports, that is what can be derived from the transactional data; and (iii) contracts, that is which transactions are expected in the future. Moreover, POETS applies domain-specific languages (DSLs) for specifying reports and contracts, in order to enable succinct declarative specifications as well as rapid adaptability and customisation. In this report we document an implementation of a generalised and extended variant of the POETS architecture. The generalisation is manifested in a detachment from the ERP domain, which is rather an instantiation of the system than a built-in assumption. The extensions amount to a customisable data model based on nominal subtyping; support for run-time changes to the data model, reports and contracts, while retaining full auditability; and support for referable data that may evolve over time, also while retaining full auditability as well as referential integrity. Besides the revised architecture, we present the DSLs used to specify data definitions, reports, and contracts respectively, and we provide the complete specification for a use case scenario, which demonstrates the conciseness and validity of our approach. Lastly, we describe technical aspects of our implementation, with focus on the techniques used to implement the tightly coupled DSLs.

## 6.1 Introduction

Enterprise resource planning (ERP) systems are comprehensive software systems used to manage daily activities in enterprises. Such activities include—but are not limited to—financial management (accounting), production planning, supply chain management and customer relationship management. ERP systems emerged as a remedy to heterogeneous systems, in which data and functionality are spread out—and duplicated—amongst dedicated subsystems. Instead, an ERP system it built around a central database, which stores all information in one place.

[⋆] *Joint work with Jesper Andersen and Patrick Bahr [51].*

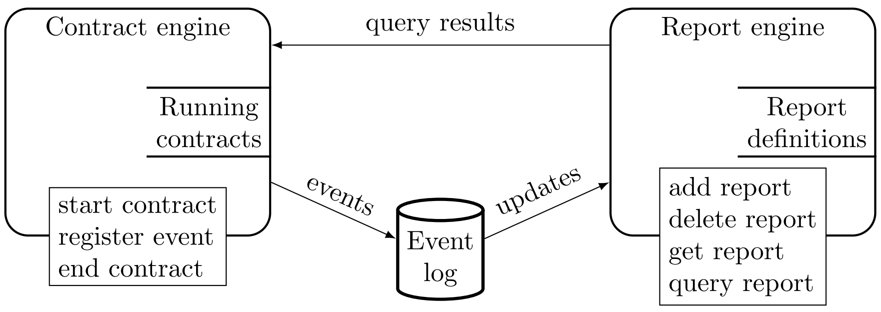


Figure 6.1: Bird's-eye view of the POETS architecture (diagram copied from [41]).

Traditional ERP systems such as Microsoft Dynamics NAV[1], Microsoft Dynamics AX[2], and SAP[3] are three-tier architectures with a client, an application server, and a centralised relational database system. The central database stores information in tables, and the application server provides the business logic, typically coded in a general purpose, imperative programming language. A shortcoming to this approach is that the state of the system is decoupled from the business logic, which means that business processes—that is, the daily activities—are not represented explicitly in the system. Rather, business processes are encoded implicitly as transition systems, where the state is maintained by tables in the database, and transitions are encoded in the application server, possibly spread out across several different logical modules.

The process-oriented event-driven transaction systems (POETS) architecture introduced by Henglein et al. [41] is a qualitatively different approach to ERP systems. Rather than storing both transactional data and implicit process state in a database, POETS employs a pragmatic separation between transactional data, which is persisted in an *event log*, and *contracts*, which are explicit representations of business processes, stored in a separate module. Moreover, rather than using general purpose programming languages to specify business processes, POETS utilises a declarative domain-specific language (DSL) [6]. The use of a DSL not only enables explicit formalisation of business processes, it also minimises the gap between requirements and a running system. In fact, Henglein et al. take it as a goal of POETS that "[...] the formalized requirements *are* the system" [41, page 382].

The bird's-eye view of the POETS architecture is presented in Figure 6.1. At the heart of the system is the event log, which is an append-only list of transactions. Transactions represent "things that take place" such as a payment by a customer, a delivery of goods by a shipping agency, or a movement of items in an inventory. The append-only restriction serves two purposes. First, it is a legal requirement in ERP systems that transactions, which are relevant for auditing, are retained. Second, the report engine utilises monotonicity of the event log for optimisation, as shown by Nissen and Larsen [77].

Whereas the event log stores historical data, contracts play the role of describing which events are expected in the future. For instance, a yearly payment of value-

[1]http://www.microsoft.com/en-us/dynamics/products/nav-overview.aspx.
[2]http://www.microsoft.com/en-us/dynamics/products/ax-overview.aspx.
[3]http://www.sap.com.

added tax (VAT) to the tax authorities is an example of a (recurring) business process. The amount to be paid to the tax authorities depends, of course, on the financial transactions that have taken place. Therefore, information has to be derived from previous transactions in the event log, which is realised as a *report*. A report provides structured data derived from the transactions in the event log. Like contracts, reports are written in a declarative domain-specific language—not only in order to minimise the semantic gap between requirements and the running system, but also in order to perform automatic optimisation.

Besides the radically different software architecture, POETS distinguishes itself from existing ERP systems by abandoning the double-entry bookkeeping (DEB) accounting principle [118] in favour of the resources, events, and agents (REA) accounting model of McCarthy [64].

In double-entry bookkeeping, each transaction is recorded as two postings in a *ledger*—a *debit* and a *credit*. When, for instance, a customer pays an amount $x$ to a company, then a debit of $x$ is posted in a cash account, and a credit of $x$ is posted in a sales account, which reflects the flow of cash from the customer to the company. The central invariant of DEB is that the total credit equals the total debit—if not, resources have either vanished or spontaneously appeared. DEB fits naturally in the relational database oriented architectures, since each ledger is similar in structure to a table. Moreover, DEB is the de facto standard accounting method, and therefore used by current ERP systems.

In REA, transactions are not registered in accounts, but rather as the events that take place. An event in REA is of the form $(a_1, a_2, r)$ meaning that agent $a_1$ transfers resource $r$ to agent $a_2$. Hence, when a customer pays an amount $x$ to a company, then it is represented by a single event $(\text{customer}, \text{company}, x)$. Since events are atomic, REA does not have the same redundancy[4] as DEB, and inconsistency is not a possibility: resources always have an origin and a destination. The POETS architecture not only fits with the REA ontology, it is based on it. Events are stored as first-class objects in the event log, and contracts describe the expected future flow of resources.[5] Reports enable computation of derived information that is inherent in DEB, and which may be a legal requirement for auditing. For instance, a sales account—which summarises (pending) sales payments—can be reconstructed from information about initiated sales and payments made by customers. Such a computation will yield the same *derived* information as in DEB, and the principles of DEB consistency will be fulfilled simply by construction.

### 6.1.1 Outline and Contributions

The motivation for our work is to assess the POETS architecture in terms of a prototype implementation. During the implementation process we have added features to the architecture that were painfully missing. Moreover, in the process we found that the architecture need not be tailored to the REA ontology—indeed to ERP systems—but the applicability of our generalised architecture to other domains remains future research. Our contributions are as follows:

[4]In traditional DEB, redundancy is a feature to check for consistency. However, in a computer system such redundancy is superfluous.

[5]Structured contracts are not part of the original REA ontology but instead due to Andersen et al. [6].

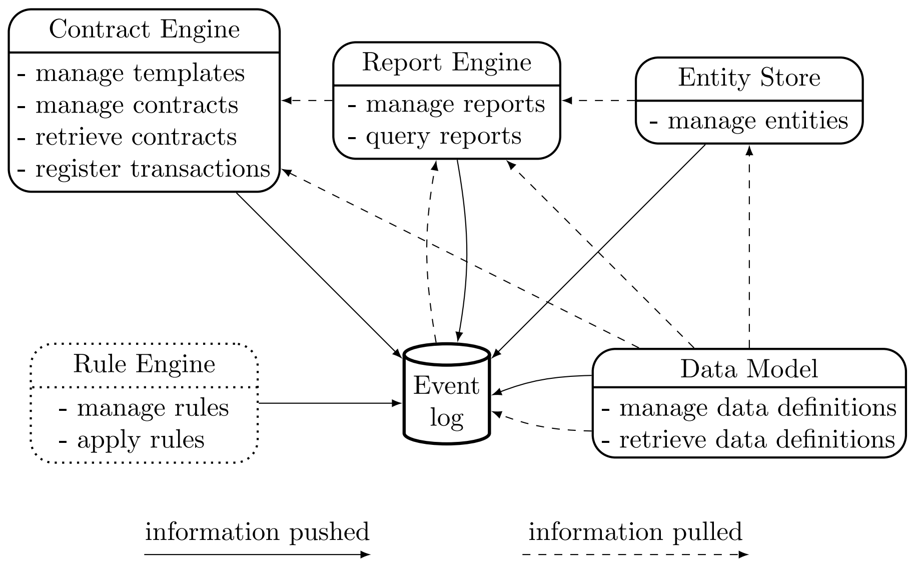


Figure 6.2: Bird's-eye view of the generalised and extended POETS architecture.

- We present a generalised and extended POETS architecture (Section 6.2) that has been fully implemented.
- We present DSLs for data modelling (Section 6.2.1), report specification (Section 6.2.4), and contract specification (Section 6.2.5).
- We demonstrate how to implement a small use case, from scratch, in our implemented system (Section 6.3). We provide the complete specification of the system, which demonstrates both the conciseness and domain-orientation[6] of our approach. We conclude that the extended architecture is indeed well-suited for implementing ERP systems—although the DSLs and the data model may require additions for larger systems. Most notably, the amount of code needed to implement the system is but a fraction of what would be have to be implemented in a standard ERP system.
- We describe how we have utilised state-of-the art software development tools in our implementation, especially how the tightly coupled DSLs are implemented (Section 6.4).

## 6.2 Revised POETS Architecture

Our generalised and extended architecture is presented in Figure 6.2. Compared to the original architecture in Figure 6.1, the revised architecture sees the addition of three new components: a *data model*, an *entity store*, and a *rule engine*. The rule engine is currently not implemented, and we will therefore not return to this module until Section 6.5.1.

As in the original POETS architecture, the event log is at the heart of the system. However, in the revised architecture the event log plays an even greater role, as it

[6]Compare the motto: "[...] the formalized requirements *are* the system" [41, page 382].

| Data Model | | |
|---|---|---|
| **Function** | **Input** | **Output** |
| *addDataDefs* | ontology specification | |
| *getRecordDef* | record name | type definition |
| *getSubTypes* | record name | list of record names |

Figure 6.3: Data model interface.

is the *only* persistent state of the system. This means that the states of all other modules are also persisted in the event log, hence the flow of information from all other modules to the event log in Figure 6.2. For example, whenever a contract is started or a new report is added to the system, then an event reflecting this operation is persisted in the event log. This, in turn, means that the state of each module can—in principle—be derived from the event log. However, for performance reasons each module—including the event log—maintains its own state in memory.

The addition of a data model constitutes the generalisation of the new architecture over the old architecture. In the data model, data definitions can be added to the system—at run-time—such as data defining customers, resources, or payments. Therefore, the system is not a priori tailored to ERP systems or the REA ontology, but it can be instantiated to that, as we shall see in Section 6.3.

The entity store is added to the architecture in order to support *entities*—unique "objects" with associated data that may evolve over time. For instance, a concrete customer can suitably be modelled as an entity: although information attributed to that customer—such as address, or even name—are likely to change over time, it is still the same customer. Moreover, we do not want a copy of the customer data in for instance a sale, but rather a reference to that customer. Hence by modelling customers as entities, we are able to derive, for instance, all transactions in which that customer has participated—even if the customer attributes have changed over time.

We describe each module of the revised architecture in the following subsections. Since we will focus on the revised architecture in the remainder of the text, we will refer to said architecture simply as POETS.

### 6.2.1 Data Model

The data model is a core component of the extended architecture, and the interface it provides is summarised in Figure 6.3. The data model defines the *types* of data that are used throughout the system, and it includes predefined types such as events. Custom types such as invoices can be added to the data model at run-time via *addDataDefs*—for simplicity, we currently only allow addition of types, not updates and deletions. Types define the structure of the data in a running POETS instance manifested as *values*. A value—such as a concrete invoice—is an instance of the data specified by a type. Values are not only communicated between the system and its environment but they are also stored in the event log, which is simply a list of values of a certain type.

#### 6.2.1.1 Types

Structural data such as payments and invoices are represented as *records*, that is typed finite mappings from field labels to values. Record types define the structure of such records by listing the constituent field labels and their associated types. In order to form a hierarchical ontology of record types, we use a nominal subtyping system [89]. That is, each record type has a unique name, and one type is a subtype of another if and only if stated so explicitly or by transitivity. For instance, a customer can be defined as a subtype of a person, which means that a customer contains all the data of a person, similar to inheritance in object oriented programming.

The choice of nominal types over structural types [89] is justified by the domain: the nominal type associated with a record may have a semantic impact. For instance, the type of customers and premium customers may be structurally equal, but a value of one type is considered different from the other, and clients of the system may for example choose to render them differently. Moreover, the purpose of the rule engine, which we return to in Section 6.5.1, is to define rules for values of a particular semantic domain, such as invoices. Hence it is wrong to apply these rules to data that happens to have the same structure as invoices. Although we use nominal types to classify data, the DSLs support full record polymorphism [79] in order to minimise code duplication. That is, it is possible for instance to use the same piece of code with customers and premium customers, even if they are not related in the subtyping hierarchy.

The grammar for types is as follows:

$$
\begin{array}{rll}
T ::= & \mathbf{Bool} \mid \mathbf{Int} \mid \mathbf{Real} \mid \mathbf{String} \mid \mathbf{Timestamp} \mid \mathbf{Duration} & \text{(type constants)} \\
\mid & \mathit{RecordName} & \text{(record type)} \\
\mid & [T] & \text{(list type)} \\
\mid & \langle \mathit{RecordName} \rangle & \text{(entity type)}
\end{array}
$$

Type constants are standard types Booleans, integers, reals, and strings, and less standard types timestamps (absolute time) and durations (relative time). Record types are named types, and the record typing environment—which we will describe shortly—defines the structure of records. For record types we assume a set $\mathit{RecordName} = \{\mathsf{Customer}, \mathsf{Address}, \mathsf{Invoice}, \dots\}$ of record names ranged over by $r$. Concrete record types are typeset in sans-serif, and they always begin with a capital letter. Likewise, we assume a set *FieldName* of all field names ranged over by $f$. Concrete field names are typeset in sans-serif beginning with a lower-case letter.

List types $[\tau]$ represent lists of values, where each element has type $\tau$, and it is the only collection type currently supported. Entity types $\langle r \rangle$ represent entity values that have associated data of type $r$. For instance, if the record type Customer describes the data of a customer, then a value of type $\langle \mathsf{Customer} \rangle$ is a (unique) customer entity, whose associated Customer data may evolve over time. The type system ensures that a value of an entity type in the system will have associated data of the given type, similar to referential integrity in database systems [13]. We will return to how entities are created and modified in Section 6.2.3.

A *record typing environment* provides the record types that are available, their subtype relation, and the fields they define.

**Definition 6.2.1.** A record typing environment is a tuple $(R, A, F, \rho, \leq)$ consisting of finite sets $R \subseteq \mathit{RecordName}$ and $F \subseteq \mathit{FieldName}$, a set $A \subseteq R$, a mapping

$\rho : R \to \mathcal{P}_{\text{fin}}(F \times T)$, and a relation $\leq \subseteq R \times R$, where $\mathcal{P}_{\text{fin}}(X)$ denotes the set of all finite subsets of a set $X$.

Intuitively, $R$ is the set of defined record types, $\rho$ gives for each defined record type its fields and their types, $\leq$ gives the subtyping relation between record types, and record types in $A$ are considered to be abstract. Abstract record types are not supposed to be instantiated, they are only used to structure the record type hierarchy. The functions *getRecordDef* and *getSubTypes* from Figure 6.3 provide the means to retrieve the record typing environment from a running system.

Record types can depend on other record types by having them as part of the type of a constituent field:

**Definition 6.2.2.** The *immediate dependency relation* of a record typing environment $\mathcal{R} = (R, A, F, \rho, \leq)$, denoted $\to_{\mathcal{R}}$, is the binary relation on $R$ such that $r_1 \to_{\mathcal{R}} r_2$ iff there is some $(f, \tau) \in \rho(r_1)$ such that a record name $r$ occurs in $\tau$ with $r_2 \leq r$. The *dependency relation* $\to_{\mathcal{R}}^{+}$ of $\mathcal{R}$ is the transitive closure of $\to_{\mathcal{R}}$.

We do not permit all record typing environments. Firstly, we do not allow the subtyping to be cyclic, that is a record type $r$ cannot have a proper subtype which has $r$ as a subtype. Secondly, the definition of field types must be unique and must follow the subtyping, that is a subtype must define at least the fields of its supertypes. Lastly, we do not allow recursive record type definitions, that is a cycle in the dependency relation. The two first restrictions are sanity checks, but the last restriction makes a qualitative difference: the restriction is imposed for simplicity, and moreover we have not encountered practical situations where recursive types were needed.

**Definition 6.2.3.** A record typing environment $\mathcal{R} = (R, A, F, \rho, \leq)$ is *well-formed*, whenever the following is satisfied:

- $\leq$ is a partial order, (acyclic inheritance)
- each $\rho(r)$ is the graph of a partial function $F \rightharpoonup T$, (unique typing)
- $r_1 \leq r_2$ implies $\rho(r_1) \supseteq \rho(r_2)$, and (consistent typing)
- $\to_{\mathcal{R}}^{+}$ is irreflexive, that is $r_1 \to_{\mathcal{R}}^{+} r_2$ implies $r_1 \neq r_2$. (non-recursive)

Well-formedness of a record typing environment combines both conditions for making it easy to reason about them—for instance, transitivity of $\leq$ and inclusion of fields of supertypes—and hard restrictions such as non-recursiveness and unique typing. If a record typing environment fails to be well-formed due to the former only, it can be uniquely closed to a well-formed one:

**Definition 6.2.4.** The *closure* of a record typing environment $\mathcal{R} = (R, A, F, \rho, \leq)$ is the record typing environment $\mathrm{Cl}(\mathcal{R}) = (R, A, F, \rho', \leq')$ such that $\leq'$ is the transitive, reflexive closure of $\leq$ and $\rho'$ is the consistent closure of $\rho$ with respect to $\leq'$, that is $\rho'(r) = \bigcup_{r \leq' r'} \rho(r')$.

The definition of closure allows us to easily build a well-formed record typing environment from an incomplete specification.

**Example 6.2.5.** As an example, we may define a record typing environment $\mathcal{R} = (R, A, F, \rho, \leq)$ for persons and customers as follows:

$$\begin{aligned} R &= \{\mathsf{Person}, \mathsf{Customer}, \mathsf{Address}\} & \rho(\mathsf{Person}) &= \{(\mathsf{name}, \mathbf{String})\} \\ A &= \{\mathsf{Person}\} & \rho(\mathsf{Customer}) &= \{(\mathsf{address}, \mathsf{Address})\} \\ F &= \{\mathsf{name}, \mathsf{address}, \mathsf{road}, \mathsf{no}\} & \rho(\mathsf{Address}) &= \{(\mathsf{road}, \mathbf{String}), (\mathsf{no}, \mathbf{Int})\}, \end{aligned}$$

with $\mathsf{Customer} \leq \mathsf{Person}$. The only properties that prevent $\mathcal{R}$ from being well-formed are the missing field typing $(\mathsf{name}, \mathbf{String})$ that $\mathsf{Customer}$ should inherit from $\mathsf{Person}$ and the missing reflexivity of $\leq$. Hence, the closure $\mathrm{Cl}(\mathcal{R})$ of $\mathcal{R}$ is indeed a well-formed record typing environment.

In order to combine record typing environments we define the union $\mathcal{R}_1 \cup \mathcal{R}_2$ of two record typing environments $\mathcal{R}_i = (R_i, A_i, F_i, \rho_i, \leq_i)$ as the pointwise union:

$$\mathcal{R}_1 \cup \mathcal{R}_2 = (R_1 \cup R_2, A_1 \cup A_2, F_1 \cup F_2, \rho_1 \cup \rho_2, \leq_1 \cup \leq_2),$$

where $(\rho_1 \cup \rho_2)(r) = \rho_1(r) \cup \rho_2(r)$ for all $r \in R_1 \cup R_2$. Note that the union of two well-formed record typing environments need not be well-formed—either due to incompleteness, which can be resolved by forming the closure of the union, or due to inconsistencies respectively cyclic dependencies, which cannot be resolved.

### 6.2.1.2 Values

The set of values *Value* supplementing the types from the previous section is defined inductively as the following disjoint union:

$$\mathit{Value} = \mathit{Bool} \uplus \mathit{Int} \uplus \mathit{Real} \uplus \mathit{String} \uplus \mathit{Timestamp} \uplus \mathit{Duration} \uplus \mathit{Record} \uplus \mathit{List} \uplus \mathit{Ent},$$

with:

$$\begin{aligned} \mathit{Bool} &= \{\mathit{true}, \mathit{false}\} & \mathit{String} &= \mathit{Char}^* & \mathit{Record} &= \mathit{RecordName} \times \mathit{Fields} \\ \mathit{Int} &= \mathbb{Z} & \mathit{Timestamp} &= \mathbb{N} & \mathit{Fields} &= \mathit{FieldName} \rightharpoonup_{\mathrm{fin}} \mathit{Value} \\ \mathit{Real} &= \mathbb{R} & \mathit{Duration} &= \mathbb{Z} & \mathit{List} &= \mathit{Value}^*, \end{aligned}$$

where $X^*$ denotes the set of all finite sequences over a set $X$; *Char* is a set of characters; *Ent* is an abstract, potentially infinite set of entity values; and $A \rightharpoonup_{\mathrm{fin}} B$ denotes the set of finite partial mappings from a set $A$ to a set $B$.

Timestamps are modelled using UNIX time[7] and durations are measured in seconds. A record $(r, m) \in \mathit{Record}$ consists of a record name $r \in \mathit{RecordName}$ together with a finite set of named values $m \in \mathit{Fields}$. Entity values $e \in \mathit{Ent}$ are abstract values that only permit equality testing and dereferencing—the latter takes place only in the report engine (Section 6.2.4), and the type system ensures that dereferencing cannot get stuck, as we shall see in the following subsection.

**Example 6.2.6.** As an example, a customer record value $c \in \mathit{Record}$ may be as follows:

$$\begin{aligned} c &= (\mathsf{Customer}, m) & m'(\mathsf{road}) &= \texttt{Universitetsparken} \\ m(\mathsf{name}) &= \texttt{John Doe} & m'(\mathsf{no}) &= 1, \\ m(\mathsf{address}) &= (\mathsf{Address}, m') \end{aligned}$$

where $m, m' \in \mathit{Fields}$.

[7]http://en.wikipedia.org/wiki/Unix_time.

$$\boxed{\mathcal{R}, \mathcal{E} \vdash v : \tau} \qquad \frac{b \in \mathit{Bool}}{\mathcal{R}, \mathcal{E} \vdash b : \mathbf{Bool}} \qquad \frac{n \in \mathit{Int}}{\mathcal{R}, \mathcal{E} \vdash n : \mathbf{Int}} \qquad \frac{r \in \mathit{Real}}{\mathcal{R}, \mathcal{E} \vdash r : \mathbf{Real}}$$

$$\frac{s \in \mathit{String}}{\mathcal{R}, \mathcal{E} \vdash s : \mathbf{String}} \qquad \frac{t \in \mathit{Timestamp}}{\mathcal{R}, \mathcal{E} \vdash t : \mathbf{Timestamp}} \qquad \frac{d \in \mathit{Duration}}{\mathcal{R}, \mathcal{E} \vdash d : \mathbf{Duration}}$$

$$\frac{(r, m) \in \mathit{Record} \qquad \begin{array}{c}\mathcal{R} = (R, A, F, \rho, \leq) \\ r \in R \setminus A\end{array} \qquad \begin{array}{c}\mathrm{dom}(\rho(r)) = \mathrm{dom}(m) \\ \forall f \in \mathrm{dom}(m) \colon \mathcal{R}, \mathcal{E} \vdash m(f) : \rho(r)(f)\end{array}}{\mathcal{R}, \mathcal{E} \vdash (r, m) : r}$$

$$\frac{(v_1, \ldots, v_n) \in \mathit{List} \qquad \forall i \in \{1, \ldots, n\}.\mathcal{R}, \mathcal{E} \vdash v_i : \tau}{\mathcal{R}, \mathcal{E} \vdash (v_1, \ldots, v_n) : [\tau]} \qquad \frac{e \in \mathit{Ent} \qquad \mathcal{E}(e) = r}{\mathcal{R}, \mathcal{E} \vdash e : \langle r \rangle}$$

$$\frac{\mathcal{R}, \mathcal{E} \vdash v : \tau' \qquad \mathcal{R} \vdash \tau' <: \tau}{\mathcal{R}, \mathcal{E} \vdash v : \tau}$$

$$\boxed{\mathcal{R} \vdash \tau_1 <: \tau_2} \qquad \frac{}{\mathcal{R} \vdash \tau <: \tau} \qquad \frac{\mathcal{R} \vdash \tau_1 <: \tau_2 \qquad \mathcal{R} \vdash \tau_2 <: \tau_3}{\mathcal{R} \vdash \tau_1 <: \tau_3}$$

$$\frac{}{\mathcal{R} \vdash \mathbf{Int} <: \mathbf{Real}} \qquad \frac{r_1 \leq r_2}{(R, A, F, \rho, \leq) \vdash r_1 <: r_2}$$

$$\frac{\mathcal{R} \vdash \tau_1 <: \tau_2}{\mathcal{R} \vdash [\tau_1] <: [\tau_2]} \qquad \frac{r_1 \leq r_2}{(R, A, F, \rho, \leq) \vdash \langle r_1 \rangle <: \langle r_2 \rangle}$$

Figure 6.4: Type checking of values $\mathcal{R}, \mathcal{E} \vdash v : \tau$ and subtyping $\mathcal{R} \vdash \tau_1 <: \tau_2$.

#### 6.2.1.3 Type Checking

All values are type checked before they enter the system, both in order to check that record values conform with the record typing environment, but also to check that entity values have valid associated data. In particular, events—which are values—are type checked before they are persisted in the event log. In order to type check entities, we assume an *entity typing environment* $\mathcal{E} : \mathit{Ent} \rightharpoonup_{\text{fin}} \mathit{RecordName}$, that is a finite partial mapping from entities to record names. Intuitively, an entity typing environment maps an entity to the record type that it has been declared to have upon creation.

The typing judgement has the form $\mathcal{R}, \mathcal{E} \vdash v : \tau$, where $\mathcal{R}$ is a well-formed record typing environment, $\mathcal{E}$ is an entity typing environment, $v \in \mathit{Value}$ is a value, and $\tau \in T$ is a type. The typing judgement uses the auxiliary subtyping judgement $\mathcal{R} \vdash \tau_1 <: \tau_2$, which is a generalisation of the subtyping relation from Section 6.2.1.1 to arbitrary types.

The typing rules are given in Figure 6.4. The typing rules for base types and lists are standard. In order to type check a record, we need to verify that the record contains all and only those fields that the record typing environment prescribes, and that the values have the right type. The typing rule for entities uses the entity typing environment to check that each entity has associated data, and that the data has the required type. The last typing rule enables values to be coerced to a supertype in accordance with the subtyping judgement, which is also given in Figure 6.4. The rules for the subtyping relation extend the relation from Section 6.2.1.1 to include subtyping of base types, and contextual rules for lists and entities. We remark that the type system in Figure 6.4 is declarative: in our implementation, an equivalent algorithmic type system is used.

**Example 6.2.7.** Reconsider the record typing environment $\mathcal{R}$ and its closure $\mathrm{Cl}(\mathcal{R})$ from Example 6.2.5, and the record value $c$ from Example 6.2.6. Using the typing rules in Figure 6.4, we can derive the typing judgement $\mathrm{Cl}(\mathcal{R}), \mathcal{E} \vdash c : \mathsf{Customer}$ for any entity typing environment $\mathcal{E}$. Moreover, since Customer is a subtype of Person we also have that $\mathrm{Cl}(\mathcal{R}), \mathcal{E} \vdash c : \mathsf{Person}$.

In the following, we want to detail how the typing rules guarantee the integrity of entities, which involves reasoning about the evolution of the system over time. To this end, we use $\mathcal{R}_t = (R_t, A_t, F_t, \rho_t, \leq_t)$ and $\mathcal{E}_t$ to indicate the record typing environment and the entity typing environment respectively, at a point in time $t \in$ *Timestamp*. In order to reason about the data associated with an entity, we assume for each point in time $t \in \mathit{Timestamp}$ an *entity environment* $\epsilon_t : \mathit{Ent} \rightharpoonup_{\mathrm{fin}} \mathit{Record}$ that maps an entity to its associated data. Entity (typing) environments form the basis of the entity store, which we will describe in detail in Section 6.2.3.

Given $T \subseteq \mathit{Timestamp}$ and sequences $(\mathcal{R}_t)_{t\in T}$, $(\mathcal{E}_t)_{t\in T}$, and $(\epsilon_t)_{t\in T}$ of record typing environments, entity typing environments, and entity environments respectively, which represent the evolution of the system over time, we require the following invariants to hold for all $t, t' \in \mathit{Timestamp}$, $r, r' \in \mathit{RecordName}$, $e \in \mathit{Ent}$, and $v \in \mathit{Record}$:

$$\begin{aligned}
&\text{if } \mathcal{E}_t(e) = r \text{ and } \mathcal{E}_{t'}(e) = r' \text{ then } r = r', && \text{(stable type)}\\
&\text{if } \mathcal{E}_t(e) \text{ is defined then so is } \epsilon_t(e), \text{ and} && \text{(well-definedness)}\\
&\text{if } \epsilon_t(e) = v \text{ then } \mathcal{E}_t(e) = r \text{ and } \mathcal{R}_{t'}, \mathcal{E}_{t'} \vdash v : r \text{ for some } t' \leq t. && \text{(well-typing)}
\end{aligned}$$

We refer to the three invariants above collectively as the *entity integrity invariants*. The *stable type* invariant states that each entity can have at most one declared type throughout its lifetime. The *well-definedness* invariant guarantees that every entity that is given a type also has an associated record value. Finally, the *well-typing* invariant guarantees that the record value associated with an entity *was* well-typed at some earlier point in time $t'$.

The well-typing invariant is, of course, not strong enough. What we need is that the value $v$ associated with an entity $e$ *remains* well-typed throughout the lifetime of the system. This is, however, dependant on the record typing environment and the entity typing environment, which both may change over time. Therefore, we need to impose restrictions on the possible evolution of the record typing environment, and we need to take into account that entities used in the value $v$ may have been deleted. We return to these issues in Section 6.2.2 and Section 6.2.3, and in the latter we will see that the entity integrity invariants are indeed satisfied by the system.

#### 6.2.1.4 Ontology Language

Section 6.2.1.1 provides the semantic account of record types, and in order to specify record types, we use a variant of Attempto Controlled English [29] due to Jønsson

Thomsen [55], referred to as the *ontology language*. The approach is to define data types in near-English text, in order to minimise the gap between requirements and specification. As an example, the record typing environment from Example 6.2.5 is specified in the ontology language as follows:

*Person is abstract.*
*Person has a String called name.*

*Address has a String called road.*
*Address has an Int called no.*

*Customer is a Person.*
*Customer has an Address.*

An ontology definition consists of a sequence of sentences as defined by the grammar below (where $[\cdot]$ denotes optionality):

$$\begin{array}{lcll}
\textit{Ontology} & ::= & \textit{Sentence}^* & \text{(ontology)} \\
\textit{Sentence} & ::= & \textit{RecordName}\ \textbf{is}\ [\textbf{a} \mid \textbf{an}]\ \textit{RecordName}. & \text{(supertype declaration)} \\
& \mid & \textit{RecordName}\ \textbf{is abstract}. & \text{(abstract declaration)} \\
& \mid & \textit{RecordName}\ \textbf{has}\ [\textbf{a} \mid \textbf{an}]\ \textit{Type} & \text{(field declaration)} \\
& & \qquad [\textbf{called}\ \textit{FieldName}]. & \\
\textit{Type} & ::= & \textbf{Bool} \mid \textbf{Int} \mid \textbf{Real} & \text{(type constants)} \\
& \mid & \textbf{String} \mid \textbf{Timestamp} \mid \textbf{Duration} & \\
& \mid & \textit{RecordName} & \text{(record type)} \\
& \mid & \textbf{list of}\ \textit{Type} & \text{(list type)} \\
& \mid & \textit{RecordName}\ \textbf{entity} & \text{(entity type)}
\end{array}$$

The language of types *Type* reflects the definition of types in $T$ and there is an obvious bijection $[\![\cdot]\!] : \textit{Type} \to T$ with $[\![\textbf{list of}\ t]\!] = [[\![t]\!]]$, $[\![r\ \textbf{entity}]\!] = \langle r \rangle$, and otherwise $[\![t]\!] = t$.

The semantics of the ontology language is given by a straightforward mapping into the domain of record typing environments. Each sentence is translated into a record typing environment. The semantics of a sequence of sentences is simply the closure of the union of each sentence's record typing environment:

$$\begin{aligned}
[\![s_1 \cdots s_n]\!] &= \mathrm{Cl}\left([\![s_1]\!] \cup [\![s_2]\!] \cup \cdots \cup [\![s_n]\!]\right), \\
[\![r_1\ \textbf{is}\ [\textbf{a} \mid \textbf{an}]\ r_2.]\!] &= (\{r_1, r_2\}, \emptyset, \emptyset, \{r_1 \mapsto \emptyset, r_2 \mapsto \emptyset\}, \{(r_1, r_2)\}), \\
[\![r\ \textbf{is abstract}.]\!] &= (\{r\}, \{r\}, \emptyset, \{r \mapsto \emptyset\}, \emptyset), \\
[\![r\ \textbf{has}\ [\textbf{a} \mid \textbf{an}]\ t\ \textbf{called}\ f.]\!] &= (\{r\}, \emptyset, \{f\}, \{r \mapsto \{(f, [\![t]\!])\}\}, \emptyset).
\end{aligned}$$

We omit the case where the optional *FieldName* is not supplied in a field declaration. We treat this form as syntactic sugar for $r$ **has** (**a** | **an**) $t$ **called** $f$. where $f$ is derived from the type $t$. In this case a default name is used based on the type, simply by changing the first letter to a lower-case. Hence, in the example above the field name of a customer's address is address. Note that the record typing environment need not be well-formed (Definition 6.2.3), and a subsequent check for well-formedness has to be performed.

Data definitions added to the system via *addDataDefs* are specified in the ontology language. We require, of course, that the result of adding data definitions must yield a well-defined record typing environment. Moreover, we impose further monotonicity constraints which ensure that existing data in the system remain well-typed.

We return to these constraints when we discuss the event log in Section 6.2.2. Type definitions retrieved via *getRecordDef* provide the semantic structure of a record type, that is its immediate supertypes, its fields, and an indication whether the record type is abstract. *getSubTypes* returns a list of immediate subtypes of a given record type, hence *getRecordDef* and *getSubTypes* provide the means for clients of the system to traverse the type hierarchy—both upwards and downwards.

#### 6.2.1.5 Predefined Ontology

Unlike the original POETS architecture [41], our generalised architecture is not fixed to an enterprise resource planning (ERP) domain. However, we require a set of predefined record types, which are included in Appendix E.1. That is, the record typing environment $\mathcal{R}_0$ denoted by the ontology in Appendix E.1 is the initial record typing environment in all POETS instances.

The predefined ontology defines five root concepts in the data model, that is record types maximal with respect to the subtype relation $\leq$. Each of these five root concepts Data, Event, Transaction, Report, and Contract are abstract and only Event and Contract define record fields. Custom data definitions added via *addDataDefs* are only permitted as subtypes of Data, Transaction, Report, and Contract. In contrast to that, Event has a predefined and fixed hierarchy.

Data types represent elements in the domain of the system such as customers, items, and resources.

Transaction types represent events that are associated with a contract, such as payments, deliveries, and issuing of invoices.

Report types are result types of report functions, that is the data of reports, such as inventory status, income statement, and list of customers. The Report structure does not define *how* reports are computed, only *what kind* of result is computed. We will return to this discussion in Section 6.2.4.

Contract types represent the different kinds of contracts, such as sales, purchases, and manufacturing procedures. Similar to Report, the structure does not define what the contract dictates, only what is required to instantiate the contract. The purpose of Contract is hence dual to the purpose of Report: the former determines an input type, and the latter determines an output type. We will return to contracts in Section 6.2.5.

Event types form a fixed hierarchy and represent events that are logged in the system. Events are conceptually separated into *internal* events and *external* events, which we describe further in the following section.

### 6.2.2 Event Log

The event log is the only persistent state of the system, and it describes the complete state of a running POETS instance. The event log is an append-only list of records of the type Event defined in Appendix E.1. Each event reflects an atomic interaction with the running system. This approach is also applied at the "meta level" of

POETS: in order to allow agile evolution of a running POETS instance, changes to the data model, reports, and contracts are reflected in the event log as well.

The monotonic nature of the event log—data is never overwritten or deleted from the system—means that the state of the system can be reconstructed at any previous point in time. In particular, transactions are never deleted, which is a legal requirement for ERP systems. The only component of the architecture that reads directly from the event log is the report engine (compare Figure 6.2), hence the only way to access data in the log is via a report.

All events are equipped with an internal timestamp (internalTimeStamp), the time at which the event is registered in the system. Therefore, the event log is always monotonically decreasing with respect to internal timestamps, as the newest event is at the head of the list. Conceptually, events are divided into *external* and *internal* events.

External events are events that are associated with a contract, and only the contract engine writes external events to the event log. The event type TransactionEvent models external events, and it consists of three parts: (i) a contract identifier (contractId), (ii) a timestamp (timeStamp), and (iii) a transaction (transaction). The identifier associates the external event with a contract, and the timestamp represents the time at which the external event takes place. Note that the timestamp need not coincide with the internal timestamp. For instance, a payment in a sales contract may be registered in the system the day after it takes place. There is hence no a priori guarantee that external events have decreasing timestamps in the event log—only external events that pertain to the same contract are required to have decreasing timestamps. The last component, transaction, represents the actual action that takes place, such as a payment from one person or company to another. The transaction is a record of type Transaction, for which the system has no presumptions.

Internal events reflect changes in the state of the system at a meta level. This is the case for example when a contract is instantiated or when a new record definition is added. Internal events are represented by the remaining subtypes of the Event record type. Figure 6.5 provides an overview of all non-abstract record types that represent internal events.

A common pattern for internal events is to have three event types to represent creation, update, and deletion of respective components. For instance, when a report is added to the report engine, a CreateReport event is persisted to the log, and when it is updated or deleted, UpdateReport and DeleteReport events are persisted accordingly. This means that previous versions of the report specification can be retrieved, and more generally that the system can be restarted simply by replaying the events that are persisted in the log on an initially empty system. Another benefit to the approach is that the report engine, for instance, does not need to provide built-in functionality to retrieve, say, the list of all reports added within the last month—such a list can instead be computed as a report itself! We will see how to write such a "meta" report in Section 6.2.4. Similarly, lists of entities, contract templates, and running contracts can be defined as reports.

Since we allow the data model of the system to evolve over time, we must be careful to ensure that the event log, and thus all data in it, remains well-typed at any point in time. Let $(\mathcal{R}_t)_{t\in T}$, $(\mathcal{E}_t)_{t\in T}$, and $(l_t)_{t\in T}$ be sequences of record typing environments, entity typing environments, and event logs respectively. Since an entity

| Event | Description |
|---|---|
| AddDataDefs | A set of data definitions is added to the system. The field defs contains the ontology language specification. |
| CreateEntity | An entity is created. The field data contains the data associated with the entity, the field recordType contains the string representation of the declared type, and the field ent contains the newly created entity value. |
| UpdateEntity | The data associated with an entity is updated. |
| DeleteEntity | An entity is deleted. |
| CreateReport | A report is created. The field code contains the specification of the report, and the fields description and tags are meta data. |
| UpdateReport | A report is updated. |
| DeleteReport | A report is deleted. |
| CreateContractDef | A contract template is created. The field code contains the specification of the contract template, and the fields recordType and description are meta data. |
| UpdateContractDef | A contract template is updated. |
| DeleteContractDef | A contract template is deleted. |
| CreateContract | A contract is instantiated. The field contractId contains the newly created identifier of the contract and the field contract contains the name of the contract template to instantiate, as well as data needed to instantiate the contract template. |
| UpdateContract | A contract is updated. |
| ConcludeContract | A contract is concluded. |

Figure 6.5: Internal events.

might be deleted over time, and thus is removed from the entity typing environment, the event log may not be well-typed with respect to the current entity typing environment. To this end, we type the event log with respect to the *accumulated entity typing environment* $\widehat{\mathcal{E}_t} = \bigcup_{t' \leq t} \mathcal{E}_{t'}$. That is, $\widehat{\mathcal{E}_t}(e) = r$ iff there is some $t' \leq t$ with $\mathcal{E}_{t'}(e) = r$. The stable type invariant guarantees that $\widehat{\mathcal{E}_t}$ is indeed well-defined.

For changes to the record typing environment, we require the following invariants for any points in time $t, t'$ and the event log $l_t$ at time $t$:

$$\text{if } t' \geq t \text{ then } \mathcal{R}_{t'} = \mathcal{R}_t \cup \mathcal{R}_\Delta \text{ for some } \mathcal{R}_\Delta \text{, and} \qquad \text{(monotonicity)}$$

$$\mathcal{R}_t, \widehat{\mathcal{E}_t} \vdash l_t : [\mathsf{Event}]\,. \qquad \text{(log typing)}$$

Note that the *log typing* invariant follows from the *monotonicity* invariant and the type checking $\mathcal{R}_t, \mathcal{E}_t \vdash e : \mathsf{Event}$ for each new incoming event, provided that for each record name $r$ occurring in the event log, no additional record fields are added to $r$, and $r$ is not made an abstract record type. We will refer to the two invariants above collectively as *record typing invariants*. They will become crucial in the following section.

### 6.2.3 Entity Store

The entity store provides very simple functionality, namely creation, deletion and updating of entities, respectively. To this end, the entity store maintains the current

| Entity Store | | |
|---|---|---|
| **Function** | **Input** | **Output** |
| *createEntity* | record name, record | entity |
| *updateEntity* | entity, record | |
| *deleteEntity* | entity | |

Figure 6.6: Entity store interface.

entity typing environment $\mathcal{E}_t$ as well as the history of entity environments $\epsilon_0, \dots, \epsilon_t$. The interface of the entity store is summarised in Figure 6.6.

The creation of a new entity via *createEntity* at time $t+1$ requires a declared type $r$ and an initial record value $v$, and it is checked that $\mathcal{R}_t, \mathcal{E}_t \vdash v : r$. If the value type checks, a *fresh* entity value $e \notin \bigcup_{t' \leq t} \operatorname{dom}(\epsilon_{t'})$ is created, and the entity environment and the entity typing environment are updated accordingly:

$$\epsilon_{t+1}(x) = \begin{cases} v & \text{if } x = e, \\ \epsilon_t(x) & \text{otherwise,} \end{cases} \qquad \mathcal{E}_{t+1}(x) = \begin{cases} r & \text{if } x = e, \\ \mathcal{E}_t(x) & \text{otherwise.} \end{cases}$$

Moreover, a $\mathsf{CreateEntity}$ event is persisted to the event log containing $e$, $r$, and $v$ for the relevant fields.

Similarly, if the data associated with an entity $e$ is updated to the value $v$ at time $t+1$, then it is checked that $\mathcal{R}_t, \mathcal{E}_t \vdash v : \mathcal{E}_t(e)$, and the entity store is updated like above. Note that the entity typing environment is unchanged, that is $\mathcal{E}_{t+1} = \mathcal{E}_t$. A corresponding $\mathsf{UpdateEntity}$ event is persisted to the event log containing $e$ and $v$ for the relevant fields.

Finally, if an entity $e$ is deleted at time $t+1$, then it is removed from both the entity store and the entity typing environment:

$$\epsilon_{t+1}(x) = \epsilon_t(x) \text{ iff } x \in \operatorname{dom}(\epsilon_t) \setminus \{e\}\,,$$
$$\mathcal{E}_{t+1}(x) = \mathcal{E}_t(x) \text{ iff } x \in \operatorname{dom}(\mathcal{E}_t) \setminus \{e\}\,.$$

A corresponding $\mathsf{DeleteEntity}$ event is persisted to the event log containing $e$ for the relevant field.

Note that, by default, $\epsilon_{t+1} = \epsilon_t$ and $\mathcal{E}_{t+1} = \mathcal{E}_t$, unless one of the situations above apply. It is straightforward to show that the *entity integrity invariants* are maintained by the operations described above (the proof follows by induction on the timestamp $t$). Internally, that is, for the report engine compare Figure 6.2, the entity store provides a lookup function $\operatorname{lookup}_t : \mathit{Ent} \times [0,t] \rightharpoonup_{\text{fin}} \mathit{Record}$, where $\operatorname{lookup}_t(e, t')$ provides the latest value associated with the entity $e$ at time $t'$, where $t$ is the current time. Note that this includes the case in which $e$ has been deleted at or before time $t'$. In that case, the value associated with $e$ just before the deletion is returned. Formally, $\operatorname{lookup}_t$ is defined in terms of the entity environments as follows:

$$\operatorname{lookup}_t(e, t_1) = v \text{ iff } \exists t_2 \leq t_1 \colon \epsilon_{t_2}(e) = v \text{ and } \forall t_2 < t_3 \leq t_1 \colon e \notin \operatorname{dom}(\epsilon_{t_3}).$$

In particular, we have that if $e \in \operatorname{dom}(\epsilon_{t_1})$ then $\operatorname{lookup}_t(e, t_1) = \epsilon_{t_1}(e)$.

From this definition and the invariants of the system, we obtain the following property:

**Corollary 6.2.8.** *Let $(\mathcal{R}_t)_{t\in T}$, $(\mathcal{E}_t)_{t\in T}$, and $(\epsilon_t)_{t\in T}$ be sequences of record typing environments, entity typing environments, and entity environments respectively, satisfying the entity integrity invariants and the record typing invariants. Then the following holds for all timestamps $t \leq t_1 \leq t_2$ and entities $e \in Ent$:*

$$\textit{If } \mathcal{R}_t, \widehat{\mathcal{E}_t} \vdash e : \langle r \rangle \ \textit{ then } \operatorname{lookup}_{t_2}(e, t_1) = v \textit{ for some } v \textit{ and } \mathcal{R}_{t_2}, \widehat{\mathcal{E}_{t_2}} \vdash v : r.$$

*Proof.* Assume that $\mathcal{R}_t, \widehat{\mathcal{E}_t} \vdash e : \langle r \rangle$. Then it follows from the typing rule for entity values and the subtyping rules that $\widehat{\mathcal{E}_t}(e) = r'$ for some $r'$ with $r' \leq_t r$. That is, there is some $t' \leq t$ with $\mathcal{E}_{t'}(e) = r'$. Hence, from the well-definedness invariant it follows that $\epsilon_{t'}(e)$ is defined. Since $t' \leq t \leq t_1$, we can thus conclude that $\operatorname{lookup}_{t_2}(e, t_1) = (r'', m)$, for some record value $(r'', m)$.

According to the definition of $\operatorname{lookup}_{t_2}$, we then have some $t_3 \leq t_1$ with $\epsilon_{t_3}(e) = (r'', m)$. Applying the well-typing invariant, we obtain some $t_4 \leq t_3$ with $\mathcal{R}_{t_4}, \mathcal{E}_{t_4} \vdash (r'', m) : \mathcal{E}_{t_3}(e)$. Since, by the stable type invariant, $\mathcal{E}_{t_3}(e) = \mathcal{E}_{t'}(e) = r'$, we then have that $\mathcal{R}_{t_4}, \mathcal{E}_{t_4} \vdash (r'', m) : r'$. Moreover, according to the typing rules, this can only be the case if $r'' \leq_{t_4} r'$.

Due to the monotonicity invariant, we know that $\mathcal{R}_{t_2} = \mathcal{R}_{t_4} \cup \mathcal{R}_\Delta$ for some $\mathcal{R}_\Delta$. In particular, this means that $r'' \leq_{t_4} r'$ implies that $r'' \leq_{t_2} r'$. Similarly, $r' \leq_t r$ implies that $r' \leq_{t_2} r$. Hence, by transitivity of $\leq_{t_2}$, we have that $r'' \leq_{t_2} r$.

According to the implementation of the entity store, we know that $\epsilon_{t_3}(e) = (r'', m)$ implies that $(r'', m)$ occurs in the event log (as part of an event of type $\mathsf{CreateEntity}$ or $\mathsf{UpdateEntity}$) at least from $t_3$ onwards, in particular in the event log $l_{t_2}$ at $t_2$. Since, by the log typing invariant, the event log $l_{t_2}$ is well-typed as $\mathcal{R}_{t_2}, \widehat{\mathcal{E}_{t_2}} \vdash l_{t_2} : [\mathsf{Event}]$, we know that $\mathcal{R}_{t_2}, \widehat{\mathcal{E}_{t_2}} \vdash (r'', m) : r''$. From the subtype relation $r'' \leq_{t_2} r$ we can thus conclude $\mathcal{R}_{t_2}, \widehat{\mathcal{E}_{t_2}} \vdash (r'', m) : r$. □

The corollary above describes the fundamental safety property with respect to entity values: if an entity value previously entered the system, and hence type checked, then all future dereferencing will not get stuck, and the obtained value will be well-typed with respect to the accumulated entity typing environment.

### 6.2.4 Report Engine

The purpose of the report engine is to provide a structured view of the database that is constituted by the system's event log. This structured view of the data in the event log comes in the form of a *report*, which provides a collection of condensed structured information compiled from the event log. Conceptually, the data provided by a report is compiled from the event log by a function of type $[\mathsf{Event}] \to \mathsf{Report}$, a *report function*. The *report language* provides a means to specify such a report function in a declarative manner. The interface of the report engine is summarised in Figure 6.7.

#### 6.2.4.1 The Report Language

In this section, we provide an overview over the report language. The report language is—much like the query fragment of *SQL*—a functional language *without side effects*. It only provides operations to non-destructively manipulate and combine values. Since the system's storage is based on a shallow event log, the report language

| Report Engine | | |
|---|---|---|
| **Function** | **Input** | **Output** |
| *addReport* | name, type, description, tags, report definition | |
| *updateReport* | name, type, description, tags, report definition | |
| *deleteReport* | name | |
| *queryReport* | name, list of values | value |

Figure 6.7: Report engine interface.

must provide operations to relate, filter, join, and aggregate pieces of information. Moreover, as the data stored in the event log is inherently heterogeneous—containing data of different kinds—the report language offers a comprehensive type system that allows us to safely operate in this setting.

**Example 6.2.9.** Consider the following simple report function that lists all reports available in the system:

$$
\begin{array}{l}
\mathit{reports} : [\mathsf{PutReport}] \\
\mathit{reports} = \mathit{nubProj}\ (\lambda x \to x.\mathit{name})\ [\mathit{pr} \mid \\
\quad \mathit{cr} : \mathsf{CreateReport} \leftarrow \mathbf{events}, \\
\quad \mathit{pr} : \mathsf{PutReport} = \mathit{first}\ \mathit{cr}\ [\mathit{ur} \mid \mathit{ur} : \mathsf{ReportEvent} \leftarrow \mathbf{events}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{ur.name} \equiv \mathit{cr.name}]]
\end{array}
$$

The report function above uses the two functions *nubProj* and *first*, which are defined in the standard library of the report language. The function *nubProj* of type $(\mathsf{Eq}\ b) \Rightarrow (a \to b) \to [a] \to [a]$ removes duplicates in the given list according to the equality on the result of the provided projection function. In the example above, reports with the same name are considered duplicates. The function $\mathit{first} : a \to [a] \to a$ returns the first element of the given list or the default value provided as first argument if the list is empty.

Every report function implicitly has as its first argument the event log of type [Event]—a list of events—bound to the name **events**. The syntax—and to large parts also the semantics—is based on Haskell [62]. The central data structure is that of lists. In order to formulate operations on lists concisely, we use list comprehensions [113] as seen in Example 6.2.9. A list comprehension of the form $[\ e \mid c\ ]$ denotes a list containing elements of the form $e$ generated by $c$, where $c$ is a sequence of *generators* and *filters*.

As we have mentioned, access to type information and its propagation to subsequent computations is essential due to the fact that the event log is a list of heterogeneously typed elements. The generator $\mathit{cr} : \mathsf{CreateReport} \leftarrow \mathbf{events}$ iterates through elements of the list **events**, binding each element to the variable *cr*. The typing $\mathit{cr} : \mathsf{CreateReport}$ restricts this iteration to elements of type CreateReport, a subtype of Event. This type information is propagated through the subsequent generators and filters of the list comprehension. In the filter $\mathit{ur.name} \equiv \mathit{cr.name}$, we use the fact that elements of type ReportEvent have a field name of type **String**. When binding the first element of the result of the nested list comprehension to the variable *pr* it is also checked whether this element is in fact of type PutReport. Thus we ignore reports that are marked as deleted via a DeleteReport event.

The report language is based on the simply typed lambda calculus extended with polymorphic (non-recursive) let expressions as well as type case expressions. The core language is given by the following grammar:

$$e ::= x \mid c \mid \lambda x.e \mid e_1\; e_2 \mid \textbf{let}\; x = e_1 \;\textbf{in}\; e_2 \mid \textbf{type}\; x = e \;\textbf{of}\; \{r \to e_1; \_ \to e_2\}\,,$$

where $x$ ranges over variables, and $c$ over constants which include integers, Booleans, tuples and list constructors as well as operations on them like $+$, *if-then-else* etc. In particular, we assume a fold operation **fold** of type $(\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta$. This is the only operation of the report language that permits recursive computations on lists. List comprehensions are mere syntactic sugar and can be reduced to **fold** and let expressions as for example in Haskell [62].

The extended list comprehensions of the report language that allow filtering according to run-time type information depend on type case expressions of the form $\textbf{type}\; x = e \;\textbf{of}\; \{r \to e_1; \_ \to e_2\}$. In such a type case expression, an expression $e$ of some record type $r_e$ gets evaluated to a record value $v$ which is then bound to a variable $x$. The record type $r$ that the record value $v$ is matched against can be any subtype of $r_e$. Further evaluation of the type case expression depends on the type $r_v$ of the record value $v$. This type can be any subtype of $r_e$. If $r_v \leq r$ then the evaluation proceeds with $e_1$, otherwise with $e_2$. Binding $e$ to a variable $x$ allows us to use the stricter type $r$ in the expression $e_1$.

Another important component of the report language consists of the dereferencing operators ! and @, which give access to the lookup operator provided by the entity store. Given an expression $e$ of an entity type $\langle r\rangle$, both dereferencing operators provide a value $v$ of type $r$. That is, both ! and @ are unary operators of type $\langle r\rangle \to r$ for any record type $r$. In the case of the operator !, the resulting record value $v$ is the latest value associated with the entity to which $e$ evaluates. More concretely, given an entity value $v$, the expression $v!$ evaluates to the record value $\mathrm{lookup}_t(v, t)$, where $t$ is the current timestamp.

On the other hand, the *contextual* dereference operator @ provides as the result the value associated with the entity at the moment the entity was used in the event log (based on the internalTimeStamp field). This is the case when the entity is extracted from some event from the event log. Otherwise, the entity value stems from an actual argument to the report function. In the latter case @ behaves like the ordinary dereference operator !. In concrete terms, every entity value $v$ that enters the event log is annotated with the timestamp of the event it occurs in. That is, each entity value embedded in an event $e$ in the event log, occurs in an annotated form $(v, s)$, where $s$ is the value of $e$'s internalTimeStamp field. Given such an annotated entity value $(v, s)$, the expression $(v{,}s)@$ evaluates to $\mathrm{lookup}_t(v, s)$ and given a bare entity value $v$ the expression $v@$ evaluates to $\mathrm{lookup}_t(v, t)$.

Note that in each case for either of the two dereference operators, Corollary 6.2.8 guarantees that the lookup operation yields a record value of the right type. That is, both $! : \langle r\rangle \to r$ and $@ : \langle r\rangle \to r$ are total functions that never get stuck.

**Example 6.2.10.** The entity store and the contextual dereferencing operator provide a solution to a recurring problem in ERP systems, namely how to maintain historical data for auditing. For example, when an invoice is issued in a sale, then a copy of the customer information *at the time* of the invoice is needed for auditing. Traditional ERP systems solve the problem by explicit copying of data, since referenced data might otherwise get destructively updated.

Since data is never deleted in a POETS system, we can solve the problem without copying. Consider the following definition of transactions that represent issuing of invoices, and invoices respectively (we assume that the record types Customer and OrderLine are already defined):

| | |
|---|---|
| *IssueInvoice is a Transaction.* | *Invoice is Data.* |
| *IssueInvoice has a Customer entity.* | *Invoice has a Customer.* |
| *IssueInvoice has a list of OrderLine.* | *Invoice has a list of OrderLine.* |

Rather than containing a Customer record, an IssueInvoice transaction contains a Customer entity, which eliminates copying of data. From an IssueInvoice transaction we can instead *derive* the invoice data by the following report function:

$$\begin{aligned}
&\mathit{invoices} : [\mathsf{Invoice}]\\
&\mathit{invoices} = [\mathsf{Invoice}\{\mathit{customer} = \mathit{ii.customer@},\ \mathit{orderLines} = \mathit{ii.orderLines}\} \mid\\
&\quad \mathit{tr} : \mathsf{TransactionEvent} \leftarrow \mathbf{events},\\
&\quad \mathit{ii} : \mathsf{IssueInvoice} = \mathit{tr.transaction}]
\end{aligned}$$

Note how the @ operator is used to dereference the customer data: since the *ii.customer* value originates from an event in the event log, the contextual dereferencing will produce data associated with the customer at the time when the invoice was issued, as required.

### 6.2.4.2 Incrementalisation

While the type system is important in order to avoid obvious specification errors, it is also important to ensure a fast execution of the thus obtained functional specifications. This is, of course, a general issue for querying systems. In our system it is, however, of even greater importance since shifting the structure of the data—from the data store to the domain of queries—means that queries operate on the complete data set of the database. In principle, the data of each report has to be recomputed after each transaction by applying the corresponding report function to the updated event log. In other words, if treated naïvely, the conceptual simplification provided by the flat event log has to be paid via more expensive computations.

This issue can be addressed by transforming a given report function $f$ into an incremental function $f'$ that updates the report data computed previously according to the changes that have occurred since the report data was computed before. That is, given an event log $l$ and an update to it $l \oplus e$, we require that $f(l \oplus e) = f'(f(l), e)$. The new report data $f(l \oplus e)$ is obtained by updating the previous report data $f(l)$ according to the changes $e$. In the case of the event log, we have a list structure. Changes only occur *monotonically*, by adding new elements to it: given an event log $l$ and a new event $e$, the new event log is $e \# l$, where $\#$ is the list constructor of type $\alpha \to [\alpha] \to [\alpha]$.

Here it is crucial that we have restricted the report language such that operations on lists are limited to the higher-order function **fold**. The fundamental idea of incrementalising report functions is based on the following equation satisfied by **fold**:

$$\mathbf{fold}\ f\ e\ (x \# xs) = f\ x\ (\mathbf{fold}\ f\ e\ (xs))$$

Based on this idea, we are able to make the computation of most report functions independent of the size of the event log but only dependent of the changes to the event log and the previous result of the report function [77]. Unfortunately, if we consider for example list comprehensions containing more than one generator, we have functions with nested folds. In order to properly incrementalise such functions, we need to move from list structures to multisets. This is, however, only rarely a practical restriction since most aggregation functions are based on commutative binary operations and are thus oblivious to ordering.

#### 6.2.4.3 Lifecycle of Reports

Like entities, the set of reports registered in a running POETS instance—and thus available for querying—can be changed via the external interface to the report engine. To this end, the report engine interface provides the operations *addReport*, *updateReport*, and *deleteReport*. The former two take a *report specification* that contains the name of the report, the definition of the report function that generates the report data and the type of the report function. Optionally, it may also contain further meta information in the form of a description text and a list of tags.

**Example 6.2.11.** Reconsider the function defined in Example 6.2.9 that lists all active reports with all their meta data. The following report specification uses the report function from Example 6.2.9 in order to define a report function that lists the names of all active report:

**name**: *ReportNames*
**description**: *A list of names of all registered reports.*
**tags**: *internal*, *report*

$reports : [\mathsf{PutReport}]$
$reports = nubProj\ (\lambda x \rightarrow x.name)\ [pr \mid$
$\quad cr : \mathsf{CreateReport} \leftarrow \mathbf{events},$
$\quad pr : \mathsf{PutReport} = \mathit{first}\ cr\ [ur \mid ur : \mathsf{ReportEvent} \leftarrow \mathbf{events},$
$\qquad\qquad\qquad\qquad\qquad\qquad ur.name \equiv cr.name]]$

$\mathbf{report} : [\mathbf{String}]$
$\mathbf{report} = [r.name \mid r \leftarrow reports]$

In the header of the report specification, the name and optionally also a description text as well as a list of tags is provided as meta data to the actual report function specification. Every report specification must define a top-level function called **report**, which provides the report function that derives the report data from the event log. In the example above, this function takes no (additional) arguments and returns a list of strings—the names of active reports.

Calls to *addReport* and *updateReport* are both reflected by a corresponding event of type CreateReport and UpdateReport respectively. Both events are subtypes of PutReport and contain the meta information as well as the original specification text of the concerning report. When a report is no longer needed, it can be removed from the report engine by a corresponding *deleteReport* operation. Note that the change and removal of reports only affect the state of the POETS system from the

Contract Engine

| **Function** | **Input** | **Output** |
|---|---|---|
| *createTemplate* | name, type, description, specification | |
| *updateTemplate* | name, type, description, specification | |
| *deleteTemplate* | name | |
| *createContract* | meta data | contract ID |
| *updateContract* | contract ID, meta data | |
| *concludeContract* | contract ID | |
| *getContract* | contract ID | contract state |
| *registerTransaction* | contract ID, timestamp, transaction | |

Figure 6.8: Contract engine interface.

given point in time. Transactions that occurred prior to a change or deletion of a report are not affected. This is important for the system's ability to fully recover after a crash by replaying the events from the event log.

The remaining operation provided by the report engine—*queryReport*—is the core functionality of the reporting system. Given a name of a registered report and a list of arguments, this operation supplies the given arguments to the corresponding report function and returns the result. For example, the *ReportNames* report specified in Example 6.2.11 does not require any arguments—its type is [**String**]—and returns the names of registered reports.

### 6.2.5 Contract Engine

The role of the contract engine is to determine which transactions—that is external events, compare Section 6.2.2—are expected by the system. Transactions model events that take place according to an *agreement*, for instance a delivery of goods in a sale, a payment in a lease agreement, or a movement of items from one inventory to another in a production plan. Such agreements are referred to as *contracts*, although they need not be legally binding contracts. The purpose of a contract is to provide a detailed description of *what* is expected, by *whom*, and *when*. A sales contract, for example, may stipulate that first the company sends an invoice, then the customer pays within a certain deadline, and finally the company delivers goods within another deadline.

The interface of the contract engine is summarised in Figure 6.8.

#### 6.2.5.1 Contract Templates

In order to specify contracts such as the aforementioned sales contract, we use an extended variant of the contract specification language (CSL) of Hvitved et al. [52], which we will refer to as the POETS contract specification language (PCSL) in the following. For reusability, contracts are always specified as *contract templates* rather than as concrete contracts. A contract template consists of four parts: (i) a template name, (ii) a template type, which is a subtype of the Contract record type, (iii) a textual description, and (iv) a PCSL specification. We describe PCSL in Section 6.2.5.3.

The template name is a unique identifier, and the template type determines the parameters that are available in the contract template.

**Example 6.2.12.** We may define the following type for sales contracts in the ontology language (assuming that the record types Customer, Company, and Goods have been defined):

*Sale is a Contract.*
*Sale has a Customer entity.*
*Sale has a Company entity.*
*Sale has a list of Goods.*
*Sale has an Int called amount.*

With this definition, contract templates of type Sale are parametrised over the fields customer, company, goods, and amount of types ⟨Customer⟩, ⟨Company⟩, [Goods], and **Int**, respectively.

The contract engine provides an interface to add contract templates (*createTemplate*), update contract templates (*updateTemplate*), and remove contract templates (*deleteTemplate*) from the system at run-time. The structure of contract templates is reflected in the external event types CreateContractDef, UpdateContractDef, and DeleteContractDef, compare Section 6.2.2. A list of (non-deleted) contract templates can hence be computed by a report, similar to the list of (non-deleted) reports from Example 6.2.11.

### 6.2.5.2 Contract Instances

A contract template is instantiated via *createContract* by supplying a record value $v$ of a subtype of Contract. Besides custom fields, which depend on the type at hand, such a record always contains the fields templateName and startDate inherited from the Contract record type, compare Appendix E.1. The field templateName contains the name of the template to instantiate, and the field startDate determines the start date of the contract. The fields of $v$ are substituted into the contract template in order to obtain a *contract instance*, and the type of $v$ must therefore match the template type. For instance, if $v$ has type Sale then the field templateName must contain the name of a contract template that has type Sale. We refer to the record $v$ as *contract meta data*.

When a contract $c$ is instantiated by supplying contract meta data $v$, a *fresh* contract identifier $i$ is created, and a CreateContract event is persisted in the event log with with contract $= v$ and contractId $= i$. Hereafter, transactions $t$ can be registered with the contract via *registerTransaction*, which will update the contract to a *residual contract* $c'$, written $c \xrightarrow{t} c'$, and a TransactionEvent with transaction $= t$ and contractId $= i$ is written to the event log. The state of the contract can be acquired from the contract engine at any given point in time via *getContract*, which enables run-time analyses of contracts, for instance in order to generate a list of expected transactions.

Registration of a transaction $c \xrightarrow{t} c'$ is only permitted if the transaction is expected in the current state $c$. That is, there need not be a residual state for all transactions. After zero or more successful transactions, $c \xrightarrow{t_1} c_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} c_n$, the contract may be concluded via *concludeContract*, provided that the residual contract

$c_n$ does not contain any outstanding obligations. This results in a ConcludeContract event to be persisted in the event log.

The lifecycle described above does not take into account that contracts may have to be updated at run-time, for example if it is agreed to extend the payment deadline in a sales contract. To this end, running contracts are allowed to be updated, simply by supplying new contract meta data (*updateContract*). The difference in the new meta data compared to the old meta data may not only be a change of, say, items to be sold, but it may also be a change in the field templateName. The latter makes it is possible to replace the old contract by a qualitatively different contract, since the new contract template may describe a different workflow. There is, however, an important restriction: a contract can only be updated if any previous transactions registered with the contract also conform with the new contract. That is, if the contract has evolved like $c \xrightarrow{t_1} c_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} c_n$, and an update to a new contract $c'$ is requested, then only if $c' \xrightarrow{t_1} c'_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} c'_n$, for some $c'_1, \ldots, c'_n$, is the update permitted. A successful update results in an UpdateContract event to be written to the event log with the new meta data.

Note that, for simplicity, we only allow the updates described above. Another possibility is to allow updates where the current state of the contract $c$ is replaced directly by a new state $c'$. Although we can achieve this effect via a suitably defined contract template and the *updateContract* function above, a direct update is preferable.

As for contract templates, a list of (non-concluded) contract instances can be computed by a report that inspects CreateContract, UpdateContract, and ConcludeContract events respectively.

#### 6.2.5.3 The Contract Language

The fourth component of contract templates—the PCSL specification—is the actual normative content of contract templates. The core grammar for PCSL is presented in Figure 6.9. PCSL extends CSL mainly at the level of expressions $E$, by adding support for the value types in POETS, as well as lambda abstractions and function applications. At the level of clauses $C$, PCSL is similar to CSL, albeit with a slightly altered syntax.

The semantics of PCSL is a straightforward extension of that of CSL [52], although we use a partial small-step semantics rather than CSL's total small-step semantics. That is, there need not be a residue for all clauses and transactions, as described in Section 6.2.5.2. This is simply in order to prevent "unexpected" events from entering the system, for instance we only allow a payment to be entered into the system if a running contract expects that payment.

The type system for clauses is identical with CSL. Typing of expressions is, however, more challenging since we have introduced (record) polymorphism as well as subtyping. We will not present the extended semantics nor the extended typing rules, but only remark that the typing serves the same purpose as in CSL: evaluation of expressions does not get stuck and always terminates, and contracts have unique blame assignment.

**Example 6.2.13.** We demonstrate PCSL by means of an example, presented in Figure 6.10. The contract template is of the type Sale from Example 6.2.12, which

$$
\begin{array}{lll}
\mathit{Tmp} ::= & \textbf{name} : \mathit{ContractName} & \text{(contract template)} \\
 & \textbf{type} : \mathit{RecordName} & \\
 & \textbf{description} : \mathit{String} & \\
 & \mathit{Def} \ldots \mathit{Def}\ \textbf{contract} = C & \\
\mathit{Def} ::= & \textbf{val}\ \mathit{Var} = E & \text{(value definition)} \\
 \mid & \textbf{clause}\ \mathit{ClauseName}(\mathit{Var} : T, \ldots, \mathit{Var} : T) & \text{(clause template)} \\
 & \qquad \langle \mathit{Var} : T, \ldots, \mathit{Var} : T \rangle = C & \\
C ::= & \textbf{fulfilment} & \text{(no obligations)} \\
 \mid & \langle E \rangle\ \mathit{RecordName}(F, \ldots, F) & \text{(obligation)} \\
 & \ \textbf{where}\ E\ \textbf{due}\ D\ \textbf{remaining}\ \mathit{Var}\ \textbf{then}\ C & \\
 \mid & \textbf{when}\ \mathit{RecordName}(F, \ldots, F) & \text{(external choice)} \\
 & \ \textbf{where}\ E\ \textbf{due}\ D\ \textbf{remaining}\ \mathit{Var}\ \textbf{then}\ C\ \textbf{else}\ C & \\
 \mid & \textbf{if}\ E\ \textbf{then}\ C\ \textbf{else}\ C & \text{(internal choice)} \\
 \mid & C\ \textbf{and}\ C & \text{(conjunction)} \\
 \mid & C\ \textbf{or}\ C & \text{(disjunction)} \\
 \mid & \mathit{ClauseName}(E, \ldots, E)\langle E, \ldots, E \rangle & \text{(instantiation)} \\
F ::= & \mathit{FieldName}\ \mathit{Var} & \text{(field binder)} \\
R ::= & \mathit{RecordName}\ \mathit{Var} & \text{(record binder)} \\
T ::= & \mathit{TypeVar} & \text{(type variable)} \\
 \mid & () & \text{(unit type)} \\
 \mid & \textbf{Bool} \mid \textbf{Int} \mid \textbf{Real} \mid \textbf{String} & \text{(type constants)} \\
 \mid & \textbf{Timestamp} \mid \textbf{Duration} & \\
 \mid & \mathit{RecordName} & \text{(record type)} \\
 \mid & [T] & \text{(list type)} \\
 \mid & \langle T \rangle & \text{(entity type)} \\
 \mid & T \to T & \text{(function type)} \\
E ::= & \mathit{Var} & \text{(variable)} \\
 \mid & \mathit{BaseValue} & \text{(base value)} \\
 \mid & \mathit{RecordName}\{\mathit{FieldName} = E, \ldots, \mathit{FieldName} = E\} & \text{(record expression)} \\
 \mid & [E, \ldots, E] & \text{(list expression)} \\
 \mid & \lambda\, \mathit{Var} \to E & \text{(function abstraction)} \\
 \mid & E\ E & \text{(function application)} \\
 \mid & E \oplus E & \text{(binary expression)} \\
 \mid & E.\mathit{FieldName} & \text{(field projection)} \\
 \mid & E\{\mathit{FieldName} = E\} & \text{(field update)} \\
 \mid & \textbf{if}\ E\ \textbf{then}\ E\ \textbf{else}\ E & \text{(conditional)} \\
 \mid & \textbf{case}\ E\ \textbf{of}\ R \to E \mid \cdots \mid R \to E & \text{(record type casing)} \\
D ::= & \textbf{after}\ E\ \textbf{within}\ E & \text{(deadline expression)} \\
\oplus ::= & \times \mid / \mid + \mid \langle\times\rangle \mid \langle + \rangle \mid \# \mid \equiv \mid \leq \mid \wedge & \text{(binary operators)}
\end{array}
$$

Figure 6.9: Grammar for the core contract language PCSL. *ContractName* is the set of all contract template names, *ClauseName* is the set of all clause template names ranged over by $k$, *Var* is the set of all variable names ranged over by $x$, *TypeVar* is the set of all type variable names ranged over by $\alpha$, and $\mathit{BaseValue} = \mathit{Bool} \uplus \mathit{Int} \uplus \mathit{Real} \uplus \mathit{String} \uplus \mathit{Timestamp} \uplus \mathit{Duration} \uplus \mathit{Ent}$.

**name**: *salesContract*
**type**: Sale
**description**: `"A simple sales contract between a company and a customer"`

**fun** $\mathit{elem}\ x = \mathit{foldr}\ (\lambda y\ b \to x \equiv y \lor b)\ \mathit{false}$
**fun** $\mathit{filter}\ f = \mathit{foldr}\ (\lambda x\ b \to$ **if** $f\ x$ **then** $x \mathbin{\#} b$ **else** $b)\ []$
**fun** $\mathit{subset}\ l1\ l2 = \mathit{all}\ (\lambda x \to \mathit{elem}\ x\ l2)\ l1$
**fun** $\mathit{diff}\ l1\ l2 = \mathit{filter}\ (\lambda x \to \neg\ (\mathit{elem}\ x\ l2))\ l1$

**clause** $\mathit{sale}(\mathit{goods} : [\mathsf{Goods}], \mathit{amount} : \mathbf{Int})\langle \mathit{comp} : \langle\mathsf{Company}\rangle, \mathit{cust} : \langle\mathsf{Customer}\rangle\rangle =$
$\langle \mathit{comp}\rangle$ IssueInvoice$(\mathit{goods}\ g, \mathit{amount}\ a)$
  **where** $g \equiv \mathit{goods} \land a \equiv \mathit{amount}$ **due immediately**
 **then**
 $\langle \mathit{cust}\rangle$ Payment$(\mathit{amount}\ a)$
  **where** $a \equiv \mathit{amount}$ **due within** $14D$
 **and**
 $\mathit{delivery}(\mathit{goods}, 1\,W)\langle \mathit{comp}\rangle$

**clause** $\mathit{delivery}(\mathit{goods} : [\mathsf{Goods}], \mathit{deadline} : \mathbf{Duration})\langle \mathit{comp} : \langle \mathit{company}\rangle\rangle =$
 **if** $\mathit{goods} \equiv []$ **then**
  **fulfilment**
 **else**
  $\langle \mathit{comp}\rangle$ Delivery$(\mathit{goods}\ g)$
   **where** $g \not\equiv [] \land \mathit{subset}\ g\ \mathit{goods}$ **due within** $\mathit{deadline}$ **remaining** $r$
  **then**
  $\mathit{delivery}(\mathit{diff}\ \mathit{goods}\ g, r)\langle \mathit{comp}\rangle$

**contract** $= \mathit{sale}(\mathit{goods}, \mathit{amount})\langle \mathit{company}, \mathit{customer}\rangle$

Figure 6.10: PCSL sales contract template of type Sale.

means that the fields goods, amount, company, and customer are available in the body of the contract template, that is the right-hand side of the **contract** keyword. Hence, concrete values are substituted from the contract meta data when the template is instantiated, as described in Section 6.2.5.2.

The example uses standard syntactic sugar at the level of expressions, for instance $\neg e$ means **if** $e$ **then** *false* **else** *true* and $e_1 \lor e_2$ means $\neg(\neg e_1 \land \neg e_2)$. Moreover, we omit the **after** part of a deadline if it is 0, we write **immediately** for **within** 0, we omit the **remaining** part if it is not used, and we write **fun** $f\,x_1 \cdots x_n = e$ for **val** $f = \lambda x_1 \to \cdots \lambda x_n \to e$.

The template implements a simple workflow: first the company issues an invoice, then the customer pays within 14 days, and simultaneously the company delivers goods within a week. Delivery of goods is allowed to take place in multiple deliveries, which is coded as the recursive clause template *delivery*. Note how the variable $r$ is bound to the remainder of the deadline: all deadlines in a **then** branch are relative to the time of the guarding transaction, hence the relative deadline for delivering the remaining goods is whatever remains of the original one week deadline. Note also that the initial reference time of a contract instance is determined by the field startDate in the contract meta data, compare Appendix E.1. Hence if the contract above is instantiated with start date $t \in \mathit{Timestamp}$, then the invoice is supposed to be issued at time $t$.

Finally, we remark that obligation clauses are binders. That is, for instance the variable $g$ is bound to value of the field goods of the IssueInvoice transaction when it takes place, and the scope of $g$ is the **where** clause and the continuation clause following the **then** keyword.

**Built-in symbols** PCSL has a small set of built-in symbols, from which other standard functions can be derived:

$foldl : (a \to b \to a) \to a \to [b] \to a$
$foldr : (a \to b \to b) \to b \to [a] \to b$
$ceil : \mathsf{Real} \to \mathbf{Int}$
$reports : \mathsf{Reports}$

The list includes fold operations in order to iterate over lists, since explicit recursion is not permitted, and a special constant *reports* of type Reports. The record type Reports is internally derived from the active reports in the report engine, and it is used only in the contract engine in order to enable querying of reports from within contracts. The record type contains one field per report. For instance, if the report engine contains a single report *Inventory* of type Inventory, then the typing of Report is (using the same notation as in Section 6.2.1.1):

$$\rho(\mathsf{Reports}) = \{(\mathsf{inventory}, () \to \mathsf{Inventory})\},$$

and the expression *reports.inventory* () invokes the report.

## 6.3 Use Case: $\mu$ERP

In this section we describe a use case instantiation of POETS, which we refer to as $\mu$ERP. With $\mu$ERP we model a simple ERP system for a small bicycle shop. Naturally, we do not intend to model all features of a full-blown ERP system, but rather we demonstrate a limited set of core ERP features. In our use case, the shop purchases bicycles from a bicycle vendor, and sells those bicycles to customers. We want to make sure that the bicycle shop only sells bicycles in stock, and we want to model a repair guarantee, which entitles customers to have their bikes repaired free of charge up until three months after purchase.

Following Henglein et al. [41], we also provide core financial reports, namely the income statement, the balance sheet, the cash flow statement, the list of open (not yet paid) invoices, and the value-added tax (VAT) report. These reports are typical, minimal legal requirements for running a business. We provide some example code in this section, and the complete specification is included in Appendix E.2. As we have seen in Section 6.2, instantiating POETS amounts to defining a data model, a set of reports, and a set of contract templates. We describe each of these components in the following subsections.

### 6.3.1 Data Model

The data model of $\mu$ERP is tailored to the ERP domain in accordance with the REA ontology [64]. Therefore, the main components of the data model are resources, transactions (that is, events associated with contracts), and agents. The complete data model is provided in Appendix E.2.1.

**Agents** are modelled as an abstract type Agent. An agent is either a Customer, a Vendor, or a special Me agent. Customers and Vendors are equipped with a name and an address. The Me type is used to represent the bicycle company itself. In a more elaborate example, the Me type will have subtypes such as Inventory or SalesPerson to represent subdivisions of, or individuals in, the company. The agent model is summarised below:

*Agent is Data.*

*Customer is an Agent.*
*Customer has a String called name.*
*Customer has an Address.*

*Me is an Agent.*

*Vendor is an Agent.*
*Vendor has a String called name.*
*Vendor has an Address.*

**Resources** are—like agents—Data. In our modelling of resources, we make a distinction between *resource types* and *resources*. A resource type represents a kind of resource, and resource types are divided into currencies (Currency) and item types (ItemType). Since we are modelling a bicycle shop, the only item type (for now) is bicycles (Bicycle). A resource is an instance of a resource type, and—similar to resource types—resources are divided into money (Money) and items (Item). Our modelling of items assumes an implicit unit of measure, that is we do not explicitly model units of measure such as pieces, boxes, pallets, etc. Our resource model is summarised below:

*ResourceType is Data.*
*ResourceType is abstract.*

*Currency is a ResourceType.*
*Currency is abstract.*

*DKK is a Currency.*
*EUR is a Currency.*

*ItemType is a ResourceType.*
*ItemType is abstract.*

*Bicycle is an ItemType.*

*Bicycle has a String called model.*

*Resource is Data.*
*Resource is abstract.*

*Money is a Resource.*
*Money has a Currency.*
*Money has a Double called amount.*

*Item is a Resource.*
*Item has an ItemType.*
*Item has a Double called quantity.*

**Transactions** (events in the REA terminology) are, not surprisingly, subtypes of the built-in Transaction type. The only transactions we consider in our use case are bilateral transactions (BiTransaction), that is transactions that have a sender and a receiver. Both the sender and the receiver are agent entities, that is a bilateral transaction contains references to two agents rather than copies of agent data. For our use case we model payments (Payment), deliveries (Delivery), issuing of invoices (IssueInvoice), requests for repair of a set of items (RequestRepair), and repair of a set of items (Repair). Issuing of invoices contain the relevant information for modelling of VAT, encapsulated in the OrderLine type. We include some of these definitions below:

*BiTransaction is a Transaction.*
*BiTransaction is abstract.*
*BiTransaction has an Agent entity called sender.*
*BiTransaction has an Agent entity called receiver.*

*Transfer is a BiTransaction.*
*Transfer is abstract.*

*Payment is a Transfer.*
*Payment is abstract.*
*Payment has Money.*

*CashPayment is a Payment.*
*CreditCardPayment is a Payment.*
*BankTransfer is a Payment.*

*IssueInvoice is a BiTransaction.*
*IssueInvoice has a list of OrderLine called orderLines.*

Besides agents, resources, and transactions, the data model defines the output types of reports (Appendix E.2.1.3) the input types of contracts (Appendix E.2.1.4), and generic data definitions such as Address and OrderLine. The report types define the five mandatory reports mentioned earlier, and additional Inventory and TopNCustomers report types. The contract types define the two types of contracts for the bicycle company, namely Purchase and Sale.

### 6.3.2 Reports

Report specifications are divided into prelude functions (Appendix E.2.2.1), domain-specific prelude functions (Appendix E.2.2.2), internal reports (Appendix E.2.2.3), and external reports (Appendix E.2.2.4).

**Prelude functions** are utility functions that are independent of the custom data model. These functions are automatically added to all POETS instances, but they are included in the appendix for completeness. The prelude includes standard functions such as *filter*, but it also includes generators for accessing event log data such as *reports*. The event log generators provide access to data that has a lifecycle such as contracts or reports, compare Section 6.2.2.

**Domain-specific prelude functions** are utility functions that depend on the custom data model. The *itemsReceived* function, for example, computes a list of all items that have been delivered to the company, and it hence relies on the Delivery transaction type (*normaliseItems* and *isMe* are also defined in Appendix E.2.2.2):

```
itemsReceived : [Item]
itemsReceived = normaliseItems [is |
  tr ← transactionEvents,
  del : Delivery = tr.transaction,
  ¬(isMe del.sender) ∧ isMe del.receiver,
  is ← del.items]
```

**Internal reports** are reports that are needed either by clients of the system or by contracts. For instance, the *ContractTemplates* report is needed by clients of

| **Report** | **Result** |
|---|---|
| *Me* | The special Me entity. |
| *Entities* | A list of all non-deleted entities. |
| *EntitiesByType* | A list of all non-deleted entities of a given type. |
| *ReportNames* | A list of names of all non-deleted reports. |
| *ReportNamesByTags* | A list of names of all non-deleted reports whose tags contain a given set and do not contain another given set. |
| *ReportTags* | A list of all tags used by non-deleted reports. |
| *ContractTemplates* | A list of names of all non-deleted contract templates. |
| *ContractTemplatesByType* | A list of names of all non-deleted contract templates of a given type. |
| *Contracts* | A list of all non-deleted contract instances. |
| *ContractHistory* | A list of previous transactions for a given contract instance. |
| *ContractSummary* | A list of meta data for a given contract instance. |

Figure 6.11: Internal reports.

the system in order to instantiate contracts, and the *Me* report is needed by the two contracts, as we shall see in the following subsection. A list of internal reports, including a short description of what they compute, is summarised in Figure 6.11. Except for the *Me* report, all internal reports are independent from the custom data model.

**External reports** are reports that are expected to be rendered directly in clients of the system, but they may also be invoked by contracts. The external reports in our use case are the reports mentioned earlier, namely the income statement, the balance sheet, the cash flow statement, the list of unpaid invoices, and the VAT report. Moreover, we include reports for calculating the list of items in the inventory, and the list of top-$n$ customers, respectively. We include the inventory report below as an example:

```
report : Inventory
report =
 let itemsSold' = map (λi → i{quantity = 0 − i.quantity}) itemsSold
 in
 −− The available items is the list of received items minus the
 −− list of reserved or sold items
 Inventory{availableItems = normaliseItems (itemsReceived ++ itemsSold')}
```

The value *itemsSold* is defined in the domain-specific prelude, similar to the value *itemsReceived*. But unlike *itemsReceived*, the computation takes into account that items can be reserved but not yet delivered. Hence when we check that items are in stock using the inventory report, we also take into account that some items in the inventory may have been sold, and therefore cannot be sold again.

The five standard reports are defined according to the specifications given by Henglein et al. [41, Section 2.1], but for simplicity we do not model fixed costs, depreciation, and fixed assets. We do, however, model multiple currencies, exemplified via Danish Kroner (DKK) and Euro (EUR). This means that financial reports, such as IncomeStatement, provide lists of values of type Money—one for each currency used.

### 6.3.3 Contracts

The specification of contracts is divided into prelude functions (Appendix E.2.3.1), domain-specific prelude functions (Appendix E.2.3.2), and contract templates (Appendix E.2.3.3).

**Prelude functions** are utility functions similar to the report engine's prelude functions. They are independent from the custom data model, and are automatically added to all POETS instances. The prelude includes standard functions such as *filter*.

**Domain-specific prelude functions** are utility functions that depend on the custom data model. The *inStock* function, for example, checks whether the items described in a list of order lines are in stock, by querying the *Inventory* report (we assume that the item types are different for each line):

```
fun $inStock\ lines =$
  let $inv = (reports.inventory\ ()).availableItems$
  in
  $all\ (\lambda l \rightarrow any\ (\lambda i \rightarrow (l.item).itemType \equiv i.itemType\ \wedge$
                              $(l.item).quantity \leq i.quantity)\ inv)\ lines$
```

**Contract templates** describe the daily activities in the company, and in our $\mu$ERP use case we only consider a purchase contract and a sales contract. The purchase contract is presented below:

```
name: $purchase$
type: Purchase
description: "Set up a purchase"

clause $purchase(lines : [\mathsf{OrderLine}])\langle me : \langle \mathsf{Me} \rangle, vendor : \langle \mathsf{Vendor} \rangle\rangle =$
 $\langle vendor \rangle$ Delivery($sender\ s$, $receiver\ r$, $items\ i$)
   where $s \equiv vendor \wedge r \equiv me \wedge i \equiv map\ (\lambda x \rightarrow x.item)\ lines$
   due within $1W$
 then
 when IssueInvoice($sender\ s$, $receiver\ r$, $orderLines\ sl$)
   where $s \equiv vendor \wedge r \equiv me \wedge sl \equiv lines$
   due within $1Y$
 then
 $payment(lines, vendor, 14D)\langle me \rangle$

clause $payment(lines : [\mathsf{OrderLine}], vendor : \langle \mathsf{Vendor} \rangle, deadline :$ Duration$)$
                   $\langle me : \langle \mathsf{Me} \rangle\rangle =$
 if $null\ lines$ then
   fulfilment
 else
   $\langle me \rangle$ BankTransfer($sender\ s$, $receiver\ r$, $money\ m$)
     where $s \equiv me \wedge r \equiv vendor \wedge checkAmount\ m\ lines$
     due within $deadline$
```

```
    remaining newDeadline
  then
  payment(remainingOrderLines m lines, vendor, newDeadline)⟨me⟩

contract = purchase(orderLines)⟨me, vendor⟩
```

The contract describes a simple workflow, in which the vendor delivers items, possibly followed by an invoice, which in turn is followed by a bank transfer of the company. Note how the *me* parameter in the contract template body refers to the value from the domain-specific prelude, which in turn invokes the *Me* report. Note also how the *payment* clause template is recursively defined in order to accommodate for potentially different currencies. That is, the total payment is split up in as many bank transfers as there are currencies in the purchase.

The sales contract is presented below:

```
name: sale
type: Sale
description: "Set up a sale"

clause sale(lines : [OrderLine])⟨me : ⟨Me⟩, customer : ⟨Customer⟩⟩ =
 ⟨me⟩ IssueInvoice(sender s, receiver r, orderLines sl)
   where s ≡ me ∧ r ≡ customer ∧ sl ≡ lines ∧ inStock lines
   due within 1H
 then
 payment(lines, me, 10m)⟨customer⟩
 and
 ⟨me⟩ Delivery(sender s, receiver r, items i)
   where s ≡ me ∧ r ≡ customer ∧ i ≡ map (λx → x.item) lines
   due within 1W
 then
 repair(map (λx → x.item) lines, customer, 3M)⟨me⟩

clause payment(lines : [OrderLine], me : ⟨Me⟩, deadline : Duration)
                    ⟨customer : ⟨Customer⟩⟩ =
 if null lines then
   fulfilment
 else
   ⟨customer⟩ Payment(sender s, receiver r, money m)
     where s ≡ customer ∧ r ≡ me ∧ checkAmount m lines
     due within deadline
     remaining newDeadline
   then
   payment(remainingOrderLines m lines, me, newDeadline)⟨customer⟩

clause repair(items : [Item], customer : ⟨Customer⟩, deadline : Duration)
                ⟨me : ⟨Me⟩⟩ =
 when RequestRepair(sender s, receiver r, items i)
   where s ≡ customer ∧ r ≡ me ∧ subset i items
   due within deadline
```

```
    **remaining** $\mathit{newDeadline}$
  **then**
  $\langle \mathit{me} \rangle$ Repair($\mathit{sender}\ s$, $\mathit{receiver}\ r$, $\mathit{items}\ i'$)
    **where** $s \equiv \mathit{me} \wedge r \equiv \mathit{customer} \wedge i \equiv i'$
    **due within** $5D$
  **and**
  $\mathit{repair}(\mathit{items}, \mathit{customer}, \mathit{newDeadline})\langle \mathit{me} \rangle$

**contract** = $\mathit{sale}(\mathit{orderLines})\langle \mathit{me}, \mathit{customer} \rangle$
```

The contract describes a workflow, in which the company issues an invoice to the customer—but only if the items on the invoice are in stock. The issuing of invoice is followed by an immediate (within an hour) payment by the customer to the company, and a delivery of goods by the company within a week. Moreover, we also model the repair guarantee mentioned in the introduction.

### 6.3.4 Bootstrapping the System

The previous subsections described the specification code for $\mu$ERP. Since data definitions, report specifications, and contract specifications are added to the system at run-time, $\mu$ERP is instantiated by invoking the following sequence of services on an initially empty POETS instance:

1. Add data definitions in Appendix E.2.1 via *addDataDefs*.
2. Create a designated Me entity via *createEntity*.
3. Add report specifications via *addReport*.
4. Add contract specifications via *createTemplate*.

Hence, the event log will, conceptually, have the form (we write the value of the field internalTimeStamp before each event):

```
$t_1$: AddDataDefs{defs = "ResourceType is ..."}

$t_2$: CreateEntity{ent = $e_1$, recordType = "Me", data = Me}

$t_3$: CreateReport{name = "Me", description = "Returns the ...",
     code = "name: Me\n ...", tags = ["internal","entity"]}
     ⋮

$t_i$: CreateReport{name = "TopNCustomers", description = "A list ...",
     code = "name: TopNCustomers\n ...",
     tags = ["external","financial","crm"]}

$t_{i+1}$: CreateContractDef{name = "Purchase", recordType = "Purchase",
     code = "name: purchase\n ...", description = "Set up ..."}

$t_{i+2}$: CreateContractDef{name = "Sale", recordType = "Sale",
     code = "name: sale\n ...", description = "Set up a sale"},
```

for some increasing timestamps $t_1 < t_2 < \ldots < t_{i+2}$. Note that the entity value $e_1$ of the CreateEntity event is automatically generated by the entity store, as described in Section 6.2.3.

Following these operations, the system is operational. That is, (i) customers and vendors can be managed via *createEntity*, *updateEntity*, and *deleteEntity*, (ii) contracts can be instantiated, updated, concluded, and inspected via *createContract*, *updateContract*, *concludeContract*, and *getContract* respectively, (iii) transactions can be registered via *registerTransaction*, and (iv) reports can be queried via *queryReport*.

For example, if a sale is initiated with a new customer John Doe, starting at time $t$, then the following events will be added to the event log:

```
t_{i+3}: CreateEntity{ent = e_2, recordType = "Customer", data = Customer{
      name = "John Doe", address = Address{
        string = "Universitetsparken 1", country = Denmark}}}

t_{i+4}: CreateContract{contractId = 0, contract = Sale{
      startDate = t, templateName = "sale", customer = e_2,
      orderLines = [OrderLine{
        item = Item{itemType = Bicycle{model = "Avenue"}, quantity = 1.0},
        unitPrice = Money{currency = DKK, amount = 4000.0},
        vatPercentage = 25.0}]}}.
```

That is, first the customer entity is created, and then we can instantiate a new sales contract. In this particular sale, one bicycle of the model “Avenue” is sold at a unit price of 4000 DKK, with an additional VAT of 25 percent. Note that the contract id 0 of the CreateContract is automatically generated and that the start time $t$ is explicitly given in the CreateContract’s startDate field independent from the internalTimeStamp field.

Following the events above, if the contract is executed successfully, events of type IssueInvoice, Delivery, and Payment will persisted in the event log with appropriate values—in particular, the payment will be 5000 DKK.

## 6.4 Implementation Aspects

In this section we briefly discuss some of the implementation techniques used in our implementation of POETS. POETS is implemented in Haskell [62], and the logical structure of the implementation reflects the diagram in Figure 6.2, that is each component is implemented as a separate Haskell module.

### 6.4.1 External Interface

The external interface to the POETS system is implemented in a separate Haskell module. We currently use Thrift [101] for implementing the communication layer between the server and its clients, but other communication layers can in principle be used. Changing the communication layer will only require a change in one module.

Besides offering an abstract, light-weight interface to communication, Thrift enables type-safe communication. The types and services of the server are specified in a language-independent description language, from which Haskell code is generated (or code in other languages for the clients). For example, the external interface to querying a report can be specified as follows:

```
Value queryReport(
  1 : string name      // name of the report to execute
  2 : list<Value> args // input arguments
) throws (
  1 : ReportNotFoundException notFound
  2 : RunTimeException runtime
  3 : TypeException type
)
```

From this specification, Thrift generates the Haskell code for the server interface, and implementing the interface amounts to supplying a function of the type $String \to [\,Value\,] \to IO\ \ Value$—namely the query function.

### 6.4.2 Domain-Specific Languages

The main ingredient of the POETS implementation is the implementation of the domain-specific languages. What is interesting in that respect—compared to implementations of domain-specific languages in isolation of each other—is the common core shared by the languages, in particular types and values.

In order to reuse and extend the structure of types and values in the report language and the contract language, we make use of the *compositional data types* [11] library. Compositional data types take the *data types as fixed points* [67] view on abstract syntax trees (ASTs), namely a separation of the recursive structure of ASTs from their signatures. As an example, we define the signatures of types from Section 6.2.1.1 as follows:

```
type RecordName     = String
data TypeConstant a = TBool | TInt | ···
data TypeRecord a   = TRecord RecordName
data TypeList a     = TList a
data TypeEnt a      = TEnt RecordName
```

The signature for the types of the data model is then obtained by combining the individual signatures above $TSig = TypeConstant :+: TypeRecord :+: TypeList :+: TypeEnt$, where $(:+:) :: (* \to *) \to (* \to *) \to * \to *$ is the sum of two functors. Finally, the data type for ASTs of types can be defined by tying the recursive knot $T = Term\ TSig$, where $Term :: (* \to *) \to *$ is the functor fixed point.

Recursive functions over ASTs are defined as type classes, with one instance per atomic signature. For instance, a pretty printer for types can be defined as follows:

```
class Functor f ⇒ Render f where
  render :: f String → String

instance Render TypeConstant where
  render TInt  = "Int"
  render TBool = "Bool"
  ...
```

> **instance** *Render TypeRecord* **where**
>   *render* (*TRecord r*) $= r$
>
> **instance** *Render TypeList* **where**
>   *render* (*TList* $\tau$) $=$ `"["` $+\!\!+\ \tau\ +\!\!+$ `"]"`
>
> **instance** *Render TypeEnt* **where**
>   *render* (*TEnt r*) $=$ `"<"` $+\!\!+\ r\ +\!\!+$ `">"`

and pretty printing of terms is subsequently obtained by lifting the *render* algebra to a catamorphism, that is a function of type *Render f* ⇒ *Term f* → *String*.

**Extensibility** The first benefit of the approach above is that we can extend the signature for types to fit, for example, the contract language as in Figure 6.9:

> **type** *TypeVar* = *String*
> **data** *TypeUnit a* = *TUnit*
> **data** *TypeVar a* = *TVar TypeVar*
> **data** *TypeFunction a* = *TFunction a a*

Extending the pretty printer amounts to only providing the new cases:

> **instance** *render TypeUnit* **where**
>   *render TUnit* = `"()"`
> **instance** *render TypeVar* **where**
>   *render* (*TVar* $\alpha$) $= \alpha$
> **instance** *render TypeFunction* **where**
>   *render* (*TFunction* $\tau_1\ \tau_2$) $= \tau_1\ +\!\!+$ `" -> "` $+\!\!+\ \tau_2$

A similar modular encoding is used for the language of values:

> **data** *Value a* = *VInt Int* | *VBool Bool* | *VString String* | $\cdots$

and the signature of expressions in the contract language of Figure 6.9 can be obtained by providing the extensions compared to the language of values:

> **type** *Var* = *String*
> **data** *Exp a* = *EVar Var* | *ELambda Var a* | *EApply a a* | $\cdots$

That is, *Term* (*Exp* :+: *Value*) represents the type of ASTs for expressions of the contract language. Reusing the signature for (core) values means that the values of Section 6.2.1.2, which are provided as input to the system for instance in the *registerTransaction* function, can be automatically coerced to the richer language of expressions. That is, values of type *Term Value* can be readily used as values of type *Term* (*Exp* :+: *Value*), without explicit copying or translation.

Notice the difference in the granularity of (core) value signatures and (core) type signatures: types are divided into three signatures, whereas values are in one signature. The rule of thumb we apply is to divide signatures only when a function needs the granularity. For instance, the type inference algorithm used in the report language and the contract language implements a simplification procedure [30], which reduces type constraints to *atomic* type constraints. In order to guarantee this transformation invariant statically, we hence need a signature of atomic types, namely *TypeConstant* :+: *TypeVar*, which prompts the finer granularity on types.

**Syntactic sugar** Besides enabling a common core of ASTs and functions on them, compositional data type enable AST transformations where the invariant of the transformation is witnessed by the type. Most notably, desugaring can be implemented by providing a signature for syntactic sugar:

$$\textbf{data}\ \mathit{ExpSug}\ a = \mathit{ELet}\ \mathit{Var}\ a\ a \mid \cdots$$

as well as a transformation to the core signature:

$$\begin{array}{l}\textbf{instance}\ \mathit{Desugar}\ \mathit{ExpSug}\ \textbf{where}\\ \quad \mathit{desugar}\ (\mathit{ELet}\ x\ e_1\ e_2) = \mathit{ELam}\ x\ e_2\ \text{`}\mathit{EApp}\text{`}\ e_1\\ \quad \ldots\end{array}$$

This approach yields a desugaring function of the type *Term* (*ExpSug* :+: *Exp* :+: *Value*) → *Term* (*Exp* :+: *Value*), which witnesses that the syntactic sugar has indeed been removed.

Moreover, since we define the desugaring translation in the style of a *term homomorphism* [11], we automatically get a lifted desugaring function that propagates AST annotations, such as source code positions, to the desugared term. This means, for instance, that type error messages can provide detailed source position information also for terms that originate from syntactic sugar.

## 6.5 Conclusion

We have presented an extended and generalised version of the POETS architecture [41], which we have fully implemented. We have presented domain-specific languages for specifying the data model, reports, and contracts of a POETS instance, and we have demonstrated an application of POETS in a small use case. The use case demonstrates the conciseness of our approach—Appendix E.2 contains the complete source needed for a running system—as well as the domain-orientation of our specification languages. We believe that non-programmers should be able to read and understand the data model of Appendix E.2.1, to some extent the contract specifications of Appendix E.2.3.3, and to a lesser extent the reports of Appendix E.2.2 (after all, reports describe computations).

### 6.5.1 Future Work

With our implementation and revision of POETS we have only taken the first steps towards a software system that can be used in practice. In order to properly verify our hypothesis that POETS is practically feasible, we want to conduct a larger use case in a live, industrial setting. Such use case will both serve as a means of testing the technical possibilities of POETS, that is whether we can model and implement more complex scenarios, as well as a means of testing our hypothesis that the use of domain-specific languages shortens the gap between requirements and implementation.

**Expressivity** As mentioned above, a larger and more realistic use case is needed in order to fully evaluate POETS. In particular, we are interested in investigating whether the data model, the report language, and the contract language have sufficient expressivity. For instance, a possible extension of the data model is to introduce finite maps. Such extension will, for example, simplify the reports from our $\mu$ERP use case that deal with multiple currencies. Moreover, finite maps will enable a modelling of resources that is closer in structure to that of Henglein et al. [41].

Another possible extension is to allow types as values in the report language. For instance, the *EntitiesByType* report in Appendix E.2.2.3 takes a string representation of a record type, rather than the record type itself. Hence the function cannot take subtypes into account, that is if we query the report with input A, then we only get entities of declared type A and not entities of declared subtypes of A.

**Rules** A rule engine is a part of our extended architecture (Figure 6.2), however it remains to be implemented. The purpose of the rule engine is to provide rules—written in a separate domain-specific language—that can constrain the values that are accepted by the system. For instance, a rule might specify that the items list of a Delivery transaction always be non-empty.

More interestingly, the rule engine will enable values to be *inferred* from the rules in the engine. For instance, a set of rules for calculating VAT will enable the field vatPercentage of an OrderLine to be inferred automatically in the context of a Sale record. That is, based on the information of a sale and the items that are being sold, the VAT percentage can be calculated automatically for each item type.

The interface to the rule engine will be very simple: A record value, as defined in Section 6.2.1.2, with zero or more *holes* is sent to the engine, and the engine will return either (i) an indication that the record cannot possibly fulfil the rules in the engine, or (ii) a (partial) substitution that assigns inferred values to (some of) the holes of the value as dictated by the rules. Hence when we, for example, initiate the sale of a bicycle in Section 6.3.4, then we first let the rule engine infer the VAT percentage before passing the contract meta data to the contract engine.

**Forecasts** A feature of the contract engine, or more specifically of the reduction semantics of contract instances, is the possibility to retrieve the state of a running contract at any given point in time. The state is essentially the AST of a contract clause, and it describes what is currently expected in the contract, as well as what is expected in the future.

Analysing the AST of a contract enables the possibility to do *forecasts*, for instance to calculate the expected outcome of a contract or the items needed for delivery within the next week. Forecasts are, in some sense, dual to reports. Reports derive data from transactions, that is facts about what has previously happened. Forecasts, on the other hand, look into the future, in terms of calculations over running contracts. We have currently implemented a single forecast, namely a forecast that lists the set of immediately expected transactions for a given contract. A more ambitious approach is to devise (yet another) language for writing forecasts, that is functions that operate on contract ASTs.

**Practicality** In order to make POETS useful in practice, many features are still missing. However, we see no inherent difficulties in adding them to POETS compared to traditional ERP architectures. To mention a few: (i) security, that is authorisation, users, roles, etc.; (ii) module systems for the report language and contract language, that is better support for code reuse; and (iii) check-pointing of a running system, that is a dump of the memory of a running system, so the event log does not have to be replayed from scratch when the system is restarted.

**Acknowledgements** We are grateful to Fritz Henglein for many fruitful discussions and for convincing us of the *POETS approach* in the first place. Morten Ib Nielsen and Mikkel Jønsson Thomsen both contributed to our implementation and design of POETS, for which we are thankful. Lastly, we thank the participants of the DIKU course “POETS Summer of Code” for valuable input.

# Appendices

# Appendix A

# Appendices for Chapter 1

## A.1 A Brief Introduction to Standard Deontic Logic

The following is a brief introduction to standard deontic logic (SDL), based on the presentations of Prakken and Sergot [92], McNamara [65], and Woleński [120]. SDL is a modal logic for expressing obligatory and permissible statements, making it appealing in the context of contract modelling. SDL is classical (as opposed to intuitionistic) propositional logic [48] extended with two modalities: O (obligation) and P (permission). The formulae $\mathrm{O}\phi$ and $\mathrm{P}\phi$ should be read "it is obligatory that $\phi$" and "it is permitted that $\phi$", respectively. Let *order* and *deliver* be propositional atoms representing the ordering and delivery of goods, respectively. We can then encode the contract-like statement "Buyer is permitted to order from Seller, and by doing so, Seller is obliged to deliver the goods to Buyer" as:

$$(\mathrm{P}\mathit{order}) \wedge (\mathit{order} \rightarrow \mathrm{O}\mathit{deliver}).$$

The grammar for SDL is as follows:

$\phi ::= \bot \mid p \mid \phi \rightarrow \phi \mid \mathrm{O}\phi,$

with the usual classical abbreviations:

$$\neg\phi \equiv \phi \rightarrow \bot \qquad \phi \vee \psi \equiv \neg\phi \rightarrow \psi \qquad \phi \wedge \psi \equiv \neg(\neg\phi \vee \neg\psi).$$

Moreover, the permission modality can be expressed as $\mathrm{P}\phi \equiv \neg\mathrm{O}\neg\phi$, and usually a prohibition modality (F) is also considered $\mathrm{F}\phi \equiv \mathrm{O}\neg\phi$. Note that the last abbreviation relies on a closed-world assumption, that is by not doing $\phi$ one does $\neg\phi$.

The semantics of SDL is given in terms of Kripke models. A Kripke model $\mathcal{M}$ for SDL is a triple:

$$\mathcal{M} = \langle W, d, V\rangle,$$

where $W$ is a non-empty set of *worlds*, $d \subseteq W \times W$ is the *deontic accessibility relation*, and $V \subseteq \mathit{Prop} \times W$ is the *valuation function* for propositional symbols, that is $(p, w) \in V$ means that $p$ holds in world $w$.

The intuition of the deontic accessibility relation is that it relates the possible worlds (or, state of affairs). Whenever $(w, w') \in d$ then $w'$ *could* have been the actual state of affairs rather than $w$. It is important to realise that $(w, w') \in d$ does not mean that $w'$ comes after $w$ temporarily—$w$ and $w'$ exist at the same point in time. There are, in other words, no temporal aspects in SDL. Perhaps surprisingly then, $d$ is not required to be symmetric nor reflexive, instead $d$ is required to be *serial*:

$$\forall w_1 \in W. \exists w_2 \in W. (w_1, w_2) \in d.$$

That is, all worlds have an alternative. Given the interpretation of the deontic relation $d$, the obligation modality is the usual box modality of modal logics, that is $\mathrm{O}\phi$ holds in a world $w$ exactly when $\phi$ holds in all possible worlds related to $w$.

A formula $\phi$ is said to be *satisfied* in the model $\mathcal{M} = \langle W, d, V \rangle$, written $\mathcal{M} \models \phi$, whenever $\mathcal{M} \models_w \phi$ holds for all $w \in W$. The latter relation is defined by structural induction on $\phi$:

$$\begin{array}{lll} \mathcal{M} \not\models_w \bot, & & \\ \mathcal{M} \models_w p & \text{iff} & (p, w) \in V, \\ \mathcal{M} \models_w \phi \to \psi & \text{iff} & \mathcal{M} \models_w \psi \text{ whenever } \mathcal{M} \models_w \phi, \\ \mathcal{M} \models_w \mathrm{O}\phi & \text{iff} & \mathcal{M} \models_{w'} \phi \text{ whenever } (w, w') \in d. \end{array}$$

A formula $\phi$ is said to be valid, written $\models \phi$, whenever $\mathcal{M} \models \phi$ for all models $\mathcal{M}$. With these definitions, it can be shown that SDL is a *KD*-style modal logic [48], that is the following holds:

$$\models \mathrm{O}(\phi \to \psi) \to \mathrm{O}\phi \to \mathrm{O}\psi, \tag{$K$}$$

$$\models \mathrm{O}\phi \to \neg\mathrm{O}\neg\phi. \tag{$D$}$$

The $K$-property is present in all normal modal logics, and it says that if a material conditional is obligatory, and its antecedent is obligatory, then so is its consequent. The $D$-property ($D$ for deontic) says that there can be no conflicts, that is it impossible for both $\phi$ and $\neg\phi$ to be obligatory. Since $\mathrm{P}\phi \equiv \neg\mathrm{O}\neg\phi$, an alternative reading of the $D$-property is that whenever $\phi$ is obligatory then $\phi$ is permitted, which we would expect. The validity of $D$ is easily seen to rely on $d$ being serial.

Besides $K$ and $D$, a series of valid statements and derived rules can be shown, some important of which are:

$$\models \mathrm{O}(\phi \wedge \psi) \to \mathrm{O}\phi \wedge \mathrm{O}\psi, \tag{A.1}$$

$$\models \mathrm{O}\phi \wedge \mathrm{O}\psi \to \mathrm{O}(\phi \wedge \psi) \tag{A.2}$$

$$\text{if } \models \phi \text{ then } \models \mathrm{O}\phi. \tag{A.3}$$

A.1 and A.2 state that O distributes over conjunction, and A.3 states that if a formula $\phi$ is valid, then it is also valid that $\phi$ must hold. The Kripke-style semantics has been constructed exactly so as to make the properties K, D, A.1, A.2 and A.3 hold, as they state expected properties of the corresponding deontic modalities. We note also that the converse of A.3 does not hold:

$$\text{if } \models \mathrm{O}\phi \text{ then } \models \phi. \tag{UNSOUND}$$

That is even though $\phi$ *should* hold, we cannot be sure that $\phi$ actually holds.

# Appendix B

# Appendices for Chapter 2

## B.1 Additional Proof Details

**Proof of Theorem 2.3.4** Assume that $s$ is well-formed with parties $P$, that is $\vdash s : \text{Contract}\langle P\rangle$ and the unfolding relation on the template names of $s$ is acyclic. Assume furthermore that $s \xrightarrow{\epsilon} s'$, where $s = \textbf{letrec}\ D\ \textbf{in}\ c\ \textbf{starting}\ \tau$. Then $s' = \textbf{letrec}\ D\ \textbf{in}\ c'\ \textbf{starting}\ \text{ts}(\epsilon)$, for some clause $c'$, where $D, \tau \vdash c \xrightarrow{\epsilon} c'$. We need to show that $s'$ is well-formed with parties $P' \subseteq P$, which amounts to showing that $\vdash s' : \text{Contract}\langle P'\rangle$ as the templates of $s$ and $s'$ are identical. Since $s$ is well-typed, we have that $\Delta \vdash D$ and $\Delta, \emptyset, \emptyset \vdash c : \text{Clause}\langle P\rangle$, so it suffices to show that $\Delta, \emptyset, \emptyset \vdash c' : \text{Clause}\langle P'\rangle$ for some $P' \subseteq P$, again since the templates do not change. We hence need to show:

> If $D, \tau \vdash c \xrightarrow{\epsilon} c'$ and $\Delta, \emptyset, \emptyset \vdash c : \text{Clause}\langle P\rangle$ then $\Delta, \emptyset, \emptyset \vdash c' : \text{Clause}\langle P'\rangle$ for some $P' \subseteq P$.

The proof is by induction on the derivation of $D, \tau \vdash c \xrightarrow{\epsilon} c'$. We do a case split on the last derivation rule:

- The last rule is:

  $$\frac{}{D, \tau \vdash \textbf{fulfilment} \xrightarrow{\epsilon} \textbf{fulfilment}}$$

  This case is trivial. (Note that $P = P' = \emptyset$.)

- The last rule is:

  $$\frac{e[\vec{v}/\vec{x}] \Downarrow \textbf{true} \qquad d \Downarrow^{\tau} (\tau_1, \tau_2) \qquad \tau_1 \leq \tau' \leq \tau_2}{D, \tau \vdash \langle p\rangle\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c_1 \xrightarrow{(\tau', k(\vec{v}))} c_1[\vec{v}/\vec{x}, \tau_2 - \tau'/z]}$$

  The typing derivation for $c$ has the form:

  $$\frac{\begin{array}{c} \Gamma' = [\vec{x} \mapsto \text{ar}(k)] \\ \Gamma_2 = \Gamma'[z \mapsto \text{Int}] \end{array} \qquad \begin{array}{c} \emptyset \vdash p : \{p\} \\ \Gamma' \vdash e : \text{Bool} \\ \emptyset \vdash d : \text{Deadline} \end{array} \qquad \overbrace{\Delta, \emptyset, \Gamma_2 \vdash c_1 : \text{Clause}\langle P_2\rangle}^{\text{(a)}}}{\Delta, \emptyset, \emptyset \vdash \langle p\rangle\ k(\vec{x})\ \textbf{where}\ e\ \textbf{due}\ d\ \textbf{remaining}\ z\ \textbf{then}\ c_1 : \text{Clause}\langle \{p\} \cup P_2\rangle}$$

  It then follows from (a) and Lemma 2.3.3 that $\Delta, \emptyset, \emptyset \vdash c_1[\vec{v}/\vec{x}, \tau_2 - \tau'/z] : \text{Clause}\langle P_2\rangle$, as required. (Note also that $P_2 \subseteq \{p\} \cup P_2$.)

- The last rule is:

$$\frac{\begin{array}{l} d \Downarrow^{\tau} (\tau_1, \tau_2) \\ \tau' \leq \tau_2 \qquad \tau' < \tau_1 \vee k' \neq k \vee e[\vec{v}/\vec{x}] \Downarrow \textbf{false} \qquad d' = \textbf{after } \tau_1 - \tau' \textbf{ within } \tau_2 - \tau_1 \end{array}}{\begin{array}{ll} D, \tau \vdash & \langle p \rangle\ k(\vec{x}) \textbf{ where } e \textbf{ due } d \textbf{ remaining } z \textbf{ then } c_1 \xrightarrow{(\tau', k'(\vec{v}))} \\ & \langle p \rangle\ k(\vec{x}) \textbf{ where } e \textbf{ due } d' \textbf{ remaining } z \textbf{ then } c_1 \end{array}}$$

We only need to show that $\emptyset \vdash d'$ : Deadline, which follows immediately.

- The last rule is:

$$\frac{e[\vec{v}/\vec{x}] \Downarrow \textbf{true} \qquad d \Downarrow^{\tau} (\tau_1, \tau_2) \qquad \tau_1 \leq \tau' \leq \tau_2}{D, \tau \vdash \textbf{if } k(\vec{x}) \textbf{ where } e \textbf{ due } d \textbf{ remaining } z \textbf{ then } c_1 \textbf{ else } c_2 \xrightarrow{(\tau', k(\vec{v}))} c_1[\vec{v}/\vec{x}, \tau_2 - \tau'/z]}$$

This case is similar to the second case.

- The last rule is:

$$\frac{d \Downarrow^{\tau} (\tau_1, \tau_2) \qquad \tau' > \tau_2 \qquad \overbrace{D, \max(\tau, \tau_2) \vdash c_2 \xrightarrow{(\tau', k'(\vec{v}))} c'}^{(a)}}{D, \tau \vdash \textbf{if } k(\vec{x}) \textbf{ where } e \textbf{ due } d \textbf{ remaining } z \textbf{ then } c_1 \textbf{ else } c_2 \xrightarrow{(\tau', k'(\vec{v}))} c'}$$

The typing derivation for $c$ has the form:

$$\frac{\begin{array}{ccc} \Gamma' = [\vec{x} \mapsto \text{ar}(k)] & \Gamma' \vdash e : \text{Bool} & \overbrace{\Delta, \emptyset, \emptyset \vdash c_2 : \text{Clause}\langle P_2 \rangle}^{(b)} \\ \Gamma_1 = \Gamma'[z \mapsto \text{Int}] & \emptyset \vdash d : \text{Deadline} & \Delta, \emptyset, \Gamma_1 \vdash c_1 : \text{Clause}\langle P_1 \rangle \end{array}}{\Delta, \emptyset, \emptyset \vdash \textbf{if } k(\vec{x}) \textbf{ where } e \textbf{ due } d \textbf{ remaining } z \textbf{ then } c_1 \textbf{ else } c_2 : \text{Clause}\langle P_1 \cup P_2 \rangle}$$

So the result follows from the induction hypothesis applied to (a) and (b), and from the fact that $P_2 \subseteq P_1 \cup P_2$.

- The last rule is:

$$\frac{\begin{array}{l} d \Downarrow^{\tau} (\tau_1, \tau_2) \\ \tau' \leq \tau_2 \qquad \tau' < \tau_1 \vee k' \neq k \vee e[\vec{v}/\vec{x}] \Downarrow \textbf{false} \qquad d' = \textbf{after } \tau_1 - \tau' \textbf{ within } \tau_2 - \tau_1 \end{array}}{\begin{array}{ll} D, \tau \vdash & \textbf{if } k(\vec{x}) \textbf{ where } e \textbf{ due } d \textbf{ remaining } z \textbf{ then } c_1 \textbf{ else } c_2 \xrightarrow{(\tau', k'(\vec{v}))} \\ & \textbf{if } k(\vec{x}) \textbf{ where } e \textbf{ due } d' \textbf{ remaining } z \textbf{ then } c_1 \textbf{ else } c_2 \end{array}}$$

This case is similar to the third case.

- The last rule is:

$$\frac{\overbrace{D, \tau \vdash c_1 \xrightarrow{\epsilon} c_1'}^{(a)} \qquad \overbrace{D, \tau \vdash c_2 \xrightarrow{\epsilon} c_2'}^{(b)}}{D, \tau \vdash c_1 \textbf{ and } c_2 \xrightarrow{\epsilon} c_1' \textbf{ and } c_2'}$$

The typing derivation for $c$ has the form:

$$\frac{\overbrace{\Delta, \emptyset, \emptyset \vdash c_1 : \text{Clause}\langle P_1 \rangle}^{(c)} \qquad \overbrace{\Delta, \emptyset, \emptyset \vdash c_2 : \text{Clause}\langle P_2 \rangle}^{(d)}}{\Delta, \emptyset, \emptyset \vdash c_1 \textbf{ and } c_2 : \text{Clause}\langle P_1 \cup P_2 \rangle}$$

So it follows from the induction hypothesis applied to (a) and (c) on one hand, and (b) and (d) on the other hand, that $\Delta, \emptyset, \emptyset \vdash c_1'$ : Clause$\langle P_1' \rangle$ and $\Delta, \emptyset, \emptyset \vdash c_2'$ : Clause$\langle P_2' \rangle$ with $P_1' \subseteq P_1$ and $P_2' \subseteq P_2$. Hence it follows that $\Delta, \emptyset, \emptyset \vdash c_1'$ **and** $c_2'$ : Clause$\langle P_1' \cup P_2' \rangle$ as required.

- The last rule is:

$$\frac{D, \tau \vdash c_1 \xrightarrow{\epsilon} c_1' \qquad D, \tau \vdash c_2 \xrightarrow{\epsilon} c_2'}{D, \tau \vdash c_1 \textbf{ or } c_2 \xrightarrow{\epsilon} c_1' \textbf{ or } \mathbf{c}_2'}$$

  This case is similar to the previous case.

- The last rule is:

$$\frac{e \Downarrow \textbf{true} \qquad \overbrace{D, \tau \vdash c_1 \xrightarrow{\epsilon} c_1'}^{\text{(a)}}}{D, \tau \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \xrightarrow{\epsilon} c_1'}$$

  The typing derivation for $c$ has the form:

$$\frac{\emptyset \vdash e : \text{Bool} \qquad \overbrace{\Delta, \emptyset, \emptyset \vdash c_1 : \text{Clause}\langle P_1 \rangle}^{\text{(b)}} \qquad \Delta, \emptyset, \emptyset \vdash c_2 : \text{Clause}\langle P_2 \rangle}{\Delta, \emptyset, \emptyset \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 : \text{Clause}\langle P_1 \cup P_2 \rangle}$$

  So it follows from the induction hypothesis applied to (a) and (b) that $\Delta, \emptyset, \emptyset \vdash c_1' : \text{Clause}\langle P_1' \rangle$ with $P_1' \subseteq P_1 \subseteq P_1 \cup P_2$ as required.

- The last rule is:

$$\frac{e \Downarrow \textbf{false} \qquad D, \tau \vdash c_2 \xrightarrow{\epsilon} c_2'}{D, \tau \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \xrightarrow{\epsilon} c_2'}$$

  This case is similar to the previous case.

- The last rule is:

$$\frac{\vec{e} \Downarrow \vec{v} \qquad (f(\vec{x})\langle\vec{y}\rangle = c') \in D \qquad \overbrace{D, \tau \vdash c'[\vec{v}/\vec{x}, \vec{p}/\vec{y}]\langle\vec{p}/\vec{y}\rangle \xrightarrow{\epsilon} c''}^{\text{(a)}}}{D, \tau \vdash f(\vec{e})\langle\vec{p}\rangle \xrightarrow{\epsilon} c''}$$

  The typing derivation for $c$ has the form:

$$\frac{\Delta(f) = (\langle t_1, \ldots, t_m \rangle, n) \qquad \overbrace{\forall i \in \{1, \ldots, m\}. \emptyset \vdash e_i : t_i}^{\text{(b)}} \qquad \forall i \in \{1, \ldots, n\}. \emptyset \vdash p_i : \{p_i\}}{\Delta, \emptyset, \emptyset \vdash f(e_1, \ldots, e_m)\langle p_1, \ldots, p_n \rangle : \text{Clause}\langle\{p_1, \ldots, p_n\}\rangle}$$

  and it follows from $\Delta \vdash D$ that $\Delta, \vec{y}, [\vec{x} \mapsto \vec{t}, \vec{y} \mapsto \overrightarrow{\text{Party}}] \vdash c' : \text{Clause}\langle\emptyset\rangle$. It then follows from Lemma 2.3.2 and (b) that $v_i \in [\![t_i]\!]$ for $i = 1, \ldots, m$, and hence via Lemma 2.3.3 that $\Delta, \emptyset, \emptyset \vdash c'[\vec{v}/\vec{x}, \vec{p}/\vec{y}]\langle\vec{p}/\vec{y}\rangle : \text{Clause}\langle\{p_1, \ldots, p_n\}\rangle$. But then the result follows from the induction hypothesis applied to (a). □

**Lemma B.1.1.** *Assume that* Sub$(c) = \{c_1, \ldots, c_n\}$*, for clauses* $c, c_1, \ldots, c_n$*. Then* Sub$(c[\theta]) = \{c_1[\theta], \ldots, c_2[\theta]\}$ *for all substitutions* $\theta$*.*

*Proof.* The proof follows by straightforward structural induction on $c$. □

**Lemma B.1.2.** *Let $c$ be a well-typed clause* $\Delta, \emptyset, \emptyset \vdash c : \text{Clause}\langle P \rangle$*. Then* $\Delta, \emptyset, \emptyset \vdash c' : \text{Clause}\langle P' \rangle$ *for all* $c' \in \text{Sub}(c)$ *with* $P' \subseteq P$*.*

*Proof.* The proof follows by straightforward structural induction on $c$ (or, equivalently by induction on the typing derivation of $\Delta, \emptyset, \emptyset \vdash c : \text{Clause}\langle P \rangle$). □

**Proof of Theorem 2.3.5** We start with a needed definition. We say that a substitution $\theta$ is *type-preserving* with regard to a variable environment $\Gamma$, if $\mathrm{dom}(\theta) = \mathrm{dom}(\Gamma)$ and $\theta(x) \in [\![\Gamma(x)]\!]$, for any $x \in \mathrm{dom}(\theta)$.

Let $s =$ **letrec** $D$ **in** $c_0$ **starting** $\tau_0$ and assume that $s$ is well-formed with parties $P$. That is, $\Rightarrow_D$ is an acyclic relation, $\Delta \vdash D$, and $\Delta, \emptyset, \emptyset \vdash c_0 : \mathrm{Clause}\langle P \rangle$ for some template environment $\Delta$.

Assume $D = \{(f(\vec{x})\langle\vec{y}\rangle = c_f) \mid f \in \mathcal{F}_D\}$ and let $\mathcal{C}_D = \{c_f \mid f \in \mathcal{F}_D\}$. We associate with $c_0$ a new template name $f_0 \notin \mathcal{F}_D$, and let $\mathcal{F}'_D = \mathcal{F}_D \cup \{f_0\}$ and $c_{f_0} = c_0$. We extend the relation $\Rightarrow_D$ from $\mathcal{F}_D$ to $\mathcal{F}'_D$ as expected: $f_0 \Rightarrow_D g$ if and only if there is a subclause $g(\vec{e}_1)\langle\vec{e}_2\rangle \in \mathrm{Sub}(c_0)$. Note that by definition there is no $g \in \mathcal{F}'_D$ such that $g \Rightarrow_D f_0$. Hence the extended relation $\Rightarrow_D$ is still acyclic. And, as $\Rightarrow_D$ is finite, $\Rightarrow_D$ is well-founded.

We let $P_f = \emptyset$ for any $f \in \mathcal{F}_D$ and $P_{f_0} = P$. As $\Delta \vdash D$, there are environments $\Lambda_f$, $\Gamma_f$ such that $\Delta, \Lambda_f, \Gamma_f \vdash c_f : \mathrm{Clause}\langle P_f \rangle$ for all $f \in \mathcal{F}'_D$, with $\Lambda_{f_0} = \emptyset$ and $\Gamma_{f_0} = \emptyset$. We will show the following claim:

**Claim:** For any $f \in \mathcal{F}'_D$, for any clause $c = c'[\theta]\langle\theta'\rangle$, where $c' \in \mathrm{Sub}(c_f)$, $\theta'$ is a party substitution with $\mathrm{dom}(\theta') = \Lambda_f$, and $\theta$ is a type-preserving substitution with regard to $\Gamma_f$, the following statement holds:

> For any event $\epsilon$ with $\mathrm{ts}(\epsilon) \geq \tau_0$ there is a unique residue $\mathbf{c}$ such that $D, \tau_0 \vdash c \xrightarrow{\epsilon} \mathbf{c}$. Moreover, if $\mathbf{c} = (\tau, B)$, then $\tau_0 \leq \tau \leq \mathrm{ts}(\epsilon)$ and $B \subseteq P_f \cup \mathrm{rng}(\theta')$.

Note that the result of the theorem then follows from the claim applied to $f_0$, the clause $c_0$, and empty (party) substitutions $\theta$ and $\theta'$.

We proceed by a nested inductive argument: an (outer) well-founded induction on $f$ and an (inner) structural induction on the clause $c$.

The following observation will be used in the proof: since $\Delta, \Lambda_f, \Gamma_f \vdash c_f : \mathrm{Clause}\langle P_f \rangle$ it follows from Lemma 2.3.3 that $\Delta, \emptyset, \emptyset \vdash c_f[\theta]\langle\theta'\rangle : \mathrm{Clause}\langle P_f \cup \mathrm{rng}(\theta')\rangle$. Hence from Lemmas B.1.1 and B.1.2 it follows that $\Delta, \emptyset, \emptyset \vdash c : \mathrm{Clause}\langle P' \rangle$ with $P' \subseteq P_f \cup \mathrm{rng}(\theta')$, so we may assume in each case that $c$ is well-typed and closed.

- $c =$ **fulfilment**. (This is a base case for the inner induction.) The claim clearly holds in this case.
- $c = \langle p \rangle\ k(\vec{x})$ **where** $e$ **due** $d$ **remaining** $z$ **then** $c_1$. Suppose $\epsilon = (\tau', k'(\vec{v}))$ for some $\tau' \geq \tau_0$ and some action $k'(\vec{v})$. As $c$ is well-typed, it follows from Lemma 2.3.2 that there is a unique Boolean value $b$ and timestamps $\tau_1, \tau_2$ such that $e[\vec{v}/\vec{x}] \Downarrow b$ and $d \Downarrow^{\tau_0} (\tau_1, \tau_2)$. We distinguish three cases:
  - $k = k'$, $b =$ **true**, and $\tau_1 \leq \tau' \leq \tau_2$. Then take $\mathbf{c} = c[\vec{v}/\vec{x}, \tau_2 - \tau'/z]$.
  - $\tau' > \tau_2$. Take $\mathbf{c} = (\max(\tau_0, \tau_2), \{p\})$. Clearly, $\tau_0 \leq \max(\tau_0, \tau_2) \leq \tau'$. And, by the observation above, we know that $\Delta, \emptyset, \emptyset \vdash c : \mathrm{Clause}\langle P' \rangle$, where $P' \subseteq P_f \cup \mathrm{rng}(\theta')$, hence $p \in P_f \cup \mathrm{rng}(\theta')$.
  - $\tau' \leq \tau_2$ and also $k \neq k'$, $b =$ **false**, or $\tau' < \tau_1$. Then take $\mathbf{c} = \langle p \rangle\ k(\vec{x})$ **where** $e$ **due** $d'$ **remaining** $z$ **then** $c$ with $d' =$ **after** $\tau_1 - \tau'$ **within** $\tau_2 - \tau_1$.

  In all three cases the residue $\mathbf{c}$ satisfies the claim.

- $c =$ **if** $k(\vec{x})$ **where** $e$ **due** $d$ **remaining** $z$ **then** $c_1$ **else** $c_2$. Suppose that $\epsilon = (\tau', k'(\vec{v}))$ for some $\tau' \geq \tau_0$ and some action $k'(\vec{v})$. As $c$ is well-typed, it follows from Lemma 2.3.2 that there is a unique Boolean value $b$ and timestamps $\tau_1, \tau_2$ such that $e[\vec{v}/\vec{x}] \Downarrow b$ and $d \Downarrow^{\tau_0} (\tau_1, \tau_2)$. As for obligations, we distinguish the same three cases, only the following one having a different treatment:

  – $\tau' > \tau_2$. By the definition of immediate subclauses, we have that $c_2 \in \mathrm{Sub}(c_f)$, hence by the inner induction hypothesis on $c_2$ there is a unique residue $\mathbf{c}$ such that $D, \max(\tau_0, \tau_2) \vdash c_2 \xrightarrow{\epsilon} \mathbf{c}$, and if $\mathbf{c} = (\tau, B)$ then $\max(\tau_0, \tau_2) \leq \tau \leq \mathrm{ts}(\epsilon)$ and $B \subseteq P$. Clearly, the residue $\mathbf{c}$ satisfies the claim.

- $c = c_1$ **and** $c_2$. By the definition of immediate subclauses, we have that $c_1, c_2 \in \mathrm{Sub}(c_f)$, hence by the inner induction hypothesis on $c_1$ and $c_2$ we obtain that there are unique residues $\mathbf{c}_1$ and $\mathbf{c}_2$ such that $D, \tau_0 \vdash c_1 \xrightarrow{\epsilon} \mathbf{c}_1$ and $D, \tau_0 \vdash c_2 \xrightarrow{\epsilon} \mathbf{c}_2$. Moreover, if $\mathbf{c}_1 = (\tau_1, B_1)$ then $\tau_0 \leq \tau_1 \leq \mathrm{ts}(\epsilon)$ and $B_1 \subseteq P$, and if $\mathbf{c}_2 = (\tau_2, B_2)$ then $\tau_0 \leq \tau_2 \leq \mathrm{ts}(\epsilon)$ and $B_2 \subseteq P$.

  Let $\mathbf{c} = \mathbf{c}_1 \owedge \mathbf{c}_2$. If $\mathbf{c}_1 = (\tau_1, B_1)$ and $\mathbf{c}_2 = (\tau_2, B_2)$, then it follows from the definition of verdict conjunction that $\tau_0 \leq \tau \leq \mathrm{ts}(\epsilon)$ and $B \subseteq P$, where $\mathbf{c} = (\tau, B) = (\tau_1, B_1) \wedge (\tau_2, B_2)$. In the other cases (that is $\mathbf{c}_1$ or $\mathbf{c}_2$ or both being clauses) the residue $\mathbf{c}$ clearly satisfies the claim.

- $c = c_1$ **or** $c_2$. This case is similar to the previous one, but in the case where $D, \tau_0 \vdash c_1 \xrightarrow{\epsilon} (\tau_1, B_1)$ and $D, \tau_0 \vdash c_2 \xrightarrow{\epsilon} (\tau_2, B_2)$, we utilise the fact that $s$ is well-formed to conclude that $B_1 = B_2 = \{p\}$, for some $p$ (due to the typing rule for clause disjunctions), which guarantees that the verdict disjunction $(\tau_1, B_1) \vee (\tau_2, B_2)$ is well-defined.

- $c =$ **if** $e$ **then** $c_1$ **else** $c_2$. As $c$ is well-typed, it follows from Lemma 2.3.2 that there is a unique Boolean value $b$ such that $e \Downarrow b$. By the definition of immediate subclauses, we have that $c_1, c_2 \in \mathrm{Sub}(c_f)$, hence by the inner induction hypothesis on $c_1$ if $b =$ **true** and on $c_2$ otherwise, the claim follows directly.

- $c = g(\vec{e})\langle\vec{p}\rangle$. As $c$ is well-typed, it follows from Lemma 2.3.2 that there are unique values $\vec{v}$ such that $\vec{e} \Downarrow \vec{v}$. Moreover, by hypothesis the clause $c$ is the instantiation of an immediate subclause $g(\vec{e_1})\langle\vec{e_2}\rangle$ of $c_f$. By the definition of $\Rightarrow_D$, we have that $f \Rightarrow_D g$. This, together with $[\vec{v}/\vec{x}, \vec{p}/\vec{y}]$ being a type-preserving substitution with regard to $\Gamma_g$ (Lemma 2.3.2) and $\langle\vec{p}/\vec{y}\rangle$ being a party substitution, allows us to apply the outer induction hypothesis on $c_g[\vec{v}/\vec{x}, \vec{p}/\vec{y}]\langle\vec{p}/\vec{y}\rangle$. The claim then follows directly. □

**Lemma B.1.3.** *Let $s$ be a well-formed specification. Then there exists a unique verdict $\nu$ such that $\vdash s \downarrow \nu$. Moreover, for a breach $(\tau, B)$, we have $\vdash s \downarrow (\tau, B)$ if and only if $s \xrightarrow{\epsilon} (\tau, B)$, for all events $\epsilon$ with $\mathrm{ts}(\epsilon) > \tau$.*

*Proof.* Existence follows by a nested inductive argument similar to, but much simpler than the proof of Theorem 2.3.5. Uniqueness follows by straightforward structural induction on $c$, where $s =$ **letrec** $D$ **in** $c$ **starting** $\tau$. The left to right implication of the second part of the lemma follows by induction on the derivation of $\vdash s \downarrow (\tau, B)$, while the other implication follows by induction on the derivation of $s \xrightarrow{\epsilon} (\tau, B)$. □

**Proof of Theorem 2.3.6** Let $s = \textbf{letrec } D \textbf{ in } c \textbf{ starting } \tau_0$ be a well-formed specification with parties $P$. We then need to show that $[\![s]\!]$ is a contract between $P$ starting at time $\tau_0$. That is, we need to show that $[\![s]\!]$ is a function from $\mathsf{Tr}^{\tau_0}$ to $\mathsf{V}$, and that it satisfies conditions (2.1) and (2.2) of Definition 2.2.1.

We first prove by induction on the length of the *finite* trace $\sigma$ that: $[\![s]\!](\sigma)$ is well-defined, that is it exists and it is unique, and if $[\![s]\!](\sigma) = (\tau, B)$ then $B \subseteq P$, $[\![s]\!](\sigma_\tau) = (\tau, B)$, and $\tau \geq \tau_0$.

*Base case:* $\sigma = \langle\rangle$. In this case it follows from Lemma B.1.3 that there is a unique verdict $\nu$ such that $\vdash s \downarrow \nu$, and hence $[\![s]\!](\sigma) = \nu$. So assume now that $[\![s]\!](\sigma) = (\tau, B)$. Then since $\sigma_\tau = \sigma$ we also have that $[\![s]\!](\sigma_\tau) = (\tau, B)$. Lastly, it follows from Lemma B.1.3 that $s \xrightarrow{\epsilon} (\tau, B)$, for any event $\epsilon$ with $\mathrm{ts}(\epsilon) > \max(\tau, \tau_0)$, and hence from Theorem 2.3.5 we have that $B \subseteq P$ and $\tau \geq \tau_0$ as required.

*Inductive case:* $\sigma = \epsilon\sigma'$. As $s$ is well-formed and $\mathrm{ts}(\epsilon) \geq \tau_0$, it follows from the progress property (Theorem 2.3.5) that there is a unique residue $\mathbf{s}$ such that $s \xrightarrow{\epsilon} \mathbf{s}$.

- If $\mathbf{s} = (\tau, B)$ then, also from Theorem 2.3.5, we have that $B \subseteq P$ and $\tau_0 \leq \tau \leq \mathrm{ts}(\epsilon)$. Now, if $\mathrm{ts}(\epsilon) = \tau$ then $\sigma_\tau = \epsilon\sigma'_\tau$ so it follows immediately that $[\![s]\!](\sigma_\tau) = (\tau, B)$. So assume that $\mathrm{ts}(\epsilon) > \tau$. It then follows from Lemma B.1.3 that $\vdash s \downarrow (\tau, B)$ and hence $[\![s]\!](\sigma_\tau) = (\tau, B)$ as required.
- If $\mathbf{s} = s'$ then, by the type-preservation property (Theorem 2.3.4), $s'$ is also well-formed with parties $P' \subseteq P$ and $s'$ has starting time $\mathrm{ts}(\epsilon)$. We have that $[\![s]\!](\sigma) = [\![s']\!](\sigma')$, so it then follows from the induction hypothesis that $[\![s']\!](\sigma')$ is well-defined and if $[\![s']\!](\sigma') = (\tau, B)$ then $B \subseteq P' \subseteq P$, $[\![s']\!](\sigma'_\tau) = (\tau, B)$, and $\tau_0 \leq \mathrm{ts}(\epsilon) \leq \tau$.

  Now if $[\![s]\!](\sigma) = (\tau, B)$ then $[\![s]\!](\epsilon\sigma') = [\![s']\!](\sigma') = (\tau, B)$ and hence by the above $[\![s']\!](\sigma'_\tau) = (\tau, B)$ with $\tau \geq \mathrm{ts}(\epsilon)$. But then $\sigma_\tau = \epsilon\sigma'_\tau$, and hence by definition $[\![s]\!](\sigma_\tau) = [\![s']\!](\sigma'_\tau) = (\tau, B)$ as required.

We now show that if $[\![s]\!](\sigma) = (\tau, B)$ for some finite trace $\sigma$ and breach $(\tau, B)$, then $[\![s]\!](\sigma') = (\tau, B)$, for any finite trace $\sigma'$ with $\sigma'_\tau = \sigma_\tau$. Let $\sigma'$ be a trace with $\sigma'_\tau = \sigma_\tau$. As shown above, we have $[\![s]\!](\sigma_\tau) = (\tau, B)$. The proof is by induction on the length of $\sigma_\tau$:

*Base case:* $\sigma_\tau = \langle\rangle$. Now $\sigma'_\tau = \langle\rangle$, so either $\sigma' = \langle\rangle$ or $\sigma' = \epsilon\sigma''$, for some $\epsilon$ and $\sigma''$ with $\mathrm{ts}(\epsilon) > \tau$. In the first case the result follows immediately, and in the second case we have that $\vdash s \downarrow (\tau, B)$, hence by Lemma B.1.3 we have that $s \xrightarrow{\epsilon} (\tau, B)$ from which the result follows.

*Inductive case:* $\sigma_\tau = \epsilon\sigma''$. Now $\sigma' = \epsilon\sigma'''$ with $\sigma'' = \sigma'''_\tau$, and $[\![s]\!](\sigma_\tau) = (\tau, B)$ can happen in two ways:

- $s \xrightarrow{\epsilon} (\tau, B)$: In this case we have by definition that $[\![s]\!](\sigma') = [\![s]\!](\epsilon\sigma''') = (\tau, B)$.
- $s \xrightarrow{\epsilon} s'$ and $[\![s']\!](\sigma'') = (\tau, B)$: In this case we have by definition that $[\![s]\!](\sigma') = [\![s']\!](\sigma''')$, and hence the result follows from the induction hypothesis as $\sigma'' = \sigma'''_\tau$.

We have now proved that the restriction of $[\![s]\!]$ to finite traces satisfies the hypotheses of Lemma 2.2.5. We can thus apply the lemma and obtain that $[\![s]\!]$ is a contract as per Definition 2.2.1. $\square$

# Appendix C

# Appendices for Chapter 3

## C.1 Additional Proof Details

### Proof of Lemma 3.2.6

*Proof.* The proof consists of two parts. First, we show that the composition exists (i). Second, we show that the composition is well-defined (ii), that is that the composition of two processes is itself a process.

(i) Let $l \in \mathcal{L}_{C_I}$ be given. We need to show that there exists some $N \in \mathbb{N}$ such that $\mathcal{I}^1_N = \mathcal{I}^1_{N+1}$ and $\mathcal{I}^2_N = \mathcal{I}^2_{N+1}$, compare Definition 3.2.4. We show by induction on $n$ that $\mathcal{I}^i_{n|n} = \mathcal{I}^i_{n+1|n}$ for $i = 1, 2$.

$n = 0$: $\mathcal{I}^i_{0|0} = \mathcal{I}^i_{1|0}$ holds trivially for $i = 1, 2$.

$n > 0$: It follows from the induction hypothesis that $\mathcal{I}^2_{n-1|n-1} = \mathcal{I}^2_{n|n-1}$ and thus $\left((\mathcal{I}^2_{n-1} \cup l)_{|C^1_I}\right)_{|n-1} = \left((\mathcal{I}^2_n \cup l)_{|C^1_I}\right)_{|n-1}$. But then strict monotonicity of $f^1$ yields

$$\mathcal{I}^1_{n|n} = f^1\left((\mathcal{I}^2_{n-1} \cup l)_{|C^1_I}\right)_{|n} = f^1\left((\mathcal{I}^2_n \cup l)_{|C^1_I}\right)_{|n} = \mathcal{I}^1_{n+1|n}.$$

By a similar argument it follows that $\mathcal{I}^2_{n|n} = \mathcal{I}^2_{n+1|n}$ as required.

The result now follows by choosing $N = \mathrm{eol}(l)$.

(ii) We first need to show that $C_I \cap C_O = \emptyset$:

$$\begin{aligned}
C_I \cap C_O &= ((C^1_I \cup C^2_I) \setminus C_{\mathrm{int}}) \cap ((C^1_O \cup C^2_O) \setminus C_{\mathrm{int}}) \\
&= ((C^1_I \cup C^2_I) \cap (C^1_O \cup C^2_O)) \setminus C_{\mathrm{int}} \\
&= ((C^1_I \cap (C^1_O \cup C^2_O)) \cup (C^2_I \cap (C^1_O \cup C^2_O))) \setminus C_{\mathrm{int}} \\
&= ((C^1_I \cap C^2_O) \cup (C^2_I \cap C^1_O)) \setminus C_{\mathrm{int}} \qquad (C^j_I \cap C^j_O = \emptyset, j = 1, 2) \\
&= C_{\mathrm{int}} \setminus C_{\mathrm{int}} \\
&= \emptyset.
\end{aligned}$$

Next, we must show that $f$ is strictly monotone. Let $l_1, l_2 \in \mathcal{L}_{C_I}$ be given and assume that $l_{1|t} = l_{2|t}$, for some timestamp $t \in \mathbb{N}$ with $t < \min(\mathrm{eol}(l_1), \mathrm{eol}(l_2))$.

Then $f(l_1) = (\mathcal{I}^1_N \cup \mathcal{I}^2_N)_{|C_O}$ and $f(l_2) = (\mathcal{J}^1_M \cup \mathcal{J}^2_M)_{|C_O}$ for some $N$ and $M$. We show by induction on $n$ that:

$$\mathcal{I}^1_{n|t+1} = \mathcal{J}^1_{n\,|t+1} \text{ and } \mathcal{I}^2_{n|t+1} = \mathcal{J}^2_{n\,|t+1}.$$

$n = 0$: This case is trivial since $\mathcal{I}^1_0(t')(c) = \mathcal{I}^2_0(t')(c) = \mathcal{J}^1_0(t')(c) = \mathcal{J}^1_0(t')(c) = \varepsilon$ for all timestamps $t' \in \mathbb{N}$ and channels $c \in C_O$ with $0 \leq t' < t+1$.

$n > 0$: By the induction hypothesis it follows that $\mathcal{I}^2_{n-1|t+1} = \mathcal{J}^2_{n-1|t+1}$ and therefore $((\mathcal{I}^2_{n-1} \cup l_1)_{|C^1_I})_{|t} = ((\mathcal{J}^2_{n-1} \cup l_2)_{|C^1_I})_{|t}$. But then strict monotonicity of $f^1$ yields:

$$\mathcal{I}^1_{n|t+1} = f^1((\mathcal{I}^2_{n-1} \cup l_1)_{|C^1_I})_{|t+1} = f^1((\mathcal{J}^2_{n-1} \cup l_2)_{|C^1_I})_{|t+1} = \mathcal{J}^2_{n\,|t+1}.$$

By a similar argument it follows that $\mathcal{I}^2_{n|t+1} = \mathcal{J}^2_{n\,|t+1}$ as required.

So for $k = \max(N, M)$ it follows that $(\mathcal{I}^1_k \cup \mathcal{I}^2_k)_{|t+1} = (\mathcal{J}^1_k \cup \mathcal{J}^2_k)_{|t+1}$, which implies that $f(l_1)_{|t+1} = f(l_2)_{|t+1}$ as required.

□

**Lemma C.1.1.** *Below follows a series of results about bisimulations.*

*(1) The identity relation $R_{\text{id}} \subseteq S \times S$ is a bisimulation for $\mathsf{a}$ and $\mathsf{a}$:*

$$R_{\text{id}} = \{(s, s) \mid s \in S\}.$$

*(2) Let $R \subseteq S_1 \times S_2$ be a bisimulation for $\mathsf{a}_1$ and $\mathsf{a}_2$. Then the inverse relation $R^{-1} \subseteq S_2 \times S_1$ is a bisimulation for $\mathsf{a}_2$ and $\mathsf{a}_1$:*

$$R^{-1} = \{(s_2, s_1) \mid (s_1, s_2) \in R\}.$$

*(3) Let $R_1 \subseteq S_1 \times S_2$ and $R_2 \subseteq S_2 \times S_3$ be bisimulations for $\mathsf{a}_1$,$\mathsf{a}_2$ and $\mathsf{a}_2$,$\mathsf{a}_3$. Then the composed relation $R_1 \circ R_2 \subseteq S_1 \times S_3$ is a bisimulation for $\mathsf{a}_1$ and $\mathsf{a}_3$:*

$$R_1 \circ R_2 = \{(s_1, s_3) \mid \exists s_2 \in S_2.\ (s_1, s_2) \in R_1 \wedge (s_2, s_3) \in R_2\}.$$

*Proof.* We show that each of the relations are bisimulations.

(1) Assume that $(s_1, s_2) \in R_{\text{id}}$. Then $s_1 = s_2$ so the conditions are trivially fulfilled.

(2) Assume that $(s_2, s_1) \in R^{-1}$. Then $(s_1, s_2) \in R$ which means that:

$$\delta^1_{\text{o}}(s_1) = \delta^2_{\text{o}}(s_2) \quad \text{and} \quad \forall m \in \mathcal{M}_{C_I}.(\delta^1_{\text{t}}(s_1, m), \delta^2_{\text{t}}(s_2, m)) \in R,$$

which gives:

$$\delta^2_{\text{o}}(s_2) = \delta^1_{\text{o}}(s_1) \quad \text{and} \quad \forall m \in \mathcal{M}_{C_I}.(\delta^2_{\text{t}}(s_2, m), \delta^1_{\text{t}}(s_1, m)) \in R^{-1}.$$

(3) Assume that $(s_1, s_3) \in R_1 \circ R_2$. Then there exists $s_2 \in S_2$ such that $(s_1, s_2) \in R_1$ and $(s_2, s_3) \in R_2$ and then we get:

$$\delta^1_{\text{o}}(s_1) = \delta^2_{\text{o}}(s_2) = \delta^3_{\text{o}}(s_3),$$

and because $(\delta^1_{\text{t}}(s_1, m), \delta^2_{\text{t}}(s_2, m)) \in R_1$ and $(\delta^2_{\text{t}}(s_2, m), \delta^3_{\text{t}}(s_3, m)) \in R_2$ then by definition $(\delta^1_{\text{t}}(s_1, m), \delta^3_{\text{t}}(s_3, m)) \in R_1 \circ R_2$.

□

**Proof of Lemma 3.3.4**

*Proof.* The properties are proved one at a time below.

**Reflexivity:** We must show that for all $\mathsf{a}$ we have that $\mathsf{a} \equiv \mathsf{a}$. This corresponds to constructing a bisimulation $R \subseteq S \times S$ such that $(s_0, s_0) \in R$. Lemma C.1.1 (1) gives such a relation.

**Symmetry:** We must show that if $\mathsf{a}_1 \equiv \mathsf{a}_2$ then $\mathsf{a}_2 \equiv \mathsf{a}_1$. Similar to the reflexivity case we must take some arbitrary bisimulation $R \subseteq S_1 \times S_2$ and then construct a bisimulation on $S_2 \times S_1$. Lemma C.1.1 (2) gives this relation.

**Transitivity:** We must show that if $\mathsf{a}_1 \equiv \mathsf{a}_2$ and $\mathsf{a}_2 \equiv \mathsf{a}_3$ then $\mathsf{a}_1 \equiv \mathsf{a}_3$. In this case we are given bisimulations $R_1 \subseteq S_1 \times S_2$ and $R_2 \subseteq S_2 \times S_3$ and we construct a bisimulation using Lemma C.1.1 (3).

**Congruence:** Now let:

$$\begin{aligned}
\mathsf{a}_1 &= (C_I, C_O, S_1, s_0^1, \delta_{\mathrm{o}}^1, \delta_{\mathrm{t}}^1),\\
\mathsf{a}_2 &= (C_I, C_O, S_2, s_0^2, \delta_{\mathrm{o}}^2, \delta_{\mathrm{t}}^2),\\
\mathsf{a}_3 &= (C_I', C_O', S_3, s_0^3, \delta_{\mathrm{o}}^3, \delta_{\mathrm{t}}^3),\\
\mathsf{a}_4 &= (C_I', C_O', S_4, s_0^4, \delta_{\mathrm{o}}^4, \delta_{\mathrm{t}}^4),\\
\mathsf{a}_1 \parallel \mathsf{a}_3 &= (C_I'', C_O'', S_1 \times S_3, \langle s_0^1, s_0^3 \rangle, \delta_{\mathrm{o}}^{1\|3}, \delta_{\mathrm{t}}^{1\|3}),\\
\mathsf{a}_2 \parallel \mathsf{a}_4 &= (C_I'', C_O'', S_2 \times S_4, \langle s_0^2, s_0^4 \rangle, \delta_{\mathrm{o}}^{2\|4}, \delta_{\mathrm{t}}^{2\|4}).
\end{aligned}$$

We show that given bisimulations $R_1 \subseteq S_1 \times S_2$ and $R_2 \subseteq S_3 \times S_4$, then the relation:

$$\begin{aligned}
&R \subseteq (S_1 \times S_3) \times (S_2 \times S_4),\\
&R = \{(\langle s_1, s_3 \rangle, \langle s_2, s_4 \rangle) \mid (s_1, s_2) \in R_1 \land (s_3, s_4) \in R_2\},
\end{aligned}$$

is a bisimulation (and clearly it relates the starting states of $\mathsf{a}_1 \parallel \mathsf{a}_3$ and $\mathsf{a}_2 \parallel \mathsf{a}_4$).

So assume that $(\langle s_1, s_3 \rangle, \langle s_2, s_4 \rangle) \in R$. This means that $(s_1, s_2) \in R_1$ and $(s_3, s_4) \in R_2$. We now have that:

$$\begin{aligned}
\delta_{\mathrm{o}}^{1\|3}(\langle s_1, s_3 \rangle) &= (\delta_{\mathrm{o}}^1(s_1) \cup \delta_{\mathrm{o}}^3(s_3))_{|C_O''} && \text{(by definition)}\\
&= (\delta_{\mathrm{o}}^2(s_2) \cup \delta_{\mathrm{o}}^4(s_4))_{|C_O''} && \text{($R_1$ and $R_2$ are bisimulations)}\\
&= \delta_{\mathrm{o}}^{2\|4}(\langle s_2, s_4 \rangle). && \text{(by definition)}
\end{aligned}$$

And if $m \in \mathcal{M}_{C_I''}$ then because $R_1$ and $R_2$ are bisimulations we have that:

$$\begin{aligned}
&(\delta_{\mathrm{t}}^1(s_1, (m \cup \delta_{\mathrm{o}}^3(s_3))_{|C_I}), \delta_{\mathrm{t}}^2(s_2, (m \cup \delta_{\mathrm{o}}^3(s_3))_{|C_I})) \in R_1,\\
&(\delta_{\mathrm{t}}^3(s_3, (m \cup \delta_{\mathrm{o}}^1(s_1))_{|C_I'}), \delta_{\mathrm{t}}^4(s_4, (m \cup \delta_{\mathrm{o}}^1(s_1))_{|C_I'})) \in R_2.
\end{aligned}$$

But then we have (using infix notation for the relation $R$):

$$\begin{aligned}\delta_{\mathrm{t}}^{1\|3}(\langle s_1, s_3\rangle, m) &= \langle \delta_{\mathrm{t}}^{1}(s_1, (m \cup \delta_{\mathrm{o}}^{3}(s_3))_{|C_I}), \delta_{\mathrm{t}}^{3}(s_3, (m \cup \delta_{\mathrm{o}}^{1}(s_1))_{|C_I'})\rangle \\ &\quad R\ \langle \delta_{\mathrm{t}}^{2}(s_2, (m \cup \delta_{\mathrm{o}}^{3}(s_3))_{|C_I}), \delta_{\mathrm{t}}^{4}(s_4, (m \cup \delta_{\mathrm{o}}^{1}(s_1))_{|C_I'})\rangle \\ &= \langle \delta_{\mathrm{t}}^{2}(s_2, (m \cup \delta_{\mathrm{o}}^{4}(s_4))_{|C_I}), \delta_{\mathrm{t}}^{4}(s_4, (m \cup \delta_{\mathrm{o}}^{2}(s_2))_{|C_I'})\rangle \\ &= \delta_{\mathrm{t}}^{2\|4}(\langle s_2, s_4\rangle, m),\end{aligned}$$

where the second to last equality follows from $R_1$ and $R_2$ being bisimulations. Hence the result follows as required.

□

**Lemma C.1.2.** *For sets with $A \cap B = \emptyset$ the following equality holds:*

$$(A \cup (B \setminus C)) \setminus (D \cup (C \setminus B)) = (A \cup B) \setminus (C \cup D).$$

*Proof.* Straightforward. □

### Proof of Lemma 3.3.5

*Proof.* The first part of the proof is to show that the input channels and output channels of the composed automata are well-defined. We show that

$$\begin{aligned}C_I^{1\|(2\|3)} &= (C_I^1 \cup C_I^2 \cup C_I^3) \setminus (C_O^1 \cup C_O^2 \cup C_O^3) = C_I^{(1\|2)\|3}, \\ C_O^{1\|(2\|3)} &= (C_O^1 \cup C_O^2 \cup C_O^3) \setminus (C_I^1 \cup C_I^2 \cup C_I^3) = C_O^{(1\|2)\|3}.\end{aligned}$$

We only show the case for the input channels, the output case is symmetric. We start with the left equality, and first note that:

$$C_I^{1\|(2\|3)} = \left(C_I^1 \cup \left((C_I^2 \cup C_I^3) \setminus (C_O^2 \cup C_O^3)\right)\right) \setminus \left(C_O^1 \cup \left((C_O^2 \cup C_O^3) \setminus (C_I^2 \cup C_I^3)\right)\right).$$

From the assumptions we get that $C_I^1 \cap (C_I^2 \cup C_I^3) = \emptyset$, which means the we can instantiate Lemma C.1.2 with $A = C_I^1$, $B = C_I^2 \cup C_I^3$, $C = C_O^2 \cup C_O^3$, and $D = C_O^1$. We thus get:

$$C_I^{1\|(2\|3)} = (C_I^1 \cup C_I^2 \cup C_I^3) \setminus (C_O^1 \cup C_O^2 \cup C_O^3).$$

For the right equality we note that:

$$C_I^{(1\|2)\|3} = \left(((C_I^1 \cup C_I^2) \setminus (C_O^1 \cup C_O^2)) \cup C_I^3\right) \setminus \left(((C_O^1 \cup C_O^2) \setminus (C_I^1 \cup C_I^2)) \cup C_O^3\right).$$

But then we have from the assumptions that $C_I^3 \cap (C_I^1 \cup C_I^2) = \emptyset$ and we can instantiate Lemma C.1.2 with $A = C_I^3$, $B = C_I^1 \cup C_I^2$, $C = C_O^1 \cup C_O^2$, and $D = C_O^3$. We thus get:

$$C_I^{(1\|2)\|3} = (C_I^1 \cup C_I^2 \cup C_I^3) \setminus (C_O^1 \cup C_O^2 \cup C_O^3).$$

We abbreviate:

$$\begin{aligned}C_I &= (C_I^1 \cup C_I^2 \cup C_I^3) \setminus (C_O^1 \cup C_O^2 \cup C_O^3), \\ C_O &= (C_O^1 \cup C_O^2 \cup C_O^3) \setminus (C_I^1 \cup C_I^2 \cup C_I^3).\end{aligned}$$

The second part of the proof is to construct a bisimulation that relates the start states. We use the following (which clearly relates the start states):

$$R \subseteq \big(S_1 \times (S_2 \times S_3)\big) \times \big((S_1 \times S_2) \times S_3\big),$$
$$R = \{(\langle s_1, \langle s_2, s_3\rangle\rangle, \langle\langle s_1, s_2\rangle, s_3\rangle) \mid s_1 \in S_1, s_2 \in S_2, s_3 \in S_3\}.$$

We prove that this is a bisimulation, so assume that:

$$(\langle s_1, \langle s_2, s_3\rangle\rangle, \langle\langle s_1, s_2\rangle, s_3\rangle) \in R.$$

First we need to compare the outputs:

$$\begin{aligned}
\delta_{\mathrm{o}}^{1\|(2\|3)}(\langle s_1, \langle s_2, s_3\rangle\rangle) &= \big(\delta_{\mathrm{o}}^1(s_1) \cup (\delta_{\mathrm{o}}^2(s_2) \cup \delta_{\mathrm{o}}^3(s_3))_{|C_O^{2\|3}}\big)_{|C_O} \\
&\overset{(a)}{=} (\delta_{\mathrm{o}}^1(s_1) \cup \delta_{\mathrm{o}}^2(s_2) \cup \delta_{\mathrm{o}}^3(s_3))_{|C_O} \\
&\overset{(b)}{=} \big((\delta_{\mathrm{o}}^1(s_1) \cup \delta_{\mathrm{o}}^2(s_2))_{|C_O^{1\|2}} \cup \delta_{\mathrm{o}}^3(s_3)\big)_{|C_O} \\
&= \delta_{\mathrm{o}}^{(1\|2)\|3}(\langle\langle s_1, s_2\rangle, s_3\rangle),
\end{aligned}$$

where (a) follows from $C_O^1 \cap (C_O^2 \cup C_O^3) = \emptyset$ and (b) follows from $(C_O^1 \cup C_O^2) \cap C_O^3 = \emptyset$. Finally, we need to show that for any $m \in \mathcal{M}_{C_I}$ the updated states are related. To show this we introduce shorthand notation:

$$\begin{aligned}
s_1' &= \delta_{\mathrm{t}}^1(s_1, (m \cup \delta_{\mathrm{o}}^2(s_2) \cup \delta_{\mathrm{o}}^3(s_3))_{|C_I^1}), \\
s_2' &= \delta_{\mathrm{t}}^2(s_2, (m \cup \delta_{\mathrm{o}}^1(s_1) \cup \delta_{\mathrm{o}}^3(s_3))_{|C_I^2}), \\
s_3' &= \delta_{\mathrm{t}}^3(s_3, (m \cup \delta_{\mathrm{o}}^1(s_1) \cup \delta_{\mathrm{o}}^2(s_2))_{|C_I^3}).
\end{aligned}$$

And we calculate:

$$\begin{aligned}
&\delta_{\mathrm{t}}^{1\|(2\|3)}(\langle s_1, \langle s_2, s_3\rangle\rangle, m) \\
&= \langle \delta_{\mathrm{t}}^1(s_1, (m \cup (\delta_{\mathrm{o}}^2(s_2) \cup \delta_{\mathrm{o}}^3(s_3))_{|C_O^{2\|3}})_{|C_I^1}), \delta_{\mathrm{t}}^{2\|3}(\langle s_2, s_3\rangle, (m \cup \delta_{\mathrm{o}}^1(s_1))_{|C_I^{2\|3}})\rangle \\
&\overset{(a)}{=} \langle \delta_{\mathrm{t}}^1(s_1, (m \cup \delta_{\mathrm{o}}^2(s_2) \cup \delta_{\mathrm{o}}^3(s_3))_{|C_I^1}), \delta_{\mathrm{t}}^{2\|3}(\langle s_2, s_3\rangle, (m \cup \delta_{\mathrm{o}}^1(s_1))_{|C_I^{2\|3}})\rangle \\
&= \langle s_1', \langle \delta_{\mathrm{t}}^2(s_2, ((m \cup \delta_{\mathrm{o}}^1(s_1))_{|C_I^{2\|3}} \cup \delta_{\mathrm{o}}^3(s_3))_{|C_I^2}), \delta_{\mathrm{t}}^3(s_3, ((m \cup \delta_{\mathrm{o}}^1(s_1))_{|C_I^{2\|3}} \cup \delta_{\mathrm{o}}^2(s_2))_{|C_I^3})\rangle\rangle \\
&\overset{(b)}{=} \langle s_1', \langle \delta_{\mathrm{t}}^2(s_2, (m \cup \delta_{\mathrm{o}}^1(s_1) \cup \delta_{\mathrm{o}}^3(s_3))_{|C_I^2}), \langle \delta_{\mathrm{t}}^3(s_3, (m \cup \delta_{\mathrm{o}}^1(s_1) \cup \delta_{\mathrm{o}}^2(s_2))_{|C_I^3})\rangle\rangle \\
&= \langle s_1', \langle s_2', s_3'\rangle\rangle,
\end{aligned}$$

where (a) follows from $C_I^1 \cap (C_I^2 \cup C_I^3) = \emptyset$, and (b) follows from $C_O^1 \cap (C_O^2 \cup C_O^3) = \emptyset$.

In a similar way we can calculate:

$$\begin{aligned}
&\delta_{\mathrm{t}}^{(1\|2)\|3}(\langle\langle s_1, s_2\rangle, s_3\rangle, m) \\
&\quad= \langle \delta_{\mathrm{t}}^{1\|2}(\langle s_1, s_2\rangle, (m \cup \delta_{\mathrm{o}}^3(s_3))_{|C_I^{1\|2}}), \delta_{\mathrm{t}}^3(s_3, (m \cup (\delta_{\mathrm{o}}^1(s_1) \cup \delta_{\mathrm{o}}^2(s_2))_{|C_O^{1\|2}})_{|C_I^3})\rangle \\
&\quad= \langle\langle s_1', s_2'\rangle, s_3'\rangle.
\end{aligned}$$

And therefore by definition we have that:

$$(\delta_{\mathrm{t}}^{1\|(2\|3)}(\langle s_1, \langle s_2, s_3\rangle\rangle, m), \delta_{\mathrm{t}}^{(1\|2)\|3}(\langle\langle s_1, s_2\rangle, s_3\rangle, m)) \in R,$$

which concludes the proof. □

**Proof of Theorem 3.5.13**

*Proof.* We will use the following definitions and abbreviations in the proof:

$$c_i = (\Lambda_{\mathsf{PA}_i}, \Lambda_{\mathsf{A}_i\mathsf{P}}, G_i, g_i, \rho_i) \qquad \mathsf{c}_i = (C_i, G_i, g_i, \rho_i') \qquad \Lambda_I = \bigcup_{i=1}^{n} \Lambda_{\mathsf{A}_i\mathsf{P}}$$

$$\Lambda_O = \bigcup_{i=1}^{n} \Lambda_{\mathsf{PA}_i} \qquad \Lambda = \Lambda_I \cup \Lambda_O \qquad C_E = (\bigcup_{i=1}^{n} C_i) \setminus C_O.$$

From the definition of automaton contract conformance we get that there exists a conformance relation $R \subseteq \mathbb{Q}\times S\times G_1\times\ldots\times G_n$ for $\mathsf{a}$ and $\mathsf{C} = \{\mathsf{c}_1, \ldots, \mathsf{c}_n\}$ that relates the start states. We must construct a conformance relation $R'$ for $i = (\mathsf{a}, r)$ and $C = \{c_1, \ldots, c_n\}$, so we claim that $R' = R$ is such a conformance relation. Clearly, it relates the start states, so if we can show that $R'$ is a conformance relation for $i$ and $C$, then the result follows.

So assume that $(k, s, g_1, \ldots, g_n) \in R'$ and let $m \in \mathcal{M}_{\Lambda_I}$ be given. Now let:

$$(g_i', k_i) = \rho_i(g_i, \overline{r}_i(\delta_{\mathrm{o}}(s), m), m_{|\Lambda_{\mathsf{A}_i\mathsf{P}}}), \tag{C.1}$$

for $i = 1, \ldots, n$. We must then show that:

$$\sum_{i=1}^{n} k_i \geq k, \tag{C.2}$$

$$(k - \sum_{i=1}^{n} k_i, \delta_{\mathrm{t}}(s, m \circ r_{\mathrm{i}}), g_1', \ldots, g_n') \in R'. \tag{C.3}$$

In order to show this, we construct a move $\widehat{m} \in \mathcal{M}_{C_E}$ that can be used in the automaton conformance relation $R$:

$$\widehat{m}(\alpha) = m(\lambda) \text{ iff } \lambda \in \Lambda_I \text{ and } \theta(\lambda) = \alpha.$$

First, we must show that $\widehat{m}$ is well-defined, that is (1) there always exists such a $\lambda$, and (2) the function value is unique.

(1) Let $\alpha \in C_E$ be given. Then there exists a link $\lambda \in \Lambda$ such that $\theta(\lambda) = \alpha$ by construction of the projection. If $\lambda \in \Lambda_I$ we are done, so assume that $\lambda \in \Lambda_O$. If $r_{\mathrm{o}}(\alpha') = \lambda$ for some $\alpha' \in C_O$ then $\alpha = \theta(\lambda) = \theta(r_{\mathrm{o}}(\alpha')) = \alpha'$ in contradiction with $\alpha \notin C_O$. Therefore, $r_{\mathrm{d}}(\lambda') = \lambda$ and hence $\theta(\lambda') = \theta(\lambda) = \alpha$ with $\lambda' \in \Lambda_I$.

(2) Let $\theta(\lambda) = \theta(\lambda')$ with $\lambda, \lambda' \in \Lambda_I$. Then by condition (4) of Definition 3.5.9 we have that either $\lambda = \lambda'$, $r_{\mathrm{d}}(\lambda) = \lambda'$, or $r_{\mathrm{d}}(\lambda') = \lambda$. But the last two cases cannot happen, since that would imply that either $\lambda \in \Lambda_O$ or $\lambda' \in \Lambda_O$. Hence $\lambda = \lambda'$.

We can now use $\widehat{m}$ in the relation $R$ because $C_I \subseteq C_E$, compare Observation 3.5.11. So let:

$$\begin{aligned}(g_i'', k_i'') &= \rho_i'(g_i, (\widehat{m} \cup \delta_{\mathrm{o}}(s))_{|C_i}) \\ &= \rho_i(g_i, (\widehat{m} \cup \delta_{\mathrm{o}}(s))_{|C_i} \circ \theta_{|\Lambda_{\mathsf{PA}_i}}, (\widehat{m} \cup \delta_{\mathrm{o}}(s))_{|C_i} \circ \theta_{|\Lambda_{\mathsf{A}_i\mathsf{P}}}).\end{aligned} \tag{C.4}$$

We now show that:

$$\overline{r}_i(\delta_{\mathrm{o}}(s), m) = (\widehat{m} \cup \delta_{\mathrm{o}}(s))_{|C_i} \circ \theta_{|\Lambda_{\mathsf{PA}_i}}, \tag{C.5}$$

$$m_{|\Lambda_{\mathsf{A}_i\mathsf{P}}} = (\widehat{m} \cup \delta_{\mathrm{o}}(s))_{|C_i} \circ \theta_{|\Lambda_{\mathsf{A}_i\mathsf{P}}}. \tag{C.6}$$

(C.5) Now,

$$\overline{r}_i(\delta_{\mathrm{o}}(s), m)(\lambda) = \begin{cases} \delta_{\mathrm{o}}(s)(\alpha) & \text{if } r_{\mathrm{o}}(\alpha) = \lambda, \\ m(\lambda') & \text{if } r_{\mathrm{d}}(\lambda') = \lambda. \end{cases}$$

So assume that $r_{\mathrm{o}}(\alpha) = \lambda$. Then LHS $= \delta_{\mathrm{o}}(s)(\alpha)$. Now by the second requirement for renaming maps it follows that $\theta_{|\Lambda_{\mathsf{PA}_i}}(\lambda) = \alpha$, so also RHS $= \delta_{\mathrm{o}}(s)(\alpha)$.

Now assume that $r_{\mathrm{d}}(\lambda') = \lambda$. Then LHS $= m(\lambda')$. Now by condition (3) of Definition 3.5.9 it follows that $\theta_{|\Lambda_{\mathsf{PA}_i}}(\lambda) = \theta_{|\Lambda_{\mathsf{PA}_i}}(\lambda') \notin C_O$, hence RHS $= \widehat{m}(\theta_{|\Lambda_{\mathsf{PA}_i}}(\lambda')) = m(\lambda')$ as required.

(C.6) Now let $\lambda \in \Lambda_{\mathsf{A}_i\mathsf{P}}$ be given. Then $\theta_{|\Lambda_{\mathsf{A}_i\mathsf{P}}}(\lambda) \notin C_O$ so RHS $= \widehat{m}(\theta_{|\Lambda_{\mathsf{A}_i\mathsf{P}}}(\lambda)) = m(\lambda) =$ LHS as required.

We have now established (C.5) and (C.6), from which it follows that $g_i' = g_i''$ and $k_i = k_i''$ for $i = 1, \ldots, n$, compare (C.1) and (C.4). Hence (C.2) follows from the fact that $\sum_{i=1}^{n} k_i'' \geq k$ (the conformance relation $R$). Now, in order to show (C.3) it suffices to show that:

$$\delta_{\mathrm{t}}(s, m \circ r_{\mathrm{i}}) = \delta_{\mathrm{t}}(s, \widehat{m}_{|C_I}),$$

which means that we must show that:

$$(m \circ r_{\mathrm{i}})(\alpha) = \widehat{m}_{|C_I}(\alpha),$$

for all $\alpha \in C_I$. But this follows from the first condition of renaming maps, and hence the result follows. □

### Proof of Theorem 3.5.15

*Proof.* We are given two conformance relations:

$$R_1 \subseteq \mathbb{Q} \times S_1 \times G_1 \times \ldots \times G_n \times G_1' \times \ldots \times G_{n_1}',$$
$$R_2 \subseteq \mathbb{Q} \times S_2 \times G_1 \times \ldots \times G_n \times G_1'' \times \ldots \times G_{n_2}'',$$

that relate the start states. We must construct a new conformance relation:

$$R \subseteq \mathbb{Q} \times (S_1 \times S_2) \times G_1' \times \ldots \times G_{n_1}' \times G_1'' \times \ldots \times G_{n_2}'',$$

that relates the start states in the parallel composition. We define $R$ in the following way:

$$\begin{aligned}(k, \langle s_1, s_2\rangle, g_1', \ldots, g_{n_1}', g_1'', \ldots, g_{n_2}'') \in R \text{ iff } & (k_1, s_1, g_1, \ldots, g_n, g_1', \ldots, g_{n_1}') \in R_1 \text{ and} \\ & (k_2, s_2, g_1, \ldots, g_n, g_1'', \ldots, g_{n_2}'') \in R_2 \text{ and} \\ & k_1 + k_2 = k \text{ for some} \\ & \quad (g_1, \ldots, g_n) \in G_1 \times \cdots \times G_n \text{ and} \\ & \quad k_1, k_2 \in \mathbb{Q}.\end{aligned}$$

It is clear that $R$ relates the start states, so it suffices to show that $R$ is a conformance relation.

So assume that $(k, \langle s_1, s_2\rangle, g'_1, \dots, g'_{n_1}, g''_1, \dots, g''_{n_2}) \in R$. This means that there exists $k_1, k_2, g_1, \dots, g_n$ such that:

$$k = k_1 + k_2, \tag{C.7}$$
$$(k_1, s_1, g_1, \dots, g_n, g'_1, \dots, g'_{n_1}) \in R_1, \tag{C.8}$$
$$(k_2, s_2, g_1, \dots, g_n, g''_1, \dots, g''_{n_2}) \in R_2. \tag{C.9}$$

Now put for easy reference:

$$C = \bigcup_{i=1}^{n} C_i \qquad C' = \bigcup_{i=1}^{n_1} C'_i \qquad C'' = \bigcup_{i=1}^{n_2} C''_i,$$

let $m \in \mathcal{M}_{(C_I \cup C' \cup C'') \setminus C_O}$ be given, and let:

$$(\underline{g'_i}, \underline{k'_i}) = \rho'_i(g'_i, (m \cup \delta_\mathrm{o}(\langle s_1, s_2\rangle))_{|C'_i}), \text{ for } i = 1 \dots n_1, \tag{C.10}$$
$$(\underline{g''_i}, \underline{k''_i}) = \rho''_i(g''_i, (m \cup \delta_\mathrm{o}(\langle s_1, s_2\rangle))_{|C''_i}), \text{ for } i = 1 \dots n_2. \tag{C.11}$$

We must then show that:

$$\sum_{i=1}^{n_1} \underline{k'_i} + \sum_{i=1}^{n_2} \underline{k''_i} \geq k, \tag{C.12}$$
$$(k - \sum_{i=1}^{n_1} \underline{k'_i} + \sum_{i=1}^{n_2} \underline{k''_i}, \delta_\mathrm{t}(\langle s_1, s_2\rangle, m_{|C_I}), \underline{g'_1}, \dots, \underline{g'_{n_1}}, \underline{g''_1}, \dots, \underline{g''_{n_2}}) \in R. \tag{C.13}$$

In order to show the above, we construct $m_1 \in \mathcal{M}_{(C^1_I \cup C \cup C') \setminus C^1_O}$ and $m_2 \in \mathcal{M}_{(C^2_I \cup C \cup C'') \setminus C^2_O}$ to use in $R_1$ and $R_2$. The idea is to define $m_1$ to behave like (a) $m$ where this is possible, (b) to the value of $\mathsf{a}_2$'s output on $C^2_O$, and (c) to some arbitrary value on the rest. $m_2$ will be defined in a similar way.

To make this intuition precise, we see that because $C' \cap C_\mathrm{int} = \emptyset$ and $C'' \cap C_\mathrm{int} = \emptyset$ then (we call the domain of $m$ for $D$):

$$\begin{aligned} D &= (C_I \cup C' \cup C'') \setminus C_O \\ &= ((C^1_I \cup C^2_I) \setminus C_\mathrm{int} \cup C' \cup C'') \setminus (C^1_O \cup C^2_O \setminus C_\mathrm{int}) \\ &= ((C^1_I \cup C^2_I \cup C' \cup C'') \setminus C_\mathrm{int}) \setminus (C^1_O \cup C^2_O \setminus C_\mathrm{int}) \\ &= (C^1_I \cup C^2_I \cup C' \cup C'') \setminus (C^1_O \cup C^2_O \cup C_\mathrm{int}). \end{aligned}$$

Hence, the elements $\alpha \in (C^1_I \cup C \cup C') \setminus C^1_O$ that are not in $D$, that is $\alpha$ on which $m_1$ cannot agree with $m$, can have two possible forms:

1. $\alpha \in C^2_O$, or
2. $\alpha \notin C^2_O$ and $\alpha \notin C_\mathrm{int}$ and $\alpha \in C$.

(The case $\alpha \in C_\mathrm{int} \setminus C^2_O$ cannot occur because $\alpha \notin C^1_O$.)

Now, for each channel $\alpha$ in the second case we choose some (fixed) value $x_\alpha \in \mathcal{A}_\alpha$ and set the value of $m_1$ to that value. So in summary we get the following definition of $m_1$:

$$m_1(\alpha) = \begin{cases} m(\alpha) & \text{if } \alpha \in D, \\ \delta_{\mathrm{o}}^2(s_2)(\alpha) & \text{if } \alpha \in C_O^2, \\ x_\alpha & \text{otherwise.} \end{cases}$$

(The three cases correspond to (a), (b), and (c) mentioned above). By a similar analysis $m_2$ is defined by:

$$m_2(\alpha) = \begin{cases} m(\alpha) & \text{if } \alpha \in D, \\ \delta_{\mathrm{o}}^1(s_1)(\alpha) & \text{if } \alpha \in C_O^1, \\ x_\alpha & \text{otherwise.} \end{cases}$$

We now apply $m_1$ in conformance relation $R_1$ (C.8):

$$(\underline{\underline{g_i}}, \underline{\underline{k_i}}) = \rho_i(g_i, (m_1 \cup \delta_{\mathrm{o}}^1(s_1))_{|C_i}), \text{ for } i = 1 \dots n, \tag{C.14}$$

$$(\underline{\underline{g_i'}}, \underline{\underline{k_i'}}) = \rho_i'(g_i', (m_1 \cup \delta_{\mathrm{o}}^1(s_1))_{|C_i'}), \text{ for } i = 1 \dots n_1, \tag{C.15}$$

and $m_2$ in conformance relation $R_2$ (C.9):

$$(\underline{\underline{\underline{g_i}}}, \underline{\underline{\underline{k_i}}}) = \overline{\rho_i}(g_i, (m_2 \cup \delta_{\mathrm{o}}^2(s_2))_{|C_i}), \text{ for } i = 1 \dots n, \tag{C.16}$$

$$(\underline{\underline{\underline{g_i''}}}, \underline{\underline{\underline{k_i''}}}) = \rho_i''(g_i'', (m_2 \cup \delta_{\mathrm{o}}^2(s_2))_{|C_i''}), \text{ for } i = 1 \dots n_2. \tag{C.17}$$

In order to show (C.12) and (C.13), we first prove three equalities:

(I) $(m_1 \cup \delta_{\mathrm{o}}^1(s_1))_{|C_i'} = (m \cup \delta_{\mathrm{o}}(\langle s_1, s_2 \rangle))_{|C_i'}$, for $i = 1, \dots, n_1$,

(II) $(m_2 \cup \delta_{\mathrm{o}}^2(s_2))_{|C_i''} = (m \cup \delta_{\mathrm{o}}(\langle s_1, s_2 \rangle))_{|C_i''}$, for $i = 1, \dots, n_2$, and

(III) $(m_1 \cup \delta_{\mathrm{o}}^1(s_1))_{|C_i} = (m_2 \cup \delta_{\mathrm{o}}^2(s_2))_{|C_i}$, for $i = 1, \dots, n$.

(I) Let $\alpha \in C_i'$ and consider the possible cases:

- $\alpha \in C_O^1$: Here we must have that $\alpha \in C_O$ because $C_{\text{int}} \cap C_i' = \emptyset$, and hence:

$$\begin{aligned} (m_1 \cup \delta_{\mathrm{o}}^1(s_1))_{|C_i'}(\alpha) &= \delta_{\mathrm{o}}^1(s_1)(\alpha) \\ &= \delta_{\mathrm{o}}(\langle s_1, s_2 \rangle)(\alpha) \\ &= (m \cup \delta_{\mathrm{o}}(\langle s_1, s_2 \rangle))_{|C_i'}(\alpha). \end{aligned}$$

- $\alpha \in C_O^2$: Here we must again have that $\alpha \in C_O$, and hence:

$$\begin{aligned} (m_1 \cup \delta_{\mathrm{o}}^1(s_1))_{|C_i'}(\alpha) &= m_1(\alpha) \\ &= \delta_{\mathrm{o}}^2(s_2)(\alpha) \\ &= \delta_{\mathrm{o}}(\langle s_1, s_2 \rangle)(\alpha) \\ &= (m \cup \delta_{\mathrm{o}}(\langle s_1, s_2 \rangle))_{|C_i'}(\alpha). \end{aligned}$$

– $\alpha \in D$: Here we have that

$$\begin{aligned}(m_1 \cup \delta_{\mathrm{o}}^1(s_1))_{|C_i'}(\alpha) &= m_1(\alpha) \\ &= m(\alpha) \\ &= (m \cup \delta_{\mathrm{o}}(\langle s_1, s_2\rangle))_{|C_i'}(\alpha).\end{aligned}$$

(II) Similar to the proof of (I).

(III) Follows directly by construction of $m_1$ and $m_2$.

Based on the equalities above, we can conclude the following:

(A) By (I) and definitions (C.10) and (C.15): $\underline{g_i'} = \underline{\underline{g_i'}}$ and $\underline{k_i'} = \underline{\underline{k_i'}}$ for $i = 1, \ldots, n_1$.

(B) By (II) and definitions (C.11) and (C.17): $\underline{g_i''} = \underline{\underline{\underline{g_i''}}}$ and $\underline{k_i''} = \underline{\underline{\underline{k_i''}}}$ for $i = 1, \ldots, n_2$.

(C) By (III) and definitions (C.14) and (C.16): $\underline{\underline{g_i}} = \underline{\underline{\underline{g_i}}}$ and $\underline{\underline{k_i}} = -\underline{\underline{\underline{k_i}}}$ for $i = 1, \ldots, n$.

And now finally the proof of (C.12):

$$\begin{aligned}\sum_{i=1}^{n_1} \underline{k_i'} + \sum_{i=1}^{n_2} \underline{k_i''} &= \sum_{i=1}^{n_1} \underline{\underline{k_i'}} + \sum_{i=1}^{n_2} \underline{\underline{\underline{k_i''}}} && \text{(by A and B)} \\ &= \sum_{i=1}^{n_1} \underline{\underline{k_i'}} + \left(\sum_{i=1}^{n} \underline{\underline{k_i}} + \sum_{i=1}^{n} \underline{\underline{\underline{k_i}}}\right) + \sum_{i=1}^{n_2} \underline{\underline{\underline{k_i''}}} && \text{(by C)} \\ &= \left(\sum_{i=1}^{n_1} \underline{\underline{k_i'}} + \sum_{i=1}^{n} \underline{\underline{k_i}}\right) + \left(\sum_{i=1}^{n} \underline{\underline{\underline{k_i}}} + \sum_{i=1}^{n_2} \underline{\underline{\underline{k_i''}}}\right) \\ &\geq k_1 + k_2 && \text{(by C.8 and C.9)} \\ &= k. && \text{(by C.7)}\end{aligned}$$

And in order to prove (C.13) it follows from the equalities in (A), (B), and (C) that it suffices to show that:

$$\delta_{\mathrm{t}}(\langle s_1, s_2\rangle, m_{|C_I}) = \langle \delta_{\mathrm{t}}^1(s_1, m_{1|C_I^1}), \delta_{\mathrm{t}}^2(s_2, m_{2|C_I^2})\rangle,$$

which follows by definition of $m_1$ and $m_2$.

This concludes the proof of the theorem. □

## C.2 Partial Magmas

In this appendix we introduce the non-standard notion of *partial magmas*, which is used to prove the equivalence result of Section 3.3.1.

**Definition C.2.1.** Let $G$ be a set with a partial binary operator $\cdot : G \times G \rightharpoonup G$ (we write $g_1 g_2$ for $g_1 \cdot g_2$). $(G, \cdot)$ is called a *partial magma* whenever the following holds for all $g_1, g_2, g_3 \in G$:

(1) if $g_1g_2$ is defined then so is $g_2g_1$, and

(2) if $g_1g_2$, $g_1g_3$, and $g_2g_3$ are defined then so is $(g_1g_2)g_3$.

**Lemma C.2.2.** *Let $(G, \cdot)$ be a partial magma. If $g_1g_2$, $g_1g_3$, and $g_2g_3$ are defined for $g_1, g_2, g_3 \in G$ then so is $g_1(g_2g_3)$.*

*Proof.* We get from condition (1) that $g_2g_1$ and $g_3g_1$ are defined, and therefore from condition (2) we get that $(g_2g_3)g_1$ is defined. The result then follows from condition (1). □

**Definition C.2.3.** Let $(G, \cdot)$ and $(H, \cdot)$ be partial magmas. A function $\phi : G \to H$ is called a *homomorphism* whenever the following holds for all $g_1, g_2 \in G$:

(1) $g_1g_2$ is defined iff $\phi(g_1)\phi(g_2)$ is defined, and

(2) $\phi(g_1g_2) = \phi(g_1)\phi(g_2)$.

$\phi$ is called an *isomorphism* whenever $\phi$ is a bijective homomorphism.

**Lemma C.2.4.** *Let $\phi : G \to H$ be an isomorphism between partial magmas $G$ and $H$. Then the following holds:*

*(a) $\phi^{-1} : H \to G$ is an isomorphism.*

*(b) If $\psi : H \to G$ satisfies $\phi(\psi(h)) = h$ for all $h \in H$ then $\psi = \phi^{-1}$, and hence $\psi$ is an isomorphism.*

*(c) If the composition in $G$ is associative then so is the composition in $H$. By associative we mean: if $g_1g_2$, $g_1g_3$, and $g_2g_3$ are defined then $(g_1g_2)g_3 = g_1(g_2g_3)$.*

*(d) If the composition in $G$ is commutative then so is the composition in $H$. By commutative we mean: if $g_1g_2$ is defined then $g_1g_2 = g_2g_1$.*

*Proof.*

(a) $\phi^{-1}$ is by definition bijective, so we need to show that it is an homomorphism. First, $h_1h_2 = \phi(g_1)\phi(g_2)$ is defined exactly when $g_1g_2 = \phi^{-1}(\phi(g_1))\phi^{-1}(\phi(g_2))$ is defined. Second,

$$\begin{aligned}
\phi^{-1}(h_1h_2) &= \phi^{-1}(\phi(g_1)\phi(g_2)) && (\phi \text{ is surjective})\\
&= \phi^{-1}(\phi(g_1g_2)) && (\phi \text{ is homomorphic})\\
&= g_1g_2 \\
&= \phi^{-1}(h_1)\phi^{-1}(h_2). && (\phi \text{ is injective})
\end{aligned}$$

(b) $\psi(h) = \phi^{-1}(\phi(\psi(h))) = \phi^{-1}(h)$.

(c) If $h_1h_2$, $h_1h_3$, and $h_2h_3$ are defined then with $\phi(g_i) = h_i$ we have that $g_1g_2$, $g_1g_3$, and $g_2g_3$ are defined because $\phi$ is a homomorphism. Now:

$$\begin{aligned} h_1(h_2h_3) &= \phi(g_1)\big(\phi(g_2)\phi(g_3)\big) \\ &= \phi(g_1)\phi(g_2g_3) && (\phi \text{ is homomorphic}) \\ &= \phi(g_1(g_2g_3)) && (\phi \text{ is homomorphic}) \\ &= \phi((g_1g_2)g_3) && (\text{associativity in } G) \\ &= \phi(g_1g_2)\phi(g_3) && (\phi \text{ is homomorphic}) \\ &= \big(\phi(g_1)\phi(g_2)\big)\phi(g_3) && (\phi \text{ is homomorphic}) \\ &= (h_1h_2)h_3. \end{aligned}$$

(d) If $h_1h_2$ is defined then with $\phi(g_i) = h_i$ we have that $g_1g_2$ is defined because $\phi$ is a homomorphism. Now:

$$\begin{aligned} h_1h_2 &= \phi(g_1)\phi(g_2) \\ &= \phi(g_1g_2) && (\phi \text{ is homomorphic}) \\ &= \phi(g_2g_1) && (\text{commutativity in } G) \\ &= \phi(g_2)\phi(g_1) && (\phi \text{ is homomorphic}) \\ &= h_2h_1. \end{aligned}$$

□

**Lemma C.2.5.** *Let $(G, \cdot)$ be a partial magma and let $R \subseteq G \times G$ be a congruence relation on $G$. That is, the following holds for all $g, g_1, g_2, g_3, g_4 \in G$:*

$$\begin{aligned} &(g, g) \in R, && (\text{reflexive}) \\ &\textit{if } (g_1, g_2) \in R \textit{ then } (g_2, g_1) \in R, && (\text{symmetric}) \\ &\textit{if } (g_1, g_2) \in R \textit{ and } (g_2, g_3) \in R \textit{ then } (g_1, g_3) \in R, && (\text{transitive}) \\ &\textit{if } (g_1, g_2) \in R \textit{ and } (g_3, g_4) \in R \textit{ then } (g_1g_3, g_2g_4) \in R, \end{aligned}$$

*where $(g_1g_3, g_2g_4) \in R$ means that either both $g_1g_3$ and $g_2g_4$ are undefined, or both are defined and related by $R$. Then $(G/R, \cdot)$ is a partial magma with*

$$[g_1][g_2] = [g_1g_2] \textit{ iff } g_1g_2 \textit{ is defined,}$$

*where $[g] = \{g' \in G \mid (g, g') \in R\}$.*

*Proof.* We must show that the composition is well-defined, that is if $[g_1] = [g_2]$ and $[g_3] = [g_4]$ then $[g_1][g_3] = [g_2][g_4]$. But this follows by definition from $R$ being a congruence relation. Next we must show that conditions (1) and (2) of Definition C.2.1 are fulfilled, but this follows from them being fulfilled for $G$. □

**Lemma C.2.6.** *Let $(G, \cdot)$ and $(H, \cdot)$ be partial magmas and let $R$ be a congruence relation on $G$. If $\phi : G \to H$ is a surjective homomorphism satisfying $\phi(g_1) = \phi(g_2)$ iff $(g_1, g_2) \in R$, then the induced function $\phi_R : G/R \to H$ defined by:*

$$\phi_R([g]) = \phi(g),$$

*is an isomorphism.*

*Proof.* We first note that $\phi_R$ is well-defined, as $\phi(g_1) = \phi(g_2)$ whenever $(g_1, g_2) \in R$. Since $\phi$ is surjective, so is $\phi_R$, and $\phi_R$ is injective since $(g_1, g_2) \in R$ whenever $\phi(g_1) = \phi(g_2)$. Finally, $\phi_R$ is homomorphic since $[g_1][g_2]$ is defined exactly when $g_1 g_2$ is defined, which is exactly when $\phi(g_1)\phi(g_2) = \phi_R([g_1])\phi_R([g_2])$ is defined. We therefore have that:

$$\phi_R([g_1][g_2]) = \phi_R([g_1 g_2]) = \phi(g_1 g_2) = \phi(g_1)\phi(g_2) = \phi_R([g_1])\phi_R([g_2]).$$

□

## C.3 Equivalence of Models

In order to show equivalence of the process model and the automaton model we show that the two models are isomorphic as partial magmas (Appendix C.2). The elements of the partial magmas are processes and automata respectively, and composition is in both cases parallel composition. That is, composition is only partial due to incompatible input channels or output channels.

**Lemma C.3.1.** $(\mathfrak{P}, \|)$ *defines a partial magma.*

*Proof.* The fact that $(\cdot \parallel \cdot) : \mathfrak{P} \times \mathfrak{P} \rightharpoonup \mathfrak{P}$ follows from Lemma 3.2.6, and the partiality of $\|$ is only due to incompatible channels, compare Lemma 3.2.6. We note that if $\mathsf{p}_1 \parallel \mathsf{p}_2$ is defined then so is $\mathsf{p}_2 \parallel \mathsf{p}_1$, hence condition (1) of Definition C.2.1 is satisfied. In order to show condition (2), assume that $\mathsf{p}_1 \parallel \mathsf{p}_2$, $\mathsf{p}_1 \parallel \mathsf{p}_3$, and $\mathsf{p}_2 \parallel \mathsf{p}_3$ are defined. This means that:

$$C_I^1 \cap C_I^2 = C_I^1 \cap C_I^3 = C_I^2 \cap C_I^3 = \emptyset,$$
$$C_O^1 \cap C_O^2 = C_O^1 \cap C_O^3 = C_O^2 \cap C_O^3 = \emptyset.$$

We therefore have that:

$$((C_I^1 \cup C_I^2) \setminus C_{\text{int}}^{1\|2}) \cap C_I^3 = \emptyset \qquad ((C_O^1 \cup C_O^2) \setminus C_{\text{int}}^{1\|2}) \cap C_O^3 = \emptyset,$$

where $C_{\text{int}}^{1\|2}$ are the internal channels of $\mathsf{p}_1 \parallel \mathsf{p}_2$ and hence $(\mathsf{p}_1 \parallel \mathsf{p}_2) \parallel \mathsf{p}_3$ is defined. □

**Lemma C.3.2.** $(\mathfrak{A}, \|)$ *defines a partial magma.*

*Proof.* The fact that $(\cdot \parallel \cdot) : \mathfrak{A} \times \mathfrak{A} \rightharpoonup \mathfrak{A}$ follows by construction, and the partiality of $\|$ is only due to incompatible input channels or output channels. Therefore, the proofs that conditions (1) and (2) of Definition C.2.1 are fulfilled are similar to the proofs in the previous lemma. □

### C.3.1 Automata as Processes

**Definition C.3.3.** Let $\mathsf{a} = (C_I, C_O, S, s_0, \delta_\text{o}, \delta_\text{t})$ be an automaton, and define the big-step relation $\boxed{t \vdash s, l_i \Downarrow l_o} \subseteq \mathbb{N} \times S \times \mathcal{L}_{C_I} \times \mathcal{L}_{C_O}$ by the two rules:

$$\textsf{e-end}\ \frac{\forall t' < \text{eol}(l_i).l_\varepsilon(t') = \varepsilon \qquad l_\varepsilon \in \mathcal{L}_{C_O}^{\text{eol}(l_i)}}{t \vdash s, l_i \Downarrow l_\varepsilon}\ (t \geq \text{eol}(l_i))$$

$$\textsf{e-step}\ \frac{\delta_{\mathrm{o}}(s) = m \qquad \delta_{\mathrm{t}}(s, l_i(t)) = s' \qquad t+1 \vdash s', l_i \Downarrow l_o}{t \vdash s, l_i \Downarrow l_o[t \mapsto m]}\ (t < \mathrm{eol}(l_i))$$

In the above, for a log $l \in \mathcal{L}_C^t$, $t' < t$, and $m \in \mathcal{M}_C$; $l[t' \mapsto m]$ denotes the log that is identical with $l$, except $l[t' \mapsto m](t') = m$.

**Lemma C.3.4.** *Big-step evaluation is total and deterministic, and whenever* $t \vdash s, l_i \Downarrow l_o$ *then* $\mathrm{eol}(l_i) = \mathrm{eol}(l_o)$.

*Proof.* Follows immediately from the definition (formally it is a proof by induction on the derivation). □

**Definition C.3.5.** We define the translation $\ulcorner \cdot \urcorner : \mathfrak{A} \to \mathfrak{P}$ by:

$$\ulcorner (C_I, C_O, S, s_0, \delta_{\mathrm{o}}, \delta_{\mathrm{t}}) \urcorner = (C_I, C_O, f),$$

where $f(l_i) = l_o$ iff $0 \vdash s_0, l_i, \Downarrow l_o$.

The definition above gives the expected process semantics to automata. We must, however, show that the translation is well-defined, that is we must show that $\ulcorner \mathsf{a} \urcorner$ defines a process for all automata $\mathsf{a}$.

**Theorem C.3.6.** *The translation of an automaton* $\mathsf{a} = (C_I, C_O, S, s_0, \delta_{\mathrm{o}}, \delta_{\mathrm{t}})$ *is a process.*

*Proof.* Let $\ulcorner \mathsf{a} \urcorner = (C_I, C_O, f)$. The conditions that $C_I$ and $C_O$ be finite and disjoint follow directly from $\mathsf{a}$ being an automaton. So we must show that $f$ is a log transformer from $\mathcal{L}_{C_I}$ to $\mathcal{L}_{C_O}$.

Lemma C.3.4 yields that $f$ represents a function from $\mathcal{L}_{C_I}$ to $\mathcal{L}_{C_O}$, so we need to show the following for all logs $l, l_1, l_2 \in \mathcal{L}_{C_I}$ and timestamps $t \in \mathbb{N}$ with $t < \min(\mathrm{eol}(l_1), \mathrm{eol}(l_2))$:

(i) $\mathrm{eol}(l) = \mathrm{eol}(f(l))$, and

(ii) if $l_{1|t} = l_{2|t}$ then $f(l_1)_{|t+1} = f(l_2)_{|t+1}$.

(i) Follows from Lemma C.3.4.

(ii) We show the following generalisation for all logs $l_1, l_2 \in \mathcal{L}_{C_I}$ and timestamps $t, t', t'' \in \mathbb{N}$ with $t' \leq t'' \leq t < \min(\mathrm{eol}(l_1), \mathrm{eol}(l_2))$:

$$\text{if } t' \vdash s, l_1 \Downarrow l_1' \text{ and } t' \vdash s, l_2 \Downarrow l_2' \text{ and } l_{1|t} = l_{2|t} \text{ then } l_1'(t'') = l_2'(t''). \quad \text{(C.18)}$$

The proof is by induction on the derivation of $t' \vdash s, l_1 \Downarrow l_1'$:

e-end: Now $t' < \min(\mathrm{eol}(l_1), \mathrm{eol}(l_2))$ and $t' \geq \mathrm{eol}(l_1)$ so the case is trivial.

e-step: Now the derivation of $t' \vdash s, l_1 \Downarrow l_1'$ has the form:

$$\frac{\delta_{\mathrm{o}}(s) = m \qquad \delta_{\mathrm{t}}(s, l_1(t')) = s' \qquad \overbrace{t'+1 \vdash s', l_1 \Downarrow l_1''}^{(*)}}{t' \vdash s, l_1 \Downarrow l_1''[t' \mapsto m]}\ (t' < \mathrm{eol}(l_1))$$

If $t' \geq \mathrm{eol}(l_2)$ then since $t' < \min(\mathrm{eol}(l_1), \mathrm{eol}(l_2))$ the case is trivial. So assume that $t' < \mathrm{eol}(l_2)$. Then the derivation of $t' \vdash s, l_2 \Downarrow l_2'$ must have used the `e-step` rule as well:

$$\frac{\delta_\mathrm{o}(s) = m \qquad \delta_\mathrm{t}(s, l_2(t')) = s'' \qquad \overbrace{t'+1 \vdash s'', l_2 \Downarrow l_2''}^{(**)}}{t' \vdash s, l_2 \Downarrow l_2''[t' \mapsto m]} \; (t' < \mathrm{eol}(l_2))$$

Now $l_1'(t') = l_2'(t') = m$ so in order to show (C.18) it suffices to show that $l_1'(t'') = l_2'(t'')$ for all $t''$ with $t'+1 \leq t'' \leq t$. If $t' = t$ then the result follows trivially, so assume that $t' < t$. Now $l_1' = l_1''[t' \mapsto m]$ and $l_2' = l_2''[t' \mapsto m]$ so the result will follow from the generalised induction hypothesis applied to $(*)$ and $(**)$ if we can show that $s' = s''$. But this is the case since $l_1(t') = l_2(t')$ as we assumed that $t' < t$. This concludes the proof of (C.18).

We can now apply the generalised induction hypothesis with $t' = 0$ to get that $f(l_1)(t'') = f(l_2)(t'')$ whenever $0 \leq t'' \leq t$ and $l_{1|t} = l_{2|t}$. But this is exactly the definition of strict monotonicity as required.

□

The following two lemmas show that bisimilarity in the automaton model coincides with equality in the process model under the translation $\ulcorner\cdot\urcorner$.

**Lemma C.3.7.** *If* $\mathsf{a}_1 \equiv \mathsf{a}_2$ *then* $\ulcorner \mathsf{a}_1 \urcorner = \ulcorner \mathsf{a}_2 \urcorner$.

*Proof.* Let $\mathsf{a}_1 = (C_I, C_O, S^1, s_0^1, \delta_\mathrm{o}^1, \delta_\mathrm{t}^1)$ and $\mathsf{a}_2 = (C_I, C_O, S^2, s_0^2, \delta_\mathrm{o}^2, \delta_\mathrm{t}^2)$ be given and assume that $\mathsf{a}_1 \equiv \mathsf{a}_2$. We then need to show that:

$$\text{if } 0 \vdash s_0^1, l \Downarrow l_1 \text{ and } 0 \vdash s_0^2, l \Downarrow l_2 \text{ then } l_1 = l_2.$$

We show the following more general result for all logs $l, l_1, l_2 \in \mathcal{L}_{C_I}$ and timestamps $t, t' \in \mathbb{N}$ with $t \leq t' < \mathrm{eol}(l)$:

$$\text{if } t \vdash s_1, l \Downarrow l_1 \text{ and } t \vdash s_2, l \Downarrow l_2 \text{ and } s_1 \equiv s_2 \text{ then } l_1(t') = l_2(t').$$

($s_1 \equiv s_2$ means that $(s_1, s_2) \in R$ for some bisimulation $R$.) The proof is by induction on the derivation of $t \vdash s_1, l \Downarrow l_1$.

`e-end`: Now $t > \mathrm{eol}(l)$ so the result follows trivially.

`e-step`: Now both derivations must have used the `e-step` rule:

$$\frac{\delta_\mathrm{o}^1(s_1) = m_1 \qquad \delta_\mathrm{t}(s_1, l(t)) = s_1' \qquad \overbrace{t+1 \vdash s_1', l \Downarrow l_1}^{(*)}}{t \vdash s_1, l \Downarrow l_1[t \mapsto m_1]} \; (t < \mathrm{eol}(l))$$

$$\frac{\delta_\mathrm{o}^2(s_2) = m_2 \qquad \delta_\mathrm{t}(s_2, l(t)) = s_2' \qquad \overbrace{t+1 \vdash s_2', l \Downarrow l_2}^{(**)}}{t \vdash s_2, l \Downarrow l_2[t \mapsto m_2]} \; (t < \mathrm{eol}(l))$$

By assumption, $s_1 \equiv s_2$ so $m_1 = m_2$. Hence it suffices to show that $l_1(t') = l_2(t')$ for all timestamps $t'$ with $t+1 \leq t' < \mathrm{eol}(l)$. But this follows by induction on $(*)$ and $(**)$ since $s_1 \equiv s_2$ implies that $s'_1 \equiv s'_2$.

□

**Lemma C.3.8.** *If* $\ulcorner \mathsf{a}_1 \urcorner = \ulcorner \mathsf{a}_2 \urcorner$ *then* $\mathsf{a}_1 \equiv \mathsf{a}_2$.

*Proof.* Let $\mathsf{a}_1 = (C_I, C_O, S^1, s^1_0, \delta^1_{\mathrm{o}}, \delta^1_{\mathrm{t}})$ and $\mathsf{a}_2 = (C_I, C_O, S^2, s^2_0, \delta^2_{\mathrm{o}}, \delta^2_{\mathrm{t}})$ be given and assume that $\ulcorner \mathsf{a}_1 \urcorner = \ulcorner \mathsf{a}_2 \urcorner$. We then need to construct a bisimulation $R \subseteq S^1 \times S^2$ such that $(s^1_0, s^2_0) \in R$. So consider the set:

$$R = \left\{(s_1, s_2) \mid \exists t \in \mathbb{N}. \forall l, l' \in \mathcal{L}_{C_I}. \left(t \vdash s_1, l \Downarrow l' \Leftrightarrow t \vdash s_2, l \Downarrow l'\right)\right\}.$$

We first show that $R$ is a bisimulation. So assume that $(s_1, s_2) \in R$ with some witness $t \in \mathbb{N}$. We then need to show that:

$$\delta^1_{\mathrm{o}}(s_1) = \delta^2_{\mathrm{o}}(s_2) \text{ and } (\delta^1_{\mathrm{t}}(s_1, m), \delta^2_{\mathrm{t}}(s_2, m)) \in R,$$

for all $m \in \mathcal{M}_{C_I}$.

Let $l \in \mathcal{L}_{C_I}$ be some log with $\mathrm{eol}(l) = t + 1$. Then $t \vdash s_1, l \Downarrow l'$ and $t \vdash s_2, l \Downarrow l'$ both using the e-step rule, and hence $\delta^1_{\mathrm{o}}(s_1) = l'(t) = \delta^2_{\mathrm{o}}(s_2)$ as needed.

We now need to show that $(\delta^1_{\mathrm{t}}(s_1, m), \delta^2_{\mathrm{t}}(s_2, m)) \in R$ for all $m \in \mathcal{M}_{C_I}$. That is, it suffices to show the following for all moves $m \in \mathcal{M}_{C_I}$:

$$\forall l, l' \in \mathcal{L}_{C_I}. \left(t + 1 \vdash \delta^1_{\mathrm{t}}(s_1, m), l \Downarrow l' \Leftrightarrow t + 1 \vdash \delta^2_{\mathrm{t}}(s_2, m), l \Downarrow l'\right). \qquad \text{(C.19)}$$

So let $l \in \mathcal{L}_{C_I}$ be given and assume that $t + 1 \vdash \delta^1_{\mathrm{t}}(s_1, m), l \Downarrow l'$ and $t + 1 \vdash \delta^2_{\mathrm{t}}(s_2, m), l \Downarrow l''$. We then need to show that $l' = l''$.

If $t + 1 \geq \mathrm{eol}(l)$ then $l' = l'' = l_\varepsilon$ so assume that $t + 1 < \mathrm{eol}(l)$. Now $(s_1, s_2) \in R$ with witness $t$, so for $l_m = l[t \mapsto m]$ we have that:

$$t \vdash s_1, l_m \Downarrow l''' \Leftrightarrow t \vdash s_2, l_m \Downarrow l'''.$$

Since $t + 1 < \mathrm{eol}(l)$ both derivations must have used the e-step rule:

$$\frac{\delta^1_{\mathrm{o}}(s_1) = m_1 \qquad \delta^1_{\mathrm{t}}(s_1, l_m(t)) = s'_1 \qquad \overbrace{t + 1 \vdash s'_1, l_m \Downarrow \hat{l}}^{(*)}}{t \vdash s_1, l_m \Downarrow \hat{l}[t \mapsto m_1]} \; (t < \mathrm{eol}(l_m))$$

$$\frac{\delta^2_{\mathrm{o}}(s_2) = m_2 \qquad \delta^2_{\mathrm{t}}(s_2, l_m(t)) = s'_2 \qquad \overbrace{t + 1 \vdash s'_2, l_m \Downarrow \hat{l}}^{(**)}}{t \vdash s_2, l_m \Downarrow \hat{l}[t \mapsto m_2]} \; (t < \mathrm{eol}(l_m))$$

But $l_m(t) = m$ so it follows that $\delta^1_{\mathrm{t}}(s_1, m) = s'_1$ and $\delta^2_{\mathrm{t}}(s_2, m) = s'_2$. We hence have that:

$$t + 1 \vdash \delta^1_{\mathrm{t}}(s_1, m), l_m \Downarrow \hat{l} \text{ and } t + 1 \vdash \delta^2_{\mathrm{t}}(s_2, m), l_m \Downarrow \hat{l}.$$

But $l(t') = l_m(t')$ for all $t' \geq t+1$, hence we can replace $l_m$ by $l$ in the relations above (formally a proof by induction on the derivation). That is, we have the following:

$$t + 1 \vdash \delta^1_{\mathrm{t}}(s_1, m), l \Downarrow \hat{l} \text{ and } t + 1 \vdash \delta^2_{\mathrm{t}}(s_2, m), l \Downarrow \hat{l}.$$

But then by the determinism of the big-step relation (Lemma C.3.4) it follows that $l' = \hat{l} = l''$, which concludes the proof of (C.19).

Now in order to conclude the lemma it suffices to show that $(s_0^1, s_0^2) \in R$. But this is the case since $\ulcorner \mathsf{a}_1 \urcorner = \ulcorner \mathsf{a}_2 \urcorner$ (that is witness 0). $\square$

The final lemma shows that the translation of automata to processes is compositional, meaning that whenever $\mathsf{a}_1 \parallel \mathsf{a}_2$ is defined, so is $\ulcorner \mathsf{a}_1 \urcorner \parallel \ulcorner \mathsf{a}_2 \urcorner$, and $\ulcorner \mathsf{a}_1 \parallel \mathsf{a}_2 \urcorner = \ulcorner \mathsf{a}_1 \urcorner \parallel \ulcorner \mathsf{a}_2 \urcorner$.

**Lemma C.3.9.** $\ulcorner \cdot \urcorner : (\mathfrak{A}, \parallel) \to (\mathfrak{P}, \parallel)$ *is a (partial magma) homomorphism.*

*Proof.* Let $\mathsf{a}_1 = (C_I^1, C_O^1, S^1, s_0^1, \delta_\mathrm{o}^1, \delta_\mathrm{t}^1)$ and $\mathsf{a}_2 = (C_I^2, C_O^2, S^2, s_0^2, \delta_\mathrm{o}^2, \delta_\mathrm{t}^2)$ be given. Since $\ulcorner \cdot \urcorner$ preserves input–output channels, $\mathsf{a}_1 \parallel \mathsf{a}_2$ is defined exactly when $\ulcorner \mathsf{a}_1 \urcorner \parallel \ulcorner \mathsf{a}_2 \urcorner$ is defined. So assume both are defined, and let $C_I$ and $C_O$ denote input channels and output channels respectively for the two compositions (which are the same compare Definition 3.2.4 and Definition 3.3.2).

Now, $\ulcorner \mathsf{a}_1 \parallel \mathsf{a}_2 \urcorner(l) = l'$ iff $0 \vdash \langle s_0^1, s_0^2 \rangle, l \Downarrow l'$ using the two rules:

$$\textsf{e-end}_{1\parallel 2}\ \frac{\forall t' < \mathrm{eol}(l). l_\varepsilon(t') = \varepsilon \qquad l_\varepsilon \in \mathcal{L}_{C_O}^{\mathrm{eol}(l)}}{t \vdash \langle s_1, s_2 \rangle, l \Downarrow l_\varepsilon}\ (t \geq \mathrm{eol}(l))$$

$$\textsf{e-step}_{1\parallel 2}\ \frac{\begin{array}{c}\delta_\mathrm{t}^1(s_1, (l(t) \cup \delta_\mathrm{o}^2(s_2))_{|C_I^1}) = s_1' \\ (\delta_\mathrm{o}^1(s_1) \cup \delta_\mathrm{o}^2(s_2))_{|C_O} = m \quad \delta_\mathrm{t}^2(s_2, (l(t) \cup \delta_\mathrm{o}^1(s_1))_{|C_I^2}) = s_2' \quad t+1 \vdash \langle s_1', s_2' \rangle, l \Downarrow l'\end{array}}{t \vdash \langle s_1, s_2 \rangle, l \Downarrow l'[t \mapsto m]}\ (t < \mathrm{eol}(l))$$

The translations of $\mathsf{a}_1$ and $\mathsf{a}_2$ use the rules $\textsf{e-end}_1$, $\textsf{e-step}_1$ and $\textsf{e-end}_2$, $\textsf{e-step}_2$ respectively:

$$\textsf{e-end}_1\ \frac{\forall t' < \mathrm{eol}(l). l_\varepsilon(t') = \varepsilon \qquad l_\varepsilon \in \mathcal{L}_{C_O}^{\mathrm{eol}(l)}}{t \vdash s_1, l \Downarrow l_\varepsilon}\ (t \geq \mathrm{eol}(l))$$

$$\textsf{e-step}_1\ \frac{\delta_\mathrm{o}^1(s_1) = m_1 \qquad \delta_\mathrm{t}^1(s_1, l(t)) = s_1' \qquad t+1 \vdash s_1', l \Downarrow l'}{t \vdash s_1, l \Downarrow l'[t \mapsto m_1]}\ (t < \mathrm{eol}(l))$$

$$\textsf{e-end}_2\ \frac{\forall t' < \mathrm{eol}(l). l_\varepsilon(t') = \varepsilon \qquad l_\varepsilon \in \mathcal{L}_{C_O}^{\mathrm{eol}(l)}}{t \vdash s_2, l \Downarrow l_\varepsilon}\ (t \geq \mathrm{eol}(l))$$

$$\textsf{e-step}_2\ \frac{\delta_\mathrm{o}^2(s_2) = m_2 \qquad \delta_\mathrm{t}^2(s_2, l(t)) = s_2' \qquad t+1 \vdash s_2', l \Downarrow l'}{t \vdash s_2, l \Downarrow l'[t \mapsto m_2]}\ (t < \mathrm{eol}(l))$$

So for $l \in \mathcal{L}_{C_I}$ we have that $(\ulcorner \mathsf{a}_1 \urcorner \parallel \ulcorner \mathsf{a}_2 \urcorner)(l) = (\mathcal{I}_N^1 \cup \mathcal{I}_N^2)_{|C_O}$ where:

$$\begin{array}{lll}\mathcal{I}_0^1(t)(c) = \varepsilon & 0 \vdash s_0^1, (\mathcal{I}_n^2 \cup l)_{|C_I^1} \Downarrow \mathcal{I}_{n+1}^1 & \text{(using } \textsf{e-end}_1 \text{ and } \textsf{e-step}_1\text{)} \\ \mathcal{I}_0^2(t)(c) = \varepsilon & 0 \vdash s_0^2, (\mathcal{I}_n^1 \cup l)_{|C_I^2} \Downarrow \mathcal{I}_{n+1}^2 & \text{(using } \textsf{e-end}_2 \text{ and } \textsf{e-step}_2\text{)},\end{array}$$

and $N$ is such that $\mathcal{I}_N^1 = \mathcal{I}_{N+1}^1$ and $\mathcal{I}_N^2 = \mathcal{I}_{N+1}^2$.

We show the following generalisation:

| | | |
|---|---|---|
| If | $t \vdash s_1, (l \cup A)_{\mid C_I^1} \Downarrow B$ | (using $\mathsf{e\text{-}end}_1$ and $\mathsf{e\text{-}step}_1$) |
| and | $t \vdash s_2, (l \cup C)_{\mid C_I^2} \Downarrow D$ | (using $\mathsf{e\text{-}end}_2$ and $\mathsf{e\text{-}step}_2$) |
| and | $t \vdash \langle s_1, s_2\rangle, l \Downarrow E$ | (using $\mathsf{e\text{-}end}_{1\parallel 2}$ and $\mathsf{e\text{-}step}_{1\parallel 2}$) |
| and | $\mathrm{eol}(A) = \mathrm{eol}(C) = \mathrm{eol}(l)$ | |
| and | $\forall t' \in \mathbb{N}. t \leq t' < \mathrm{eol}(l) \Rightarrow B(t') = C(t') \wedge D(t') = A(t')$ | |
| then | $\forall t' \in \mathbb{N}. t \leq t' < \mathrm{eol}(l) \Rightarrow E(t') = (B \cup D)_{\mid C_O}(t')$. | |

The proof is by induction on $n = \mathrm{eol}(l) - t$.

$n = 0$: In this case the result follows trivially.

$n > 0$: Now $\mathrm{eol}(l) = \mathrm{eol}((l \cup A)_{\mid C_I^1}) = \mathrm{eol}((l \cup C)_{\mid C_I^2}) > t$, so the three derivations must have used the rules $\mathsf{e\text{-}step}_1$, $\mathsf{e\text{-}step}_2$, and $\mathsf{e\text{-}step}_{1\parallel 2}$ respectively:

$$\frac{\delta^1_{\mathrm{o}}(s_1) = m_1 \qquad \delta^1_{\mathrm{t}}(s_1, (l \cup A)_{\mid C_I^1}(t)) = s_1' \qquad \overbrace{t+1 \vdash s_1', (l \cup A)_{\mid C_I^1} \Downarrow B'}^{(*)}}{t \vdash s_1, (l \cup A)_{\mid C_I^1} \Downarrow B'[t \mapsto m_1]}\ (t < \mathrm{eol}(l))$$

$$\frac{\delta^2_{\mathrm{o}}(s_2) = m_2 \qquad \delta^2_{\mathrm{t}}(s_2, (l \cup C)_{\mid C_I^2}(t)) = s_2' \qquad \overbrace{t+1 \vdash s_2', (l \cup C)_{\mid C_I^2} \Downarrow D'}^{(**)}}{t \vdash s_2, (l \cup C)_{\mid C_I^2} \Downarrow D'[t \mapsto m_2]}\ (t < \mathrm{eol}(l))$$

$$\frac{(\delta^1_{\mathrm{o}}(s_1) \cup \delta^2_{\mathrm{o}}(s_2))_{\mid C_O} = m \qquad \begin{array}{c}\delta^1_{\mathrm{t}}(s_1, (l(t) \cup \delta^2_{\mathrm{o}}(s_2))_{\mid C_I^1}) = s_1'' \\ \delta^2_{\mathrm{t}}(s_2, (l(t) \cup \delta^1_{\mathrm{o}}(s_1))_{\mid C_I^2}) = s_2''\end{array} \qquad \overbrace{t+1 \vdash \langle s_1'', s_2''\rangle, l \Downarrow E'}^{(***)}}{t \vdash \langle s_1, s_2\rangle, l \Downarrow E'[t \mapsto m]}\ (t < \mathrm{eol}(l))$$

We first show that $E(t) = (B \cup D)_{\mid C_O}(t)$:

$$\begin{aligned}(B \cup D)_{\mid C_O}(t) &= (B(t) \cup D(t))_{\mid C_O} \\ &= (m_1 \cup m_2)_{\mid C_O} \\ &= (\delta^1_{\mathrm{o}}(s_1) \cup \delta^2_{\mathrm{o}}(s_2))_{\mid C_O} \\ &= E(t).\end{aligned}$$

So in order to show the generalised induction hypothesis it suffices to show that $E(t') = (B \cup D)_{\mid C_O}(t')$ for all timestamps $t' \in \mathbb{N}$ with $t + 1 \leq t' < \mathrm{eol}(l)$. But this follows if we can show that:

$$\forall t' \in \mathbb{N}. t + 1 \leq t' < \mathrm{eol}(l) \Rightarrow E'(t') = (B' \cup D')_{\mid C_O}(t'),$$

which in turn follows from the induction hypothesis applied to $(*)$, $(**)$, and $(***)$ if we can show that (i) $\forall t'.\ t+1 \leq t' < \mathrm{eol}(l) \Rightarrow B'(t') = C(t') \wedge D'(t') = A(t')$ and (ii) $\langle s_1', s_2'\rangle = \langle s_1'', s_2''\rangle$.

(i) Follows from the assumption because $B'[t \mapsto m_1] = B$ and $D'[t \mapsto m_2] = A$.

(ii) We have that:

$$\begin{aligned}
s_1' &= \delta_{\mathrm{t}}^1(s_1, (l \cup A)_{|C_I^1}(t)) && \text{(by definition)}\\
&= \delta_{\mathrm{t}}^1(s_1, (l(t) \cup A(t))_{|C_I^1}) && \text{(by definition)}\\
&= \delta_{\mathrm{t}}^1(s_1, (l(t) \cup D(t))_{|C_I^1}) && \text{(by assumption)}\\
&= \delta_{\mathrm{t}}^1(s_1, (l(t) \cup \delta_{\mathrm{o}}^2(s_2))_{|C_I^1}) && \text{(by definition)}\\
&= s_1''. && \text{(by definition)}
\end{aligned}$$

By an analogous argument it can be shown that $s_2' = s_2''$ hence the generalised lemma follows.

The result of the lemma now follows from the generalised induction hypothesis with $A = \mathcal{I}_N^2$, $B = \mathcal{I}_{N+1}^1$, $C = \mathcal{I}_N^1$, and $D = \mathcal{I}_{N+1}^2$. □

## C.3.2 Processes as Automata

**Definition C.3.10.** We define the translation $\llcorner\cdot\lrcorner : \mathfrak{P} \to \mathfrak{A}$ by:

$$\llcorner(C_I, C_O, f)\lrcorner = (C_I, C_O, \mathcal{L}_{C_I}, l_\varepsilon \in \mathcal{L}_{C_I}^0, \delta_{\mathrm{o}}, \delta_{\mathrm{t}}),$$

where

$$\delta_{\mathrm{o}}(l) = f(l \mathbin{@} m_d)(\mathrm{eol}(l)) \qquad\qquad \delta_{\mathrm{t}}(l, m) = l \mathbin{@} m.$$

$l_\varepsilon$ denotes the empty log in $\mathcal{L}_{C_I}^0$; $m_d$ is any "dummy" move of $\mathcal{M}_{C_I}$; and for a log $l \in \mathcal{L}_C^t$ and a move $m \in \mathcal{M}_C$, $l \mathbin{@} m \in \mathcal{L}_C^{t+1}$ behaves like $l$ except $(l \mathbin{@} m)(t) = m$.

The intuition behind the definition is that the automaton keeps a trace of all that has happened so far in the state. The output is then determined by applying the log transformer to the input log extended with some move. The reason why we need to extend the input log is due to the condition (1) of Definition 3.2.2. Condition (2) will then guarantee that *any* extension will produce the same output, which makes the translation sound. Note also that we utilise the fact that the set of states for an automaton may be infinite, as $\mathcal{L}_{C_I}$ is infinite.

It will now be natural to show compositionality of the translation $\llcorner\cdot\lrcorner$, similar to Lemma C.3.9. However, we need not show this directly, as the result will follow automatically from the theory of partial magmas and the results about the two translations following in the next section.

## C.3.3 Equivalence

**Lemma C.3.11.** $\ulcorner\llcorner \mathsf{p} \lrcorner\urcorner = \mathsf{p}$ *for all processes* $\mathsf{p} \in \mathfrak{P}$.

*Proof.* Let $\mathsf{p} = (C_I, C_O, f) \in \mathfrak{P}$ be given. Then $\llcorner\mathsf{p}\lrcorner = (C_I, C_O, \mathcal{L}_{C_I}, l_\varepsilon, \delta_{\mathrm{o}}, \delta_{\mathrm{t}})$, where

$$\delta_{\mathrm{o}}(l) = f(l \mathbin{@} m_d)(\mathrm{eol}(l)) \qquad\qquad \delta_{\mathrm{t}}(l, m) = l \mathbin{@} m.$$

Hence $\ulcorner\llcorner\mathsf{p}\lrcorner\urcorner(l) = l'$ iff $0 \vdash l_\varepsilon, l \Downarrow l'$. So we need to show that $f(l) = l'$ iff $0 \vdash l_\varepsilon, l \Downarrow l'$. We show the following generalisation for all logs $l_1, l_2, l_3 \in \mathcal{L}_{C_I}$ and timestamps $t, t' \in \mathbb{N}$ with $t \leq t' < \mathrm{eol}(l_2)$:

$$\text{if } t \vdash l_1, l_2 \Downarrow l_3 \text{ and } l_1 = l_{2|t} \text{ then } l_3(t') = f(l_2)(t').$$

The proof is by induction on the derivation of $t \vdash l_1, l_2 \Downarrow l_3$.

**e-end:** Now $t \geq \mathrm{eol}(l_2)$ so the result follows trivially.

**e-step:** Now the derivation of $t \vdash l_1, l_2 \Downarrow l_3$ has the form:

$$\frac{\delta_{\mathrm{o}}(l_1) = m_1 \qquad \delta_{\mathrm{t}}(l_1, l_2(t)) = l_1' \qquad \overbrace{t+1 \vdash l_1', l_2 \Downarrow l_3'}^{(*)}}{t \vdash l_1, l_2 \Downarrow l_3'[t \mapsto m_1]} \; (t < \mathrm{eol}(l_2))$$

By assumption $l_1 = l_{2|t}$, hence $\mathrm{eol}(l_1) = t$. Now it follows from the definition of $\delta_{\mathrm{o}}$ that $m_1 = f(l_1 \mathbin{@} m_d)(t)$. Furthermore, $(l_1 \mathbin{@} m_d)_{|t} = l_{2|t}$ so by strict monotonicity it follows that $f(l_1 \mathbin{@} m_d)(t) = f(l_2)(t)$. So $m_1 = f(l_2)(t)$ and hence;

$$l_3'[t \mapsto m_1](t) = f(l_2)(t).$$

So it suffices to show that $l_3'(t') = f(l_2)(t')$ for all timestamps $t'$ with $t + 1 \leq t' < \mathrm{eol}(l_2)$. But this follows from the induction hypothesis applied to $(*)$ if we can show that $l_1' = l_{2|t+1}$. By definition of $\delta_{\mathrm{t}}$ we have that $l_1' = l_1 \mathbin{@} l_2(t)$, so since $l_1 = l_{2|t}$ the result follows.

The lemma now follows from the generalisation since from $0 \vdash l_\varepsilon, l \Downarrow l'$ we get that $l_\varepsilon = l_{|0}$. Hence $l'(t') = f(l)(t')$ for all timestamps $t'$ with $0 \leq t' < \mathrm{eol}(l)$, meaning exactly $l' = f(l)$ as desired. $\square$

It does not hold that $\llcorner\cdot\lrcorner$ is the left inverse of $\ulcorner\cdot\urcorner$, simply because $\ulcorner\cdot\urcorner$ is not injective. However, when we consider automata modulo bisimilarity then $\ulcorner\cdot\urcorner$ is an injection:

**Theorem C.3.12.** $\ulcorner\cdot\urcorner : (\mathfrak{A}/{\equiv}, \|) \to (\mathfrak{P}, \|)$ *defined by:*

$$\ulcorner[a]\urcorner = \ulcorner a\urcorner$$

*is an isomorphism.*

*Proof.* We know from Lemma C.3.9 that $\ulcorner\cdot\urcorner : (\mathfrak{A}, \|) \to (\mathfrak{P}, \|)$ is a homomorphism. Furthermore, this homomorphism is surjective by Lemma C.3.11 since $\ulcorner\llcorner \mathsf{p}\lrcorner\urcorner = \mathsf{p}$ for all processes $\mathsf{p} \in \mathfrak{P}$. It then follows from Lemma C.3.7 and Lemma C.3.8 that:

$$\mathsf{a}_1 \equiv \mathsf{a}_2 \text{ iff } \ulcorner\mathsf{a}_1\urcorner = \ulcorner\mathsf{a}_2\urcorner,$$

hence by Lemma C.2.6 ($\equiv$ is a congruence relation on $\mathfrak{A}$) the result follows. $\square$

We now get "for free" that $\llcorner\cdot\lrcorner$ is compositional, and the inverse of $\ulcorner\cdot\urcorner$:

**Corollary C.3.13.** $\llcorner\cdot\lrcorner : (\mathfrak{P}, \|) \to (\mathfrak{A}/{\equiv}, \|)$ *is an isomorphism with inverse isomorphism* $\ulcorner\cdot\urcorner$*, that is:*

$$\llcorner\cdot\lrcorner : (\mathfrak{P}, \|) \simeq (\mathfrak{A}/{\equiv}, \|) : \ulcorner\cdot\urcorner$$

*Proof.* By Theorem C.3.12 we know that $\ulcorner\cdot\urcorner : (\mathfrak{A}/{\equiv}, \|) \to (\mathfrak{P}, \|)$ is an isomorphism. We also know from Lemma C.3.11 that $\ulcorner\llcorner \mathsf{p}\lrcorner\urcorner = \mathsf{p}$ for all processes $\mathsf{p} \in \mathfrak{P}$, and hence by Lemma C.2.4 (b) that $\llcorner\cdot\lrcorner$ is the inverse of $\ulcorner\cdot\urcorner$. Finally Lemma C.2.4 (a) implies that $\llcorner\cdot\lrcorner$ is itself an isomorphism as required. $\square$

Having the equivalence result of Corollary C.3.13 means that we can automatically transfer the results from the process model to the automaton model and vice versa.

**Corollary C.3.14.** *Let* $\mathsf{p}_i = (C_I^i, C_O^i, f^i) \in \mathfrak{P}$ *be processes for* $i = 1, 2, 3$*, with:*

$$C_I^1 \cap C_I^2 = C_I^1 \cap C_I^3 = C_I^2 \cap C_I^3 = \emptyset,$$
$$C_O^1 \cap C_O^2 = C_O^1 \cap C_O^3 = C_O^2 \cap C_O^3 = \emptyset.$$

*Then* $\mathsf{p}_1 \parallel (\mathsf{p}_2 \parallel \mathsf{p}_3) = (\mathsf{p}_1 \parallel \mathsf{p}_2) \parallel \mathsf{p}_3$*.*

*Proof.* Follows from Lemma C.2.4 (c) and Lemma 3.3.5. □

**Corollary C.3.15.** *Let* $\mathsf{a}_1 = (C_I^1, C_O^1, S^1, s_0^1, \delta_\mathrm{o}^1, \delta_\mathrm{t}^1)$ *and* $\mathsf{a}_2 = (C_I^2, C_O^2, S^2, s_0^2, \delta_\mathrm{o}^2, \delta_\mathrm{t}^2)$ *be two automata with* $C_I^1 \cap C_I^2 = C_O^1 \cap C_O^2 = \emptyset$*. Then* $\mathsf{a}_1 \parallel \mathsf{a}_2 \equiv \mathsf{a}_2 \parallel \mathsf{a}_1$*.*

*Proof.* Follows from Lemma C.2.4 (d) and Lemma 3.2.7. We note that this proposition would be fairly easy to prove directly—but the proof illustrates that the equivalence works in both directions. □

# Appendix D

# Appendices for Chapter 5

## D.1 Parametric Compositional Data Types Examples

### D.1.1 From Names to PHOAS and back

```
--------------------------------------------------------------------------------
-- |
-- Module      :  Examples.Param.Names
-- Copyright   :  (c) 2011 Patrick Bahr, Tom Hvitved
-- License     :  BSD3
-- Maintainer  :  Tom Hvitved <hvitved@diku.dk>
-- Stability   :  experimental
-- Portability :  non-portable (GHC Extensions)
--
-- From names to parametric higher-order abstract syntax and back
--
-- The example illustrates how to convert a parse tree with explicit names into
-- an AST that uses parametric higher-order abstract syntax, and back again. The
-- example shows how we can easily convert object language binders to Haskell
-- binders, without having to worry about capture avoidance.
--
--------------------------------------------------------------------------------

module Examples.Param.Names where

import Data.Comp.Param hiding (Var)
import qualified Data.Comp.Param as P
import Data.Comp.Param.Derive
import Data.Comp.Param.Ditraversable
import Data.Comp.Param.Show ()
import Data.Maybe
import qualified Data.Map as Map
import Control.Monad.Reader

data Lam a b  = Lam (a → b)
data App a b  = App b b
data Lit a b  = Lit Int
data Plus a b = Plus b b
type Name     = String                -- The type of names
data NLam a b = NLam Name b
data NVar a b = NVar Name
type SigB     = App :+: Lit :+: Plus
type SigN     = NLam :+: NVar :+: SigB -- The name signature
type SigP     = Lam :+: SigB           -- The PHOAS signature

$(derive [makeDifunctor, makeShowD, makeEqD, smartConstructors]
         [''Lam, ''App, ''Lit, ''Plus, ''NLam, ''NVar])
$(derive [makeDitraversable]
         [''App, ''Lit, ''Plus, ''NLam, ''NVar])

--------------------------------------------------------------------------------
-- Names to PHOAS translation
--------------------------------------------------------------------------------

type M f a = Reader (Map.Map Name (Trm f a))

class N2PTrans f g where
  n2pAlg :: Alg f (M g a (Trm g a))

$(derive [liftSum] [''N2PTrans])

n2p :: (Difunctor f, N2PTrans f g) ⇒ Term f → Term g
n2p t = Term $ runReader (cata n2pAlg t) Map.empty

instance (Ditraversable f, f :<: g) ⇒ N2PTrans f g where
  n2pAlg = liftM inject . disequence . dimap (return . P.Var) id -- default

instance (Lam :<: g) ⇒ N2PTrans NLam g where
  n2pAlg (NLam x b) = do vars ← ask
                         return $ iLam $ λy → runReader b (Map.insert x y vars)

instance N2PTrans NVar g where
  n2pAlg (NVar x) = liftM fromJust (asks (Map.lookup x))

en :: Term SigN
en = Term $ iNLam "x1" $ iNLam "x2" (iNLam "x3" $ iNVar "x2") `iApp` iNVar "x1"

ep :: Term SigP
ep = n2p en

--------------------------------------------------------------------------------
-- PHOAS to names translation
--------------------------------------------------------------------------------

type M' = Reader [Name]

class P2NTrans f g where
  p2nAlg :: Alg f (M' (Trm g a))

$(derive [liftSum] [''P2NTrans])

p2n :: (Difunctor f, P2NTrans f g) ⇒ Term f → Term g
p2n t = Term $ runReader (cata p2nAlg t) ['x' : show n | n ← [1..]]

instance (Ditraversable f, f :<: g) ⇒ P2NTrans f g where
  p2nAlg = liftM inject . disequence . dimap (return . P.Var) id -- default

instance (NLam :<: g, NVar :<: g) ⇒ P2NTrans Lam g where
  p2nAlg (Lam f) = do n:names ← ask
                      return $ iNLam n (runReader (f (return $ iNVar n)) names)

ep' :: Term SigP
```

```
ep' = Term $ iLam $ λa → iLam (λb → (iLam $ λa → b)) ‘iApp‘ a

en' :: Term SigN
en' = p2n ep'
```

### D.1.2 First-Order Logic à la Carte

```
--------------------------------------------------------------------------------
-- |
-- Module      :  Examples.MultiParam.FOL
-- Copyright   :  (c) 2011 Patrick Bahr, Tom Hvitved
-- License     :  BSD3
-- Maintainer  :  Tom Hvitved <hvitved@diku.dk>
-- Stability   :  experimental
-- Portability :  non-portable (GHC Extensions)
--
-- First-Order Logic a la Carte
--
-- This example illustrates how to implement First-Order Logic a la Carte
-- (Knowles, The Monad.Reader Issue 11, '08) using Generalised Parametric
-- Compositional Data Types.
--
-- Rather than using a fixed domain 'Term' for binders as Knowles, our encoding
-- uses a mutually recursive data structure for terms and formulae. This makes
-- terms modular too, and hence we only introduce variables when they are
-- actually needed in stage 5.
--
--------------------------------------------------------------------------------

module Examples.MultiParam.FOL where

import Data.Comp.MultiParam hiding (Var)
import qualified Data.Comp.MultiParam as MP
import Data.Comp.MultiParam.Show ()
import Data.Comp.MultiParam.Derive
import Data.Comp.MultiParam.FreshM (Name, withName, evalFreshM)
import Data.List (intercalate)
import Data.Maybe
import Control.Monad.State
import Control.Monad.Reader

-- Phantom types indicating whether a (recursive) term is a formula or a term
data TFormula
data TTerm

-- Terms
data Const :: (∗ → ∗) → (∗ → ∗) → ∗ → ∗ where
    Const :: String → [e TTerm] → Const a e TTerm
data Var :: (∗ → ∗) → (∗ → ∗) → ∗ → ∗ where
    Var :: String → Var a e TTerm

-- Formulae
data TT :: (∗ → ∗) → (∗ → ∗) → ∗ → ∗ where
    TT :: TT a e TFormula
data FF :: (∗ → ∗) → (∗ → ∗) → ∗ → ∗ where
    FF :: FF a e TFormula
data Atom :: (∗ → ∗) → (∗ → ∗) → ∗ → ∗ where
    Atom :: String → [e TTerm] → Atom a e TFormula
data NAtom :: (∗ → ∗) → (∗ → ∗) → ∗ → ∗ where
```

```
    NAtom :: String → [e TTerm] → NAtom a e TFormula
data Not :: (∗ → ∗) → (∗ → ∗) → ∗ → ∗ where
    Not :: e TFormula → Not a e TFormula
data Or :: (∗ → ∗) → (∗ → ∗) → ∗ → ∗ where
    Or :: e TFormula → e TFormula → Or a e TFormula
data And :: (∗ → ∗) → (∗ → ∗) → ∗ → ∗ where
    And :: e TFormula → e TFormula → And a e TFormula
data Impl :: (∗ → ∗) → (∗ → ∗) → ∗ → ∗ where
    Impl :: e TFormula → e TFormula → Impl a e TFormula
data Exists :: (∗ → ∗) → (∗ → ∗) → ∗ → ∗ where
    Exists :: (a TTerm → e TFormula) → Exists a e TFormula
data Forall :: (∗ → ∗) → (∗ → ∗) → ∗ → ∗ where
    Forall :: (a TTerm → e TFormula) → Forall a e TFormula

$(derive [makeHDifunctor, smartConstructors]
         [''Const, ''Var, ''TT, ''FF, ''Atom, ''NAtom,
          ''Not, ''Or, ''And, ''Impl, ''Exists, ''Forall])

--------------------------------------------------------------------------------
-- (Custom) pretty printing of terms and formulae
--------------------------------------------------------------------------------

instance ShowHD Const where
  showHD (Const f t) = do ts ← mapM unK t
                          return $ f ++ "(" ++ intercalate ",␣" ts ++ ")"

instance ShowHD Var where
  showHD (Var x) = return x

instance ShowHD TT where
  showHD TT = return "true"

instance ShowHD FF where
  showHD FF = return "false"

instance ShowHD Atom where
  showHD (Atom p t) = do ts ← mapM unK t
                         return $ p ++ "(" ++ intercalate ",␣" ts ++ ")"

instance ShowHD NAtom where
  showHD (NAtom p t) = do ts ← mapM unK t
                          return $ "not␣" ++ p ++ "(" ++ intercalate ",␣" ts ++ ")"

instance ShowHD Not where
  showHD (Not (K f)) = liftM (λx → "not␣(" ++ x ++ ")") f

instance ShowHD Or where
  showHD (Or (K f1) (K f2)) =
      liftM2 (λx y → "(" ++ x ++ ")␣or␣(" ++ y ++ ")") f1 f2

instance ShowHD And where
  showHD (And (K f1) (K f2)) =
      liftM2 (λx y → "(" ++ x ++ ")␣and␣(" ++ y ++ ")") f1 f2

instance ShowHD Impl where
  showHD (Impl (K f1) (K f2)) =
      liftM2 (λx y → "(" ++ x ++ ")␣→␣(" ++ y ++ ")") f1 f2

instance ShowHD Exists where
```

```
  showHD (Exists f) =
      withName (λx → do b ← unK (f x)
                         return $ "exists␣" ++ show x ++ ".␣" ++ b)

instance ShowHD Forall where
  showHD (Forall f) =
      withName (λx → do b ← unK (f x)
                         return $ "forall␣" ++ show x ++ ".␣" ++ b)

--------------------------------------------------------------------------------
-- Stage 0
--------------------------------------------------------------------------------

type Input = Const :+:
             TT :+: FF :+: Atom :+: Not :+: Or :+: And :+: Impl :+:
             Exists :+: Forall

foodFact :: Term Input TFormula
foodFact = Term $
  iExists (λp → iAtom "Person" [p] `iAnd`
                iForall (λf → iAtom "Food" [f] `iImpl`
                              iAtom "Eats" [p,f])) `iImpl`
  iNot (iExists $ λf → iAtom "Food" [f] `iAnd`
                       iNot (iExists $ λp → iAtom "Person" [p] `iAnd`
                                            iAtom "Eats" [p,f]))

--------------------------------------------------------------------------------
-- Stage 1: Eliminate Implications
--------------------------------------------------------------------------------

type Stage1 = Const :+:
              TT :+: FF :+: Atom :+: Not :+: Or :+: And :+: Exists :+: Forall

class HDifunctor f ⇒ ElimImp f where
  elimImpHom :: Hom f Stage1

$(derive [liftSum] [''ElimImp])

elimImp :: Term Input :→ Term Stage1
elimImp (Term t) = Term (appHom elimImpHom t)

instance (HDifunctor f, f :<: Stage1) ⇒ ElimImp f where
  elimImpHom = simpCxt . inj

instance ElimImp Impl where
  elimImpHom (Impl f1 f2) = iNot (Hole f1) `iOr` (Hole f2)

foodFact1 :: Term Stage1 TFormula
foodFact1 = elimImp foodFact

--------------------------------------------------------------------------------
-- Stage 2: Move Negation Inwards
--------------------------------------------------------------------------------

type Stage2 = Const :+:
              TT :+: FF :+: Atom :+: NAtom :+: Or :+: And :+: Exists :+: Forall

class HDifunctor f ⇒ Dualize f where
  dualizeHom :: f a (Cxt h Stage2 a b) :→ Cxt h Stage2 a b
```

```
$(derive [liftSum] [''Dualize])

dualize :: Trm Stage2 a :→ Trm Stage2 a
dualize = appHom (dualizeHom . hfmap Hole)

instance Dualize Const where
  dualizeHom (Const f t) = iConst f t

instance Dualize TT where
  dualizeHom TT = iFF

instance Dualize FF where
  dualizeHom FF = iTT

instance Dualize Atom where
  dualizeHom (Atom p t) = iNAtom p t

instance Dualize NAtom where
  dualizeHom (NAtom p t) = iAtom p t

instance Dualize Or where
  dualizeHom (Or f1 f2) = f1 `iAnd` f2

instance Dualize And where
  dualizeHom (And f1 f2) = f1 `iOr` f2

instance Dualize Exists where
  dualizeHom (Exists f) = inject $ Forall f

instance Dualize Forall where
  dualizeHom (Forall f) = inject $ Exists f

class PushNot f where
  pushNotAlg :: Alg f (Trm Stage2 a)

$(derive [liftSum] [''PushNot])

pushNotInwards :: Term Stage1 :→ Term Stage2
pushNotInwards t = Term (cata pushNotAlg t)

instance (HDifunctor f, f :<: Stage2) ⇒ PushNot f where
  pushNotAlg = inject . hdimap MP.Var id -- default

instance PushNot Not where
  pushNotAlg (Not f) = dualize f

foodFact2 :: Term Stage2 TFormula
foodFact2 = pushNotInwards foodFact1

--------------------------------------------------------------------------------
-- Stage 4: Skolemization
--------------------------------------------------------------------------------

type Stage4 = Const :+:
              TT :+: FF :+: Atom :+: NAtom :+: Or :+: And :+: Forall

type Unique = Int
data UniqueSupply = UniqueSupply Unique UniqueSupply UniqueSupply
```

```
initialUniqueSupply :: UniqueSupply
initialUniqueSupply = genSupply 1
    where genSupply n = UniqueSupply n (genSupply (2 ∗ n))
                                       (genSupply (2 ∗ n + 1))

splitUniqueSupply :: UniqueSupply → (UniqueSupply, UniqueSupply)
splitUniqueSupply (UniqueSupply _ l r) = (l,r)

getUnique :: UniqueSupply → (Unique, UniqueSupply)
getUnique (UniqueSupply n l _) = (n,l)

type Supply = State UniqueSupply
type S a = ReaderT [Trm Stage4 a TTerm] Supply

evalS :: S a b → [Trm Stage4 a TTerm] → UniqueSupply → b
evalS m env = evalState (runReaderT m env)

fresh :: S a Int
fresh = do supply ← get
           let (uniq,rest) = getUnique supply
           put rest
           return uniq

freshes :: S a UniqueSupply
freshes = do supply ← get
             let (l,r) = splitUniqueSupply supply
             put r
             return l

class Skolem f where
  skolemAlg :: AlgM' (S a) f (Trm Stage4 a)

$(derive [liftSum] [''Skolem])

skolemize :: Term Stage2 :→ Term Stage4
skolemize f = Term (evalState (runReaderT (cataM' skolemAlg f) [])
                              initialUniqueSupply)

instance Skolem Const where
  skolemAlg (Const f t) = liftM (iConst f) $ mapM getCompose t

instance Skolem TT where
  skolemAlg TT = return iTT

instance Skolem FF where
  skolemAlg FF = return iFF

instance Skolem Atom where
  skolemAlg (Atom p t) = liftM (iAtom p) $ mapM getCompose t

instance Skolem NAtom where
  skolemAlg (NAtom p t) = liftM (iNAtom p) $ mapM getCompose t

instance Skolem Or where
  skolemAlg (Or (Compose f1) (Compose f2)) = liftM2 iOr f1 f2

instance Skolem And where
  skolemAlg (And (Compose f1) (Compose f2)) = liftM2 iAnd f1 f2
```

```
instance Skolem Forall where
  skolemAlg (Forall f) = do
    supply ← freshes
    xs ← ask
    return $ iForall $ λx → evalS (getCompose $ f x) (x : xs) supply

instance Skolem Exists where
  skolemAlg (Exists f) = do
    uniq ← fresh
    xs ← ask
    getCompose $ f (iConst ("Skol" ++ show uniq) xs)

foodFact4 :: Term Stage4 TFormula
foodFact4 = skolemize foodFact2

--------------------------------------------------------------------------------
-- Stage 5: Prenex Normal Form
--------------------------------------------------------------------------------

type Stage5 = Const :+: Var :+:
              TT :+: FF :+: Atom :+: NAtom :+: Or :+: And

class Prenex f where
  prenexAlg :: AlgM' (S a) f (Trm Stage5 a)

$(derive [liftSum] [''Prenex])

prenex :: Term Stage4 :→ Term Stage5
prenex f = Term (evalState (runReaderT (cataM' prenexAlg f) [])
                           initialUniqueSupply)

instance Prenex Const where
  prenexAlg (Const f t) = liftM (iConst f) $ mapM getCompose t

instance Prenex TT where
  prenexAlg TT = return iTT

instance Prenex FF where
  prenexAlg FF = return iFF

instance Prenex Atom where
  prenexAlg (Atom p t) = liftM (iAtom p) $ mapM getCompose t

instance Prenex NAtom where
  prenexAlg (NAtom p t) = liftM (iNAtom p) $ mapM getCompose t

instance Prenex Or where
  prenexAlg (Or (Compose f1) (Compose f2)) = liftM2 iOr f1 f2

instance Prenex And where
  prenexAlg (And (Compose f1) (Compose f2)) = liftM2 iAnd f1 f2

instance Prenex Forall where
  prenexAlg (Forall f) = do uniq ← fresh
                            getCompose $ f (iVar ('x' : show uniq))

foodFact5 :: Term Stage5 TFormula
foodFact5 = prenex foodFact4
```

```
--------------------------------------------------------------------------------
-- Stage 6: Conjunctive Normal Form
--------------------------------------------------------------------------------

type Literal a     = Trm (Const :+: Var :+: Atom :+: NAtom) a
newtype Clause a i = Clause {unClause :: [Literal a i]} -- implicit disjunction
newtype CNF a i    = CNF {unCNF :: [Clause a i]}        -- implicit conjunction

instance (HDifunctor f, ShowHD f) ⇒ Show (Trm f Name i) where
  show = evalFreshM . showHD . toCxt

instance Show (Clause Name i) where
  show c = intercalate "␣or␣" $ map show $ unClause c

instance Show (CNF Name i) where
  show c = intercalate "λn" $ map show $ unCNF c

class ToCNF f where
  cnfAlg :: f (CNF a) (CNF a) i → [Clause a i]

$(derive [liftSum] [''ToCNF])

cnf :: Term Stage5 :→ CNF a
cnf = cata (CNF . cnfAlg)

instance ToCNF Const where
  cnfAlg (Const f t) =
      [Clause [iConst f (map (head . unClause . head . unCNF) t)]]

instance ToCNF Var where
  cnfAlg (Var x) = [Clause [iVar x]]

instance ToCNF TT where
  cnfAlg TT = []

instance ToCNF FF where
  cnfAlg FF = [Clause []]

instance ToCNF Atom where
  cnfAlg (Atom p t) =
      [Clause [iAtom p (map (head . unClause . head . unCNF) t)]]

instance ToCNF NAtom where
  cnfAlg (NAtom p t) =
      [Clause [iNAtom p (map (head . unClause . head . unCNF) t)]]

instance ToCNF And where
  cnfAlg (And f1 f2) = unCNF f1 ++ unCNF f2

instance ToCNF Or where
  cnfAlg (Or f1 f2) =
      [Clause (x ++ y) | Clause x ← unCNF f1, Clause y ← unCNF f2]

foodFact6 :: CNF a TFormula
foodFact6 = cnf foodFact5
```

```
--------------------------------------------------------------------------------
-- Stage 7: Implicative Normal Form
--------------------------------------------------------------------------------

type T              = Const :+: Var :+: Atom :+: NAtom
newtype IClause a i = IClause ([Trm T a i], -- implicit conjunction
                               [Trm T a i]) -- implicit disjunction
newtype INF a i     = INF [IClause a i]     -- implicit conjunction

instance Show (IClause Name i) where
  show (IClause (cs,ds)) = let cs' = intercalate "␣and␣" $ map show cs
                               ds' = intercalate "␣or␣" $ map show ds
                           in "(" ++ cs' ++ ")␣→␣(" ++ ds' ++ ")"

instance Show (INF Name i) where
  show (INF fs) = intercalate "\n" $ map show fs

inf :: CNF a TFormula → INF a TFormula
inf (CNF f) = INF $ map (toImpl . unClause) f
    where toImpl :: [Literal a TFormula] → IClause a TFormula
          toImpl disj = IClause ([iAtom p t | NAtom p t ← mapMaybe proj1 disj],
                                 [inject t | t ← mapMaybe proj2 disj])
          proj1 :: NatM Maybe (Trm T a) (NAtom a (Trm T a))
          proj1 = project
          proj2 :: NatM Maybe (Trm T a) (Atom a (Trm T a))
          proj2 = project

foodFact7 :: INF a TFormula
foodFact7 = inf foodFact6
```

# Appendix E

# Appendices for Chapter 6

## E.1 Predefined Ontology

### E.1.1 Data

*Data is abstract.*

### E.1.2 Event

*Event is abstract.*
*Event has a Timestamp*
*called internalTimeStamp.*

***# Add data definitions to the system***
*AddDataDefs is an Event.*
*AddDataDefs has a String called defs.*

***# Events associated with entities***
*EntityEvent is an Event.*
*EntityEvent is abstract.*
*EntityEvent has a Data entity called ent.*

***# Put entity event***
*PutEntity is an EntityEvent.*
*PutEntity has Data.*
*PutEntity is abstract.*

***# Create entity event***
*CreateEntity is a PutEntity.*
*CreateEntity has a String called recordType.*

***# Update entity event***
*UpdateEntity is a PutEntity.*

***# Delete entity event***
*DeleteEntity is an EntityEvent.*

***# Events associated with a report definition***
*ReportEvent is an Event.*
*ReportEvent has a String called name.*

***# Put report definition event***
*PutReport is a ReportEvent.*
*PutReport is abstract.*
*PutReport has a String called code.*
*PutReport has a String called description.*
*PutReport has a list of String called tags.*

***# Create report definition event***
*CreateReport is a PutReport.*

***# Update report definition event***
*UpdateReport is a PutReport.*

***# Delete report definition event***
*DeleteReport is a ReportEvent.*

***# Events associated with a contract template***
*ContractDefEvent is an Event.*
*ContractDefEvent has a String called name.*

***# Put contract template event***
*PutContractDef is a ContractDefEvent.*
*PutContractDef is abstract.*
*PutContractDef has a String called recordType.*
*PutContractDef has a String called code.*
*PutContractDef has a String called description.*

***# Create contract template event***
*CreateContractDef is a PutContractDef.*

***# Update contract template event***
*UpdateContractDef is a PutContractDef.*

***# Delete contract template event***
*DeleteContractDef is a ContractDefEvent.*

***# Events associated with a contract***
*ContractEvent is an Event.*
*ContractEvent is abstract.*
*ContractEvent has an Int called contractId.*

***# Put contract event***
*PutContract is a ContractEvent.*
*PutContract has a Contract.*

*PutContract is abstract.*

***# Create contract event***
*CreateContract is a PutContract.*

***# Update contract event***
*UpdateContract is a PutContract.*

***# Conclude contract event***
*ConcludeContract is a ContractEvent.*

***# Transaction super class***
*TransactionEvent is a ContractEvent.*
*TransactionEvent has a Timestamp.*
*TransactionEvent has a Transaction.*

### E.1.3 Transaction

*Transaction is abstract.*

### E.1.4 Report

*Report is abstract.*

### E.1.5 Contract

*Contract is abstract.*
*Contract has a Timestamp called startDate.*
*Contract has a String called templateName.*

## E.2 $\mu$ERP Specification

### E.2.1 Ontology

#### E.2.1.1 Data

*ResourceType is Data.*
*ResourceType is abstract.*

*Currency is a ResourceType.*
*Currency is abstract.*

*DKK is a Currency.*
*EUR is a Currency.*

*ItemType is a ResourceType.*
*ItemType is abstract.*

*Bicycle is an ItemType.*
*Bicycle has a String called model.*

*Resource is Data.*
*Resource is abstract.*

*Money is a Resource.*
*Money has a Currency.*
*Money has a Double called amount.*

*Item is a Resource.*
*Item has an ItemType.*
*Item has a Double called quantity.*

*Agent is Data.*

*Me is an Agent.*

*Customer is an Agent.*
*Customer has a String called name.*
*Customer has an Address.*

*Vendor is an Agent.*

*Vendor has a String called name.*
*Vendor has an Address.*

*Address is Data.*
*Address has a String.*
*Address has a Country.*

*Country is Data.*
*Country is abstract.*

*Denmark is a Country.*

*OrderLine is Data.*
*OrderLine has an Item.*
*OrderLine has Money called unitPrice.*
*OrderLine has a Double called vatPercentage.*

*CurrentAssets is Data.*
*CurrentAssets has a list of Money called currentAssets.*
*CurrentAssets has a list of Money called inventory.*
*CurrentAssets has a list of Money called accountsReceivable.*
*CurrentAssets has a list of Money called cashPlusEquiv.*

*Liabilities is Data.*
*Liabilities has a list of Money called liabilities.*
*Liabilities has a list of Money called accountsPayable.*
*Liabilities has a list of Money called vatPayable.*

*Invoice is Data.*
*Invoice has an Agent called sender.*
*Invoice has an Agent called receiver.*
*Invoice has a list of OrderLine called orderLines.*

*UnpaidInvoice is Data.*
*UnpaidInvoice has an Invoice.*
*UnpaidInvoice has a list of Money called remainder.*

*CustomerStatistics is Data.*
*CustomerStatistics has a Customer entity.*
*CustomerStatistics has Money called totalPaid.*

#### E.2.1.2 Transaction

*BiTransaction is a Transaction.*
*BiTransaction is abstract.*
*BiTransaction has an Agent entity called sender.*
*BiTransaction has an Agent entity called receiver.*

*Transfer is a BiTransaction.*
*Transfer is abstract.*

*Payment is a Transfer.*
*Payment is abstract.*
*Payment has Money.*

*CashPayment is a Payment.*
*CreditCardPayment is a Payment.*
*BankTransfer is a Payment.*

*Delivery is a Transfer.*
*Delivery has a list of Item called items.*

*IssueInvoice is a BiTransaction.*
*IssueInvoice has a list of OrderLine called orderLines.*

*RequestRepair is a BiTransaction.*
*RequestRepair has a list of Item called items.*

*Repair is a BiTransaction.*
*Repair has a list of Item called items.*

#### E.2.1.3 Report

*IncomeStatement is a Report.*
*IncomeStatement has a list of Money called revenue.*
*IncomeStatement has a list of Money called costOfGoodsSold.*
*IncomeStatement has a list of Money called contribMargin.*
*IncomeStatement has a list of Money called fixedCosts.*
*IncomeStatement has a list of Money called depreciation.*
*IncomeStatement has a list of Money called netOpIncome.*

*BalanceSheet is a Report.*
*BalanceSheet has a list of Money called fixedAssets.*
*BalanceSheet has CurrentAssets.*
*BalanceSheet has a list of Money called totalAssets.*
*BalanceSheet has Liabilities.*
*BalanceSheet has a list of Money called ownersEquity.*
*BalanceSheet has a list of Money called totalLiabilitiesPlusEquity.*

*CashFlowStatement is a Report.*
*CashFlowStatement has a list of Payment called expenses.*
*CashFlowStatement has a list of Payment called revenues.*
*CashFlowStatement has a list of Money called revenueTotal.*
*CashFlowStatement has a list of Money called expenseTotal.*

*UnpaidInvoices is a Report.*
*UnpaidInvoices has a list of UnpaidInvoice called invoices.*

*VATReport is a Report.*
*VATReport has a list of Money called outgoingVAT.*
*VATReport has a list of Money called incomingVAT.*
*VATReport has a list of Money called vatDue.*

*Inventory is a Report.*
*Inventory has a list of Item called availableItems.*

*TopNCustomers is a Report.*
*TopNCustomers has a list of CustomerStatistics.*

#### E.2.1.4 Contract

*Purchase is a Contract.*
*Purchase has a Vendor entity.*
*Purchase has a list of OrderLine called orderLines.*

*Sale is a Contract.*
*Sale has a Customer entity.*
*Sale has a list of OrderLine called orderLines.*

### E.2.2 Reports

#### E.2.2.1 Prelude Functions

$-- $ ***Arithmetic***
$\mathit{min} : (\mathbf{Ord}\ a) \Rightarrow a \rightarrow a \rightarrow a$
$\mathit{min}\ x\ y = \mathbf{if}\ x < y\ \mathbf{then}\ x\ \mathbf{else}\ y$

$\mathit{max} : (\mathbf{Ord}\ a) \Rightarrow a \rightarrow a \rightarrow a$
$\mathit{max}\ x\ y = \mathbf{if}\ x > y\ \mathbf{then}\ x\ \mathbf{else}\ y$

$-- $ ***List functions***
$\mathit{null} : [a] \rightarrow \mathbf{Bool}$
$\mathit{null} = \mathbf{fold}\ (\lambda e\ r \rightarrow \mathbf{False})\ \mathbf{True}$

$\mathit{first} : a \rightarrow [a] \rightarrow a$

```
first = fold (λx a → x)

head : [a] → a
head = first (error "'head' applied to empty list")

elemBy : (a → a → Bool) → a → [a] → Bool
elemBy f e = fold (λx a → a ∨ f x e) False

elem : (Ord a) ⇒ a → [a] → Bool
elem = elemBy (≡)

sum : (a < Double, Int < a) ⇒ [a] → a
sum = fold (+) 0

length : [a] → Int
length = fold (λ x y → y+1) 0

map : (a → b) → [a] → [b]
map f = fold (λx a → (f x) # a) []

filter : (a → Bool) → [a] → [a]
filter f = fold (λx a → if f x then x # a else a) []

nubBy : (a → a → Bool) → [a] → [a]
nubBy f = fold (λx a → x # filter (λ y → ¬ (f x y)) a) []

nub : (Ord a) ⇒ [a] → [a]
nub = nubBy (≡)

all : (a → Bool) → [a] → Bool
all f = fold (λx a → f x ∧ a) True

any : (a → Bool) → [a] → Bool
any f = fold (λx a → f x ∨ a) False

concat : [[a]] → [a]
concat = fold (λx a → x ++ a) []

concatMap : (a → [b]) → [a] → [b]
concatMap f l = concat (map f l)

take : Int → [a] → [a]
take n l = (fold (λx a → if a.2 > 0 then (x # a.1,a.2 − 1) else a) ([],n) l).1

−− Grouping functions
addGroupBy : (a → a → Bool) → a → [[a]] → [[a]]
addGroupBy f a ll =
  let felem l = fold (λ el r → f el a) False l
      run el r =
        if r.1 then (True,el # r.2)
        else if felem el then (True, (a # el) # r.2)
        else (False, el # r.2)
      res = fold run (False,[]) ll
  in if res.1 then res.2 else [a] # res.2

groupBy : (a → a → Bool) → [a] → [[a]]
groupBy f = fold (addGroupBy f) []

addGroupProj : (Ord b) ⇒ (a → b) → a → [(b,[a])] → [(b,[a])]
addGroupProj f a ll =
  let run el r =
        if r.1 then(True,el # r.2)
        else if el.1 ≡ f a then (True, (el.1,a # el.2) # r.2)
        else (False, el # r.2)
      res = fold run (False,[]) ll
  in if res.1 then res.2 else (f a,[a]) # res.2

groupProj : (Ord b) ⇒ (a → b) → [a] → [(b, [a])]
groupProj f = fold (addGroupProj f) []


-- Sorting functions
insertBy : (a → a → Bool) → a → [a] → [a]
insertBy le a l =
  let ins e r =
        if r.1 then (True, e # r.2)
        else if le e a then (True,e # a # r.2)
        else (False, e # r.2)
      res = fold ins (False,[]) l
  in if res.1 then res.2 else a # res.2

insertProj : (Ord b) ⇒ (a → b) → a → [a] → [a]
insertProj proj = insertBy (λx y → proj x ≤ proj y)

insert : (Ord a) ⇒ a → [a] → [a]
insert = insertBy (≤)

sortBy : (a → a → Bool) → [a] → [a]
sortBy le = fold (λe r → insertBy le e r) []

sortProj : (Ord b) ⇒ (a → b) → [a] → [a]
sortProj proj = sortBy (λx y → proj x ≤ proj y)

sort : (Ord a) ⇒ [a] → [a]
sort = sortBy (≤)


-- Generators for 'lifecycled' data
reports : [PutReport]
reports = nubBy (λpr1 pr2 → pr1.name ≡ pr2.name) [pr |
  cr : CreateReport ← events,
  pr : PutReport = first cr [ur | ur : ReportEvent ← events, ur.name ≡ cr.name]]

entities : [(⟨Data⟩,String)]
entities = [(ce.ent,ce.recordType) |
  ce : CreateEntity ← events,
  null [de | de : DeleteEntity ← events, de.ent ≡ ce.ent]]

contracts : [PutContract]
contracts = [pc |
  cc : CreateContract ← events,
  pc = first cc [uc | uc : UpdateContract ← events, uc.contractId ≡ cc.contractId],
  null [cc | cc : ConcludeContract ← events, cc.contractId ≡ pc.contractId]]

contractDefs : [PutContractDef]
contractDefs = nubBy (λpcd1 pcd2 → pcd1.name ≡ pcd2.name) [pcd |
  ccd : CreateContractDef ← events,
  pcd : PutContractDef = first ccd [ucd | ucd : ContractDefEvent ← events, ucd.name ≡ ccd.name]]

transactionEvents : [TransactionEvent]
transactionEvents = [tr | tr : TransactionEvent ← events]

transactions : [Transaction]
transactions = [tr.transaction | tr ← transactionEvents]
```

#### E.2.2.2 Domain-Specific Prelude Functions

```
-- Check if an agent is the company itself
isMe : ⟨Agent⟩ → Bool
isMe a = a :? ⟨Me⟩

-- Normalise a list of money by grouping currencies together
normaliseMoney : [Money] → [Money]
normaliseMoney ms = [Money{currency = m.1, amount = sum (map (λm → m.amount) m.2)} |
  m ← groupProj (λm → m.currency) ms]
```

```
-- Add one list of money from another
addMoney : [Money] → [Money] → [Money]
addMoney m1 m2 = normaliseMoney (m1 ++ m2)

-- Subtract one list of money from another
subtractMoney : [Money] → [Money] → [Money]
subtractMoney m1 m2 = addMoney m1 (map (λm → m{amount = 0 − m.amount}) m2)

-- Produce normalised list of all items given in list
normaliseItems : [Item] → [Item]
normaliseItems its = [Item{itemType = i.1, quantity = sum (map (λis → is.quantity) i.2)} |
  i ← groupProj (λis → is.itemType) its]

-- List of all invoices and their associated contract ID
invoices : [(Int,IssueInvoice)]
invoices = [(tr.contractId,inv) |
  tr ← transactionEvents,
  inv : IssueInvoice = tr.transaction]

-- List of all received invoices and their associated contract ID
invoicesReceived : [(Int,IssueInvoice)]
invoicesReceived =
  filter (λinv → ¬ (isMe (inv.2).sender) ∧ isMe (inv.2).receiver) invoices

-- List of all sent invoices and their associated contract ID
invoicesSent : [(Int,IssueInvoice)]
invoicesSent = filter (λinv → isMe inv.2.sender ∧ ¬ (isMe inv.2.receiver)) invoices

-- Calculate the total price including VAT on an invoice
invoiceTotal : (a.orderLines : [OrderLine]) ⇒ a → [Money]
invoiceTotal inv = normaliseMoney [line.unitPrice{amount = price} |
  line ← inv.orderLines,
  quantity = line.item.quantity,
  price = ((100 + line.vatPercentage) × line.unitPrice.amount × quantity) / 100]

-- List of all items delivered to the company
itemsReceived : [Item]
itemsReceived = normaliseItems [is |
  tr ← transactionEvents,
  del : Delivery = tr.transaction,
  ¬(isMe del.sender) ∧ isMe del.receiver,
  is ← del.items]

-- List of all items that have been sold
itemsSold : [Item]
itemsSold = normaliseItems [line.item | inv ← invoicesSent, line ← inv.2.orderLines]

-- Inventory acquisitions, that is a list of all received items and the unit
-- price of each item, exluding VAT.
invAcq : [(Item,Money)]
invAcq = [(item,line.unitPrice) |
  inv ← invoicesReceived,
  tr ← transactionEvents,
  tr.contractId ≡ inv.1,
  deliv : Delivery = tr.transaction,
  item ← deliv.items,
  line ← inv.2.orderLines,
  line.item.itemType ≡ item.itemType]

-- FIFO costing: Calculate the cost of all sold goods based on FIFO costing.
fifoCost : [Money]
fifoCost = let
   -- Check whether a set of items equals the current set of items in the
   -- inventory. If so, 'take' as many of the inventory items as possible
   -- and add the price of these items to the totals.
   checkInventory y x = let
       invItem = y.1  -- The current item in the inventory
       invPrice = y.2 -- The price of the current item in the inventory
```

```
      oldInv = x.1 -- The part of the inventory that has been processed
      item = x.2 -- The item to find in the inventory
      total = x.3 -- The total costs so far
    in
    if item.itemType ≡ invItem.itemType then let
        deltaInv =
          if invItem.quantity ≤ item.quantity then
            []
          else
            [(invItem{quantity = invItem.quantity − item.quantity},invPrice)]
        remainingItem = item{quantity = max 0 (item.quantity − invItem.quantity)}
        price = invPrice{amount = invPrice.amount × (min item.quantity invItem.quantity)}
      in
      (oldInv ++ deltaInv, remainingItem, price # total)
    else
      (oldInv ++ [(invItem,invPrice)], item, total)

  -- Process a sold item
  processSoldItem soldItem x = let
      total = x.1 -- the total costs so far
      inv = x.2   -- the remaning inventory so far
      y = fold checkInventory ([],soldItem,total) inv
    in
    (y.3,y.1)
  in
  normaliseMoney ((fold processSoldItem ([],invAcq) itemsSold).1)

-- Outoing VAT
vatOutgoing : [Money]
vatOutgoing = normaliseMoney [price |
  inv ← invoicesReceived,
  l ← inv.2.orderLines,
  price = l.unitPrice{amount = (l.vatPercentage × l.unitPrice.amount × l.item.quantity) / 100}]

-- Incoming VAT
vatIncoming : [Money]
vatIncoming = normaliseMoney [price |
  inv ← invoicesSent,
  l ← inv.2.orderLines,
  price = l.unitPrice{amount = (l.vatPercentage × l.unitPrice.amount × l.item.quantity) / 100}]
```

#### E.2.2.3 Internal Reports

### Me

**name**: *Me*
**description**:
*Returns the pseudo entity 'Me' that represents the company.*
**tags**: *internal*, *entity*

```
report : ⟨Me⟩
report = head [me | me : ⟨Me⟩ ← map (λe → e.1) entities]
```

### Entities

**name**: *Entities*
**description**:
*A list of all entities.*
**tags**: *internal*, *entity*

```
report : [⟨Data⟩]
report = map (λe → e.1) entities
```

### EntitiesByType

**name**: *EntitiesByType*
**description**:
*A list of all entities with the given type.*
**tags**: *internal*, *entity*

$\textbf{report} : \textbf{String} \rightarrow [\langle\mathsf{Data}\rangle]$
$\textbf{report}\ t = map\ (\lambda e \rightarrow e.1)\ (filter\ (\lambda e \rightarrow e.2 \equiv t)\ entities)$

## ReportNames

**name**: *ReportNames*
**description**:
*A list of names of all registered reports.*
**tags**: *internal*, *report*

$\textbf{report} : [\textbf{String}]$
$\textbf{report} = [r.name \mid r \leftarrow reports]$

## ReportNamesByTags

**name**: *ReportNamesByTags*
**description**:
*A list of reports that have the all* **tags** *provided as first argument to the function and none of the* **tags** *provided as second argument.*
**tags**: *internal*, *report*

$filt\ allOf\ noneOf\ rep =$
$\quad all\ (\lambda x \rightarrow elem\ x\ rep.tags)\ allOf \wedge$
$\quad \neg\ (any\ (\lambda x \rightarrow elem\ x\ rep.tags)\ noneOf)$

$\textbf{report} : [\textbf{String}] \rightarrow [\textbf{String}] \rightarrow [\textbf{String}]$
$\textbf{report}\ allOf\ noneOf = [r.name \mid r \leftarrow filter\ (filt\ allOf\ noneOf)\ reports]$

## ReportTags

**name**: *ReportTags*
**description**:
*A list of* **tags** *that are used in registered reports.*
**tags**: *internal*, *report*

$\textbf{report} : [\textbf{String}]$
$\textbf{report} = nub\ (concatMap\ (\lambda x \rightarrow x.tags)\ reports)$

## ContractTemplates

**name**: *ContractTemplates*
**description**:
*A list of 'PutContractDef events for each non−deleted contract template.*
**tags**: *internal*, *contract*

$\textbf{report} : [\mathsf{PutContractDef}]$
$\textbf{report} = contractDefs$

## ContractTemplatesByType

**name**: *ContractTemplatesByType*
**description**:
*A list of 'PutContractDef events for each non−deleted contract template of the given type.*
**tags**: *internal*, *contract*

$\textbf{report} : \textbf{String} \rightarrow [\mathsf{PutContractDef}]$
$\textbf{report}\ r = filter\ (\lambda x \rightarrow x.recordType \equiv r)\ contractDefs$

### Contracts

**name**: *Contracts*
**description**:
*A list of all running (i.e. non−concluded) contracts.*
**tags**: *internal*, *contract*

```
report : [PutContract]
report = contracts
```

### ContractHistory

**name**: *ContractHistory*
**description**:
*A list of previous transactions for the given contract.*
**tags**: *internal*, *contract*

```
report : Int → [TransactionEvent]
report cid = [transaction |
  transaction : TransactionEvent ← events,
  transaction.contractId ≡ cid]
```

### ContractSummary

**name**: *ContractSummary*
**description**:
*A list of meta data for the given contract.*
**tags**: *internal*, *contract*

```
report : Int → [PutContract]
report cid = [createCon |
  createCon : PutContract ← contracts,
  createCon.contractId ≡ cid]
```

## E.2.2.4 External Reports

### IncomeStatement

**name**: *IncomeStatement*
**description**:
*The Income Statement.*
**tags**: *external*, *financial*

```
−− Revenue
revenue = normaliseMoney [line.unitPrice{amount = amount} |
 inv ← invoicesSent,
 line ← inv.2.orderLines,
 amount = line.unitPrice.amount × line.items.numberOfItems]

costOfGoodsSold = fifoCost
contribMargin = subtractMoney revenue fifoCost
fixedCosts = [] −− For simplicity
depreciation = [] −− For simplicity
netOpIncome = subtractMoney (subtractMoney contribMargin fixedCosts) depreciation

report : IncomeStatement
report = IncomeStatement{
  revenue = revenue,
  costOfGoodsSold = costOfGoodsSold,
  contribMargin = contribMargin,
  fixedCosts = fixedCosts,
  depreciation = depreciation,
  netOpIncome = netOpIncome}
```

## BalanceSheet

**name**: *BalanceSheet*
**description**:
*The Balance Sheet.*
**tags**: *external*, *financial*

```
-- List of all payments and their associated contract ID
payments : [(Int,Payment)]
payments = [ (tr.contractId,payment) |
  tr ← transactionEvents,
  payment : Payment = tr.transaction]

-- List of all received payments and their associated contract ID
paymentsReceived : [(Int,Payment)]
paymentsReceived = filter (λp → ¬ (isMe p.2.sender) ∧ isMe p.2.receiver) payments

-- List of all payments made and their associated contract ID
paymentsMade : [(Int,Payment)]
paymentsMade = filter (λp → isMe p.2.sender ∧ ¬ (isMe p.2.receiver)) payments

cashReceived : [Money]
cashReceived = normaliseMoney (map (λp → p.2.money) paymentsReceived)

cashPaid : [Money]
cashPaid = normaliseMoney (map (λp → p.2.money) paymentsMade)

netCashFlow : [Money]
netCashFlow = subtractMoney cashReceived cashPaid

depreciation : [Money]
depreciation = [] -- For simplicity

fAssetAcq : [Money]
fAssetAcq = [] -- For simplicity

fixedAssets : [Money]
fixedAssets = subtractMoney fAssetAcq depreciation

inventory : [Money]
inventory =
  let inventoryValue = [price |
        item ← invAcq,
        price = item.2{amount = item.2.amount × item.1.quantity}]
  in
  subtractMoney inventoryValue fifoCost

accReceivable : [Money]
accReceivable =
  let paymentsDue = normaliseMoney [line.unitPrice{amount = amount} |
        inv ← invoicesSent,
        line ← inv.2.orderLines,
        amount = line.unitPrice.amount × line.item.quantity]
  in
  subtractMoney paymentsDue cashReceived

currentAssets : [Money]
currentAssets = addMoney inventory (addMoney accReceivable netCashFlow)

totalAssets : [Money]
totalAssets = addMoney fixedAssets currentAssets

accPayable : [Money]
accPayable =
  let paymentsDue = [line.unitPrice{amount = amount} |
        inv ← invoicesReceived,
        line ← inv.2.orderLines,
        amount = line.unitPrice.amount × line.item.quantity]
  in
  subtractMoney paymentsDue cashPaid

vatPayable : [Money]
vatPayable = subtractMoney vatIncoming vatOutgoing

liabilities : [Money]
liabilities = addMoney accPayable vatPayable

ownersEq : [Money]
ownersEq = subtractMoney totalAssets liabilities

totalLiabPlusEq : [Money]
totalLiabPlusEq = addMoney liabilities ownersEq

report : BalanceSheet
report = BalanceSheet{
  fixedAssets = fixedAssets,
  currentAssets = CurrentAssets{
    currentAssets = currentAssets,
    inventory = inventory,
    accountsReceivable = accReceivable,
    cashPlusEquiv = netCashFlow},
  totalAssets = totalAssets,
  liabilities = Liabilities{
    liabilities = liabilities,
    accountsPayable = accPayable,
    vatPayable = vatPayable},
  ownersEquity = ownersEq,
  totalLiabilitiesPlusEquity = totalLiabPlusEq}
```

## CashFlowStatement

**name**: *CashFlowStatement*
**description**:
*The Cash Flow Statement.*
**tags**: *external*, *financial*

```
sumPayments : [Payment] → [Money]
sumPayments ps = normaliseMoney (map (λp → p.money) ps)

report : CashFlowStatement
report = let
    payments = [payment | payment : Payment ← transactions]
    mRevenues = [payment | payment ← payments, isMe (payment.receiver)]
    mExpenses = [payment | payment ← payments, isMe (payment.sender)]
  in
  CashFlowStatement{
    revenues = mRevenues,
    expenses = mExpenses,
    revenueTotal = sumPayments mRevenues,
    expenseTotal = sumPayments mExpenses}
```

## UnpaidInvoices

**name**: *UnpaidInvoices*
**description**:
*A list of unpaid invoices.*
**tags**: *external*, *financial*

```
-- Generate a list of unpaid invoices
unpaidInvoices : [UnpaidInvoice]
unpaidInvoices = [UnpaidInvoice{invoice = inv, remainder = remainder} |
  invS ← invoicesSent,
  inv = Invoice{
    sender = invS.2.sender @,
    receiver = invS.2.receiver @,
    orderLines = invS.2.orderLines},
  payments = [payment.money |
    tr ← transactionEvents,
    tr.contractId ≡ invS.1,
    payment : Payment = tr.transaction],
  remainder = subtractMoney (invoiceTotal inv) payments,
  any (λm → m.amount > 0) remainder]

report : UnpaidInvoices
report = UnpaidInvoices{invoices = unpaidInvoices}
```

## VATReport

**name**: *VATReport*
**description**:
*The VAT report.*
**tags**: *external*, *financial*

```
report : VATReport
report = VATReport{
  outgoingVAT = vatOutgoing,
  incomingVAT = vatIncoming,
  vatDue = subtractMoney vatIncoming vatOutgoing}
```

## Inventory

**name**: *Inventory*
**description**:
*A list of items in the inventory available for sale (regardless of whether we have paid for them).*
**tags**: *external*, *inventory*

```
report : Inventory
report =
  let itemsSold' = map (λi → i{quantity = 0 − i.quantity}) itemsSold
  in
  −− The available items is the list of received items minus the list of reserved
  −− or sold items
  Inventory{availableItems = normaliseItems (itemsReceived ++ itemsSold')}
```

## TopNCustomers

**name**: *TopNCustomers*
**description**:
*A list of customers who have spent must money in the given currency.*
**tags**: *external*, *financial*, *crm*

```
customers : [⟨Customer⟩]
customers = [c | c : ⟨Customer⟩ ← map (λe → e.1) entities]

totalPayments : Currency → ⟨Customer⟩ → Double
totalPayments c cu = sum [d |
  p : Payment ← transactions,
  p.sender ≡ cu ∨ p.receiver ≡ cu,
  p.money.currency ≡ c,
  d = if p.sender ≡ cu then p.money.amount else 0 − p.money.amount]

customerStatistics : Currency → [CustomerStatistics]
customerStatistics c = [CustomerStatistics{customer = cu, totalPaid = p} |
  cu ← customers,
  p = Money{currency = c, amount = totalPayments c cu}]

topN : Int → [CustomerStatistics] → [CustomerStatistics]
topN n cs = take n (sortBy (λcs1 cs2 → cs1.totalPaid > cs2.totalPaid) cs)

report : Int → Currency → TopNCustomers
report n c = TopNCustomers{customerStatistics = topN n (customerStatistics c)}
```

## E.2.3 Contracts

### E.2.3.1 Prelude

```
// Arithmetic
fun floor x = let n = ceil x in if n > x then n − 1 else n
fun round x = let n1 = ceil x in let n2 = floor x in if n1 + n2 > 2 × x then n2 else n1
fun max a b = if a > b then a else b
fun min a b = if a > b then b else a

// List functions
fun filter f = foldr (λx b → if f x then x # b else b) []
fun map f = foldr (λx b → (f x) # b) []
val length = foldr (λx b → b + 1) 0
fun null l = l ≡ []
fun elem x = foldr (λy b → x ≡ y ∨ b) false
fun all f = foldr (λx b → b ∧ f x) true
fun any f = foldr (λx b → b ∨ f x) false
val reverse = foldl (λa e → e # a) []
fun append l1 l2 = foldr (λe a → e # a) l2 l1

// Lists as sets
fun subset l1 l2 = all (λx → elem x l2) l1
fun diff l1 l2 = filter (λx → ¬ (elem x l2)) l1
```

### E.2.3.2 Domain-Specific Prelude

```
// Check if 'lines' are in stock by invoking the 'Inventory' report
fun inStock lines =
  let inv = (reports.inventory ()).availableItems
  in
  all (λl → any (λi → (l.item).itemType ≡ i.itemType ∧ (l.item).quantity ≤ i.quantity) inv) lines

// Check that amount 'm' equals the total amount in m's currency of a list of sales lines
fun checkAmount m orderLines =
  let a = foldr (λx acc →
          if (x.unitPrice).currency ≡ m.currency then
            (x.item).quantity × (100 + x.vatPercentage) × (x.unitPrice).amount + acc
          else
            acc) 0 orderLines
  in
  m.amount × 100 ≡ a

// Remove sales lines that have the currency of 'm'
fun remainingOrderLines m = filter (λx → (x.unitPrice).currency ≢ m.currency)

// A reference to the designated entity that represents the company
val me = reports.me ()
```

### E.2.3.3 Contract Templates

#### Purchase

```
name: purchase
type: Purchase
description: "Set up a purchase"

clause purchase(lines : [OrderLine])⟨me : ⟨Me⟩, vendor : ⟨Vendor⟩⟩ =
 ⟨vendor⟩ Delivery(sender s, receiver r, items i)
   where s ≡ vendor ∧ r ≡ me ∧ i ≡ map (λx → x.item) lines
   due within 1W
 then
 when IssueInvoice(sender s, receiver r, orderLines sl)
   where s ≡ vendor ∧ r ≡ me ∧ sl ≡ lines
   due within 1Y
 then
 payment(lines, vendor, 14D)⟨me⟩
```

```
clause payment(lines : [OrderLine], vendor : ⟨Vendor⟩, deadline : Duration)
                    ⟨me : ⟨Me⟩⟩ =
 if null lines then
   fulfilment
 else
   ⟨me⟩ BankTransfer(sender s, receiver r, money m)
     where s ≡ me ∧ r ≡ vendor ∧ checkAmount m lines
     due within deadline
     remaining newDeadline
   then
   payment(remainingOrderLines m lines, vendor, newDeadline)⟨me⟩

contract = purchase(orderLines)⟨me, vendor⟩
```

## Sale

**name**: *sale*
**type**: Sale
**description**: `"Set up a sale"`

```
clause sale(lines : [OrderLine])⟨me : ⟨Me⟩, customer : ⟨Customer⟩⟩ =
 ⟨me⟩ IssueInvoice(sender s, receiver r, orderLines sl)
   where s ≡ me ∧ r ≡ customer ∧ sl ≡ lines ∧ inStock lines
   due within 1H
 then
 payment(lines, me, 10m)⟨customer⟩
 and
 ⟨me⟩ Delivery(sender s, receiver r, items i)
   where s ≡ me ∧ r ≡ customer ∧ i ≡ map (λx → x.item) lines
   due within 1W
 then
 repair(map (λx → x.item) lines, customer, 3M)⟨me⟩

clause payment(lines : [OrderLine], me : ⟨Me⟩, deadline : Duration)
                    ⟨customer : ⟨Customer⟩⟩ =
 if null lines then
   fulfilment
 else
   ⟨customer⟩ Payment(sender s, receiver r, money m)
     where s ≡ customer ∧ r ≡ me ∧ checkAmount m lines
     due within deadline
     remaining newDeadline
   then
   payment(remainingOrderLines m lines, me, newDeadline)⟨customer⟩

clause repair(items : [Item], customer : ⟨Customer⟩, deadline : Duration)
                ⟨me : ⟨Me⟩⟩ =
 when RequestRepair(sender s, receiver r, items i)
   where s ≡ customer ∧ r ≡ me ∧ subset i items
   due within deadline
   remaining newDeadline
 then
 ⟨me⟩ Repair(sender s, receiver r, items i')
   where s ≡ me ∧ r ≡ customer ∧ i ≡ i'
   due within 5D
 and
 repair(items, customer, newDeadline)⟨me⟩

contract = sale(orderLines)⟨me, customer⟩
```

# Bibliography

[1] 3rd generation Enterprise Resource Planning, 2011. URL http://www.3gerp.org.

[2] Samson Abramsky and Radha Jagadeesan. Games and Full Completeness for Multiplicative Linear Logic. In *Foundations of Software Technology and Theoretical Computer Science*, pages 291–301. Springer Berlin / Heidelberg, 1992.

[3] Ki Yung Ahn and Tim Sheard. A Hierarchy of Mendler style Recursion Combinators: Taming Inductive Datatypes with Negative Occurrences. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional programming*, pages 234–246, New York, NY, USA, 2011. ACM.

[4] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.

[5] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[6] Jesper Andersen, Ebbe Elsborg, Fritz Henglein, Jakob Simonsen, and Christian Stefansen. Compositional specification of commercial contracts. *International Journal on Software Tools for Technology Transfer (STTT)*, 8:485–516, 2006.

[7] Robert Atkey. Syntax for Free: Representing Syntax with Binding Using Parametricity. In Pierre-Louis Curien, editor, *Typed Lambda Calculi and Applications*, pages 35–49. Springer Berlin / Heidelberg, 2009.

[8] Robert Atkey, Sam Lindley, and Jeremy Yallop. Unembedding Domain-Specific Languages. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 37–48, New York, NY, USA, 2009. ACM.

[9] Ralph-Johan Back and Joakim von Wright. Contracts, Games, and Refinement. *Information and Computation*, 156(1-2):25–45, 2000.

[10] Patrick Bahr and Tom Hvitved. Parametric Compositional Data Types. Technical report, Department of Computer Science, University of Copenhagen, 2011. Submitted to MSFP 2012.

[11] Patrick Bahr and Tom Hvitved. Compositional Data Types. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming*, pages 83–94, New York, NY, USA, 2011. ACM.

[12] T. J. M. Bench-Capon and F. P. Coenen. Isomorphism and Legal Knowledge Based Systems. *Artificial Intelligence and Law*, 1:65–86, 1992.

[13] Arthur J. Bernstein and Michael Kifer. *Databases and Transaction Processing: An Application-Oriented Approach.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 2001.

[14] Azer Bestavros. The Input Output Timed Automaton: A model for real-time parallel computation. In *Proceedings of 1990 ACM Intl. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (Tau '90)*, 1990.

[15] Richard Bird and Ross Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 11:200–222, 1999.

[16] Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A Theory of Design-by-Contract for Distributed Multiparty Interactions. In *CONCUR 2010 - Concurrency Theory*, pages 162–176. Springer Berlin / Heidelberg, 2010.

[17] Abdel Boulmakoul and Mathias Sallé. Integrated contract management. Technical Report HPL-2002-183, HP Laboratories Bristol, Bristol, United Kingdom, 2002.

[18] Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(05):509–543, 2009.

[19] Adam Chlipala. Parametric Higher-Order Abstract Syntax for Mechanized Semantics. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 143–156, New York, NY, USA, 2008. ACM.

[20] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications.* 2007.

[21] *Contract Expression Language (CEL) – an UN/CEFACT BCF Compliant Technology.* Content Reference Forum, 2004.

[22] Pierre-Louis Curien. Notes on game semantics. Technical report, CNRS – Université Paris 7, 2006.

[23] Chris Exton and Jian Chen. Programming by Contract in a Distributed Object Environment. In *Proceedings of the International Symposium on Future Software Technology (ISFST-96)*, pages 272–278. Software Engineers Association (Japan), 1996.

[24] *General rules and guidelines for the PhD programme.* Faculty of Science University of Copenhagen, 2010. Adopted on 8th of January 2010.

[25] Leonidas Fegaras and Tim Sheard. Revisiting Catamorphisms over Datatypes with Embedded Functions (or, Programs from Outer Space). In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 284–294, New York, NY, USA, 1996. ACM.

[26] Stephen Fenech, Gordon Pace, and Gerardo Schneider. Automatic Conflict Detection on Contracts. In *Theoretical Aspects of Computing - ICTAC 2009*, pages 200–214. Springer Berlin / Heidelberg, 2009.

[27] Maarten Fokkinga. Monadic Maps and Folds for Arbitrary Datatypes. Technical Report 94-28, Department of Computer Science, University of Twente, Enschede, The Netherlands, 1994.

[28] James William Forrester. Gentle Murder, or the Adverbial Samaritan. *The Journal of Philosophy*, 81(4):193–197, 1984.

[29] Norbert Fuchs, Kaarel Kaljurand, and Tobias Kuhn. Attempto Controlled English for Knowledge Representation. In *Reasoning Web*, pages 104–124. Springer Berlin / Heidelberg, 2008.

[30] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73(2):155–175, 1990.

[31] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A Short Cut to Deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 223–232, New York, NY, USA, 1993. ACM.

[32] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50:1–102, 1987.

[33] Andrew Goodchild, Charles Herring, and Zoran Milosevic. Business Contracts for B2B. In *Proceedings of the CAiSE 2000 Workshop on Infrastructure for Dynamic Business-to-Business Service Outsourcing (ISDO)*, pages 63–74, 2000.

[34] Guido Governatori. Representing Business Contracts in RuleML. *International Journal of Cooperative Information Systems (IJCIS)*, 14(2-3):181–216, 2005.

[35] Guido Governatori and Zoran Milosevic. A Formal Analysis of a Business Contract Language. *International Journal of Cooperative Information Systems (IJCIS)*, 15(4):659–685, 2006.

[36] Guido Governatori and Duy Hoang Pham. DR-CONTRACT: an architecture for e-contracts in defeasible logic. *International Journal of Business Process Integration and Management*, 4(3):187–199, 2009.

[37] Guido Governatori and Antonino Rotolo. Logic of Violations: A Gentzen System for Reasoning with Contrary-To-Duty Obligations. *Australasian Journal of Logic*, 4:193–215, 2006.

[38] Guido Governatori, Zoran Milosevic, and Shazia Sadiq. Compliance checking between business processes and business contracts. In *Enterprise Distributed Object Computing Conference*, pages 221–232, Hong Kong, 2006. IEEE.

[39] Ichiro Hasuo, Bart Jacobs, and Tarmo Uustalu. Categorical Views on Computations on Trees (Extended Abstract). In *Automata, Languages and Programming*, pages 619–630. Springer Berlin / Heidelberg, 2007.

[40] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In *Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 169–180, New York, NY, USA, 1990. ACM.

[41] Fritz Henglein, Ken Friis Larsen, Jakob Grue Simonsen, and Christian Stefansen. POETS: Process-oriented event-driven transaction systems. *Journal of Logic and Algebraic Programming*, 78(5):381–401, May 2009.

[42] Anders Starcke Henriksen, Tom Hvitved, and Andrzej Filinski. A Game-Theoretic Model for Distributed Programming by Contract. In *GI Jahrestagung*, pages 3473–3484, 2009.

[43] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12:576–580, October 1969.

[44] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1984.

[45] Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Programming Languages and Systems*, pages 122–138. Springer Berlin / Heidelberg, 1998.

[46] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 273–284, New York, NY, USA, 2008. ACM.

[47] Jozef Hooman. Extending Hoare Logic to Real-Time. *Formal Aspects of Computing*, 6:801–825, 1994.

[48] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, Cambridge, United Kingdom, 2004.

[49] Tom Hvitved. Contracts in Programming and in Enterprise Systems. Master's thesis, Department of Computer Science, University of Copenhagen, 2009. Ph.D. Progress Report.

[50] Tom Hvitved. A Survey of Formal Languages for Contracts. In *Formal Languages and Analysis of Contract-Oriented Software (FLACOS)*, pages 29–32, 2010.

[51] Tom Hvitved, Patrick Bahr, and Jesper Andersen. Domain-Specific Languages for Enterprise Systems. Technical report, Department of Computer Science, University of Copenhagen, 2011.

[52] Tom Hvitved, Felix Klaedtke, and Eugen Zălinescu. A trace-based model for multiparty contracts. *Journal of Logic and Algebraic Programming*, In Press, Accepted Manuscript:–, 2011.

[53] Patricia Johann and Neil Ghani. Foundations for Structured Programming with GADTs. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 297–308, New York, NY, USA, 2008. ACM.

[54] C B Jones. Wanted: a compositional approach to concurrency. In Annabelle McIver and Carroll Morgan, editors, *Programming Methodology*, pages 1–15. Springer Verlag, 2000.

[55] Mikkel Jønsson Thomsen. Using Controlled Natural Language for specifying ERP Requirements. Master's thesis, University of Copenhagen, Department of Computer Science, 2010.

[56] Neelakantan R. Krishnaswami and Nick Benton. Ultrametric Semantics of Reactive Programs. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science*, pages 257–266. IEEE Computer Society, 2011.

[57] Marcel Kyas, Cristian Prisacariu, and Gerardo Schneider. Run-Time Monitoring of Electronic Contracts. In *Automated Technology for Verification and Analysis*, pages 397–407. Springer Berlin / Heidelberg, 2008.

[58] Ronald M. Lee. A logic model for electronic contracting. *Decision Support Systems*, 4(1):27–44, 1988.

[59] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[60] P. F. Linington, Z. Milosevic, J. Cole, S. Gibson, S. Kulkarni, and S. Neal. A unified behavioural model and a contract language for extended enterprise. *Data & Knowledge Engineering*, 51(1):5–29, 2004.

[61] Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2-3):255–279, 1990.

[62] Simon Marlow. *Haskell 2010 Language Report*, 2010.

[63] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(01):1–13, 2008.

[64] William E. McCarthy. The REA Accounting Model: A Generalized Framework for Accounting Systems in a Shared Data Environment. *The Accounting Review*, LVII(3):554–578, 1982.

[65] Paul McNamara. Deontic Logic, 2006. URL http://plato.stanford.edu/entries/logic-deontic/.

[66] Erik Meijer and Graham Hutton. Bananas in Space: Extending Fold and Unfold to Exponential Types. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 324–333, New York, NY, USA, 1995. ACM.

[67] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 124–144, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[68] Elliott Mendelson. *Introducing Game Theory and its Applications.* Discrete Mathematics and Its Applications. CRC Press, 2004.

[69] Bertrand Meyer. Applying “Design by Contract”. *Computer*, 25(10):40–51, 1992.

[70] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[71] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, I. *Information and Computation*, 100(1):1–40, 1992.

[72] Z. Milosevic, S. Gibson, P. F. Linington, J. Cole, and S. Kulkarni. On Design and Implementation of a Contract Monitoring Facility. In *Proceedings of the First IEEE International Workshop on Electronic Contracting*, pages 62–70, Washington, DC, USA, 2004. IEEE Computer Society.

[73] Neil Mitchell and Colin Runciman. Uniform Boilerplate and List Processing. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 49–60, New York, NY, USA, 2007. ACM.

[74] Carlos Molina-Jimenez, Santosh Shrivastava, Ellis Solaiman, and John Warne. Run-time monitoring and enforcement of electronic contracts. *Electronic Commerce Research and Applications*, 3(2):108–125, 2004.

[75] J. Garret Morris and Mark P. Jones. Instance Chains: Type Class Programming Without Overlapping Instances. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 375–386, New York, NY, USA, 2010. ACM.

[76] George C. Necula. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM.

[77] Michael Nissen and Ken Friis Larsen. FunSETL—Functional Reporting for ERP Systems. In *Implementation and Application of Functional Languages, 19th International Symposium, IFL 2007*, pages 268–289, 2007.

[78] Donald Nute. Defeasible logic. In *Handbook of logic in artificial intelligence and logic programming (vol. 3)*, pages 353–395. Oxford University Press, Inc., New York, NY, USA, 1994.

[79] Atsushi Ohori. A Polymorphic Record Calculus and Its Compilation. *ACM Trans. Program. Lang. Syst.*, 17:844–895, November 1995.

[80] Nir Oren, Sofia Panagiotidi, Javier Vázquez-Salceda, Sanjay Modgil, Michael Luck, and Simon Miles. Towards a Formalisation of Electronic Contracting Environments. In *Coordination, Organizations, Institutions and Norms in Agent Systems IV*, pages 156–171. Springer Berlin / Heidelberg, 2009.

[81] Susan Owicki and David Gries. An Axiomatic Proof Technique for Parallel Programs I. *Acta Informatica*, 6:319–340, 1976.

[82] Gordon Pace and Gerardo Schneider. Challenges in the Specification of Full Contracts. In *Integrated Formal Methods*, pages 292–306. Springer Berlin / Heidelberg, 2009.

[83] Gordon Pace, Cristian Prisacariu, and Gerardo Schneider. Model Checking Contracts – A Case Study. In *Automated Technology for Verification and Analysis*, pages 82–97. Springer Berlin / Heidelberg, 2007.

[84] Vishal Patel. The Contract Management Benchmark Report: Procurement Contracts. Technical report, Aberdeen Group, Boston, MA, USA, 2006.

[85] Vishal Patel and Christopher J Dwyer. Contract Lifecycle Management and the CFO: Optimizing Revenues and Capturing Savings. Technical report, Aberdeen Group, Boston, MA, USA, 2007.

[86] Simon Peyton Jones and Jean-Marc Eber. How to write a financial contract. In *The Fun of Programming*, pages 105–130. Palgrave Macmillan Ltd., London, United Kingdom, 2003.

[87] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the Rules: Rewriting as a practical optimisation technique in GHC. In *Haskell Workshop*, pages 203–233, 2001.

[88] F. Pfenning and C. Elliot. Higher-Order Abstract Syntax. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 199–208, New York, NY, USA, 1988. ACM.

[89] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 2002.

[90] Benjamin C. Pierce, editor. *Advanced Topics in Types and Programming Languages.* The MIT Press, 2005.

[91] Andrew M. Pitts. Alpha-Structural Recursion and Induction. *J. ACM*, 53: 459–506, May 2006.

[92] Henry Prakken and Marek Sergot. Contrary-to-duty obligations. *Studia Logica*, 57:91–115, 1996.

[93] Cristian Prisacariu and Gerardo Schneider. A Formal Language for Electronic Contracts. In *Formal Methods for Open Object-Based Distributed Systems*, pages 174–189. Springer Berlin / Heidelberg, 2007.

[94] Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. Comparing Libraries for Generic Programming in Haskell. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 111–122, New York, NY, USA, 2008. ACM.

[95] Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. Comparing Libraries for Generic Programming in Haskell. Technical Report UU-CS-2008-010, Utrecht University, 2008.

[96] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type Checking with Open Type Functions. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 51–62, New York, NY, USA, 2008. ACM.

[97] Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. Complete and decidable type inference for GADTs. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 341–352, New York, NY, USA, 2009. ACM.

[98] Carsten Schürmann, Joëlle Despeyroux, and Frank Pfenning. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, 266(1-2): 1–57, 2001.

[99] Tim Sheard and Simon Peyton Jones. Template Meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16, New York, NY, USA, 2002. ACM.

[100] Mark R. Shinwell, Andrew M. Pitts, and Murdoch J. Gabbay. FreshML: Programming with Binders Made Simple. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 263–274, New York, NY, USA, 2003. ACM.

[101] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation. Technical report, Facebook, 156 University Ave, Palo Alto, CA, 2007.

[102] Perdita Stevens and Rob Pooley. *Using UML: Software Engineering with Objects and Components.* Object Technology Series. Addison-Wesley Longman, 1999.

[103] Doaitse S. Swierstra, Pablo R. Azero Alcocer, and Joao Saraiva. Designing and Implementing Combinator Languages. In *Third Summer School on Advanced Functional Programming, volume 1608 of LNCS*, pages 150–206. Springer-Verlag, 1999.

[104] Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(04):423–436, 2008.

[105] Yao-Hua Tan and Walter Thoen. INCAS: a legal expert system for contract terms in electronic commerce. *Decision Support Systems*, 29(4):389–411, 2000.

[106] Yao-Hua Tan, Walter Thoen, and Somasundaram Ramanathan. A Survey of Electronic Contracting Related Developments. In *BLED 2001 Proceedings*, pages 495–507, 2001.

[107] James W. Thatcher. Tree automata: an informal survey. In *Currents in the theory of computing*, pages 143–178. Prentice Hall, 1973.

[108] Martijn Van Steenbergen, José Pedro Magalhães, and Johan Jeuring. Generic Selections of Subexpressions. In *Proceedings of the 6th ACM SIGPLAN workshop on Generic programming*, pages 37–48, New York, NY, USA, 2010. ACM.

[109] Varmo Vene. *Categorical Programming with Inductive and Coinductive Types.* PhD thesis, Faculty of Mathematics, University of Tartu, Estonia, 2000.

[110] Sebastiaan Visser and Andres Löh. Generic Storage in Haskell. In *Proceedings of the 6th ACM SIGPLAN workshop on Generic programming*, pages 25–36, New York, NY, USA, 2010. ACM.

[111] G. H. von Wright. Deontic Logic. *Mind*, 60(237):1–15, 1951.

[112] Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.

[113] Philip Wadler. Comprehending Monads. *Mathematical Structures in Computer Science*, 2(04):461–493, 1992.

[114] Philip Wadler. The Expression Problem, 1998. URL http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt.

[115] Philip Wadler and Robert Findler. Well-Typed Programs Can't Be Blamed. In *Programming Languages and Systems*, pages 1–16. Springer Berlin / Heidelberg, 2009.

[116] Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. *Journal of Functional Programming*, 18(01):87–140, 2008.

[117] Hans Weigand and Lai Xu. Contracts in E-Commerce. In *Proceedings of the IFIP TC2/WG2.6 Ninth Working Conference on Database Semantics: Semantic Issues in E-Commerce Systems*, pages 3–17, Deventer, The Netherlands, The Netherlands, 2003. Kluwer, B.V.

[118] Jerry J. Weygandt, Donald E. Kieso, and Paul D. Kimmel. *Financial Accounting, with Annual Report.* Wiley, 2004.

[119] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, Cambridge, MA, USA, 1993.

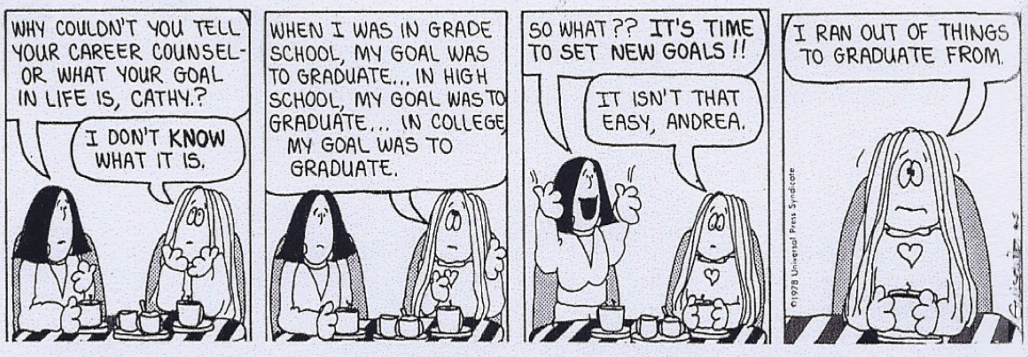
[120] J. Woleński. Deontic Logic and Possible Worlds Semantics: A Historical Sketch. *Studia Logica*, 49:273–282, 1990.

[121] Lai Xu. A Multi-party Contract Model. *SIGecom Exchanges*, 5:13–23, July 2004.

[122] Lai Xu and Manfred A. Jeusfeld. Pro-active Monitoring of Electronic Contracts. In *Advanced Information Systems Engineering*, pages 584–600. Springer Berlin / Heidelberg, 2003.

[123] Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic Programming with Fixed Points for Mutually Recursive Datatypes. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 233–244, New York, NY, USA, 2009. ACM.

[124] Brent Yorgey. Typed type-level functional programming in GHC. In *Haskell Implementors Workshop*, 2010.


WHY COULDN'T YOU TELL YOUR CAREER COUNSELOR WHAT YOUR GOAL IN LIFE IS, CATHY.?
I DON'T KNOW WHAT IT IS.
WHEN I WAS IN GRADE SCHOOL, MY GOAL WAS TO GRADUATE... IN HIGH SCHOOL, MY GOAL WAS TO GRADUATE... IN COLLEGE, MY GOAL WAS TO GRADUATE.
SO WHAT?? IT'S TIME TO SET NEW GOALS!!
IT ISN'T THAT EASY, ANDREA.
I RAN OUT OF THINGS TO GRADUATE FROM.
©1978 Universal Press Syndicate