Universidad de la República
PEDECIBA Informática
Uruguay

Reasoning about Functional Programs by Combining Interactive and Automatic Proofs

Andrés Sicard-Ramírez

Tesis presentada en cumplimiento parcial de los
requisitos para el grado de **Doctor en Informática**
Universidad de la República - Pedeciba Informática

Montevideo, Uruguay, Julio de 2014

Directores de Tesis: Dra. Ana Bove and Dr. Peter Dybjer
Chalmers University of Technology, Suecia

Director de Estudios: Dr. Alberto Pardo
Universidad de la República, Uruguay

Abstract

We propose a new approach to computer-assisted verification of lazy functional programs where functions can be defined by general recursion. We work in first-order theories of functional programs which are obtained by translating Dybjer's programming logic (Dybjer, P. [1985]. Program Verification in a Logical Theory of Constructions. In: Functional Programming Languages and Computer Architecture. Ed. by Jouannaud, J.-P. Vol. 201. Lecture Notes in Computer Science. Springer, pp. 334–349) into a first-order theory, and by extending this programming logic with new (co-)inductive predicates. Rather than building a special purpose system, we formalise our theories in Agda, a proof assistant for dependent type theory which can be used as a generic theorem prover. Agda provides support for interactive reasoning by representing first-order theories using the propositions-as-types principle. Further support is provided by off-the-shelf automatic theorem provers for first-order-logic called by a Haskell program that translates our Agda representations of first-order formulae into the TPTP language understood by the provers. We show some examples where we combine interactive and automatic reasoning, covering both proofs by induction and coinduction. The examples include functions defined by structural recursion, simple general recursion, nested recursion, higher-order recursion, guarded and unguarded co-recursion.

Keywords: automatic proofs, first-order theories, functional program correctness, general recursion, interactive proofs, lazy evaluation, total languages, type theory

Resumen

Proponemos un nuevo enfoque a la verificación asistida por computador de programas funcionales perezosos, en los cuales las funciones pueden ser definidas por recursión general. Empleamos teorías de primer orden para programas funcionales las cuales fueron obtenidas de traducir la lógica para la programación de Dybjer (Dybjer, P. [1985]. Program Verification in a Logical Theory of Constructions. En: Functional Programming Languages and Computer Architecture. Ed. por Jouannaud, J.-P. Vol. 201. Lecture Notes in Computer Science. Springer, págs. 334-349) a una teoría de primer orden, y de extender esta lógica para la programación con nuevos predicados (co-)inductivos. En lugar de construir un sistema para formalizar nuestras teorías, formalizamos éstas en Agda, un asistente de pruebas para teoría de tipos dependientes que puede ser usado como un demostrador de teoremas genérico. Agda proporciona soporte para el razonamiento interactivo representando las teorías de primer orden mediante el principio de *propositions-as-types*. Se obtiene soporte adicional mediante demostradores automáticos de teoremas genéricos para lógica de primer orden, los cuales son llamados por un programa desarrollado en Haskell, que traslada nuestra representación en Agda de las fórmulas de primer orden al lenguaje TPTP entendido por los demostradores automáticos. Mostramos ejemplos de combinación de razonamiento interactivo y automático en pruebas por inducción y por co-inducción. Nuestros ejemplos incluyen funciones definidas por recursión estructural, recursión general simple, recursión anidada, recursión de orden superior y co-recursión.

Palabras claves: demostración automática de teoremas, demostración interactiva de teoremas, evaluación perezosa, lenguajes totales, recursión general, teoría de tipos, teorías de primer orden, verificación de programas funcionales

Part of this thesis is based on the work contained in the following papers:

- Bove, A., Dybjer, P. and Sicard-Ramírez, A. [2012]. Combining Interactive and Automatic Reasoning in First Order Theories of Functional Programs. In: Foundations of Software Science and Computation Structures (FoSSaCS 2012). Ed. by Birkedal, L. Vol. 7213. Lecture Notes in Computer Science. Springer, pp. 104–118
- Bove, A., Dybjer, P. and Sicard-Ramírez, A. [2009]. Embedding a Logical Theory of Constructions in Agda. In: Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification (PLPV 2009), pp. 59–66

Acknowledgements

I would like to thank to my supervisors, Ana Bove and Peter Dybjer. Their generosity to share their knowledge, their patience at my naive questions, the precision of their answers, the high level demanded in its fair proportions, the promptness of their answers, the words of encouragement in the adequate moments as well as their understanding before personal difficulties; apart from allowing me to finish my doctorate satisfactorily, were highly valuable teachings at academic and at personal level.

My thanks to my director of studies Alberto Pardo, the warmth with which he accepted me as student at the University of the Republic, as well as his handling of the aspects related to the academic management of my doctorate.

My thanks to Francisco José Correa-Zabala for managing the participation of the EAFIT University in LERnet—an ALFA project approved by the European Community that foresees PhD student mobility from Latin America to the European Union, and vice versa—and inviting me to be one of the students who benefited from this network. I also appreciate his continuous motivation during the completion of my doctorate.

My thanks to the members of my thesis tribunal, Gustavo Betarte, Daniel Fridlender, Mauro Jaskelioffe, Alexandre Miquel and Nora Szasz.

The completion of my doctorate would have not been possible without the support offered and the confidence placed by the EAFIT University. I would like to thank to the board of directors, to the administrative staff and to my superiors for their continuous support.

I would also like to thank to the Department of Computer Science and Engineering of Chalmers University of Technology for having received me as a visiting doctoral student. The experience gathered with my internship in Gothenburg was something very valuable within my process of academic training.

My appreciation to the following individuals for their help and feedback related to different aspects of my thesis: Andreas Abel, Juan Francisco Cardona-McCormick, Koen Claessen, Nils Anders Danielsson, Marco Gaboardi, Kryštof Hoder, Joe Leslie-Hurd, Ulf Norell, Bengt Nordström, Jens Otten, Stephan Schulz, Geoff Sutcliffe and Makoto Takeyama. I apologize

in advance with those persons whose name I have omitted unintentionally.

During my visits to Chalmers University of Technology, Emil Karlen, Miguel Pagano and David Wahlstedt were good friends who made my stay more pleasant.

Finally, but not less importantly, I want to thank to my wife Andrea and my children Sofía and Nicolás for reminding me with enough frequency some of the most important things in life.

Contents

Chapter 1

Introduction

The development of programming languages and methods for reasoning about programs have been interwoven since their beginnings. It is often claimed that *pure* functional programming languages, that is, functional programming languages where the functions *"take all their input as explicit arguments, and produce all their output as explicit results"* [Hutton 2007, p. 87], are suitable for formal reasoning (see, for example, Wadler [1987] and Hughes [1989]).

Achten et al. [2010] state that research on reasoning about functional programs can be divided into four categories: (i) defining semantics, (ii) transformational reasoning, (iii) formal verification of functional properties and (iv) formal verification of non-functional properties. For these categories, research focuses on defining new concepts, on programming transformations, on the input and output correspondence of programs, and on properties such as memory consumption or parallel performance, respectively.

The research performed to support this thesis belongs to (iii) above on a certain kind of programs. Namely, our research focuses on the verification of functional properties of higher-order (pure) lazy functional programs.

The term 'lazy' denotes at most two different aspects: (i) an evaluation strategy—lazy evaluation—that does not re-evaluate expressions which have already been evaluated (see, for example, Peyton Jones [1987]) and (ii) the fact that the evaluation of arguments proceeds in a lazy manner, that is, the arguments of a function are evaluated when it is strictly necessary. On this thesis, we use the term 'lazy' to refer to (ii).

Assume that we want to formally verify programs written in a lazy functional language like Haskell [Peyton Jones 2003], where functions can be defined by *general* recursion. What do we do? In other words:

- What *programming logic* and what *proof assistant* should we use?
  - by programming logic we mean a logic in which programs and specifications can be expressed and in which it can be proved or

disproved that a certain program meets a certain specification, and

– by proof assistant we mean a computer system which helps with the development of formal proofs.

- Can part of the job be *automated*?

A significant challenge is how to deal with the possible use of general recursion. By allowing unrestricted recursion, the semantics of Haskell must deal with undefined values representing errors and non-terminating programs. Moreover, most of the proof assistants lack a direct treatment for general recursive functions [Bove, Krauss and Sozeau 2012].

The goal of this thesis is to build a computer-assisted framework for reasoning about programs written in Haskell-like lazy functional languages. We suggest that this goal can be achieved by defining a *first-order* programming logic for lazy functional programs and by building on existing state-of-the-art systems in interactive and automatic theorem proving. Using Turner's terminology where total values includes finite data and potentially infinite co-data (see, for example, Turner [1995, 2004]), our approach makes it possible to reason about arbitrary functional programs, also those which denote partial functions. However, typically, valid inputs and outputs are total values, and we use predicates for expressing the properties of correct inputs and correct outputs.

For reasoning about functional programs, we can use equational reasoning, induction and co-induction (see, for example, Burstall [1969], Wadler [1987] and Gordon [1995]). In this thesis, we shall use an interactive proof assistant to handle the high-level proofs steps (for example, introduction of hypothesis, case analysis and the use of (co-)inductive principles). Moreover, we shall use off-the-shelf automatic theorem provers for first-order logic (henceforth, ATPs) for the proof steps involving equational reasoning or simple first-order reasoning. Therefore in the sequel, we shall talk about interactive, automatic and combined proofs.

Our answers to the questions at the beginning of this introduction provide our main contributions, as follows:

- In the thesis, we define and formalise two programming logics. We define our first programming logic for reasoning about programs using total and finite natural numbers and Booleans for a version of a core lazy functional programming language, Plotkin's [1977] PCF language. We use a type-free language where the property of being a total and finite value is represented by two predicates (one for natural numbers and other for Booleans). Moreover, these predicates allow to express that a certain program terminates with a finite total natural number

or a Boolean value, respectively. Our theory is not strictly a first-order theory, since it uses λ-abstraction. Our theory can deal with general recursion (structural and non-structural recursion), higher-order functions, inductive definitions of natural numbers and Boolean data types, and proofs by induction. We shall use this programming logic for basing the consistency of the second programming logic described in the next item.

- We define our second programming logic for reasoning about programs for extended versions of PCF. As in our previous programming logic, we use a type-free language where the properties of being a total value are represented by predicates, and these predicates allow to express that a certain program represents a total value. Unlike our previous programming logic, our theory is a first-order theory because we use λ-lifting to remove the λ-abstractions. Our theory can deal with general recursion (structural and non-structural recursion, guarded and unguarded co-recursion), higher-order functions, (co-)inductive definitions of data types, and proofs by (co-)induction. This programming logic will be the one used for reasoning about Haskell-like lazy functional programs.

- We formalise our programming logics in the Agda proof assistant [Norell 2007b; The Agda Development Team 2014]. In particular, we use Agda as a logical framework, that is, as a meta-logical system for formalising other logics, such as classical first-order logic. The basic methodology is similar to that of the Edinburgh Logical Framework [Harper, Honsell and Plotkin 1993]—another logical framework based on dependent type theory. However, since Agda is a language with support for inductive definition and pattern matching, we can use this support when formalising inductive definitions and proofs by induction. This helps making proof construction easier and more user-friendly. In addition, by using Agda, we get access to advanced features for interactively building proofs, such as, commands for refining proof terms, flexible mixfix syntax accepting Unicode, a fine interface, and so on. We call this approach "the inductive approach" to logical frameworks.

- We provide a translation of our Agda representation of first-order formulae into TPTP [Sutcliffe 2009]—a language understood by many off-the-shelf ATPs—so we can use them when proving the properties of our programs. For this purpose we extended Agda with an ATP-pragma, which instructs Agda to interact with the ATPs. Moreover, we wrote the Apia program, a Haskell program which performs the above translation, calls the ATPs and uses Agda as a Haskell library.

- We illustrate our approach with some examples where we verify some general (co-)recursive programs including nested recursive functions, higher-order recursive functions, functions without a termination proof, and guarded and unguarded co-recursive functions.

1.1 Context

Our approach to computer-assisted verification of functional programs combines three strands of research: (i) foundational frameworks based on partial functions and a separation of propositions and types, and the use of these foundational frameworks as programming logics for lazy functional programs, (ii) using ATPs for proving properties of functional programs by translating them into first-order logic and (iii) connecting ATPs to proof assistants based on Martin-Löf's type theory.

Based on a foundational framework proposed by Aczel [1977b], Dybjer [1985] proposed a programming logic for reasoning about programs using total and finite naturals numbers in a lazy functional program. Dybjer used a type-free partial language where the set of total and finite naturals numbers are represented by an inductive predicate. Dybjer's programming logic is compared with others programming logics in [Dybjer 1990]. Moreover, Dybjer and Sander [1989] proposed a higher-order programming logic for proving properties of lazy functional programs by induction and co-induction based on Park's higher-order μ-calculus [Park 1976]. Dybjer and Sander formalised the μ-calculus in the Isabelle proof assistant [Paulson 1994b]. Our programming logic for reasoning about programs in PCF using total and finite values is based on [Dybjer 1985] and our first-order programming logic for reasoning about programs in extensions of PCF using total finite and potentially infinite values is based on [Dybjer 1985; Dybjer and Sander 1989].

Some ideas behind this thesis arose during the CoVer (Combining Verification Methods in Software Development) project, a joint project involving the Programming Logic, Functional Programming and Formal Methods research groups at Chalmers University of Technology [Abel, Benke, Bove, Claessen et al. 2003–2005]. The goal of that project was to build a system for verifying Haskell programs using a combination of automatic and interactive theorem proving, and random testing. To its disposal the project already had several separate tools: for example, Agda 1 [Augustsson et al. 2004]—an earlier version of Agda—for interactive proof in dependent type theory, automatic theorem provers for classical first-order logic, and the random testing tool QuickCheck [Claessen and Hughes 2000]. We benefit from the experience which was accumulated during the CoVer project, in particular, that of using ATPs for proving properties of functional programs by translating them into first-order logic as proposed by Claessen and Hamon

in their unpublished work on the so-called "The CoVer Translator".

AgdaLight [Norell and Abel 2006] was an experimental version of Agda. The implementation of the Apia program took some ideas from the connection via a plug-in mechanism of AgdaLight to the Gandalf ATP [Tammet 1997], developed by Abel, Coquand and Norell [2005]. A related work to this connection is the translation of fragments of Martin-Löf's type theory to first-order logic [Tammet and Smith 1996].

1.2 Related Work

There are a few practical computer-assisted frameworks for the verification of *lazy* and *general* recursive functional programs.

Hip—the Haskell Inductive Prover [Rosén 2012]—is a tool to automatically prove properties about Haskell programs using structural induction, Scott's fixed-point induction or the approximation lemma. Hip proofs are based on the translation of (a subset of) Haskell programs into an intermediate language, which is then translated into first-order logic. The above higher-order principles of (co-)induction are handled by Hip at the meta-level, and the first-order reasoning is handled by off-the-shelf ATPs. In Hip's future work [Rosén 2012, § 3.4.3 and § 5.2.2], the possibility of using two predicates for representing total finite values and total potentially infinite values, respectively, is discussed. However, Rosén mentions the difficulty of using the predicate for total and finite values on values of different inductive data types, and the impossibility to express the totality of potentially infinite values in first-order logic without axiomatising set theory [Rosén 2012, § 3.4.3]. We have shown that by using inductive predicates, we can represent total finite values for different inductive data types (see § 5.3), and that by using co-inductive predicates (see § 5.5), we can express the totality of potentially infinite values based on a much simpler axiomatisation [Bove, Dybjer and Sicard-Ramírez 2012].

HipSpec is an automatic inductive theorem prover and theory exploration system for deriving and proving properties about Haskell programs [Claessen, Johansson et al. 2013]. HipSpec uses the automatic inductive prover Hip and QuickSpec, which automatically conjectures equations about a Haskell program using testing [Claessen, Smallbone and Hughes 2010]. By using both tools, HipSpec can use auxiliary lemmas conjectured by QuickSpec and proved by Hip to prove a given property. Although Hip supports (co-)inductive reasoning, HipSpec only supports the inductive one. In addition, since many proofs are too complex to be fully automated, there is ongoing work in integrating HipSpec with the Isabelle system [Johansson 2013].

The function package [Krauss 2010, 2013] provides support for general recursion in Isabelle/HOL [Nipkow, Paulson and Wenzel 2002]. Given a set of recursive equations for a function, the function package defines inductively

the graph of the function and a predicate representing it. From the graph of the function, the package defines an underspecified—assigning arbitrary values to elements outside the domain of the function—total function. The package automates (or partially automates) the definition of nested recursive functions (see § 7.1) or higher-order recursive functions (see § 7.2).

The Sledgehammer tool [Blanchette, Böhme and Paulson 2013; Blanchette and Paulson 2013] is a component of Isabelle/HOL that allows us to use off-the-shelf ATPs and Satisfiability Modulo Theories (henceforth, SMT) solvers [Barret et al. 2009] to prove properties arising in the construction of interactive proofs.

Zeno [Sonnex, Drossopoulou and Eisenbach 2012] is an inductive theorem prover for the automatic verification of properties of Haskell programs. Zeno only works with terminating functions and total and finite values.

One proposal for verifying Haskell programs was by translating them into Agda 1 [Abel, Benke, Bove, Hughes et al. 2005]. Different monads can be chosen for translating different Haskell programs. If a direct translation into Agda 1 is possible, then the `identity` monad is chosen. If the Haskell program terminates on a decidable subset of the input type, then the `Maybe` monad can be chosen. For the translation, the approach adopted by the authors was to use GHC's (The Glorious Glasgow Haskell Compilation System) reduction of full Haskell to its core language and then provide a translator from this core language into Agda 1. Potentially, other monads can be used for dealing with general recursion where the termination predicate is a priori undecidable, although this possibility was not explored in the above paper.

Sparkle [de Mol, van Eekelen and Plasmeijer 2002] is a tactic-based proof assistant for the lazy functional programming language Clean [Plasmeijer and van Eekelen 1999]. Sparkle's term language is Core-Clean—a subset of Clean—and its specification language is a first-order logic. To state properties of programs, the logic is extended with equalities on terms. Although predicates and relations cannot be expressed in the specification language, a subset of decidable ones can be modelled [de Mol, van Eekelen and Plasmeijer 2008; de Mol 2009]. Given a goal—a property that has to be proven—automation in Sparkle is based on a hint mechanism. This mechanism produces a list of applicable tactics and it assigns a score to every tactic in the list. If a tactic satisfies a threshold condition then this tactic is automatically applied. Since every type represents a different domain of quantification, the specification logic is a many-sorted logic. Also, because most of the off-the-shelf ATPs use unsorted first-order logic, it would not be possible to use them directly to automate the proof steps involving first-order or equational reasoning.

The aim of the Programatica project [Diatchki et al. 2001] was to integrate programming with reasoning in Haskell programs. The programming logic used is a higher-order modal μ-calculus called P-logic [W. L. Harrison and Kieburtz 2005]. Given a Haskell program embedding one or more properties expressed formally in P-logic, Plover—an automatic P-logic verifier [Kieburtz

2007]—attempts to find a proof, without user interaction, for the embedding properties. Unfortunately, it seems that work on the Programatica project has been discontinued.

A related tool for proving correctness of imperative programs is Why3 [Filliâtre and Paskevich 2013; Bobot et al. 2014]. The specification language is a typed first-order logic with several extensions such as recursive definitions, algebraic data types, pattern matching, (co-)inductive predicates, among others. The programming language can be seen as a dialect of ML [Milner et al. 1997] extended for supporting pre- and post-conditions annotations and restricted to first-order functions. The logical goals can be proved using automatic and interactive tools including some ATPs, SMT solvers and proof assistants. The programming language can also be used as an intermediate language for program verification (see, for example, Marché [2014]).

Many authors have considered the question of reasoning about *general* recursive (and possibly partial) functions in dependent type theories that only allow restricted forms of recursion. For an overview, the interested reader should consult [Bove, Krauss and Sozeau 2012].

1.3 Preliminaries

Notation. In order to avoid cumbersome notations, some symbols and names will be overloaded. For example, the names `zero` and `succ` will stand for the data constructors of the inductively defined type of natural numbers and for first-order logical terms of our programming logics, and the symbol $\bot$ will stand for the logical falsehood and for the bottom element in a certain domain. However, it will be clear from the context in which sense the symbols and names are used.

Haskell **code.** In the Haskell examples that we shall show, we make use of the data type of natural numbers defined by

```
data Nat = Zero | Succ Nat.
```

In addition, we use some relations and functions on natural numbers which we assume are defined in the usual way.

Source codes. The programs and examples described in this thesis are available as Git repositories at GitHub:

- The extended version of Agda: `https://github.com/asr/eagda`.
- The Apia program: `https://github.com/asr/apia`.
- The Agda implementation of our programming logics, some first-order theories and examples of verification of functional programs: `https://github.com/asr/fotc`.

The file `README.md` in the above repository contains the instructions for the installation and use of our programs and examples.

Acronyms list. Since we frequently use acronyms, we present here a list that contains the acronym, its definition and the page where it is defined for the first time.

ABP	Alternating bit protocol, p. 123
ATP	Automatic theorem prover for first-order logic, p. 2
FOL	Classical first-order logic with identity, p. 22
FOTC	First-order theory of combinators, p. 71
LF	Edinburgh Logical Framework, p. 22
LTC	Logical theory of constructions, p. 49
$\mathsf{LT}_{\mathsf{PCF}}$	Logical theory for PCF, p. 50
PA	Peano arithmetic, p. 37
SMT	Satisfiability Modulo Theories, p. 6

1.4 Overview of the Thesis

In Chapter 2, we present a brief introduction to Agda, our interactive proof assistant. This chapter may be skipped by readers familiar with Agda.

Chapter 3 describes our inductive approach to logical frameworks for representing first-order logic and theories. This chapter introduces the idea of having a logical framework which uses inductively defined data types and pattern matching in Agda.

We would like to emphasise that Chapter 2 is an incomplete survey of Agda and the inductive approach introduced in Chapter 3, although it is an interesting idea, it is a variation of the well-known logical framework idea. Therefore, we would like to warn to the reader that the main contributions of this thesis begins in the next chapter.

Chapter 4 defines and formalises our programming logic for reasoning about lazy PCF-programs.

In Chapter 5, we introduce and formalise a first-order version of the previous programming logic and we extend it to deal with (co-)inductive definitions of data types.

So far, all the proofs formalised in the previous chapters are interactive ones. Chapter 6 describes our approach for combining interactive and automatic proofs in first-order theories using our Apia program and our extended version of Agda.

In Chapter 7, we use the programming logic described in Chapter 5 and our approach for combining interactive and automatics proofs for verifying some general (co-)recursive programs including nested recursive functions,

higher-order recursive functions, functions without a termination proof, and guarded and unguarded co-recursive functions.

Chapter 8 contains our conclusions and future work.

Finally, Appendix A contains some definitions and theorems from domain theory used on this thesis. In Appendix B, we show the proofs of equivalence of two inductive principles associated to an inductive predicate. Appendix C contains the proofs of some streams properties. Appendix D contains the combined proofs of some properties used in the correctness proof of the `mirror` function. In Appendix E, we show a Haskell program for the alternating bit protocol. Appendix F contains the combined proofs of some properties used in the correctness proof of the alternating bit protocol.

Chapter 2

A Brief Introduction to Agda

Agda version 2.4.0.2 (released on 2014-07-29) is the latest proof assistant and dependently typed functional programming language in the ALF (see, for example, Magnusson and Nordström [1994]) and Agda families developed at Chalmers University of Technology and University of Gothenburg over the last twenty-five years. Agda is an interactive system for constructing proofs and programs, based on Martin-Löf's type theory (see, for example, Nordström, Petersson and Smith [1990]) and extended with records, parametrised modules, coverage and termination checkers, inductive families, among other features.

This chapter contains a brief introduction to Agda, explaining the features of Agda used in this thesis. A reader interested in learning more about Agda and how to use it for proofs and programming can look at the gentle introductions by Bove and Dybjer [2009], Norell [2009] and Abel [2009], and at the Agda website.[1] This chapter is based on these references. In § 2.1, we describe general features of the system and in § 2.2, we describe a set of combinators for equational reasoning.

As highlighted in the introduction of this thesis, this chapter may be skipped by readers familiar with Agda.

2.1 Agda's Features

Here we describe some features of Agda used in the formalisations developed in this thesis.

Construction of programs and proofs interactively. Writing programs and proofs in a system based on dependent types like Agda would be difficult without an *interactive* interface where the terms and the types can be refined with the aid of the type checker. Agda has an Emacs [Stallman et al. 2012] interface for this purpose.

[1] http://wiki.portal.chalmers.se/agda.

Proof terms. Martin-Löf's type theory (and Agda) is based on the propositions-as-types principle, also called the Curry-Howard correspondence (see, for example, Bove and Dybjer [2009] and Nordström and Smith [1984] for an introduction to this principle in the context of Agda and Martin-Löf's type theory, respectively), in which a proposition is identified with the type of its proofs. The colon ":" denotes type membership in Agda, that is, `a : A` denotes that `a` is a term of type `A`. We say that `a` is a proof term if the type `A` represents a proposition.

Unicode support. Agda supports Unicode characters. For example, we can write `A → B` instead of `A -> B` for the type of functions from `A` to `B`, or we can write `ℕ` instead of `N` for the type of natural numbers.

Infinite hierarchy of universes. In Agda, following terminology introduced by Martin-Löf [1984], types are called *sets*. Let ω be the set of the natural numbers; Agda has an infinite hierarchy of universes [Norell 2007b] $\mathbf{Set}_{i \in \omega}$, such that, $\mathbf{Set}_i$ `:` $\mathbf{Set}_{i+1}$, that is, $\mathbf{Set}_0$ is of type $\mathbf{Set}_1$, $\mathbf{Set}_1$ is of type $\mathbf{Set}_2$ and so on. The first universe $\mathbf{Set}_0$—called the universe of small types—is usually written as **Set**. The universe of small types will be the only universe necessary in our formalisations.

Remark 2.1. From a discussion on the Agda mailing list, it seems that the highest universe used in a "real" Agda development is $\mathbf{Set}_3$.[2]

Modules. Modules are used for organising names of Agda definitions. An Agda file does not needed one top-level module. If a top-level module is defined then the file name and the name of this module should be the same. Modules can be nested and parametrised.

Dependent function types. The dependent function type `(x : A) → B` denotes the type of functions taking an argument `x` of type `A` and returning a result of type `B`, where `x` may appear in `B` (Bove and Dybjer [2009] use the notation `(x : A) → B[x]`, which is more appropriate, but is not common in Agda literature).

A special case is when `x` is itself a type. For example, we can define an identity function for the small types by

```
id₁ : (A : Set) → A → A
id₁ A x = x.
```

The `id₁` function is a dependent function that takes a small type `A` and an element of `A` and returns the element.

For dependent function types, telescopic notation is allowed. For example, `(x : A) → (x' : A) → B` can be replaced by `(x x' : A) → B`. Moreover, we can also replace `(x : A) → (y : B) → C` by `(x : A)(y : B) → C`, and we can use the alternative notation `∀ x → A` instead of `(x : A) → B`, when the domain `A` can be deduced by the type checker.

[2] https://lists.chalmers.se/pipermail/agda/2012/004882.html.

λ-notation. Agda supports various notations for λ-abstraction. In this thesis, we shall use the notation `λ x → e`.

Using this notation, the above `id₁` function can be defined by

```
id₁ = λ A → λ x → x
```

or, by using an abbreviation, the function can be defined by

```
id₁ = λ A x → x.
```

Implicit arguments. Curly brackets "`{,}`" declare implicit arguments, that is, arguments that do not appear explicitly in the terms.

For example, by using implicit arguments the identity function for the small types can be defined by

```
id₂ : {A : Set} → A → A
id₂ x = x.
```

An implicit argument can be explicitly provided using the following syntax: `f {v}` gives `v` as the left-most implicit argument to `f` and `f {x = v}` gives `v` as the implicit argument called `x` to `f`. For example, we can define the `id₂` function by

```
id₂ {A} x = x.
```

By making an argument implicit does not mean Agda necessary will be able to infer it. In this case, the argument is highlighted in yellow when using (the standard configuration of) the Emacs interface.

Prefix, postfix, infix and mixfix operators. If the name of a function contains underscores "`_`" the function can be used as a prefix, postfix, infix or mixfix operator. The position of the underscore in the name of the function indicates the place of the argument. For example, `_+_` denotes an infix binary operator that can be used either in the form `m + n` or as an prefix operator in the form `_+_ m n`.

Indentation. In Agda, as in Haskell, indentation is important.

Associativity and precedence of operators. The keywords **`infix`**, **`infixl`** and **`infixr`** are used to declare the associativity and precedence of an operator. We shall show examples of the use of these keywords later.

Inductively defined types and families. Inductively defined types and families are introduced with the **`data`** construct. The formation rule says how we form a certain type from other types. The introduction rules say how elements of the type are constructed from other elements. These introduction rules are implemented by giving the name of the data constructors and their types.

For example, the type of natural numbers is defined by

```
data ℕ : Set where
  zero : ℕ
  succ : ℕ → ℕ.
```

This declaration introduces the new small type `ℕ` in the first line—the formation rule—with a nullary constructor `zero` and a unary, recursive constructor `succ`—the introduction rules.

Parametric inductive types are defined by providing a sequence of parameter declarations just after the name of the inductive type, that is, to the left of the colon. All the parameter names become implicit arguments to the data constructors.

For example, the type of lists parametrised by small types is defined by

```
infixr 5 _::_
data List (A : Set) : Set where
  []  : List A
  _::_ : A → List A → List A.
```

In this case, we defined that the type of the list former is

```
List : Set → Set
```

and we defined that the types of the data constructors are

```
[]  : {A : Set} → List A
_::_ : {A : Set} → A → List A → List A.
```

The constructor `[]` generates an empty list, and the constructor `_::_`, a right-associative constructor with precedence 5 (given by the line **`infixr`** `5 _::_`), generates a new list by adding an element of type `A` to a list of elements of type `A`.

The canonical example of an inductive family is the set of vectors, that is, lists of elements of a certain type `A` of a certain length `n`, is defined by

```
data Vec (A : Set) : ℕ → Set where
  []  : Vec A zero
  _::_ : {n : ℕ} → A → Vec A n → Vec A (succ n).
```

The type of `Vec` is **`Set`** `→ ℕ →` **`Set`** and the type of `Vec A` is `ℕ →` **`Set`**, which means that `Vec A` is a family of types indexed by the natural numbers; in other words, for each natural number `n`, `Vec A n` is the set of lists of elements of `A` of length `n`. Note that while the parameter `A` remains fixed for the whole list, the index `n` varies for each constructor. Note also that while the parameter is placed to the left of the colon, the index is placed to the right of it. The constructor `[]` generates a vector of length 0, and the constructor `_::_` generates a vector of length $n + 1$ by adding an element to a vector of length n.

Remark 2.2. Constructor names are not required to be unique, so we can use the same constructor names for vectors as we used for the `List` inductive data type.

Another example of an inductive family is the family of finite sets defined as follows:

```
data Fin : ℕ → Set where
  fzero : {n : ℕ} → Fin (succ n)
  fsucc : {n : ℕ} → Fin n → Fin (succ n).
```

This declaration introduces the family of types `Fin` indexed by the natural numbers. For each `n`, the type `Fin n` contains exactly `n` elements.

Structurally recursive functions and pattern matching. Like in other modern functional programming languages, we can define structurally recursive functions by pattern matching over elements that belong to an inductive data type.

For example, we can define addition of natural numbers by structural recursion and case analysis over the first argument by

```
_+_ : ℕ → ℕ → ℕ
zero   + n = n
succ m + n = succ (m + n)
```

and we can define the `map` function on lists of small types by

```
map : {A B : Set} → (A → B) → List A → List B
map f []       = []
map f (x ∷ xs) = f x ∷ map f xs.
```

Where clauses. We can use a `where` clause to introduce local definition. The local definition can be an abbreviation or a recursive definition by pattern matching.

Wild card pattern. The underscore "`_`" is a wild card pattern when matching patterns.

For example, the function `f : ℕ → ℕ` that returns `0` if its input is `0` and returns `1` otherwise can be defined by

```
f : ℕ → ℕ
f zero = zero
f _    = succ zero.
```

The absurd pattern. The absurd pattern "`()`" is used when there are no possible constructor patterns for a given argument.

For example, since there are not elements of type `Fin zero`

```
magic : {A : Set} → Fin zero → A
magic ()
```

is a function defined by using the absurd pattern.

Mutual definitions. Agda accepts several kinds of mutual definitions. In this thesis we shall use two of them, mutually inductive definitions of *types* and *families*, and mutually recursive definitions of *functions*.

For example, to define the mutually recursive functions `even` and `odd`, we first declare the functions

```
even : ℕ → Bool
odd  : ℕ → Bool
```

and we then write their definitions

```
even zero     = true
even (succ n) = odd n

odd zero     = false
odd (succ n) = even n.
```

As an example of mutually defined inductive data types, to define the types `EvenList` and `OddList` representing lists of natural numbers of even and odd lengths respectively, we first declare the inductive data types

```
data EvenList : Set
data OddList  : Set
```

and we then write their definitions

```
data EvenList where
  []  : EvenList
  _::_ : ℕ → OddList → EvenList

data OddList where
  _::_ : ℕ → EvenList → OddList.
```

Adding axioms. Agda allows the addition of certain constants, without actually defining them. In other words we can add axioms. The keyword **`postulate`** is used for this purpose.

For example, we can postulate the type of the natural numbers and some of its elements by

```
postulate
  ℕ : Set
  zero one two : ℕ.
```

Of course, after adding postulates the consistency of the system *relies on* the user.

Normalisation. By building interactive programs using the Emacs interface, Agda programs are constructed incrementally with "holes" for parts which have not yet been written. In the technical literature, a hole is a meta-variable—a variable ranging over terms—standing for a term which is not yet known. Agda performs type-checking and normalisation (simplification) for programs with holes in them.

We shall illustrate the normalisation process with the construction of a simple proof.

We start by defining the propositional equality on small types using an inductive family.

```
infix 4 _≡_
data _≡_ {A : Set} : A → A → Set where
  refl : {x : A} → x ≡ x.
```

Notation 2.3. We use the symbol "≡" for the propositional equality because the symbol "=" stands for Agda definitional equality.

Next, we define the `length` and `_++_` (concatenation) functions on lists.

```
length : {A : Set} → List A → ℕ
length []       = zero
length (x ∷ xs) = succ zero + length xs

infixr 5 _++_
_++_ : {A : Set} → List A → List A → List A
[]       ++ ys = ys
(x ∷ xs) ++ ys = x ∷ xs ++ ys.
```

We now want to prove that for all lists `xs` and `ys`, `length (xs ++ ys)` is equal to `length xs + length ys`.

We start by doing pattern matching on the list `xs`.

```
length-++ : {A : Set}(xs ys : List A) →
            length (xs ++ ys) ≡ length xs + length ys
length-++ []       ys = ?
length-++ (x ∷ xs) ys = ?.
```

In the Emacs interface, the symbol "?" denotes an Agda goal, that is, something that the programmer has left to do.

By using its type checker algorithm, Agda informs us that, before normalisation, the type of the first goal is

```
length ([] ++ ys) ≡ length [] + length ys.
```

By using the first equation in the definition of `_++_`, the type checker normalises the left hand side of this type to `length ys`, and by using the first equations in the definitions of `length` and `_+_`, the type checker normalises the right hand side of this type to `length ys`. In other words, after normalisation, the type of the first goal becomes `length ys ≡ length ys`, which can be proved using `refl`.

Now,

```
length ((x :: xs) ++ ys) ≡ length (x :: xs) + length ys
```

is the type of the missing goal before normalisation. By using the second equations in the definitions of `_++_` and `length`, the left hand side of this type is normalised to `succ (length xs ++ ys)`, and by using the second equations in the definitions of `_+_` and `length`, the right side of this type is normalised to `succ (length xs + length ys)`. In other words, the type of the goal becomes

```
succ (length (xs ++ ys)) ≡ succ (length xs + length ys)
```

which can be proved using the inductive hypothesis and a proof that, for all natural numbers `m` and `n`, if `m ≡ n` then `succ m ≡ succ n`.

```
succCong : {m n : ℕ} → m ≡ n → succ m ≡ succ n
succCong refl = refl

length-++ [] ys = refl
length-++ (x :: xs) ys = succCong (length-++ xs ys).
```

Coverage and termination checkers. Since Agda can be used as a theorem prover, it must be logically consistent and hence, all the defined functions should be *terminating*. For this purpose, Agda performs a coverage check for pattern matching definitions and a termination check for (co-)recursive calls; both checks are purely syntactical.

For example, Agda's coverage checker rejects the partial function

```
head : {A : Set} → List A → A
head (x :: xs) = x
```

due to the missing pattern matching on the list constructor `[]`.

Agda's termination checker (and Martin-Löf's type theory) accepts functions which are structurally recursive in one argument at a time, as in the functions showed so far. The termination checker, under certain conditions, also accepts structurally recursive functions on several arguments simultaneously or permuted arguments [Lee, Jones and Ben-Amram 2001; Abel and Altenkirch 2002].

For example, the Ackermman function

```
ack : ℕ → ℕ → ℕ
ack zero     n        = succ n
ack (succ m) zero     = ack m (succ zero)
ack (succ m) (succ n) = ack m (ack (succ m) n)
```

is accepted by Agda's termination checker since it is structurally recursive with respect to the lexicographic ordering on $\mathbb{N} \times \mathbb{N}$.

If Agda's termination checker cannot see that a function is terminating it will be highlighted in light salmon when using (the standard configuration of) the Emacs interface.

The Agda options `--no-coverage-check` and `--no-termination-check` disable the coverage and termination checkers, respectively. However, since we use Agda as a theorem prover, these options are not used in our formalisations.

2.2 Combinators for Equational Reasoning

Given the propositional equality for small types `_≡_` defined in the previous section, to directly construct a proof of a proposition like `s ≡ t` using the properties of `_≡_`—reflexivity, symmetry, transitivity and substitutivity—can be very tedious, and the resulting proof can be difficult to read.

By using Agda facilities for implicit arguments, mixfix operators and Unicode, it is possible to define a set of combinators which allows us to construct a proof term of type `s ≡ t` using an algebraic style. In fact, these combinators can be defined for any binary relation that is both reflexive and transitive, that is, a preorder [Norell 2007b; Mu, Ko and Jansson 2009].

Given a preorder `_~_`

```
A       : Set
_~_     : A → A → Set
~-refl  : ∀ {x} → x ~ x
~-trans : ∀ {x y z} → x ~ y → y ~ z → x ~ z
```

we define the following set of combinators for preorder reasoning, parametrised by the preorder `_~_` [Mu, Ko and Jansson 2009]:

```
infixr 5 _~⟨_⟩_
infix  5 _∎

_~⟨_⟩_ : ∀ x {y z} → x ~ y → y ~ z → x ~ z
_ ~⟨ x~y ⟩ y~z = ~-trans x~y y~z

_∎ : ∀ x → x ~ x
_∎ _ = ~-refl.
```

As Mu, Ko and Jansson [2009] point out, the combinator `_∎` takes a term `e` and produces a proof of `e ~ e` using the reflexivity of the preorder `_~_`. In addition, the combinator `_~⟨_⟩_` takes three explicit arguments: `e` on the left, a proof that `e ~ e'` in the angle brackets, and a proof of `e' ~ e''` on the right. This combinator produces a proof of `e ~ e''` using the transitivity of `_~_`.

Using the combinators for preorder reasoning, a chain reasoning on the preorder `_~_` will be formalised as follow:

```
e₁ ~⟨ reason₁ ⟩
   ⋮
eₙ₋₁ ~⟨ reasonₙ₋₁ ⟩
eₙ ∎
```

where `reasonᵢ : eᵢ ~ eᵢ₊₁`.

Given the associativity and precedence of the combinators, the above chain reasoning is bracketed as

```
e₁ ~⟨ reason₁ ⟩ ... (eₙ₋₁ ~⟨ reasonₙ₋₁ ⟩ (eₙ ∎)).
```

The following example shows the use of the combinators for equational reasoning.

Example 2.4. Let `_*_ : ℕ → ℕ → ℕ` be the multiplication operation for natural numbers. Given the properties

```
*-comm          : ∀ m n → m * n ≡ n * m
*-rightIdentity : ∀ n → n * succ zero ≡ n
```

we can prove that `succ zero` is the left-identity for the multiplication using the combinators for preorder and the fact that the propositional equality `_≡_` is a preorder, in other words, we can parametrise these combinators with `_≡_`.

Renaming the combinator `_~⟨_⟩_` as `_≡⟨_⟩_`, the proof is given by

```
*-leftIdentity : ∀ n → succ zero * n ≡ n
*-leftIdentity n = succ zero * n ≡⟨ *-comm (succ zero) n ⟩
                   n * succ zero ≡⟨ *-rightIdentity n ⟩
                   n             ∎
```

Chapter 3

Using Agda with Data and Pattern Matching as a Logical Framework

In our approach to the computer-assisted verification of functional programs, we use Agda to formalise our programing logics. In this chapter, we describe how to use Agda as a logical framework—a meta-language for the formalisation of deductive systems [Pfenning 2002]—for first-order logic and theories with equality. The first step in the representation of a first-order theory is the representation of the underlying logic, for which we explore two ways. In § 3.1, we set up our notation and conventions for first-order logic. Moreover, we formalise first-order logic using the Edinburgh Logical Framework approach. This is a survey of known techniques and how they are implemented in Agda. In § 3.2, we introduce our inductive approach to logical frameworks by formalising first-order logic using Agda's support for inductive notions.

One of the main strengths of Agda is its support for writing proofs, which we shall call Agda's *proof engine* and it consists of: (i) support for inductively defined types, including inductive families, and function definitions using pattern matching on such types, (ii) normalisation during type-checking, (iii) commands for refining proof terms, (iv) coverage checker and (v) termination checker. The inductive approach for representing first-order logic is better because we benefit from Agda's proof engine.

We represent a first-order theory using the inductive formalisation of first-order logic in § 3.2 as the underlying logic, for the reasons explained above. For representing the non-logic axioms of the theory, we can also follow two ways: adding postulates or using inductive notions. In § 3.3, we formalise first-order theories. In particular, we formalise group theory and Peano arithmetic by adding postulates for representing their non-logic axioms, and we also formalise Peano arithmetic using inductive notions for

representing its non-logical axioms.

In § 3.4, we discuss the adequacy problem of our Agda inductive representation of first-order logic and theories.

3.1 Edinburgh Logical Framework Approach for Representing First-Order Logic

We can represent a first-order theory in Agda along the lines of the Edinburgh Logical Framework (henceforth, LF). Like Agda, LF is a λ-calculus with dependent types and a few universes. LF has two universes, *types* and *kinds*. As we have already mentioned in § 2.1, Agda has an infinite hierarchy of universes, so we can use the first two to implement LF types and kinds, but **`Set`**—its first universe—is all that is necessary to represent first-order theories. A formal system is implemented in LF by adding constants of the appropriate types. Some constants implement the syntactic constructs (for example term formers, predicate and function symbols) of the formal system, others implement proof terms for its axioms and inference rules. In this way, we can use Agda as a logical framework in essentially the same manner as we can use LF.

Remark 3.1. Agda's underlying logical framework is Martin-Löf's logical framework [Nordström, Petersson and Smith 1990, Part III]. In relation to the representation of FOL using the LF-approach both LF and Martin-Löf's logical framework are the same. But in other situations they lead to different implementations, since they formalise new equations differently. In Martin-Löf's logical framework, we are allowed to add new equations as equality judgments, whereas in LF this is not allowed. For example, let `A` be a type. In Martin-Löf's logical framework, we can add the equation `l = r : A`, but in LF this equation has to be implemented by adding a constant `e : I l r`, where `I` represents a propositional equality for `A`.

We start by following the LF-approach for representing classical first-order logic with identity (henceforth, FOL) in Agda. In the representation of FOL using Agda's support for inductive notions in § 3.2, we shall define classical logic as intuitionistic logic with the principle of the excluded middle. Each logical constant will be defined by a **`data`** declaration in Agda which inductively generates the proof terms for the proposition in question. This yields intuitionistic logic. The principle of the excluded middle will be added afterwards as an axiom. This is the reason for the particular selection of the logical constants in the following grammar, which generates the terms and formulae of FOL.

$$
\begin{array}{rll}
\textsf{Terms} \ni t ::= & x & \text{variable} \\
& \mid c & \text{constant} \\
& \mid f(t, \ldots, t) & \text{function} \\
\\
\textsf{Formulae} \ni A ::= & \top \mid \bot & \text{truth, falsehood} \\
& \mid A \supset A \mid A \wedge A \mid A \vee A & \text{binary logical connectives} \\
& \mid \forall x.A \mid \exists x.A & \text{quantifiers} \\
& \mid t = t & \text{equality} \\
& \mid P(t, \ldots, t) & \text{predicate} \\
\\
\textsf{Abbreviations} \quad \neg A \stackrel{\text{def}}{=} & A \supset \bot & \text{negation} \\
t \neq t' \stackrel{\text{def}}{=} & \neg(t = t') & \text{inequality}
\end{array}
$$

Conventions 3.2. The logical connectives obey the standard precedence rules: $\neg$ has higher precedence than $\wedge$, $\wedge$ higher than $\vee$ and $\vee$ higher than $\supset$. In addition, they have higher precedence than the quantifiers. Conditional is right-associative, so, for example, $A_1 \supset A_2 \supset \cdots \supset A_n$ should be read $A_1 \supset (A_2 \supset \cdots \supset A_n)$.

Following the LF-approach, we introduce each logical constant as a type former, and each axiom and inference rule as a constants of the corresponding type. For the purpose of the representation of FOL, we need no more than a subset of all Agda expressions in normal form, with a few type formers and constants. We use Agda normal forms because our Apia program works with the Agda internal representation of expressions (see § 6.3), which is in normal form.

The subset of Agda expressions required is generated by the following grammar:

$$
\begin{array}{rll}
\textsf{Normal Forms} \ni a ::= & x \ a \ \cdots \ a & \text{variable applied to terms} \\
& \mid c \ a \ \cdots \ a & \text{constant applied to terms} \\
& \mid \lambda x.a & \lambda\text{-abstraction} \\
& \mid (x : a) \to a & \text{dependent function type}
\end{array}
\tag{3.1}
$$

Note that Agda has a common syntactic category of types and terms. The required type formers and constants are

$$\{\mathrm{N}_0, \mathrm{N}_1, +, \times, \Sigma, \mathrm{I}, f^*, P^*, \mathrm{D}\}, \tag{3.2}$$

where N_0 represents the empty type, N_1 the unit type, $+$ the disjoint union type, $\times$ the Cartesian product type, Σ the dependent product type and I the

identity type. The symbol f^* represents a function symbol f, P^* a predicate symbol P and D the domain of quantification (or domain of discourse). Following the propositions-as-types principle, we translate both terms and formulae of FOL into Agda expressions by [Martin-Löf 1998]

$$\begin{aligned}
\textsf{Terms} \qquad & x \mapsto x^* : \mathrm{D} \\
& c \mapsto c^* : \mathrm{D} \\
& f(t_1, \ldots, t_n) \mapsto f^*\, t_1^*\, \cdots\, t_n^* \\
\\
\textsf{Formulae} \qquad & \bot \mapsto \mathrm{N}_0 \\
& \top \mapsto \mathrm{N}_1 \\
& A \vee B \mapsto A^* + B^* \\
& A \wedge B \mapsto A^* \times B^* \\
& A \supset B \mapsto A^* \to B^* \\
& \forall x.A \mapsto (x^* : \mathrm{D}) \to A^* \\
& \exists x.A \mapsto \Sigma\ \mathrm{D}\ (\lambda x^*.A^*) \\
& t = t' \mapsto \mathrm{I}\ \mathrm{D}\ t^*\, t'^* \\
& P(t_1, \ldots, t_n) \mapsto P^*\, t_1^*\, \cdots\, t_n^*
\end{aligned} \tag{3.3}$$

Notation 3.3. We shall use a logical notation instead of a type-theoretic notation in our implementation of the type formers and constants (3.2). In particular, we shall use the symbol `⊥` for the empty type N_0, the symbol `⊤` for the unit type N_1, the symbol `∨` for the disjoint union type $+$, the symbol `∧` for the Cartesian product type $\times$, the symbol `∃` for the dependent product type Σ on the domain D and the symbol `≡` for the identity type I on the domain D.

Now, we explain our formalisation of FOL showing the implementation of some logical constants. In order to implement the LF-approach in Agda, we shall postulate each required type former and constant (3.2). In Agda, first-order formulae will be represented by the type **`Set`**. We start with the implementation of *disjunction*. The inference rules for disjunction in Gentzen's natural deduction (see, for example, van Dalen [2004]) are given by (3.4), where $\vee\mathrm{I}_1$ and $\vee\mathrm{I}_2$ are the introduction rules, and $\vee\mathrm{E}$ is the elimination rule.

$$\frac{A}{A \vee B}\,(\vee\mathrm{I}_1) \qquad \frac{B}{A \vee B}\,(\vee\mathrm{I}_2) \qquad \frac{\begin{array}{ccc} & [A] & [B] \\ & \vdots & \vdots \\ A \vee B & C & C \end{array}}{C}\,(\vee\mathrm{E}) \tag{3.4}$$

Following the LF-approach and based on the FOL to Agda translation (3.3),

we implement *disjunction* by the disjoint union type.

```
postulate
  _∨_  : Set → Set → Set
  inj₁ : {A B : Set} → A → A ∨ B
  inj₂ : {A B : Set} → B → A ∨ B
  case : {A B C : Set} → (A → C) → (B → C) → A ∨ B → C.
```
(3.5)

The first constant `_∨_` implements the formation rule. This constant represents disjunction by an infix operator that takes two formulae (that is, two small types) and returns the formula (that is, an element in **Set**) of proofs representing the disjunction of the two formulae. The second constant `inj₁` implements the introduction rule $\vee\mathrm{I}_1$ in (3.4). Given two small types `A` and `B`, the constant `inj₁` states that if we have a proof of `A`, then we can construct a proof of the disjunction of `A` and `B`. The third constant `inj₂` implements the introduction rule $\vee\mathrm{I}_2$ in (3.4). The constant `inj₂` is similar to the previous one, but in this case we start from a proof of `B` instead of a proof of `A`. The last constant `case` implements the elimination rule $\vee\mathrm{E}$ in (3.4). Given three formulae `A`, `B` and `C`, the constant `case` constructs a proof of `C` from a method that takes a proof of `A` into a proof of `C`, a method that takes a proof of `B` into a proof of `C`, and a proof of the disjunction of `A` and `B`.

Remark 3.4. The inference rules $\vee\mathrm{I}_1, \vee\mathrm{I}_2$ and $\vee\mathrm{E}$ are logical *schemata*—sets of first-order formulae—one for each instance of A, B and C. Because Agda is a higher-order logic, we could represent each schema by one dependent function type using implicit quantification over **Set**. In § 6.3.1, we shall discuss how the Apia program deals with logical schemata.

Example 3.5. Using the implementation of disjunction in (3.5), the proof of the commutativity of disjunction is represented by the following Agda definition

```
∨-comm : {A B : Set} → A ∨ B → B ∨ A
∨-comm = case inj₂ inj₁.
```

Now, to implement *classical* FOL—it will be the underlying logic for our programming logics because most ATPs implement it—we postulate the *principle of the excluded middle*. Recalling that $\neg A$ is an abbreviation of $A \supset \bot$, that is,

```
postulate ⊥ : Set


¬_ : Set → Set
¬ A = A → ⊥
```

the implementation of the principle of the excluded middle is given by

```
postulate pem : {A : Set} → A ∨ ¬ A.
```

We now continue with the implementation of the quantifiers and the FOL-equality. First, we need to postulate the existence of the *domain of quantification*.

```
postulate D : Set.
```

As can be seen in (3.3), the *universal quantifier* on the domain D is implemented by the dependent function type `(x : D) → A`. We shall use the alternative notation `∀ x → A` when the domain D can be deduced by Agda's type checker.

The introduction rule ∃I and the elimination rule ∃E for the *existential quantifier* are given by (3.6), with the usual side condition for the rule ∃E, that is, x is not free in B or in any of the assumptions of the proof of B other than $A(x)$ (see, for example, van Dalen [2004]).

$$\frac{A(t)}{\exists x.A(x)}\ (\exists\mathrm{I}) \qquad\qquad \frac{\exists x.A(x) \qquad \begin{array}{c}[A]\\ \vdots \\ B\end{array}}{B}\ (\exists\mathrm{E}) \tag{3.6}$$

The existential quantifier on the domain D is implemented by the dependent product type.

```
postulate
  ∃       : (A : D → Set) → Set
  _,_     : {A : D → Set}(t : D) → A t → ∃ A
  ∃-elim : {A : D → Set}{B : Set} → ∃ A → (∀ {x} → A x → B) → B.
```

The formation rule for the existential quantifier is implemented by the operator `∃` that takes a propositional function and returns a small type representing the proofs of the corresponding existential proposition. The constant `_,_` implements the introduction rule ∃I and the constant `∃-elim` implements the elimination rule ∃E in (3.6). The constant `_,_` establishes that a proof of an existential proposition is a pair whose first component is a term of type `D`, called the *witness*, and whose second component is a proof that the witness satisfies the proposition. Given proofs of both an existential proposition `∃ A` and a method that constructs a proof of `B` from a proof of `A x`, the constant `∃-elim` returns a proof of the formula `B`. Note that the side condition in (3.6) is implemented by the type of the constant `∃-elim`. Note also that the rules ∃I and ∃E are logical schemata too—there is an instance for each propositional function—which are represented using an implicit quantification over the type of propositional functions.

Notation 3.6. It is possible to replace `∃ (λ x → e)` by `∃[ x ] e` using Agda's support for declaring user-defined syntax available since Agda version 2.2.8 (released on 2010-09-27). We use this notation in this thesis.

The following example shows the use of the implementation of the quantifiers for proving a FOL theorem.

Example 3.7. Let $A(x, y)$ be a propositional function. The proof of

$$\exists x.\forall y.A(x, y) \supset \forall y.\exists x.A(x, y),$$

is represented by

```
∃∀ : {A : D → D → Set} → ∃[ x ](∀ y → A x y) → ∀ y → ∃[ x ] A x y
∃∀ h y = ∃-elim h (λ {x} ah → x , ah y).
```

In the above proof, we applied the elimination rule `∃-elim` to the existential hypothesis `h` of type `∃[ x ](∀ y → A x y)`, and to a proof of `∃[ x ] A x y`. The last proof is based on the auxiliary hypothesis `∀ y → A x y`.

Remark 3.8. We emphasise that in our implementation of FOL in Agda, it is only a subset of the Agda types and terms that correspond to FOL-formulae and FOL-proofs, respectively. Our implementation of the existential quantifier enables the construction of Agda terms which do not represent anything in FOL.

For example, the use of the elimination rule `∃-elim` makes it possible to build the following term:

```
non-FOL : {A : D → Set} → ∃ A → D
non-FOL h = ∃-elim h (λ {x} _ → x).
```

This term does not represent anything in FOL, simply because in FOL there is no such thing as a function from a formula to an element in the domain. This issue is related to logical frameworks that handle proof terms. Our implementation makes it possible to access the first argument of a proof term of an existential proof, and we can use this proof term to build an element of D.

Unfortunately, Agda cannot tell us if we are using our implementation of the existential quantifier strictly inside FOL, so it is the user's responsibility to avoid the non-FOL uses of our implementation.

Remark 3.9. When working with classical logic and the elimination rule for the existential quantifier, it is necessary to keep in mind that the term of type `D` obtained by using the elimination `∃-elim` on an existential proof using the principle of the excluded middle is not a witness in the sense of intuitionistic logic.

Remark 3.10. In FOL, the domain of quantification is non-empty (see, for example, Mendelson [1997]), but this is not the case in our implementation. For example, let $A(x)$ be a propositional function, the theorem

$$\forall x.A(x) \supset \exists x.A(x) \tag{3.7}$$

cannot be proved in our implementation of FOL without postulating an element `d` of type `D`. This will not be an issue in our programming logics because their domains of quantification will be non-empty.

Now, we continue with the implementation of the FOL-*equality* (called propositional equality). The introduction rule (reflexivity of equality) and the elimination rule (substitutivity property) for the propositional equality are given by

$$\frac{}{x = x}\ (\text{refl}) \qquad\qquad \frac{x = y \qquad A(x)}{A(y)}\ (\text{subst})$$

The propositional equality is implemented by the identity type over the domain D.

```
postulate
  _≡_   : D → D → Set
  refl  : ∀ {x} → x ≡ x
  subst : (A : D → Set) → ∀ {x y} → x ≡ y → A x → A y.
```

The formation rule for the propositional equality over the domain `D` is represented by the predicate `_≡_` that takes two elements in `D` and returns the set of proofs that the two elements of `D` are equal. Using the introduction rule `refl`, we can only construct proofs that an element is equal to itself. The elimination rule `subst` states that, if two elements of `D` are equal, and if we are able to prove an arbitrary propositional function `A` over `D` for the first of these two elements, then we can produce a proof that the second element also satisfies the propositional function `A`.

Using the implementation of the propositional equality, we can prove that it is an equivalence relation, that is, the propositional equality is a reflexive, symmetric and transitive relation. In the next example, we show the proof of symmetry.

Example 3.11. We prove that the propositional equality is a symmetric relation.

```
sym : ∀ {x y} → x ≡ y → y ≡ x
sym {x} h = subst (λ t → t ≡ x) h refl.
```

The proof of symmetry uses the elimination rule for equality—`subst`—which requires three arguments. The first is the propositional function `λ t → t ≡ x`. The second is a proof that `x ≡ y`, given by the hypothesis. The third argument is a proof of the propositional function `λ t → t ≡ x` applied to `x`, that is, `x ≡ x`, which is an instance of reflexivity of the equality.

We could use the LF-approach—postulating each logical constant as a type former, and its introduction and elimination rules as constants of the corresponding types—for implementing the missing logic constants.

3.2 Inductive Approach for Representing First-Order Logic

The main disadvantage of representing FOL (and first-order theories) using the LF-approach above is that it makes limited use of Agda's support for writing proofs. To benefit from Agda's support for inductive notions, which it is not possible when we use postulates, we represent FOL using this support.

When using inductive notions for representing FOL, we need to choose between using a *shallow* or a *deep* embedding (see, for example, Garillot and Werner [2007]). In a shallow embedding, the FOL-formulae are written directly using the constants underlying Agda as a logical framework (see § 3.4.1). On the other hand, in a deep embedding, the FOL-formulae are represented using an inductive data type. In this thesis, we use a shallow embedding for representing FOL because: (i) this avoid us explicitly deal with binding of variables, (ii) we do not prove meta-theorems about FOL and (iii) we do not need to reason by induction over the structure of FOL.

Although intuitionistic logical constants are naturally inductive notions, since their sets of canonical proofs are inductively generated by the introduction rules, this is not the case for classical logic. The logical constants of classical logic are not normally inductive notions in the same way. However, since classical logic is intuitionistic logic plus the principle of the excluded middle, we can use the implementation of intuitionistic logical constants as inductive notions and then add the principle of the excluded middle as an axiom.

In our inductive approach to logical frameworks, the formation and the introduction rules of the logical constants are represented by inductive types, and their elimination rules will not be postulated, but instead defined by pattern matching. All these rules have the same type than in the LF-approach. Moreover, we shall use postulates in this approach when necessary.

In the inductive approach, the formation and the introduction rules for *disjunction* are defined by the parametric inductive data type

```
data _∨_ (A B : Set) : Set where
  inj₁ : A → A ∨ B
  inj₂ : B → A ∨ B.
```

As we already mentioned in § 2.1, the above declaration introduce three constants to the theory

```
_∨_  : Set → Set → Set
inj₁ : {A B : Set} → A → A ∨ B
inj₂ : {A B : Set} → B → A ∨ B
```

that is, it gives us direct access to the formation and introduction rules for disjunction in exactly the same way as the LF-approach (see Eq. 3.5). On

the other hand, we do not get the elimination rule automatically. This rule has to be defined separately by pattern matching on a proof of `A ∨ B`

```
case : {A B C : Set} → (A → C) → (B → C) → A ∨ B → C
case f g (inj₁ a) = f a
case f g (inj₂ b) = g b.
```

Here, the type of the new constant `case` is the same as in the LF-representation of FOL (see Eq. 3.5). In addition, we now get two proof normalisation equations, not present in the LF-representation. However, these equations do not change the set of provable formulae.

The formation and the introduction rules for the *existential quantifier* are defined by the parametric inductive type

```
data ∃ (A : D → Set) : Set where
  _,_ : (t : D) → A t → ∃ A.
```

Once again, we need to define the elimination rule. We define this rule by pattern matching on a proof of `∃ A`

```
∃-elim : {A : D → Set}{B : Set} → ∃ A → (∀ {x} → A x → B) → B
∃-elim (_ , Ax) h = h Ax.
```

The formation and the introduction rules for the FOL-*equality*—the identity type on D—are defined by the inductive family

```
data _≡_ (x : D) : D → Set where
  refl : x ≡ x
```

and its elimination rule is defined using pattern matching on a proof that `x ≡ y`

```
subst : (A : D → Set) → ∀ {x y} → x ≡ y → A x → A y
subst A refl Ax = Ax.
```

Since the types of the formation, introduction and elimination rules of the logical constants defined using the inductive and the LF-approaches have the same type, the proofs of the theorems in Examples 3.5, 3.7, and 3.11 are the same using the inductive approach.

If we only use pattern matching to define the elimination rules associated with the logical constants, we arrive at an Agda implementation of FOL that should be equivalent to the LF-implementation (see § 3.4.1). However, to make full use of Agda's support for proof by pattern matching, we shall not restrict ourselves to using the elimination rules above. Instead, we shall allow proofs by pattern matching in general, as long as they are accepted by Agda's coverage and termination checker. Strictly speaking we are obliged to show that such uses of pattern matching can be reduced to the standard elimination rules. For most of our uses of pattern matching this is obvious,

but it would be hard to show a general theorem about the reduction of Agda's general notion of pattern matching to elimination rules (see § 3.4).

Using Agda's full support for proofs by pattern matching we can rewrite the proofs in Examples 3.5, 3.7 and 3.11 as it is shown by the following example.

Example 3.12. The proof of the commutativity of disjunction is given by

```
v-comm : {A B : Set} → A ∨ B → B ∨ A
v-comm (inj₁ a) = inj₂ a
v-comm (inj₂ b) = inj₁ b
```

where we pattern match on a proof that `A ∨ B`.

The proof of the theorem in Example 3.7 is given by

```
∃∀ : {A : D → D → Set} → ∃[ x ](∀ y → A x y) → ∀ y → ∃[ x ] A x y
∃∀ (x , Ax) y = x , Ax y
```

where we pattern match on a proof that `∃[ x ](∀ y → A x y)`.

Finally, if we pattern match on a proof that `x ≡ y`, the proof that the FOL-equality is a symmetric relation is given by

```
sym : ∀ {x y} → x ≡ y → y ≡ x
sym refl = refl.
```

The main difference between the three proofs above and those using the LF-approach is that we did not need to use the elimination rules of the logical constants. In addition, observe that, thanks to the pattern matching facility feature, the proofs presented here look simpler.

The representation of FOL using the propositions-as-types principle and Agda's inductive notions is well known (see Bove and Dybjer [2009], for a gentle introduction to how this can be done) and therefore we omit a detailed presentation of the missing logic constants using the inductive approach. The complete representation of FOL used in this thesis is shown in Fig. 3.1.

3.3 Inductive Representation of First-Order Theories

Using the inductive representation of FOL described in § 3.2 as the underlying logic, we can represent (the non-logical axioms of) first-orders theories in two ways: (i) by adding postulates, that is, we formalise a first-order theory representing its domain of quantification, its signature elements, and its axioms by postulated constants of the appropriate types, or (ii) by using inductive notions, that is, we formalise a first-order theory using an inductive data type to represent the domain of quantification, and functions defined

Falsehood	`data ⊥ : Set where`
	
	`⊥-elim : {A : Set} → ⊥ → A`
	`⊥-elim ()`

Truth	`data ⊤ : Set where tt : ⊤`

Disjunction	`data _∨_ (A B : Set) : Set where`
	`  inj₁ : A → A ∨ B`
	`  inj₂ : B → A ∨ B`
	
	`case : ∀ {A B} → {C : Set} →`
	`       (A → C) → (B → C) → A ∨ B → C`
	`case f g (inj₁ a) = f a`
	`case f g (inj₂ b) = g b`

Conjunction	`data _∧_ (A B : Set) : Set where`
	`  _,_ : A → B → A ∧ B`
	
	`∧-proj₁ : ∀ {A B} → A ∧ B → A`
	`∧-proj₁ (a , _) = a`
	
	`∧-proj₂ : ∀ {A B} → A ∧ B → B`
	`∧-proj₂ (_ , b) = b`

Conditional	`A → B` (non-dependent function type)
Negation	`¬_ : Set → Set` `¬ A = A → ⊥`
Principle of the excluded middle	`postulate pem : ∀ {A} → A ∨ ¬ A`
Domain of discourse	`postulate D : Set`
Universal quantifier	`(x : D) → A` (dependent function type)

Existential quantifier	`data ∃ (A : D → Set) : Set where`
	`  _,_ : (t : D) → A t → ∃ A`
	
	`∃-elim : {A : D → Set}{B : Set} →`
	`         ∃ A → (∀ {x} → A x → B) → B`
	`∃-elim (_ , Ax) h = h Ax`

Equality	`data _≡_ (x : D) : D → Set where refl : x ≡ x`
	
	`subst : (A : D → Set) → ∀ {x y} → x ≡ y →`
	`        A x → A y`
	`subst A refl Ax = Ax`

Fig. 3.1: Representation of FOL using Agda's inductive notions.

by structural recursion to represent the operations. As a result of using the inductive representation of FOL and the possibility of using postulates for representing the non-logical axioms of a theory in (i), our inductive representation of first-order theories has postulates. In § 3.3.1, we formalise group theory using (i). In § 3.3.2 and § 3.3.3, we apply (i) and (ii) to formalise Peano arithmetic, respectively, and compare them.

Remark 3.13. We shall use the combinators from § 2.2 for implementing the equational reasoning in first-order theories with equality, where we rename the combinator `_~⟨_⟩_` as `_≡⟨_⟩_`.

3.3.1 Inductive Representation of Group Theory: Using Postulates for Representing the Non-Logical Axioms

We show an example of adding postulates to the inductive representation of FOL for representing a first-order theory: group theory.

Let $\mathfrak{L} = \{\cdot, ^{-1}, \varepsilon\}$ be the formal language of group theory, where $\cdot$ is a left-associative binary function symbol (multiplication operation), $^{-1}$ is a unary function symbol (inverse function), and ε is a constant symbol (identity element with respect to the multiplication operation). The theory of groups has the following non-redundant axioms (see, for example, Hodges [1993]):

$$\forall a\ b\ c.\ a \cdot b \cdot c = a \cdot (b \cdot c) \qquad \text{(associativity)}$$
$$\forall a.\ \varepsilon \cdot a = a \qquad \text{(left-identity)}$$
$$\forall a.\ a^{-1} \cdot a = \varepsilon \qquad \text{(left-inverse)}$$

Let us consider FOL as in Fig. 3.1 where the domain of quantification is called G. We start our formalisation of group theory by postulating the existence of an identity element, a left-associative product operation and an inverse function.

```
infix  11 _⁻¹
infixl 10 _·_

postulate
  ε   : G
  _·_ : G → G → G
  _⁻¹ : G → G.
```

Using the above signature, the axioms of group theory are implemented by

```
postulate
  assoc       : ∀ a b c → a · b · c ≡ a · (b · c)
  leftIdentity : ∀ a → ε · a ≡ a
  leftInverse  : ∀ a → a ⁻¹ · a ≡ ε.
```
(3.8)

In the following example, we show the formalisation of a textbook-style proof.

Example 3.14. Let us consider Mac Lane and Birkhoff's [1999] proof that a right-identity element is unique.

Theorem. Let $(\mathrm{G}, \cdot, ^{-1}, \varepsilon)$ be a group. Any right-identity r is equal to ε, that is, for all $a \in \mathrm{G}$, $a \cdot r = a$ implies $r = \varepsilon$.

Proof.

1.	$r = \varepsilon \cdot r$	(left-identity)
2.	$\varepsilon \cdot r = \varepsilon$	(by hypothesis, r is a right-identity)
3.	$r = \varepsilon$	(transitive property for equality in 1, 2) $\square$

Proofs in mathematical textbooks, although rigorous, often leave some trivial details for the reader to complete. In this case, the proof omits the use of the symmetric property for the equality in step 1. In our formalisation, we need to be explicit about this step and therefore we use symmetry on a proof that `e · r ≡ r`, which is the `leftIdentity` postulate.

```
rightIdentityUnique : ∀ r → (∀ a → a · r ≡ a) → r ≡ ε
rightIdentityUnique r h = trans (sym (leftIdentity r)) (h ε).
```

In the following example, we show how to use the combinators for equational reasoning (see remark 3.13) in the formalisation of a textbook-style proof.

Example 3.15. We prove the left-cancellation property following Mac Lane and Birkhoff's [1999] proof.

Theorem. Let $(\mathrm{G}, \cdot, ^{-1}, \varepsilon)$ be a group. For all $a, b, c \in \mathrm{G}$, $a \cdot b = a \cdot c$ implies $b = c$.

Proof.

1.	$a^{-1} \cdot (a \cdot b) = a^{-1} \cdot (a \cdot c)$	(by hypothesis)
2.	$a^{-1} \cdot a \cdot b = a^{-1} \cdot a \cdot c$	(associativity)
3.	$\varepsilon \cdot b = \varepsilon \cdot c$	(left-inverse)
4.	$b = c$	(left-identity) $\square$

A few details have been omitted here too. For example, the hypothesis of the theorem does not completely justify why we could substitute the equal terms $a \cdot b$ and $a \cdot c$ in step 1. To justify this step in the formal proof, we use the fact that the propositional equality is a congruence relation—it is an equivalence relation compatible with the group structure—on groups [Birkhoff and Mac Lane 1977], which is implemented by the following properties:

```
·-leftCong : ∀ {a b c} → a ≡ b → a · c ≡ b · c
·-leftCong refl = refl
```

```
·-rightCong : ∀ {a b c} → b ≡ c → a · b ≡ a · c
·-rightCong refl = refl

⁻¹-cong : ∀ {a b} → a ≡ b → a ⁻¹ ≡ b ⁻¹
⁻¹-cong refl = refl.
```

In the formalisation of the proof, we reread the textbook proof as a chain reasoning from the left-hand side to the right-hand side of step 4: the left-hand side through steps 3, 2 and 1 and then the right-hand side through steps 1, 2 and 3.

```
1  leftCancellation : ∀ {a b c} → a · b ≡ a · c → b ≡ c
2  leftCancellation {a} {b} {c} h =
3    b             ≡⟨ sym (leftIdentity b) ⟩
4    ε · b         ≡⟨ ·-leftCong (sym (leftInverse a)) ⟩
5    a ⁻¹ · a · b  ≡⟨ assoc (a ⁻¹) a b ⟩
6    a ⁻¹ · (a · b) ≡⟨ ·-rightCong h ⟩
7    a ⁻¹ · (a · c) ≡⟨ sym (assoc (a ⁻¹) a c) ⟩
8    a ⁻¹ · a · c  ≡⟨ ·-leftCong (leftInverse a) ⟩
9    ε · c         ≡⟨ leftIdentity c ⟩
10   c             ∎
```

As can be seen in line 6, the justification of step 1 of the textbook proof uses the right-compatibility of the propositional equality with the multiplication operation `·-rightCong`, with a proof of `a · b ≡ a · c` given by the hypothesis of the theorem.

When formalising proofs in first-order theories it is useful to introduce defined operators as illustrated by the following example.

Example 3.16. We prove that the inverse of a commutator is a commutator.

Theorem. Let $(\mathrm{G}, \cdot, ^{-1}, \varepsilon)$ be a group. The commutator of $a, b \in \mathrm{G}$ is defined by $[a, b] = a^{-1} \cdot b^{-1} \cdot a \cdot b$. For all $a, b \in \mathrm{G}$, the inverse of the commutator $[a, b]$ is the commutator $[b, a]$ [Kurosh 1960].

In the formalisation of the proof of this theorem, we start for defining the group commutator by

```
[_,_] : G → G → G
[ a , b ] = a ⁻¹ · b ⁻¹ · a · b.
```

After proving the right-inverse property given by

$$\forall a.\ a \cdot a^{-1} = \varepsilon,$$

the interactive proof of theorem consists of 15 tedious proof steps of equational reasoning.

```
commutatorInverse : ∀ a b → [ a , b ] · [ b , a ] ≡ ε
commutatorInverse a b =
  a ⁻¹ · b ⁻¹ · a · b · (b ⁻¹ · a ⁻¹ · b · a)
    ≡⟨ assoc (a ⁻¹ · b ⁻¹ · a) b (b ⁻¹ · a ⁻¹ · b · a) ⟩
  a ⁻¹ · b ⁻¹ · a · (b · (b ⁻¹ · a ⁻¹ · b · a))
    ≡⟨ ·-rightCong (·-rightCong (assoc (b ⁻¹ · a ⁻¹) b a)) ⟩
  a ⁻¹ · b ⁻¹ · a · (b · (b ⁻¹ · a ⁻¹ · (b · a)))
    ≡⟨ ·-rightCong (·-rightCong (assoc (b ⁻¹) (a ⁻¹) (b · a))) ⟩
  a ⁻¹ · b ⁻¹ · a · (b · (b ⁻¹ · (a ⁻¹ · (b · a))))
    ≡⟨ ·-rightCong (sym (assoc b (b ⁻¹) (a ⁻¹ · (b · a)))) ⟩
  a ⁻¹ · b ⁻¹ · a · (b · b ⁻¹ · (a ⁻¹ · (b · a)))
    ≡⟨ ·-rightCong (·-leftCong (rightInverse b)) ⟩
  a ⁻¹ · b ⁻¹ · a · (ε · (a ⁻¹ · (b · a)))
    ≡⟨ ·-rightCong (leftIdentity (a ⁻¹ · (b · a))) ⟩
  a ⁻¹ · b ⁻¹ · a · (a ⁻¹ · (b · a))
    ≡⟨ assoc (a ⁻¹ · b ⁻¹) a (a ⁻¹ · (b · a)) ⟩
  a ⁻¹ · b ⁻¹ · (a · (a ⁻¹ · (b · a)))
    ≡⟨ ·-rightCong (sym (assoc a (a ⁻¹) (b · a))) ⟩
  a ⁻¹ · b ⁻¹ · (a · a ⁻¹ · (b · a))
    ≡⟨ ·-rightCong (·-leftCong (rightInverse a)) ⟩
  a ⁻¹ · b ⁻¹ · (ε · (b · a))
    ≡⟨ ·-rightCong (leftIdentity (b · a)) ⟩
  a ⁻¹ · b ⁻¹ · (b · a)
    ≡⟨ assoc (a ⁻¹) (b ⁻¹) (b · a) ⟩
  a ⁻¹ · (b ⁻¹ · (b · a))
     ≡⟨ ·-rightCong (sym (assoc (b ⁻¹) b a)) ⟩
  a ⁻¹ · ((b ⁻¹ · b) · a)
     ≡⟨ ·-rightCong (·-leftCong (leftInverse b)) ⟩
  a ⁻¹ · (ε · a)
     ≡⟨ ·-rightCong (leftIdentity a) ⟩
  a ⁻¹ · a
     ≡⟨ leftInverse a ⟩
  ε ∎
```

When reasoning with our programming logics, the formalised equational reasoning is often done in the manner described in the examples above. On the other hand, in the context of the first-order theories, this kind of reasoning could be automated by the ATPs (in particular, all the theorems in the above examples are automatically proved by the ATPs from the appropriate axioms). As explained in the introduction of this thesis, one of the achievements of our work is the combination of interactive and automatic reasoning in formalised first-order theories for reasoning about functional programs. In Chapter 6, we shall describe how to get the most out of the integration of our Agda representation of first-order theories with the ATPs,

and how this integration was implemented.

3.3.2 Inductive Representation of Peano Arithmetic: Using Postulates for Representing the Non-Logical Axioms

We formalise first-order Peano arithmetic (henceforth, PA) using postulates for representing its non-logic axioms.

Keeping in mind that there are different (equivalent) sets of axioms for PA, let $\mathfrak{L} = \{\mathsf{succ}, +, *, 0\}$ be the formal language of PA, where succ is a unary function symbol (successor function), $+$ and $*$ are two binary function symbols (addition and multiplication operations, respectively), and 0 is a constant symbol (zero element). The axioms of PA are (see, for example, Machover [1996] and Hájek and Pudlák [1998]):

$$\begin{aligned}
\forall n.\ 0 \neq \mathsf{succ}(n) &\quad (\mathrm{PA}_1)\\
\forall m\ n.\ \mathsf{succ}(m) = \mathsf{succ}(n) \supset m = n &\quad (\mathrm{PA}_2)\\
\forall n.\ 0 + n = n &\quad (\mathrm{PA}_3)\\
\forall m\ n.\ \mathsf{succ}(m) + n = \mathsf{succ}(m + n) &\quad (\mathrm{PA}_4)\\
\forall n.\ 0 * n = 0 &\quad (\mathrm{PA}_5)\\
\forall m\ n.\ \mathsf{succ}(m) * n = n + (m * n) &\quad (\mathrm{PA}_6)\\
A(0) \supset (\forall n. A(n) \supset A(\mathsf{succ}(n))) \supset \forall n. A(n), & \\
\text{for all formulae } A &\quad \text{(axiom schema of induction)}
\end{aligned}$$

To formalise PA, we use FOL as in Fig. 3.1 where the domain of quantification is called $\mathbb{N}$. Writing `zero` for the constant 0, our implementation of axioms PA_1 to PA_6 is given by

```
infixl 10 _*_
infixl 9  _+_

postulate
  zero    : ℕ
  succ    : ℕ → ℕ
  _+_ _*_ : ℕ → ℕ → ℕ


postulate
  PA₁ : ∀ {n} → zero ≢ succ n
  PA₂ : ∀ {m n} → succ m ≡ succ n → m ≡ n
  PA₃ : ∀ n → zero + n ≡ n
  PA₄ : ∀ m n → succ m + n ≡ succ (m + n)
  PA₅ : ∀ n → zero * n ≡ zero
  PA₆ : ∀ m n → succ m * n ≡ n + m * n.
```
(3.9)

Agda's higher-order logic features allow us to implement the axiom schema of induction by an explicit quantification over the type of propositional functions.

```
postulate
  ℕ-ind : (A : ℕ → Set) →
          A zero →
          (∀ n → A n → A (succ n)) →
          ∀ n → A n.
```
(3.10)

In the following example, we use the proof of the commutativity of addition to show the use of the inductive approach—using postulates for representing PA axioms—when proving properties of PA.

Example 3.17.

Theorem. For all $m, n \in \mathbb{N}$, $m + n = n + m$.

Proof. The proof is by induction on m, that is, we use the axiom of induction on the propositional function

$$A(m) \stackrel{\text{def}}{=} \forall n.\ m + n = n + m.$$

The proof of the base case $A(0)$ is given by

$$\begin{aligned} 0 + n &= n && (\mathrm{PA}_3) \\ &= n + 0 && \text{(right-identity of addition, that is, } n + 0 = n) \end{aligned}$$

The proof of the inductive step, $\forall m.A(m) \supset A(\mathsf{succ}(m))$, is given by

$$\begin{aligned} \mathsf{succ}(m) + n &= \mathsf{succ}(m + n) && (\mathrm{PA}_4) \\ &= \mathsf{succ}(n + m) && \text{(induction hypothesis)} \\ &= n + \mathsf{succ}(m) && \text{(arithmetical property)} \end{aligned}$$

□

Let us implement $A(m)$ by the propositional function

```
A : ℕ → Set
A m = ∀ n → m + n ≡ n + m.
```

In the formalisation of this proof, we need to consider some details which were omitted.

Given a proof of the right-identity of addition

```
+-rightIdentity : ∀ n → n + zero ≡ n
```

and the definition

```
+-leftIdentity : ∀ n → zero + n ≡ n
+-leftIdentity = PA₃
```

the two steps used in the proof of the base case are formalised using the combinators for equational reasoning.

```
A0 : A zero
A0 n = zero + n ≡⟨ +-leftIdentity n ⟩
       n        ≡⟨ sym (+-rightIdentity n) ⟩
       n + zero ∎
```

Now, given a proof of the compatibility of the propositional equality with the successor function

```
succCong : ∀ {m n} → m ≡ n → succ m ≡ succ n
```

and a proof of the arithmetical property

```
x+Sy≡S[x+y] : ∀ m n → m + succ n ≡ succ (m + n)
```

the formalisation of the step case is given by

```
1 is : ∀ m → A m → A (succ m)
2 is m ih n = succ m + n   ≡⟨ PA₄ m n ⟩
3             succ (m + n) ≡⟨ succCong (ih n) ⟩
4             succ (n + m) ≡⟨ sym (x+Sy≡S[x+y] n m) ⟩
5             n + succ m ∎
```

Note our use of the compatibility of the propositional equality with the successor function in line 3 to justify the omitted detail required in the textbook proof when using the inductive hypothesis `ih`.

Finally, we integrate the definition of the propositional function `A`, the proof of the base case `A0` and the proof of the inductive step `is` in the proof of `+-comm` using the axiom of induction `ℕ-ind` and Agda's **`where`** clauses.

```
+-comm : ∀ m n → m + n ≡ n + m
+-comm m n = ℕ-ind A A0 is m
  where
  A : M → Set
  A i = i + n ≡ n + i

  A0 : A zero
  A0 = zero + n ≡⟨ +-leftIdentity n ⟩
       n        ≡⟨ sym (+-rightIdentity n) ⟩
       n + zero ∎

  is : ∀ i → A i → A (succ i)
  is i ih = succ i + n   ≡⟨ PA₄ i n ⟩
            succ (i + n) ≡⟨ succCong ih ⟩
            succ (n + i) ≡⟨ sym (x+Sy≡S[x+y] n i) ⟩
            n + succ i   ∎
```

3.3.3 Inductive Representation of Peano Arithmetic: Using Inductive Notions for Representing the Non-Logical Axioms

We formalise PA with the inductive approach using inductive notions for representing its non-logical axioms. We use an inductive data type for representing the PA domain, and function defined by structural recursion for representing the arithmetical operations and the axiom schema of induction. By comparing the two representations of PA, we hope the benefits of using inductive notions in the formalisation of first-order theories will become even more clear.

We start by defining the domain of PA with the inductive data type

```
data ℕ : Set where
  zero : ℕ
  succ : ℕ → ℕ.
```

As explained in § 2.1, the above declaration introduces three constants to the theory

```
ℕ    : Set
zero : ℕ
succ : ℕ → ℕ
```

that is, a new small type called ℕ, with a nullary constructor `zero` and a unary, recursive constructor `succ`.

Before we move to the inductive implementation of PA, we point out some aspects related to the inductive representation of FOL. Some of the constants in the representation of FOL in Fig. 3.1 were defined on the postulated domain of quantification D. Given that we are now working with the inductive domain ℕ, it is necessary to define those constants over ℕ instead of over D. The FOL constants on both domains are formally the same, being concrete instances of more general versions parametrised by small types. We shall therefore avoid cumbersome notations using the names in Fig. 3.1 for both sets of constants. For example, the propositional equality on the domain ℕ is defined by

```
data _≡_ (x : ℕ) : ℕ → Set where
  refl : x ≡ x.
```

Now, we continue with the inductive implementation of PA. We define two structural recursive operations + and ∗ over the elements in our domain ℕ.

```
infixl 10 _*_
infixl 9  _+_

_+_ : ℕ → ℕ → ℕ
zero   + n = n
succ m + n = succ (m + n)

_*_ : ℕ → ℕ → ℕ
zero   * n = zero
succ m * n = n + m * n.
```

By providing theses definitions, Agda will be able to use normalisation when performing equational reasoning for `_+_` and `_*_`, which results in shorter proofs.

In the following example, we compare the formalisation of the proof of the commutativity of addition using and not using Agda's proof engine.

Example 3.18. Using Agda's proof engine, the implementation of the proof of the commutativity of addition can be conducted by pattern matching on the first argument because it belongs to an inductive domain.

```
1  +-comm : ∀ m n → m + n ≡ n + m
2  +-comm zero     n = sym (+-rightIdentity n)
3  +-comm (succ m) n = succ (m + n) ≡⟨ succCong (+-comm m n) ⟩
4                      succ (n + m) ≡⟨ sym (x+Sy≡S[x+y] n m) ⟩
5                      n + succ m   ∎
```

The first difference with respect to the proof in Example 3.17 is that we did not use the constant `ℕ-ind` which implements the axiom schema of induction. Instead, we use pattern matching and structural recursion for writing the proof. Note that this is more of a notational that fundamental difference: the proof by pattern matching can be translated into a proof by `ℕ-ind`. Line 2 corresponds to the proof of the base case. Lines 3-5 correspond to the proof of the inductive step. Both proofs have one fewer equational reasoning steps than the proofs in Example 3.17, due to the normalisation of the equations of the addition function `_+_` performed by Agda's type checker. In line 3, we replaced the use of the inductive hypothesis by a recursive call to the function `+-comm` on structurally smaller arguments.

It is noteworthy that the inductive representation of PA using inductive notions for representing its non-logical axioms entails the definitions used in the inductive representation of PA using postulates for representing its non-logical axioms presented in § 3.3.2. Using the inductive domain $\mathbb{N}$, the propositional equality on this domain, and the structural recursive functions `_+_` and `_*_`, it is possible to prove all the PA axioms introduced in § 3.3.2.

The axiom PA_1 is proved using the absurd pattern.

```
PA₁ : ∀ {n} → zero ≠ succ n
PA₁ ().
```

The proof of axiom PA_2 is reduced to a proof of the trivial identity `n ≡ n` after pattern match on a proof of the hypothesis.

```
PA₂ : ∀ {m n} → succ m ≡ succ n → m ≡ n
PA₂ refl = refl.
```

The axioms PA_3 and PA_4 for addition and the axioms PA_5 and PA_6 for multiplication are consequences of Agda's normalisation of the definitions of `_+_` and `_*_`, respectively.

```
PA₃ : ∀ n → zero + n ≡ n
PA₃ n = refl

PA₄ : ∀ m n → succ m + n ≡ succ (m + n)
PA₄ m n = refl

PA₅ : ∀ n → zero * n ≡ zero
PA₅ n = refl

PA₆ : ∀ m n → succ m * n ≡ n + m * n
PA₆ m n = refl.
```

Finally, rather than postulating the axiom of induction, we implement it using the inductive domain of PA and a higher-order, structural recursive function defined by pattern matching on this domain.

```
ℕ-ind : (A : ℕ → Set) →
        A zero →
        (∀ n → A n → A (succ n)) →
        ∀ n → A n
ℕ-ind A A0 h zero     = A0
ℕ-ind A A0 h (succ n) = h n (ℕ-ind A A0 h n).
```

Note that the axiom of induction corresponds to the induction principle associated with the inductive domain of $\mathbb{N}$.

Remark 3.19. The induction principle associated with a (recursive) data type corresponds to the elimination rule of such data type. In the sequel, we shall used either concepts when referring to induction principles.

3.4 On the Adequacy of the Inductive Representation of First-Order Logic and Theories

When using Agda as a logical framework, the proof of a formula in a logic reduces to a type inhabitation problem in the Agda representation of that logic. In this context, a question arises as to whether the logic as originally presented is indeed the logic that has been represented in the logical framework. An *adequacy theorem* stating that the consequence relation in the logic has been well represented in the logical framework is needed (see, for example, Gardner [1993, 1995] and Harper and Licata [2007]). Since Agda is a research system with some features whose meta-theory have not been formalised (see, for example, Danielsson [2010], Forsberg and Setzer [2010] and Abel, Pientka et al. [2013]), rigorous proofs that our inductive representations of first-order logic and theories are adequate, besides being very difficult, are beyond the scope of this thesis. In the next sections, we shall discuss the possibilities and/or difficulties of showing an adequacy theorem for our inductive representation of first-order logic and theories. In Sections § 3.4.1 and § 3.4.2, the discussion is based on the inductive representation of FOL and the use of pattern matching, respectively. In Section § 3.4.3, the discussion is based on the inductive representation of first-order theories.

3.4.1 Adequacy of the Inductive Representation of First-Order Logic

In Fig. 3.1, we show the representation of FOL using Agda's inductive notions. Following Gardner [1993, 1995], we could show an adequacy theorem for FOL which states that: a formula `A` can be proved in FOL if and only if there is a proof term `a : A*` in the Agda representation of FOL, where `A*` is the Agda representation of `A` using (3.3). The proof term `a` can only use the constants introduced in Fig. 3.1 and the constants underlying Agda as a logical framework, that is, the universe of small types **`Set`**, the non-dependent function type `A → B`, the dependent function type `(x : A) → B`, function application `f a`, λ-abstraction `λ x → e`, the judgments `A : Set`, `a : A` and `l = r : A`, and the inductively defined types and families.

Convention 3.20. We shall call 'basic inductive constants of FOL' to the set of constants above described.

Remark 3.21. In the use of Agda as a logical framework, there are additional constants such as the infinite hierarchy of universes $\mathbf{Set}_{i\in\omega}$, which are not used in our inductive representation of first-order logic and theories.

Example 3.22. According to Fig. 3.1, the only constants related to disjunction that may be used are `_∨_`, `inj₁`, `inj₂` and `case`; similarly for the

other logical constants.

Remark 3.23. Note from Fig. 3.1 that the conditional and the universal quantifier on `D` are not represented by inductive constants. We could inductively represent both constants and their corresponding applications as follows:

```
data _⊃_ (A B : Set) : Set where
  fun : (A → B) → A ⊃ B

app : {A B : Set} → A ⊃ B → A → B
app (fun f) a = f a

data ForAll (A : D → Set) : Set where
  dfun : ((t : D) → A t) → ForAll A

dapp : {A : D → Set}(t : D) → ForAll A → A t
dapp t (dfun f) = f t.
```

This is however not necessary because we are using a shallow embedding for representing FOL.

Remark 3.24. Since the type formers (the formation rules) and the constants (introduction and elimination rules) introduced in the inductive and LF- approaches have the same types, we basically reduce the adequacy problem for the inductive representation of FOL, which only uses the basic inductive constants of FOL, to the adequacy of its LF-representation.

3.4.2 Adequacy of the Use of Pattern Matching with the Inductive Representation of First-Order Logic

Since in Fig. 3.1, we allow the introduction of constants defined by pattern matching on types introduced with Agda's **`data`** constructor an adequacy theorem should be based on the fact that the new constants are non-recursive—they represent FOL-proofs—and on the fact that the pattern matching used can be eliminated and replaced by terms only using the basic inductive constants of FOL (see convention 3.20).

In what follows, we show a few examples where we show how to eliminate pattern matching in non-recursive definitions.

Example 3.25. The **`data`** construct for disjunction gives access to pattern matching on `_∨_`. If the user introduces a new non-recursive constant with a definition of the form

```
g : (A → C) → (B → C) → A ∨ B → C
g f₁ f₂ (inj₁ a) = f₁ a
g f₁ f₂ (inj₂ b) = f₂ b
```

for some `A B C : Set`, `f₁ : A → C` and `f₂ : B → C`, then we can replace the above pattern matching on a proof of `A ∨ B` with

```
g f₁ f₂ = case f₁ f₂.
```

In the simplest situation, the constants `A`, `B` and `C` are closed formulae (small types) and `f₁` and `f₂` are closed proof terms, but Agda allows us to introduce schematic proofs as well (see Example 3.27).

Example 3.26. Using the inductive definition of the FOL-equality, we can also replace a non-recursive proof by pattern matching like

```
g : ∀ {a b} → a ≡ b → C a b
g refl = d
```

where `C : D → D → Set` and `d : ∀ {a} → C a a`, by

```
g {a} h = subst (λ x → C a x) h d.
```

Example 3.27. We can also remove the pattern matching used in schematic proofs. In Example 3.12, we introduced the following non-recursive schematic proofs

```
∨-comm : {A B : Set} → A ∨ B → B ∨ A
∃∀     : {A : D → D → Set} → ∃[ x ](∀ y → A x y) → ∀ y →
         ∃[ x ] A x y
```

by pattern matching on a proof of `A ∨ B` and on a proof of `∃[ x ](∀ y → A x y)`, respectively. These patterns matching can be eliminated and replaced by terms which only use the basic inductive constants of FOL (see convention 3.20).

Agda allows for more advanced forms of pattern matching than discussed until now. Although there is no formal description of Agda's pattern matching facility, an adequacy theorem should be based on showing that those advanced forms of pattern matching used for introducing new non-recursive constants can also be replaced by the basic inductive constants of FOL.

In the following example, we show how to eliminate a non-trivial pattern matching in a non-recursive definitions.

Example 3.28. Let `A B C E : Set` be small types, and

```
f₁ : A → E
f₂ : B → E
f₃ : C → E
```

be constants. If we want to define the proof

```
g : (A ∨ B) ∨ C → E
```

by pattern matching, we can directly build nested patterns

```
g (inj₁ (inj₁ a)) = f₁ a
g (inj₁ (inj₂ b)) = f₂ b
g (inj₂ c)        = f₃ c.
```

The previous patterns matching can be replaced by repeatedly using the `case` elimination rule

```
g = case (case f₁ f₂) f₃.
```

Remark 3.29. Norell's thesis states very clearly that Agda's pattern matching is a *non-conservative* extension of Martin-Löf's type theory: by using pattern matching on a proof of the identity type, we can prove the Streicher-Altenkirch `K` axiom which has been shown to not be derivable from the elimination rule `subst` for the propositional equality (see Norell [2007b] and references therein). In our inductive approach, the FOL's theorems are translated into types using (3.3), since these types do not contain proof terms of the identity type, our possible adequacy theorem is not affected by Agda's possibility of proving the Streicher-Altenkirch `K` axiom.

3.4.3 Adequacy of the Inductive Representation of First-Order Theories

When we use inductively defined types and/or families in the representation of a first-order theory T, the set of the basic inductive constants of T is the set of the basic inductive constants of FOL plus the new inductively defined constants, that is, the new formation, introduction and elimination rules added. In this case, an adequacy theorem should be based on showing that the pattern matching used for defining new constants can be replaced by terms only using the basic inductive constants of T. Whether a pattern matching can or cannot be eliminated depend on Agda's type checker.

In the following example, we show how to use the induction principle associated with an inductive data type to replace recursive definitions made by pattern matching.

Example 3.30. In our inductive representation of PA (see § 3.3.3), we introduced the following constants:

```
ℕ     : Set
zero  : ℕ
succ  : ℕ → ℕ
ℕ-ind : (A : ℕ → Set) →
        A zero →
        (∀ n → A n → A (succ n)) →
        ∀ n → A n.
```

As highlighted in Example 3.18, we can replace a proof by pattern matching on ℕ by a proof using the axiom schema of induction implemented by the constant ℕ-ind. The key point in an adequacy theorem for our inductive representation of PA is that recursive functions—like _*_ or _*_—defined by pattern matching can be directly reduced to definitions by ℕ-ind.

Let h be a primitive recursive function (see, for example, Kleene [1952]) defined by pattern matching on ℕ

```
h : {A : Set} → ℕ → A
h zero     = d
h (succ n) = e n (h n)
```

where d : A and e : ℕ → A → A. The h function can be defined without using pattern matching by using the combinator

```
ℕ-rec : {A : Set} → A → (ℕ → A → A) → ℕ → A
ℕ-rec {A} = ℕ-ind (λ _ → A).
```

For example, the definitions of the functions _+_ and _*_ are given by

```
_+_ : ℕ → ℕ → ℕ
m + n = ℕ-rec n (λ _ x → succ x) m

_*_ : ℕ → ℕ → ℕ
m * n = ℕ-rec zero (λ _ x → n + x) m.
```

Using the ℕ-rec combinator we can define all the *first-order* primitive functions—using Harper's [2013] terminology—and using ℕ-ind we can also define *higher-order* primitive functions like the Ackermann function (see § 2.1). In fact, using ℕ-ind we can define all the functions from ℕ to ℕ provably total in PA (see, for example, Girard [1990]).

Remark 3.31. In the formalisation of the theories described in Chapters 4 and 5, we use inductively defined families and Agda's full support for proofs by pattern matching. In addition, these formalisations are based on our inductive formalisation of FOL described in Fig. 3.1. Consequently, the discussions in § 3.4 also apply to those theories.

Chapter 4

Logical Theory of Constructions

Having now explained how we shall use Agda (with postulates, data declarations and pattern matching) for implementing proofs in (classical) first-order logic and theories, we shall move to the main topic of this thesis, namely how to prove properties about functional programs in first-order logic. Agda will provide the environment for interactive proofs. In Chapter 6, we shall show how we interact with automatic first-order theorem provers.

Before moving to first-order logic, we shall discuss the formalisation of a programming logic for reasoning about a core lazy functional programming language, Plotkin's PCF language. In this programming logic we can reason about λ-terms and since they are not permitted in first-order logic, we shall need to replace them by combinators. We shall discuss this replacement in Chapter 5.

We use the name 'logical theory of constructions' (henceforth, LTC) as a generic name for a family of related logical systems which have been used by Aczel [1977b, 1980, 1989] and Smith [1978, 1984] to interpret Martin-Löf's type theory in type-free logical systems. Given the existence of an interpretation, we can conclude that LTC is at least as strong as Martin-Löf's type theory. Some of these logical systems are based on combinators and stay strictly within the realm of FOL, whereas others are based on the λ-calculus and do not. Since these systems play a role for constructive foundations, they are usually intuitionistic but we can also consider classical versions.

Dybjer [1985, 1990] showed that LTC is appropriate for verification of *lazy* and *general* recursive functional programs. Since general recursive functions cannot be directly formalised in Martin-Löf's type theory, LTC is strictly more general than it. In particular, we can write arbitrary general recursive functions defined by fixed-points. Dybjer's programming logic is closely related to Aczel's first-order theory of combinatory arithmetic [Aczel 1977b]

but is based on the λ-calculus, and it is hence not a first-order theory.

Dybjer's [1985] LTC logic will not suffice for proving all interesting properties of functional programs: to reason for example about streams (potentially infinite lists) we need principles, such as co-induction [Gordon 1995], which are not available in LTC. However, as Dybjer [1985] emphasised, LTC will go a long way if we restrict ourselves to behaviours of programs on *total* and *finite* elements of data structures.

In § 4.1, we adapt Dybjer's programming logic to a version of the lazy PCF language. In § 4.2, we prove the consistency of our programming logic by building a (logical) model which uses denotational semantics and the theory of inductive definitions. In § 4.3, we represent our programming logic in Agda using the inductive approach. In § 4.4 and § 4.5, we show how to use our programming logic to prove properties by structural recursion and to verify general recursive programs, respectively.

4.1 Logical Theory for PCF

Although our long term goal is to verify "real" lazy functional programs (written in Haskell, for example), we define our programming logic for a simple setting. In this chapter, we adapt Dybjer's LTC logic to the PCF language, where we only have Booleans and natural numbers as basic data. We depart from Plotkin's presentation by considering PCF as a type-free language. We also make the fixed-point operator a binding construct following [Winskel 1994].

Our LTC-style programming logic for PCF is a logical system with equality. The programming logic described in Chapter 5 will be a first-order theory whose underlying logic will be *classical* logic. Since the consistency of this programming logic will be based on a translation of it into the logic theory for PCF (henceforth, $\mathsf{LT}_{\mathsf{PCF}}$), it is required that the underlying logic of $\mathsf{LT}_{\mathsf{PCF}}$ be classical logic too. One might object that we should not work with non-intuitionist axioms such as the principle of the exclude middle. We reject this objection on the grounds that in this thesis we are not attempting to contribute to constructive foundations but to outline a genuinely practical approach to verification of lazy functional programs. To this end, we do not mind using classical logic (for reasons given above) and axioms which are proved consistent using classical techniques.

$\mathsf{LT}_{\mathsf{PCF}}$-terms are generated by the following grammar:

$$\begin{aligned}
\mathsf{Terms} \ni t ::= {} & x && \text{variable}\\
| {} & t \cdot t && \text{application}\\
| {} & \lambda x.t && \lambda\text{-abstraction}\\
| {} & \mathsf{fix}\ x.t && \text{fixed-point operator}\\
| {} & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if} && \text{partial Boolean constants}\\
| {} & 0 \mid \mathsf{succ} \mid \mathsf{pred} \mid \mathsf{iszero} && \text{partial natural number constants}
\end{aligned}$$

$\mathsf{LT}_{\mathsf{PCF}}$ has the standard FOL-predicate symbols $P(t, \dots, t)$. Moreover, $\mathsf{LT}_{\mathsf{PCF}}$ has two unary inductive predicate symbols $\mathcal{B}ool$ and $\mathcal{N}$, where $\mathcal{B}ool(t)$ means that t is a *total* and *finite* Boolean value (true or false), and $\mathcal{N}(t)$ means that t is a *total* and *finite* natural number.

The $\mathsf{LT}_{\mathsf{PCF}}$-formulae are generated by the following grammar:

$$\begin{aligned}
\mathsf{Formulae} \ni A ::= {} & \top \mid \bot && \text{truth, falsehood}\\
| {} & A \supset A \mid A \wedge A \mid A \vee A && \text{binary logical connectives}\\
| {} & \forall x.A \mid \exists x.A && \text{quantifiers}\\
| {} & t = t && \text{equality}\\
| {} & P(t, \dots, t) && \text{predicate}\\
| {} & \mathcal{B}ool(t) && \text{total Booleans predicate}\\
| {} & \mathcal{N}(t) && \text{total natural numbers predicate}
\end{aligned}$$

Conventions 4.1. The binary application function symbol "$\cdot$" is left-associative. In addition, we follow the same abbreviations and conventions as with the FOL-formulae (see conventions 3.2).

Terms t and formulae A are formed in the usual way. The non-logical axioms and axiom schemata of $\mathsf{LT}_{\mathsf{PCF}}$ can be classified into three groups: (i) conversion rules for the combinators, (ii) discrimination rules expressing that terms beginning with different constructors are not convertible and (iii) introduction and elimination rules for $\mathcal{B}ool$ and $\mathcal{N}$.

- Conversion rules for the $\mathsf{LT}_{\mathsf{PCF}}$-terms:

$$\begin{aligned}
\forall t\ t'.\ \mathsf{if} \cdot \mathsf{true} \cdot t \cdot t' &= t,\\
\forall t\ t'.\ \mathsf{if} \cdot \mathsf{false} \cdot t \cdot t' &= t',\\
\mathsf{pred} \cdot 0 &= 0,\\
\forall t.\ \mathsf{pred} \cdot (\mathsf{succ} \cdot t) &= t,\\
\mathsf{iszero} \cdot 0 &= \mathsf{true},\\
\forall t.\ \mathsf{iszero} \cdot (\mathsf{succ} \cdot t) &= \mathsf{false},
\end{aligned} \tag{4.1a}$$

$$\begin{aligned} \forall t\ t'.\ (\lambda x.t) \cdot t' &= t[x := t'], \\ \forall t.\ \mathsf{fix}\ x.t &= t[x := \mathsf{fix}\ x.t], \end{aligned} \tag{4.1b}$$

where $t[x := t']$ is the capture-free substitution of x for t' in t.

- Discrimination rules for constructors:

$$\begin{aligned} \mathsf{true} &\neq \mathsf{false}, \\ \forall t.\ 0 &\neq \mathsf{succ} \cdot t. \end{aligned} \tag{4.2}$$

- Introduction and elimination (expressing proof by case analysis on total and finite Boolean values) rules for the inductive predicate $\mathcal{B}ool$:

$$\frac{}{\mathcal{B}ool(\mathsf{true})}, \qquad \frac{}{\mathcal{B}ool(\mathsf{false})}, \tag{4.3a}$$

$$\frac{\mathcal{B}ool(t) \qquad A(\mathsf{true}) \qquad A(\mathsf{false})}{A(t)}. \tag{4.3b}$$

- Introduction and elimination (expressing proof by mathematical induction) rules for the inductive predicate $\mathcal{N}$ (see, for example, Martin-Löf [1971]):

$$\frac{}{\mathcal{N}(0)}, \qquad \frac{\mathcal{N}(t)}{\mathcal{N}(\mathsf{succ} \cdot t)}, \tag{4.4a}$$

$$\frac{\mathcal{N}(t) \qquad A(0) \qquad \begin{array}{c}[A(t')] \\ \vdots \\ A(\mathsf{succ} \cdot t')\end{array}}{A(t)}. \tag{4.4b}$$

Remark 4.2. Note that (some of) the above conversion and discrimination rules only hold if the arguments of the functions are evaluated lazily. For example, the equation

$$\forall t.\ \mathsf{pred} \cdot (\mathsf{succ} \cdot t) = t$$

does not hold if it is evaluated strictly and the term t loops.

In $\mathsf{LT}_{\mathsf{PCF}}$, we use the inductive predicates $\mathcal{B}ool$ and $\mathcal{N}$ to assert that a certain (possibly non-terminating) program *terminates* with a total and finite Boolean value or with a total and finite natural number, respectively. This result is based on a translation of these inductive predicates to types in Martin-Löf's type theory (see Dybjer [1985] for a detailed discussion of it). For example, we express that a function f terminates and it maps a total and finite natural number to a total and finite natural number by the formula

$$\forall t.\ \mathcal{N}(t) \supset \mathcal{N}(f \cdot t). \tag{4.5}$$

4.2 Consistency

The consistency of our programming logic $\mathsf{LT}_{\mathsf{PCF}}$ is based on standard results from domain theory and the theory of inductive definitions. The consistency of $\mathsf{LT}_{\mathsf{PCF}}$ is a consequence of the existence of a domain model for its term language—a type-free version of PCF—and the interpretation of its inductively defined predicates as subsets of this domain model.

The definitions related to domain theory vary considerably in the literature. Appendix A contains a summary of notions used on this thesis. In particular, we shall use the term 'domain' to refer to an ω-complete partial order (Definition A.4).

We know from domain theory that a domain model for $\mathsf{LT}_{\mathsf{PCF}}$-terms, the term language of $\mathsf{LT}_{\mathsf{PCF}}$—where self-application is allowed and where the terms will have values in the Booleans or the lazy natural numbers—is a solution to the recursive domain equation (see, for example, Plotkin [1992])

$$\mathbf{D} \cong \mathbf{B}_{\perp} \oplus \mathbf{LN} \oplus (\mathbf{D} \to \mathbf{D})_{\perp}, \tag{4.6}$$

where $\oplus$ is the coalesced sum on domains (Definition A.11), $\mathbf{B}_{\perp}$ is the lifted Boolean domain (Example A.6), $\mathbf{LN}$ is the lazy natural numbers domain (Example A.7) and $(\mathbf{D} \to \mathbf{D})_{\perp}$ is the lifted function space domain (Definition A.5), which is required because the operator $\oplus$ is defined on domains.

Notation 4.3. Let $\mathbf{D}$ be a domain and let ρ be a valuation on $\mathbf{D}$, that is, a function from the set of variables in $\mathsf{LT}_{\mathsf{PCF}}$-terms to $\mathbf{D}$. The notation $\rho(x \mapsto \mathbf{d})$ indicates the valuation which maps x to $\mathbf{d}$ and otherwise acts like ρ. The notation $\boldsymbol{\lambda}\mathbf{x}.\mathbf{e}$ denotes λ-abstraction on $\mathbf{D}$.

Let $\mathbf{D}$ be a solution to (4.6) and $[\mathbf{D} \to \mathbf{D}]$ the function space of continuous functions on $\mathbf{D}$ (Definition A.9). The domain $\mathbf{D}$ comes equipped with the continuous functions [Barendregt 2004]

$$\begin{aligned} \mathrm{F} &: \mathbf{D} \to [\mathbf{D} \to \mathbf{D}], \\ \mathrm{G} &: [\mathbf{D} \to \mathbf{D}] \to \mathbf{D}. \end{aligned}$$

Let ρ be a valuation on $\mathbf{D}$. Based on Pitts [1994a], we define the interpretation $[\![\ ]\!]_{\rho} : \mathsf{LT}_{\mathsf{PCF}}\text{-terms} \to \mathbf{D}$ by induction on $\mathsf{LT}_{\mathsf{PCF}}$-terms by

$$[\![x]\!]_{\rho} = \rho(x), \qquad [\![t \cdot t']\!]_{\rho} = \begin{cases} \mathbf{f}([\![t']\!]_{\rho}) & \text{if } [\![t]\!]_{\rho} = \mathrm{G}(\mathbf{f}), \\ \perp & \text{otherwise,} \end{cases}$$

$$[\![\lambda x.t]\!]_{\rho} = \mathrm{G}(\boldsymbol{\lambda}\mathbf{d}.[\![t]\!]_{\rho(x \mapsto \mathbf{d})}), \quad [\![\mathsf{fix}\ x.t]\!]_{\rho} = \mathrm{Fix}(\boldsymbol{\lambda}\mathbf{d}.[\![t]\!]_{\rho(x \mapsto \mathbf{d})}),$$

$$[\![\mathsf{true}]\!]_{\rho} = \mathbf{true}, \quad [\![\mathsf{false}]\!]_{\rho} = \mathbf{false},$$

$$[\![\mathsf{if}]\!]_{\rho} = \mathrm{G}(\mathbf{if}), \quad [\![0]\!]_{\rho} = \mathbf{0},$$

$$[\![\mathsf{succ}]\!]_{\rho} = \mathrm{G}(\mathbf{succ}), \quad [\![\mathsf{pred}]\!]_{\rho} = \mathrm{G}(\mathbf{pred}),$$

$$[\![\mathsf{iszero}]\!]_{\rho} = \mathrm{G}(\mathbf{iszero}),$$

where we omit the use of the injection functions in_i (Definition A.11) in order to simplify the presentation, Fix is the fixed-point operator given by Theorem A.10, and the continuous functions **if**, **succ**, **pred** and **iszero** from **D** to **D** are defined by

$$\mathbf{if}(d) = \begin{cases} \boldsymbol{\lambda}\mathbf{xy.x} & \text{if } d = \mathbf{true}, \\ \boldsymbol{\lambda}\mathbf{xy.y} & \text{if } d = \mathbf{false}, \\ \bot & \text{otherwise}, \end{cases} \qquad \mathbf{succ}(d) = \begin{cases} \mathbf{n+1} & \text{if } d = \mathbf{n} \in \mathbf{LN}, \\ \underline{\mathbf{n+1}} & \text{if } d = \underline{\mathbf{n}} \in \mathbf{LN}, \\ \bot & \text{otherwise}, \end{cases}$$

$$\mathbf{pred}(d) = \begin{cases} \mathbf{0} & \text{if } d = \mathbf{0}, \\ d' & \text{if } d = \mathbf{succ}(d'), \\ \bot & \text{otherwise}, \end{cases} \qquad \mathbf{iszero}(d) = \begin{cases} \mathbf{true} & \text{if } d = \mathbf{0}, \\ \mathbf{false} & \text{if } d = \mathbf{succ}(d'), \\ \bot & \text{otherwise}. \end{cases}$$

If the $\mathsf{LT}_{\mathsf{PCF}}$-equality is interpreted as the equality in **D**, it is possible to verify that the conversion rules (4.1) and the discrimination rules (4.2) are satisfied in **D**.

Now, for the interpretation of the inductively defined predicates $\mathcal{B}ool$ and $\mathcal{N}$, let **D** be a domain model of $\mathsf{LT}_{\mathsf{PCF}}$. We shall use Aczel's set-theoretic notion of *rule set* (sets of rules) to interpret these predicates [Aczel 1977a].

A *rule* on a set U is a pair $\langle X, x\rangle$, usually written $\frac{X}{x}$, where X is a set called the set of premises, and x is the conclusion, such that $X \cup \{x\} \subseteq U$. A rule $\langle X, x\rangle$ is *finitary* if the set X is finite.

Convention 4.4. In this thesis, it is only required to use finitary rules—infinitary rules are used for example by Dybjer [1991] for interpreting arbitrary inductive types and families of Martin-Löf's type theory—in the sequel, we shall use the word 'rule' to refer to 'finitary rule'.

Example 4.5. From the introduction rules (4.3a) and (4.4a) of the predicates $\mathcal{B}ool$ and $\mathcal{N}$ respectively, we define the following rule sets on **D**:

$$\Phi_{\mathcal{B}ool} = \left\{\frac{\emptyset}{\mathbf{true}}\right\} \cup \left\{\frac{\emptyset}{\mathbf{false}}\right\}, \qquad \Phi_{\mathcal{N}} = \left\{\frac{\emptyset}{\mathbf{0}}\right\} \cup \left\{\frac{\{n\}}{\mathbf{succ}(n)} \mid n \in \mathbf{D}\right\}.$$

Let Φ be a rule set on U. A set A is Φ-*closed* if $\langle X, x\rangle \in \Phi$ and $X \subseteq A$ implies $x \in A$. The *inductively defined set* by Φ is the least Φ-closed set defined by

$$I(\Phi) = \bigcap\{A \subseteq U \mid A \text{ is } \Phi\text{-closed}\}. \tag{4.7}$$

Example 4.6. The inductive predicates $\mathcal{B}ool$ and $\mathcal{N}$ are interpreted by the following subsets of **D**:

$$\begin{aligned} \mathbf{Bool} &= I(\Phi_{\mathcal{B}ool}) \\ &= \{\mathbf{true}, \mathbf{false}\}, \end{aligned} \qquad \begin{aligned} \mathbf{N} &= I(\Phi_{\mathcal{N}}) \\ &= \{\mathbf{0}, \mathbf{succ}(\mathbf{0}), \mathbf{succ}(\mathbf{succ}(\mathbf{0})), \ldots\} \\ &= \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \ldots\}. \end{aligned}$$

Remark 4.7. The inductively defined sets by a rule set can also be defined as least fixed-points of monotone operators. Let $\mathrm{Pow}(A)$ be the power set of a set A and let Φ be a rule set on a set U. The rule set Φ induces an operator $\widehat{\Phi}$ given by

$$\begin{aligned}\widehat{\Phi} : \mathrm{Pow}(U) &\to \mathrm{Pow}(U)\\ A &\mapsto \{x \in U \mid \langle X, x\rangle \in \Phi \text{ for some } X \subseteq A\}.\end{aligned}$$

Let $\widehat{\Phi} : \mathrm{Pow}(U) \to \mathrm{Pow}(U)$ be the operator induced by a (finite) rule set Φ on U. The operator $\widehat{\Phi}$ is monotone on the partial order $(\mathrm{Pow}(U), \subseteq)$. The inductively defined set by Φ is the least fixed-point of $\widehat{\Phi}$ defined by (see, for example, Winskel [1994] and Pitts [1994b])

$$I(\Phi) = \bigcup_{n\in\omega} \widehat{\Phi}^n(\emptyset),$$

where for any $A \subseteq \mathrm{Pow}(U)$

$$\begin{aligned}\widehat{\Phi}^0(A) &= A,\\ \widehat{\Phi}^{n+1}(A) &= \widehat{\Phi}(\widehat{\Phi}^n(A)).\end{aligned}$$

4.3 Inductive Representation of the Logical Theory for PCF

We benefit from Agda's inductively defined types, including inductive families, and function definitions using pattern matching on such types, in our representation of $\mathsf{LT}_{\mathsf{PCF}}$. In the inductive approach, the $\mathsf{LT}_{\mathsf{PCF}}$ logical constants are represented as type formers defined by their introduction rules, and the predicates for total and finite Boolean values, and total and finite natural numbers are represented as inductively defined predicates over the domain of $\mathsf{LT}_{\mathsf{PCF}}$. We can then define the elimination rules for the logical constants and the totality predicates by pattern matching.

For the same reason we mentioned in the inductive representation of FOL, that is, to make full use of Agda's support for proof by pattern matching, we shall not restrict ourselves to using the elimination rules above. In the inductive formalisation of $\mathsf{LT}_{\mathsf{PCF}}$, we shall also allow proofs by pattern matching in general, as long as they are accepted by Agda's coverage and termination checker. This means that we actually work in an extension of $\mathsf{LT}_{\mathsf{PCF}}$ since, by working in this way, new, more general induction principles become available. This extension can be expected to be conservative—but we do not know this for certain—although it would be quite hard to prove that rigorously, and such proof is outside the scope of this thesis (see § 3.4).

We could also inductively define the domain of $\mathsf{LT}_{\mathsf{PCF}}$ in Agda, but if we were to do so, we would be unable to use Agda's identity type for equality

of domain elements; equality of domain elements would be represented by a binary non-trivial equivalence relation `_≐_` and as a consequence we would need to work with a *setoid* `(D, ≐)`.

Forgetting for the moment the higher-order terms `lam` and `fix`, we could inductively define the domain `D` by

```
data D : Set where
  _·_ : D → D → D
  true false if zero succ pred iszero : D
```

and the setoid equality by

```
data _≐_ : D → D → Set where
  ≐-refl   : ∀ {x} → x ≐ x
  ≐-sym    : ∀ {x y} → x ≐ y → y ≐ x
  ≐-trans  : ∀ {x y z} → x ≐ y → y ≐ z → x ≐ z
  ≐-cong   : ∀ {x x' y y'} → x ≐ y → x' ≐ y' → x · x' ≐ y · y'
  if-true  : ∀ t t' → if · true · t · t' ≐ t
  if-false : ∀ t t' → if · false · t · t' ≐ t'
  pred-0   : pred · zero ≐ zero
  pred-S   : ∀ n → pred · (succ · n) ≐ n
  iszero-0 : iszero · zero ≐ true
  iszero-S : ∀ n → iszero · (succ · n) ≐ false.
```

Next, given the substitutivity property

```
≐-subst : (A : D → Set) → ∀ {x y} → x ≐ y → A x → A y
```

we could prove

```
≐→≡ : ∀ {x y} → x ≐ y → x ≡ y
≐→≡ {x} h = ≐-subst (λ z → x ≡ z) h refl
```

however, since the setoid equality is not substitutive (see, for example, Altenkirch and McBride [2006]), we cannot prove `≐-subst`, and therefore `_≐_` is a non-trivial equivalence relation and not all properties preserve it.

Since we shall *never* prove properties by induction on `D`, it is preferable to postulate the existence of both `D` and the conversion rules for terms on `D` using Agda's identity type.

To sum up, some of the axioms and axiom schemata of $\mathsf{LT}_{\mathsf{PCF}}$ will still be postulated, where others will be consequences of inductive definitions. We now present how this is done.

Terms. We employ the usual method for representing the type-free λ-calculus. First we postulate a domain of terms:

```
postulate D : Set.
```

Next, we postulate the term constructors for $\mathsf{LT}_{\mathsf{PCF}}$ using *higher-order abstract syntax*—using the binders of the meta-language to represent the binding structure of the object language—to represent the variable binding operations λ and fix as Agda higher-order functions. Writing `zero` for 0 and `lam` for λ, the postulates are:

```
postulate
  _·_                  : D → D → D
  lam fix              : (D → D) → D
  true false if        : D
  zero succ pred iszero : D.
```
(4.8)

Classical predicate logic with equality. Although $\mathsf{LT}_{\mathsf{PCF}}$ is not a first-order theory because it has λ-abstraction and a fixed-point operator, it does not have the features associated with higher-order logics—predicates having other predicates or functions as arguments, or quantification over functions or predicates [Mendelson 1997]. Since the fixed-point operator can be removed—it is definable in the type-free λ-calculus—and we can perform λ-lifting to remove the λ-abstractions [Peyton Jones 1987], we might say that $\mathsf{LT}_{\mathsf{PCF}}$ is "morally" first-order but not "officially" first-order. Hence, we use the logical constants and axioms implemented in Fig. 3.1 for representing the classical predicate logic of $\mathsf{LT}_{\mathsf{PCF}}$.

Conversion rules. The conversion rules (4.1) are implemented by the following postulates:

```
postulate
  if-true  : ∀ t {t'} → if · true · t · t' ≡ t
  if-false : ∀ {t} t' → if · false · t · t' ≡ t'
  pred-0   : pred · zero ≡ zero
  pred-S   : ∀ n → pred · (succ · n) ≡ n
  iszero-0 : iszero · zero ≡ true
  iszero-S : ∀ n → iszero · (succ · n) ≡ false
  beta     : ∀ f a → lam f · a ≡ f a
  fix-eq   : ∀ f → fix f ≡ f (fix f).
```
(4.9)

Discrimination rules. The discrimination rules (4.2) are implemented as follows:

```
postulate
  t≠f : true ≠ false
  0≠S : ∀ {n} → zero ≠ succ · n.
```

Inference rules for the total and finite Booleans predicate. The inductive predicate *Bool* for total and finite Booleans, given by the rules (4.3), is represented as an inductive family:

```
data Bool : D → Set where
  btrue  : Bool true
  bfalse : Bool false
```

The above declaration defines the unary predicate symbol `Bool` and the introduction rules `btrue` and `bfalse`.

We can now define the elimination rule for $\mathcal{B}ool$ by pattern matching.

```
Bool-ind : (A : D → Set) → A true → A false → ∀ {b} → Bool b → A b
Bool-ind A At Af btrue  = At
Bool-ind A At Af bfalse = Af.
```

Inference rules for the total and finite natural numbers predicate. The inductive predicate $\mathcal{N}$ for total and finite natural numbers, given by rules (4.4), is also represented as an inductive family:

```
data N : D → Set where
  nzero : N zero                                          (4.10)
  nsucc : ∀ {n} → N n → N (succ · n).
```

The above declaration introduces the unary predicate symbol `N` and the introduction rules `nzero` and `nsucc`.

The elimination rule for $\mathcal{N}$ is defined by pattern matching.

```
N-ind : (A : D → Set) →
        A zero →
        (∀ {n} → A n → A (succ · n)) →                    (4.11)
        ∀ {n} → N n → A n
N-ind A A0 h nzero      = A0
N-ind A A0 h (nsucc Nn) = h (N-ind A A0 h Nn).
```

Remark 4.8. When proving a property, we shall not use the elimination rules `Bool-ind` or `N-ind` because we shall use Agda's pattern matching on the total and finite Booleans, and on the total and finite natural numbers.

Remark 4.9. Sometimes the associated induction principle `N-ind`—given by (4.11)—for the inductive family `N`—given by (4.10)—has a slightly different shape. For example, the induction principle generated by Coq [The Coq Development Team 2014] has the hypothesis [Bertot and Castéran 2004]

```
∀ {n} → N n → A n → A (succ · n)
```

instead of the hypothesis

```
∀ {n} → A n → A (succ · n).
```

This difference is not important because the two versions of the induction principle `N-ind` are equivalents (see Appendix B).

Convention 4.10. Instead of using the constants `if`, `succ`, `pred` and `iszero` of type `D`, we define more readable and writable function symbols of the appropriate types.

```
if_then_else_ : D → D → D → D
if b then t else t' = if · b · t · t'

succ₁ : D → D
succ₁ n = succ · n

pred₁ : D → D
pred₁ n = pred · n

iszero₁ : D → D
iszero₁ n = iszero · n.
```

Remark 4.11. By using the above convention, instead of implementing the inductive predicate $\mathcal{N}$ by the inductive family (4.10) and the induction principle (4.11), we implement it by the following *equivalent* implementation, where we use the unary function symbol `succ₁`:

```
data N : D → Set where
  nzero : N zero                                          (4.12)
  nsucc : ∀ {n} → N n → N (succ₁ n).
```

```
N-ind : (A : D → Set) →
        A zero →
        (∀ {n} → A n → A (succ₁ n)) →                     (4.13)
        ∀ {n} → N n → A n
N-ind A A0 h nzero      = A0
N-ind A A0 h (nsucc Nn) = h (N-ind A A0 h Nn).
```

4.4 Proving Properties by Structural Recursion

We show the formalisation of some $\mathsf{LT}_{\mathsf{PCF}}$-proofs using the inductive representation described in § 4.3. In Chapter 3, the methodology for showing the proofs was as follows: first, we showed a proof using textbook-style, and then, we showed the corresponding formalised proof using our Agda representation. In the sequel, we shall omit the textbook-style proof and directly present the Agda version of the proof.

In our first examples, we show the termination proofs of some arithmetical properties on total and finite natural numbers.

Example 4.12. We start by introducing an $\mathsf{LT}_{\mathsf{PCF}}$-combinator for defining primitive recursive functions on the (partial) natural numbers using the fixed-point operator fix:

$$\mathsf{fix}\, r.\lambda n.\lambda a.\lambda f.\mathsf{if} \cdot (\mathsf{iszero} \cdot n) \cdot a \ \cdot (f \cdot (\mathsf{pred} \cdot n) \cdot (r \cdot (\mathsf{pred} \cdot n) \cdot a \cdot f)).$$

Using the representation of $\mathsf{LT}_{\mathsf{PCF}}$-terms as elements of the domain D presented in § 4.3 and convention 4.10, we implement the above combinator by

```
rech : D → D
rech r = lam (λ n → lam (λ a → lam (λ f →
           if (iszero₁ n)
             then a
             else f · (pred₁ n) · (r · (pred₁ n) · a · f))))

rec : D → D → D → D
rec n a f = fix rech · n · a · f.
```

The combinator `rec` satisfies the conversion rules

```
rec-0 : ∀ a {f} → rec zero a f ≡ a
rec-S : ∀ n a f → rec (succ₁ n) a f ≡ f · n · (rec n a f)
```

which can be proved by equational reasoning using the combinators introduced in § 2.2, since the $\mathsf{LT}_{\mathsf{PCF}}$-equality, implemented by the FOL-equality, is a preorder.

Addition of the (partial) natural numbers is recursively defined on the first argument by

```
_+_ : D → D → D                                                    (4.14)
m + n = rec m n (lam (λ _ → lam succ₁))
```

where its conversion rules

```
+-0x : ∀ n → zero + n ≡ n
+-Sx : ∀ m n → succ₁ m + n ≡ succ₁ (m + n)
```

can also be proved by equational reasoning.

The addition of total and finite natural numbers terminates. This is represented following (4.5) by the theorem

```
+-N : ∀ {m n} → N m → N n → N (m + n).                             (4.15)
```

Given the property

```
+-leftIdentity : ∀ n → zero + n ≡ n
+-leftIdentity = +-0x
```

the proof of (4.15) is implemented by pattern matching on the proof that the first argument is a total and finite natural number, and using the conversion rules for the addition and the substitutivity property of equality.

```
1  +-N : ∀ {m n} → N m → N n → N (m + n)
2  +-N {n = n} nzero Nn = subst N (sym (+-leftIdentity n)) Nn
3  +-N {n = n} (nsucc {m} Nm) Nn =
4    subst N (sym (+-Sx m n)) (nsucc (+-N Nm Nn)).
```

Line 2 corresponds to the proof of the base case `nzero`, where we use the totality hypothesis for `n` and a proof that `n ≡ zero + n`. Lines 3 and 4 correspond to the proof of the inductive step `nsucc Nm`, where we use the inductive hypothesis represented by the recursive call `+-N Nm Nn`, and a proof that `succ₁ (m + n) ≡ succ₁ m + n`.

The proof in the previous example shows the methodology we use in our programming logic $\mathsf{LT}_{\mathsf{PCF}}$ (which will also be used, with some improvements, in our programming logic described in Chapter 5). We first define a function `f` over the domain D of possibly partial elements. Recall that this domain is not an inductive data type, so we cannot use Agda's support for inductive types to prove properties of `f`. However, since the partial function `f` is intended to be used on *total* and *finite* elements, when we want to prove a property `P` of `f`, we add hypotheses requiring that the elements on which `f` will be applied should be total and finite. Given that the predicates stating the totality of the elements are inductively defined, the property `P` can be proved by *structural recursion* over these inductive predicates.

Example 4.13. Subtraction and multiplication on the (partial) natural numbers are also recursively defined using the `rec` combinator, and their associated conversion rules can also be proved by equational reasoning.

```
_∸_ : D → D → D
m ∸ n = rec n m (lam (λ _ → lam pred₁))

∸-x0 : ∀ n → n ∸ zero ≡ n
∸-xS : ∀ m n → m ∸ succ₁ n ≡ pred₁ (m ∸ n)

_*_ : D → D → D
m * n = rec m zero (lam (λ _ → lam (λ x → n + x)))

*-0x : ∀ n → zero * n ≡ zero
*-Sx : ∀ m n → succ₁ m * n ≡ n + m * n.
```

The termination of these operations is represented by the theorems

```
∸-N : ∀ {m n} → N m → N n → N (m ∸ n)
*-N : ∀ {m n} → N m → N n → N (m * n)
```

and their proofs are similar to the termination proof of the addition in Example 4.12. Namely, these proofs are performed by pattern matching on proofs that the arguments `m` and `n` are total and finite natural numbers, and using the conversion rules for the operations and the substitutivity property of equality.

The following example, we show how to prove by structural recursion properties of postulated functions when they are applied to total and finite values.

Example 4.14. In this example we prove the commutativity of addition of total and finite natural numbers represented by the formula

$$\forall m\ n.\ \mathcal{N}(m) \supset \mathcal{N}(n) \supset m + n = n + m. \tag{4.16}$$

The proof is implemented by pattern matching on the proof that the first argument is a total and finite natural number

```
+-comm : ∀ {m n} → N m → N n → m + n ≡ n + m.
```

The proof of the base case `nzero` is based on equational reasoning

```
+-comm {n = n} nzero Nn = zero + n ≡⟨ +-leftIdentity n ⟩
                          n        ≡⟨ sym (+-rightIdentity Nn) ⟩
                          n + zero ∎
```

where

```
+-rightIdentity : ∀ {n} → N n → n + zero ≡ n.
```

The proof of the step case is also based on equational reasoning

```
1  +-comm {n = n} (nsucc {m} Nm) Nn =
2    succ₁ m + n   ≡⟨ +-Sx m n ⟩
3    succ₁ (m + n) ≡⟨ succCong (+-comm Nm Nn) ⟩
4    succ₁ (n + m) ≡⟨ sym (x+Sy≡S[x+y] Nn m) ⟩
5    n + succ₁ m   ∎
```

where

```
succCong    : ∀ {m n} → m ≡ n → succ₁ m ≡ succ₁ n
x+Sy≡S[x+y] : ∀ {m} → N m → ∀ n → m + succ₁ n ≡ succ₁ (m + n).
```

This simple proof illustrates some features common to many $\mathsf{LT}_{\mathsf{PCF}}$-proofs. Given that the domain `D` is not inductively defined, we cannot defined `_+_` in (4.14) by structural recursion; still, we can use induction to prove the property `+-comm` by requiring that `m` and `n` are total and finite. On the other hand, since the function `_+_` is defined from *postulated* combinators (with associated *postulated* conversion rules), we cannot use Agda's type checker to fully normalise it as we did in the Example 3.18 (note the extra equational reasoning step in line 2 in the above proof). In general, the problem is that Agda does not know how to normalise an $\mathsf{LT}_{\mathsf{PCF}}$-program, so each normalisation step in the proof has to be performed manually. The conversion rules for the $\mathsf{LT}_{\mathsf{PCF}}$ basic combinators are given by postulates and hence they do not contribute to the usual normalisation provided by Agda's type checker. To compensate for this loss, we developed the Apia program, which will allow us to automate much of the reasoning needed in order to

prove this kind of theorems. The use of this program will be demonstrated in Chapter 6.

In the next example, we define some functions and predicates, and we prove a couple of properties related to inequalities of total and finite natural numbers, required by later examples,

Example 4.15. Using the fixed-point operator `fix`, we can define the less-than function

```
lt : D → D → D
```

on (partial) natural numbers, satisfying the following conversion rules:

```
lt-00 : lt zero zero ≡ false
lt-0S : ∀ n → lt zero (succ₁ n) ≡ true
lt-S0 : ∀ n → lt (succ₁ n) zero ≡ false
lt-SS : ∀ m n → lt (succ₁ m) (succ₁ n) ≡ lt m n.
```

Associated with the less-than function (`lt`) returning a truth value, we define the less-than relation (`_<_`), that is, an operator that returns a set, by

```
_<_ : D → D → Set
m < n = lt m n ≡ true.
```

Similarly, we define the not-less-than relation (`_≮_`):

```
_≮_ : D → D → Set
m ≮ n = lt m n ≡ false.
```

If `m` and `n` are total and finite natural numbers, the fact that `lt m n` terminates on a total and finite Boolean value formalised by

```
lt-Bool : ∀ {m n} → N m → N n → Bool (lt m n)
```

can be proved similarly to the proof of termination of the addition function in Example 4.12: the proof is by pattern matching on proofs that the arguments `m` and `n` are total and finite natural numbers, and it uses the conversion rules associated with the function, the substitutivity property of equality, and a recursive call.

The less-than-or-equal (`le`), greater-than (`gt`) and greater-than-or-equal (`ge`) functions are defined from the function `lt`. The related relations `_≤_`, `_≰_`, `_>_`, `_≯_`, `_≥_` and `_≱_` are defined similarly to the relations `_<_` and `_≮_`.

Using the above definitions, we can represent the theorem that if `m` and `n` are total and finite natural numbers, the order `_>_` is decidable

```
x>yvx≯y : ∀ {m n} → N m → N n → m > n ∨ m ≯ n
```

which can be proved by pattern matching on proofs that the arguments `m` and `n` are total and finite natural numbers.

4.5 Verification of General Recursive Programs

To illustrate how to reason about *general* recursive programs using $\mathsf{LT}_{\mathsf{PCF}}$, we shall verify the program which computes the greatest common divisor of two natural numbers using Euclid's algorithm. A naive Haskell implementation of the algorithm is given by

```
gcd :: Nat → Nat → Nat
gcd m n =
  if n == 0
    then m
    else if m == 0
           then n
           else if m > n then gcd (m - n) n else gcd m (n - m)
```

where we follow the common convention that `gcd 0 0 = 0` (see, for example, Knuth [1997]). The above program is not structurally recursive because its recursive calls are not on structurally smaller arguments.

The gcd algorithm. The $\mathsf{LT}_{\mathsf{PCF}}$-program which computes the `gcd` program above is given by

$$
\begin{array}{l}
\mathsf{fix}\ g.\lambda m.\lambda n. \\
\ \mathsf{if} \cdot (\mathsf{iszero} \cdot n) \cdot \\
\quad m \cdot \\
\quad (\mathsf{if} \cdot (\mathsf{iszero} \cdot m) \cdot n \cdot (\mathsf{if} \cdot (\mathsf{gt}\ m\ n) \cdot (g \cdot (m \dot{-} n) \cdot n) \cdot (g \cdot m \cdot (n \dot{-} m))))
\end{array}
$$

We now use the representation of the $\mathsf{LT}_{\mathsf{PCF}}$-terms to formalise the `gcd` algorithm:

```
gcdh : D → D
gcdh g = lam (λ m → lam (λ n →
           if (iszero₁ n)
             then m
             else (if (iszero₁ m)
                     then n
                     else (if (gt m n)
                             then g · (m ∸ n) · n
                             else g · m · (n ∸ m)))))

gcd : D → D → D
gcd m n = fix gcdh · m · n.
```

From the definition of `gcd`, we prove the following five conversion rules,

which will be useful when proving properties about the algorithm:

```
gcd-00  : gcd zero zero ≡ zero
gcd-S0  : ∀ n → gcd (succ₁ n) zero ≡ succ₁ n
gcd-0S  : ∀ n → gcd zero (succ₁ n) ≡ succ₁ n
gcd-S>S : ∀ m n → succ₁ m > succ₁ n →
            gcd (succ₁ m) (succ₁ n) ≡
            gcd (succ₁ m ∸ succ₁ n) (succ₁ n)
gcd-S≯S : ∀ m n → succ₁ m ≯ succ₁ n →
            gcd (succ₁ m) (succ₁ n) ≡
            gcd (succ₁ m) (succ₁ n ∸ succ₁ m).
```
(4.17)

Although the above conversion rules follow rather straightforwardly from the definition of `gcd` and the conversion rules of $\mathsf{LT}_{\mathsf{PCF}}$, proving them using equational reasoning is a surprisingly time-consuming task in our formalisation. As mentioned in connection to Example 4.14, the problem is that Agda does not know how to normalise the `gcd` program and many steps need to be performed manually. However, the above conversion rules can be automatically proved by the ATPs with the help of our Apia program, as we shall illustrate in Chapter 6.

Next, we shall show that the gcd algorithm is correct and that it terminates with the greatest common divisor of its two inputs.

Termination of gcd algorithm. The termination theorem for `gcd` states that if `m` and `n` are total and finite natural numbers, then `gcd m n` is also a total and finite natural number:

```
gcd-N : ∀ {m n} → N m → N n → N (gcd m n).
```

We first prove the following theorems using the substitutivity property of equality:

```
gcd-00-N  : N (gcd zero zero)

gcd-0S-N  : ∀ {n} → N n → N (gcd zero (succ₁ n))

gcd-S0-N  : ∀ {n} → N n → N (gcd (succ₁ n) zero)

gcd-S>S-N : ∀ {m n} → N m → N n →
            N (gcd (succ₁ m ∸ succ₁ n) (succ₁ n)) →
            succ₁ m > succ₁ n →
            N (gcd (succ₁ m) (succ₁ n))

gcd-S≯S-N : ∀ {m n} → N m → N n →
            N (gcd (succ₁ m) (succ₁ n ∸ succ₁ m)) →
            succ₁ m ≯ succ₁ n →
            N (gcd (succ₁ m) (succ₁ n)).
```

These five theorems show that the left-hand sides of the conversion rules of `gcd` defined in (4.17) terminate. Note that the last two theorems have an extra hypothesis stating that the result of the recursive call corresponding to the case we are considering (that is, the right-hand side of the corresponding conversion rule) also terminates.

Now, given the lexicographic order on pairs of (partial) natural numbers

```
Lexi : D → D → D → D → Set
Lexi m n m' n' = m < m' ∨ m ≡ m' ∧ n < n'
```

we prove two auxiliary theorems. The first theorem concerns the termination of `gcd m n` when the total and finite numbers `m` and `n` are such that `m > n`.

```
gcd-x>y-N : ∀ {m n} → N m → N n →
            (∀ {o p} → N o → N p → Lexi o p m n → N (gcd o p)) →
            m > n →
            N (gcd m n).
```

The other auxiliary theorem is similar but concerns the case where `m ≯ n`.

```
gcd-x≯y-N : ∀ {m n} → N m → N n →
            (∀ {o p} → N o → N p → Lexi o p m n → N (gcd o p)) →
            m ≯ n →
            N (gcd m n).
```

Both theorems are proved by pattern matching on proofs that the arguments `m` and `n` are total and finite natural numbers. They use (some of) the previous five theorems associated with the termination of the conversion rules of the `gcd`.

Next, we use well-founded induction on the lexicographical order `Lexi`

```
Lexi-wfind :
  (A : D → D → Set) →
  (∀ {m₁ n₁} → N m₁ → N n₁ →
    (∀ {m₂ n₂} → N m₂ → N n₂ → Lexi m₂ n₂ m₁ n₁ → A m₂ n₂) →
      A m₁ n₁) →
  ∀ {m n} → N m → N n → A m n
```

to prove that `gcd` terminates. The induction principle `Lexi-wfind` is proved by pattern matching on proofs that `m` and `n` are total and finite natural numbers.

Note that if in `Lexi-wfind` we take `A` such that

```
A : D → D → Set
A m n = N (gcd m n)
```

then, the type that results from `Lexi-wfind`, when applied to `m` and `n` and proofs that they are total and finite, is the same as the type that results from the auxiliary theorems `gcd-x>y-N` and `gcd-x≯y-N`. Moreover, the third explicit premises in `gcd-x>y-N` and `gcd-x≯y-N` correspond to the premise of the step case in the induction principle `Lexi-wfind`.

Finally, the proof of termination of `gcd` is formalised as

```
gcd-N : ∀ {m n} → N m → N n → N (gcd m n)
gcd-N = Lexi-wfind A h
  where
  A : D → D → Set
  A i j = N (gcd i j)

  h : ∀ {i j} → N i → N j →
      (∀ {k l} → N k → N l → Lexi k l i j → A k l) → A i j
  h Ni Nj ah =
    case (gcd-x>y-N Ni Nj ah) (gcd-x≯y-N Ni Nj ah)
         (x>yvx≯y Ni Nj).
```

Correctness of gcd algorithm. We now show that `gcd` returns the greatest common divisor of two total and finite numbers.

Let us first define the divisibility relation by

```
_|_ : D → D → Set
m | n = ∃[ k ] N k ∧ n ≡ k * m.
```

That is, `m` divides `n` if there exists a total and finite natural number `k` such `n ≡ k * m`. Note that, according to our convention that the greatest common divisor of `zero` and `zero` is `zero`, `zero` divides `zero`. Therefore the `gcd` of two numbers `m` and `n` is a common divisor of `m` and `n` that is divisible by any common divisor of them; that is, `gcd m n` is the greatest common divisor of `m` and `n` according to the partial order `_|_`, and not to the usual greater-than order.

That two elements have a common divisor is expressed by

```
CD : D → D → D → Set
CD m n cd = cd | m ∧ cd | n.
```

That a certain element is divisible by any common divisor is expressed by

```
Divisible : D → D → D → Set
Divisible m n o = ∀ cd → N cd → CD m n cd → cd | o.
```

That an element is the greatest common divisor is expressed by

```
gcdSpec : D → D → D → Set
gcdSpec m n gcd = CD m n gcd ∧ Divisible m n gcd.
```

Our correctness theorem is thus

```
gcdCorrect : ∀ {m n} → N m → N n → gcdSpec m n (gcd m n).
```

To prove this theorem we need only to combine a proof that the result of `gcd m n` is a common divisor of `m` and of `n`, and a proof that `gcd m n` is divisible by any common divisor of `m` and `n`.

The proof that `gcd m n` is a common divisor of `m` and `n`

```
gcdCD : ∀ {m n} → N m → N n → CD m n (gcd m n)
```

follows a similar structure to the proof of the termination of the `gcd`. Hence, we show only the main steps.

First, we prove the following five theorems:

```
gcd-00-CD  : CD zero zero (gcd zero zero)

gcd-0S-CD  : ∀ {n} → N n → CD zero (succ₁ n) (gcd zero (succ₁ n))

gcd-S0-CD  : ∀ {m} → N m → CD (succ₁ m) zero (gcd (succ₁ m) zero)

gcd-S>S-CD : ∀ {m n} → N m → N n →
             (CD (succ₁ m ∸ succ₁ n) (succ₁ n)
               (gcd (succ₁ m ∸ succ₁ n) (succ₁ n))) →
             succ₁ m > succ₁ n →
             CD (succ₁ m) (succ₁ n) (gcd (succ₁ m) (succ₁ n))

gcd-S≯S-CD : ∀ {m n} → N m → N n →
             (CD (succ₁ m) (succ₁ n ∸ succ₁ m)
               (gcd (succ₁ m) (succ₁ n ∸ succ₁ m))) →
             succ₁ m ≯ succ₁ n →
             CD (succ₁ m) (succ₁ n) (gcd (succ₁ m) (succ₁ n))
```

which state that the common divisor relation holds for the result of the left-hand side of the five conversion rules of `gcd` presented in (4.17). As in the termination proof of the `gcd` program, in the two cases where it performs a recursive call, we need to add an extra hypothesis stating that the property holds for the result of the recursive call.

Next, we prove the auxiliary theorems stating that `gcd m n` is a common divisor of `m` and `n`, both when `m` and `n` are such that `m > n` and when `m ≯ n`. Then, we wrap up all the intermediate results using the well-founded induction `Lexi-wfind` on the lexicographic order `Lexi` to obtain the desired result, namely, that `gcd m n` is a common divisor of `m` and of `n`.

Now, the proof that `gcd m n` is divisible by any common divisor of `m` and `n`

```
gcdDivisible : ∀ {m n} → N m → N n → Divisible m n (gcd m n)
```

follows the same structure as the previous proofs.

Putting all this together, our correctness proof becomes:

```
gcdCorrect : ∀ {m n} → N m → N n → gcdSpec m n (gcd m n)
gcdCorrect Nm Nn = gcdCD Nm Nn , gcdDivisible Nm Nn.
```

Chapter 5

First-Order Theory of Combinators

A goal of this thesis is to provide a programming logic for proving properties of lazy functional programs by combining interactive and automatic proofs. The interactive proofs will be performed with the help of Agda's proof engine and the automatic proofs will be performed by off-the-shelf ATPs. The first-order theory of combinators (henceforth, FOTC) is our programming logic defined for this purpose.

$\mathsf{LT}_{\mathsf{PCF}}$ is a programming logic for a version of PCF, which is not strictly a first-order theory, where we deal with inductive data types for Booleans and natural numbers. FOTC is a strictly first-order theory, where we deal with additional inductive and co-inductive data types like lists, trees, streams and so on, provided that they meet certain properties.

Using FOTC, we can prove properties of programs over total (finite or potentially infinite) values defined by nested recursion, higher-order recursion, without a termination proof, guarded co-recursion or unguarded co-recursion. In Chapter 7, we shall verify some programs with these characteristics. The properties are proved using equational reasoning, induction and/or co-induction.

Since we want to use automatic proofs performed by ATPs, in § 5.1 we define FOTC as a first-order version of $\mathsf{LT}_{\mathsf{PCF}}$. In § 5.2, we discuss how to represent higher-order functions in FOTC. In § 5.3, we establish the conditions needed in order to add new inductive data types while keeping the consistency of the theory, and we show some examples of such new inductively defined types. In § 5.4, we show an alternative formalisation of the inductive data types, which will be the base of our formalisation of the co-inductive data types. In § 5.5, we establish the conditions needed in order to add co-inductive data types keeping the consistency of the theory as before, and we show some examples of such co-inductively defined types.

5.1 A First-Order Theory

FOTC is a classical first-order theory with equality. We start by defining FOTC as a first-order version of $\mathsf{LT}_{\mathsf{PCF}}$. A λ-expression can be turned into an FOTC-term by performing λ-lifting [Peyton Jones 1987]. Instead of using λ-expressions, we add a new function symbol for each recursive function definition of the form

$$\mathsf{f} \cdot x_1 \cdot \cdots \cdot x_n = e[\mathsf{f}, x_1, \ldots, x_n]. \tag{5.1}$$

Consequently, the grammar for the terms of FOTC is now

$$\begin{array}{lll}
\mathsf{Terms} \ni t ::= x & & \text{variable} \\
\quad \mid t \cdot t & & \text{application} \\
\quad \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if} & & \text{partial Boolean constants} \\
\quad \mid 0 \mid \mathsf{succ} \mid \mathsf{pred} \mid \mathsf{iszero} & & \text{partial natural number constants} \\
\quad \mid \mathsf{f} & & \text{function constant}
\end{array}$$

where f ranges over new combinators defined by (5.1).

Note that FOTC is a first-order theory: there are no λ-abstractions. Note also that there is no fixed-point operator. If we start with a term in $\mathsf{LT}_{\mathsf{PCF}}$, we can always translate it into an equation in FOTC by performing λ-lifting and replacing sub-expressions of the form $\mathsf{fix}\, x.t$ by recursive definitions. For this, we need to introduce new function symbols and new (possibly recursive) equations.

The FOTC-formulae are the same as the $\mathsf{LT}_{\mathsf{PCF}}$-formulae, that is, they are generated by the following grammar:

$$\begin{array}{lll}
\mathsf{Formulae} \ni A ::= \top \mid \bot & & \text{truth, falsehood} \\
\quad \mid A \supset A \mid A \wedge A \mid A \vee A & & \text{binary logical connectives} \\
\quad \mid \forall x.A \mid \exists x.A & & \text{quantifiers} \\
\quad \mid t = t & & \text{equality} \\
\quad \mid P(t, \ldots, t) & & \text{predicate} \\
\quad \mid \mathcal{B}ool(t) & & \text{total Booleans predicate} \\
\quad \mid \mathcal{N}(t) & & \text{total natural numbers predicate}
\end{array}$$

Conventions 5.1. We follow the same conventions as with the $\mathsf{LT}_{\mathsf{PCF}}$-formulae (see conventions 4.1). Moreover, we continue using the more convenient function symbols `if_then_else_`, `succ`$_1$, and so on (see convention 4.10).

Terms t and formulae A are formed in the usual way. The FOTC non-logical axioms and axiom schemata (conversion rules, discrimination rules, introduction and elimination rules for the inductive predicates $\mathcal{B}ool$ and $\mathcal{N}$) are the same as those for $\mathsf{LT}_{\mathsf{PCF}}$ with the following exceptions: (i) we removed

the conversion rules (4.1b) related to the λ-abstraction and the fixed-point operator and (ii) we added the conversion rules generated by (5.1).

For our Agda formalisation of FOTC, we use an inductive representation like the one used for $\mathsf{LT}_{\mathsf{PCF}}$ and described in § 4.3, removing the four postulates related to the λ-abstraction and the fixed-point operator—we removed the terms `lam` and `fix` in (4.8), and the conversion rules `beta` and `fix-eq` in (4.9). Although FOTC is a first-order version of $\mathsf{LT}_{\mathsf{PCF}}$ which will allow the combination of interactive and automatics proofs as described in Chapter 6, most of the $\mathsf{LT}_{\mathsf{PCF}}$-proofs are valid FOTC-proofs in the following sense. In $\mathsf{LT}_{\mathsf{PCF}}$, we define a new function symbol from the fixed set of terms of $\mathsf{LT}_{\mathsf{PCF}}$ (or from a function symbol defined from this fixed set) and we prove some conversion rules associated with it using equational reasoning. From these conversion rules, we can prove in $\mathsf{LT}_{\mathsf{PCF}}$ properties of the new function symbol. In FOTC, we introduce the new function symbol and its associated recursive equations (5.1) as axioms. From these equations, we also prove the same conversion rules by equational reasoning, therefore an $\mathsf{LT}_{\mathsf{PCF}}$-proof of a property, which does not use the non-FOL features of $\mathsf{LT}_{\mathsf{PCF}}$, is also a valid proof of the corresponding property in FOTC.

In the following example, we show how to define a new recursive function using an equation of the form (5.1).

Example 5.2. For our first FOTC example, we shall not use the function symbols added in convention 4.10.

Instead of defining the addition of natural numbers using the combinator `rec`—defined using the λ-abstraction and the fixed-point operator—we add the function constant `add` and its associated recursive equation.

```
postulate
  add    : D
  add-eq : ∀ m n → add · m · n ≡
             if · (iszero · m) · n ·
               (succ · (add · (pred · m) · n)).
```

From the equation `add-eq` it is possible to prove the conversion rules

```
postulate
  add-0x : ∀ n → add · zero · n ≡ n
  add-Sx : ∀ m n → add · (succ · m) · n ≡ succ · (add · m · n)
```

by equational reasoning.

Where appropriate, we shall define n-ary function symbols instead of (nullary) function constants. Convention 4.10 illustrated how to define n-ary function symbols from function constants and the FOTC-terms. In the following examples, we illustrate this.

Example 5.3. The $\mathsf{LT}_{\mathsf{PCF}}$-proofs proving the termination and the commutativity of the addition of total and finite natural numbers shown in Examples 4.12 and 4.14, respectively, are implemented in FOTC as follows.

Instead of defining the addition of natural numbers by adding a function constant, we add the binary function symbol `_+_` and its associated recursive equation.

```
postulate
  _+_  : D → D → D
  +-eq : ∀ m n →
         m + n ≡ if (iszero₁ m) then n else succ₁ (pred₁ m + n).
```

From the equation `+-eq` it is possible to prove by equational reasoning the conversion rules

```
+-0x : ∀ n → zero + n ≡ n
+-Sx : ∀ m n → succ₁ m + n ≡ succ₁ (m + n).
```

As the proofs of the Theorems 4.15 and 4.16 in $\mathsf{LT}_{\mathsf{PCF}}$ are based on these conversion rules, the $\mathsf{LT}_{\mathsf{PCF}}$-proofs in Examples 4.12 and 4.14 are also valid proofs of the corresponding theorems in FOTC.

For convenience, we might actually define function symbols by directly introducing the conversion rules associated to the recursive definition, whenever it is clear that these conversion rules can be replaced by a single recursive equation of the form (5.1) using the FOTC-terms. In the following examples, we illustrate this.

Example 5.4. Instead of defining the addition of natural numbers using the equation `+-eq` in Example 5.3, we define it by the following equations:

```
postulate
  _+_  : D → D → D
  +-0x : ∀ n → zero + n ≡ n
  +-Sx : ∀ m n → succ₁ m + n ≡ succ₁ (m + n).
```

Similarly, we define subtraction and multiplication of natural numbers by the following equations:

```
postulate
  _∸_  : D → D → D
  ∸-x0 : ∀ n → n ∸ zero ≡ n
  ∸-xS : ∀ m n → m ∸ succ₁ n ≡ pred₁ (m ∸ n)

  _*_  : D → D → D
  *-0x : ∀ n → zero * n ≡ zero
  *-Sx : ∀ m n → succ₁ m * n ≡ n + m * n.
```

In the following example, we discuss the verification of the program which computes the greatest common divisor of two natural numbers using Euclid's algorithm, which was showed in § 4.5, using FOTC.

Example 5.5. The representation of the gcd algorithm in FOTC is given by

```
postulate
  gcd    : D → D → D
  gcd-eq : ∀ m n → gcd m n ≡
             if (iszero₁ n)
               then m
               else (if (iszero₁ m)
                      then n
                      else (if (gt m n)
                             then gcd (m ∸ n) n
                             else gcd m (n ∸ m))).
```

From this representation of gcd, the conversion rules gcd-00, gcd-S0, gcd-0S, gcd-S>S and gcd-S≯S defined in (4.17) can be proved by equational reasoning. Since the termination proof and the correctness proof of the gcd algorithm given in § 4.5 are based on these conversion rules, both $\mathsf{LT}_{\mathsf{PCF}}$-proofs are also valid FOTC-proofs.

To establish the consistency of FOTC, we can build a model of FOTC as follows. We already discussed in § 4.2 how to build a domain model of $\mathsf{LT}_{\mathsf{PCF}}$. We can replace all function symbols and their defining equations (5.1) by introducing λ-abstractions for explicit definitions, fixed-point combinators for recursive definitions, and case analysis constants for definitions by pattern matching (see, for example, Dybjer [2004] and Mitchell [1996]).

5.2 Representation of Higher-Order Functions

Since FOTC is a first-order theory with application as a binary function symbol, that is `_·_ : D → D → D`, we can represent higher-order functions that for example take one function as argument by representing this function-argument as a constant and using `_·_` for applying this function-argument to its own arguments.

In the following example, we show how to represent a higher-order function in FOTC.

Example 5.6. The higher-order function that applies a unary function twice is formalised by the axioms

```
postulate
  twice    : D → D → D
  twice-eq : ∀ f x → twice f x ≡ f · (f · x).
```

Using the axiom `twice-eq`, we can prove for example, the following simple theorem:

```
twice-succ : ∀ n → twice succ n ≡ succ · (succ · n)
twice-succ n = twice succ n      ≡⟨ twice-eq succ n ⟩
               succ · (succ · n) ∎
```

In this particular example, given that the function `twice` is not recursive, it is better to formalise it by the following definition, which uses Agda's definitional equality (`=`) instead of FOTC's propositional equality (`≡`):

```
twice : D → D → D
twice f x = f · (f · x).
```

Using the above definition and Agda's normalisation, the proof of the theorem `twice-succ` is just

```
twice-succ : ∀ n → twice succ n ≡ succ · (succ · n)
twice-succ n = refl.
```

5.3 Adding New Inductive Predicates

Magnusson and Nordström [1994, p. 213] state: *"During the years we have learned that there is no such thing as 'the logic of programming' ... Programs are manipulating different kinds of objects, and it would be very awkward to code these objects into a fixed set of objects."* Consistent with the previous claim, FOTC is not *one* first-order theory, it is a *family* of first-order theories. We work with one FOTC for *each* verification problem. The function symbols are determined by the program we want to verify. The predicate symbols are determined by the inductively defined predicates we need in our proofs. In this section, we show some examples where we add new inductively defined predicates, and we spell out the conditions required for keeping the consistency of the theory.

The inductively defined predicates might not only be used for representing totality properties. In the following example, we illustrate this.

Example 5.7. We may add a new unary predicate symbol $\mathcal{E}ven$ with introduction rules stating that 0 is an even number and that even numbers are closed under the function which adds 2 to an even number, and the induction schema stating that $\mathcal{E}ven$ is the least predicate with those properties.

$$\frac{}{\mathcal{E}ven(0)}\,, \qquad \frac{\mathcal{E}ven(t)}{\mathcal{E}ven(\mathsf{succ} \cdot (\mathsf{succ} \cdot t))}\,,$$

$$\frac{\mathcal{E}ven(t) \qquad A(0) \qquad \begin{array}{c}[A(t')]\\ \vdots \\ A(\mathsf{succ}\cdot(\mathsf{succ}\cdot t'))\end{array}}{A(t)}\,.$$

Let $\mathbf{D}$ be a domain model of $\mathsf{LT}_{\mathsf{PCF}}$, that is, $\mathbf{D}$ is a domain model of FOTC as described in § 5.1. In order to characterise those inductively defined predicates that can be interpreted on $\mathbf{D}$, we shall describe an alternative presentation of them.

Let $\Psi(X, \overline{x})$ be a formula with a free predicate variable X and n free variables $\overline{x}$. The formula $\Psi(X, \overline{x})$ is called *X-positive* if all occurrences of the predicate variable X are positive.

Dybjer and Sander [1989] used the μ-calculus of Park [1976] as a logical setting for proving properties of functional programs by induction and co-induction. Park's μ-calculus is an extension of first-order classical logic with a μ-operator: for any X-positive formula $\Psi(X, \overline{x})$ we can form the formula $\mu X.\Psi(X, \overline{x})$, which represents an n-ary inductive predicate $\mathcal{P}$, with axioms stating that:

$$\mu X.\Psi(X, \overline{x}) \text{ is a } \textit{pre-fixed point} \text{ of } \Psi(X, \overline{x}), \tag{5.2a}$$

$$\mu X.\Psi(X, \overline{x}) \text{ is } \textit{least among all pre-fixed points} \text{ of } \Psi(X, \overline{x}). \tag{5.2b}$$

The axioms (5.2a) and (5.2b) correspond to the introduction and elimination rules for the predicate $\mathcal{P}$, respectively. These axioms together express that the formula $\mu X.\Psi(X, \overline{x})$ is the *least fixed-point* of the formula $\Psi(X, \overline{x})$. The n-ary inductively defined predicates $\mathcal{P}$ represented by the formulae $\mu X.\Psi(X, \overline{x})$ can be interpreted as subsets of $\mathbf{D}$ modelled as least fixed-points of monotone operators on subsets of $\mathbf{D}$ induced by the formulae $\Psi(X, \overline{x})$ [Moschovakis 1974]. The monotonicity of these operators is a consequence of the X-positivity of the formulae $\Psi(X, \overline{x})$ (see, for example, Moschovakis [1974], Dybjer and Sander [1989] and J. Harrison [1995]), and therefore it will be the condition required to add new inductively defined predicates.

Remark 5.8. The interpretation of inductively defined predicates as least fixed-points of monotone operators on subsets of $\mathbf{D}$ induced by formulae $\Psi(X, \overline{x})$ or induced by rule sets as shown in § 4.2 are inter-definable (see, for example, Dybjer [1988]). We use the latter interpretation because we think it is more adequate to inductive definitions of predicates, and we use the former interpretation because we think it is more adequate to spell out the conditions required for keeping the consistency of FOTC under the addition of new inductively defined predicates. This remark will also be applicable to the interpretation of the co-inductively defined predicates shown in § 5.5.

In the following example, we represent an inductive predicate by an X-positive formula.

Example 5.9. We can represent the inductive predicate $\mathcal{E}ven$ (see Example 5.7) by the formula

$$\Psi(X, x) \stackrel{\text{def}}{=} x = 0 \vee (\exists x'.\, x = \mathsf{succ} \cdot (\mathsf{succ} \cdot x') \wedge X(x')). \tag{5.3}$$

Given that the formula (5.3) is X-positive, the predicate $\mathcal{E}ven$ can be interpreted by an inductively defined set $I(\Phi)$ associated with a rule set Φ on $\mathbf{D}$, as we did for the interpretation of the predicates $\mathcal{B}ool$ and $\mathcal{N}$ in § 4.2.

Example 5.10 (Counterexample)**.** An alternative definition for the predicate of being an even natural numbers is:

- 0 is an even number, and
- $\mathsf{succ} \cdot n$ is an even number if n is *not* an even number.

The introduction rules for a unary inductive predicate $\mathcal{E}ven$ from the above description are

$$\frac{}{\mathcal{E}ven(0)}\,, \qquad \frac{\neg\, \mathcal{E}ven(t)}{\mathcal{E}ven(\mathsf{succ} \cdot t)}\,,$$

and the associated formula is

$$\Psi(X, x) \stackrel{\text{def}}{=} x = 0 \vee (\exists x'.\, x = \mathsf{succ} \cdot x' \wedge \neg\, X(x')). \tag{5.4}$$

Given that the formula (5.4) is not X-positive, the predicate $\mathcal{E}ven$ defined in this way is not a valid FOTC-predicate.

Remark 5.11. In the sequel, the new conversion and discrimination rules only hold in a lazy functional programming language and (some of them) should be different for a non-lazy language.

In the following example, we add new FOTC-terms as preparation for the addition of a new inductive predicate.

Example 5.12. FOTC basic data types are Booleans and natural numbers. To use lists we add the constants

$$\{[\,], \mathsf{cons}, \mathsf{null}, \mathsf{head}, \mathsf{tail}\} \tag{5.5}$$

to the set of FOTC-terms. Moreover, we add the conversion rules

$$\begin{aligned}
\mathsf{null} \cdot [\,] &= \mathsf{true},\\
\forall t\ ts.\ \mathsf{null} \cdot (\mathsf{cons} \cdot t \cdot ts) &= \mathsf{false},\\
\forall t\ ts.\ \mathsf{head} \cdot (\mathsf{cons} \cdot t \cdot ts) &= t,\\
\forall t\ ts.\ \mathsf{tail} \cdot (\mathsf{cons} \cdot t \cdot ts) &= ts,
\end{aligned}$$

and the discrimination rule

$$\forall t\ ts.\ [\,] \neq \mathsf{cons} \cdot t \cdot ts.$$

Let **LL** be the ω-cpo of lazy lists (see, for example, Schmidt [1986]). Let us add **LL** to the domain model **D**, that is, let **D** be a solution to the recursive domain equation

$$\mathbf{D} \cong \mathbf{B}_{\perp} \oplus \mathbf{LN} \oplus \mathbf{LL} \oplus (\mathbf{D} \to \mathbf{D})_{\perp}.$$

Based on the fixed-point operator of the new domain, it is possible to interpret the terms, conversion and discrimination rules for partial lists on the new domain.

For the implementation of the lists, we postulate the terms and the conversion and discrimination rules in the usual way.

Convention 5.13. We define more readable function symbols of the appropriate types for some of the constants in (5.5). In particular, we define the right-associative function symbol

```
_::_ : D → D → D
x :: xs = cons · x · xs.
```

In the following example, we add a new inductive predicate to FOTC.

Example 5.14. Given the combinators for lists (5.5), we define a unary inductive predicate $\mathcal{L}ist(ts)$ representing that ts is a total and finite list of elements. The introduction rules for $\mathcal{L}ist$ are

$$\frac{}{\mathcal{L}ist([\,])}, \qquad \frac{\mathcal{L}ist(ts)}{\mathcal{L}ist(\mathsf{cons} \cdot t \cdot ts)}\ (t \in \mathsf{Terms}),$$

and its elimination rule is

$$\frac{\mathcal{L}ist(ts) \qquad A([\,]) \qquad \begin{array}{c}[A(ts')]\\ \vdots \\ A(\mathsf{cons} \cdot t \cdot ts')\end{array}}{A(ts)}$$

The predicate $\mathcal{L}ist$ is interpretable in the domain model **D** (appropriately extended) because the formula that represents it, given by

$$\Psi(X, x) \stackrel{\text{def}}{=} x = [\,] \vee (\exists x'\ xs'.\ x = \mathsf{cons} \cdot x' \cdot xs' \wedge X(xs'))$$

is X-positive.

We implement the introduction rules of $\mathcal{L}ist$ by an inductive family on `D`, as we did for the inductive implementation of the predicates $\mathcal{B}ool$ and $\mathcal{N}$ in § 4.3. It is not necessary to implement the elimination rule of $\mathcal{L}ist$ because we shall use Agda's pattern matching instead.

```
data List : D → Set where
  lnil  : List []
  lcons : ∀ x {xs} → List xs → List (x ∷ xs).
```

Note that in our implementation we have used the convention 5.13.

Recall that FOTC is a type-free theory. Using the inductive predicate $\mathcal{L}ist$, we can prove that functions defined on lists are terminating functions. In the following example, we illustrate this.

Example 5.15. We first define the length of a list by a function returning an element of `D`, then we use the inductive predicate $\mathcal{L}ist$ to prove that the length of a total and finite list is a total and finite natural number. The proof is by pattern matching on a proof that the list is total and finite.

```
postulate
  length    : D → D
  length-[] : length [] ≡ zero
  length-∷  : ∀ x xs → length (x ∷ xs) ≡ succ₁ (length xs)

length-N : ∀ {xs} → List xs → N (length xs)
length-N lnil = subst N (sym length-[]) nzero
length-N (lcons x {xs} Lxs) =
  subst N (sym (length-∷ x xs)) (nsucc (length-N Lxs)).
```

Using the inductive predicate $\mathcal{L}ist$, we can also prove the usual properties of total and finite lists. In the following examples, we prove some of these properties.

Example 5.16. We prove that the concatenation of total and finite lists is associative.

First, we define a right-associative concatenation function by postulating the following equations:

```
infixr 8 _++_
postulate
  _++_  : D → D → D
  ++-[] : ∀ ys → [] ++ ys ≡ ys
  ++-∷  : ∀ x xs ys → (x ∷ xs) ++ ys ≡ x ∷ (xs ++ ys).
```

Now, given the properties

```
++-leftCong : ∀ {xs ys zs} → xs ≡ ys → xs ++ zs ≡ ys ++ zs
++-leftCong refl = refl

∷-rightCong : ∀ {x xs ys} → xs ≡ ys → x ∷ xs ≡ x ∷ ys
∷-rightCong refl = refl
```

```
++-leftIdentity : ∀ xs → [] ++ xs ≡ xs
++-leftIdentity = ++-[]
```

the proof of the associativity of `_++_` is given by

```
++-assoc : ∀ {xs} → List xs → ∀ ys zs →
           (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)

++-assoc lnil ys zs =
  ([] ++ ys) ++ zs ≡⟨ ++-leftCong (++-leftIdentity ys) ⟩
  ys ++ zs         ≡⟨ sym (++-leftIdentity (ys ++ zs)) ⟩
  [] ++ ys ++ zs   ∎

++-assoc (lcons x {xs} Lxs) ys zs =
  ((x :: xs) ++ ys) ++ zs ≡⟨ ++-leftCong (++-:: x xs ys) ⟩
  (x :: (xs ++ ys)) ++ zs ≡⟨ ++-:: x (xs ++ ys) zs ⟩
  x :: ((xs ++ ys) ++ zs) ≡⟨ ::-rightCong (++-assoc Lxs ys zs) ⟩
  x :: (xs ++ ys ++ zs)   ≡⟨ sym (++-:: x xs (ys ++ zs)) ⟩
  (x :: xs) ++ ys ++ zs   ∎
```

Note that the proof is by pattern matching on a proof that the first explicit argument is `List xs`, therefore it is not necessary that the arguments `ys` and `zs` are total and finite lists.

Example 5.17. Using our approach for formalising higher-order functions described in § 5.2, we define the `map` function on lists by the following axioms:

```
postulate
  map    : D → D → D
  map-[] : ∀ f → map f [] ≡ []
  map-:: : ∀ f x xs → map f (x :: xs) ≡ f · x :: map f xs.
```

Now, using the inductive predicate `List` and given the property

```
mapRightCong : ∀ {f xs ys} → xs ≡ ys → map f xs ≡ map f ys
mapRightCong refl = refl
```

we prove the `map f` distributes through concatenation:

```
map-++ : ∀ f {xs} → List xs → ∀ ys →
         map f (xs ++ ys) ≡ map f xs ++ map f ys

map-++ f lnil ys =
  map f ([] ++ ys)     ≡⟨ mapRightCong (++-leftIdentity ys) ⟩
  map f ys             ≡⟨ sym (++-leftIdentity (map f ys)) ⟩
  [] ++ map f ys       ≡⟨ ++-leftCong (sym (map-[] f)) ⟩
  map f [] ++ map f ys ∎
```

```
map-++ f (lcons x {xs} Lxs) ys =
  map f ((x ∷ xs) ++ ys)
    ≡⟨ mapRightCong (++-∷ x xs ys) ⟩
  map f (x ∷ xs ++ ys)
    ≡⟨ map-∷ f x (xs ++ ys) ⟩
  f · x ∷ map f (xs ++ ys)
    ≡⟨ ∷-rightCong (map-++ f Lxs ys) ⟩
  f · x ∷ (map f xs ++ map f ys)
    ≡⟨ sym (++-∷ (f · x) (map f xs) (map f ys)) ⟩
  (f · x ∷ map f xs) ++ map f ys
    ≡⟨ ++-leftCong (sym (map-∷ f x xs)) ⟩
  map f (x ∷ xs) ++ map f ys ∎
```

Similarly to Example 5.16, the proof is by pattern matching on a proof that the second explicit argument is `List xs`, therefore it is not necessary that the argument `ys` is a total and finite list.

Mutually defined inductive predicates can be added to FOTC much in the same way as simple inductive predicates. The validity of them is based on the fact that one could instead use a simple inductive predicate and the combinators for disjoint union in order to model the formalisation of mutually defined inductive predicates [Paulson 1994a].

In the following example, we show how to add mutually defined inductive predicates to FOTC.

Example 5.18. Let $\mathbf{D}$ be a domain model of FOTC, the mutually defined inductive predicates

$$\frac{}{\mathcal{E}ven(0)}\,, \qquad \frac{\mathcal{E}ven(t)}{\mathcal{O}dd(\mathsf{succ}\cdot t)}\,, \qquad \frac{\mathcal{O}dd(t)}{\mathcal{E}ven(\mathsf{succ}\cdot t)}\,, \tag{5.6}$$

can instead be defined by [Blanchette 2013]

$$\mathcal{E}ven(t) \stackrel{\text{def}}{=} \mathcal{E}ven\mathcal{O}dd(\mathsf{inl}\ t),$$
$$\mathcal{O}dd(t) \stackrel{\text{def}}{=} \mathcal{E}ven\mathcal{O}dd(\mathsf{inr}\ t),$$

where inl and inr would be the combinators for disjoint union, and $\mathcal{E}ven\mathcal{O}dd$ is the inductive predicate

$$\frac{}{\mathcal{E}ven\mathcal{O}dd(\mathsf{inl}\ 0)}\,, \qquad \frac{\mathcal{E}ven\mathcal{O}dd(\mathsf{inl}\ t)}{\mathcal{E}ven\mathcal{O}dd(\mathsf{inr}(\mathsf{succ}\cdot t))}\,, \qquad \frac{\mathcal{E}ven\mathcal{O}dd(\mathsf{inr}\ t)}{\mathcal{E}ven\mathcal{O}dd(\mathsf{inl}(\mathsf{succ}\cdot t))}\,,$$

whose interpretation on $\mathbf{D} \oplus \mathbf{D}$ is valid given that its associated formula $\Psi(X, x)$ is X-positive.

Given Agda's support for mutual definitions, we formalise the inductive predicates in (5.6) by two mutually inductive families.

```
data Even : D → Set
data Odd  : D → Set

data Even where
  ezero : Even zero
  esucc : ∀ {n} → Odd n → Even (succ₁ n)

data Odd where
  osucc : ∀ {n} → Even n → Odd (succ₁ n).
```

5.4 Alternative Formalisation of Inductive Predicates

Before describing the addition of co-inductive predicates to FOTC, we show an alternative formalisation in Agda of the inductive predicates based on the axioms (5.2). This formalisation will be the base for our formalisation of the co-inductive predicates within the realm of FOL.

We do not have a general μ-operator (a second-order construct) in FOTC. Instead, given an X-positive formula $\Psi(X, \overline{x})$, we can introduce a new n-ary predicate symbol $\mathcal{P}$, with introduction and elimination rules corresponding to the fact that the meaning of $\mathcal{P}$ is $\mu X.\Psi(X, \overline{x})$.

In the following example, we show an alternative formalisation of the inductive predicate $\mathcal{N}$, which is used for representing total and finite natural numbers.

Example 5.19. Given the X-positive formula

$$\Psi(X, x) \stackrel{\text{def}}{=} x = 0 \vee (\exists x'.\, x = \mathsf{succ} \cdot x' \wedge X(x')), \tag{5.7}$$

we could implement the unary inductive predicate $\mathcal{N}$ (given by the introduction and elimination rules (4.4)) without using Agda's **data** constructor. The implementation represents that $\mathcal{N}$ is $\mu X.\Psi(X, x)$ and it is given by postulating the following constants: the predicate symbol `N`, the introduction rule `N-in`, that is, `N` is a pre-fixed point of (5.7), and the elimination rule `N-ind'`, that is, `N` is the least pre-fixed point of (5.7).

```
postulate
  N : D → Set

  N-in : ∀ {n} → n ≡ zero ∨ (∃[ n' ] n ≡ succ₁ n' ∧ N n') → N n

  N-ind' : (A : D → Set) →
           (∀ {n} → n ≡ zero ∨                                  (5.8)
             (∃[ n' ] n ≡ succ₁ n' ∧ A n') → A n) →
           ∀ {n} → N n → A n.
```

In the following example, we show that the implementation of the inductive predicate $\mathcal{N}$ using the least-fixed point approach presented above and the inductive approach presented in Section 5.1 are equivalents.

Example 5.20. Given the least-fixed point implementation of $\mathcal{N}$, that is, axioms (5.8), we can define the data constructors `nzero` and `nsucc` given by (4.12), and the induction principle `N-ind` given by (4.13).

```
nzero : N zero
nzero = N-in (inj₁ refl)

nsucc : ∀ {n} → N n → N (succ₁ n)
nsucc Nn = N-in (inj₂ (_ , refl , Nn))

N-ind : (A : D → Set) →
        A zero →
        (∀ {n} → A n → A (succ₁ n)) →
        ∀ {n} → N n → A n
N-ind A A0 h = N-ind' A h'
  where
  h' : ∀ {m} → m ≡ zero ∨ (∃[ m' ] m ≡ succ₁ m' ∧ A m') → A m
  h' (inj₁ m≡0)              = subst A (sym m≡0) A0
  h' (inj₂ (m' , prf , Am')) = subst A (sym prf) (h Am').
```

Now, given the inductive implementation of $\mathcal{N}$, that is, the inductive family (4.12) and the induction principle (4.13), we can define the axioms (5.8).

```
N-in : ∀ {n} → n ≡ zero ∨ (∃[ n' ] n ≡ succ₁ n' ∧ N n') → N n
N-in {n} h = case prf₁ prf₂ h
  where
  prf₁ : n ≡ zero → N n
  prf₁ n≡0 = subst N (sym n≡0) nzero

  prf₂ : ∃[ n' ] n ≡ succ₁ n' ∧ N n' → N n
  prf₂ (n' , prf , Nn') = subst N (sym prf) (nsucc Nn')

N-ind' :
  (A : D → Set) →
  (∀ {n} → n ≡ zero ∨ (∃[ n' ] n ≡ succ₁ n' ∧ A n') → A n) →
  ∀ {n} → N n → A n
N-ind' A h = N-ind A h₁ h₂
  where
  h₁ :  A zero
  h₁ = h (inj₁ refl)

  h₂ : ∀ {m} → A m → A (succ₁ m)
  h₂ {m} Am = h (inj₂ (m , refl , Am)).
```

Remark 5.21. We could represent all the inductively defined predicates following the least-fixed point approach shown in Example 5.19, but to get the most out of Agda's support for inductive types, we instead use Agda's **data** construct.

5.5 Adding Co-Inductive Predicates

We shall use co-inductively defined predicates for reasoning about functional programs with total and potentially infinite elements (see, for example, Gordon [1995] and Gibbons and Hutton [2005]). For the definition of co-inductive predicates, Dybjer and Sander [1989] define a ν-operator by dualisation of the μ-operator. Let $\Psi(X, \overline{x})$ be an X-positive formula, then we can form the formula $\nu X.\Psi(X, \overline{x})$ representing an n-ary co-inductive predicate $\mathcal{P}$. From the axioms for the μ-operator, they derive two properties for the ν-operator:

$$\nu X.\Psi(X, \overline{x}) \text{ is a } \textit{post-fixed point} \text{ of } \Psi(X, \overline{x}), \tag{5.9a}$$

$$\nu X.\Psi(X, \overline{x}) \text{ is } \textit{greatest among all post-fixed points} \text{ of } \Psi(X, \overline{x}). \tag{5.9b}$$

The properties (5.9a) and (5.9b) correspond to the unfolding rule and the co-induction rule for the co-inductive predicate $\mathcal{P}$, respectively. These properties together express that the formula $\nu X.\Psi(X, \overline{x})$ is the *greatest fixed-point* of $\Psi(X, \overline{x})$. In this case, we shall introduce a new n-ary predicate symbol $\mathcal{P}$, with unfolding and co-induction rules corresponding to the fact that the meaning of $\mathcal{P}$ is $\nu X.\Psi(X, \overline{x})$.

In the following example, we show the implementation of a co-inductive predicate in FOTC.

Example 5.22. As an example of a co-inductive definition, we take the natural numbers with the number $\infty = \mathsf{succ} \cdot \infty$—the fixed-point of the successor function obtained by lazy evaluation. From (5.7), we implement a co-inductive predicate $\mathcal{C}onat(t)$ representing that t is a total and potentially infinite natural number.

```
postulate
  Conat : D → Set

  Conat-out : ∀ {n} → Conat n →
              n ≡ zero ∨ (∃[ n' ] n ≡ succ₁ n' ∧ Conat n')

  Conat-coind : (A : D → Set) →
                (∀ {n} → A n →
                  n ≡ zero ∨ (∃[ n' ] n ≡ succ₁ n' ∧ A n')) →
                ∀ {n} → A n → Conat n.
```

The constant `Conat` defines the unary predicate. The constant `Conat-out` implements the unfolding rule, that is, the predicate `Conat` is a post-fixed point of (5.7), and the constant `Conat-coind` implements the co-induction rule, that is, the predicate `Conat` is the greatest post-fixed point of (5.7).

Remark 5.23. Unlike the implementation of the inductive predicates, we shall not use Agda's co-inductive operators for the implementation of the co-inductive predicates of FOTC because they are still very experimental.

In the following example, we illustrate the use of the co-induction rule of the co-inductive predicate $\mathcal{C}onat$.

Example 5.24. We want to prove that the number ∞ is a total and potentially infinite natural number. First, we add the recursive equation `∞ ≡ succ₁ ∞`, which implements that `∞` is a fixed-point of the successor function. To prove the intended property, we instantiate `Conat-coind` with the predicate `λ n → n ≡ ∞`.

```
postulate
  ∞    : D
  ∞-eq : ∞ ≡ succ₁ ∞

∞-Conat : Conat ∞
∞-Conat = Conat-coind A h refl
  where
  A : D → Set
  A n = n ≡ ∞

  h : ∀ {n} → A n → n ≡ zero ∨ (∃[ n' ] n ≡ succ₁ n' ∧ A n')
  h An = inj₂ (∞ , trans An ∞-eq , refl).
```

In FOTC, we can nicely mix inductive and co-inductive notions. In the following example, we show this.

Example 5.25. The proof that every total and finite natural number is a co-inductive natural number is given by

```
N→Conat : ∀ {n} → N n → Conat n
N→Conat Nn = Conat-coind N h Nn
 where
  h : ∀ {m} → N m → m ≡ zero ∨ (∃[ m' ] m ≡ succ₁ m' ∧ N m')
  h nzero          = inj₁ refl
  h (nsucc {m} Nm) = inj₂ (m , refl , Nm).
```

The co-induction rule `Conat-coind` is instantiated with the inductive predicate `N` for total and finite natural numbers, and the hypothesis required is proved by pattern matching on a proof that the number `n` is a total and finite natural number.

Before showing additional examples of co-inductively defined predicates in FOTC, we discuss the consistency of FOTC when working with co-inductive predicates. The n-ary co-inductively defined predicates $\mathcal{P}$ represented by formulae $\nu X.\Psi(X, \overline{x})$ can be interpreted as greatest fixed-points of monotone operators on subsets of the domain model $\mathbf{D}$ induced by the formulae $\Psi(X, \overline{x})$. Again, the monotonicity of these operators is a consequence of the X-positivity of the formulae $\Psi(X, \overline{x})$, and therefore it will also be the condition required to add co-inductively defined predicates.

As we did in § 4.2, we shall use the set-theoretic notion of rule set to interpret the co-inductive predicates on the domain model $\mathbf{D}$, based on Aczel's definition of the dual of an inductive definition [Aczel 1977a].

Let Φ be a rule set on U. A set A is Φ-*dense* if for every $x \in A$ there is a set $X \subseteq A$ such that $\langle X, x\rangle \in \Phi$. The *co-inductively defined set* by Φ is the largest Φ-dense set defined by

$$K(\Phi) = \bigcup\{A \subseteq U \mid A \text{ is } \Phi\text{-dense}\}. \tag{5.10}$$

Example 5.26. Given that introduction rules for the predicates $\mathcal{N}$ and $\mathcal{C}onat$ are the same, we have that the rule set $\Phi_{\mathcal{C}onat}$ on $\mathbf{D}$ associated with the predicate $\mathcal{C}onat$ is the rule set $\Phi_{\mathcal{N}}$ (Example 4.5). If we add ∞ to $\mathbf{D}$, the interpretation of co-inductive predicate $\mathcal{C}onat$ on $\mathbf{D}$ is given by

$$\begin{aligned}\mathbf{Conat} &= K(\Phi_{\mathcal{C}onat})\\ &= \mathbf{N} \cup \infty.\end{aligned}$$

Remark 5.27. The co-inductively defined sets by a rule set can also be defined as greatest fixed-points of monotone operators. A rule set Φ is *finite in the conclusions*, if for each x, the set $\{X \mid \langle X, x\rangle \in \Phi\}$ is finite, that is, there is only a finite number of rules whose conclusion is x. Let be Φ a rule set on U finite in the conclusions, and let $\widehat{\Phi} : \mathrm{Pow}(U) \to \mathrm{Pow}(U)$ be the monotone operator induced by Φ. The co-inductively defined set by Φ is the greatest fixed-point of $\widehat{\Phi}$ defined by [Sangiorgi 2012]

$$K(\Phi) = \bigcap_{n\in\omega} \widehat{\Phi}^n(U). \tag{5.11}$$

Now, we show additional examples of co-inductively defined predicates in FOTC. In the following example, we implement a new co-inductive predicate for total and potentially infinite list of elements.

Example 5.28. From the X-positive formula

$$\Psi(X, xs) \stackrel{\text{def}}{=} \exists x'\, xs'.\, xs = \mathsf{cons} \cdot x' \cdot xs' \wedge X(xs'), \tag{5.12}$$

we implement the co-inductive predicate $\mathcal{S}tream(ts)$ representing that ts is a stream, that is, a total and potentially infinite list of elements.

```
postulate
  Stream : D → Set

  Stream-out : ∀ {xs} → Stream xs →
               ∃[ x' ] ∃[ xs' ] xs ≡ x' :: xs' ∧ Stream xs'

  Stream-coind : (A : D → Set) →
                 (∀ {xs} → A xs →
                   ∃[ x' ] ∃[ xs' ] xs ≡ x' :: xs' ∧ A xs') →
                 ∀ {xs} → A xs → Stream xs.
```

The constant `Stream` defines the unary predicate, the constant `Stream-out` implements the unfolding rule, that is, the predicate `Stream` is a post-fixed point of (5.12), and the constant `Stream-coind` implements the co-induction rule, that is, the predicate `Stream` is the greatest post-fixed point of (5.12).

In the following example, we show how to use the co-inductive predicate *Stream* for proving a property.

Example 5.29. Using the unfolding rule `Stream-out`, we can prove for example that if `x :: xs` is a stream, then `xs` is also a stream.

Given a proof that the function _::_ is injective

```
::-injective : ∀ {x y xs ys} → x :: xs ≡ y :: ys → x ≡ y ∧ xs ≡ ys
```

the proof of the desired property is given by

```
::-Stream : ∀ {x xs} → Stream (x :: xs) → Stream xs
::-Stream {x} {xs} h = ::-Stream-helper (Stream-out h)
  where
  ::-Stream-helper :
    ∃[ x' ] ∃[ xs' ] x :: xs ≡ x' :: xs' ∧ Stream xs' → Stream xs
  ::-Stream-helper (x' , xs' , prf , Sxs') =
    subst Stream (sym (∧-proj₂ (::-injective prf))) Sxs'.
```

We have `h : Stream (x :: xs)`. Then using the auxiliary function `::-Stream-helper`, we pattern match on the intermediate values given from `Stream-out h`: `x'` and `xs'` of type `D`, `prf` of type `x :: xs ≡ x' :: xs'` and `Sxs'` of type `Stream xs'`. From `prf` and `Sxs'`, given the injectivity of _::_, we can prove `Stream xs`.

Suitable notions of equality between total and potentially infinite terms can be defined as binary co-inductive relations. In the following examples, we show the definition and the use of one of these notions of equality, respectively.

Example 5.30. The equality of total and potentially infinite numbers (which includes ∞) can be defined as the greatest fixed-point of the following X-positive formula:

$$\Psi(X, x, y) \stackrel{\text{def}}{=} (x = 0 \wedge y = 0) \vee (\exists x'\ y'.\ x = \mathsf{succ} \cdot x' \wedge y = \mathsf{succ} \cdot y' \wedge X(x', y')).$$

This can be formalised in Agda using the following postulates, which express the post-fixed point and the greatest post-fixed point properties:

```
postulate
  _≈N_ : D → D → Set

  ≈N-out :
    ∀ {m n} → m ≈N n →
    m ≡ zero ∧ n ≡ zero
      ∨ (∃[ m' ] ∃[ n' ] m ≡ succ₁ m' ∧ n ≡ succ₁ n' ∧ m' ≈N n')

  ≈N-coind :
    (R : D → D → Set) →
    (∀ {m n} → R m n →
      m ≡ zero ∧ n ≡ zero
        ∨ (∃[ m' ] ∃[ n' ]
             m ≡ succ₁ m' ∧ n ≡ succ₁ n' ∧ R m' n')) →
    ∀ {m n} → R m n → m ≈N n.
```

Example 5.31. As a simple example of the use of `≈N`, we prove that the length of a stream is ∞:

```
streamLength : ∀ {xs} → Stream xs → length xs ≈N ∞.
```

We use the co-induction rule `≈N-coind` for proving this property. Based on Sander [1992], we define the auxiliary relation

```
R : D → D → Set
R m n = ∃[ xs ] Stream xs ∧ m ≡ length xs ∧ n ≡ ∞.
```

To prove the first hypothesis required by `≈N-coind`, we assume `R m n` for arbitrary `m` and `n`. Hence `m = length xs` for some `xs` such that `Stream xs` and `n = ∞`. Moreover, by unfolding `Stream xs`, we can conclude that there are `x'` and `xs'` of type `D` such that `xs = x' ∷ xs'` and `Stream xs'`. Hence `m = succ₁ (length xs')` and `n = succ₁ ∞`, and therefore `m ≈N n` holds.

The second hypothesis required by `≈N-coind`, that is, `R (length xs) ∞` holds as a consequence of the assumption `Stream xs`.

The Appendix C contains the proof of the `streamLength` property.

Chapter 6

Combining Interactive and Automatic Proofs

The reasoning about programs such as gcd (Example 5.5) in our Agda implementation of FOTC is at a very low level compared with ordinary reasoning about programs in Agda when used as a proof assistant for dependent type theories. For example, judgments of the form `n : ℕ` (where ℕ is the inductive type of natural numbers defined in § 2.1) are automatically checked by Agda, whereas propositions of the form `N n` (where `N` is the inductive predicate for total and finite natural numbers implemented in (4.10)) have to be proved manually by constructing proof terms of type `N n`. Moreover, Agda can automatically normalise some terms by using definitional equality, whereas simplification using the postulated conversion rules for elements in the universe `D` of FOTC has to be done manually. However, much of this low-level reasoning can be done automatically with the help of, for example, automatic theorem provers for FOL.

By staying *strictly* within FOL, we shall be able to employ powerful ATPs for reasoning about functional programs. To this end, we have written the Apia program, a translator from our Agda representation of first-order formulae into the TPTP language understood by many ATPs. We have also extended the Agda system with a new pragma (henceforth, ATP-pragma) that instructs Agda, via the Apia program, to interact with the theorem provers by asking them for proofs, giving them hypotheses, and informing them of new definitions and axioms.

At the moment, we use the ATPs as oracles via the Apia program, that is, we trust the ATPs when they tell us that a proof exists. Therefore, in the sequel, when we write about an 'automatic proof of a formula A' or 'we automatically prove the formula A' this means that the user must: (i) to add to the Agda program the required ATP-pragmas, (ii) to run the Apia program on the corresponding Agda file and (iii) to verify that some ATP could prove the formula A. As a result of the above steps, the consistency

of our approach is also user's responsibility and this consistency also relies on the correct implementation of the Apia program.

In this chapter, we show how to combine interactive and automatic proofs when working in FOL for reasoning about programs. In § 6.1, we describe how to combine Agda with the ATPs. In § 6.2, we show examples in various first-order theories of our combined proof approach. In § 6.3, we describe the Apia program, which translates our Agda representation of first-order formulae into TPTP and it calls the ATPs, and we present some statistics and conclusions related to the use of ATPs in program verification.

6.1 Combining Agda with Automatic Theorem Provers

We use the TPTP language as the input language for the ATPs. This language is part of the TPTP World: *"A well known and established infrastructure that supports research, development, and deployment of automated theorem proving systems for classical logics"* [Sutcliffe 2010, p. 1].

In TPTP syntax, each problem contains one or more annotated formulae of the form

$$\texttt{fof(name, role, formula)}, \tag{6.1}$$

where `name` identifies the formula within the problem, `formula` is a FOL-formula and

$$\texttt{role} \in \{\ \texttt{axiom}, \texttt{definition}, \texttt{hypothesis}, \texttt{conjecture}\ \}.$$

For TPTP, conjectures are the formulae to be proved. Moreover, axioms are accepted without proofs, as a basis for proving conjectures, hypotheses are assumed to be true for a particular problem, and they are used like axioms, and definitions are used to introduce symbols, and they are also used like axioms [Sutcliffe 2013].

In Fig. 6.1, we show the role of ATPs in our approach. We have extended the development version of Agda by adding the built-in ATP-pragma containing information to be used by the ATPs. The ATP-pragma instructs Agda to add information to an interface file which is generated by Agda after type-checking a file. In this way, we tell the ATPs, via the Apia program, that a certain formula is a conjecture, an axiom, a hypothesis, or that a certain constant is a definition.

Using the ATP-pragma, we tell the ATPs that the formulae `A`, `B` and `C` are axioms by

```
{-# ATP axiom A #-}
{-# ATP axiom B #-}
{-# ATP axiom C #-}.
```

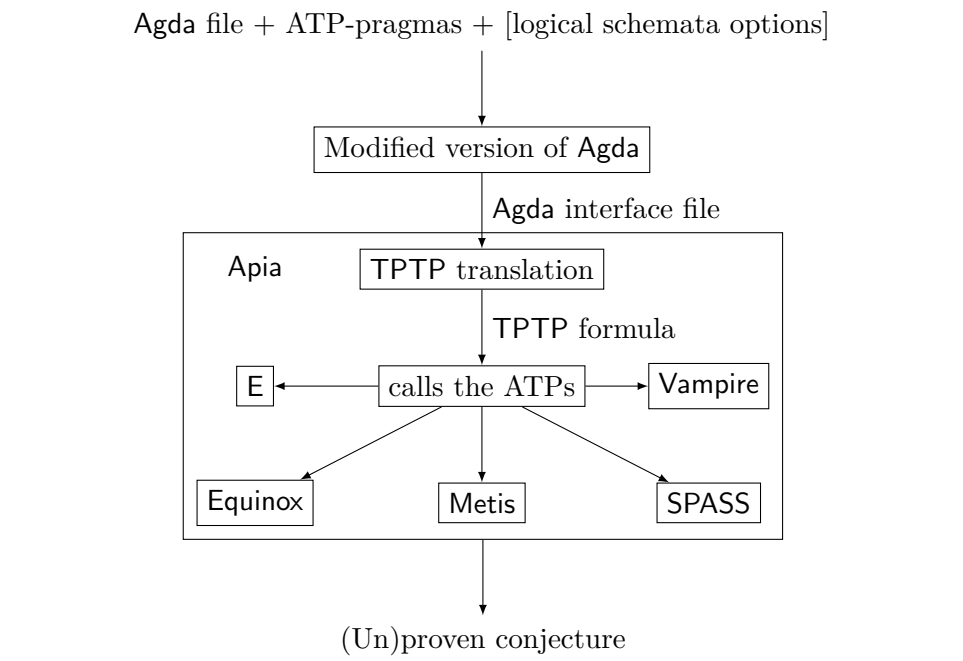


Fig. 6.1: Combining Agda with ATPs.

We can also use the alternative syntax

```
{-# ATP axiom A B C #-}.
```

As we show later, we shall tell the ATPs that inductive data type constructors, conversion rules and equations defining (recursive) functions are ATP axioms.

The ATP-pragma syntax for the TPTP hypotheses and definitions is obtained by replacing the word '`axiom`' with the words '`hint`' or '`definition`', respectively. We can also use the above alternative syntax with hypotheses and definitions.

To automatically prove a formula `A`, we shall postulate it and add the ATP-pragma

```
{-# ATP prove A #-}
```

that instructs the ATPs to prove the conjecture `A`.

After type-checking the Agda file with the conjecture(s), we run the Apia program, which first translates all axioms, hypotheses, definitions, and conjectures in the generated interface file into the TPTP language, and then tries to prove the conjectures by calling *simultaneously* the ATPs. At the

moment, the ATPs that can be used are shown in Fig. 6.1: The E theorem prover [Schulz 2013], Equinox [Claessen 2011b], Metis [Hurd 2003], SPASS [Weidenbach et al. 2009] or Vampire [Kovác and Voronkov 2013]. The user can choose which ATP(s) to use from the previous list. The user can also use the default ATPs called by the Apia program which are E, Equinox and Vampire.

For example, to automatically prove the commutativity of disjunction using the default ATPs, we first write

```
postulate
  A B    : Set
  v-comm : A ∨ B → B ∨ A
{-# ATP prove v-comm #-}
```

in an Agda file called `CommDisjunction.agda`. Next, we run the command

```
$ apia CommDisjunction.agda
```

and in the terminal, we get information about where the TPTP file corresponding to the property to be proved is located, which property is being proved, and which ATP was able to prove the property first, if any.

```
Proving the conjecture in /tmp/CommDisjunction/10-8744-comm.tptp
Vampire 0.6 (revision 903) proved the conjecture
```

In this case, the TPTP file is created in the directory `/tmp/CommDisjunction/`, the property `v-comm` is renamed `10-8744-comm`, where `10` is the line number where the property is postulated and `8744` is the decimal code point of the Unicode symbol `∨`, and Vampire was the first ATP to prove the conjecture.

If no ATP can prove a particular conjecture within four minutes (default time), the Apia program informs us about it.

If we want to only use one or more particular ATPs, we can use the `--atp` command-line option. For example, to use, say, Equinox and Metis in the previous example, we should run the following command instead:

```
$ apia --atp=equinox --atp=metis CommDisjunction.agda
```

6.2 Applying the Combined Proofs Approach

To illustrate the application of our combined proofs approach, we shall show how to prove some simples examples on different first-orders theories including FOTC. These examples do not present a major difficulty to be directly formalised in dependent type theory. In Chapter 7, we shall show more complicated examples whose formalisation in dependent type theory is not trivial or, to the best of our knowledge, not even possible.

Remark 6.1. All the automatic proofs were tested on a 2.70 GHz personal computer with six GB of RAM using a timeout of four minutes.

6.2.1 First-Order Logic

We start by showing how to use the ATPs when proving properties in the context of FOL.

Logical constants. In our combined proof approach, there is no need to tell the ATPs about the logical constants used in our Agda formalisation of FOL (see Fig. 3.1) by means of the ATP-pragma, because the ATPs implement them. It is also unnecessary to tell the ATPs about the rules of the propositional equality because the equality of the ATPs is reflexive, symmetric, transitive and satisfies the substitutivity property.

Pure first-order logic. For pure First-Order Logic—FOL without identity and where the only terms are variables [Kleene 1952]—all the theorems tested were automatically proved.

Classical logic. As mentioned in § 5.1, the ATPs in Fig. 6.1 implement classical logic. For example, we can automatically prove the principle of the excluded middle `pem` (postulated in Fig. 3.1) or the (equivalent) principle of indirect proof `¬-elim`.

```
postulate pem : ∀ {A} → A ∨ ¬ A
{-# ATP prove pem #-}

postulate ¬-elim : ∀ {A} → (¬ A → ⊥) → A
{-# ATP prove ¬-elim #-}.
```

Nonempty domain. As mentioned in remark 3.10, the domain of quantification in FOL is non-empty, which is implemented by the ATPs in Fig. 6.1. For example, Theorem 3.7 is automatically proved without the need of adding an element to the domain of quantification.

```
postulate
  A   : D → Set
  ∀→∃ : (∀ x → A x) → ∃ A
{-# ATP prove ∀→∃ #-}.
```

6.2.2 First-Order Theories

6.2.2.1 Automatic Proofs

As for the proofs of pure first-order logic, some proofs in first-order theories are completely automatic when using the ATPs. In the following example, we show this.

Example 6.2. The three theorems of group theory which we proved interactively in § 3.3.1 can now be automatically proved by the ATPs.

We start by using the ATP-pragma to inform the ATPs about the axioms of group theory defined in (3.8).

```
{-# ATP axiom assoc leftIdentity leftInverse #-}.
```

Using the ATP-pragma, we tell the ATPs to prove the theorems `rightIdentityUnique` and `leftCancellation` in Examples 3.14 and 3.15, respectively.

```
postulate rightIdentityUnique : ∀ r → (∀ a → a · r ≡ a) → r ≡ ε
{-# ATP prove rightIdentityUnique #-}

postulate leftCancellation : ∀ {a b c} → a · b ≡ a · c → b ≡ c
{-# ATP prove leftCancellation #-}.
```

For the automatic proof of the `commutatorInverse` theorem in Example 3.16, we use the ATP-pragma to inform the ATPs about the commutator of two elements of a group is an ATP definition.

```
{-# ATP definition [_,_] #-}.
```

Now, we avoid the tedious proof steps of equational reasoning required for the interactive proof of the theorem `commutatorInverse` by telling the ATPs that prove the theorem.

```
postulate commutatorInverse : ∀ a b → [ a , b ] · [ b , a ] ≡ ε
{-# ATP prove commutatorInverse #-}.
```

6.2.2.2 Combined Proofs

To prove some first-order theorems, we actually need to combine interactive and automatic reasoning. This approach is required when the proof of a theorem requires: (i) a chain of equational reasoning steps which cannot be proved by the ATPs or (ii) an instance of a higher-order logical schemata, such as the axiom of induction of Peano arithmetic, the induction principles associated to the inductive predicates or the co-induction rules of the co-inductive predicates.

In what follow, we present examples of the two cases mentioned above.

Example 6.3. Given the first-order theory of a left and right distributive binary operation, the task B of Stanovský [2008] is a fairly small example of a proof which requires a combination of interactive and automatic proof steps.

Let us consider FOL as in Fig. 3.1. For the formalisation of the first-order theory used in this example, we postulate a left-associative binary operation and two distributive axioms.

```
infixl 10 _·_
postulate
  _·_               : D → D → D
  leftDistributive  : ∀ x y z → x · (y · z) ≡ (x · y) · (x · z)
  rightDistributive : ∀ x y z → (x · y) · z ≡ (x · z) · (y · z)
{-# ATP axiom leftDistributive rightDistributive #-}.
```

The proof of the following theorem is Stanovský's task B:

```
prop₂ : ∀ u x y z →
          (x · y · (z · u)) ·
            (( x · y · ( z · u)) · (x · z · (y · u))) ≡
          x · z · (y · u).
```

The paper proof of task B requires 35 very simple equational reasoning steps. Our interactive proof of task B is almost a literal translation from Stanovský's proof. However, unlike the paper proof, our proof is considerably more tedious because we need to formalise the justification of every step in the equational reasoning (often omitted in the paper proofs).

To automatically prove the theorem `prop₂` using our approach, we first postulate it and add the ATP-pragma

```
{-# ATP prove prop₂ #-}.
```

Let `TaskB.agda` be the file which contains the postulated theorem `prop₂` and the above ATP-pragma. Running the command

```
$ apia --atp=e --atp=equinox --atp=metis --atp=spass --atp=vampire \
TaskB.agda
```

in the terminal we get the following information:

```
Proving the conjecture in /tmp/TaskB/16-prop2.tptp
Vampire 0.6 (revision 903) *did not* prove the conjecture
SPASS *did not* prove the conjecture
Equinox, version 5.0alpha, 2010-06-29 *did not* prove the conjecture
metis 2.3 (release 20120927) *did not* prove the conjecture
E 1.8-001 Gopaldhara Jun Chiabari *did not* prove the conjecture
apia: the ATP(s) did not prove the conjecture in /TaskB/16-prop2.tptp
```

That is, none of the ATPs in Fig. 6.1 could prove the property.

Instead, we combine interactive and automatic proof steps in order to prove the theorem. Using this approach, we could reduce the 35 original proof steps to 9 proof steps where each required justification was automatically proved by the ATPs.

```
prop₂ : ∀ u x y z →
          (x · y · (z · u)) ·
```

```
           (( x · y · ( z · u)) · (x · z · (y · u))) ≡
         x · z · (y · u)
prop₂ u x y z =
  xy·zu · (xy·zu · xz·yu)
    ≡⟨ j₁₋₅ ⟩
  xy·zu · (xz · xu·yu · (y·zu · xz·yu))
    ≡⟨ j₅₋₉ ⟩
  xy·zu · (xz · xyu · (yxz · yu))
    ≡⟨ j₉₋₁₄ ⟩
  xz · xyu · (yz · xyu) · (xz · xyu · (y·xu · z·yu))
    ≡⟨ j₁₄₋₂₀ ⟩
  xz · xyu · (y·xu · (y·yu · z·yu) · (z · xu·yu · (y·xu · z·yu)))
    ≡⟨ j₂₀₋₂₃ ⟩
  xz · xyu · (y · xu·zu · (z · xu·yu · (y·xu · z·yu)))
    ≡⟨ j₂₃₋₂₅ ⟩
  (xz · xyu) · (y · xu·zu · (z·xu · y·xu · z·yu))
    ≡⟨ j₂₅₋₃₀ ⟩
  xz · xyu · (y·zy · xzu)
    ≡⟨ j₃₀₋₃₅ ⟩
  xz·yu ∎
```

In the above proof, we omitted the postulated justifications $\texttt{j}_{1-5}$, ..., $\texttt{j}_{30-35}$, which were automatically proved, and their ATP-pragmas. Moreover, for any `u`, `x`, `y` and `z`, we use the following definitions:

```
xy xyz x·yz ux·yz : D
xy    = x · y
xyz   = x · y · z
x·yz  = x · (y · z)
ux·yz = u · x · (y · z)
{-# ATP definition xy xyz x·yz ux·yz #-}.
```

In the following example, we illustrate the use of combined interactive and automatic reasoning when the proof requires the instantiation of an induction principle.

Example 6.4. To automate the proof of the commutativity of addition in PA, we start by using the ATP-pragma to inform the ATPs about the axioms $\texttt{PA}_1$ to $\texttt{PA}_6$ defined in (3.9). Note that we do not inform the ATPs about the induction principle `ℕ-ind` defined in (3.10) because the ATPs do not handle inductive proofs. In our combined approach, we interactively use the induction principle `ℕ-ind`, properly instantiated for the property we want to prove, and we use the ATPs to automatically prove the required base and step cases of the induction.

```
+-comm : ∀ m n → m + n ≡ n + m
+-comm m n = ℕ-ind A A0 is m
  where
  A : M → Set
  A i = i + n ≡ n + i
  {-# ATP definition A #-}

  postulate A0 : A zero
  {-# ATP prove A0 +-rightIdentity #-}

  postulate is : ∀ i → A i → A (succ i)
  {-# ATP prove is x+Sy≡S[x+y] #-}.
```

The proof requires the auxiliary properties

```
+-rightIdentity : ∀ n → n + zero ≡ n
x+Sy≡S[x+y]     : ∀ m n → m + succ n ≡ succ (m + n)
```

in order to prove the base and step cases, respectively. These properties are used as local hypotheses in the ATP-pragmas by giving their names after the name of the conjecture to be proved.

Instead of using properties as local hypotheses to the conjectures, we can use them as global hypotheses. In this way, we avoid to explicitly pass these hypotheses to every conjecture that requires them. In the following example, we illustrate this.

Example 6.5. For the combined proof of the commutativity of addition in Example 6.4, we tell the ATPs that the properties `+-rightIdentity` and `x+Sy≡S[x+y]` are ATP hypotheses by

```
{-# ATP hint +-rightIdentity x+Sy≡S[x+y] #-}
```

and then we can rewrite the ATP-pragmas associated to the conjectures `A0` and `is` by

```
{-# ATP prove A0 #-}
{-# ATP prove is #-}.
```

Since in the TPTP semantics the global hypotheses are used as axioms, and since the ATPs are very sensitive to the number of axioms in a TPTP problem, we do not use global hypotheses in our formalisation of first-order theories to avoid unnecessarily blowing up the ATPs' search space.

6.2.3 First-Order Theory of Combinators

6.2.3.1 Automatic Proofs for the Conversion Rules

We start by telling the ATPs that the conversion and discrimination rules associated with the FOTC-terms

`true`, `false`, `if`, `zero`, `succ`, `pred` and `iszero`

are ATP axioms.

However, for reasons that will be explained in § 6.3.1, it is more convenient for the ATPs to formalise the conversion and discrimination rules for the above terms using the following terms and function symbols instead (see convention 4.10):

`true`, `false`, `if_then_else_`, `zero`, `succ₁`, `pred₁` and `iszero₁`.

We use then the following conversion and discrimination rules:

```
postulate
  if-true  : ∀ t {t'} → if true then t else t' ≡ t
  if-false : ∀ {t} t' → if false then t else t' ≡ t'
{-# ATP axiom if-true if-false #-}

postulate
  pred-0 : pred₁ zero ≡ zero
  pred-S : ∀ n → pred₁ (succ₁ n) ≡ n
{-# ATP axiom pred-0 pred-S #-}

postulate
  iszero-0 : iszero₁ zero ≡ true
  iszero-S : ∀ n → iszero₁ (succ₁ n) ≡ false
{-# ATP axiom iszero-0 iszero-S #-}

postulate
  t≠f : true ≠ false
  0≠S : ∀ {n} → zero ≠ succ₁ n
{-# ATP axiom t≠f 0≠S #-}.
```

Our combined proof approach allows completely automatic proofs for the conversion rules of the function symbols added by using recursive equations in FOTC. For each new function symbol added by a recursive equation of the form (5.1), we inform the ATPs that the equation is an ATP axiom. An important consequence of this is that the conversion rules for the new function symbol can be automatically proved.

In what follows, we show a couple of examples where the conversion rules associated to a new symbol are automatically proved.

Example 6.6. After telling the ATPs that the equation `+-eq` defined in Example 5.3 is an ATP axiom, the conversion rules for addition are automatically proved.

```
postulate +-0x : ∀ n → zero + n ≡ n
{-# ATP prove +-0x #-}

postulate +-Sx : ∀ m n → succ₁ m + n ≡ succ₁ (m + n)
{-# ATP prove +-Sx #-}.
```

Example 6.7. In Example 6.6, we use automatic proofs for proofs with very few equational reasoning steps. However, by telling the ATPs that the equation `gcd-eq` defined in Example 5.5 is an ATP axiom, the automatic proofs of the conversion rules for the gcd algorithm avoid tedious interactive proofs.

```
postulate
  gcd-00  : gcd zero zero ≡ zero

  gcd-S0  : ∀ n → gcd (succ₁ n) zero ≡ succ₁ n

  gcd-0S  : ∀ n → gcd zero (succ₁ n) ≡ succ₁ n

  gcd-S>S : ∀ m n → succ₁ m > succ₁ n →
              gcd (succ₁ m) (succ₁ n) ≡
              gcd (succ₁ m ∸ succ₁ n) (succ₁ n)

  gcd-S≯S : ∀ m n → succ₁ m ≯ succ₁ n →
              gcd (succ₁ m) (succ₁ n) ≡
              gcd (succ₁ m) (succ₁ n ∸ succ₁ m)

{-# ATP prove gcd-00 gcd-S0 gcd-0S gcd-S>S gcd-S≯S #-}.
```

Finally, when a new function symbol is added by a set of conversion rules, we tell the ATPs that they are axioms. In the following example, we illustrate this.

Example 6.8. Given the definitions of addition of natural numbers (see Example 5.4) and of the concatenation of lists (see Example 5.16), we tell the ATPs that they are axioms.

```
{-# ATP axiom +-0x +-Sx #-}
{-# ATP axiom length-[] length-∷ #-}.
```

6.2.3.2 Combined Inductive Proofs

The inductively defined predicates of FOTC are formalised as inductive families using Agda's **data** constructor and, as already mentioned, we instantiate their induction principles using Agda's pattern matching. In our combined proof approach for inductive proofs, we first inform the ATPs that the inductive data type constructors of the inductive predicates, that is, their introduction rules, are ATP axioms. In the following example, we illustrate this.

Example 6.9. We tell the ATPs that the inductive data type constructors `nzero` and `nsucc` of the inductive predicate `N` for total and finite natural numbers are ATP axioms as follows:

```
data N : D → Set where
  nzero : N zero
  nsucc : ∀ {n} → N n → N (succ₁ n)
{-# ATP axiom nzero nsucc #-}.
```

In our combined proof approach, we use in general the following methodology for inductive proofs: (i) we instruct Agda to do pattern matching on the argument(s) that satisfy the inductive predicate and (ii) we try to automatically prove the base and step cases required by the induction principle.

In the following example, we show a combined inductive proof using the above methodology.

Example 6.10. The proof that the addition of total and finite natural numbers returns a total and finite natural number (interactively proved in Example 5.3) using our combined approach is given by

```
+-N : ∀ {m n} → N m → N n → N (m + n)

+-N {n = n} nzero Nn = prf
  where postulate prf : N (zero + n)
        {-# ATP prove prf #-}

+-N {n = n} (nsucc {m} Nm) Nn = prf (+-N Nm Nn)
  where postulate prf : N (m + n) → N (succ₁ m + n)
        {-# ATP prove prf #-}.
```

Here, we do pattern matching on the first explicit argument of `+-N`, and we automatically prove the base and step cases required by the proof.

Given that in our representation of higher-order functions in FOTC we stay *strictly* within FOL, we can also use our combined proof approach to prove some properties of such functions.

In the following example, we show a combined proof using the first-order version of the `map` function.

Example 6.11. To prove the `map-++` property (interactively proved in Example 5.17), we first tell the ATPs that the inductive data type constructors `lnil` and `lcons` of the inductive predicate `List` for total and finite lists (see Example 5.14), the conversions rules `++-[]` and `++-∷` for the concatenation of lists (see Example 5.16), and the conversion rules `map-[]` and `map-∷` for the `map` function (see Example 5.17) are ATP axioms.

To prove the property, we do pattern matching on its second explicit argument, and then we automatically prove the base and step cases required by the proof.

```
map-++ : ∀ f {xs} → List xs → ∀ ys →
         map f (xs ++ ys) ≡ map f xs ++ map f ys

map-++ f lnil ys = prf
  where postulate prf : map f ([] ++ ys) ≡ map f [] ++ map f ys
        {-# ATP prove prf #-}

map-++ f (lcons x {xs} Lxs) ys = prf (map-++ f Lxs ys)
  where
  postulate
    prf : map f (xs ++ ys) ≡ map f xs ++ map f ys →
          map f ((x ∷ xs) ++ ys) ≡ map f (x ∷ xs) ++ map f ys
  {-# ATP prove prf #-}
```

6.2.3.3 Combined Co-Inductive Proofs

The co-inductively defined predicates of FOTC are formalised by postulating their unfolding and co-induction rules. In our combined proof approach for co-inductive proofs, we first inform the ATPs that the unfolding rule of the co-inductive predicate is an ATP axiom. In the following example, we illustrate this.

Example 6.12. We tell the ATPs that the unfolding rule `Stream-out` of the co-inductive predicate `Stream` for total and potentially infinite list of elements (see Example 5.28) is an ATP axiom as follows:

```
postulate
  Stream-out : ∀ {xs} → Stream xs →
               ∃[ x' ] ∃[ xs' ] xs ≡ x' ∷ xs' ∧ Stream xs'
{-# ATP axiom Stream-out #-}.
```

In our combined proof approach, we use in general the following methodology for co-inductive proofs: (i) we interactively instantiate the co-induction rule and (ii) we automatically prove the hypotheses required by the co-induction rule.

In the following example, we show a combined co-inductive proof which only requires the step (i) of the above methodology.

Example 6.13. To prove that if `x ∷ xs` is stream, then `xs` is also a stream (interactively proved in Example 5.29), it is only required to inform the ATPs that the unfolding rule `Stream-out` is an ATP axiom.

```
postulate ∷-Stream : ∀ {x xs} → Stream (x ∷ xs) → Stream xs
{-# ATP prove ∷-Stream #-}.
```

In the following two examples, we show two combined co-inductive proofs which require both steps of the above methodology.

Example 6.14. To prove that the number ∞ is a total and possibly infinite natural number (interactively proved in Example 5.24), we interactively instantiate the co-induction rule `Conat-coind` with the predicate `λ n → n ≡ ∞` using an ATP definition, and we automatically prove the required hypothesis.

```
∞-Conat : Conat ∞
∞-Conat = Conat-coind A h refl
  where
  A : D → Set
  A n = n ≡ ∞
  {-# ATP definition A #-}

  postulate
    h : ∀ {n} → A n → n ≡ zero ∨ (∃[ n' ] n ≡ succ₁ n' ∧ A n')
  {-# ATP prove h #-}.
```

Example 6.15. To prove that the length of a stream is ∞ (interactively proved in Example 5.31), we interactively instance the co-induction rule `≈N-coind` with the relation `R`. Then, we tell the ATPs that this relation is an ATP definition, and we automatically prove the two hypotheses required by the co-induction principle `≈N-coind`.

```
streamLength : ∀ {xs} → Stream xs → length xs ≈N ∞
streamLength {xs} Sxs = ≈N-coind R h₁ h₂
  where
  R : D → D → Set
  R m n = ∃[ xs ] m ≡ length xs ∧ n ≡ ∞ ∧ Stream xs
  {-# ATP definition R #-}

  postulate
    h₁ : ∀ {m n} → R m n →
         m ≡ zero ∧ n ≡ zero
           ∨ (∃[ m' ] ∃[ n' ]
                m ≡ succ₁ m' ∧ n ≡ succ₁ n' ∧ R m' n')
  {-# ATP prove h₁ #-}
```

```
  postulate h₂ : R (length xs) ∞
  {-# ATP prove h₂ #-}.
```

As highlighted in Section 5.5, we can mix inductive and co-inductive notions in FOTC. This feature is also present when we combine interactive and automatic proofs. In the following example, we show this.

Example 6.16. To prove that every total and finite natural number is a total and possibly infinite natural number, we interactively instance the co-induction rule `Conat-coind` with the inductive predicate `N` for total and finite natural numbers. To prove the hypothesis required by `Conat-coind`, we do pattern matching on a proof that the natural number is total and finite, and we automatically prove the base and step cases require by the proof.

```
N→Conat : ∀ {n} → N n → Conat n
N→Conat Nn = Conat-coind N h Nn
  where
  h : ∀ {m} → N m → m ≡ zero ∨ (∃[ m' ] m ≡ succ₁ m' ∧ N m')
  h nzero = prf
    where
    postulate prf : zero ≡ zero ∨ (∃[ m' ] zero ≡ succ₁ m' ∧ N m')
    {-# ATP prove prf #-}
  h (nsucc {m} Nm) = prf
    where
    postulate
       h : succ₁ m ≡ zero ∨ (∃[ m' ] succ₁ m ≡ succ₁ m' ∧ N m')
    {-# ATP prove h #-}.
```

The `map-iterate` property is a common example to illustrate the use of co-induction (see, for example, Gordon [1995] and Gibbons and Hutton [2005]). In the following example, we show a proof of this property in FOTC.

Example 6.17. For our formalisation of the `map-iterate` property in FOTC, we use the `map` function defined in Example 5.17 and we define the `iterate` function by the following recursive equation:

```
postulate
  iterate    : D → D → D
  iterate-eq : ∀ f x → iterate f x ≡ x ∷ iterate f (f · x)
{-# ATP axiom iterate-eq #-}.
```

The `map-iterate` property asserts that the potentially infinite lists `map f (iterate f x)` and `iterate f (f · x)` are equals.

Given the bisimulation relation represented by the formula

$$\begin{aligned}&\Psi(X, xs, ys) \stackrel{\text{def}}{=}\\ &\exists x'\, xs'\, ys'.\, xs = \mathsf{cons} \cdot x' \cdot xs' \wedge ys = \mathsf{cons} \cdot x' \cdot ys' \wedge X(xs', ys'),\end{aligned} \tag{6.2}$$

the equality between total and potentially infinite lists (streams) is defined as the greatest fixed-point of (6.2), which we implemented by the co-inductive bisimilarity relation (see, for example, Dybjer and Sander [1989] and Turner [1995]) denoted by `_≈_`.

```
postulate
  _≈_ : D → D → Set

  ≈-out:
    ∀ {xs ys} → xs ≈ ys →
    ∃[ x' ] ∃[ xs' ] ∃[ ys' ]
      xs ≡ x' ∷ xs' ∧ ys ≡ x' ∷ ys' ∧ xs' ≈ ys'

  ≈-coind :
    (B : D → D → Set) →
    (∀ {xs ys} → B xs ys →
      ∃[ x' ] ∃[ xs' ] ∃[ ys' ]
        xs ≡ x' ∷ xs' ∧ ys ≡ x' ∷ ys' ∧ B xs' ys') →
    ∀ {xs ys} → B xs ys → xs ≈ ys
{-# ATP axiom ≈-out #-}.
```

To prove the `map-iterate` property, we use the `≈-coind` rule on a particular bisimulation `B` [Giménez and Casterán 2007], and the hypotheses required by `≈-coind` are automatically proved by the ATPs.

```
≈-map-iterate : ∀ f x → map f (iterate f x) ≈ iterate f (f · x)
≈-map-iterate f x = ≈-coind B h₁ h₂
  where
  B : D → D → Set
  B xs ys =
    ∃[ y ] xs ≡ map f (iterate f y) ∧ ys ≡ iterate f (f · y)
  {-# ATP definition B #-}

  postulate
    h₁ : ∀ {xs ys} → B xs ys → ∃[ x' ] ∃[ xs' ] ∃[ ys' ]
           xs ≡ x' ∷ xs' ∧ ys ≡ x' ∷ ys' ∧ B xs' ys'
  {-# ATP prove h₁ #-}

  postulate h₂ : B (map f (iterate f x)) (iterate f (f · x))
  {-# ATP prove h₂ #-}.
```

6.3 The Apia Program

The Apia program implements the translation from our Agda representation of first-order formulae into the TPTP language, that is, the Apia program

implements the inverse translation from first-order formulae into Agda described in (3.3) using the TPTP language as the target language. In addition, the Apia program calls the ATPs on the above translation.

6.3.1 Translation of Agda Types into TPTP

The source of our translation to TPTP is the Agda internal syntax (3.1) read from an Agda interface file, as illustrated in Fig. 6.1. Our translation only uses a subset of the First-Order Form part of the TPTP language represented by (6.1). For a complete description of the First-Order Form and other parts of the TPTP language, see Sutcliffe [2009, 2010, 2013].

Notation 6.18. In TPTP syntax, variables start with upper case letters, and constant, predicate and function symbols start with lower case letters.

Nullary predicates. Nullary predicates in FOL represent propositional symbols (also called propositional letters), that is, atomic units that represent concrete propositions (see, for example, van Dalen [2004]). For example, the translation of the propositional symbol

```
postulate A : Set
```

into TPTP is the nullary predicate `a`.

Non-nullary predicates. Let `P : D → Set` be a unary predicate. The translation of `P t`, where `t : D`, into TPTP is

$$\texttt{p(s)},$$

where `s` is the translation of the term `t`.

Let n be a positive integer. The most direct way to translate an n-ary predicate

$$\texttt{P t}_1 \cdots \texttt{t}_n \texttt{ : Set}, \quad \text{with } \texttt{t}_i \texttt{ : D}, \tag{6.3}$$

into TPTP is

$$\texttt{p(s}_1\texttt{,...,s}_n\texttt{)}, \tag{6.4}$$

where $\texttt{s}_i$ corresponds to the translation of the term $\texttt{t}_i$. However, we shall follow Claessen's [2011a] suggestion for the translation of predicates into TPTP. As we shall explain below, Claessen's approach facilitates the translation of universal quantified propositional functions.

For example, for the translation of a unary predicate

$$\texttt{P t : Set}, \quad \text{with } \texttt{t : D}, \tag{6.5}$$

we introduce a constant `p` and a binary predicate symbol $\texttt{appP}_1$, so

$$\texttt{appP}_1\texttt{(p,s)}$$

represents the predicate (6.5), where `s` is the translation of the term `t`. In general, for the translation of an n-ary predicate (6.3), we introduce a constant `p` and an $(n+1)$-ary predicate symbol $\texttt{appP}_n$, so

$$\texttt{appP}_n\texttt{(p,s}_1\texttt{,...,s}_n\texttt{)} \tag{6.6}$$

represents the predicate (6.3), provided $\texttt{s}_i$ is the translation of $\texttt{t}_i$.

Remark 6.19. The `--without-predicate-constants` command-line option can be used to translate n-ary predicates as in (6.4) instead of as in (6.6).

Function terms. Officially, FOTC is a first-order theory with `_·_` as the binary application symbol, and one constant for each function (see Example 5.2). However, as we mentioned in conventions 5.1 and 5.13, we sometimes introduce n-ary function symbols in our implementation of a FOTC theory. Recall for example the two versions of the constructor `nsucc` defined in (4.10) and (4.12), respectively,

```
nsucc : ∀ {n} → N n → N (succ · n)          (6.7)
nsucc : ∀ {n} → N n → N (succ₁ n).          (6.8)
```

When translating the above types to TPTP, the function value `succ · n` in (6.7) will be translated to

```
app(succ,N),                                (6.9)
```

where `app` represents the binary application `_·_`, while the function value $\texttt{succ}_1$ `n` in (6.8), will be translated to

$$\texttt{succ}_1\texttt{(N).} \tag{6.10}$$

Given that (6.10) instead of (6.9) improves the performance of the ATPs [Claessen 2010a], whenever possible, we use n-ary functions, such as $\texttt{succ}_1$, instead of constant functions, such as `succ`, when working in FOTC.

In general, for the translation of a function value

$$\texttt{f t}_1 \cdots \texttt{t}_n \texttt{ : D}, \quad \text{with } \texttt{t}_i \texttt{ : D}, \tag{6.11}$$

we introduce a constant `f`, so

$$\texttt{f(s}_1\texttt{,...,s}_n\texttt{)}$$

represents the function value (6.11), where $\texttt{s}_i$ corresponds to the translation of the term $\texttt{t}_i$.

Remark 6.20. In the theories used by Abel, Coquand and Norell [2005] and Meng and Paulson [2008], function values can be expressed by currying—applying a function to fewer than the maximum numbers of arguments. Therefore, these authors use a binary function symbol `@` for the translation of function values into TPTP. For example, for the translation of

$$\texttt{f t} \text{ and } \texttt{g t}_1\ \texttt{t}_2,$$

they introduce constants `f` and `g`, so

$$\texttt{@(f,s)} \text{ and } \texttt{@(@(g,s}_1\texttt{),s}_2\texttt{)}$$

represent the function values `f t` and `g t₁ t₂`, respectively, provided `s` represents `t` and `sᵢ` represent `tᵢ`. In general, for the translation of (6.11), they introduce a constant `f`, so

$$\texttt{@(... @(@(f,s}_1\texttt{),s}_2\texttt{),...,s}_n\texttt{)}$$

represents the function value (6.11).

The `--with-function-constants` command-line option performs this kind of translation to allow experimenting with theories where function values can be expressed by currying.

Propositional functions. In our implementation of FOTC, an n-ary propositional function

$$\texttt{A x}_1 \cdots \texttt{x}_n\ \texttt{:}\ \textbf{Set}, \quad \text{with } \texttt{x}_i\ \texttt{:}\ \texttt{D} \tag{6.12}$$

has the same type as an n-ary predicate (6.3), that is, it has the type

$$\underbrace{\texttt{D} \to \cdots \to \texttt{D}}_{n} \to \textbf{Set}.$$

Hence, we use the same approach for translating n-ary predicates as for translating propositional functions. Therefore, the translation of (6.12) into TPTP is given by

$$\texttt{appP}_n\texttt{(a,X}_1\texttt{,...,X}_n\texttt{)}, \tag{6.13}$$

where `a` is a constant, `appPₙ` is an $(n+1)$-ary predicate symbol, and `Xᵢ` corresponds to the translation of the variable `xᵢ`.

No logical schemata in TPTP. As mentioned in remark 3.4, it is easy to represent logical schemata in Agda. Given that TPTP only has syntax for single formulae, not schematic ones, we have added two pragma options to our modified version of Agda to automatically prove theorems with schematic propositional symbols and schematic propositional functions.

- Schematic propositional symbols

  While, for example, in

```
postulate
  A B     : Set
  v-comm₁ : A ∨ B → B ∨ A
{-# ATP prove v-comm₁ #-}
```

the formula `v-comm₁` corresponds to an *instance* of the commutativity of disjunction, in

```
postulate v-comm₂ : {A B : Set} → A ∨ B → B ∨ A
{-# ATP prove v-comm₂ #-}
```

the formula `v-comm₂` corresponds to the *schema* for the commutativity of disjunction

$$\mathcal{A} \vee \mathcal{B} \supset \mathcal{A} \vee \mathcal{B},$$

where $\mathcal{A}$ and $\mathcal{B}$ range over arbitrary well-formed formulae.

For the translation of schematic formulae like `v-comm₂` into TPTP, we have added the

```
--schematic-propositional-symbols
```

Agda pragma option. By using the above pragma in the correspondent Agda file, the Apia program translates the property `v-comm₂` into TPTP by erasing the quantification on the propositional symbols `A B : Set` and representing them with new nullary predicate symbols `a` and `b`, respectively.

- Schematic propositional functions

  For the translation of schematic propositional functions, such as the `A` in

  ```
  postulate thm : (A : D → Set) → ∀ {x y} → A x ∨ A y → A y ∨ A x
  {-# ATP prove thm #-}
  ```

  we have added the

  ```
  --schematic-propositional-functions
  ```

  Agda pragma option. This pragma tells the Apia program to replace the constant `a` in (6.13), associated with the translation of the propositional function `A : D → Set`, by a universal quantified variable.

  As previously mentioned, since n-ary propositional functions and predicates have the same type, the above pragma can also be used for the translation of schematic predicates.

6.3.2 Translation of Functions and Propositional Functions into TPTP

So far, we have only translated Agda types that represent first-order terms or formulae. In our implementation of first-order theories, we also need to translate function definitions (see Examples 6.2 and 6.3) and propositional function definitions (see Examples 6.4, 6.14, 6.15 and 6.17) into TPTP.

Since our domain universe is postulated (see § 4.3), the definition of functions and propositional functions will only have an equation.

Let `f` be a unary function

```
f : D → D
f x = RHS
{-# ATP definition f #-}
```

and `A` a unary propositional function

```
A : D → Set
A x = RHS
{-# ATP definition A #-}.
```

The translations of both the function `f` and the propositional function `A` into TPTP are given, respectively, by

```
! [X] : f(X) = RHS_t,
```

```
! [X] : a(X) <=> RHS_t,
```

where `! [X]` and `<=>` represent the universal quantification on the variable `X` and the bi-conditional in TPTP syntax, respectively, and `RHS_t` represents the translation of `RHS`.

The generalisation to n-ary functions and propositional functions is not complicated.

6.3.3 Implementation

Although we shall not describe the full implementation of the Apia program, here we shall describe some aspects related to it.

Using Agda as a Haskell library. The Agda system is both a Haskell program and a Haskell library. Following Norell's suggestion [Norell 2007a], the Apia program was implemented in Haskell and it uses Agda as a Haskell library.

By using Agda as a Haskell library, we have access to many functions that otherwise would have been necessary to implement; for example, the functions related to processing the Agda interface files. Unfortunately, the library's API is not stable, as stated in the description of the current version of Agda:[1] *"Note that the Agda library does not follow the package versioning policy, because it is not intended to be used by third-party packages"*. However, we have chosen to rely on the current development of Agda because the Agda team frequently solves performance issues, adds new features convenient for our formalisation, fix known issues, and so on. Given the constant modification of Agda and the instability of the library's API keeping the Apia program and our modified version of Agda updated is a time-consuming task.

[1] See http://hackage.haskell.org/package/Agda-2.4.0.2.

Logical symbols. The logical symbols in Fig. 3.1, that is, ⊥ (falsehood), ∧ (conjunction), _≡_ (propositional equality), and so on, are *hard-coded* in the implementation of the Apia program; therefore, they must be used for implementing the logical constants.

Agda **η-contraction.** Agda performs η-contraction in the internal representation of their types. For example, the Agda internal representation of the following types are the same

```
t  : ∀ d → ∃[ e ] d ≡ e
t' : ∀ d → ∃ (_≡_ d).
```

Since there is no notion of η-contraction in first-order theories, the Apia program performs an η-expansion on the Agda internal types before their translation to TPTP.

Erasing proof terms. Since there is no notion of proof term in FOL, it is necessary to erase the proof terms when translating the Agda types into TPTP.

For example, the following versions of the constructor `nsucc` of the inductive predicate `N` defined in (4.12)

```
nsucc  : ∀ {n} → N n → N (succ₁ n)
nsucc' : ∀ {n} → (Nn : N n) → N (succ₁ n)
```

are valid from Agda's point of view. However, from FOL's point of view, the type of `nsucc'` is invalid because the proof term `Nn` is not a FOL concept. Therefore, our translation of `nsucc` and `nsucc'` are the same, that is, we erase the proof term `Nn` when translating the type of `nsucc'` into TPTP.

Where clauses. The definitions and the type declarations inside a **`where`** clause inherit the arguments of function they belong to as proof terms of the corresponding types. Therefore, it is also necessary to erase the proof terms in the internal representation of Agda types and definitions inside a **`where`** clause when they are translated into TPTP.

For example, in FOTC, for the combined proof (using the induction principle for `N`)

```
+-rightIdentity : ∀ {n} → N n → n + zero ≡ n
+-rightIdentity Nn = N-ind A A0 is Nn
  where
  A : D → Set
  A i = i + zero ≡ i
  {-# ATP definition A #-}

  postulate A0 : A zero
  {-# ATP prove A0 #-}
```

```
postulate is : ∀ {i} → A i → A (succ₁ i)
{-# ATP prove is #-}
```

the Agda internal representation of `A` has the same proof term `Nn` as the internal representation of

```
A' : ∀ {n} → (Nn : N n) → D → Set
A' Nn i = i + zero ≡ i
```

and the Agda internal types of `A0` and `is` have the same proof term `Nn` that have the Agda internal types of

```
postulate
  A0' : ∀ {n} → (Nn : N n) → A' Nn zero
  is' : ∀ {n} → (Nn : N n) → ∀ {i} → A' Nn i → A' Nn (succ₁ i).
```

Therefore, for the reason explained before, in the translation of `A`, `A0` and `is` into TPTP, it is necessary to erase the proof term `Nn`.

One TPTP file for each ATP conjecture. To facilitate the interchange of our generated TPTP files and their testing using the web interface to ATPs from the TPTP World,[2] the Apia program only generates one TPTP file with all the required information for each ATP conjecture.

Running the program with the `--only-files` command-line option, the TPTP files associated to the ATP conjectures are generated without calling the ATPs.

Imported ATP-pragmas. The use of imported modules is very common in Agda. For example, all the Agda modules related to the verification of programs in FOTC import the module `FOTC.Base`, which contains the conversion and discrimination rules of FOTC (see § 5.1).

For each ATP conjecture, the Apia program search in conjecture's top level module and their imported modules all the required information (axioms, definitions, and global and local hypotheses) for the conjecture.

Parallel ATPs invocation. The Apia program calls the ATPs chosen by the user in parallel. Let's suppose the i-th ATP was the first ATP to finish. If this ATP could prove the conjecture, the program kills the running processes for the other ATPs and reports that the conjecture was proved by the i-th ATP. If not, the processes for the remaining ATPs continue running. In the end, if no ATP could prove the conjecture in a given time, the Apia program reports it.

Using other automatic theorems provers. Although all the ATPs in Fig. 6.1 support the TPTP language, their functionality (command-line options, output, and so on) differs. As a consequence, the invocation of the ATPs is *hard-coded* in the Apia program. However, it should not be too difficult to adapt the program to other ATPs supporting the TPTP language.

[2] http://www.cs.miami.edu/~tptp/cgi-bin/SystemOnTPTP.

6.3.4 The Automatic Theorem Provers

The overall performance of the ATPs in our formalisation of first-order theories is quite satisfactory. Table 6.1 shows the total number of proven and unproven theorems using the indicated version of the ATPs for the following first-order theories: pure first-order logic, the theory of Example 6.3, group theory, Peano arithmetic and FOTC.

ATP (total theorems: 855)	**Proven theorems**	**Unproven theorems**	**% Success**
E 1.8-001 Gopaldhara	828	27	97%
Vampire 0.6 (revision 903)	828	27	97%
Equinox 5.0 alpha (2010-06-29)	775	80	91%
SPASS 3.7	755	100	88%
Metis 2.3 (release 2012-09-27)	588	267	69%

Table 6.1: Proven/unproven theorems by the ATPs.

Remark 6.21. We implemented a large set of interactive proofs in FOTC as part of our experiments, and in neither of those proofs we needed to use classical logic. In principle, by using ATPs for *intuitionistic* FOL, we can avoid using classical logic in FOTC. Unfortunately, the community of ATPs for intuitionistic logic is small and we did not obtain good results in our experiments. For example, ileanCoP v1.3beta1 [Otten 2005]—which is the only ATP for intuitionistic logic that we know supports the TPTP language—proved only 40% of the theorems in FOTC.

From our experiments in combining interactive and automatic proofs in our Agda formalisation of FOTC (and others first-order theories too), we can conclude that the ATPs we use are complementary, that is, where one ATP succeed, other ATPs fail, and the other way around. For this reason, the parallel ATPs invocation is an important feature. We have also realised that the ATPs are very sensitive to the information contained in a TPTP problem, that is, minor changes to the number of, the order or even the name of the formulae involved in an automatic proof, can convert a proven/unproven theorem into an unproven/proven one. This happens due to arbitrary choices made by all the ATPs [Claessen 2010b].

Chapter 7

Verification of Lazy Functional Programs

We have already shown how to use our representation of FOTC and our computer-assisted approach of combining interactive and automatic proofs when dealing with simple general recursive programs such as the `gcd` algorithm or with guarded co-recursive functions such as the `map-iterate` property. Here, we illustrate that our approach allows the verification of mainstream lazy functional programs including those that use *nested* recursive functions (see § 7.1), *higher-order* recursive functions (see § 7.2), functions *without* a proof of termination (see § 7.3) or *unguarded* co-recursive functions (see § 7.4). In § 7.5, we show some figures related to the number of theorems automatically proved in the previous examples.

It is worth mentioning that none of the examples shown here can be directly formalised in Agda or Coq because they do not pass the termination checker.

Notation 7.1. In the examples that we shall show, we use natural literal numbers (0, 1, 2, ...) to simplify the reading; they stand for natural numbers generated by `zero` and `succ₁`.

7.1 McCarthy's 91-function: A Nested Recursive Function

In our first example, we work with McCarthy's 91-function [Manna and McCarthy 1969] whose Haskell definition is

```
f91 :: Nat → Nat
f91 n = if n > 100 then n - 10 else f91 (f91 (n + 11)).
```

McCarthy's 91-function is formalised in FOTC by the following axiom:

```
postulate
  f₉₁    : D → D
  f₉₁-eq : ∀ n → f₉₁ n ≡ if (gt n 100)
                         then n ∸ 10
                         else f₉₁ (f₉₁ (n + 11))
{-# ATP axiom f₉₁-eq #-}.
```

We shall prove some properties of the function `f₉₁`.

First property. For any `n`, if `n > 100`, then `f₉₁ n ≡ n ∸ 10`.

This property is automatically proved by the ATPs.

```
postulate f₉₁-x>100 : ∀ n → n > 100 → f₉₁ n ≡ n ∸ 10
{-# ATP prove f₉₁-x>100 #-}.
```

Second property. For all total and finite natural numbers `n`, if `n ≯ 100`, then `f₉₁ n ≡ 91`.

The property is formalised by

```
f₉₁-x≯100 : ∀ {n} → N n → n ≯ 100 → f₉₁ n ≡ 91.
```

The proof is done using our combined approach with the well-founded induction principle

```
◁-wfind : (A : D → Set) →
          (∀ {n} → N n → (∀ {m} → N m → m ◁ n → A m) → A n) →
          ∀ {n} → N n → A n
```

associated with the well-founded relation

```
_◁_ : D → D → Set
m ◁ n = 101 ∸ m < 101 ∸ n
{-# ATP definition _◁_ #-}.
```

Most of the auxiliary properties are proved with the help of the ATPs. We show only a few of them here.

First, we prove that `f₉₁ 100 ≡ 91` by using the ATPs.

```
postulate f₉₁-100 : f₉₁ 100 ≡ 91
{-# ATP prove f₉₁-100 100+11>100 100+11∸10>100 101≡100+11∸10
                      91≡100+11∸10∸10 #-}
```

where the local hypotheses

```
postulate
  100+11>100      : 100 + 11 > 100
  100+11∸10>100   : 100 + 11 ∸ 10 > 100
  101≡100+11∸10   : 101 ≡ 100 + 11 ∸ 10
  91≡100+11∸10∸10 : 91 ≡ 100 + 11 ∸ 10 ∸ 10
```

are arithmetic properties, which are also automatically proved.

Remark 7.2. We would want to point out once more that, when using Agda as a logical framework, even the proofs of simple arithmetical properties like `100 + 11 > 100` have to be done either by manually constructing proof terms or by the ATPs. This reasoning is at a very low level when compared to the situation where we use Agda as a proof assistant for dependent type theories in the usual way, where computing numbers would just be a special case of the normalisation procedure which is always called during type-checking.

To prove the remaining cases, we use the following property:

```
postulate
  f₉₁-x≯100-helper : ∀ m n → m ≯ 100 →
                     f₉₁ (m + 11) ≡ n → f₉₁ n ≡ 91 → f₉₁ m ≡ 91
{-# ATP prove f₉₁-x≯100-helper #-}.
```

Let `n < 100`. To compute `f₉₁ n`, we use the equation `f₉₁-eq`, for which we first need to compute `f₉₁ (n + 11)`. Which branch of the definition of `f₉₁` we use for this computation depends on the value of `n`.

If `90 ≤ n ≤ 99`, then `n + 11 > 100`, so we apply the true-branch in the definition of `f₉₁` and obtain that the result of `f₉₁ (n + 11)` is `(n + 11) ∸ 10`, and we again apply `f₉₁` to this result. We now use the `f₉₁-x≯100-helper` property to prove that `f₉₁ n` returns `91`. For example, for the case of `n = 98`, we have:

```
postulate
  f₉₁-109 : f₉₁ (98 + 11) ≡ 99
  f₉₁-99  : f₉₁ 99 ≡ 91
{-# ATP prove f₉₁-109 98+11>100 x+11∸10≡Sx #-}
{-# ATP prove f₉₁-99 f₉₁-x≯100-helper f₉₁-110 f₉₁-100 #-}
```

where the new local hypotheses

```
postulate
  98+11>100 : [98] + [11] > [100]
  f₉₁-110   : f₉₁ (99 + 11) ≡ 100
```

are automatically proved, and

```
x+11∸10≡Sx : ∀ {n} → N n → n + 11 ∸ 10 ≡ succ₁ n
```

is proved using our combined approach.

On the other hand, if `n ≤ 89`, then `n + 11 ≯ 100`. Hence, our inductive hypothesis tells us that `f₉₁ (n + 11) ≡ 91`. Using the `f₉₁-x≯100-helper` property on the inductive hypothesis and on the proof that `f₉₁ 91 ≡ 91`, we obtain the desired result.

Third property. The function `f₉₁` is total on total and finite natural numbers.

Given the following automatic proofs:

```
postulate
  100-N : N 100
  91-N  : N 91
{-# ATP prove 100-N #-}
{-# ATP prove 91-N #-}
```

and the property

```
x>yvx≯y : ∀ {m n} → N m → N n → m > n ∨ m ≯ n
```

the proof that the function `f₉₁` returns a total and finite natural number is done by pattern matching on a proof that the argument is a total and finite natural number, and it uses the first and second properties proved above, that is, `f₉₁-x>100` and `f₉₁-x≯100`.

```
f₉₁-N : ∀ {n} → N n → N (f₉₁ n)
f₉₁-N {n} Nn = case prf₁ prf₂ (x>yvx≯y Nn 100-N)
  where
  prf₁ : n > 100' → N (f₉₁ n)
  prf₁ n>100 = subst N (sym (f₉₁-x>100 n n>100)) (∸-N Nn 10-N)

  prf₂ : n ≯ 100' → N (f₉₁ n)
  prf₂ n≯100 = subst N (sym (f₉₁-x≯100 Nn n≯100)) 91-N.
```

In this proof, we use disjunction elimination to establish whether if the natural number is greater than `100` or not. If so, we use the property `f₉₁-x>100`; if not, we use the property `f₉₁-x≯100`.

Fourth property. For all total and finite natural numbers `n`

```
n < f₉₁ n + 11.
```

The property formalised by

```
f₉₁-ineq : ∀ {n} → N n → n < f₉₁ n + 11
```

is also proved by well-founded induction on the relation `_◁_` by using the totality of the function `f₉₁` and some properties related to the inequalities.

7.2 Mirror: A Higher-Order Recursive Function

Here, we define the `mirror` function for general trees, that is, tree structures with an arbitrary branching factor [Bird and Wadler 1988]. In Haskell, for

example, general trees can be represented either as two mutually recursive data types, or as a single type using the list data type. In our FOTC-formalisation, we shall follow the first approach.

First, we extend our language with a constructor for trees:

```
postulate node : D → D → D.
```

Using the constructors for lists `[]` and `_∷_` (see Example 5.12), we mutually define predicates for total and finite forests, and for total and finite trees, that is, trees with a finite numbers of branches.

```
data Forest : D → Set
data Tree   : D → Set

data Forest where
  fnil  : Forest []
  fcons : ∀ {t ts} → Tree t → Forest ts → Forest (t ∷ ts)
{-# ATP axiom fnil fcons #-}

data Tree where
  tree : ∀ d {ts} → Forest ts → Tree (node d ts)
{-# ATP axiom tree #-}.
```

Furthermore, we define the `reverse` function for lists:

```
postulate
  rev    : D → D → D
  rev-[] : ∀ ys → rev [] ys ≡ ys
  rev-∷  : ∀ x xs ys → rev (x ∷ xs) ys ≡ rev xs (x ∷ ys)
{-# ATP axiom rev-[] rev-∷ #-}

reverse : D → D
reverse xs = rev xs []
{-# ATP definition reverse #-}
```

and the `mirror` function for trees:

```
postulate
  mirror    : D
  mirror-eq : ∀ d ts →
                mirror · node d ts ≡
                node d (reverse (map mirror ts))
{-# ATP axiom mirror-eq #-}.
```

Given the previous definitions, we mutually prove the following properties:

```
mirror-involutive : ∀ {t} → Tree t → mirror · (mirror · t) ≡ t
helper : ∀ {ts} → Forest ts →
         reverse (map mirror (reverse (map mirror ts))) ≡ ts
```

that is, we prove that `mirror` is an involution and we prove that the double composition of `reverse` and `map mirror` is the identity.

To prove the `mirror-involutive` property, we first do case analysis on a proof that the tree is total and finite, and then we do case analysis on its underlying forest; we obtain two cases, depending on whether or not the forest is empty.

```
mirror-involutive (tree d fnil) = prf
  where postulate  prf : mirror · (mirror · node d []) ≡ node d []
        {-# ATP prove prf #-}

mirror-involutive (tree d (fcons {t} {ts} Tt Fts)) = prf
  where
  postulate
    prf : mirror · (mirror · node d (t ∷ ts)) ≡ node d (t ∷ ts)
 {-# ATP prove prf helper #-}.
```

The local hypothesis `helper` follows by induction on forests. Both cases are automatically proved. The case for the non-empty forest uses the inductive hypothesis, and the following local hypotheses:

```
reverse-Forest : ∀ {xs} → Forest xs → Forest (reverse xs)

reverse-++ : ∀ {xs ys} → Forest xs → Forest ys →
             reverse (xs ++ ys) ≡ reverse ys ++ reverse xs

reverse-∷ : ∀ {x ys} → Tree x → Forest ys →
            reverse (x ∷ ys) ≡ reverse ys ++ (x ∷ [])

mirror-Tree : ∀ {t} → Tree t → Tree (mirror · t)
```

which are also proved using our combined approach.

The proofs of `reverse-Forest`, `reverse-++` and `reverse-∷` are in Appendix D. To prove the `mirror-Tree` property, we need induction principles that cover the mutual structure of the types `Tree` and `Forest`. Based on Coq's `Scheme` command, we define the following mutual induction principles:

```
Tree-mutual-ind :
  {A B : D → Set} →
  (∀ d {ts} → Forest ts → B ts → A (node d ts)) →
  B [] →
  (∀ {t ts} → Tree t → A t → Forest ts → B ts → B (t ∷ ts)) →
  ∀ {t} → Tree t → A t
```

```
Forest-mutual-ind :
  {A B : D → Set} →
  (∀ d {ts} → Forest ts → B ts → A (node d ts)) →
  B [] →
  (∀ {t ts} → Tree t → A t → Forest ts → B ts → B (t ∷ ts)) →
  ∀ {ts} → Forest ts → B ts.
```

Using the `Tree-mutual-ind` principle, the combined proof of the `mirror-Tree` property is given by

```
mirror-Tree : ∀ {t} → Tree t → Tree (mirror · t)
mirror-Tree = Tree-mutual-ind {A} {B} hA B[] hB
  where
  A : D → Set
  A t = Tree (mirror · t)
  {-# ATP definition A #-}

  B : D → Set
  B ts = Forest (map mirror ts)
  {-# ATP definition B #-}

  postulate
    hA  : ∀ d {ts} → Forest ts → B ts → A (node d ts)
    B[] : B []
    hB  : ∀ {t ts} → Tree t → A t → Forest ts → B ts → B (t ∷ ts)
  {-# ATP prove hA reverse-Forest #-}
  {-# ATP prove B[] #-}
  {-# ATP prove hB #-}
```

7.3 Collatz: A Function without a Termination Proof

The Collatz conjecture, also known as the $3n+1$ conjecture (see, for example, Lagarias [2012]), asserts that for every positive integer, if we iterate the function

$$C(n) = \begin{cases} 3n+1 & \text{for } n \text{ odd,} \\ n/2 & \text{otherwise,} \end{cases}$$

it eventually would terminate in 1.

For example, if we start the iteration of $C(n)$ with 7 we obtain the following values: 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2 and 1.

Since the Collatz conjecture remains unsolved, we do not know if the following Haskell function is total:

```
collatz :: Nat → Nat
collatz 0 = 1
collatz 1 = 1
collatz n =
  if even n then collatz (div n 2) else collatz (3 * n + 1).
```

Despite of this, we can prove some properties of the function, such as: for all total and finite natural numbers `n`, `collatz (2^n) ≡ 1`.

To formalise the `collatz` function in FOTC, we start by adding axioms for defining the `div` function, and the mutually recursive functions `even` and `odd`.

```
postulate
  div     : D → D → D
  div-x<y : ∀ {m n} → n ≢ 0 → m < n → div m n ≡ 0
  div-x≥y : ∀ {m n} → n ≢ 0 → m ≥ n →
            div m n ≡ succ₁ (div (m ∸ n) n)
{-# ATP axiom div-x<y div-x≥y #-}.

postulate
  even odd : D → D
  even-0   : even 0 ≡ true
  even-S   : ∀ n → even (succ₁ n) ≡ odd n
  odd-0    : odd 0 ≡ false
  odd-S    : ∀ n → odd (succ₁ n) ≡ even n
{-# ATP axiom even-0 even-S odd-0 odd-S #-}.
```

Now, we can formalise the function by adding the following axioms:

```
postulate
  collatz         : D → D
  collatz-0       : collatz 0 ≡ 1
  collatz-1       : collatz 1 ≡ 1
  collatz-even    : ∀ {n} → Even (succ₁ (succ₁ n)) →
                    collatz (succ₁ (succ₁ n)) ≡
                    collatz (div (succ₁ (succ₁ n)) 2)
  collatz-noteven : ∀ {n} → NotEven (succ₁ (succ₁ n)) →
                    collatz (succ₁ (succ₁ n)) ≡
                    collatz (3 * (succ₁ (succ₁ n)) + 1)
{-# ATP axiom collatz-0 collatz-1 collatz-even collatz-noteven #-}
```

where the predicates `Even` and `NotEven` are defined by

```
Even : D → Set
Even n = even n ≡ true
{-# ATP definition Even #-}
```

```
NotEven : D → Set
NotEven n = even n ≡ false
{-# ATP definition NotEven #-}.
```

To prove that `collatz (2^n) ≡ 1` using our combined approach, we first define the exponential function by the recursive equations

```
postulate
  _^_ : D → D → D
  ^-0 : ∀ n → n ^ 0 ≡ 1
  ^-S : ∀ m n → m ^ succ₁ n ≡ m * m ^ n
{-# ATP axiom ^-0 ^-S #-}.
```

Finally, we do induction on the proof that `n` is a total and finite natural number.

```
collatz-2^x : ∀ {n} → N n → collatz (2 ^ n) ≡ 1

collatz-2^x nzero = prf
  where postulate prf : collatz (2 ^ 0) ≡ 1
        {-# ATP prove prf #-}

collatz-2^x (nsucc {n} Nn) = prf (collatz-2^x Nn)
  where
  postulate prf : collatz (2 ^ n) ≡ 1 → collatz (2 ^ succ₁ n) ≡ 1
 {-# ATP prove prf helper #-}.
```

The local hypothesis

```
helper : ∀ {n} → N n → collatz (2 ^ succ₁ n) ≡ collatz (2 ^ n)
```

follows by induction on a proof that `n` is a total and finite natural number. Both cases are automatically proved.

7.4 Alternating Bit Protocol: A Program Using Unguarded Co-Recursive Functions

We shall now show how to prove the correctness of a network protocol: the alternating bit protocol (henceforth, ABP). The purpose of this protocol is to ensure safe communication over unreliable transmission channels. To achieve this, the sender tags the message with an *alternating* bit, which is checked by the receiver. In the case of proper transmission, the receiver sends the bit back to the sender as an acknowledgement. Otherwise, the receiver sends the opposite bit back to signal that the message needs to be resent.

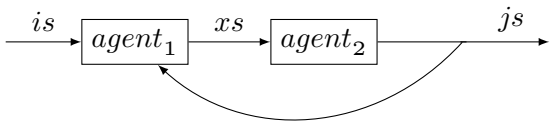


Fig. 7.1: Network of Communicating Process.

We follow Dybjer and Sander [1989], who showed how to represent the ABP as an (incompletely specified) Kahn network (see, for example, MacQueen [2009]), that is, as a network of communicating mutually recursive stream transformers. They implemented the ABP in the lazy functional programming language Miranda [Turner 1986], a precursor of Haskell. In Appendix E, we rewrote Dybjer and Sander's program for the ABP in Haskell. There, we can see that the `outH` function is not a guarded-by-constructors co-recursive function (see, for example, Giménez [1995]).

Dybjer and Sander proved the ABP correct in Park's *higher-order* μ-calculus, which uses the μ-operator for representing n-ary inductive predicates (see § 5.3) and the ν-operator for representing n-ary co-inductive predicates (see § 5.5). They implemented the μ-calculus in the Isabelle system, and the proof was mechanically checked using Isabelle's tactics.

Here, we show how to modify Dybjer and Sander's approach so that it fits within FOTC. Rather than using the μ-operator and the ν-operator (second-order constructs) for inductive and co-inductive predicates, respectively, we add new predicate symbols to FOTC with axioms and axiom schemata corresponding to the least and greatest fixed-point properties. We have shown how this can be done in § 5.3 and § 5.5, respectively.

Remark 7.3. Dybjer and Sander [1989] used a higher-order logic in the formalisation of the ABP. Therefore, we also use a higher-order logic to formalise the ABP, in which function application is represented by juxtaposition. This logic is used in § 7.4.1, § 7.4.3 and § 7.4.4.

7.4.1 Recursive Definition of Networks of Communicating Process

Following Kahn's work, Sander [1992] showed how to define the set of mutually recursive equations associated to a network of communicating process.

For example, for the network of Fig. 7.1—where the nodes are computing agents which communicate with each other along the arcs—representing the messages as streams and the agents as stream transformers, we can

determine js from is using the following system of equations:

$$\begin{aligned} xs &= f_1\ is\ js, \\ js &= f_2\ xs, \end{aligned} \tag{7.1}$$

where the functions f_1 and f_2 corresponded to $agent_1$ and $agent_2$, respectively.

In addition, for the network of Fig. 7.1, there is a network transfer function $trans$ returning the output js from the input is and the functions computing the input-output of each agent.

$$js = trans\ f_1\ f_2\ is. \tag{7.2}$$

Now, from (7.1) and (7.2), we get the mutually recursive equations for the $trans$ function by making xs into a function h_{xs}, which operates on f_1, f_2 and is

$$\begin{aligned} \forall f_1\ f_2\ is.\ h_{xs}\ f_1\ f_2\ is &= f_1\ is\ (trans\ f_1\ f_2\ is), \\ \forall f_1\ f_2\ is.\ trans\ f_1\ f_2\ is &= f_2\ (h_{xs}\ f_1\ f_2\ is). \end{aligned} \tag{7.3}$$

7.4.2 Non-Deterministic Agents

Kahn networks consist of deterministic agents. At the specification level, Dybjer and Sander view a non-deterministic agent as an incompletely specified deterministic one, hence, it is a property of a stream transformer. At the implementation level, they supply the missing information in the form of an oracle stream which does not appear at the non-deterministic level and can be thought of as generated at run time, as highlighted in [Sander 1992].

An example of a non-deterministic agent is an unreliable transmission channel, where each item is either correctly or erroneously transmitted. In this case, Dybjer and Sander assume a fairness property which means that for each time, there is a later time when the unreliable transmission channel transmits an item correctly.

7.4.3 Specification of Networks of Communicating Process

Dybjer and Sander showed the analogy between the specification of a concurrent system satisfying certain conditions and the specification of a single functional program.

The specification of a functional program focuses on the input and output correspondence of the program, that is, the specification can be divided into an input condition (a logical predicate) P and an input-output logical relation R. In consequence, a specification has the form

$$\mathsf{spec}\ f \stackrel{\text{def}}{=} \forall x.\ P(x) \supset R(x, (f\ x)),$$

where any function f that satisfies the above formula satisfies the specification.

Can the above notion of specification of a functional program be generalised to concurrent systems? Dybjer and Sander's [1989, p. 308] answer is affirmative:

> *The idea is to model a system as a Kahn-network or, if there are non-deterministic agents, as an incompletely specified Kahn-network.*
>
> *Consider a system which satisfies the following criteria:*
>
> 1. *The topology of the system is fixed. Input and output channels are determined. Streams of messages are communicated between the agents.*
> 2. *Agents are separated into those which we wish to program and those which are predetermined. The latter kind may be non-deterministic.*
> 3. *The purpose of the system is to realise a certain relation R between inputs and outputs, provided that the inputs satisfy certain conditions P, and the predetermined agents satisfy certain other conditions Q.*
>
> *Let $\vec{h}$ be the stream transformers associated with the agents (the vector notation indicates that there may be several agents), let $\overrightarrow{is}$ be the input streams, and let $\overrightarrow{js}$ be the output streams. Then, since the topology is fixed, there are network transfer functions $\overrightarrow{trans}$ (one for each output channel) such that*
>
> $$\overrightarrow{js} = \overrightarrow{trans}\ \vec{h}\ \overrightarrow{is}.$$
>
> *Note that these functions $\overrightarrow{trans}$ are Kahn-network analogues of function application and that they only depend on the topology of the network.*
>
> *By our second assumption, agents (and thus stream transformers) are separated into those which we wish to program and those which are predetermined. If we use $\vec{f}$ for the former and $\vec{g}$ for the latter, we can rewrite the equation:*
>
> $$\overrightarrow{js} = \overrightarrow{trans}\ \vec{f}\ \vec{g}\ \overrightarrow{is}.$$
>
> *By our third assumption, the intended behaviour of the network is specified by the input conditions P, by the conditions Q on the predetermined agents, and by the input-output relation R. Thus the specification of our task, that is, to program $\vec{f}$, is*
>
> $$\mathsf{spec}\ \vec{f} = \forall \vec{g}\ \overrightarrow{is}.\ (Q(\vec{g}) \wedge P(\overrightarrow{is})) \supset R(\overrightarrow{is}, (\overrightarrow{trans}\ \vec{f}\ \vec{g}\ \overrightarrow{is})).$$

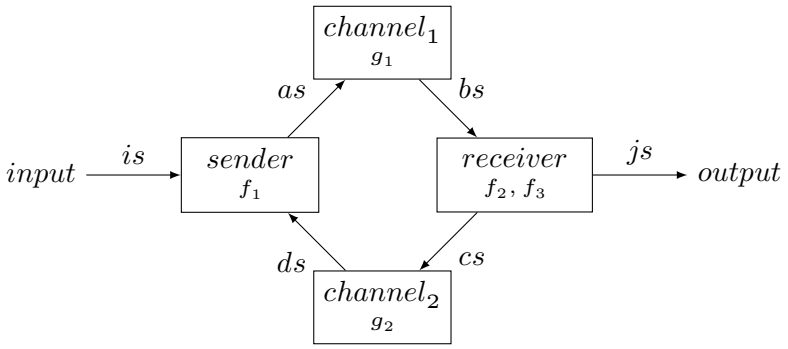


Fig. 7.2: Network topology for the alternating bit protocol.

7.4.4 Specification Based on the Network Topology

The ABP satisfies the three conditions highlighted in § 7.4.3 for the specification of a network of communicating process:

1. It has a fixed topology given by Fig. 7.2.
2. There are four agents: it is necessary to program the *sender* and *receiver* agents; the $channel_1$ and $channel_2$ agents are predetermined.
3. The input and the predetermined agents satisfy certain conditions, namely, is is a stream, and $channel_1$ and $channel_2$ are fair unreliable transmission channels (see § 7.4.2). The purpose of the ABP is to produce output which is in a bisimilarity relation $\_\approx\_$ (see Example 6.17) with the input.

Let f_1 be the function associated with the *sender*, f_2 and f_3 (one function for each output) the functions associated with the *receiver*, and g_1 and g_2 the functions associated with $channel_1$ and $channel_2$, respectively. The *trans* function associated with the topology of the ABP satisfies

$$js = trans\ f_1\ f_2\ f_3\ g_1\ g_2\ is,$$

and we can determine the output js from the input is using the following system of equations:

$$\begin{aligned} as &= f_1\ is\ ds, \\ bs &= g_1\ as, \\ cs &= f_2\ bs, \\ ds &= g_2\ cs, \\ js &= f_3\ bs. \end{aligned} \tag{7.4}$$

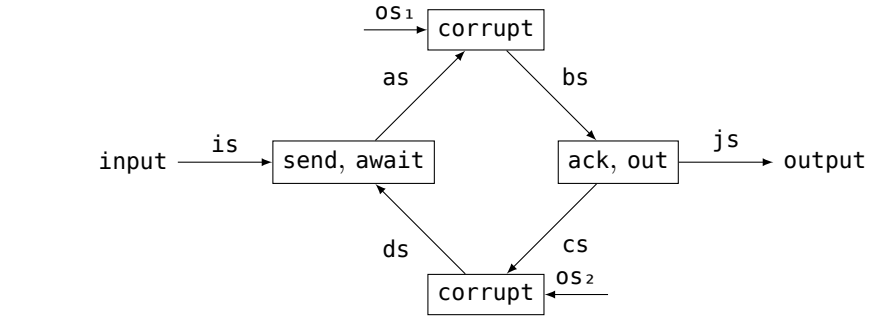


Fig. 7.3: Stream transformers for the alternating bit protocol.

In consequence, the specification based on the network topology of the ABP is given by

$$\mathsf{protocol}\ f_1\ f_2\ f_3 \stackrel{\text{def}}{=} \forall g_1\ g_2\ is.\ (\mathcal{F}air(g_1) \wedge \mathcal{F}air(g_2) \wedge \mathcal{S}tream(is)) \supset is \approx trans\ f_1\ f_2\ f_3\ g_1\ g_2\ is, \tag{7.5}$$

where $\mathcal{S}tream$ is the co-inductive predicate defined in Example 5.28 and $\mathcal{F}air$ is a co-inductive predicate to be defined in § 7.4.6. We get the mutually recursive equations for the $trans$ function by making as, bs, cs and ds into functions h_{as}, h_{bs}, h_{cs} and h_{ds}, which operate on f_1, f_2, f_3, g_1, g_2 and is:

$$\begin{aligned}
&\forall\ f_1\ f_2\ f_3\ g_1\ g_2\ is.\ h_{as}\ f_1\ f_2\ f_3\ g_1\ g_2\ is = f_1\ is\ (h_{ds}\ f_1\ f_2\ f_3\ g_1\ g_2\ is),\\
&\forall\ f_1\ f_2\ f_3\ g_1\ g_2\ is.\ h_{bs}\ f_1\ f_2\ f_3\ g_1\ g_2\ is = g_1\ (h_{as}\ f_1\ f_2\ f_3\ g_1\ g_2\ is),\\
&\forall\ f_1\ f_2\ f_3\ g_1\ g_2\ is.\ h_{cs}\ f_1\ f_2\ f_3\ g_1\ g_2\ is = f_2\ (h_{bs}\ f_1\ f_2\ f_3\ g_1\ g_2\ is),\\
&\forall\ f_1\ f_2\ f_3\ g_1\ g_2\ is.\ h_{ds}\ f_1\ f_2\ f_3\ g_1\ g_2\ is = g_2\ (h_{cs}\ f_1\ f_2\ f_3\ g_1\ g_2\ is),\\
\forall\ f_1\ f_2\ f_3\ g_1\ g_2\ is.\ &trans\ f_1\ f_2\ f_3\ g_1\ g_2\ is = f_3\ (h_{bs}\ f_1\ f_2\ f_3\ g_1\ g_2\ is).
\end{aligned} \tag{7.6}$$

7.4.5 First-Order Implementation

Dybjer and Sander programmed the ABP in Miranda. They programmed the sender by two mutually recursive stream transformers, `send` and `await`. The receiver is programmed as a pair of stream transformers, `ack`, which returns the acknowledgement stream `cs`, and `out`, which returns the output stream `js`. Moreover, an unreliable transmission channel is programmed as a stream transformer, which non-deterministically corrupts the messages in the stream. They model the channels of the ABP as a stream transformer `corrupt`, which accepts an oracle bit stream as an auxiliary argument (see § 7.4.2). The above stream transformers are illustrated in Fig. 7.3.

Before introducing the FOTC-equations satisfied by the ABP's stream transformers, we need to add some FOTC-terms. We first add the constant symbols `T` and `F` for bits. They are just synonyms for the FOTC-terms `true` and `false`, respectively.

```
F T : D
F = false
T = true
{-# ATP definition F T #-}
```

Moreover, we shall use the `not` function for negation of bits, `<_,_>` for pairs, `error` for a corrupted message and `ok` for the constructor of a proper message, corresponding to the constructors `Nothing` and `Just`, respectively, of Haskell's `Maybe` data type (see `Err` data type in Appendix E).

```
not : D → D
not b = if b then false else true
{-# ATP definition not #-}
```

```
postulate
  <_,_> : D → D → D
  ok    : D → D
  error : D.
```

Now, the recursive equations for the `corrupt` stream transformer are:

```
postulate
  corrupt   : D → D
  corrupt-T : ∀ os x xs →
              corrupt (T :: os) · (x :: xs) ≡ ok x :: corrupt os · xs
  corrupt-F : ∀ os x xs →
                corrupt (F :: os) · (x :: xs) ≡
                error :: corrupt os · xs
{-# ATP axiom corrupt-T corrupt-F #-}.
```

Before introducing the missing FOTC-equations for the stream transformers of the ABP, we shall justify the type `D → D` for the `corrupt` stream transformer, that is, we shall explain our methodology for staying within first-order in the formalisation of the ABP.

From the **`where`** clause for the `abpTransH` function in the Appendix E, the system of equations in (7.4) corresponding to Fig. 7.3 is given by

$$\begin{aligned} as &= \mathsf{sendH}\ is\ ds, \\ bs &= \mathsf{corruptH}\ os_1\ as, \\ cs &= \mathsf{ackH}\ bs, \\ ds &= \mathsf{corruptH}\ os_1\ cs, \\ js &= \mathsf{outH}\ bs. \end{aligned}$$

From this system of equations, given the types of the functions `sendH`, `ackH`, `outH` and `corruptH` in Appendix E, the Haskell definitions of the higher-order mutually recursive equations in (7.6) are given by

```
type SendTy a     = Stream a → Stream (Err Bit) → Stream (a, Bit)
type AckTy a      = Stream (Err (a, Bit)) → Stream Bit
type OutTy a      = Stream (Err (a, Bit)) → Stream a
type CorruptTy1 a = Stream (a, Bit) → Stream (Err (a, Bit))
type CorruptTy2   = Stream Bit → Stream (Err Bit)

hasH :: SendTy a → AckTy a → OutTy a → CorruptTy1 a → CorruptTy2 →
       Stream a →
       Stream (a, Bit)
hasH f1 f2 f3 g1 g2 is = f1 is (hdsH f1 f2 f3 g1 g2 is)

hbsH :: SendTy a → AckTy a → OutTy a → CorruptTy1 a → CorruptTy2 →
       Stream a →
       Stream (Err (a, Bit))
hbsH f1 f2 f3 g1 g2 is = g1 (hasH f1 f2 f3 g1 g2 is)

hcsH :: SendTy a → AckTy a → OutTy a → CorruptTy1 a → CorruptTy2 →
       Stream a →
       Stream Bit
hcsH f1 f2 f3 g1 g2 is = f2 (hbsH f1 f2 f3 g1 g2 is)

hdsH :: SendTy a → AckTy a → OutTy a → CorruptTy1 a → CorruptTy2 →
       Stream a →
       Stream (Err Bit)
hdsH f1 f2 f3 g1 g2 is = g2 (hcsH f1 f2 f3 g1 g2 is)

transferH :: SendTy a →
             AckTy a →
             OutTy a →
             CorruptTy1 a →
             CorruptTy2 →
             Stream a →
             Stream a
transferH f1 f2 f3 g1 g2 is = f3 (hbsH f1 f2 f3 g1 g2 is).
```

In addition, we redefine the `abpTransH` function using the `transferH` function by

```
abpTransH :: Bit → Stream Bit → Stream Bit → Stream a → Stream a
abpTransH b os1 os2 is =
  transferH (sendH b) (ackH b) (outH b)
            (corruptH os1) (corruptH os2) is.
```

In § 7.4.6, the first-order version of the `transferH` function will be the function

```
transfer : D → D → D → D → D → D → D
```

whose fourth and fifth arguments will be `corrupt os₁` and `corrupt os₂`, respectively. For this reason, the type of the `corrupt` stream transformer is `D → D`.

We may also define the `corrupt` stream transformer as a constant using one additional FOTC-application `_·_` as follows:

```
postulate
  corrupt   : D
  corrupt-T : ∀ os x xs →
                corrupt · (T :: os) · (x :: xs) ≡
                ok x :: corrupt · os · xs
  corrupt-F : ∀ os x xs →
                corrupt · (F :: os) · (x :: xs) ≡
                error :: corrupt · os · xs.
```

However, we only use the FOTC-application when it is strictly necessary for staying within a first-order formalisation.

Note also we cannot define the `corrupt` stream transformer as a binary function by removing the FOTC-application, that is,

```
postulate
  corrupt   : D → D → D
  corrupt-T : ∀ os x xs →
              corrupt (T :: os) (x :: xs) ≡ ok x :: corrupt os xs
  corrupt-F : ∀ os x xs →
              corrupt (F :: os) (x :: xs) ≡ error :: corrupt os xs
```

because for example, `corrupt os₁` would not be of type `D` as required by the first-order `transfer` function.

The previous discussion about the type of the `corrupt` stream transformer also applies to the types of the `send`, `ack` and `out` stream transformers. On the other hand, we shall use a 4-ary function for representing the `await` stream transformer, because it is not an argument of the `transfer` function. Perhaps for reasons of "symmetry", we may also use a binary function for representing the `await` stream transformer. However, as was highlighted above, we only use the FOTC-application when it is strictly necessary.

Now, we continue with the definition of the missing stream transformers for the ABP. The `send`, `await`, `ack` and `out` stream transformers satisfy the following recursive equations:

```
postulate
  send out ack : D → D
```

```
  await        : D → D → D → D → D

  send-eq : ∀ b i is ds →
            send b · (i :: is) · ds ≡ < i , b > :: await b i is ds

  await-ok≡   : ∀ b b' i is ds → b ≡ b' →
                  await b i is (ok b' :: ds) ≡
                  send (not b) · is · ds
  await-ok≢   : ∀ b b' i is ds → b ≢ b' →
                  await b i is (ok b' :: ds) ≡
                  < i , b > :: await b i is ds
  await-error : ∀ b i is ds →
                  await b i is (error :: ds) ≡
                  < i , b > :: await b i is ds

  out-ok≡   : ∀ b b' i bs → b ≡ b' →
              out b · (ok < i , b' > :: bs) ≡ i :: out (not b) · bs
  out-ok≢   : ∀ b b' i bs → b ≢ b' →
              out b · (ok < i , b' > :: bs) ≡ out b · bs
  out-error : ∀ b bs → out b · (error :: bs) ≡ out b · bs

  ack-ok≡   : ∀ b b' i bs → b ≡ b' →
              ack b · (ok < i , b' > :: bs) ≡ b :: ack (not b) · bs
  ack-ok≢   : ∀ b b' i bs → b ≢ b' →
              ack b · (ok < i , b' > :: bs) ≡ not b :: ack b · bs
  ack-error : ∀ b bs → ack b · (error :: bs) ≡ not b :: ack b · bs

{-# ATP axiom send-eq await-ok≡ await-ok≢ await-error
              out-ok≡ out-ok≢ out-error
              ack-ok≡ ack-ok≢ ack-error #-}.
```

7.4.6 Correctness Proof

We shall start by adding some inductive and co-inductive predicates, which will be used in the proof of the correctness of the ABP.

First, we formalise the fairness property of the unreliable transmission channels using a co-inductive predicate `Fair : D → Set`. This property will be encoded in terms of oracle bit streams, where the bits `T` and `F` represent proper and improper transmission, respectively. Fairness here means that the bit stream contains a potentially infinite number of `T`'s and is defined as follows:

```
data F*T : D → Set where
  f*tnil  : F*T (T :: [])
  f*tcons : ∀ {ft} → F*T ft → F*T (F :: ft)
```

```
  postulate
    Fair : D → Set

    Fair-out : ∀ {os} → Fair os →
               ∃[ ft ] ∃[ os' ] F*T ft ∧ os ≡ ft ++ os' ∧ Fair os'

    Fair-coind : (A : D → Set) →
                 (∀ {os} → A os → ∃[ ft ] ∃[ os' ] F*T
                   ft ∧ os ≡ ft ++ os' ∧ A os') →
                 ∀ {os} → A os → Fair os
  {-# ATP axiom Fair-out #-}.
```

Here, `F*T ft` is an inductive predicate expressing that `ft` is a (possibly empty) finite list of `F`'s followed by a final `T`.

In the proofs below, we need some properties related to the co-inductive predicate `Fair`. These properties are proved using our combined approach.

One of the properties required states an introduction rule for the `Fair` predicate:

```
  Fair-in : ∀ {os} →
            ∃[ ft ] ∃[ os' ] F*T ft ∧ os ≡ ft ++ os' ∧ Fair os' →
            Fair os
  Fair-in h = Fair-coind A h' h
    where
    A : D → Set
    A os = ∃[ ft ] ∃[ os' ] F*T ft ∧ os ≡ ft ++ os' ∧ Fair os'
    {-# ATP definition A #-}

    postulate
      h' : ∀ {os} → A os →
           ∃[ ft ] ∃[ os' ] F*T ft ∧ os ≡ ft ++ os' ∧ A os'
    {-# ATP prove h' #-}.
```

To prove the property, we interactively instantiate the co-induction rule `Fair-coind` with the predicate `A` and the required hypothesis `h'` is automatically proved by the ATPs.

In the proofs below, we shall also make use of the unary inductive predicate `Bit : D → Set`, which is just a synonym for the inductive predicate `Bool`.

```
  Bit : D → Set
  Bit b = Bool b
  {-# ATP definition Bit #-}.
```

The ABP has two behaviours depending on the starting bit `b : Bit`. Following Dybjer and Sander [1989], we shall prove both behaviours simultaneously with the help of two auxiliary lemmas called $\texttt{lemma}_1$ and $\texttt{lemma}_2$.

The first lemma states that given a starting state `S` of the ABP, we will arrive at a state `S'`, where the message has been received by the receiver but the acknowledgement has not yet been received by the sender. Formally, the states `S` and `S'` are represented by two FOTC-relations.

```
S : D → D → D → D → D → D → D → D → D → Set
S b is os₁ os₂ as bs cs ds js =
  as ≡ send b · is · ds
  ∧ bs ≡ corrupt os₁ · as
  ∧ cs ≡ ack b · bs
  ∧ ds ≡ corrupt os₂ · cs
  ∧ js ≡ out b · bs
{-# ATP definition S #-}
```

```
S' : D → D → D → D → D → D → D → D → D → D → Set
S' b i' is' os₁' os₂' as' bs' cs' ds' js' =
  as' ≡ await b i' is' ds'
  ∧ bs' ≡ corrupt os₁' · as'
  ∧ cs' ≡ ack (not b) · bs'
  ∧ ds' ≡ corrupt os₂' · (b :: cs')
  ∧ js' ≡ out (not b) · bs'
{-# ATP definition S' #-}.
```

Formally, the first lemma is represented by

```
lemma₁ :
  ∀ {b i' is' os₁ os₂ as bs cs ds js} →
  Bit b →
  Fair os₁ →
  Fair os₂ →
  S b (i' :: is') os₁ os₂ as bs cs ds js →
  ∃[ os₁' ] ∃[ os₂' ] ∃[ as' ] ∃[ bs' ] ∃[ cs' ] ∃[ ds' ] ∃[ js' ]
    Fair os₁'
    ∧ Fair os₂'
    ∧ S' b i' is' os₁' os₂' as' bs' cs' ds' js'
    ∧ js ≡ i' :: js'.
```

The second lemma states that given a state `S'`, we will arrive at a new starting state, which is identical to the old starting state except that the bit has been alternated and the first item in the input stream has been removed. Formally, the second lemma is represented by

```
lemma₂ :
  ∀ {b i' is' os₁' os₂' as' bs' cs' ds' js'} →
  Bit b →
  Fair os₁' →
  Fair os₂' →
  S' b i' is' os₁' os₂' as' bs' cs' ds' js' →
  ∃[ os₁'' ] ∃[ os₂'' ] ∃[ as'' ] ∃[ bs'' ] ∃[ cs'' ] ∃[ ds'' ]
    Fair os₁''
    ∧ Fair os₂''
    ∧ S (not b) is' os₁'' os₂'' as'' bs'' cs'' ds'' js'.
```

(Dybjer and Sander [1989] require the additional hypothesis `Stream is'` in both lemmas. The hypotheses were not used in their formalisation nor in ours).

The proofs of both `lemma₁` and `lemma₂` have the same shape. For `lemma₁`, given that `Fair os₁`, we can unfolding `os₁` using `Fair-out` and then we know there exists `ft₁ os₁' : D` such that `F*T ft₁`, `os₁ ≡ ft₁ ++ os₁'` and `Fair os₁'`. Similarly, for `lemma₂`, by unfolding `os₂'`, we know there exists `ft₂ os₂'' : D` such that `F*T ft₂`, `os₂' ≡ ft₂ ++ os₂''` and `Fair os₂''`. The proofs of `lemma₁` and `lemma₂` proceed then by induction on `ft₁` and `ft₂`, respectively. Both lemmas are proved using our combined approach in Appendix F.

Proving that the ABP is correct amounts to proving that each message is eventually transmitted properly. Formally, this means that the input stream is bisimilar to the output stream. This property can only hold if one assumes that the transmission channels are fair in the sense described above.

The correctness of the ABP is stated by

```
abpCorrect : ∀ {b} → Bit b → protocol (send b) (ack b) (out b),
```

where `protocol` corresponds to the specification based on the network topology given in (7.5). Since the `protocol` specification uses the higher-order equations in (7.6), we need a first-order version of them for working within FOTC.

```
postulate
  transfer    : D → D → D → D → D → D → D
  transfer-eq : ∀ f₁ f₂ f₃ g₁ g₂ is →
                  transfer f₁ f₂ f₃ g₁ g₂ is ≡
                  f₃ · (hbs f₁ f₂ f₃ g₁ g₂ is)
{-# ATP axiom transfer-eq #-}

postulate
  has hbs hcs hds : D → D → D → D → D → D → D
```

```
  has-eq : ∀ f₁ f₂ f₃ g₁ g₂ is →
             has f₁ f₂ f₃ g₁ g₂ is ≡
             f₁ · is · (hds f₁ f₂ f₃ g₁ g₂ is)

  hbs-eq : ∀ f₁ f₂ f₃ g₁ g₂ is →
           hbs f₁ f₂ f₃ g₁ g₂ is ≡ g₁ · (has f₁ f₂ f₃ g₁ g₂ is)

  hcs-eq : ∀ f₁ f₂ f₃ g₁ g₂ is →
           hcs f₁ f₂ f₃ g₁ g₂ is ≡ f₂ · (hbs f₁ f₂ f₃ g₁ g₂ is)

  hds-eq : ∀ f₁ f₂ f₃ g₁ g₂ is →
           hds f₁ f₂ f₃ g₁ g₂ is ≡ g₂ · (hcs f₁ f₂ f₃ g₁ g₂ is)
{-# ATP axiom has-eq hbs-eq hcs-eq hds-eq #-}.
```

Here, `transfer`, `has`, `hbs`, `hcs` and `hds` are the first-order versions of those in (7.6). The `transfer` function simultaneously computes the output `js` and the streams `as`, `bs`, `cs` and `ds`, given the first-order stream transformers `f₁`, `f₂`, `f₃`, `g₁` and `g₂` (see Fig. 7.2.)

Now, we can rewrite the `abpCorrect` theorem as

```
abpCorrect : ∀ {b os₁ os₂ is} →
             Bit b → Fair os₁ → Fair os₂ → Stream is →
             is ≈ abpTransfer b os₁ os₂ is
```

where the auxiliary `abpTransfer` function computes the output `js` from the input `is`, and accepts three more arguments: the initial bit `b` and the two oracle bit streams `os₁` and `os₂`.

```
postulate
  abpTransfer : D → D → D → D → D
  abpTransfer-eq :
    ∀ b os₁ os₂ is →
      abpTransfer b os₁ os₂ is ≡
      transfer (send b) (ack b) (out b)
               (corrupt os₁) (corrupt os₂) is
{-# ATP axiom abpTransfer-eq #-}.
```

The proof of `abpCorrect` is by co-induction using the `≈-coind` rule (see Example 6.17). We prove that `is` and `js` are in the greatest bisimulation `_≈_` by finding another bisimulation `B` which they are in.

Following Dybjer and Sander [1989], the bisimulation `B` is defined by

```
B : D → D → Set
B is js = ∃[ b ] ∃[ os₁ ] ∃[ os₂ ] ∃[ as ] ∃[ bs ] ∃[ cs ] ∃[ ds ]
            Stream is
            ∧ Bit b
            ∧ Fair os₁
            ∧ Fair os₂
            ∧ S b is os₁ os₂ as bs cs ds js
{-# ATP definition B #-}.
```

Our combined proof of the correctness of the ABP is given by:

```
abpCorrect :
  ∀ {b is os₁ os₂} → Bit b → Stream is → Fair os₁ → Fair os₂ →
  is ≈ abpTransfer b os₁ os₂ is
abpCorrect {b} {is} {os₁} {os₂} Bb Sis Fos₁ Fos₂ = ≈-coind B h₁ h₂
  where
  postulate h₁ : ∀ {is js} → B is js → ∃[ i' ] ∃[ is' ] ∃[ js' ]
                   is ≡ i' ∷ is' ∧ js ≡ i' ∷ js' ∧ B is' js'
  {-# ATP prove h₁ lemma₁ lemma₂ not-Bool #-}

  postulate h₂ : B is (abpTransfer b os₁ os₂ is)
  {-# ATP prove h₂ #-}.
```

Both hypotheses `h₁` and `h₂` are automatically proved by the ATPs. The proof of the first hypothesis uses `not-Bool` as local hypothesis, which states the totality of the `not` function, and the auxiliary lemmas `lemma₁` and `lemma₂`.

Finally, since the FOTC is a type-free theory, we automatically prove that the output of the ABP is a `Stream`.

```
postulate
  abpTransfer-Stream :
    ∀ {b os₁ os₂ is} → Bit b → Fair os₁ → Fair os₂ → Stream is →
      Stream (abpTransfer b os₁ os₂ is)
{-# ATP prove abpTransfer-Stream ≈→Stream₂ abpCorrect #-}.
```

For this proof, we use the fact that the input and output of the ABP are in the bisimilarity relation `_≈_`, and we use the `≈→Stream₂` theorem which is proved using our combined approach.

```
≈→Stream₂ : ∀ {xs ys} → xs ≈ ys → Stream ys
≈→Stream₂ {xs} {ys} h = Stream-coind A h' (xs , h)
  where
  A : D → Set
  A zs = ∃[ ws ] ws ≈ zs
  {-# ATP definition A #-}

  postulate h' : ∀ {ys} → A ys → ∃[ y' ] ∃[ ys' ]
                   ys ≡ y' ∷ ys' ∧ A ys'
  {-# ATP prove h' #-}.
```

7.5 Using the Automatic Theorem Provers

As previously mentioned, the overall performance of the ATPs in our formalisations is quite satisfactory. Just to give an idea of our use of the ATPs in the examples presented in this chapter, we show in Table 7.1 the number of theorems which were automatically proved. Since some of these theorems are based on previous theorems, also automatically proved, Table 7.1 also shows the number of theorems which were automatically proved in some common libraries used in our examples.

	Proven theorems
Libraries	
Inequalities properties	179
Arithmetic properties	75
Booleans properties	38
List properties	28
Examples	
McCarthy's 91-function (§ 7.1)	75
Alternating Bit Protocol (§ 7.4)	30
Collatz function (§ 7.3)	25
Mirror function (§ 7.2)	22

Table 7.1: Theorems automatically proved by the ATPs.

Chapter 8

Conclusions

In this final chapter, we summarise the main ideas and results of this thesis, and we show some possible improvements to our work.

8.1 Results

The main goal of this thesis has been to build a computer-assisted framework for reasoning about programs written in Haskell-like lazy functional languages. To achieve this goal, we have worked on different subjects.

- Based on $\mathsf{LT}_{\mathsf{PCF}}$, we defined FOTC, which is a *first-order* programming logic suitable for reasoning about mainstream lazy functional programs including those that use *general* recursion (structural and non-structural recursion, and guarded and unguarded co-recursion). FOTC can deal with higher-order functions, (co-)inductive definitions of data types and proofs by (co-)induction. The consistency of our theory was established by the existence of a domain model for its term language, where the (co-)inductively defined predicates were interpreted as subsets of this domain model. Moreover, we can extend FOTC with new *positive* (co-)inductively defined predicates keeping the consistency of the theory.

- We chose a mature system as our interactive proof assistant to formalise our programming logics. We think this was a good idea because building a mature proof assistant from scratch for this purpose would have been a daunting task. By using Agda as a *logical framework*, we could use its support for interactively building proofs, Agda's *proof engine*: (i) support for inductively defined types, including inductive families, and function definitions using pattern matching on such types, (ii) normalisation during type-checking, (iii) commands for refining proof terms, (iv) Agda's coverage checker and (v) Agda's termination

checker. We used Agda's *proof engine* to handle the high-level proofs steps (for example, introduction of hypothesis, case analysis and the use of (co-)induction principles) required in the verification of lazy functional programs.

- Since the verification of lazy functional programs requires the use of equational reasoning or simple first-order reasoning, we used off-the-shelf ATPs to deal with both kinds of reasoning. For this purpose, we extended Agda with the ATP-pragma, which instructs Agda to interact with the ATPs. In addition, we wrote the Apia program, a Haskell program using Agda as a Haskell *library*, that translated our Agda representation of first-order formulae into the TPTP language understood by many ATPs. The Apia program also calls the ATPs to try to prove the translated conjectures. The overall result of using off-the-shelf ATPs (E, Equinox, Metis, SPASS and Vampire) is quite satisfactory.

In addition, Agda seems to work well as an interface to ATPs. We have used it not only for FOTC, but also for other first-order theories such as group theory and Peano arithmetic, and we had encouraging results. Therefore, we are interested in making our Apia program available in the Agda standard distribution because we think there is great scope for a new use of Agda: as an interface to TPTP-based first-order theorem provers.

8.2 Future Work

Our approach for reasoning about lazy functional programs can be improved in several ways. We present some of them below.

8.2.1 Proof Term Reconstruction

At the moment, the communication between Agda and the ATPs is unidirectional because we use the ATPs as oracles. A first-order conjecture represented in Agda is sent to the ATPs via the Apia program, and the ATPs prove or disprove it (using a fixed timeout). We would like to modify our Apia program so that it can return witnesses for the automatically generated proofs so that they can be checked by Agda, as done by the Sledgehammer tool for Isabelle/HOL. This modification would increase the reliability of our approach.

8.2.2 Using Agda's Standard Library in the First-Order Theory of Combinators

Agda's standard library [Danielsson et al. 2014] contains many useful standard functions on Booleans, natural numbers, lists, and other data types.

In addition, this library contains (the proofs of) many properties of these functions. Is it possible to transfer these functions and the proofs of their properties to FOTC? An affirmative answer to this question would avoid *rewriting* many structural recursive functions and the proofs of their properties in FOTC. Theoretically we know that the transference is possible given the existence of a translation of Martin-Löf's type theory into LTC [Aczel 1977b; Smith 1978, 1984], but further research is needed to carry this out in practice.

8.2.3 Connection to SMT Solvers

An interesting improvement to our Apia program would be to integrate SMT solvers into it. For example, the proofs of inequality properties required for the verification of McCarthy's 91-function (see § 7.1) are easier using a system like MetiTarski [Akbarpour and Paulson 2010], which is designed for proving universally quantified inequalities in a certain theory with the help of SMT solvers, such as Z3 [de Moura and Bjørner 2008].

8.2.4 Connection to Inductive Theorem Provers

Other interesting future work is to integrate our Apia program into systems that can automatically do proofs by induction; currently we only automate FOL reasoning. In fact, Agda comes with its own automatic theorem prover called Agsy—the Agda Synthesiser [Lindblad and Benke 2006]—which can do proofs by induction. By using (automatic) inductive theorem provers, our methodology for inductive proofs should improve.

For example, let P and Q be theorems which must be proved by induction. The latest prover in the Boyer-Moore line of inductive theorem provers, ACL2 [Kaufmann, Manolios and Moore 2000], automatically proves P or suggests that Q is needed for the proof of P. The required theorem Q could be proved using ACL2, or if it fails, using our combined approach for inductive proofs, that is, by instructing Agda to do pattern matching and the ATPs automatically proving the base and step cases of the induction.

8.2.5 Polymorphism

FOTC is a type-free logic. For example, the inductive predicate `List` defined in Example 5.14 represents "heterogeneous" total and finite lists, such as

```
xs : D
xs = 0 ∷ true ∷ 1 ∷ false ∷ [].
```

We can however also represent "homogeneous" total and finite lists. For example, the total and finite lists of total and finite natural numbers can be represented by the inductive predicate

```
data ListN : D → Set where
  lnnil  : ListN []
  lncons : ∀ {n ns} → N n → ListN ns → ListN (n :: ns).
```

An example of such lists is given by

```
ys : D
ys = 0 :: 1 :: 2 :: [].
```

Instead of representing total and finite lists of terms satisfying a particular unary inductive predicate `P` using a new "list" predicate, we can define an inductive predicate `Plist` of polymorphic total and finite lists parametrised by the predicate `P`.

```
data Plist (P : D → Set) : D → Set where
  lnil  : Plist P []
  lcons : ∀ {x xs} → P x → Plist P xs → Plist P (x :: xs).
```

By using the predicate `Plist`, the "heterogeneous" total and finite lists and the total and finite lists of total and finite natural numbers can be defined, respectively, by

```
List ListN : D → Set
List  = Plist (λ d → d ≡ d)
ListN = Plist N.
```

Unfortunately, this approach for representing polymorphic total and finite lists is not within FOL because we cannot use `P : D → Set` as a parameter of the data type `PList`. In other words, `Plist (λ d → d ≡ d)` and `Plist N` are not first-order formulae. This discussion also applies to coinductive types such as `Stream` (see Example 5.28). Therefore, further research is needed to determine if it is possible to represent polymorphic total and finite lists (or streams) in FOTC.

8.2.6 Strict Functional Programs

When reasoning about functional programs one should also consider strict languages (see, for example, Kimmell et al. [2012] and Longley and Pollack [2004]). Can the approach presented on this thesis be used for reasoning about strict functional programs?

First, it should be noted that (some of) the equations for strict λ-calculus are not the same as for the lazy λ-calculus.

For example, in Haskell a lazy application can be defined by (this operator is redundant, since Haskell ordinary application is lazy)

```
($) :: (a → b) → a → b
f $ a = f a
```

and a strict application can be defined by

```
($!) :: (a → b) → a → b
f $! x = x `seq` f x.
```

The result of the application of the function `\_ → 0` to the term `loop` depends on the kind of application used. The term

```
(\_ → 0) $ loop
```

normalises to `0` because it is not necessary to normalise the term `loop`, but the term

```
(\_ → 0) $! loop
```

loops because it is necessary to normalise that term.

On the other hand, we could formalise a lazy application by `beta` in (4.9) because Agda application is non-strict. Further research is needed to determine if it is possible to formalise a strict application using a similar approach.

Appendix A

Some Definitions from Domain Theory

Here, we present some definitions and theorems from domain theory used on this thesis.

Definition A.1 (Partially ordered set (poset))**.** A partially ordered set (poset) $(D, \sqsubseteq)$ is a set D on which the binary relation $\sqsubseteq$ satisfies the following properties:

$$\begin{aligned} \forall x.\ x \sqsubseteq x && \text{(reflexive)} \\ \forall x\ y\ z.\ x \sqsubseteq y \wedge y \sqsubseteq z \supset x \sqsubseteq z && \text{(transitive)} \\ \forall x\ y.\ x \sqsubseteq y \wedge y \sqsubseteq x \supset x = y && \text{(antisymmetry)} \end{aligned}$$

Definition A.2 (Monotone function)**.** Let $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ be two posets. A function $f : D \to D'$ is monotone if

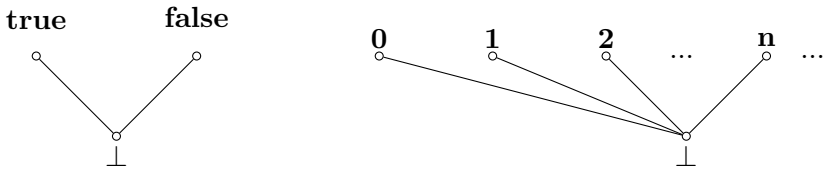
$$\forall x\ y.\ x \sqsubseteq y \supset f(x) \sqsubseteq' f(y).$$

Definition A.3 (ω-chain)**.** Let $\mathbf{D} = (D, \sqsubseteq)$ be a poset. A ω-chain of $\mathbf{D}$ is an increasing chain $d_0 \sqsubseteq d_1 \sqsubseteq \cdots \sqsubseteq d_n \sqsubseteq \cdots$, where $d_i \in D$.

Definition A.4 (ω-complete partial order (ω-cpo))**.** Let $\mathbf{D} = (D, \sqsubseteq)$ be a poset. The poset $\mathbf{D}$ is a ω-complete partial order (ω-cpo) if [Plotkin 1992]

1. There is a least element $\bot \in D$, that is, $\forall x.\ \bot \sqsubseteq x$. The element $\bot$ is called *bottom*.
2. For every ω-chain $d_0 \sqsubseteq d_1 \sqsubseteq \cdots \sqsubseteq d_n \sqsubseteq \cdots$, the least upper bound $\bigsqcup_{n \in \omega} d_n \in D$ exists.

Definition A.5 (Lifted set)**.** Let A be a set. The symbol $A_\bot$ denotes the ω-cpo whose elements $A \cup \{\bot\}$ are ordered by $x \sqsubseteq y$, if and only if, $x = \bot$ or $x = y$ [Mitchell 1996]. The ω-cpo $A_\bot$ is called A lifted.



(a) The $\mathbf{B}_\perp$ ω-cpo.

(b) The $\mathbf{N}_\perp$ ω-cpo.

Fig. A.1: Lifted sets.

Example A.6. The lifted Booleans $\mathbf{B}_\perp$ and the lifted natural numbers $\mathbf{N}_\perp$ are depicted in Fig. A.1a and Fig. A.1b, respectively.

Example A.7. The ω-cpo **LN** of lazy natural numbers arises from a non-strict successor function, that is, $\mathsf{succ}(\perp) \neq \perp$. The partial ordering on **LN** is depicted in Fig. A.2, where $\underline{\mathbf{0}} = \perp$, $\underline{\mathbf{n}+1} = \mathsf{succ}(\underline{\mathbf{n}})$ and $\infty = \bigsqcup_{n\in\omega} \underline{\mathbf{n}}$ (see, for example, Escardó [1993]).

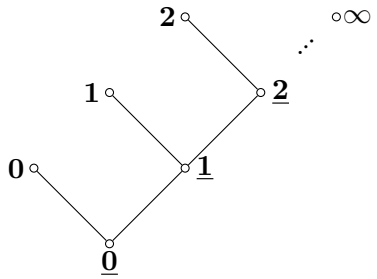


Fig. A.2: The **LN** ω-cpo.

Definition A.8 (Continuous function)**.** Let $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ be two ω-cpos. A function $f : D \to D'$ is continuous if [Plotkin 1992]

1. The function is monotone.
2. The function preserves the least upper bounds of the ω-chains, that is,

$$\bigsqcup_{n\in\omega} f(d_n) = f\left(\bigsqcup_{n\in\omega} d_n\right), \quad \text{for all } \omega\text{-chains } d_0 \sqsubseteq d_1 \sqsubseteq \cdots \sqsubseteq d_n \sqsubseteq \cdots$$

Definition A.9 (Function space of continuous functions)**.** Let $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ be two ω-cpos. The function space of continuous functions is the set [Winskel 1994]

$$[D \to D'] = \{f : D \to D' \mid f \text{ is continuous}\}.$$

This set can be partially ordered point-wise by

$$f \sqsubseteq g \stackrel{\text{def}}{=} \forall d \in D.\ f(d) \sqsubseteq' g(d),$$

and its bottom element is $\lambda x.\bot_{D'}$. The function space $[D \to D']$ is an ω-cpo.

Theorem A.10 (The fixed-point theorem)**.** *Let $(D, \sqsubseteq)$ be an ω-cpo. Given $f \in [D \to D]$, then*

$$\mathrm{Fix}(f) = \bigsqcup_{n \in \omega} f^n(\bot),$$

is the least fixed-point of f [Winskel 1994], that is,

$$\begin{aligned} \forall d.\ f(d) \sqsubseteq d \supset \mathrm{Fix}(f) &\sqsubseteq d, \\ f(\mathrm{Fix}(f)) &= \mathrm{Fix}(f). \end{aligned}$$

Definition A.11 (Coalesced sum)**.** Let $\mathbf{D}_1 = (D_1, \sqsubseteq_1), \ldots, \mathbf{D}_n = (D_n, \sqsubseteq_n)$ be ω-cpos. The coalesced sum—the disjoint union with bottom elements identified—$\mathbf{D}_1 \oplus \cdots \oplus \mathbf{D}_n$ is the ω-cpo [Plotkin 1992]

$$\left(\bigcup_{i \le n} \{(i, d) \mid d \in D_i \wedge d \neq \bot\} \right) \cup \bot$$

with the order

$$x \sqsubseteq y \stackrel{\text{def}}{=} x = \bot \text{ or } \exists i \le n. \exists d, d' \in D_i.\ d \sqsubseteq_i d' \wedge x = (i, d) \wedge y = (i, d').$$

Associated with the coalesced sum are the injection functions

$$in_i : \mathbf{D}_i \to \mathbf{D}_1 \oplus \cdots \oplus \mathbf{D}_n$$

defined by

$$in_i(d) = \begin{cases} \bot & \text{if } d = \bot, \\ (i, d) & \text{otherwise.} \end{cases}$$

Appendix B

Two Induction Principles for `N` and their Equivalence

In remark 4.9, we showed two induction principles for the inductive predicate `N`. Here, we present the proof of the equivalence of these inductive principles.

```
-- The inductive predicate for the total and finite natural
-- numbers.
data N : D → Set where
  nzero : N zero
  nsucc : ∀ {n} → N n → N (succ₁ n)


-- The induction principle using the hypothesis N n.
N-ind₁ : (A : D → Set) →
         A zero →
         (∀ {n} → N n → A n → A (succ₁ n)) →
         ∀ {n} → N n → A n
N-ind₁ A A0 h nzero      = A0
N-ind₁ A A0 h (nsucc Nn) = h Nn (N-ind₁ A A0 h Nn)


-- The induction principle without using the hypothesis N n.
N-ind₂ : (A : D → Set) →
          A zero →
         (∀ {n} → A n → A (succ₁ n)) →
         ∀ {n} → N n → A n
N-ind₂ A A0 h nzero      = A0
N-ind₂ A A0 h (nsucc Nn) = h (N-ind₂ A A0 h Nn)
```

```
-- N-ind₂ from N-ind₁.
N-ind₂' : (A : D → Set) →
          A zero →
          (∀ {n} → A n → A (succ₁ n)) →
          ∀ {n} → N n → A n
N-ind₂' A A0 h = N-ind₁ A A0 (λ _ → h)

-- N-ind₁ from N-ind₂.
N-ind₁' : (A : D → Set) →
          A zero →
          (∀ {n} → N n → A n → A (succ₁ n)) →
          ∀ {n} → N n → A n
N-ind₁' A A0 h {n} Nn = ∧-proj₂ (N-ind₂ B B0 h' Nn)
  where
  B : D → Set
  B n = N n ∧ A n

  B0 : B zero
  B0 = nzero , A0

  h' : ∀ {m} → B m → B (succ₁ m)
  h' (Nm , Am) = nsucc Nm , h Nm Am
```

Appendix C

Streams Properties

In Example 5.31, we described the proof that the length of a stream is ∞. Here, we present this proof.

```
lengthCong : ∀ {xs ys} → xs ≡ ys → length xs ≡ length ys
lengthCong refl = refl

streamLength : ∀ {xs} → Stream xs → length xs ≈N ∞
streamLength {xs} Sxs = ≈N-coind R h₁ h₂
  where
  R : D → D → Set
  R m n = ∃[ xs ] Stream xs ∧ m ≡ length xs ∧ n ≡ ∞

  h₁ : ∀ {m n} → R m n →
       m ≡ zero ∧ n ≡ zero
         ∨ (∃[ m' ] ∃[ n' ] m ≡ succ₁ m' ∧ n ≡ succ₁ n' ∧ R m' n')
  h₁ {m} {n} (xs , Sxs , m=lxs , n≡∞) = helper₁ (Stream-out Sxs)
    where
    helper₁ : (∃[ x' ] ∃[ xs' ] xs ≡ x' ∷ xs' ∧ Stream xs') →
              m ≡ zero ∧ n ≡ zero
                ∨ (∃[ m' ] ∃[ n' ]
                     m ≡ succ₁ m' ∧ n ≡ succ₁ n' ∧ R m' n')
    helper₁ (x' , xs' , xs≡x'∷xs' , Sxs') =
      inj₂ (length xs'
           , ∞
           , helper₂
           , trans n≡∞ ∞-eq
           , (xs' , Sxs' , refl , refl))
        where
        helper₂ : m ≡ succ₁ (length xs')
        helper₂ =
          trans m=lxs (trans (lengthCong xs≡x'∷xs') (length-∷ x' xs'))
```

```
h₂ : R (length xs) ∞
h₂ = xs , Sxs , refl , refl
```

Appendix D

The `mirror` Function: Proofs of Some Properties

We present here the proofs of some properties using our combined proof approach. These properties are used in the correctness proof of the `mirror` function described in § 7.2.

```
postulate reverse-[x]≡[x] : ∀ x → reverse (x ∷ []) ≡ x ∷ []
{-# ATP prove reverse-[x]≡[x] #-}

++-rightIdentity : ∀ {xs} → Forest xs → xs ++ [] ≡ xs
++-rightIdentity fnil = ++-leftIdentity []
++-rightIdentity (fcons {x} {xs} Tx Fxs) =
  prf (++-rightIdentity Fxs)
  where postulate prf : xs ++ [] ≡ xs → (x ∷ xs) ++ [] ≡ x ∷ xs
        {-# ATP prove prf #-}

rev-Forest : ∀ {xs ys} → Forest xs → Forest ys →
             Forest (rev xs ys)
rev-Forest {ys = ys} fnil Fys = prf
  where postulate prf : Forest (rev [] ys)
        {-# ATP prove prf #-}
rev-Forest {ys = ys} (fcons {x} {xs} Tx Fxs) Fys =
  prf (rev-Forest Fxs (fcons Tx Fys))
  where postulate prf : Forest (rev xs (x ∷ ys)) →
                        Forest (rev (x ∷ xs) ys)
        {-# ATP prove prf #-}

postulate
  reverse-Forest : ∀ {xs} → Forest xs → Forest (reverse xs)
{-# ATP prove reverse-Forest rev-Forest #-}
```

```
reverse-++ : ∀ {xs ys} → Forest xs → Forest ys →
             reverse (xs ++ ys) ≡ reverse ys ++ reverse xs
reverse-++ {ys = ys} fnil Fys = prf
  where
  postulate prf : reverse ([] ++ ys) ≡ reverse ys ++ reverse []
  {-# ATP prove prf ++-rightIdentity reverse-Forest #-}

reverse-∷ : ∀ {x ys} → Tree x → Forest ys →
            reverse (x ∷ ys) ≡ reverse ys ++ (x ∷ [])
reverse-∷ {x} Tx fnil = prf
  where postulate prf : reverse (x ∷ []) ≡ reverse [] ++ x ∷ []
        {-# ATP prove prf #-}

reverse-∷ {x} Tx (fcons {y} {ys} Ty Fys) = prf
  where
  postulate prf : reverse (x ∷ y ∷ ys) ≡
                  reverse (y ∷ ys) ++ x ∷ []
  {-# ATP prove prf reverse-[x]≡[x] reverse-++ #-}
```

Appendix E

The Alternating Bit Protocol Written in Haskell

This program is an adaptation of the Miranda [Turner 1986] program written in [Dybjer and Sander 1989]. Although Miranda's and Haskell's list data types allow the definitions of potentially infinite lists, we use the `Stream` data type from the streams library [Kmett 2014] to avoid the warning `NonExhaustivePatternMatch` generated by GHC [The GHC Development Team 2014] given that we do not pattern match on the empty list.

```
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE UnicodeSyntax       #-}

module Main where

import Data.Stream.Infinite ( Stream( (:>) ) )

type Bit = Bool

-- Data type used to model possible corrupted messages.
data Err a = Error | Ok a

-- The mutual sender functions.
sendH :: Bit → Stream a → Stream (Err Bit) → Stream (a, Bit)
sendH b input@(i :> _) ds = (i , b) :> awaitH b input ds

awaitH :: Bit → Stream a → Stream (Err Bit) → Stream (a, Bit)
awaitH b input@(i :> is) (Ok b' :> ds) =
  if b == b'
  then sendH (not b) is ds
  else (i, b) :> awaitH b input ds
awaitH b input@(i :> _) (Error :> ds) = (i, b) :> awaitH b input ds

-- The receiver functions.
```

```
ackH :: Bit → Stream (Err (a, Bit)) → Stream Bit
ackH b (Ok (_, b') :> bs) =
  if b == b' then b :> ackH (not b) bs else not b :> ackH b bs
ackH b (Error :> bs) = not b :> ackH b bs

outH :: Bit → Stream (Err (a, Bit)) → Stream a
outH b (Ok (i, b') :> bs) =
  if b == b' then i :> outH (not b) bs else outH b bs
outH b (Error :> bs) = outH b bs

-- The fair unreliable transmission channel.
corruptH :: Stream Bit → Stream a → Stream (Err a)
corruptH (False :> os) (_ :> xs) = Error :> corruptH os xs
corruptH (True :> os)  (x :> xs) = Ok x  :> corruptH os xs

-- The ABP transfer function.
abpTransH :: ∀ a. Bit → Stream Bit → Stream Bit → Stream a → Stream a
abpTransH b os1 os2 is = js
  where
  as :: Stream (a, Bit)
  as = sendH b is ds

  bs :: Stream (Err (a, Bit))
  bs = corruptH os1 as

  cs :: Stream Bit
  cs = ackH b bs

  ds :: Stream (Err Bit)
  ds = corruptH os2 cs

  js :: Stream a
  js = outH b bs
```

Appendix F

The Alternating Bit Protocol: Proofs of Some Properties

We present here the proofs of the lemmas `lemma₁` and `lemma₂` using our combined proof approach. Both lemmas are used in the correctness proof of the alternating bit protocol as described in § 7.4.6.

F.1 Properties Required by the Lemmas

The following properties related to the inductive predicate `Bool` and the co-inductive predicate `Fair` are required by the lemmas. All the properties are proved using our combined proof approach.

```
x≢not-x : ∀ {b} → Bool b → b ≢ not b
x≢not-x btrue h = prf
  where postulate prf : ⊥
        {-# ATP prove prf #-}
x≢not-x bfalse h = prf
  where postulate prf : ⊥
        {-# ATP prove prf #-}
```

```
not-x≢x : ∀ {b} → Bool b → not b ≢ b
not-x≢x Bb h = prf
  where postulate prf : ⊥
        {-# ATP prove prf x≢not-x #-}
```

```
not-involutive : ∀ {b} → Bool b → not (not b) ≡ b
not-involutive btrue = prf
  where postulate prf : not (not true) ≡ true
        {-# ATP prove prf #-}
not-involutive bfalse = prf
  where postulate prf : not (not false) ≡ false
        {-# ATP prove prf #-}

head-tail-Fair : ∀ {os} → Fair os →
                 os ≡ T ∷ tail₁ os ∨ os ≡ F ∷ tail₁ os
head-tail-Fair {os} Fos with Fair-out Fos
... | (.(T ∷ []) , os' , f*tnil , h , Fos') = prf
  where
  postulate prf : os ≡ T ∷ tail₁ os ∨ os ≡ F ∷ tail₁ os
  {-# ATP prove prf #-}

... | (.(F ∷ ft) , os' , f*tcons {ft} FTft , h , Fos') = prf
  where
  postulate prf : os ≡ T ∷ tail₁ os ∨ os ≡ F ∷ tail₁ os
  {-# ATP prove prf #-}.

tail-Fair : ∀ {os} → Fair os → Fair (tail₁ os)
tail-Fair {os} Fos with Fair-out Fos
... | .(T ∷ []) , os' , f*tnil , h , Fos' = prf
  where
  postulate prf : Fair (tail₁ os)
  {-# ATP prove prf #-}
... | .(F ∷ ft) , os' , f*tcons {ft} FTft , h , Fos' = prf
  where
  postulate prf : Fair (tail₁ os)
  {-# ATP prove prf Fair-in #-}.
```

F.2 First Lemma

```
-- Auxiliary definitions.

as^ : ∀ b i' is' ds → D
as^ b i' is' ds = await b i' is' ds
{-# ATP definition as^ #-}

bs^ : D → D → D → D → D → D
bs^ b i' is' ds os₁^ = corrupt os₁^ · (as^ b i' is' ds)
{-# ATP definition bs^ #-}

cs^ : D → D → D → D → D → D
cs^ b i' is' ds os₁^ = ack b · (bs^ b i' is' ds os₁^)
{-# ATP definition cs^ #-}

ds^ : D → D → D → D → D → D → D
ds^ b i' is' ds os₁^ os₂^ = corrupt os₂^ · cs^ b i' is' ds os₁^
{-# ATP definition ds^ #-}
```

```
os₁^ : D → D → D
os₁^ os₁' ft₁^ = ft₁^ ++ os₁'
{-# ATP definition os₁^ #-}

os₂^ : D → D
os₂^ os₂ = tail₁ os₂
{-# ATP definition os₂^ #-}

-- Helper function for Lemma 1.
helper₂ :
  ∀ {b i' is' os₁ os₂ as bs cs ds js} →
  Bit b →
  Fair os₂ →
  S b (i' ∷ is') os₁ os₂ as bs cs ds js →
  ∀ ft₁ os₁' → F*T ft₁ → Fair os₁' → os₁ ≡ ft₁ ++ os₁' →
  ∃[ os₁' ] ∃[ os₂' ] ∃[ as' ] ∃[ bs' ] ∃[ cs' ] ∃[ ds' ] ∃[ js' ]
    Fair os₁'
    ∧ Fair os₂'
    ∧ S' b i' is' os₁' os₂' as' bs' cs' ds' js'
    ∧ js ≡ i' ∷ js'
helper₂ {b} {i'} {is'} {js = js}
        Bb Fos₂ s .(T ∷ []) os₁' f*tnil Fos₁' os₁-eq = prf
  where
  postulate
    prf :
      ∃[ os₁' ] ∃[ os₂' ] ∃[ as' ] ∃[ bs' ] ∃[ cs' ] ∃[ ds' ] ∃[ js' ]
        Fair os₁'
          ∧ Fair os₂'
          ∧ (as' ≡ await b i' is' ds'
            ∧ bs' ≡ corrupt os₁' · as'
            ∧ cs' ≡ ack (not b) · bs'
            ∧ ds' ≡ corrupt os₂' · (b ∷ cs')
            ∧ js' ≡ out (not b) · bs')
          ∧ js ≡ i' ∷ js'
  {-# ATP prove prf #-}

helper₂ {b} {i'} {is'} {os₁} {os₂} {as} {bs} {cs} {ds} {js} Bb Fos₂ s
       .(F ∷ ft₁^) os₁' (f*tcons {ft₁^} FTft₁^) Fos₁' os₁-eq =
  helper₂ Bb (tail-Fair Fos₂) ihS ft₁^ os₁' FTft₁^ Fos₁' refl
  where
  postulate os₁-eq-helper : os₁ ≡ F ∷ os₁^ os₁' ft₁^
  {-# ATP prove os₁-eq-helper #-}

  postulate as-eq : as ≡ < i' , b > ∷ (as^ b i' is' ds)
  {-# ATP prove as-eq #-}

  postulate bs-eq : bs ≡ error ∷ (bs^ b i' is' ds (os₁^ os₁' ft₁^))
  {-# ATP prove bs-eq os₁-eq-helper as-eq #-}

  postulate cs-eq : cs ≡ not b ∷ cs^ b i' is' ds (os₁^ os₁' ft₁^)
```

```
{-# ATP prove cs-eq bs-eq #-}

postulate
  ds-eq :
    ds ≡ ok (not b) ∷ ds^ b i' is' ds (os₁^ os₁' ft₁^) (os₂^ os₂)
      ∨ ds ≡ error ∷ ds^ b i' is' ds (os₁^ os₁' ft₁^) (os₂^ os₂)
{-# ATP prove ds-eq head-tail-Fair cs-eq #-}

postulate
  as^-eq-helper₁ :
    ds ≡ ok (not b) ∷ ds^ b i' is' ds (os₁^ os₁' ft₁^) (os₂^ os₂) →
    as^ b i' is' ds ≡
      send b · (i' ∷ is')
        · ds^ b i' is' ds (os₁^ os₁' ft₁^) (os₂^ os₂)
{-# ATP prove as^-eq-helper₁ x≢not-x #-}

postulate
  as^-eq-helper₂ :
    ds ≡ error ∷ ds^ b i' is' ds (os₁^ os₁' ft₁^) (os₂^ os₂) →
    as^ b i' is' ds ≡
      send b · (i' ∷ is')
        · ds^ b i' is' ds (os₁^ os₁' ft₁^) (os₂^ os₂)
{-# ATP prove as^-eq-helper₂ #-}

as^-eq : as^ b i' is' ds ≡
         send b · (i' ∷ is') · ds^ b i' is' ds (os₁^ os₁' ft₁^)
           (os₂^ os₂)
as^-eq = case as^-eq-helper₁ as^-eq-helper₂ ds-eq

postulate js-eq : js ≡ out b · bs^ b i' is' ds (os₁^ os₁' ft₁^)
{-# ATP prove js-eq bs-eq #-}

ihS : S b
        (i' ∷ is')
        (os₁^ os₁' ft₁^)
        (os₂^ os₂)
        (as^ b i' is' ds)
        (bs^ b i' is' ds (os₁^ os₁' ft₁^))
        (cs^ b i' is' ds (os₁^ os₁' ft₁^))
        (ds^ b i' is' ds (os₁^ os₁' ft₁^) (os₂^ os₂))
        js
ihS = as^-eq , refl , refl , refl , js-eq
```

```
-- Lemma 1.
lemma₁ :
  ∀ {b i' is' os₁ os₂ as bs cs ds js} →
  Bit b →
  Fair os₁ →
  Fair os₂ →
  S b (i' ∷ is') os₁ os₂ as bs cs ds js →
  ∃[ os₁' ] ∃[ os₂' ] ∃[ as' ] ∃[ bs' ] ∃[ cs' ] ∃[ ds' ] ∃[ js' ]
    Fair os₁'
    ∧ Fair os₂'
    ∧ S' b i' is' os₁' os₂' as' bs' cs' ds' js'
    ∧ js ≡ i' ∷ js'
lemma₁ {b} {i'} {is'} {os₁} {js = js} Bb Fos₁ Fos₂ s =
  helper₁ (Fair-out Fos₁)
  where
  helper₁ :
    (∃[ ft ] ∃[ os₁' ] F*T ft ∧ os₁ ≡ ft ++ os₁' ∧ Fair os₁') →
    ∃[ os₁' ] ∃[ os₂' ] ∃[ as' ] ∃[ bs' ] ∃[ cs' ] ∃[ ds' ] ∃[ js' ]
      Fair os₁'
      ∧ Fair os₂'
      ∧ S' b i' is' os₁' os₂' as' bs' cs' ds' js'
      ∧ js ≡ i' ∷ js'
  helper₁ (ft , os₁' , FTft , os₁-eq ,  Fos₁') =
    helper₂ Bb Fos₂ s ft os₁' FTft Fos₁' os₁-eq
```

F.3 Second Lemma

```
-- Auxiliary definitions.

ds^ : D → D → D
ds^ cs' os₂^ = corrupt os₂^ · cs'
{-# ATP definition ds^ #-}

as^ : D → D → D → D → D → D
as^ b i' is' cs' os₂^ = await b i' is' (ds^ cs' os₂^)
{-# ATP definition as^ #-}

bs^ : D → D → D → D → D → D → D
bs^ b i' is' cs' os₁^ os₂^ = corrupt os₁^ · as^ b i' is' cs' os₂^
{-# ATP definition bs^ #-}

cs^ : D → D → D → D → D → D → D
cs^ b i' is' cs' os₁^ os₂^ = ack (not b) · bs^ b i' is' cs' os₁^ os₂^
{-# ATP definition cs^ #-}

os₁^ : D → D
os₁^ os₁' = tail₁ os₁'
{-# ATP definition os₁^ #-}
```

```
os₂^ : D → D → D
os₂^ ft₂ os₂'' = ft₂ ++ os₂''
{-# ATP definition os₂^ #-}

-- Helper function for Lemma 2.
helper₂ :
  ∀ {b i' is' os₁' os₂' as' bs' cs' ds' js'} →
  Bit b →
  Fair os₁' →
  S' b i' is' os₁' os₂' as' bs' cs' ds' js' →
  ∀ ft₂ os₂'' → F*T ft₂ → Fair os₂'' → os₂' ≡ ft₂ ++ os₂'' →
  ∃[ os₁'' ] ∃[ os₂'' ] ∃[ as'' ] ∃[ bs'' ] ∃[ cs'' ] ∃[ ds'' ]
    Fair os₁''
    ∧ Fair os₂''
    ∧ S (not b) is' os₁'' os₂'' as'' bs'' cs'' ds'' js'
helper₂ {b} {i'} {is'} {js' = js'} Bb Fos₁' s'
        .(T :: []) os₂'' f*tnil Fos₂'' os₂'-eq = prf
  where
  postulate
    prf :
      ∃[ os₁'' ] ∃[ os₂'' ] ∃[ as'' ] ∃[ bs'' ] ∃[ cs'' ] ∃[ ds'' ]
        Fair os₁''
        ∧ Fair os₂''
        ∧ as'' ≡ send (not b) · is' · ds''
        ∧ bs'' ≡ corrupt os₁'' · as''
        ∧ cs'' ≡ ack (not b) · bs''
        ∧ ds'' ≡ corrupt os₂'' · cs''
        ∧ js'  ≡ out (not b) · bs''
  {-# ATP prove prf #-}

helper₂ {b} {i'} {is'} {os₁'} {os₂'} {as'} {bs'} {cs'} {ds'} {js'}
        Bb Fos₁' s'
        .(F :: ft₂) os₂'' (f*tcons {ft₂} FTft₂) Fos₂'' os₂'-eq =
        helper₂ Bb (tail-Fair Fos₁') ihS' ft₂ os₂'' FTft₂ Fos₂'' refl
  where
  postulate os₂'-eq-helper : os₂' ≡ F :: os₂^ ft₂ os₂''
  {-# ATP prove os₂'-eq-helper #-}

  postulate ds'-eq : ds' ≡ error :: ds^ cs' (os₂^ ft₂ os₂'')
  {-# ATP prove ds'-eq os₂'-eq-helper #-}

  postulate
    as'-eq : as' ≡ < i' , b > :: as^ b i' is' cs' (os₂^ ft₂ os₂'')
  {-# ATP prove as'-eq #-}

  postulate
    bs'-eq :
      bs' ≡ ok < i' , b > :: bs^ b i' is' cs' (os₁^ os₁') (os₂^ ft₂ os₂'')
        ∨ bs' ≡ error :: bs^ b i' is' cs' (os₁^ os₁') (os₂^ ft₂ os₂'')
  {-# ATP prove bs'-eq as'-eq head-tail-Fair #-}
```

```
postulate
  cs'-eq-helper₁ :
    bs' ≡ ok < i' , b > ∷ bs^ b i' is' cs' (os₁^ os₁') (os₂^ ft₂ os₂'') →
    cs' ≡ b ∷ cs^ b i' is' cs' (os₁^ os₁') (os₂^ ft₂ os₂'')
{-# ATP prove cs'-eq-helper₁ not-x≢x not-involutive #-}

postulate
  cs'-eq-helper₂ :
    bs' ≡ error ∷ bs^ b i' is' cs' (os₁^ os₁') (os₂^ ft₂ os₂'') →
    cs' ≡ b ∷ cs^ b i' is' cs' (os₁^ os₁') (os₂^ ft₂ os₂'')
{-# ATP prove cs'-eq-helper₂ not-involutive #-}

cs'-eq : cs' ≡ b ∷ cs^ b i' is' cs' (os₁^ os₁') (os₂^ ft₂ os₂'')
cs'-eq = case cs'-eq-helper₁ cs'-eq-helper₂ bs'-eq

postulate
  js'-eq :
    js' ≡ out (not b) · bs^ b i' is' cs' (os₁^ os₁') (os₂^ ft₂ os₂'')
{-# ATP prove js'-eq not-x≢x bs'-eq #-}

postulate
  ds^-eq : ds^ cs' (os₂^ ft₂ os₂'') ≡
           corrupt (os₂^ ft₂ os₂'') ·
             (b ∷ cs^ b i' is' cs' (os₁^ os₁') (os₂^ ft₂ os₂''))
{-# ATP prove ds^-eq cs'-eq #-}

ihS' : S' b i' is'
          (os₁^ os₁')
          (os₂^ ft₂ os₂'')
          (as^ b i' is' cs' (os₂^ ft₂ os₂''))
          (bs^ b i' is' cs' (os₁^ os₁') (os₂^ ft₂ os₂''))
          (cs^ b i' is' cs' (os₁^ os₁') (os₂^ ft₂ os₂''))
          (ds^ cs' (os₂^ ft₂ os₂''))
          js'
ihS' = refl , refl , refl , ds^-eq , js'-eq
```

```
-- Lemma 2.
lemma₂ :
  ∀ {b i' is' os₁' os₂' as' bs' cs' ds' js'} →
  Bit b →
  Fair os₁' →
  Fair os₂' →
  S' b i' is' os₁' os₂' as' bs' cs' ds' js' →
  ∃[ os₁'' ] ∃[ os₂'' ] ∃[ as'' ] ∃[ bs'' ] ∃[ cs'' ] ∃[ ds'' ]
    Fair os₁''
    ∧ Fair os₂''
    ∧ S (not b) is' os₁'' os₂'' as'' bs'' cs'' ds'' js'
lemma₂ {b} {is' = is'} {os₂' = os₂'} {js' = js'} Bb Fos₁' Fos₂' s' =
  helper₁ (Fair-out Fos₂')
  where
  helper₁ :
    (∃[ ft₂ ] ∃[ os₂'' ] F*T ft₂ ∧ os₂' ≡ ft₂ ++ os₂'' ∧ Fair os₂'') →
    ∃[ os₁'' ] ∃[ os₂'' ] ∃[ as'' ] ∃[ bs'' ] ∃[ cs'' ] ∃[ ds'' ]
      Fair os₁''
      ∧ Fair os₂''
      ∧ S (not b) is' os₁'' os₂'' as'' bs'' cs'' ds'' js'
  helper₁ (ft₂ , os₂'' , FTft₂ , os₂'-eq , Fos₂'') =
    helper₂ Bb Fos₁' s' ft₂ os₂'' FTft₂ Fos₂'' os₂'-eq
```

Bibliography

Abel, A. (2009). An Introduction to Dependent Types and Agda. URL: http://www2.tcs.ifi.lmu.de/~abel/DepTypes.pdf (visited on 29/07/2014) (cit. on p. 11).

Abel, A. and Altenkirch, T. (2002). A Predicative Analysis of Structural Recursion. Journal of Functional Programming 12.1, pp. 1–41 (cit. on p. 18).

Abel, A., Benke, M., Bove, A., Claessen, K., Coquand, C., Coquand, T., Danielsson, N. A., Dybjer, P., Hamon, G., Hughes, J., Lindblad, F., Jansson, P., Norell, U. and Sheeran, M. (2003). The CoVer (Combining Verification Methods in Software Development) Project. See Coquand, Dybjer, Hughes and Sheeran [2001]. Chalmers University of Technology (cit. on p. 4).

Abel, A., Benke, M., Bove, A., Hughes, J. and Norell, U. (2005). Verifying Haskell Programs Using Constructive Type Theory. In: Proceedings of the ACM SIGPLAN 2005 Haskell Workshop, pp. 62–73 (cit. on p. 6).

Abel, A., Coquand, T. and Norell, U. (2005). Connecting a Logical Framework to a First-Order Logic Prover. In: Frontiers of Combining Systems (FroCoS 2005). Ed. by Gramlich, B. Vol. 3717. Lecture Notes in Artifical Intellingence. Springer, pp. 285–301 (cit. on pp. 5, 108).

Abel, A., Pientka, B., Thibodeau, D. and Setzer, A. (2013). Copatterns: Programming Infinite Structures by Observations. In: Proceedings of the 40th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013), pp. 27–38 (cit. on p. 43).

Achten, P., van Eekelen, M., Koopam, P. and Morazán, M. T. (2010). Trends in *Trends in Functional Programming* 1999/2000 versus 2007/2008. Higher-Order Symbolic Computation 23.4, pp. 465–487 (cit. on p. 1).

Aczel, P. (1977a). An Introduction to Inductive Definitions. In: Handbook of Mathematical Logic. Ed. by Barwise, J. Vol. 90. Studies in Logic and the Foundations of Mathematics. Elsevier. Chap. C.7 (cit. on pp. 54, 87).

——— (1977b). The Strength of MartinLöf's Intuitionistic Type Theory with One Universe. In: Proceedings of the Symposium on Mathematical Logic (Oulu, 1974). Ed. by Miettinen, S. and Väänanen, J. Report No. 2, Department of Philosophy, University of Helsinki, Helsinki, pp. 1–32 (cit. on pp. 4, 49, 141).

——— (1980). Frege Structures and the Notions of Proposition, Truth and Set. In: The Kleene Symposium. Ed. by Barwise, J., Keisler, H. J. and Kunen, K. Vol. 101. Studies in Logic and the Foundations of Mathematics. North-Holland Publishing Company, pp. 31–59 (cit. on p. 49).

——— (1989). What Might the Objects of the Logical Theory of Constructions be? In: Proceedings of the Worshop on Programming Logic. Ed. by Dybjer, P.

et al. Programming Methodology Group Reports 54. Chalmers University of Technology, pp. 122–139 (cit. on p. 49).

Akbarpour, B. and Paulson, L. C. (2010). MetiTarski: An Automatic Theorem Prover for Real-Valued Special Functions. Journal of Automated Reasoning 44 (3 2010), pp. 175–205 (cit. on p. 141).

Altenkirch, T. and McBride, C. (2006). Towards Observational Type Theory. Manuscript, available online (cit. on p. 56).

Augustsson, L., Bencke, M., Coquand, C., Coquand, T., Hallgren, T. and Takeyama, M. (2004). Agda 1. URL: http://ocvs.cfv.jp/Agda/index.html (visited on 29/07/2014) (cit. on p. 4).

Barendregt, H. (2004). The Lambda Calculus. Its Syntax and Semantics. 2nd ed. Vol. 103. Studies in Logic and the Foundations of Mathematics. 6th impression. Elsevier (cit. on p. 53).

Barret, C., Sebastiani, R., Seshia, S. A. and Tinelli, C. (2009). Satisfiability Modulo Theories. In: Handbook of Satisfiability. Ed. by Biere, A., Heule, M., van Maaren, H. and Walsh, T. IOS Press. Chap. 26 (cit. on p. 6).

Bertot, Y. and Castéran, P. (2004). Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer (cit. on p. 58).

Bird, R. and Wadler, P. (1988). Introduction to Functional Programming. Prentice Hall International (cit. on p. 118).

Birkhoff, G. and Mac Lane, S. (1977). A Survey of Modern Algebra. 4th ed. Macmillan Publishing Co., Inc. (cit. on p. 34).

Blanchette, J. C. (2013). Relational Analysis of (Co)Inductive Predicates, (Co)Algebraic Datatypes, and (Co)Recursive Functions. Software Quality Journal 21.1, pp. 101–126 (cit. on p. 82).

Blanchette, J. C., Böhme, S. and Paulson, L. C. (2013). Extending Sledgehammer with SMT Solvers. Journal of Automated Reasoning 51.1, pp. 109–128 (cit. on p. 6).

Blanchette, J. C. and Paulson, L. C. (2013). Hammering Away. A User's Guide to Sledgehammer for Isabelle/HOL. Isabelle2013-2 Documentation. 5th Dec. 2013 (cit. on p. 6).

Bobot, F., Filliâtre, J.-C., Marché, C. and Paskevich, A. (2014). Let's verify this with Why3. International Journal on Software Tools for Technology Transfer. In press. DOI: 10.1007/s10009-014-0314-5 (cit. on p. 7).

Bove, A. and Dybjer, P. (2009). Dependent Types at Work. In: LerNet ALFA Summer School 2008. Ed. by Bove, A., Soares Barbosa, L., Pardo, A. and Sousa Pinto, J. Vol. 5520. Lecture Notes in Computer Science. Springer, pp. 57–99 (cit. on pp. 11, 12, 31).

Bove, A., Dybjer, P. and Sicard-Ramírez, A. (2009). Embedding a Logical Theory of Constructions in Agda. In: Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification (PLPV 2009), pp. 59–66 (cit. on p. vii).

——— (2012). Combining Interactive and Automatic Reasoning in First Order Theories of Functional Programs. In: Foundations of Software Science and Computation Structures (FoSSaCS 2012). Ed. by Birkedal, L. Vol. 7213. Lecture Notes in Computer Science. Springer, pp. 104–118 (cit. on pp. vii, 5).

Bove, A., Krauss, A. and Sozeau, M. (2012). Partiality and Recursion in Interactive Theorem Provers. An Overview. Accepted for publication at Mathematical Structures in Computer Science, special issue on DTP 2010 (cit. on pp. 2, 7).

Burstall, R. M. (1969). Proving Properties of Programs by Structural Induction. The Computer Journal 12.1, pp. 41–48 (cit. on p. 2).

Claessen, K. (2010a). Private Communication. Mar. 2010 (cit. on p. 108).

——— (2010b). Private Communication. 24th Apr. 2010 (cit. on p. 114).

——— (2011a). Private Communication with Peter Dybjer. 17th Mar. 2011 (cit. on p. 107).

——— (2011b). The Anatomy of Equinox – An Extensible Automated Reasoning Tool for First-Order Logic and Beyond (Talk Abstract). In: Automated Deduction (CADE-23). Ed. by Bjørner, N. and Sofronie-Stokkermans, V. Vol. 6803. Lecture Notes in Artifical Intellingence. Springer, pp. 1–3 (cit. on p. 94).

Claessen, K. and Hughes, J. (2000). QuickCheck: A Lightweight Tool for Random Testing of Haskell programs. In: Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming (ICFP 2000), pp. 268–279 (cit. on p. 4).

Claessen, K., Johansson, M., Rosén, D. and Smallbone, N. (2013). Automating Inductive Proofs Using Theory Exploration. In: Automated Deduction (CADE-24). Ed. by Bonacina, M. P. Vol. 7898. Lecture Notes in Artifical Intellingence. Springer, pp. 392–406 (cit. on p. 5).

Claessen, K., Smallbone, N. and Hughes, J. (2010). QuickSpec: Guessing Formal Specifications Using Testing. In: Tests and Proofs (TAP 2010). Ed. by Fraser, G. and Garfantini, G. Vol. 6143. Lecture Notes in Computer Science. Springer, pp. 6–21 (cit. on p. 5).

Coquand, T., Dybjer, P., Hughes, J. and Sheeran, M. (2001). Combining Verification Methods in Software Development. Proposal to SSF (Swedish Strategic Research Foundation). URL: http://www.cse.chalmers.se/~rjmh/SSF/ (visited on 29/07/2014) (cit. on p. 165).

Danielsson, N. A. (2010). Total Parser Combinators. In: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP 2010), pp. 285–296 (cit. on p. 43).

Danielsson, N. A. et al. (2014). The Agda Standard Library v0.8. 11th June 2014. URL: http://wiki.portal.chalmers.se/agda (visited on 29/07/2014) (cit. on p. 140).

de Mol, M. J. (2009). Reasoning About Functional Programs. Sparkle, a Proof Assistant for Clean. PhD thesis. Radboud University Nijmegen (cit. on p. 6).

de Mol, M., van Eekelen, M. and Plasmeijer, R. (2002). Theorem Proving for Functional Programmers. Sparkle: A Functional Theorem Prover. In: Implementation of Functional Languages. 13th International Workshop (IFL 2001). Ed. by Arts, T. and Mohnen, M. Vol. 2312. Lecture Notes in Computer Science. Springer, pp. 55–71 (cit. on p. 6).

——— (2008). Proving Properties of Lazy Functional Programs with Sparkle. In: Central European Functional Programming School (CEFP 2007). Ed. by Horváth, Z., Plasmeijer, R., Soós, A. and Zsók, V. Vol. 5161. Lecture Notes in Computer Science, pp. 41–86 (cit. on p. 6).

de Moura, L. and Bjørner, N. (2008). Z3: An Efficient SMT Solver. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008). Ed. by Ramakrishnan, C. R. and Rehof, J. Vol. 4963. Lecture Notes in Computer Science. Springer, pp. 337–340 (cit. on p. 141).

Diatchki, I., Hallgren, T., Harke, T., Harrison, B., Hook, J., Huffman, B., Jones, M. P., Kieburtz, R., Leslie, R., Matthews, J., Tolmach, A. and White, P.

(2001). The Programatica Project. Integrating Programming, Properties and Validation. URL: http://programatica.cs.pdx.edu/index.html (visited on 29/07/2014) (cit. on p. 6).

Dybjer, P. (1985). Program Verification in a Logical Theory of Constructions. In: Functional Programming Languages and Computer Architecture. Ed. by Jouannaud, J.-P. Vol. 201. Lecture Notes in Computer Science. Springer, pp. 334–349 (cit. on pp. iii, v, 4, 49, 50, 52).

——— (1988). Inductively Defined Sets in Martin-Löf's Set Theory. In: Proceedings of the Worshop on General Logic, Edinburgh, February 1987. Ed. by Avron, A., Harper, R., Honsell, F., Mason, I. and Plotkin, G. Vol. ECS-LFCS-88-52. LFCS Report Series. Departament of Computer Science, University of Edinburgh (cit. on p. 77).

——— (1990). Comparing Integrated and External Logics of Functional Programs. Science of Computer Programming 14, pp. 59–79 (cit. on pp. 4, 49).

——— (1991). Inductive Sets and Families in Martin-Löf's Type Theory and Their Set-Theoretic Semantics. In: Logical Frameworks. Ed. by Huet, G. and Plotkin, G. Cambridge University Press, pp. 280–306 (cit. on p. 54).

——— (2004). Computability via PCF. URL: http://www.cse.chalmers.se/edu/year/2012/course/DAT140_Types/ (visited on 29/07/2014) (cit. on p. 75).

Dybjer, P. and Sander, H. P. (1989). A Functional Programming Approach to the Specification and Verification of Concurrent Systems. Formal Aspects of Computing 1, pp. 303–319 (cit. on pp. 4, 77, 85, 106, 124–126, 128, 133, 135, 136, 155).

Escardó, M. H. (1993). On Lazy Natural Numbers with Applications to Computability Theory and Functional Programming. SIGACT News 24 (1 1993), pp. 61–67 (cit. on p. 146).

Filliâtre, J.-C. and Paskevich, A. (2013). Why3 — Where Programs Meet Provers. In: Programming Languages and Systems (ESOP 2013). Ed. by Felleisen, M. and Gardner, P. Vol. 7792. Lecture Notes in Computer Science. Springer, pp. 125–128 (cit. on p. 7).

Forsberg, F. N. and Setzer, A. (2010). Inductive-Inductive Definitions. In: Computer Science Logic (CSL 2010). Ed. by Dawar, A. and Helmut, V. Vol. 6247. Lecture Notes in Computer Science. Springer, pp. 454–468 (cit. on p. 43).

Gardner, P. (1993). A New Type Theory for Representing Logics. In: Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 1993). Ed. by Voronkov, A. Vol. 698. Lecture Notes in Artifical Intellingence. Springer, pp. 146–157 (cit. on p. 43).

——— (1995). Equivalences Between Logics and Their Representing Type Theories. Mathematical Structures in Computer Science 5.3, pp. 323–349 (cit. on p. 43).

Garillot, F. and Werner, B. (2007). Simple Types in Type Theory: Deep and Shallow Encodings. In: Theorem Proving in Higher Order Logics (TPHOLs 2007). Ed. by Schneider, K. and Brandt, J. Vol. 4732. Lecture Notes in Computer Science. Springer, pp. 368–382 (cit. on p. 29).

Gibbons, J. and Hutton, G. (2005). Proof Methods for Corecursive Programs. Fundamenta Informaticae XX, pp. 1–14 (cit. on pp. 85, 105).

Giménez, E. (1995). Codifying Guarded Definitions with Recursive Schemes. In: Types for Proofs and Programs (TYPES 1994). Ed. by Dybjer, P., Nordström, B. and Smith, J. Vol. 996. Lecture Notes in Computer Science. Springer, pp. 39–59 (cit. on p. 124).

Bibliography

Giménez, E. and Casterán, P. (2007). A Tutorial on [Co-]Inductive Types in Coq. URL: http://coq.inria.fr/documentation (visited on 29/07/2014) (cit. on p. 106).

Girard, J.-Y. (1990). Proofs and Types. Cambridge University Press (cit. on p. 47).

Gordon, A. D. (1995). An Tutorial in Co-induction and Functional Programming. In: 1994 Glasgow Workshop on Functional Programming. Ed. by Hammond, K., Turner, D. N. and Sansom, P. Workshops in Computing. Springer-Verlag, pp. 78–95 (cit. on pp. 2, 50, 85, 105).

Hájek, P. and Pudlák, P. (1998). Metamathematics of First-Order Arithmetic. 2nd printing. Springer (cit. on p. 37).

Harper, R. (2013). Practical Foundations for Programming Languages. Cambridge University Press (cit. on p. 47).

Harper, R., Honsell, F. and Plotkin, G. (1993). A Framework for Defining Logics. Journal of the ACM 40.1, pp. 143–184 (cit. on p. 3).

Harper, R. and Licata, D. R. (2007). Mechanizing Metatheory in a Logical Framework. Journal of Functional Programming 17.4 & 5, pp. 613–673 (cit. on p. 43).

Harrison, J. (1995). Inductive Definitions: Automation and Application. In: Higher Order Logic Theorem Proving and Its Applications. Ed. by Schubert, E. T., Windley, P. J. and Alves-Foss, J. Vol. 971. Lecture Notes in Computer Science. Springer, pp. 200–213 (cit. on p. 77).

Harrison, W. L. and Kieburtz, R. B. (2005). The Logic of Demand in Haskell. Journal of Functional Programming 15.5, pp. 837–891 (cit. on p. 6).

Hodges, W. (1993). Model Theory. Vol. 42. Encyclopedia of Mathematics and its Applications. Cambridge University Press (cit. on p. 33).

Hughes, J. (1989). Why Functional Programming Matters. The Computer Journal 32.2, pp. 98–107 (cit. on p. 1).

Hurd, J. (2003). First-Order Proof Tactics in Higher-Order Logic Theorem Provers. In: Proceedings of Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003). Ed. by Archer, M., Di Vito, B. and Muñoz, C. Vol. NASA/CP-2003-212448. NASA Technical Reports, pp. 56–68 (cit. on p. 94).

Hutton, G. (2007). Programming in Haskell. Cambridge University Press (cit. on p. 1).

Johansson, M. (2013). Theory Exploration for Interactive Theorem Proving. In: 4th International Workshop on Artificial Intelligence for Formal Methods (AI4FM 2013). Ed. by Grov, G., Maclean, E. and Freitas, L. (cit. on p. 5).

Kaufmann, M., Manolios, P. and Moore, J. S. (2000). Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers (cit. on p. 141).

Kieburtz, R. B. (2007). Programmed Strategies for Programm Verification. Electronic Notes in Theoretical Computer Science 174, pp. 3–38 (cit. on p. 6).

Kimmell, G., Stump, A., Eades III, H. D., Fu, P., Sheard, T., Weirich, S., Casinghino, C., Sjöberg, V., Collins, N. and Ahn, K. Y. (2012). Equational Reasoning About Programs with General Recursion and Call-by-value Semantics. In: Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification (PLPV 2012), pp. 15–26 (cit. on p. 142).

Kleene, S. C. (1952). Introduction to Metamathematics. North-Holland (cit. on pp. 47, 95).

Kmett, E. A. (2014). The Streams Library v3.2. URL: http://hackage.haskell.org/package/streams-3.2 (visited on 29/07/2014) (cit. on p. 155).

Bibliography

Knuth, D. E. (1997). The Art of Computer Programming. Vol. 2. Seminumerical Algorithms. Addison-Wesley Professional (cit. on p. 64).

Kovác, L. and Voronkov, A. (2013). First-Order Theorem Proving and Vampire. In: Computer Aided Verification (CAV 2013). Ed. by Sharygina, N. and Veith, H. Vol. 8044. Lecture Notes in Computer Science. Springer, pp. 1–35 (cit. on p. 94).

Krauss, A. (2010). Partial and Nested Recursive Function Definitions in Higher-order Logic. Journal of Automated Reasoning 44.4, pp. 303–336 (cit. on p. 5).

——— (2013). Defining Recursive Functions in Isabelle/HOL. Isabelle2013-2 Documentation (cit. on p. 5).

Kurosh, A. G. (1960). The Theory of Groups. 2nd ed. Vol. 1. Translated and edited by K. A. Hirsch. Chelsea Publising Company (cit. on p. 35).

Lagarias, J. C. (2012). The $3x+1$ Problem: An Annotated Bibliography, II (2000–2009) v6. URL: http://arxiv.org/abs/math/0608208v6 (visited on 29/07/2014) (cit. on p. 121).

Lee, C. S., Jones, N. D. and Ben-Amram, A. M. (2001). The Size-Change Principle for Program Termination. In: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2001), pp. 81–92 (cit. on p. 18).

Lindblad, F. and Benke, M. (2006). A Tool for Automated Theorem Proving in Agda. In: Types for Proofs and Programs (TYPES 2004). Ed. by Filliâtre, J.-C., Paulin-Mohring, C. and Werner, B. Vol. 3839. Lecture Notes in Computer Science. Springer, pp. 154–169 (cit. on p. 141).

Longley, J. and Pollack, R. (2004). Reasoning About CBV Functional Programs in Isabelle/HOL. In: Theorem Proving in Higher Order Logics (TPHOLs 2004). Ed. by Slind, K., Bunker, A. and Gopalakrishnan, G. Vol. 3223. Lecture Notes in Computer Science. Springer, pp. 201–216 (cit. on p. 142).

Mac Lane, S. and Birkhoff, G. (1999). Algebra. 3rd ed. AMS Chelsea Publishing (cit. on p. 34).

Machover, M. (1996). Set Theory, Logic and their Limitations. Cambridge University Press (cit. on p. 37).

MacQueen, D. B. (2009). Kahn Networks at the Dawn of Functional Programming. In: From Semantics to Computer Science. Essays in Honour of Gilles Kahn. Ed. by Bertot, Y., Huet, G., Lévy, J.-J. and Plotkin, G. Cambridge University Press. Chap. 5 (cit. on p. 124).

Magnusson, L. and Nordström, B. (1994). The ALF Proof Editor and its Proof Engine. In: Types for Proofs and Programs (TYPES 1993). Ed. by Barendregt, H. and Nipkow, T. Vol. 806. Lecture Notes in Computer Science. Springer, pp. 213–237 (cit. on pp. 11, 76).

Manna, Z. and McCarthy, J. (1969). Properties of Programs and Partial Function Logic. In: Machine Intelligence. Ed. by Meltzer, B. and Michie, D. Vol. 5. Edinburgh University Press, pp. 27–37 (cit. on p. 115).

Marché, C. (2014). Verification of the Functional Behavior of a Floating-Point Program: an Industrial Case Study. Science of Computer Programming. In press. DOI: 10.1016/j.scico.2014.04.003 (cit. on p. 7).

Martin-Löf, P. (1971). Hauptsatz for the Intuitionistic Theory of Iterated Inductive Definitions. In: Proceedings of the Second Scandinavian Logic Symposium. Ed. by Fenstad, J. E. Vol. 63. Studies in Logic and the Foundations of Mathematics. North-Holland Publishing Company, pp. 179–216 (cit. on p. 52).

——— (1984). Intuitionistic Type Theory. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980. Bibliopolis (cit. on p. 12).

——— (1998). An Intuitionistic Theory of Types. In: Twenty-five Years of Constructive Type Theory. Ed. by Sambin, G. and Smith, J. M. (Originally a 1972 preprint from the Department of Mathematics, University of Stockholm). Oxford University Press. Chap. 8 (cit. on p. 24).

Mendelson, E. (1997). Introduction to Mathematical Logic. 4th ed. Chapman & Hall (cit. on pp. 27, 57).

Meng, J. and Paulson, L. C. (2008). Translating Higher-Order Clauses to First-Order Clauses. Journal of Automated Reasoning 40.1, pp. 35–60 (cit. on p. 108).

Milner, R., Tofte, M., Harper, R. and MacQueen, D. (1997). The Definition of Standard ML (Revised). MIT Press (cit. on p. 7).

Mitchell, J. C. (1996). Foundations for Programming Languages. MIT Press (cit. on pp. 75, 145).

Moschovakis, Y. N. (1974). Elementary Induction on Abstract Structures. North-Holland Publishing Company (cit. on p. 77).

Mu, S.-C., Ko, H.-S. and Jansson, P. (2009). Algebra of Programming in Agda: Dependent Types for Relational Program Derivation. Journal of Functional Programming 19.5, pp. 545–579 (cit. on pp. 19, 20).

Nipkow, T., Paulson, L. C. and Wenzel, M. (2002). Isabelle/HOL. A Proof Assistant for Higher-Order Logic. Vol. 2283. Lecture Notes in Computer Science. Springer (cit. on p. 5).

Nordström, B., Petersson, K. and Smith, J. M. (1990). Programming in Martin-Löf's Type Theory. Oxford University Press (cit. on pp. 11, 22).

Nordström, B. and Smith, J. (1984). Propositions and Specifications of Programs in Martin-Löf's Type Theory. BIT 24, pp. 288–301 (cit. on p. 12).

Norell, U. (2007a). Private Communication. Sept. 2007 (cit. on p. 111).

——— (2007b). Towards a Practical Programming Language Based on Dependent Type Theory. PhD thesis. Department of Computer Science and Engineering. Chalmers University of Technology and University of Gothenburg (cit. on pp. 3, 12, 19, 46).

——— (2009). Dependently Typed Programming in Agda. In: Advanced Functional Programming (AFP 2008). Ed. by Koopman, P., Plasmeijer, R. and Swierstra, D. Vol. 5832. Lecture Notes in Computer Science. Springer, pp. 230–266 (cit. on p. 11).

Norell, U. and Abel, A. (2006). AgdaLight. URL: http://www.cse.chalmers.se/~ulfn/agdaLight/index.html (visited on 29/07/2014) (cit. on p. 5).

Otten, J. (2005). Clausal Connection-Based Theorem Proving in Intuitionistic First-Order Logic. In: Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2005). Ed. by Beckert, B. Vol. 3702. Lecture Notes in Artifical Intellingence. Springer, pp. 245–261 (cit. on p. 114).

Park, D. (1976). Finitess is mu-Ineffable. Theoretical Computer Science 3, pp. 173–181 (cit. on pp. 4, 77).

Paulson, L. C. (1994a). A Fixedpoint Approach to Implementing (Co)inductive Definitions. In: Automated Deduction (CADE-12). Ed. by Bundy, A. Vol. 814. Lecture Notes in Artifical Intellingence. Springer, pp. 148–161 (cit. on p. 82).

——— (1994b). Isabelle. A Generic Theorem Prover. Vol. 828. Lecture Notes in Computer Science. (With a contribution by T. Nipkow). Springer (cit. on p. 4).

Peyton Jones, S., ed. (2003). Haskell 98 Language and Libraries. The Revised Report. Cambridge University Press (cit. on p. 1).

Peyton Jones, S. L. (1987). The Implementation of Functional Programming Languages. Prentice-Hall International (cit. on pp. 1, 57, 72).

Pfenning, F. (2002). Logical Frameworks — A Brief Introduction. In: Proof and System-Realiability. Ed. by Schwichtenberg, H. and Steinbrüggen, R. Vol. 62. NATO Science Series II: Mathematics, Physics and Chemistry. Springer, pp. 137–166 (cit. on p. 21).

Pitts, A. M. (1994a). Computational Adequacy via 'Mixed' Inductive Definitions. In: Mathematical Foundations of Programming Semantics. Ed. by Brookes, S., Main, M., Melton, A., Mislove, M. and Schmidt, D. Vol. 802. Lecture Notes in Computer Science. Springer, pp. 72–82 (cit. on p. 53).

——— (1994b). Some Notes on Inductive and Co-Inductive Techniques in the Semantics of Functional programs. Draft Version. Tech. rep. NS-94-5. BRICS (cit. on p. 55).

Plasmeijer, R. and van Eekelen, M. (1999). Keep it Clean: A Unique Approach to Functional Programming. SIGPLAN Notices 34.6, pp. 23–31 (cit. on p. 6).

Plotkin, G. D. (1977). LCF Considered as a Programming Language. Theoretical Computer Science 5.3, pp. 223–255 (cit. on p. 2).

Plotkin, G. (1992). Post-graduate Lecture Notes in Advance Domain Theory (Incorporating the "Pisa Notes"). Electronic edition prepared by Yugo Kashiwagi and Hidetaka Kondoh. URL: http://homepages.inf.ed.ac.uk/gdp/ (visited on 29/07/2014) (cit. on pp. 53, 145–147).

Rosén, D. (2012). Proving Equational Haskell Properties Using Automated Theorem Provers. MA thesis. University of Gothenburg (cit. on p. 5).

Sander, H. P. (1992). A Logic of Functional Programs with an Application to Concurrency. PhD thesis. Department of Computer Sciences. Chalmers University of Technology and University of Gothenburg (cit. on pp. 89, 124, 125).

Sangiorgi, D. (2012). Introduction to Bisimulation and Coinduction. Cambridge University Press (cit. on p. 87).

Schmidt, D. A. (1986). Denotational Semantics. A Methodology for Language Development. Allyn and Bacon, Inc. (cit. on p. 79).

Schulz, S. (2013). System Description: E 1.8. In: Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-19). Ed. by McMillan, K., Middeldorp, A. and Voronkov, A. Vol. 8312. Lecture Notes in Computer Science. Springer, pp. 735–743 (cit. on p. 94).

Smith, J. (1978). On the Relation Between a Type Theoretic and a Logic Formulation of the Theory of Constructions. PhD thesis. Department of Mathematics. Chalmers University of Technology and University of Gothenburg (cit. on pp. 49, 141).

——— (1984). An Interpretation of Martin-Löf's Type Theory in a Type-Free Theory of Propositions. The Journal of Symbolic Logic 49.3, pp. 730–753 (cit. on pp. 49, 141).

Sonnex, W., Drossopoulou, S. and Eisenbach, S. (2012). Zeno: An Automated Prover for Properties of Recursive Data Structures. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2012). Ed. by Flanagan, C. and König, B. Vol. 7214. Lecture Notes in Computer Science. Springer, pp. 407–421 (cit. on p. 6).

Stallman, R. et al. (2012). GNU Emacs Manual. 16th. Updated for Emacs Version 24.1. Free Software Foundation, Inc (cit. on p. 11).

Stanovský, D. (2008). Distributive Groupoids are Symmetrical-by-Medial: An Elementary Proof. Commentations Mathematicae Universitatis Carolinae 49.4, pp. 541–546 (cit. on p. 96).

Sutcliffe, G. (2009). The TPTP Problem Library and Associated Infrastructure. The FOT and CNF Parts, v3.5.0. Journal of Automated Reasoning 43.4, pp. 337–362 (cit. on pp. 3, 107).

——— (2010). The TPTP World — Infrastructure for Automated Reasoning. In: Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16). Ed. by Clarke, E. and Voronkov, A. Vol. 6355. Lecture Notes in Computer Science. Springer, pp. 1–12 (cit. on pp. 92, 107).

——— (2013). The TPTP Problem Library. TPTP v6.0.0. URL: http://www.cs.miami.edu/~tptp/TPTP/TR/TPTPTR.shtml (visited on 29/07/2014) (cit. on pp. 92, 107).

Tammet, T. (1997). Gandalf. Journal of Automated Reasoning 18.2, pp. 199–204 (cit. on p. 5).

Tammet, T. and Smith, J. M. (1996). Optimized Encodings of Fragments of Type Theory in First Order Logic. In: Types for Proofs and Programs (TYPES 1995). Ed. by Berardi, S. and Coppo, M. Vol. 1158. Lecture Notes in Computer Science. Springer, pp. 265–287 (cit. on p. 5).

The Agda Development Team (2014). Agda 2.4.0.2. 29th July 2014. URL: http://wiki.portal.chalmers.se/agda/pmwiki.php (visited on 29/07/2014) (cit. on p. 3).

The Coq Development Team (2014). The Coq Proof Assistant. Reference Manual. Version 8.4pl4. 12th May 2014 (cit. on p. 58).

The GHC Development Team (2014). The Glorious Glasgow Haskell Compilation System v7.8.3. 11th July 2014 (cit. on p. 155).

Turner, D. A. (1995). Elementary Strong Functional Programming. In: Functional Programming Languages in Education (FPLE 1995). Ed. by Hartel, P. H. and Plasmeijer, R. Vol. 1022. Lecture Notes in Computer Science. Springer, pp. 1–13 (cit. on pp. 2, 106).

——— (2004). Total Functional Programming. Journal of Universal Computer Science 10.7, pp. 751–768 (cit. on p. 2).

Turner, D. (1986). An Overview of Miranda. SIGPLAN Notices 21.12, pp. 158–166 (cit. on pp. 124, 155).

van Dalen, D. (2004). Logic and Structure. 4th ed. Springer (cit. on pp. 24, 26, 107).

Wadler, P. (1987). A Critique of Abelson and Sussman or Why Calculating is Better Than Scheming. SIGPLAN Notices 22.3, pp. 83–94 (cit. on pp. 1, 2).

Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M. and Wischneski, P. (2009). SPASS Version 3.5. In: Automated Deduction (CADE-22). Ed. by Schmidt, R. A. Vol. 5663. Lecture Notes in Artifical Intellingence. Springer, pp. 140–145 (cit. on p. 94).

Winskel, G. (1994). The Formal Semantics of Programming Languages. An Introduction. 2nd printing. MIT Press (cit. on pp. 50, 55, 147).