UNIVERSITY
of
GLASGOW

Department of
Computing Science

Supporting Formal Reasoning about Functional Programs

Graham Richard McFarlane Collins

A dissertation submitted for the Doctor of Philosophy Degree at the University of Glasgow

January 2001

Abstract

It is often claimed that functional programming languages, and in particular pure functional languages, are suitable for formal reasoning. This claim is supported by the fact that many people in the functional programming community do reason about languages and programs in a formal or semi-formal way. Different reasoning principles, such as equational reasoning, induction and co-induction, are used, depending on the nature of the problem.

Using a computer program to check the application of rules and to mechanise the tedious bookkeeping involved can simplify proofs and provide more confidence in their correctness. When reasoning about programs, this can also allow experiments with new rules and reasoning styles, where a user may not be confident about structuring a proof on paper. Checking the applicability of a rule can eliminate the risk of mistakes caused by misunderstanding the theory being used. Just as there are different ways in which formal or informal reasoning can be applied in functional programming, there are different ways in which tools can be provided to support this reasoning.

This thesis describes an investigation of how to develop a mechanised reasoning system to allow reasoning about algorithms as a functional programmer would write them, not an encoding of the algorithm into a significantly different form. In addition, the work aims to develop a system to support a user who is not a theorem proving expert or an expert in the theoretical foundations of functional programming. The work is aimed towards a system that could be used by a functional programmer developing real programs and wishing to prove some or all of the programs correct or to prove that two programs are equivalent.

Acknowledgements

I would like to acknowledge three years of funding from the Engineering and Physical Sciences Research Council during the course of this work.

I would also like to thank the following people for their help and support over the last few years:

My supervisor, Professor Tom Melham, who provided all the support, criticism and advice necessary for me to complete this work, and the Department of Computing Science, especially the Formal Analysis, Theory and Algorithms research group, the Functional Programming research group, Simon Peyton Jones and John O'Donnel from my supervisory team. The wider theorem proving community, especially the Glasgow Provers, the Automated Reasoning Group at the University of Cambridge and the Prosper team. All the other researchers that have commented on my work or shared details of theirs, especially Andy Gordon, Andy Pitts and all the referees of my papers. All my office mates, fellow students and friends, especially Claire Quigley, Jonathan Hogg and Aileen Callaghan.

Finally, this would not have been completed without the love and support of my wife, Barbara, and my parents whose love, understanding and support throughout my university career has been invaluable.

Contents

List of Figures

Chapter 1

Introduction

It is often claimed that functional programming languages, and in particular pure functional languages, are suitable for formal reasoning. This claim is supported by the fact that many people in the functional programming community do reason about languages and programs in a formal or semi-formal way. Different reasoning principles, such as equational reasoning, induction and co-induction, are used, depending on the nature of the problem.

Using a computer program to check the application of rules and to mechanise the tedious bookkeeping involved can simplify proofs and provide more confidence in their correctness. When reasoning about programs, this can also allow experiments with new rules and reasoning styles, where a user may not be confident about structuring a proof on paper. Checking the applicability of a rule can eliminate the risk of mistakes caused by misunderstanding the theory being used. Just as there are different ways in which formal or informal reasoning can be applied in functional programming, there are different ways in which tools can be provided to support this reasoning.

The rest of this chapter looks first at the reasoning tasks that can be supported and the types of tools that can be built to support them. It then outlines the specific aims, approach and contribution of this thesis.

1.1 Reasoning and functional programming

Two significant applications of reasoning to functional programming are proving properties and equivalence of programs and deriving new rules needed to prove these results. These activities require different levels of knowledge to produce a correct proof.

The highest level of knowledge and formal rigour is required to derive new rules. New rules are derived within some semantic framework, typically an operational or denotational semantics. For example, fixpoint induction is often derived from a denotational semantics, as is parametricity [Wad89]. Leading advocates of coinduction often present work derived from an operational semantics framework [Gor94].

In many cases there is work showing either that the rules derived from one semantic framework are transferable to other semantic frameworks, or that the semantic frameworks are equivalent. It requires a greater range and depth of mathematical knowledge to understand how these results can be derived and combined in different semantic frameworks than to apply the rules to specific problems. But when a mixture of rules is used to reason about programs, without checking that the underlying semantics are compatible, then the resulting proofs, while quite possibly correct, are not as rigorous as they could be.

Mechanised reasoning support can fill this gap by providing a framework for expert users to derive rules and for a more general group of users to apply them. This would allow the less expert user to apply the rules, without being aware of the underlying semantics, and with the knowledge that if the rule is applied incorrectly then the mechanised tool will detect the error. In addition, with a suitable level of automation a tool can also simplify the discovery of the proof itself.

1.2 Supporting formal reasoning

There are three main ways of developing a program to mechanise the construction of a proof about some property of a program or language. The choice of method is influenced by whether the aim is to reason about programs, to derive new rules to aid reasoning, or to reason about the language semantics. The first approach is to develop a tool with no mechanised formal basis but which can manipulate terms in the language. Such a tool could be a custom rewrite engine with a user interface [Gil96] or an informal translation of the function definitions and derived rules for the language into axioms in a theorem prover [Tho89, Tho93, Tho94]. Both are useful, but the lack of a formal basis means there is no way to check that the rules themselves are correct and no internal way of generating new rules.

A second approach is to directly use the underlying logic in an existing theorem prover. Functions, and function definitions, can be expressed in the logics of most theorem provers. Some, such as HOL [GM93], have a logic that restricts the functions that can be introduced to ones that are terminating and total, providing a useful, but limited, basis for reasoning about such programs. LCF [Pau87] provides a logic where non-terminating functions can be introduced, but it forces the user to reason about a 'bottom element' in all types—even when this may be unnecessary because the bottom element cannot arise [GMW79, Pau87]. While such systems are the simplest way to provide formal support for mechanised reasoning, the restriction on programs expressible and the inability to derive new rules directly from the semantics may prevent reasoning about many real programs.

For example, syntactic restrictions on the form of programs to ensure termination can mean that to reason about a program it is necessary not just to extend it as in the requirement for totality, but to rewrite the program completely. In the lazy functional programming community, infinite data structures and non-terminating programs are used not just where infinite behaviour is required, but also to simplify the structure of a program by allowing the programmer to write functions that define an infinite structure and then relying on lazy evaluation to ensure the infinite structures are not generated. Restrictions on the termination behaviour will disallow many of the programs that are actually written. In addition, since new proof rules and other meta-theoretic results cannot be derived within the system, there will be restrictions on the range of proof styles that may be applied to problems.

The third approach for developing a mechanised tool is to first define the semantics of the language in a theorem prover and then develop reasoning principles on top of this. This allows the meaning to be given to programs that cannot be expressed directly in the theorem prover's logic and provides a means to derive new rules from the semantics. As long as the semantics are correct, any program for the language will be able to be entered, and any new meta-theory should be derivable. Varieties of denotational semantics and operational semantics can be used to express the semantics of the language.

Denotational semantics determines the meaning of a program by translation into a mathematical model. One of the advantages of this approach is that the meaning of the equality of two programs is easily expressed as the equality of the programs' denotations in the underlying model. The definitions of some lanuages within HOL have used this approach [Reg95, Age94].

Operational semantics works by defining the meaning in terms of the program's execution on some abstract machine. The mathematical concepts used to give the meaning to programs are often simpler than in a denotational semantics, since nothing more complicated than relations is needed. This makes operational semantics suitable for expressing the meaning of terms in the language and for reasoning about the language, but not necessarily for reasoning about equality of programs since this is not given by the definition of the abstract machine. The semantics of SML is specified using an operational semantics [MTH90, MT91, MTHM97]. Numerous systems have been built by formalising parts of this definition in a theorem prover [Sym92, Sym93, VG93, Van94, MG94, CO92].

While equality of programs that yield a value can easily be expressed as the equality of the results of the evaluation of both programs, there is no simple definition of equality for programs or functions that do not terminate to yield a value. Instead the equality can be defined as a new relation that captures the intended meaning and can be proved to be a congruence. The operational approach leads to a system that does not impose any restrictions on the programs that can be expressed. Any program that can be expressed in the language can be entered.

Whether or not a particular semantics for a language, and hence the derived meaning of equality of two programs, correctly captures our intentions depends on the choices made in defining the semantics and equality. For domain theory most of these choices are distilled into the choice of the model for the semantics. In the operational approach the choices are divided between the choice of execution rules for the abstract machine and the choice of equivalence relation. Any of these choices may lead to a semantics that gives a different meaning to a program than the meaning intended by the programmer. For example, the semantics may specify a different termination behaviour from that expected by the programmer. Such a semantics may be incorrect, but there may also be no mathematical inconsistencies. One way to determine the correctness of the semantics is to compare the meaning given to programs with the meaning assigned by some other semantics we already believe to be correct.

One definition of equivalence that is more abstract than either of the above approaches is contextual equivalence. This states that two programs are equivalent if they have the same behaviour when placed in any larger program or context. This is often used as the benchmark for full abstraction results about the semantics. The models most often used in the domain theory do not lead to a semantics that is fully abstract, although models can be found based on game theory that are [McC98, AJM94]. Correctness with respect to contextual equivalence is easier to obtain with an operational semantics and a defined equivalence than with a denotational semantics. The equivalence relation for the language could be defined to be contextual equivalence but using different relations can give rise to more useful reasoning principles for proving the equivalence of programs.

1.3 Aims

The aim of the work described here is to investigate how to develop a mechanised reasoning system to allow reasoning about algorithms as a functional programmer would write them, not an encoding of the algorithm into a significantly different form. In addition, the work aims to develop a system to support a user who is not a theorem proving expert or an expert in the theoretical foundations of functional programming. The work is aimed towards a system that could be used by a functional programmer developing real programs and wishing to prove some or all of the programs correct or just prove two programs are equivalent. Such a user will understand some of the concepts of induction and other proof principles but should not have to handle fine-grained mathematical details.

Such a system removes two obligations from the programmer in the generation of rigorous proof:

- the obligation to understand how proof rules are derived from the underlying semantics, and
- the obligation to produce and check every step of the proof.

The system should be extensible, allowing the addition of new rules and tools. This involves supporting a second class of user who, as an expert in the theoretical foundation or theorem proving, can derive these new rules.

The work is based on the definition of an operational semantics for a lazy functional programming language in a theorem prover. While this language is smaller and simpler than a full strength functional language, it still allows the expression of many of the styles of programming used. Such a set-up has in the past been used primarily to reason about language semantics; here it is extended with a definition and theory of equality. This provides a semantic base from which to define a range of reasoning principles.

One primary difficulty with this approach is that much of the reasoning is at a very low level and consists of many more steps than a paper proof. One aim of the work here is to show it is possible to use sufficient automation and derived rules within the system to move towards a system that allows reasoning at the same or higher level than semi-formal reasoning on paper.

1.4 Contributions

A major product of the work described here is a system that satisfies the aims described in the previous section. In particular there is a theory, formalised in HOL, along with associated proof tools that allows reasoning, at a high level, about lazy functional programs without placing restrictions on the form and termination behaviour of the functions. The system makes use of a variety of tools written in the ML language and ideas from the functional programming community, such as strictness analysis, to achieve this level of reasoning.

This result shows that it is possible to define the semantics of a language by a defining the syntax and a hierarchy of semantic relations and still recover a practical system by use of the ML language to construct proof tools.

Some more specific contributions that resulted from this work are:

- A formalisation of coinduction, finite maps [CS95] and closing substitutions. These general theories, which have not previously been mechanised in HOL, are used throughout the rest of the formalisation and form libraries of results that could be used elsewhere.
- The operational semantics of a small language has been embedded in HOL and a large collection of standard results, such as the uniqueness of type assignments and determinacy results about reduction, has been proved. The treatment here is different from other embeddings of functional languages in that it is aimed at reasoning about programs, rather than the semantics.
- A mechanisation of Gordon's theory of applicative bisimulation [Gor95a]. The equivalence relation is defined relative to an operational semantics and is proved to be a congruence. This was the first substantial development of this theory in a theorem prover and illustrates that deriving equality of functional programs from an operational semantics is feasible in a mechanised setting.
- A mechanised operational treatment of a restricted form of parametric polymorphism ("theorems for free") [Wad89, Pit98]. Many presentations begin by setting up a semantic framework tailored to the development of parametric polymorphism but not the same semantic framework used to derive other results. The work here uses the same operational semantics that the rest of the work is based on. This shows that this semantic framework is suitable for extension with new reasoning principles.
- A set of examples demonstrating the range of proofs possible, including results about programs that cannot expressed or proved correct without dealing with undefined and infinite values.

While the work described here is not a fully fledged system suitable for general use—it lacks a full-blown user interface, support for derived syntax, and a rich library of pre-derived results—it does illustrate the practicality of the approach and deals with many issues not brought together in one semantic framework and system before.

1.5 Outline

This thesis can be divided into three main parts. Chapters 2 to 4 contain a discussion of other tools and theory relevant to the rest of the thesis and an informal exposition of the target language and its semantics. Chapters 5 to 7 discuss the development of the theory in the HOL theorem prover and the basic tools necessary to mechanise the semantics. Chapters 8 and 9 develop this platform further and describe the higher level tools to support reasoning and the development of extensions to the theory. These chapters also contain examples of the use of the tools developed. A more detailed breakdown of the contents is as follows:

Chapter 2 - Background. This chapter describes some of the theory used in the subsequent chapters. This includes a description of the HOL theorem proving system, a discussion of the style of operational semantics used, a brief introduction to coinduction, and a brief description of the finite map library.

Chapter 3 - Design Choices. This chapter discusses some of the choices made in deciding the style of the development employed. The chapter looks in more detail at the language features to be supported, alternative styles of language semantics, and ways to embed languages in theorem provers.

Chapter 4 - Overview of Language and Architecture. This chapter contains a discussion and informal overview of the syntax and semantics of the functional programming language and the definition of equivalence. It also provides an overview of the structure of the reasoning system developed.

Chapter 5 - Embedding the Syntax and Semantics. This is the first chapter discussing the development of the system itself. The embedding of the language syntax and semantics as new types and relations in the HOL theorem prover is given and the main results about this embedding are proved.

Chapter 6 - Automation of Low Level Inference. Use of the syntax and semantics introduced involves a large number of trivial proof steps that make application of the rules by hand tedious and impractical. This chapter presents the tools that fully automate all these steps by using an implementation of the interpreter specified by the operational semantics and mirroring this execution in the logic by application of the rules.

Chapter 7 - Equivalence. This chapter describes a theory of coinduction that is added to the HOL system and then used to define an equivalence relation. The key properties of this relation, including the fact that it is a congruence and is equivalent to contextual equivalence, are proved.

Chapter 8 - Supporting Formal Reasoning. The tools to allow reasoning about the equality of programs are developed. These include tools for defining new functions and proving their basic properties, an equational reasoning system, and tools to partially automate proofs by coinduction. A range of examples is presented to illustrate the use of these tools.

Chapter 9 - Styles of Reasoning. New reasoning principles are derived and the associated tools are developed to illustrate how the system can be extended. The principles derived include structural induction, a variant of parametric polymorphism, and the take lemma.

Chapter 10 - Conclusions. This chapter provides some conclusions that can be drawn from this work and suggests further work to be considered.

Chapter 2

Background

This chapter gives an overview of some existing tools and mathematical theory used in the rest of this work. There are several approaches that could have been used as the basis of this work; some of these alternative approaches are discussed in the next chapter.

2.1 Theorem proving

There are two constraints on the choice of theorem prover that will be the starting point for the system described here. It must support a formalism suitable for embedding the semantics of the language, and it must provide a means of writing a rich set of tools to partially automate proof. The logical requirements for developing the system are minimal; the major requirement for a mechanisation of the theory for both co-induction and operational semantics is simply the ability to reason about relations.

The HOL theorem prover [GM93] was used because of its use of Standard ML as the meta-language. This is a fully featured programming language and allows complex proof tools to be programmed that can perform proof search if necessary. ML was first used as a meta-language in the LCF theorem prover [GMW79, Pau87]. The HOL theorem prover is one of the descendants of Edinburgh LCF and supports Higher Order Logic instead of the Logic for Computable Functions supported by LCF. In principle all the theory and tools developed here could be ported to other theorem provers in the LCF family.

HOL is a theorem proving environment for classical higher order logic [GM93]. There is a tradition in the HOL community of taking a purely definitional approach to using logic; instead of postulating axioms to give meaning to new notations, as is typical in the use of theorem provers such as LP [GG89], new concepts are defined in terms of existing ones that already have the required semantics. For example, the user must define any new type in terms of a precisely suitable subset of an existing type. This is guaranteed to preserve the consistency of the system, but leads to complex definitions. Packages are provided to perform definitions automatically from natural specifications of some important classes of types and functions. It is also possible to add new axioms to HOL and, although most of the work here follows the definitional approach, two axioms, the characteristic theorems for the syntax of the language, are added. The automated tools that would normally be used to support the definitions of these do not support some features of the syntax but the axioms are relatively simple and easily justified on paper. The axioms are discussed in detail in chapter 5.

HOL allows both forward and backward, or goal-directed, proof. For forward proof, an inference rule is applied to some theorems to derive a new theorem. One such inference rule is MP which implements Modus Ponens. This takes the theorems $\Gamma_1 \vdash t_1 \supset t_2$ and $\Gamma_2 \vdash t_1$ and yields the theorem $\Gamma_1 \cup \Gamma_2 \vdash t_2$ where Γ_1 and Γ_2 are sets of assumptions.

Goal directed proof is supported by the HOL subgoal package. This allows the goal to be interactively decomposed into subgoals that can eventually be proved. The current goal is a term together with a list of terms representing the assumptions that are made when proving the goal. The decomposition of a goal is usually performed by tactics, functions that transform one goal into a list of subgoals. An example of a tactic is CONJ_TAC, which breaks a conjunction into subgoals corresponding to the conjuncts. Once each of these subgoals is proved the original goal is proved. The tactics can themselves be combined by other functions, tacticals, such as THEN, which allows the compositions of two tactics in sequence.

An important feature of HOL is that the meta-language, Standard ML, is a fully featured programming language. This allows complex tactics to be programmed which may perform arbitrary proof search. A proof in HOL is generated by an ML program. This is usually developed interactively and can be saved and used again. The program can also be modified so that if the goal to be proved is changed then the existing proof can be modified rather than having to develop a new proof. If a pattern exists in the proofs of similar properties for many terms, then a generalised proof tool can often be written to automatically generate the proofs for a whole class of problems.

2.2 Operational semantics

An operational semantics is the description of the meaning of a language in terms of an abstract machine. The machine is normally expressed as a relation that relates each term to the term that results from the evaluation of the first term. Its arguments may also include various pieces of context information, such as an environment mapping identifiers to values or maintaining information about state. As the lazy language being discussed here contains no state, the latter is not needed in this work. If state were added to the language then many of the results relating to the semantics would be similar but the rules for proving the properties and equivalence of programs would not.

One feature common to all the abstract machines is that, as with a compiler or interpreter, the form of the rules is dictated by the syntax of the language. There will typically be one or more rules for each syntactic construct of the language. Such presentations of the semantics are referred to as structural operational semantics [Plo91]. There are several books that describe various approaches to the formalisation of an operational semantics for both functional and imperative languages [Win93, Gun92]. The Definition of Standard ML [MTH90] is also given as an operational semantics. The semantics of a language is normally divided into two sections, the static and dynamic semantics, which describe how to type and evaluate programs respectively. These correspond to two important stages in any compiler or interpreter for a functional language.

The discussion in the rest of this section will be illustrated by the rules for a simple, non-strict, functional language with variables, functions and numbers. The types for this language are

$$\begin{array}{lcl} ty & ::= & \mathsf{Num} \\ & | & ty_1 \rightarrow ty_2 \end{array}$$

and the expressions are

$$\begin{array}{lcll} exp & ::= & \mathsf{num}\ num & \text{Natural number constant} \\ & | & \mathsf{var}\ id & \text{Variables} \\ & | & \lambda id : ty.\ exp & \text{Function abstraction} \\ & | & exp_1\ exp_2 & \text{Function application} \end{array}$$

2.2.1 Static semantics

The static semantics is a collection of rules relating expressions to their type. The relation is written $\Gamma \vdash e : \alpha$, where e is the expression, α is the type assigned and Γ is the context that maps variables to their types. $\Gamma[x \mapsto \alpha]$ is a context with the same mappings as Γ, but mapping x to α. The rules defining this relation for the example language are

$$\frac{}{\Gamma[x \mapsto \alpha] \vdash \mathsf{var}\ x : \alpha} \qquad \frac{}{\Gamma \vdash \mathsf{num}\ n : \mathsf{Num}}$$

$$\frac{\Gamma[x \mapsto \alpha] \vdash e : \beta}{\Gamma \vdash (\lambda x : \alpha.\ e) : \alpha \rightarrow \beta} \qquad \frac{\Gamma \vdash e_1 : (\alpha \rightarrow \beta) \qquad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash (e_1\ e_2) : \beta}$$

The rules here are straightforward. For example, the type of any lambda abstraction, $\lambda x : \alpha.\ e$, is a function from the type of the argument, α, to the type of the body, β. The type of the body is determined in a context where the bound variable is mapped to the type of the argument.

2.2.2 Dynamic semantics

The approach to the static semantics illustrated in the last section is standard and similar to that used in most static semantics. There are more varied approaches to the formalisation of the dynamic semantics. Some differences centre around how variable binding is formalised, which can be done by recording the mapping from variables to the values they are bound to in an environment, or by substituting the value to which the variable is bound throughout the expression. The substitution of an expression e_2 for a variable x in expression e_1 will be written $e_2[e_1/x]$. It is not necessary to specify that the expressions are closed or well typed although we are mainly interested in expressions that are. Other differences centre on whether the meaning of a term is defined as what it evaluates to or in terms of several intermediate steps that are closer to the final value. A semantics that gives the meaning of a program in terms of its final value is called a "big step" semantics and will be represented by the relation $\Downarrow$. A semantics that gives the meaning in terms of several reduction steps is called a "small step" semantics and will be represented by the relation $\longrightarrow$. For a substitution based semantics, the big step semantics of the small language above would be:

$$\frac{}{\mathsf{num}\ n \Downarrow \mathsf{num}\ n} \tag{2.1}$$

$$\frac{}{\lambda x : t.\ e_3 \Downarrow \lambda x : t.\ e_3} \tag{2.2}$$

$$\frac{e_1 \Downarrow \lambda x : t.\ e_3 \qquad e_3[e_2/x] \Downarrow e_4}{(e_1\ e_2) \Downarrow e_4} \tag{2.3}$$

A small step semantics would be:

$$\frac{}{(\lambda x : t.\ e)\ e_1 \longrightarrow e[e_1/x]} \tag{2.4}$$

$$\frac{e_1 \longrightarrow e_3}{e_1\ e_2 \longrightarrow e_3\ e_2} \tag{2.5}$$

There are no reduction rules for numbers and lambda abstractions since these cannot be reduced further. The body of a lambda abstraction is reduced only after the function is applied to an argument. Other approaches would be possible but are not considered here.

The two styles give the same meaning to a program that returns a value, in this case a lambda abstraction or number. For the application $e_1 e_2$, repeated use of rule 2.5 to e_1 will either continue forever producing no value or will result in a value of the form $\lambda x : t.\ e_3$ that will be the same value produced by the big step evaluation relation $e_1 \Downarrow \lambda x : t.\ e_3$. Rule 2.4 will then apply giving an expression $e_3[e_2/x]$. Other rules will then apply to reduce this to some term e_4 that will be the same as that produced by the big step evaluation:

$$e_3[e_2/x] \Downarrow e_4$$

It is also possible that no reduction rules will apply indicating a "run-time" error in the evaluation of the program. In the richer language described later, this may happen if the program uses partial functions.

The difference in the two semantics relates to programs that do not evaluate to some value — such as partial or non-terminating. In this case the small step semantics allows for easier analysis of the intermediate results. Because of this, a big step semantics is often useful for specifying compilers where the main interest is in programs that return values. The small step semantics can be useful for theoretical work on languages and programs.

If an environment is used instead of substitution, then the form of the relation changes from a relation between expressions to a relation between expressions in the context of an environment. The big step semantics is very similar in structure to the static semantics.

$$\frac{}{(E,\ \mathsf{num}\ n) \Downarrow \mathsf{num}\ n} \tag{2.6}$$

$$\frac{(E,\ e_1) \Downarrow e_2}{(E[x \mapsto e_1],\ \mathsf{var}\ x) \Downarrow e_2} \tag{2.7}$$

$$\frac{}{(E,\ \lambda x : t.\ e_3) \Downarrow \lambda x : t.\ e_3} \tag{2.8}$$

$$\frac{(E,\ e_1) \Downarrow \lambda x : t.\ e_3 \qquad (E[x \mapsto e_2],\ e_3) \Downarrow e_4}{(E, (e_1\ e_2)) \Downarrow e_4} \tag{2.9}$$

A small step semantics using environments would look like this

$$\frac{}{(E[x \mapsto e_1],\ \mathsf{var}\ x) \longrightarrow (E,\ e_1)} \tag{2.10}$$

$$\frac{}{(E,\ (\lambda x : t.\ e)\ e_1) \longrightarrow (E[x \mapsto e_1],\ e)} \tag{2.11}$$

$$\frac{(E_1, e_1) \longrightarrow (E_2, e_3)}{(E_1,\ e_1\ e_2) \longrightarrow (E_2,\ e_3\ e_2)} \tag{2.12}$$

In the rest of this thesis we use a small step semantics with substitution and derive a big step semantics with substitution. This is discussed is greater detail later.

2.3 Co-induction and bisimulation

This section gives an overview of the motivation and theory for the use of co-induction and bisimulation. We begin with a description of process calculi, the problem area where many of the ideas were developed, and discuss how the same ideas have been applied to functional programs. This section concentrates mainly on the co-inductively defined relations, as these will be used later in this thesis. At the end of this section we discuss how this relates to co-inductively defined types that offer an alternative approach.

2.3.1 Process calculi

Many of the ideas related to co-induction and the labelled transition systems that will be used in the formalisation of equality in functional languages are closely related to the use of co-induction in process calculi such as CCS [Mil89] and the π-calculus [MPW92a, MPW92b]. In particular these process calculi describe systems with infinite behaviour. Both these calculi have been embedded in the HOL theorem prover [Nes92, Mel94]. This section gives a very brief overview of co-induction and equality in CCS.

CCS models processes and the communication between them. An example of a simple process, or agent, in CCS is the agent A defined by

$$A = a.A$$

where A is the name of the agent being defined and a is the name of a "port" waiting to send information. After sending, referred to as an output action, the agent returns to its initial state. This is represented by the recursive call to A. The semantics of CCS are given in terms of labelled transitions that indicate the actions that happen and the state of the agent before and after the transition. For example the only possible transition for the above process is

$$a.A \xrightarrow{a} A$$

This means that the agent $a.A$ may perform an output action on a and then evolve into A. Since $A = a.A$, this transition could be then be repeated giving an infinite graph with every transition having the same label. The a in the above example refers to an output action on the port. An agent B which waits for an input action $\bar{a}$ and then returns to its initial state would be defined by

$$B = \bar{a}.B$$

The overbar on the $\bar{a}$ indicates that an input is expected instead of an output.

A process that can perform more than one action is defined using a summation operator $+$. For example, a process which can perform either of two outputs with names a and b would be defined as

$$C = a.C + b.C$$

This process would have the transition graph

$$\begin{tikzcd} & C \arrow[dl, "a"'] \arrow[dr, "b"] & \\ C & & C \end{tikzcd}$$

Agents can be combined using a composition operator $\mid$. The two agents A and B can be combined to form an agent $A \mid B$ that could either perform an input or output transition with the label a. In addition, the two agents can also communicate, or synchronise, with each other since the output from A and the input to B have the same label a. The transition representing this internal communication is given a special label τ. This gives three possible transitions for $A \mid B$.

$$\begin{tikzcd} & A \mid B \arrow[dl, "a"'] \arrow[d, "\tau"'] \arrow[dr, "b"] & \\ A & A \mid B & B \end{tikzcd}$$

The final construction from CCS that is used is the restriction operator $\setminus$. This operator restricts the labels that are visible outside of the agent. If the composition of A and B is restricted to exclude the label a

$$(A \mid B) \setminus \{a\}$$

then this new construction cannot perform any transitions visible outside the new agent. The internal transition is the only possible one

$$(a.A \mid \bar{a}.B) \setminus \{a\} \xrightarrow{\tau} (A \mid B) \setminus \{a\}$$

For the purpose of defining the equality of two agents, these internal transitions will not normally be taken into consideration.

Agents will often be defined by mutually recursive definitions. For example two agents A and B could be defined by

$$A = a.B$$
$$B = b.A$$

This process A can make two transitions:

$$A \xrightarrow{a} B \xrightarrow{b} A$$

Now consider the process

$$C = a.b.C$$

This has transition graph

$$C \xrightarrow{a} b.C \xrightarrow{b} C$$

While A and C are not syntactically equivalent they have the same labels on the transition graph. This can be used to indicate the equivalence of the two agents.

Finally an agent F can be defined, in terms of two simpler agents D and E, which includes some internal communication

$$\begin{aligned} D &= a.b.c.D \\ E &= \bar{c}.E \\ F &= (D|E) \setminus \{c\} \end{aligned}$$

This gives a transition graph

$$F = (D|E) \setminus \{c\} \xrightarrow{a} ((b.c.D)|\bar{c}.E) \setminus \{c\} \xrightarrow{b} ((c.D)|\bar{c}.E) \setminus \{c\} \xrightarrow{\tau} (D|E) \setminus \{c\}$$

If the internal transition τ is ignored then this is a transition graph with the same labels as A and C. There is no external transition that the two agents can make that allows them to be distinguished. We could therefore regard them as equal, and indeed take this property to define equality.

The key idea in defining what it means to for the two processes A and F to be equivalent is captured by the set of pairs of states

$$\{(A,\ F),\ (B,\ ((b.c.D)|\bar{c}.E) \setminus \{c\}),\ (A,\ ((c.D)|\bar{c}.E) \setminus \{c\})\}$$

This set of pairs is called a bisimulation and has the following important properties:

- For any pair, if one member of the pair can make a transition then the other can make a transition with the same label, or in the case of a τ-actions do nothing.
- The pair of results of these transitions is still in the set.
- The pair $(A,\ F)$ is in the set.

The proof that two processes are equal will involve finding a relation containing the processes and proving that the relation is a bisimulation. The exact details of how to formalise these properties, particularly with respect to the τ-actions is not given here. There are several different approaches that can give different meanings to the equality of processes [Mil89].

2.3.2 Functional programming languages

Lazy functional programs can exhibit similar behaviour to the processes in the previous section. For two functions to be equal they must accept the same arguments and produce the same results. In a functional language any two functions of the same type can accept the same arguments, but if two functions produce infinite lists then the results cannot be compared in full. Instead they must be compared in terms of what another function can do with the result. For a list, any processing of the result can be decomposed into taking the head and tail of the result and then processing these. If the heads are equal and the tails are equal then the results will be equal. To view the results in terms of transitions we consider the following two transitions for infinite lists.

$$
\begin{array}{l}
x :: xs \xrightarrow{\mathsf{Hd}} x \\
x :: xs \xrightarrow{\mathsf{Tl}} xs
\end{array}
$$

The labels Hd and Tl represent taking the head and tail of the list respectively. The following example shows how these transitions can lead to a similar view of equality to that in the previous section. Consider the following three lists and a function that merges lists.

$$
\begin{array}{lll}
\mathsf{tflist} == \mathsf{True}::\mathsf{False}::\mathsf{tflist} & \mathsf{merge}_\alpha\, [\,]\, xs &== [\,] \\
\mathsf{flist} == \mathsf{False}::\mathsf{flist} & \mathsf{merge}\, xs\, [\,] &== [\,] \\
\mathsf{tlist} == \mathsf{True}::\mathsf{tlist} & \mathsf{merge}\, (x::xs)\, (y::ys) &== x::y::(\mathsf{merge}\, xs\, ys)
\end{array}
$$

The theorem

$$\vdash \mathsf{tflist} == \mathsf{merge\ tlist\ flist}$$

can be proven by analysing the transitions. The graphs of the transitions are:

$$\begin{tikzcd}
 & \mathsf{tflist} \arrow[d] & & \\
 & \mathsf{True}::\mathsf{False}::\mathsf{tflist} \arrow[dl, "\mathsf{Hd}"'] \arrow[dr, "\mathsf{Tl}"] & & \\
\mathsf{True} & & \mathsf{False}::\mathsf{tflist} \arrow[dl, "\mathsf{Hd}"'] \arrow[dr, "\mathsf{Tl}"] & \\
 & \mathsf{False} & & \mathsf{tflist}
\end{tikzcd}$$

$$\begin{tikzcd}
 & \mathsf{merge\ tlist\ flist} \arrow[d] & & \\
 & \mathsf{True}::\mathsf{False}::\mathsf{merge\ tlist\ flist} \arrow[dl, "\mathsf{Hd}"'] \arrow[dr, "\mathsf{Tl}"] & & \\
\mathsf{True} & & \mathsf{False}::\mathsf{merge\ tlist\ flist} \arrow[dl, "\mathsf{Hd}"'] \arrow[dr, "\mathsf{Tl}"] & \\
 & \mathsf{False} & & \mathsf{merge\ tlist\ flist}
\end{tikzcd}$$

As the leaves of the graphs are either literals or the expression we started with, they are a finite presentation of all the possible transitions for these expressions.

The role of the evaluation arrows is equivalent to the role of the τ-actions in CCS. They represent some internal processing that is not visible to the outside world. In both cases equivalence can be defined in terms of the other observable transitions.

Although in this case the unlabeled evaluation arrows match, in general they need not and only the labelled transitions are considered. Whenever we take the heads of the two lists the results are the same. Whenever we take the tails then the left and right hand sides are one of the following two pairs:

(tflist, merge tlist flist) (False::tflist, False::merge tlist flist)

These pairs in fact make up a bisimulation. An equivalence based on these ideas is referred to as an observational equivalence. In the context of functional programming language the equivalence is referred to as applicative bisimulation.

The above example is only one simple instance of using bisimulation to reason about functional programming. Much of the original work in this subject is due to Abramsky [Abr90] and the approach used here is based directly on that used by Gordon [Gor93a, Gor93b, Gor94, Gor95a, Gor95b]. Bisimulation can also be derived from a domain theoretic semantics [Pit94, Pit96]. Finally, a proof principle can be derived from a co-inductive definition of a type rather than from a language semantics [Pau94, Tur95, JR97]. These approaches will be discussed in the next chapter. The rest of this section describes how to formalise a co-inductive equality based on a small step operational semantics.

2.3.3 Underlying theory

Co-inductive definitions are the dual of inductive definitions [Acz77] and depend on much of the same relational theory. The definition of an inductively defined relation given below depends on two concepts, monotonic functions and F-closed sets.

A function F, mapping sets to sets, is monotonic if

$$\forall X\, Y.\, (X \subseteq Y) \supset (F(X) \subseteq F(Y))$$

and a set X is F-closed if $F(X) \subseteq X$

The least fixpoint of F, denoted $\mathsf{lfp}\ F$, is defined to be the intersection of all F-closed sets. For any monotonic function F this can be proved to be the smallest F-closed set and a fixpoint. The principle of induction follow directly and is that for any X:

$$F(X) \subseteq X \supset \mathsf{lfp}\, F \subseteq X$$

To see how this relates to the most common form of induction, mathematical induction on natural numbers, consider an element 0 and a function s with appropriate behaviour for the successor function (if s is applied to 0 n times then this will return a different value from any other number of applications). A monotone function F can be defined by

$$F(X) = \{0\} \cup \{s(x) \mid x \in X\}$$

The natural numbers can be defined to be the least fixpoint of this monotonic function.

$$\mathsf{Nat} = \mathsf{lfp}\, F$$

The principle of induction for this set will then be

$$F(X) \subseteq X \supset \mathsf{Nat} \subseteq X$$

which simplifies to

$$\{0\} \cup \{s(x) \mid x \in X\} \subseteq X \supset \mathsf{Nat} \subseteq X \tag{2.13}$$

Suppose addition, +, is defined in terms of s by the following equations

$$0 + x = x \tag{2.14}$$

$$s(x) + y = s(x + y) \tag{2.15}$$

Then we can prove the property

$$\forall x.\, x + 0 = x$$

by induction as follows. If Y is the set of terms for which this property holds then we can prove that it holds for all natural numbers by showing that $\mathsf{Nat} \subseteq Y$. From equation 2.13 it is necessary to show that

$$\{0\} \cup \{s(x) \mid x \in Y\} \subseteq Y$$

Simplifying this gives two cases, the base and step cases of an ordinary induction:

$$0 + 0 = 0$$

$$(x + 0 = x) \supset (s(x) + 0 = s(x))$$

Both of these are easily proved from the equations for addition. There are other ways to define the natural numbers and other proof principles that can be used to reason about them. The construction given above is used as an example because co-inductively defined relations are the dual of such inductively defined relations and because the corresponding principle of co-induction gives rise to the central proof principle for reasoning about the equivalence of programs introduced later.

The definition of a co-inductively defined relation given below depends on the definition of monotonic functions and F-dense sets. A set X is F-dense if for the function

F, $X \subseteq F(X)$. The greatest fixpoint of F, denoted gfp F, is defined to be the union of all F-dense sets. For any monotonic function F this can be proved to be the largest F-dense set and a fixpoint. The principle of co-induction is, for any X,

$$X \subseteq F(X) \supset X \subseteq \mathsf{gfp}\, F$$

A second principle, sometimes referred to as strong co-induction [Gor95a], can be easily derived from co-induction:

$$X \subseteq F(X \cup \mathsf{gfp}\, F) \supset X \subseteq \mathsf{gfp}\, F$$

This variation can simplify the choice of the relation X in a proof by co-induction.

The coinductive definition for the relation capturing what it means for two programs to be equal will involve choosing a suitable function F. In the work presented here this definition will be based on an small step operational semantics and a labelled transition system.

2.3.4 Applicative bisimulation

The operational semantics used will be similar to the semantics given by the rules 2.4 and 2.5. For the small language presented there this has the effect of reducing everything to a normal form where either the outermost construct is a number or lambda abstraction. The third possible outcome of repeated evaluation will be an infinite chain of reduction with no normal form being reached. The transition system will define what it means to observe properties of these expressions. In particular it will be possible to observe the value of a number or the result of applying a function to another term.

$$\frac{}{\mathsf{num}\ n \xrightarrow{\mathsf{num}\ n} 0} \qquad \frac{\vdash a : t_1 \to t_2 \qquad \vdash b : t_1}{a \xrightarrow{\mathsf{app}\ b} a\ b}$$

It only remains to somehow integrate the reduction relation into the transition system. Following the CCS style of adding a τ-action would be possible

$$\frac{a \longrightarrow b \qquad \vdash a : \mathsf{Num}}{a \xrightarrow{\tau} b}$$

but the approach here is to include the reduction in an inductive definition of the transition system by adding rules of the form:

$$\frac{a \longrightarrow b \qquad b \xrightarrow{\alpha} c \qquad \vdash a : \mathsf{Num}}{a \xrightarrow{\alpha} c}$$

This has the advantage of simplifying the definition of equivalence since there are not extra τ-actions to consider. In the two transition trees presented in section 2.3.2 this approach to the definition of transition would remove the unlabelled reduction arrows. It is possible to take this simplified approach for the labelled transition systems for functional programs because the reduction relation is deterministic. In the semantics for CCS the τ-actions may represent a choice between two possibilities in a non-deterministic system. As the reduction in the functional language is deterministic there is no information that can be inferred and the reduction can be safely hidden.

Equivalence is defined. by choosing a function $F_{==}$ based on the labelled transition system, so that two programs e_1 and e_2 are equivalent if they can make the same transitions to terms that are also bisimilar. The principle of co-induction derived from this definition allows a proof that $x == y$ by finding a relation S such that: $(x, y) \in S$ and for any $(a, b) \in S$

$$\begin{array}{l} (\forall a'.\, \forall l.\, a \xrightarrow{l} a' \supset (\exists b'.\, b \xrightarrow{l} b' \wedge (a', b') \in S \vee a' == b')) \wedge \\ (\forall b'.\, \forall l.\, b \xrightarrow{l} b' \supset (\exists a'.\, a \xrightarrow{l} a' \wedge (a', b') \in S \vee a' == b')) \end{array}$$

This formalises the idea of looking at the two transition graphs and ensuring that every pair of nodes are either equal or in the relation S.

2.4 Finite maps

The results in subsequent chapters make heavy use of functions that are defined on finite domains. These functions, commonly referred to as finite maps or finite partial functions, are used in reasoning about the semantics of the functional language, where they model type contexts and substitutions. This section gives a brief introduction to a formalisation of finite maps developed for this work, in collaboration with Donald Syme. An updated version of a publication presenting this theory in is given in appendix B.

The core theory of finite maps defines four constants. Finite maps are constructed from the empty mapping, FEmpty, and the update function that takes a map $\bar{f}$ and a pair (a, b) and returns the mapping that is equal to $\bar{f}$ but with a mapped to b, $\bar{f}[a \mapsto b]$. The application of a finite map $\bar{f}$ to a term a is represented by FApply $\bar{f}$ a. This will be written $\bar{f}$ a when it cannot be confused with function application. Finally, the domain of a finite map $\bar{f}$ is given by FDom $\bar{f}$.

The meaning of these constants is given by the following properties:

$$\vdash \forall \bar{f}\ a\ b.\ (\bar{f}[a \mapsto b])\ a\ =\ b$$

$$\vdash \forall x\ a.\ (x \neq a)\ \supset\ \forall \bar{f}\ b.\ (\bar{f}[a \mapsto b])\ x\ =\ \bar{f}\ x$$

$$\begin{array}{l} \vdash \forall a\ c.\ (a \neq c) \supset \\ \quad \forall \bar{f}\ b\ d. \\ \qquad \bar{f}[a \mapsto b][c \mapsto d]\ =\ \bar{f}[c \mapsto d][a \mapsto b] \end{array}$$

$$\vdash \forall \bar{f}\ a\ b\ c.\ \bar{f}[a \mapsto b][a \mapsto c]\ =\ \bar{f}[a \mapsto c]$$

$$\vdash \forall a.\ \neg(\mathsf{FDom}\ \ \mathsf{FEmpty}\ \ a)$$

$$\vdash \forall \bar{f}\ a\ b\ x.\ \mathsf{FDom}\ \ (\bar{f}[a \mapsto b])\ \ x\ \ =\ \ (x\ =\ a)\ \ \vee\ \ \mathsf{FDom}\ \bar{f}\ x$$

together with an induction theorem

$$\begin{array}{l} \vdash \forall P. \\ \quad P\ \ \mathsf{FEmpty} \wedge \\ \quad (\forall \bar{f}.\ P\ \bar{f} \supset (\forall x.\ \neg(\mathsf{FDom}\ \bar{f}\ x)\ \supset\ \forall y.\ P\ \ (\bar{f}[x \mapsto y]))) \\ \quad \supset \\ \quad \forall \bar{f}.\ P\ \bar{f} \end{array}$$

Equality of two finite maps can be proved using the following theorems, which follow from the above facts.

$$\vdash \forall \bar{f}\ \bar{g}.\ ((\mathsf{FDom}\ \bar{f} = \mathsf{FDom}\ \bar{g})\ \wedge\ (\forall x.\ \bar{f}\ x\ =\ \bar{g}\ x))\ =\ (\bar{f} = \bar{g})$$

$$\begin{array}{l} \vdash \forall \bar{f}\ \bar{g}. \\ \quad ((\mathsf{FDom}\ \bar{f} = \mathsf{FDom}\ \bar{g})\ \wedge\ (\forall x.\ \mathsf{FDom}\ f\ x\ \supset\ (\bar{f}\ x\ =\ \bar{g}\ x)) \\ \qquad =\ (\bar{f} = \bar{g}) \end{array}$$

A number of additional constants are defined to enrich the theory. All the basic constants introduced above either increase or preserve the domain of a finite map. A constant DRestrict can be defined which reduces the domain of a finite map to those elements satisfying some predicate. Some useful properties that can be proved of DRestrict are

$$\begin{array}{l} \vdash \forall \bar{f}\ p. \\ \quad (\forall x.\ \mathsf{FDom}\ (\mathsf{DRestrict}\ \bar{f}\ p)\ x\ =\ \mathsf{FDom}\ f\ x\ \wedge\ p\ x)\ \wedge \\ \quad (\forall x.\ \mathsf{FDom}\ \bar{f}\ x\ \wedge\ p\ x\ \supset\ (\mathsf{FApply}\ (\mathsf{DRestrict}\ \bar{f}\ p)\ x\ =\ \mathsf{FApply}\ f\ x)) \end{array}$$

$$\vdash \forall p.\ \mathsf{DRestrict}\ \mathsf{FEmpty}\ p\ =\ \mathsf{FEmpty}$$

$$\begin{array}{l} \vdash \forall \bar{f}\ p\ a\ b. \\ \quad \mathsf{DRestrict}\ (\bar{f}[a \mapsto b])\ p\ = \\ \qquad ((p\ a)\ \Rightarrow\ ((\mathsf{DRestrict}\ \bar{f}\ p)[a \mapsto b])\ \mid\ (\mathsf{DRestrict}\ \bar{f}\ p)) \end{array}$$

The expression of the form $b \Rightarrow e_1 \mid e_2$ uses the HOL syntax for a conditional expression. This is equal to e_1 if b is true and e_2 if b is false. Another important concept is composition, either of two finite maps or a finite map and a function. Three infix composition functions are defined:

$$\begin{aligned}
&\mathsf{f\_o\_f} : (\beta,\gamma)\mathit{fmap} \rightarrow (\alpha,\beta)\mathit{fmap} \rightarrow (\alpha,\gamma)\mathit{fmap}\\
&\mathsf{o\_f} \;\; : (\beta \rightarrow \gamma) \rightarrow (\alpha,\beta)\mathit{fmap} \rightarrow (\alpha,\gamma)\mathit{fmap}\\
&\mathsf{f\_o} \;\; : (\beta,\gamma)\mathit{fmap} \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha,\gamma)\mathit{fmap}
\end{aligned}$$

The notation is designed to show the link with composition of functions

$$\mathsf{o} : (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)$$

Chapter 3

Design choices

This chapter discusses the choice of object language, language semantics and embedding technology on which the work in the rest of this thesis is based. For each aspect the whole space of design choices is outlined, the current research in the field reviewed and the choice made for the work here explained. The issues in each section are not independent and the chapter concludes with a summary of the set of consistent choices made here.

3.1 Language

Two strategies for introducing formal proof into the process of writing functional programs are to support proof about programs that have already been written and introduce new languages and methodologies that make formal proof easier or incorporate proof into the programming methodology itself. The latter approach will, in general, require less work to support in a theorem proving environment because restrictions can be made on the form of the programs being reasoned about.

The ideal target for any project that aims towards a system that can be used by functional programmers to reason about their programs is an established programming language. The two languages that have the greatest user bases are Standard ML [MTH90, Pau96] and Haskell [H^{+}92, Bir98, Tho96]. These represent the two main styles of functional programming implementation. Standard ML is a strict language, where the arguments to a function are evaluated before the function application. Standard ML has side effects, assignment to state variables and exceptions. These features are the main obstacle to reasoning; they make the meaning of equality of two programs less clear and the need to deal with them significantly complicates the semantics.

Haskell is a lazy language, where the arguments to functions are not evaluated before the application of functions. Haskell does not allow side effects, since the interaction between side effects and laziness makes it hard to determine the behaviour of programs. In particular, it can be difficult to determine in what order side effects occur. Instead, features that would normally be handled by side effects, such as exceptions and input/output, are handled by use of laziness.

Languages such as Haskell are claimed to be easier to reason about because it is easier to understand and model what it means for two functions to be equal and to consider the meaning of separate parts of programs separately and combine the results. In particular, even if an expression appears in different places in a program it will always evaluate to the same value. This is called referencial transparency and does not hold in languages with side effects where the value of an expression may depend on assignments elsewhere in the program. The disadvantage is that these languages allow the possibility of infinite data and non-terminating functions to occur throughout programs, even where they have no clear utility. Indeed such structures are fundamental to the programming styles used.

3.1.1 Type theory

There are some type theories for functional languages [Tho91], such as The Calculus of Constructions [Luo94] or Martin-Löf Type Theory [NPS90], where the type language for programs is much richer than in functional languages such as Haskell and ML. Instead of the types simply stating what kind of values a variable or program denotes, the types can contain more information. Instead of just lists, types such as "sorted lists" can be introduced. The richer language allows specifications of the programs to be included in their types. Proof that a program has a certain specification can be reduced to proving that the program has the specified type.

There is a family of theorem proving systems, including Lego [LP92], ALF [ACN90], COQ [DFH$^+$93] and NuPRL [CAB$^+$86], based on type theory. In these systems properties of functions can be proved by determining that they have a suitable type. In addition, programs can in principle be constructed from proofs. A proof of existence of an object with a particular type will give rise to a program with the property corresponding to the type. Thus a new methodology for writing programs is introduced where the program is derived by finding a witness for the existence of a program of a particular type.

These systems can be used to reason about functional programs written using this methodology without the need to develop any new tools. It is not the approach used here, since the aim of this work is to reason about programs as a functional programmer would write them and investigate what can be achieved without imposing a new style of writing programs on the user.

3.1.2 Inductive and co-inductive types

The presentation of co-induction in the previous chapter concentrated on the use of co-inductive definitions to define a new relation giving the meaning of equality for programs. This is not the only way that co-induction can be used to give rise to a theory of equality over infinite data structures.

Instead of defining relations by co-induction an alternative is to define data types using co-induction. Such co-inductive types are the dual of inductive types. An inductive type is one where any member of the type is finitely generated by a set of constructors. For example, the list type is defined in many theorem provers to be the inductive type generated by the nil and cons constructors. Any list consists of a series of cons constructors terminated by a nil constructor. Such lists must be finite. Lists in Haskell and other lazy languages cannot be defined in this way since they contain infinite elements.

A (necessarily) infinite list can be defined as a co-inductive type. This can be thought of as a list which is always of the form cons x xs where the xs is another infinite list. The difference here is that there is no requirement to be able to enumerate the list in terms of only cons constructors. Indeed, for an infinite list this will be impossible since without a nil constructor there is no way to terminate the construction. Such types can be thought of in terms of destructors instead of constructors. An element of a type of infinite lists is a value to which two destructors head and tail can be applied.

Lists which may be infinite or finite can also be defined in this way by adding the nil constructor, or a corresponding destructor. This would allow the possibility, but not the requirement, of enumerating the list.

Inductive and co-inductive types can be defined using the same theory discussed in the previous chapter. Inductive types are the least fix-point of an equation describing the types and co-inductive types are defined in as the greatest fix-point of the same equation. This is mechanised in the Isabelle system [Pau94].

These types can also be derived using ideas from category theory. The induction theorem for a type follows from the initiality property of a type expressed as an algebra and the co-induction property from a type expressed as a co-algebra. This approach is explained in full by Jacobs and Rutten [JR97]. The Charity [FT96] system implements many of these ideas.

Turner [Tur95] advocates the use of these types along with a restriction to primitive recursion and co-recursion as a way of producing programs which are more easily understood. He refers to this as Strong Functional Programming. This gives an easier route to theorem proving support, as many of the complexities in real languages arise because of programming styles that are ruled out by these restrictions.

This approach is, in effect, to take a well-behaved, common subset of both Haskell and ML — whether the language is lazy or not becomes an issue for compiler designers. The order of evaluation does not affect the meaning of programs in this setting, although it may affect the efficiency of the programs.

As with the type theory approach, there is no need to develop new tools for this approach as existing theorem proving tools can be used. There is, however, the same requirement to write programs in a different style and the restriction to either write programs with very simple recursion, such as primitive recursion, which can be easily checked for termination conditions, or to prove properties of the recursion in a program in order to define new function.

The aim of the present work is to design a system when programs can be entered into the system regardless of the form of the recursion used and without any distinction between data and co-data.

3.1.3 The target language

The use of either an advanced type theory or a system treating data and co-data separately would give a system that would be suited for deriving new programs to match a specification. But the required restrictions on the way in which the programs can be written would prevent this being applied to many existing programs or programming styles. Because of this, the target language for the rest of this thesis is a subset of Haskell specified by a semantics that allows reasoning about programs that can contain general recursion or partial functions even if the programs do not produce any result. This will allow reasoning about existing algorithms to determine when they produce a value and when they fail.

A subset of Haskell is chosen because the Haskell community already uses informal reasoning to reason about their programs. For practical reasons of time and complexity, rather than supporting in full the large range of syntactic constructs in Haskell, only a small but significant subset is supported. The details of the language are given in the next chapter.

In addition, formalising the semantics of the language in the login of a theorem prover allows reasoning about the language itself. The logic in the theorem prover can be used to express and prove properties of the semantics that cannot be expressed in terms of the language alone. Results such as parametric polymorphism require such reasoning.

3.2 Reasoning technology

One method by which a language semantics can be embedded in the logic of a theorem prover is to translate its syntax directly into expressions within the logic. Each expression in the language is mapped meta-linguistically to its denotation in the logic.

For example, a conditional operator would be represented as a function in the logic of type $bool \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$ that takes the condition as its first argument and is equal to the value of its second or third argument if the first argument is true or false respectively. The expression **if-then-else true** e_1 e_2 would be mapped to this function applied to **true**, e_1, and e_2 and so would be provably equal to e_1 in the logic. This approach is referred to as a shallow embedding [BGG+92].

A second method is to represent the syntax of the language by the values of one or more data types in the logic. Typically each expression in the language, such as a conditional expression or a function abstraction, will be represented using one of the constructors of these data-types. A denotational semantics can then be given to the language by defining a function within the logic to map each value of this type to its denotation. Alternatively, an operational semantics can be given by defining relations between the types representing the syntax. This is referred to as a deep embedding [BGG+92].

One obvious difference between the two approaches is that in a shallow embedding the syntax of the language does not appear in the logic. It then becomes impossible to state some meta-theoretic results that involve quantification over expressions within the language, since no type of expressions exists.

The Definition of Standard ML [MTH90] has been the starting point for much of the work embedding the semantics of programming languages in theorem provers. The most complete approach to the dynamic semantics is the HOL-ML project [MG94, VG93]. Here the dynamic semantics of the language is investigated, including the imperative features and the module system. The project uses a deep embedding of the Definition of Standard ML in the HOL theorem prover. The major results of this project are meta-theoretic, such as confirmation that the dynamic semantics of Standard ML are deterministic.

Little of the work embedding the semantics of programming languages in HOL has been based on a domain theoretic approach, due to the lack of formalisms of sufficient domain theory in theorem provers to make a deep embedding of a language with recursive types practical. Formalising domain theory in theorem provers like HOL is an area of current research [Age94, Reg95]. The LCF theorem prover [Pau87] provide a means to reason about a functional language with a mechanised logic using a domain theoretic approach. Here the domain theory is part of the theorem prover's logic rather than an embedding in a logic such as Higher Order Logic.

The work presented here uses a deep embedding for two reasons. Firstly, it is not possible to find a representation in the logic of most theorem provers for non-terminating or partial functions. Secondly, we can reason about the semantics of the language, allowing proofs of properties of the semantics that give rise to new rules for use in reasoning about programs. For example, in a shallow embedding the types of the language will be types in the logic of the theorem prover. Results such as parametric polymorphism require reasoning about the types of terms. There is no mechanism by which to reason about the types of terms in the logic of a theorem prover such as HOL within the theorem prover itself.

3.3 Semantics

In order to be useful there are two main kinds of results that must be derivable for a program in the semantic framework used. These are the result of evaluating a program and a proof principle for showing equality of two programs. Where the definition of a language is presented in terms of an operational semantics, then the aim of the semantics is to state what a program evaluates to. While this is both important in its own right, and an important part of deciding what equality between programs means, it is the meaning of equality and how to prove equalities that are the main requirements for the work presented here. Equality can be defined in several ways.

3.3.1 Contextual equivalence

Contextual equivalence defines the meaning of the equality of two programs in terms of the behaviour produced when they are substituted into large programs, or contexts.

A context $C[\,]$ is an expression in the language with a "hole" (formally modelled by a free variable). $C[e]$ will be used to denote a context with an expression e replacing this free variable. Two expressions, e_1 and e_2, are equal if for any C, $C[e_1]$ and $C[e_2]$ either both diverge or both converge. It is not necessary to state that they converge to the same value for all contexts, as this follows from the definition. If there is some context, $C[\,]$, for which they do not converge to the same value then there must be another context that would make one diverge and one converge. For example, if $C[e_1]$ converges to a value v_1 and $C[e_2]$ converges to a value v_2 then the extended context, $C_1[\,]$ is given by

$$C_1[\,] \;=\; \mathsf{if}\ (C[\,] \;=\; v_1)\ \mathsf{then}\ v_1\ \mathsf{else}\ \bot$$

where $\perp$ is a program which diverges. $C_1[e_1]$ converges and $C_1[e_2]$ diverges if v_1 and v_2 differ.

This is an abstract view of equality since it makes no reference to the language semantics. The other equalities described below refer to the semantics and are often evaluated for correctness relative to contextual equivalence. The main practical weakness of contextual equivalence is that it does not provide an easily used proof principle. In order to prove the equality of two programs it is necessary to quantify over all contexts, requiring a proof by structural induction over the syntax of the language.

3.3.2 Models for equality

Denotational semantics determines the meaning of a program by translation into a mathematical model. Each term in the language is translated to an element, its denotation, in the model. One of the advantages of this approach is that the meaning of the equality of two programs is easily expressed as the equality of the programs' denotations in the model. Domain theory provides a well-understood way of building these models. Some embeddings in HOL have used a domain theoretic approach [Reg95, Age94]. But any shallow embedding in a theorem prover can be considered to be a denotational semantics, with the logic of the theorem prover providing the model.

The ability to inherit the meaning for equality, along with ways of proving properties of programs, is one of the advantages of a denotational semantics. A disadvantage is that in order to model recursive data types and programs, the mathematics used in the model is complex. In addition small changes to the language, or differences between different languages can require large changes to the model.

Another problem is that while the equality of elements in the model may be genuine mathematical equality, the equality may not be the correct relation between terms in the programming language being defined. One way to test the equality is to compare it with contextual equivalence. If the two equalities coincide then the semantics is said to be fully abstract. Semantics based on domain theory are often not fully abstract. Because the semantics were developed to give meaning to the evaluation of terms, typically any terms that evaluate to return a value will be equal in the model and under a contextual equivalence.

The problems usually arise with terms that do not return a value and are of interest in proving equivalence of lazy functional programs but not in specifying a compiler [AJM94]. Recently, models based on game theory have been developed which are fully abstract [McC98]. There has been no work on embedding such models in theorem proving systems.

3.3.3 Bisimulation

An alternative to defining equality via translation into a model is to take a semantics specifying how a term is evaluated and define explicitly what it means for two programs to be equal in terms of it. Observational equivalences and, in particular, a co-inductive definition of bisimulation, are one way of defining such an equivalence. This has the advantage of requiring much simpler mathematics than a denotational semantics, in part because the evaluation behaviour and meaning of equality are defined separately. This was introduced in the previous chapter. In addition similar definitions and theory can be applied to many different languages and it can be proved that such equivalences coincide with contextual equivalence.

3.4 Summary

This aim of this work is to develop a reasoning system for non-strict functional programs that does not restrict the programming style used to develop programs. For this reason a language based on advanced type theories or a language based on separate use of data and co-data were not chosen. This rules out a shallow embedding in a theorem prover and makes it necessary to make a deep embedding of the semantics of the language. Either an operational semantics with a defined equality or a denotational semantics could be used. The former is chosen because the theory required to produce a system in which the equality of programs is the desired equality, and is equal to contextual equivalence is simpler. In addition this approach will give rise to more results that can be reused for other languages.

Chapter 4

Overview of language and architecture

This chapter discusses the syntax and semantics of the programming language which is to be embedded in HOL. Ideally the language used would be a full strength functional programming language such as Haskell [H+92]. But, for both practical and theoretical reasons, a simpler language is used. The language is a variety of second order lambda calculus with datatypes. The language is referred to as SDT to stand for Second order with Data Types.

4.1 Language features

There are several features that the language would ideally have. SDT is a compromise between these feature and constraints on the time and complexity of the embedding process. Two important inclusions in the language are polymorphism and datatypes. These are discussed separately later in this section.

Many desirable features of a real language are left out. These include primitive support for pattern matching, a module system and a wider range of primitive operations and types. It is possible to add tool support for many of these. Pattern matching could be supported by providing tools to derive the correct underlying functions from a specification containing pattern matching and then proving that the rules given in the specification follow from the definition. Such a mechanism has been implemented for defining functions using pattern matching in the HOL logic by Konrad Slind [Sli96].

The Haskell module system is relatively simple compared to the more complex ML module system and the ability to reason at the level of these modules is not likely to be crucial to any proofs attempted using this system. Additional types and operations can be added by defining them in terms of the basic syntax. These possibilities are discussed in later chapters.

4.1.1 Polymorphism

On first inspection SDT's treatment of polymorphism is not similar to that in Haskell. It is similar to a language called *core* which is the language to which Haskell is translated in the Glasgow Haskell compiler [Jon96]. We know Haskell programs can be translated automatically to this language, but some difficulties with the practical use of such a translation are discussed in chapter 10. The fundamental difference is the need for explicit type abstraction and application. This change removes the need to do type inference, which is not trivial to express and reason about formally. As types guide much of the proof, this type inference has to be performed repeatedly, increasing the number of proof steps or slowing down automated tools. A similar motivation influenced the choice of *core* as the language in the Haskell compiler, because here the types drive the transformation steps and it is again inefficient to repeatedly infer types for terms and subterms.

Additionally many theoretical results, such as parametric polymorphism, can be proved more easily in a second order language and these results can then be applied to Haskell terms by reasoning about their translation into second order terms. This translation can be done simply by removing the additional type information contained in the SDT term. Such a translation is not always possible. While Haskell style programs can be translated into SDT the reverse is not true for all SDT programs. The syntax of SDT allows for quantification over type variables inside type expression and not just at the top level as in Haskell. Programs such as these cannot be translated back into Haskell, but such programs are easily identified or avoided.

4.1.2 Datatypes

In order to reason about a range of real programs the language must contain datatypes and the datatypes available must be extensible by the user. The way in which the types are formalised must also give rise to usable reasoning principles for datatypes, such as induction and co-induction.

It is possible to encode datatypes in terms of the function type in a second order language without the addition of primitive language constructs for datatypes. We do not take this approach since a programmer will expect the primitive constructs and any encoding would have to be hidden from the user. Maintaining this illusion when proving properties would be difficult during a proof attempt, particularly when automated tools need to work with the underlying term. Reverse translation may be impossible if two syntactically different datatypes in the users view of the system have the same encoding. In this work, constructors and datatypes are introduced as primitive syntactic constructs and destructors are introduced as primitive transitions. Some aspects of this approach are similar to those used in Gordon's thesis [Gor94]. An example of how this works in practice is given for lists at the end of this chapter.

4.2 Embedding of primitive types

In order to reason about programs the language has to include, or be able to encode, some primitive types. This section considers natural numbers and booleans. There are three main ways to add these primitive types to the language.

Datatypes In the previous section it was stated that the language will include the necessary constructions to add new datatypes. Both naturals and booleans can be encoded as new datatypes in the language with the constructors true and false for booleans and zero and successor for naturals. The if-then-else and case statements are simply special cases of the case statement for datatypes. Because of this it is not necessary to explicitly provide any support for these types in the language. This is elegant but there are other alternatives, and the practical merits for some of these alternatives are discussed below.

Primitive syntax Instead of defining types like numbers and Booleans in terms of existing language contructs, the syntax of the language can be extended to include them. In this approach a new type would be added for booleans or naturals and new syntax added for the constructors of the type. In this case booleans would require the addition of the constructors for true and false and a conditional statement. Numbers would require the addition of zero, successor and a case split function.

In earlier work [Col96b], where the language considered did not contain the facility to define datatypes, this approach was used for booleans and numbers. The presence of the datatypes in the language considered here means this approach is not required, as the new constructors and if-then-else and case functions can be easily added using the datatypes. General results for datatypes can be applied to numbers and booleans without needing to prove the result for each additional syntactic construct.

Lifting of HOL types The above approaches both require definitions to be made for all common functions over natural numbers and booleans, and the proof of all basic properties of these functions. But since we are working in the HOL system, we are in an environment where all these results have already been proven for the naturals and booleans. It would be of obvious practical advantage to make this existing body of results available for reasoning about programs written in SDT. Similar observations have previously been made by Agerholm [Age94].

For example, we can lift the type of numbers and some operators, such as addition, by adding new syntax to the language. Numbers would by expressed using a constructor `num` taking a HOL number as an argument. Addition could represented by another constructor `plus`. The expression $2 + 2$ would be written `plus (num 2) (num 2)`. The semantics would show that this evaluated to `num` $(2 + 2)$. In the HOL logic this is provably equal to `num 4`. This works in HOL because all elements of the number and boolean types correspond to some literal. The reverse is not true. There are elements of the number type in SDT, such as non-terminating programs, that do not correspond to any literal and so do not have any corresponding element in the HOL numbers. In the terminology introduced earlier this is a shallow embedding of numbers in a system that is otherwise a deep embedding.

The disadvantage of this approach is that it requires adding several new primitive elements to the syntax for each type and these new elements have to be handled as special cases in many places.

The approach taken here is to encode booleans as an SDT datatype and naturals as a primitive type with the literals and operations lifted from HOL. The details of these encodings are given in the next chapter. The choice to lift naturals and not booleans is motivated in part by the greater desire to lift the naturals in the anticipation of more complex reasoning about numbers than booleans in the system and by the fact that naturals are not commonly thought of, by a programmer, as a type consisting of only zero and successor while booleans are thought of as a type with two elements. In addition, choosing to treat the types differently gives the opportunity to compare the two approaches.

4.3 Syntax

The syntax of the types is given in figure 4.1. The only unusual feature is the syntax for datatypes. A datatype has an identifier (a string) that will be used to make recursive calls to the definition. The finite map from strings to a list of types is a mapping from the constructors to the types of their arguments. A finite map is precisely the right formalism, since we require a finite number of distinct constructors.

The syntax of expressions is given in figure 4.2. Most expressions are annotated with

$$\begin{array}{rcl} \mathit{ty} & ::= & \mathsf{Num} \\ & | & \mathsf{Var}\ \mathit{id} \\ & | & \forall \mathit{id}.\mathit{ty} \\ & | & \mathit{ty}_1 \to \mathit{ty}_2 \\ & | & \mathsf{Data}\ \mathit{id}\ (\mathit{id} \mapsto [\mathit{ty}]) \end{array}$$

Figure 4.1: The syntax of SDT types

exp	$::=$	$\mathsf{num}\ \mathit{num}$	Natural number
	$\vert$	$\mathsf{nop}\ (\mathit{num} \to \mathit{num} \to \mathit{num})\ \mathit{exp}_1\ \mathit{exp}_2$	Binary operation on numbers
	$\vert$	$\mathsf{bop}\ (\mathit{num} \to \mathit{num} \to \mathit{bool})\ \mathit{exp}_1\ \mathit{exp}_2$	Binary relation on numbers
	$\vert$	$\mathsf{var}\ \mathit{id}$	Variables
	$\vert$	$\lambda \mathit{id} : \mathit{ty}.\ \mathit{exp}$	Function abstraction
	$\vert$	$\mathit{exp}_1\ \mathit{exp}_2$	Function application
	$\vert$	$\Lambda \mathit{id}.\ \mathit{exp}$	Type abstraction
	$\vert$	$\mathit{exp}_{\mathit{ty}}$	Type application
	$\vert$	$\mathsf{rec}\ \mathit{id}_{\mathit{ty}}\ \mathit{exp}$	Recursive value
	$\vert$	$\mathsf{con}\ \mathit{id}_{\mathit{ty}}\ [\mathit{exp}]$	Constructor
	$\vert$	$\mathsf{case}\ \mathit{exp}\ (\mathit{id} \mapsto \mathit{exp})$	Case expression

Figure 4.2: The syntax of SDT expressions

their types. For the expressions representing natural numbers and operations on natural numbers the syntax is a little strange since we are lifting the natural numbers from the underlying logic rather than defining them here.

4.4 Substitution

In order to give both the static and dynamic semantics of the language it is necessary to define the substitution of expressions or types for free expression or type variables. For example, an abstraction can have the form $\lambda x : \alpha.\ e$ where x is a variable and e is some expression, possibly containing x. This can be applied to a term y of type α to produce the term formed by replacing x by y in e. Much of the detail in this thesis will involve the formalisation of the substitution by which the term y is substituted into e. This substitution will be represented by $e[y/x]$.

The main complexity in the treatment of substitution arises because of the possibility of variable capture. If the variable x is substituted for y in $\lambda x : \alpha.\ y\ x$ without renaming

$$(\lambda x : \alpha.\ y\ x)[x/y] \;=\; \lambda x : \alpha.\ (y\ x)[x/y] \;=\; \lambda x : \alpha.\ x\ x$$

then the meaning is changed by the capture of the variable x by the binding. Instead we must rename the bound variable x to get

$$(\lambda x : \alpha.\, y\ x)[x/y] \;=\; \lambda x' : \alpha.\, (y\ x')[x/y] \;=\; \lambda x' : \alpha.\, x\ x'$$

The terms $\lambda x : \alpha.\, x\ x$ and $\lambda x' : \alpha.\, x\ x'$ are different terms with different meanings and so correct treatment of substitution is important. Several other authors have already treated these issues in HOL [HM94, Mel94, GM96] and the treatment discussed later is a variant of these.

These issues may seem to have little relevance to real programming. When a well-typed program is substituted into another well-typed program there are no free variables in the incoming program and so variable capture can never occur. Much of the work in this thesis involved identifying when such simplifying assumptions can be made and developing a simpler theory to cover these cases.

Substitution itself can be formalised as a substitution of a single variable as described above, or as the simultaneous substitution of terms for several variables. As simultaneous substitutions are needed later we define this first and define the single substitution as a special case.

In some work functions from variables to terms are used to formalise the substitution function. This thesis uses a finite map from variables to terms, as there are only a finite number of variables being substituted for at any one time. A type substitution is a finite map from type variables to types and an expression substitution is a finite map from variables to expressions.

A type substitution can be applied to either a type or an expression while an expression substitution can only be applied to expressions. The three simultaneous substitution functions are:

$$\mathsf{eeSubs}{:}\mathit{exp} \rightarrow (\mathit{string} \mapsto \mathit{exp}) \rightarrow \mathit{exp}$$
$$\mathsf{teSubs}{:}\mathit{exp} \rightarrow (\mathit{string} \mapsto \mathit{ty}) \rightarrow \mathit{exp}$$
$$\mathsf{ttSubs}{:}\mathit{ty} \rightarrow (\mathit{string} \mapsto \mathit{ty}) \rightarrow \mathit{ty}$$

Instead of representing the application of a substitution $\bar{s}$ to an expression e as $\mathsf{eeSubs}\ e\ \bar{s}$ we write $[e]_{\bar{s}}$. The same notation is used for all three substitutions if the types of the substitution and term can be inferred.

From these definitions of simultaneous substitution the definition for substituting for one variable can easily be derived:

$$\mathsf{eeSub}{:}\mathit{exp} \rightarrow (\mathit{string}, \mathit{exp}) \rightarrow \mathit{exp}$$
$$\mathsf{teSub}{:}\mathit{exp} \rightarrow (\mathit{string}, \mathit{ty}) \rightarrow \mathit{exp}$$
$$\mathsf{ttSub}{:}\mathit{ty} \rightarrow (\mathit{string}, \mathit{ty}) \rightarrow \mathit{ty}$$

Details of the definitions of all the substitution functions are given in the next chapter.

$$\frac{}{\Gamma \vdash \mathsf{num}\ n : \mathsf{Num}} \qquad \frac{}{\Gamma[x \mapsto t] \vdash \mathsf{var}\ x : t}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Num} \qquad \Gamma \vdash e_2 : \mathsf{Num}}{\Gamma \vdash \mathsf{nop}\ op\ e_1\ e_2 : \mathsf{Num}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Num} \qquad \Gamma \vdash e_2 : \mathsf{Num}}{\Gamma \vdash \mathsf{bop}\ op\ e_1\ e_2 : \mathsf{Bool}}$$

$$\frac{\Gamma[x \mapsto t_1] \vdash e : t_2}{\Gamma \vdash (\lambda x : t_1 .\ e) : t_1 \rightarrow t_2} \qquad \frac{\Gamma \vdash e_1 : (t_1 \rightarrow t_2) \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash (e_1\ e_2) : t_2}$$

$$\frac{\Gamma \vdash e : t \quad x \text{ not free in } \Gamma}{\Gamma \vdash \Lambda x .\ e : \forall x . t} \qquad \frac{\Gamma \vdash e : \forall x . t}{\Gamma \vdash e_{t_1} : t[t_1/x]}$$

$$\frac{\Gamma[x \mapsto t] \vdash e : t}{\Gamma \vdash \mathsf{rec}\ x_t\ e : t}$$

$$\frac{\Gamma \vdash e_1 : t_1[t/x] \quad .. \quad \Gamma \vdash e_n : t_n[t/x]}{\Gamma \vdash \mathsf{con}\ c_t\ [e_1..e_n] : t} \qquad \begin{array}{l} t = \mathsf{Data}\ x\ \overline{m} \\ [t_1..t_n] = \overline{m}\ c \end{array}$$

$$\frac{\begin{array}{l} \Gamma \vdash e : \mathsf{Data}\ x\ \overline{m} \\ \forall s . \mathsf{FDom}\ \overline{c}\ s \supset \Gamma \vdash (\overline{c}\ s) : (\mathsf{makefun}\ t\ (\overline{m}\ s))[\mathsf{Data}\ x\ m/x] \end{array}}{\Gamma \vdash \mathsf{case}\ e\ \overline{c} : t}$$

Figure 4.3: Static Semantics

4.5 Static semantics

The static semantics of the language is given in figure 4.3. The general form of the rules has been explained in section 2.2.1. The only complex rules are the rules for constructors and for the case expression. The function makefun takes a list of types (the arguments to a constructor) and generates a function type (the type of a function to consume the arguments to a constructor). It is defined by

$$\begin{aligned} \mathsf{makefun}\ t\ [\,] &= t \\ \mathsf{makefun}\ t\ (x :: xs) &= (x \ \rightarrow\ (\mathsf{makefun}\ t\ xs)) \end{aligned}$$

4.6 Dynamic semantics

The dynamic semantics of the language is given in figure 4.4. This a small step semantics using substitution to handle variable binding as discussed in section 2.2.2. This is a subset of all the possible reduction rules. The subset chosen specifies the reduction order

$$\frac{}{(\lambda x : t.\ e)\ e_1 \longrightarrow e[e_1/x]} \qquad \frac{}{(\Lambda x.\ e)_t \longrightarrow e[t/x]}$$

$$\frac{f_1 \longrightarrow f_2}{f_1\ e_1 \longrightarrow f_2\ e_1} \qquad \frac{e_1 \longrightarrow e_2}{e_{1t} \longrightarrow e_{2t}}$$

$$\frac{}{\mathsf{rec}\ x_t\ e \longrightarrow e[\mathsf{rec}\ x_t\ e/x]}$$

$$\frac{}{\mathsf{case}\ (\mathsf{con}\ x_t\ [e_1\ ..\ e_n])\ \bar{c} \longrightarrow (\bar{c}\ x)\ e_1\ ..\ e_n}$$

$$\frac{e_1 \longrightarrow e_2}{\mathsf{case}\ e_1\ \bar{c} \longrightarrow \mathsf{case}\ e_2\ \bar{c}}$$

Figure 4.4: Reduction Rules

$$\begin{array}{lcl} \textit{label} & ::= & \mathsf{numL}\ \textit{num} \\ & | & \mathsf{appL}\ \textit{exp} \\ & | & \mathsf{AppL}\ \textit{exp} \\ & | & \mathsf{destL}\ \textit{string ty num} \end{array}$$

Figure 4.5: Labels for labelled transition system.

and gives a non-strict semantics. There are no reduction rules for numbers, variables, functions or constructors as these cannot be reduced.

4.7 Labelled transition system and equivalence

The labelled transition system is central to the definition of equivalence. Before introducing the rules for the transition system, the syntax of the labels must be given. This is shown in figure 4.5. The rules for the transition system follow the examples given in section 2.3 and are given in figure 4.6.

The choice of including datatypes as a primitive in the language is important here as there is only one destructor step to go from one element of a type to the elements of the component types. If an encoding were used, then more than one destructor step would be necessary.

From this transition system applicative bisimulation can be defined as in section 2.3. This will give rise to a proof principle that will allow two programs, x and y, to be proved equivalent by finding a relation S that contains the pair x and y and is a bisimulation.

$$\dfrac{}{\mathsf{num}\ n \xrightarrow{\mathsf{numL}\ n} 0} \qquad \dfrac{a \longrightarrow b \quad b \xrightarrow{\alpha} c \quad \vdash a : \mathsf{Num}}{a \xrightarrow{\alpha} c}$$

$$\dfrac{\vdash a : t_1 \to t_2 \quad \vdash b : t_1}{a \xrightarrow{\mathsf{appL}\ b} a\ b} \qquad \dfrac{\vdash a : \forall x.t_1}{a \xrightarrow{\mathsf{AppL}\ t} a_t}$$

$$\dfrac{\vdash \mathsf{con}\ c_t\ [] : \mathsf{Data}\ x\ \overline{m}}{\mathsf{con}\ c_t\ [] \xrightarrow{\mathsf{destL}\ c\ t\ 0} 0} \qquad \dfrac{\vdash \mathsf{con}\ c_t\ [e_1 \mathinner{..} e_n] : \mathsf{Data}\ x\ \overline{m}}{\mathsf{con}\ c_t\ [e_1 \mathinner{..} e_n] \xrightarrow{\mathsf{destL}\ c\ t\ i} e_i}\ 1 \le i \le n$$

$$\dfrac{a \longrightarrow b \quad b \xrightarrow{\alpha} c \quad \vdash a : \mathsf{Data}\ x\ \overline{m}}{a \xrightarrow{\alpha} c}$$

Figure 4.6: Rules for labelled transition system.

That is it has the property that for any $(a, b) \in S$

$$(\forall a'.\, \forall \alpha.\, a \xrightarrow{\alpha} a' \supset (\exists b'.\, b \xrightarrow{\alpha} b' \wedge (a', b') \in S \vee a' == b')) \wedge$$
$$(\forall b'.\, \forall \alpha.\, b \xrightarrow{\alpha} b' \supset (\exists a'.\, a \xrightarrow{\alpha} a' \wedge (a', b') \in S \vee a' == b'))$$

4.8 Example: lists

To illustrate many of the language constructs introduced in this chapter we discuss the introduction of the type of possibly infinite lists (streams). This also illustrates the use of logical constants of the HOL logic to give names to the new types and constructors.

There are four identifiers that give rise to SDT types: Num, $\forall$, $\rightarrow$ and Data. The identifier Data is distinct from the rest in that it will not appear in programs but only in definitions.

One instance of a datatype in SDT is

$$\begin{aligned} &\mathsf{Data}\ \mathit{nlist}\ [\mathit{nil} \mapsto [], \\ &\qquad\qquad \mathit{cons} \mapsto [\mathsf{Num}, \mathsf{Var}\ \mathit{nlist}]] \end{aligned}$$

This syntax corresponds to a list of numbers. It is a legitimate piece of syntax for an SDT type and could be used in programs. In order to make SDT programs look more familiar we assign this piece of syntax a name in HOL:

$$\begin{aligned} \mathsf{nlist} \ = \ &\mathsf{Data}\ \mathit{nlist}\ [\mathit{nil} \mapsto [], \\ &\qquad\qquad \mathit{cons} \mapsto [\mathsf{Num}, \mathsf{Var}\ \mathit{nlist}]] \end{aligned}$$

where $[\mathsf{Num}, \mathsf{Var}\ \mathit{nlist}]$ is the list of types that should be supplied to the *cons* constructor. The *nil* constructor takes no arguments. nlist is a newly introduced logical constant in HOL. This constant can then be used in programs to represent the type of lists of numbers. The same process can be repeated for constructors and for functions. The remainder of this section considers the more general type of polymorphic lists.

4.8.1 Introducing the list type

As the polymorphic type for lists is a generalisation of the type for lists of numbers, a first guess at the definition of the list type would be:

$$\begin{aligned}\mathsf{list}\ \alpha \ = \ \mathsf{Data}\ \mathit{list}\ [&\mathit{nil} \mapsto [\,],\\ &\mathit{cons} \mapsto [\alpha, \mathsf{Var}\ \mathit{list}]]\end{aligned}$$

where α is an arbitrary type. This definition is incorrect because of the possibility of the capture of a type variable if α is instantiated to a type with the variable *list* free in it. The simplest example would be the instantiation of α to Var *list*. This gives:

$$\begin{aligned}\mathsf{list}\ (\mathsf{Var}\ \mathit{list}) \ = \ \mathsf{Data}\ \mathit{list}\ [&\mathit{nil} \mapsto [\,],\\ &\mathit{cons} \mapsto [\mathsf{Var}\ \mathit{list}, \mathsf{Var}\ \mathit{list}]]\end{aligned}$$

which is in fact a type of lists with two tails and no heads.

The solution is to state explicity that the instantiation of α must use the renaming type substitution defined earlier.

$$\mathsf{list}\ \alpha \ = \ \left(\begin{aligned}\mathsf{Data}\ \mathit{list}\ [&\mathit{nil} \mapsto [\,],\\ &\mathit{cons} \mapsto [\mathsf{Var}\ a, \mathsf{Var}\ \mathit{list}]]\end{aligned}\right)[\alpha/a]$$

Instantiating α to Var *list* with this definition and performing the substitution renames the bound *list*.

$$\begin{aligned}\mathsf{list}\ (\mathsf{Var}\ \mathit{list}) \ = \ \mathsf{Data}\ \mathit{list'}\ [&\mathit{nil} \mapsto [\,],\\ &\mathit{cons} \mapsto [\mathsf{Var}\ \mathit{list}, \mathsf{Var}\ \mathit{list'}]]\end{aligned}$$

As often happens, where there is possibility of renaming we can ensure that in practice this renaming will never occur. In particular, whenever such a substitution may occur, the rest of the program will ensure that the free variable is replaced by a closed type before the substitution needs to be evaluated.

4.8.2 Introducing the constructors

Out of the many possible fragments of code that can be formed from the SDT constructor syntax con, only two forms will be well typed with type list α according to the static semantics. These are

$$\mathsf{con}\ \mathit{nil}_{\mathsf{list}\ \alpha}\ [\,]$$

$$\mathsf{con}\ \mathit{cons}_{\mathsf{list}\ \alpha}\ [h\ ,t]$$

We define new constants to represent these constructors.

$$\begin{aligned}\mathsf{nil}_\alpha &= \mathsf{con}\ \mathit{nil}_{\mathsf{list}\ \alpha}\ []\\ \mathsf{cons}_\alpha\ h\ t &= \mathsf{con}\ \mathit{cons}_{\mathsf{list}\ \alpha}\ [h\ ,t]\end{aligned}$$

The following rules hold:

$$\frac{}{C \vdash \mathsf{nil}_\alpha : \mathsf{list}\ \alpha} \qquad \frac{C \vdash h : \alpha \qquad C \vdash t : \mathsf{list}\ \alpha}{C \vdash \mathsf{cons}_\alpha\ h\ t : \mathsf{list}\ \alpha}$$

It can also be proved that these introduced constants make no reductions and hence evaluate to themselves. The specialised case theorems are:

$$\frac{C \vdash e : \mathsf{list}\ \alpha \quad C \vdash (\bar{c}\ \mathit{nil}) : \beta \quad C \vdash (\bar{c}\ \mathit{cons}) : \alpha \rightarrow \mathsf{list}\ \alpha \rightarrow \beta}{C \vdash \mathsf{case}\ e\ \bar{c} : \beta}$$

$$\frac{}{\mathsf{case}\ \mathsf{nil}_\alpha\ \bar{c} \longrightarrow (\bar{c}\ \mathit{nil})}$$

$$\frac{}{\mathsf{case}\ (\mathsf{cons}_\alpha\ h\ t)\ \bar{c} \longrightarrow (\bar{c}\ \mathit{cons})\ h\ t}$$

4.8.3 Labelled transitions for lists

The three possible types of transitions for lists can be abbreviated by introducing three constants Nil, Hd and Tl with the definitions

$$\begin{aligned}\mathsf{Nil}_\alpha &= \mathsf{destL}\ \mathit{nil}_{\mathsf{list}\ \alpha}\ 0\\ \mathsf{Hd}_\alpha &= \mathsf{destL}\ \mathit{cons}_{\mathsf{list}\ \alpha}\ 1\\ \mathsf{Tl}_\alpha &= \mathsf{destL}\ \mathit{cons}_{\mathsf{list}\ \alpha}\ 2\end{aligned}$$

The rules for the transitions for lists are:

$$\frac{}{\mathsf{nil}_\alpha \xrightarrow{\mathsf{Nil}_\alpha} 0}$$

$$\frac{\vdash (\mathsf{cons}_\alpha\ h\ t) : \mathsf{list}\ \alpha}{(\mathsf{cons}_\alpha\ h\ t) \xrightarrow{\mathsf{Hd}_\alpha} h}$$

$$\frac{\vdash (\mathsf{cons}_\alpha\ h\ t) : \mathsf{list}\ \alpha}{(\mathsf{cons}_\alpha\ h\ t) \xrightarrow{\mathsf{Tl}_\alpha} t}$$

$$\frac{a \longrightarrow b \qquad b \xrightarrow{\alpha} c \qquad \vdash a : \mathsf{list}\ \alpha}{a \xrightarrow{\alpha} c}$$

The introduced constants and rules for lists are now identical to the rules given for PCF plus steams [Col96a], with the addition of the type argument in places to allow for the more expressive type system used here.

Chapter 5

Embedding the syntax and semantics

This chapter presents the formalisation in the HOL theorem prover of the language syntax and semantics discussed in the previous chapter. This is done by making a deep embedding of the syntax and semantics of the language [BGG+92] into HOL's logic. The abstract syntax of the types and expressions are represented by two new types in HOL. The substitution functions are implemented by functions in the HOL logic and the semantics are expressed as a series of relations in the logic.

5.1 Syntax

When using the HOL theorem prover there is a convention of taking a definitional approach to using logic. This has been discussed in chapter 2. The two definitions in this thesis that do not follow this convention are the definition, as new types in the logic, of the abstract syntax of SDT types and of expressions. These are introduced by axioms rather than deriving the characteristic properties. There is a large amount of work involved in introducing such syntax and while automated tools are provided for introducing many kinds of abstract syntax [Mel89], none of these tools can handle the inclusion of finite maps in the definitions. The form of the two axioms introduced is standard and it has been shown elsewhere that syntax including finite maps can be introduced into HOL definitionally [CS95].

5.1.1 Types

The syntax of types, as introduced into the HOL system, is given in figure 5.1. For the rest of this thesis however, the simplified notation first introduced in figure 4.1 is used.

		Syntax in HOL	Simplified Syntax
ty	$::=$	Num	Num
	$\mid$	$\mathsf{Var}\ id$	$\mathsf{Var}\ id$
	$\mid$	$\mathsf{All}\ id\ ty$	$\forall id.ty$
	$\mid$	$ty_1 \rightarrow ty_2$	$ty_1 \rightarrow ty_2$
	$\mid$	$\mathsf{Data}\ id\ (id, ty\ list) fmap$	$\mathsf{Data}\ id\ (id \mapsto [ty])$

Figure 5.1: The type of SDT types in HOL

This type is characterised by the following axiom.

Axiom 5.1 (Characteristic axiom for the type of types)

$$
\begin{array}{l}
\forall v\ n\ a\ d\ f.\\
\quad \exists! g.\\
\qquad (\forall x_1.\ g\ (\mathsf{Var}\ x_1) = v\ x_1) \land \\
\qquad (g\ \mathsf{Num}\ =\ n) \land \\
\qquad (\forall x_1\ x_2.\ g\ (\forall x_1.x_2)\ =\ a\ (g\ x_2)\ x_1\ x_2) \land \\
\qquad (\forall x\ \vec{m}.\ g\ (\mathsf{Data}\ x\ \vec{m})\ = d\ ((\mathsf{Map}\ g)\ \mathsf{o\_f}\ \vec{m})\ x\ \vec{m}) \land \\
\qquad (\forall x_1\ x_2.\ g\ (x_1 \rightarrow x_2)\ =\ f\ (g\ x_1)\ (g\ x_2)\ x_1\ x_2)
\end{array}
$$

This axiom provides a complete characterisation of the type. It expresses the fact that functions can be defined over the type using primitive recursion. These functions are introduced as the unique function g satisfying the equations formed by a particular choice for v, n, a, d and f in the axiom. This axiom, and the corresponding axiom for expressions are the only axioms added to HOL. All other definitions and results have been formally developed and proved within HOL. This axiom is standard apart from the case for datatypes, which depends on the correct treatment of the finite map. This case is very similar to the theorem that could have been derived automatically if the constructors for the datatype had taken a list as an argument instead of a finite map. With this change the datatype case would be:

$$\forall x_1\ x_2.\ g\ (\mathsf{DataList}\ x_1\ x_2)\ = d\ (\mathsf{Map}\ (\mathsf{Map}\ g \circ \mathsf{Snd})\ x_2)\ x_1\ x_2$$

In both the list and finite map versions, the recursive call of the function g is mapped across the list of types representing the arguments for each constructor. If the constructors for the datatype are represented by a list, then this function is mapped over the list of constructors by the outer application of the Map function. If the constructors are represented in a finite map, then the function Map g is applied to the range of the finite map. The list version could easily be added to HOL using Gunter's extension of the type definition package [Gun93].

From axiom 5.1 we can derive the induction theorem for types:

Theorem 5.2 (Structural induction for the type of types)

$$
\begin{array}{l}
\forall P. \\
\quad (\forall x.\ P\ (\mathsf{Var}\ x)) \wedge \\
\quad P\ \mathsf{Num} \wedge \\
\quad (\forall \alpha . P\ \alpha\ \supset\ (\forall x.\ P\ (\forall x.\alpha))) \wedge \\
\quad (\forall \alpha\ \beta.\ P\ \alpha\ \wedge\ P\ \beta\ \supset\ P\ (\alpha \rightarrow \beta)) \wedge \\
\quad (\forall y\ \bar{m}. \\
\qquad \mathsf{FEvery}\ (\lambda x.\mathsf{Every}\ P\ (\mathsf{Snd}\ x))\ \bar{m}\ \supset \\
\qquad P(\mathsf{Data}\ y\ \bar{m})) \\
\qquad\quad \supset \\
\quad (\forall \alpha.\ P\ \alpha)
\end{array}
$$

Proof. Follows from the axiom 5.1. The proof is closely modelled on that used by the tools supplied with HOL for proving the corresponding theorem for the types that can be introduced automatically.

The standard results about the distinctiveness and one to one properties of the constructors can be derived.

Theorem 5.3 *Distinctness of types*

$$
\begin{array}{l}
(\forall x.\ \mathsf{Num}\ \neq\ \mathsf{Var}\ x) \wedge \\
(\forall x\ \alpha.\ \mathsf{Num}\ \neq\ \forall x.\alpha) \wedge \\
(\forall \alpha\ \beta.\ \mathsf{Num}\ \neq\ \alpha \rightarrow \beta) \wedge \\
(\forall x\ \bar{m}.\ \mathsf{Num}\ \neq\ \mathsf{Data}\ x\ \bar{m}) \wedge \\
(\forall x\ v\ \alpha.\ \mathsf{Var}\ x\ \neq\ \forall v.\alpha) \wedge \\
(\forall x\ \alpha\ \beta.\ \mathsf{Var}\ x\ \neq\ \alpha \rightarrow \beta) \wedge \\
(\forall x\ v\ \bar{m}.\ \mathsf{Var}\ x\ \neq\ \mathsf{Data}\ v\ \bar{m}) \wedge \\
(\forall x\ \alpha\ \beta\ \gamma.\ \forall x.\alpha\ \neq\ \beta \rightarrow \gamma) \wedge \\
(\forall x\ v\ \alpha\ \beta.\ \forall x.\alpha\ \neq\ \mathsf{Data}\ v\ \bar{m}) \wedge \\
(\forall x\ \bar{m}\ \beta\ \gamma.\ \neg(\mathsf{Data}\ x\ \bar{m}\ \neq\ (\beta \rightarrow \gamma)))
\end{array}
$$

Theorem 5.4 *One-to-one property of types*

$$\begin{array}{l}
(\forall x\ y.\ (\mathsf{Var}\ x\ =\ \mathsf{Var}\ y)\ = (x\ =\ y))\ \wedge \\
(\forall x\ \alpha\ y\ \beta.\ (\forall x.\alpha\ =\ \forall y.\beta)\ =\ (x\ =\ y)\ \wedge\ (\alpha\ =\ \beta))\ \wedge \\
(\forall \alpha\ \beta\ \gamma\ \delta.\ ((\alpha \rightarrow \beta)\ = (\gamma \rightarrow \delta))\ =\ (\alpha\ =\ \gamma)\ \wedge\ (\beta\ =\ \delta))\wedge \\
(\forall x\ \tilde{m}\ y\ \tilde{n}.\ ((\mathsf{Data}\ x\ \tilde{m})\ = (\mathsf{Data}\ y\ \tilde{n}))\ =\ (x\ =\ y)\ \wedge\ (\tilde{m}\ =\ \tilde{n}))
\end{array}$$

Theorem 5.4 illustrates a problem with the given equality over these types; equality of two types requires bound variables to have the same names. But it is usual to equate types up to a renaming of the bound variables. Possible solutions to this problem and the approach taken here are discussed later in this chapter, after giving the definition of substitution necessary to formalise and reason about the renaming.

Using axiom 5.1 we can introduce recursive functions over types. One such function is the free type variable test. This function takes any type and a variable and tests whether that variable is free in the type. The definition makes use of the function Any, a disjunction operation over Boolean lists.

Definition 5.5 *The function* Any *is defined by:*

$$\begin{array}{l}
\mathsf{Any}\ [\,]\ =\ \mathsf{F} \\
\mathsf{Any}\ (\mathsf{Cons}\ x\ y)\ =\ x\ \vee\ \mathsf{Any}\ y
\end{array}$$

Using this the definition of the free variable function for types is:

Definition 5.6 *Free type variables*

$$\begin{array}{lcl}
\mathsf{ftv}\ (\mathsf{Var}\ x_1)\ x & = & (x\ =\ x_1) \\
\mathsf{ftv}\ \mathsf{Num}\ x & = & \mathsf{F} \\
\mathsf{ftv}\ (\forall x_1.x_2)\ x & = & (x\ \neq x_1)\ \wedge\ \mathsf{ftv}\ x_2\ x \\
\mathsf{ftv}\ (x_1 \rightarrow x_2)\ x & = & \mathsf{ftv}\ x_1\ x\ \vee\ \mathsf{ftv}\ x_2\ x \\
\mathsf{ftv}\ (\mathsf{Data}\ x_1\ \tilde{m})\ x & = & (x \neq x_1)\ \wedge \\
 & & \exists y.\ \mathsf{FRange}\ \tilde{m}\ y\ \wedge \mathsf{Any}\ (\mathsf{Map}\ (\lambda z.\mathsf{ftv}\ z\ x)\ y)
\end{array}$$

5.1.2 Expressions

The syntax of expressions as introduced into HOL is given in figure 5.2. For the rest of this thesis we will use the simplified mathematical notation for the syntax except where there is a risk of confusion with the HOL syntax for quantification and abstraction. The syntax for the constructors and case expressions was introduced in the previous chapter. The num, nnnop and nnbop constructs provide a means for lifting numbers from the HOL logic for use in SDT.

		Syntax in HOL	Simplified Syntax (where different)
exp	::=	num *num*	
	|	nop (*num* → *num* → *num*) *num num*	
	|	bop (*num* → *num* → *bool*) *num num*	
	|	var *id*	
	|	lambda *id ty exp*	$\lambda id : ty.\ exp$
	|	app exp_1 exp_2	$exp_1\ exp_2$
	|	Lambda *id ty*	$\Lambda id.\ exp$
	|	App *exp ty*	exp_{ty}
	|	rec *id ty exp*	rec id_{ty} *exp*
	|	con *id ty* (*exp list*)	con id_{ty} [*exp*]
	|	case *exp* (*id*, *exp*)*fmap*	case *exp* (*id* ↦ *exp*)

Figure 5.2: The type of SDT expressions in HOL

This type is characterised by an axiom added to the logic in a similar way to axiom 5.1.

Axiom 5.7 *Characteristic theorem for expressions*

$$
\begin{array}{l}
\forall nc\ nnc\ nbc\ vc\ lc\ rc\ ltc\ ac\ atc\ cc\ cac.\\
\quad \exists! y.\\
\quad\quad (\forall x_1.\ y\ (\mathsf{num}\ x_1)\ =\ nc\ x_1)\ \wedge\\
\quad\quad (\forall x_1\ e_1\ e_2.\ y\ (\mathsf{nop}\ x_1\ e_1\ e_2)\ =\ nnc\ (y\ e_1)\ (y\ e_2)\ x_1\ e_1\ e_2)\ \wedge\\
\quad\quad (\forall x_1\ e_1\ e_2.\ y\ (\mathsf{bop}\ x_1\ e_1\ e_2)\ =\ nbc\ (y\ e_1)\ (y\ e_2)\ x_1\ e_1\ e_2)\ \wedge\\
\quad\quad (\forall x_1.\ y\ (\mathsf{var}\ x_1)\ =\ vc\ x_1)\ \wedge\\
\quad\quad (\forall x_1\ x_2\ x_3.\ y\ (\lambda x_1 : x_2.\ x_3)\ =\ lc\ (y\ x_3)\ x_1\ x_2\ x_3)\ \wedge\\
\quad\quad (\forall x\ t\ x_1.\ y\ (\mathsf{rec}\ x_t\ x_1)\ =\ rc\ (y\ x_1)\ x\ t\ x_1)\ \wedge\\
\quad\quad (\forall x_1\ x_2.\ y\ (\Lambda x_1.\ x_2)\ =\ ltc\ (y\ x_2)\ x_1 x_2)\ \wedge\\
\quad\quad (\forall x_1\ x_2.\ y\ (x_1\ x_2)\ =\ ac\ (y\ x_1)\ (y\ x_2)\ x_1\ x_2)\ \wedge\\
\quad\quad (\forall x_1\ t.\ y\ (x_1 t)\ =\ atc\ (y\ x_1)\ x_1\ t)\ \wedge\\
\quad\quad (\forall x\ t\ x_1.\ y\ (\mathsf{con}\ x_t\ x_1)\ =\ cc\ (\mathsf{Map}\ y\ x_1)\ x\ t\ x_1)\ \wedge\\
\quad\quad (\forall x\ \bar{m}.\ y\ (\mathsf{case}\ x\ \bar{m})\ =\ cac\ (y\ x)\ (y\ \mathsf{o\_f}\ \bar{m})\ x\ \bar{m})
\end{array}
$$

From this axiom, similar theorems to those derived from the axiom for types can be derived. These are the induction theorem for expressions and theorems stating that the constructors are distinct and there is a one-to-one correspondence.

Axiom 5.7 allows the definition of free variable functions for expressions. There are two functions, one to test for free type variables in expressions and one to test for free expression variables.

Definition 5.8 *Free expression variables in expressions.*

$$\begin{array}{lcl}
\mathsf{fv}\ (\mathsf{var}\ v)\ x & = & (x = v) \\
\mathsf{fv}\ (\mathsf{num}\ n)\ x & = & \mathsf{F} \\
\mathsf{fv}\ (\mathsf{nop}\ n\ e_1\ e_2)\ x & = & (\mathsf{fv}\ e_1\ x)\ \vee\ (\mathsf{fv}\ e_2\ x) \\
\mathsf{fv}\ (\mathsf{bop}\ b\ e_1\ e_2)\ x & = & (\mathsf{fv}\ e_1\ x)\ \vee\ (\mathsf{fv}\ e_2\ x) \\
\mathsf{fv}\ (e_1\ e_2)\ x & = & (\mathsf{fv}\ e_1\ x)\ \vee\ (\mathsf{fv}\ e_2\ x) \\
\mathsf{fv}\ (t_e)\ x & = & (\mathsf{fv}\ e\ x) \\
\mathsf{fv}\ (\lambda y : t.\ e)\ x & = & (\mathsf{fv}\ e\ x)\ \wedge\ (x \neq y) \\
\mathsf{fv}\ (\Lambda y.\ e)\ x & = & \mathsf{fv}\ e\ x \\
\mathsf{fv}\ (\mathsf{rec}\ y_t\ e)\ x & = & \mathsf{fv}\ e\ x\ \wedge\ (x \neq y) \\
\mathsf{fv}\ (\mathsf{con}\ y_t\ ys)\ x & = & \mathsf{Any}\ (\mathsf{Map}\ (\lambda z.\mathsf{fv}\ z\ x)\ ys) \\
\mathsf{fv}\ (\mathsf{case}\ e\ \vec{m})\ x & = & (\mathsf{fv}\ e\ x)\ \vee\ (\exists y.\ (\mathsf{FRange}\ \vec{m}\ y)\ \wedge\ (\mathsf{fv}\ y\ x))
\end{array}$$

Definition 5.9 *Free type variables in expressions.*

$$\begin{array}{lcl}
\mathsf{ftve}\ (\mathsf{var}\ v)\ x & = & \mathsf{F} \\
\mathsf{ftve}\ (\mathsf{num}\ n)\ x & = & \mathsf{F} \\
\mathsf{ftve}\ (\mathsf{nop}\ n\ e_1\ e_2) & = & (\mathsf{ftve}\ e_1\ x)\ \vee\ (\mathsf{ftve}\ e_2\ x) \\
\mathsf{ftve}\ (\mathsf{bop}\ n\ e_1\ e_2) & = & (\mathsf{ftve}\ e_1\ x)\ \vee\ (\mathsf{ftve}\ e_2\ x) \\
\mathsf{ftve}\ (e_1\ e_2) & = & (\mathsf{ftve}\ e_1\ x)\ \vee\ (\mathsf{ftve}\ e_2\ x) \\
\mathsf{ftve}\ (e_{1t_1}) & = & (\mathsf{ftve}\ e_1\ x)\ \vee\ (\mathsf{ftv}\ t_1\ x) \\
\mathsf{ftve}\ (\lambda y : t_1.\ e_1) & = & (\mathsf{ftve}\ e_1\ x)\ \vee\ (\mathsf{ftv}\ t_1\ x) \\
\mathsf{ftve}\ (\Lambda y.\ e_1) & = & (\mathsf{ftve}\ e_1\ x)\ \wedge\ (x \neq y) \\
\mathsf{ftve}\ (\mathsf{rec}\ y_{t_1}\ e_1) & = & (\mathsf{ftve}\ e_1\ x)\ \vee\ (\mathsf{ftv}\ t_1\ x) \\
\mathsf{ftve}\ (\mathsf{con}\ y_t\ ys)\ x & = & (\mathsf{ftv}\ t\ x)\ \vee\ (\mathsf{Any}\ (\mathsf{Map}\ (\lambda y.\ \mathsf{ftve}\ y\ x)\ ys)) \\
\mathsf{ftve}\ (\mathsf{case}\ e\ \vec{m})\ x & = & (\mathsf{ftve}\ e\ x)\ \vee\ (\exists y.\ (\mathsf{FRange}\ \vec{m}\ y)\ \wedge\ (\mathsf{ftve}\ \ y\ x))
\end{array}$$

5.1.3 Adding booleans

The syntax for types and expressions introduced in the last section contains the syntactic constructs necessary to lift the HOL type of numbers for use in SDT programs. The syntax does not make any such provision for the booleans, which will be defined in terms of datatypes as discussed in section 4.2. The definition of booleans is:

Definition 5.10 *The three constants in HOL representing the boolean type and constructors are:*

$$\begin{aligned}\mathsf{Bool} &= \mathsf{Data}\ \textit{“bool”}\ [\ \textit{“true”} \mapsto [\ ],\ \textit{“false”} \mapsto [\ ]\ ]\\ \mathsf{True} &= \mathsf{con}\ \textit{“true”}_{\mathsf{Bool}}\ [\ ]\\ \mathsf{False} &= \mathsf{con}\ \textit{“false”}_{\mathsf{Bool}}\ [\ ]\end{aligned}$$

Constants for the booleans are introduced before the semantics of the language is defined so that the rules can, for clarity, be defined in terms of these constants. This is not essential. The rules for binary relations could be given directly in terms of the primitive syntax for datatypes and then rewritten with the definitions Bool, True and False later.

5.2 Substitutions

The six substitution functions were introduced in section 4.4. Their definition is necessary in order to define both the static and dynamic semantics, and to express and reason about many of the meta-theoretic results. The approach here is to define a simultaneous substitution function, taking a substitution represented as a finite map of the appropriate type. Substitution for a single variable is then defined as a special case. Only these simpler substitutions are used in the definition of the static and dynamic semantics, but simultaneous substitutions are needed for the meta-theory.

The approach here is similar to that used by others to embed imperative programs [HM94] and the π-calculus [Mel94, GM96] in HOL. The definition of substitution depends on the definition of a function to pick new variables distinct from those free in a term.

5.2.1 A choice function

Before the definition of the functions can be given it is necessary to deal with the possibility of variable capture when substituting under a binding construct. A renaming function is defined which takes a variable, x, and a set of variables, s, and returns a variant of x that is not in s. Providing the set is finite it will always be possible to find such a variant. If x is not in s then the variant will be x itself. The set of variables is represented by its characteristic function. For any finite set L, the two key properties of the choice function, ch, are:

$$\forall x\ L.\ \mathsf{ch}(x,\ L)\ \notin\ L$$

$$\forall x\ L.\ \mathsf{ch}\ (x,\ L)\ =\ (x\ \in L\ \Rightarrow\ \mathsf{ch}\ (\mathsf{prime}\ x,\ L)\ \mid\ x)$$

where prime is the function that adds a $'$ to the string x.

The choice function is not a primitive recursive function and cannot be defined easily using the standard function definition package in HOL. Instead it is defined using Konrad Slind's TFL package [Sli96], the documentation for which contains a similar function definition.

Definition 5.11 *The choice function is introduced with the property*

$$\mathsf{ch}\ (x,\ L)\ =\ (x\ \in L\ \wedge \mathsf{Finite}\ L\ \Rightarrow\ \mathsf{ch}\ (\mathsf{prime}\ x,\ L)\ \mid\ x)$$

under the assumption that the function is terminating. This assumption is expressed by a measure which is assumed to be decreasing. The measure is

$$\lambda(x, L).\ \mathsf{CARD}\ (\{y \mid \mathsf{SLENGTH}\ x\ \leq\ \mathsf{SLENGTH}\ y\}\ \cap\ L)$$

where CARD *is the cardinality of a set and* SLENGTH *is the length of a string.*

To discharge the assumption that the measure is decreasing the following lemma is proved.

Lemma 5.12

$$\begin{array}{l} \forall x\ L.\ \mathsf{Finite}\ L\ \supset \\ \qquad x\ \in\ L \supset \\ \qquad\quad \mathsf{CARD}\ (\{y \mid \mathsf{SLENGTH}\ (\mathsf{prime}\ x)\ \leq\ \mathsf{SLENGTH}\ y\}\ \cap\ L)\ < \\ \qquad\quad \mathsf{CARD}\ (\{y \mid \mathsf{SLENGTH}\ x\ \leq\ \mathsf{SLENGTH}\ y\}\ \cap\ L) \end{array}$$

Proof By induction over the finite set L and simplification.

This allows the required properties of the choice function to be proved.

Theorem 5.13 *The choice function,* ch, *has the property:*

$$\begin{array}{l} \forall x\ L.\ \mathsf{Finite}\ L\ \supset \\ \quad \mathsf{ch}\ (x,\ L)\ =\ (x\ \in L\ \Rightarrow\ \mathsf{ch}\ (\mathsf{prime}\ x,\ L)\ \mid\ x) \end{array}$$

Proof. Follows from definition 5.11 and lemma 5.12.

An induction principle can be derived from the definition of the choice function

Theorem 5.14 (Recursion induction)

$$\begin{array}{l} P. \\ \quad (\forall x\ L.\ (x \in L\ \wedge\ \mathsf{Finite}\ L\ \supset\ P\ (\mathsf{prime}\ x,\ L))\ \supset\ P\ (x,\ L))\ \supset \\ \quad (\forall x\ L.\ P(x, L)) \end{array}$$

Proof. Simplification of the induction theorem introduced with definition 5.11 by lemma 5.12.

Finally the correctness of the choice function can be stated.

Theorem 5.15

$$\forall x\ L.\ \mathsf{ch}(x,\ L)\ \notin\ L$$

Proof. By induction using theorem 5.14 and simplification using theorem 5.13.

This choice function is used to rename bound variables to avoid capture. If a type α with a free type variable x is substituted into the type $\forall x.\beta$ then the free variable x is 'captured' by the binding $\forall x$. To avoid this the bound variable is renamed to a variable that cannot occur free in the body of the term after the substitution.

If f is the appropriate free variable test (fv, ftv or ftve) then the set of variables in an abstraction with the body t and bound variable x is

$$\{y \mid f\ t\ y\ \wedge\ y \neq x\}$$

The free variables after substitution are the variables in this set which are not replaced by the substitution, plus the free variables in the image of this set under the substitution. Two new functions freeR and free capture this informal description in HOL.

Definition 5.16 *If f is a free variable function and $\bar{s}$ a substitution then* freeR *tests for the free variables in the range of $\bar{s}$ and is defined by:*

$$\mathsf{freeR}\ f\ \bar{s}\ x\ =\ \exists e.\ \mathsf{FRange}\ \bar{s}\ e\ \wedge f\ e\ x$$

Definition 5.17 *If f is a free variable function, $\bar{s}$ a substitution and g represents a set of variables, then* free *tests for the free variables in the image of the set represented by g under the substitution $\bar{s}$ or in the variables of g that are not replaced by $\bar{s}$.* free *is defined by:*

$$\mathsf{free}\ f\ \bar{s}\ g\ x\ =\ (\mathsf{freeR}\ f\ (\mathsf{DRestrict}\ \bar{s}\ g)\ x)\ \vee\ (g\ x\ \wedge\ \neg(\mathsf{FDom}\ \bar{s}\ x))$$

For a substitution $\bar{s}$ applied to a type $\forall x.\beta$ or any similar binding construct, the renaming of the bound variable is chosen to be:

$$\mathsf{ch}\ (x,\ \mathsf{free}\ f\ \bar{s}\ (\lambda y.\ f\ \beta\ y\ \wedge\ y \neq x))$$

For the example of a type, $\forall x.\beta$, the function f would be ftv.

It remains only to show that this new variable is distinct from any of the free variables that could be captured.

We begin with a series of lemmas about the finiteness properties of the free variable functions, lists, finite maps and the functions freeR and free. The first two are not of any real significance themselves but are included to illustrate a pattern that occurs in many later proofs and will not be shown in detail again. The important result is lemma 5.20, where the quantification is over expressions and types in the language. As some of the syntactic constructs contain finite maps and lists, some steps of an inductive proof for these constructs may require further inductive proofs over the lists and finite maps. The first two results are used in these proofs.

Lemma 5.18 *If f is a free variable function and ys is a list then*

$$\mathsf{Every}\ (\lambda e.\ \mathsf{Finite}\ (f\ e))\ ys\ \supset\ \mathsf{Finite}(\lambda x.\ \mathsf{Any}\ (\mathsf{Map}\ ((\lambda z.\ z\ x)\circ f)\ ys))$$

Proof. Follows by an easy structural induction over the list ys.

Lemma 5.19 *If f is a free variable function for terms of type α and $\bar{m}$ is finite map from variables to a list of type α, then*

$$(\forall x.\ \mathsf{FDom}\ \bar{m}\ x\ \supset\ \mathsf{Every}\ (\lambda e.\ \mathsf{Finite}\ (f\ e))(\bar{m}\ x))\ \supset$$
$$\mathsf{Finite}\ (\lambda x.\ \exists y.\ \mathsf{FDom}\ \bar{m}\ y\ \wedge\ \mathsf{Any}\ (\mathsf{Map}\ ((\lambda z.\ z\ x)\circ f)\ (\bar{m}\ y)))$$

Proof. Follows by an easy induction over the finite map $\bar{m}$ and lemma 5.18.

Lemma 5.20 *Each free variable function* ftv, fv *and* ftve, *produces only a finite number of free variables.*

$$\forall\alpha.\ \mathsf{Finite}\ (\mathsf{ftv}\ \alpha)$$

$$\forall e.\ \mathsf{Finite}\ (\mathsf{fv}\ e)$$

$$\forall e.\ \mathsf{Finite}\ (\mathsf{ftve}\ e)$$

Proof. Each lemma is proved by a structural induction over the term or type using lemmas 5.18 and 5.19.

The following two lemmas capture the finiteness properties of the function freeR and free. The assumption

$$(\forall e.\ \mathsf{Finite}\ (f\ e))$$

that appears in these lemmas can be discharged by the above results and so the lemmas can be instantiated for each free variable function.

Lemma 5.21

$$\forall f.\ (\forall e.\ \mathsf{Finite}\ (f\ e))\ \supset\ (\forall \bar{s}.\ \mathsf{Finite}\ (\mathsf{freeR}\ f\ \bar{s}))$$

Proof. Follows by an easy induction over the finite map $\bar{s}$.

Lemma 5.22

$$\forall f g.\ (\forall e.\ \mathsf{Finite}\ (f\ e))\ \wedge \mathsf{Finite}\ g\ \supset\ (\forall \bar{s}.\mathsf{Finite}\ (\mathsf{free}\ f\ \bar{s}\ g))$$

Proof. Follows from definition 5.17 and lemma 5.21.

Finally, theorem 5.13 can be instantiated to the particular set of variables used here and the assumption about the finiteness of this set simplified using theorem 5.22. This gives the behaviour of the choice function as it will be used in the next section.

Theorem 5.23

$$\begin{array}{l}
\forall f.\ (\forall e.\ \mathsf{Finite}\ (f\ e))\ \supset \\
\quad (\forall e\ \bar{s}\ v. \\
\qquad \mathsf{ch}\ (v,\ \mathsf{free}\ f\ \bar{s}\ (\lambda x.\ f\ e\ x\ \wedge\ (x \neq v))) \\
\qquad\quad = \\
\qquad ((v\ \in\ (\mathsf{free}\ f\ \bar{s}\ (\lambda x.\ f\ e\ x\ \wedge\ (x \neq v)))) \\
\qquad \Rightarrow\ (\mathsf{ch}\ (\mathsf{prime}\ v,\ (\mathsf{free} f\ s\ (\lambda x.\ f\ e\ x\ \wedge\ (x \neq v))))) \\
\qquad \mid\ v))
\end{array}$$

As with the previous two theorems, the function f can be instantiated to any of the free variable functions and the assumption about the finiteness of f discharged.

A useful property of the choice function is that if it is applied to a variable and an empty set of variables, then the function returns the original variable. The empty set of variables is represented by the function $\lambda x.\mathsf{F}$ that always returns false.

Theorem 5.24

$$\forall x.\ \mathsf{ch}\ (x,\ \lambda x.\mathsf{F})\ =\ x$$

Proof. A simple corollary of theorem 5.13

5.2.2 The substitution functions

The definition of the simultaneous substitution function

$$\mathsf{ttSubs} : ty \rightarrow (string, ty)fmap \rightarrow ty$$

which substitutes types into types is given figure 5.3.The application of a substitution $\bar{s}$ to a type α, ttSubs $\alpha\ \bar{s}$, is abbreviated by $[\alpha]_{\bar{s}}$.
From this, the single substitution function

$$\mathsf{ttSub} : ty \rightarrow (string, ty) \rightarrow ty$$

is easily defined. If the substitution of β for x in the type α, ttSub $\alpha\ (x, \beta)$, is abbreviated by $\alpha[\beta/x]$, the definition is as follows.

Definition 5.25 (Substitution for a single variable)

$$\alpha[\beta/x] \ = \ [\alpha]_{[x \mapsto \beta]}$$

The substitution functions for types into expressions and expressions into expressions are similar. Their definitions are given in figures 5.4 and 5.5. From these the single substitution functions can be defined in exactly the same way as for types.

5.3 Properties of substitution

This section gives some of the properties of the substitution functions. Unless specifically mentioned, these properties have been proved for all the varieties of substitution. Many of the following results are concerned with simplifying the mappings used in substitution. This is essential for many of the proofs in later sections.

The first result states that a substitution can be restricted in its domain to the free variables in the term it is applied to.

$$\begin{array}{rcll}
[\mathsf{Var}\ v]_{\bar{s}} & = & ((\mathsf{FDom}\ \bar{s}\ v) \Rightarrow\ \bar{s}\ v \mid \mathsf{Var}\ v) & (5.1)\\
[\mathsf{Num}]_{\bar{s}} & = & \mathsf{Num} & (5.2)\\
[\forall v.\alpha]_{\bar{s}} & = & \mathsf{let}\ y\ =\ \mathsf{ch}\ (v,\ \mathsf{free}\ \mathsf{ftv}\ \bar{s}\ (\lambda x.\ \mathsf{ftv}\ \alpha\ x\ \wedge\ x \neq v)) & \\
 & & \mathsf{in}\ \ \forall y.[\alpha]_{\bar{s}[v \mapsto \mathsf{Var}\ y]} & (5.3)\\
[\alpha \rightarrow \beta]_{\bar{s}} & = & [\alpha]_{\bar{s}} \rightarrow [\beta]_{\bar{s}} & (5.4)\\
[\mathsf{Data}\ x\ \bar{f}]_{\bar{s}} & = & \mathsf{let}\ y = \mathsf{ch}\ (v,\ \mathsf{free}\ \mathsf{ftv}\ \bar{s}\ (\lambda x.\ \mathsf{ftv}\ \alpha\ x\ \wedge\ x \neq v)) & \\
 & & \mathsf{in}\ \ \mathsf{Data}\ x\ (\mathsf{Map}\ (\lambda\alpha.\ [\alpha]_{\bar{s}[x \mapsto \mathsf{Var}\ y]})\ \mathsf{o\_f}\ \bar{f}) & (5.5)
\end{array}$$

Figure 5.3: Substituting types into types

$$\begin{aligned}
[\mathsf{num}\ n]_{\bar{s}} &= \mathsf{num}\ n \tag{5.6}\\
[\mathsf{nop}\ n\ e_1\ e_2]_{\bar{s}} &= \mathsf{nop}\ n\ [e_1]_{\bar{s}}\ [e_2]_{\bar{s}} \tag{5.7}\\
[\mathsf{bop}\ n\ e_1\ e_2]_{\bar{s}} &= \mathsf{bop}\ n\ [e_1]_{\bar{s}}\ [e_2]_{\bar{s}} \tag{5.8}\\
[\mathsf{var}\ v]_{\bar{s}} &= (\mathsf{FDom}\ \bar{s}\ v\ \Rightarrow\ \bar{s}\ v\ \mid\ \mathsf{var}\ v) \tag{5.9}\\
[\lambda y : \alpha.\ e]_{\bar{s}} &= \mathsf{let}\ z\ =\ (\mathsf{ch}\ (y, (\mathsf{free}\ \mathsf{fv}\ \bar{s}\ (\lambda x.\ \mathsf{fv}\ e\ x\ \wedge\ x \neq y))))\\
&\quad\ \mathsf{in}\ (\lambda z : \alpha.\ [e]_{\bar{s}[y \mapsto \mathsf{var}\ z]}) \tag{5.10}\\
[\Lambda x.\ e_1]_{\bar{s}} &= \Lambda x.\ [e_1]_{\bar{s}} \tag{5.11}\\
[e_1\ e_2]_{\bar{s}} &= [e_1]_{\bar{s}}\ [e_2]_{\bar{s}} \tag{5.12}\\
[e_{1\alpha}]_{\bar{s}} &= [e_1]_{\bar{s}\alpha} \tag{5.13}\\
[\mathsf{rec}\ y_\alpha\ e]_{\bar{s}} &= \mathsf{let}\ z\ =\ (\mathsf{ch}\ (y, (\mathsf{free}\ \mathsf{fv}\ \bar{s}\ (\lambda x.\ \mathsf{fv}\ e\ x\ \wedge\ x \neq y))))\\
&\quad\ \mathsf{in}\ (\mathsf{rec}\ z_\alpha\ [e]_{\bar{s}[y \mapsto \mathsf{var}\ z]}) \tag{5.14}\\
[\mathsf{con}\ c_\alpha\ xs]_{\bar{s}} &= \mathsf{con}\ c_\alpha\ (\mathsf{Map}\ (\lambda x.\ [x]_{\bar{s}})\ xs) \tag{5.15}\\
[\mathsf{case}\ e\ \bar{m}]_{\bar{s}} &= \mathsf{case}\ [e]_{\bar{s}}\ ((\lambda e_1.\ [e1]_{\bar{s}})\ \mathsf{o\_f}\ \bar{m}) \tag{5.16}
\end{aligned}$$

Figure 5.4: Substituting expressions into expressions

$$\begin{aligned}
[\mathsf{num}\ n]_{\bar{s}} &= \mathsf{num}\ n \tag{5.17}\\
[\mathsf{nop}\ n\ e_1\ e_2]_{\bar{s}} &= \mathsf{nop}\ n\ [e_1]_{\bar{s}}\ [e_2]_{\bar{s}} \tag{5.18}\\
[\mathsf{bop}\ n\ e_1\ e_2]_{\bar{s}} &= \mathsf{bop}\ n\ [e_1]_{\bar{s}}\ [e_2]_{\bar{s}} \tag{5.19}\\
[\mathsf{var}\ v]_{\bar{s}} &= \mathsf{var}\ v \tag{5.20}\\
[\lambda y : \alpha.\ e]_{\bar{s}} &= \lambda y : [\alpha]_{\bar{s}}.\ [e]_{\bar{s}} \tag{5.21}\\
[\Lambda x.\ e_1]_{\bar{s}} &= \mathsf{let}\ y\ =\ \mathsf{ch}\ (x,\ \mathsf{free}\ \mathsf{ftv}\ \bar{s}\ (\lambda z.\ \mathsf{ftve}\ e_1\ z\ \wedge\ x \neq z))\\
&\quad\ \mathsf{in}\ \ (\Lambda y.\ [e_1]_{\bar{s}[x \mapsto \mathsf{Var}\ y]}) \tag{5.22}\\
[e_1\ e_2]_{\bar{s}} &= [e_1]_{\bar{s}}\ [e_2]_{\bar{s}} \tag{5.23}\\
[e_{1\alpha}]_{\bar{s}} &= [e_1]_{\bar{s}\alpha} \tag{5.24}\\
[\mathsf{rec}\ y_\alpha\ e]_{\bar{s}} &= \mathsf{rec}\ y_{[\alpha]_{\bar{s}}}\ [e]_{\bar{s}} \tag{5.25}\\
[\mathsf{con}\ c_\alpha\ xs]_{\bar{s}} &= \mathsf{con}\ c_{[\alpha]_{\bar{s}}}\ (\mathsf{Map}\ (\lambda x.\ [x]_{\bar{s}})\ xs) \tag{5.26}\\
[\mathsf{case}\ e\ \bar{m}]_{\bar{s}} &= \mathsf{case}\ [e]_{\bar{s}}\ ((\lambda e_1.\ [e1]_{\bar{s}})\ \mathsf{o\_f}\ \bar{m}) \tag{5.27}
\end{aligned}$$

Figure 5.5: Substituting types into expressions

Theorem 5.26 *If* $[\_]$ *is a substitution function and* f *a free variable function then*

$$\forall t\ \bar{s}.\ [t]_{\bar{s}} \ = \ [t]_{\mathsf{DRestrict}\ \bar{s}\ (f\ t))}$$

Proof. By structural induction over types or expressions, depending on which of the three substitution functions is being considered.

It can be shown that the identity substitution, which maps variables to themselves, leaves a term unchanged.

Theorem 5.27 *If* $[\_]$ *is the function for substituting types into types or expressions then*

$$\forall t\ \bar{s}.\ (\forall x.\ \mathsf{FDom}\ \bar{s}\ x\ \supset\ \bar{s}\ x = \mathsf{Var}\ x)\ \supset\ [t]_{\bar{s}} = t$$

If $[\_]$ *is the function for substituting expressions into expressions, then*

$$\forall t\ \bar{s}.\ (\forall x.\ \mathsf{FDom}\ \bar{s}\ x\ \supset\ \bar{s}\ x = \mathsf{var}\ x)\ \supset\ [t]_{\bar{s}} = t$$

Proof. By structural induction over types or expressions, according to which substitution function is used, and then simplification with the definitions of ch, free and freeR. The proof relies on the fact that an identity substitution cannot cause renaming of variables.

It follows from this that the empty substitution also leaves a term unchanged.

Theorem 5.28

$$\forall t.\ [t]_{\mathsf{FEmpty}} \ = \ t$$

Proof. Follows from theorem 5.27, since the empty mapping satisfies the property

$$\forall x.\ \mathsf{FDom}\ \mathsf{FEmpty}\ x\ \supset\ \mathsf{FEmpty}\ x \ = \ \mathsf{var}\ x$$

Another immediate consequence of theorem 5.27 is that replacing a variable by itself leaves the term unchanged.

Theorem 5.29

$$\forall x\ t.\ t[\mathsf{Var}\ x/x] \ = \ t$$

Proof. Follows immediately from theorem 5.27 and the definition of single substitution.

The final result is that if two terms are equal under all substitutions then they are the same term.

Theorem 5.30

$$\forall t_1\ t_2.\ (\forall s.[t_1]_s = [t_2]_s) \supset (t_1 = t_2)$$

Proof. Since $[t_1]_s = [t_2]_s$ holds for all substitutions s it also holds for the empty substitution. This gives $[t_1]_{\mathsf{FEmpty}} = [t_2]_{\mathsf{FEmpty}}$. The result follows from theorem 5.28.

The above theorems have been proved without having to deal with renaming of variables, since the identity and empty substitutions do not cause renaming. Another important class of substitutions that do not cause renaming are those where the range consists only of terms with no free variables. The next section considers how to formalise this and the results that can be proved.

5.4 Closed substitutions

In practice the need to consider whether or not to rename variables can greatly complicate proofs. But in almost all cases, this can be eliminated by considering properties of the expressions that we are substituting into the program. In particular, if the expression being introduced contains no free variables, then variable capture cannot occur.

The key idea is that at the top level of a program we have well-typed functions (which can be shown to have no free variables) and that these are applied to closed terms as arguments. The application generates a new substitution with a closed term in the range. As substitutions are propagated throughout the term, all the terms in the range of the substitution are closed. When looking at a subprogram in isolation there may be free variables, but we can work under the assumption that these variables will always be replaced by a closed term. This assumption will be propagated through the proofs and will eventually be discharged.

There are a number of exceptions to this ideal model. The first is the creation of identity substitutions when renaming does not occur. This can happen in all the substitution functions where the choice function is used. Any substitution containing identity mappings can be proved equal to a substitution without; this is detailed later in this section.

A more significant exception is that while function application will behave as described above, type application may cause a renaming because a well-typed expression may contain free type variables. In section 5.5.1, some results will be considered for substitutions that map to types where renaming does occur.

Before examining the properties of the substitution functions, we introduce some predicates to capture the idea of closed substitutions and to simplify the presentation of results and propagation of information.

Definition 5.31 *The fact that a term t is closed with respect to some free variable function f is stated by:*

$$\mathsf{Closed}\ f\ t\ =\ \forall x.\ \neg(f\ t\ x)$$

One important and frequently used property of a closed term is that it is unchanged by any substitution. This can be shown for each of the substitution functions and f is used to stand for the appropriate free variable function.

Theorem 5.32

$$\forall t.\ \mathsf{Closed}\ f\ t\ \supset\ (\forall \bar{s}.\ [t]_{\bar{s}}\ =\ t)$$

Proof. By theorem 5.26, the domain of the substitution $\bar{s}$ can be restricted to the free variables in t. As t is closed, there are no free variables, so $[t]_{\bar{s}}$ is equal to $[t]_{\mathsf{FEmpty}}$. The result then follows from theorem 5.28, which states that the empty substitution has no effect on a term.

From the definition of a closed term, we can define a closed substitution.

Definition 5.33 *For any free variable function f*

$$\mathsf{FClosed}\ f\ \bar{s}\ =\ (\forall y.\ \mathsf{FDom}\ \bar{s}\ y\ \supset\ \mathsf{Closed}\ f\ (\bar{s}\ y))$$

A closed finite map can be built from an empty finite map, which is closed, by adding any closed terms to the range of the mapping. The following results capture this construction.

Theorem 5.34

$$\forall f.\ \mathsf{FClosed}\ f\ \mathsf{FEmpty}$$

$$\forall f\ \bar{s}\ \alpha.\ \mathsf{FClosed}\ f\ \bar{s}\ \wedge\ \mathsf{Closed}\ f\ \alpha\ \supset\ (\forall x.\ \mathsf{FClosed}\ f\ (\bar{s}[x \mapsto\ \alpha]))$$

Proof. Both results are proved by simplifying with the definitions of closed terms and simple properties of finite maps.

For any expression, if there is a closed finite map whose domain covers all of the free variables then we sometimes refer to this finite map as *closing substitution* for the expression.

Definition 5.35

$$\forall f\ \bar{s}\ \alpha.\ \mathsf{Closing}\ f\ \bar{s}\ \alpha\ =\ \mathsf{FClosed}\ f\ \bar{s}\ \wedge\ (\forall x.\ f\ \alpha\ x\ \supset\ \mathsf{FDom}\ \bar{s}\ x)$$

The definition of FClosed substitutions allows us to prove properties which state that they do not cause renaming. For the substitution into types, the two clauses where a substitution is moved through a bound variable can, when the substitution is closed, be simplified as follows.

Theorem 5.36

$$\begin{array}{l} \forall v\ t\ \bar{s}. \\ \quad \mathsf{FClosed}\ \mathsf{ftv}\ \bar{s}\ \supset \\ \qquad [\forall v.t]_{\bar{s}} \;=\; \forall v.[t]_{\bar{s}[v \mapsto \mathsf{Var}\ v]} \end{array}$$

$$\begin{array}{l} \forall x\ \bar{f}\ \bar{s}. \\ \quad \mathsf{FClosed}\ \mathsf{ftv}\ \bar{s}\ \supset \\ \qquad [\mathsf{Data}\ x\ \bar{f}]_{\bar{s}} = \mathsf{Data}\ x\ (\mathsf{Map}\ (\lambda t.\ [t]_{\bar{s}[x \mapsto \mathsf{Var}\ x]})\ \mathsf{o\_f}\ \bar{f}) \end{array}$$

Theorem 5.37

$$\begin{array}{l} \forall y\ \bar{s}. \\ \quad \mathsf{FClosed}\ \mathsf{fv}\ \bar{s}\ \supset \\ \qquad [\lambda y : \alpha.\ e]_{\bar{s}} \;=\; \lambda y : \alpha.\ [e]_{\bar{s}[y \mapsto \mathsf{Var}\ y]} \end{array}$$

$$\begin{array}{l} \forall y\ \bar{s}. \\ \quad \mathsf{FClosed}\ \mathsf{fv}\ \bar{s}\ \supset \\ \qquad [\mathsf{rec}\ y_\alpha\ e]_{\bar{s}} \;=\; \mathsf{rec}\ y_\alpha\ [e]_{\bar{s}[y \mapsto \mathsf{Var}\ y]} \end{array}$$

Proof. From the definition of substitution and properties of the choice function, it can be shown that renaming does not occur. The clauses in the definitions of substitution can then be simplified to produce the above results.

The theorems above indicate one remaining problem with the use of FClosed substitutions to simplify proofs. While the substitution that is applied to the abstractions is closed the substitution generated and applied to the body of the abstraction is not. This can lead to a problem in several proofs later in the section where there is an induction over the structure of an expression of type. For types, a general example is form:

$$\forall t\ \bar{s}.\ \mathsf{FClosed}\ \mathsf{ftv}\ \bar{s}\ \supset\ P([t]_{\bar{s}}) \tag{5.28}$$

In attempt to prove this by induction over t, the step case for type quantification will be

$$\forall \bar{s}.\ \mathsf{FClosed}\ \mathsf{ftv}\ \bar{s}\ \supset\ P([\forall x.t]_{\bar{s}}) \tag{5.29}$$

under the assumption

$$\forall t\ \bar{s}.\ \mathsf{FClosed}\ \mathsf{ftv}\ \bar{s}\ \supset\ P([t]_{\bar{s}})$$

This will typically be simplified using the definition of substitution and other properties to

$$P([t]_{\bar{s}[x \mapsto \mathsf{Var}\ x]}) \tag{5.30}$$

under the assumptions

$$\forall t\ \bar{s}.\ \mathsf{FClosed\ ftv}\ \bar{s}\ \supset\ P([t]_{\bar{s}})$$
$$\mathsf{FClosed\ ftv}\ \bar{s}$$

As $\bar{s}[x \mapsto \mathsf{Var}\ x]$ is not closed the induction hypothesis cannot be applied. In the remainder of this section we show that a substitution that is closed apart from identity mappings has the same effect as a closed substitution and so the induction hypothesis in the previous example can be applied. Substitutions which map variables to either the same variable or a closed term are described as follows:

Definition 5.38

$$\mathsf{Ty\_FClosed\_Id}\ \bar{s}\ =\ (\forall x.\ \mathsf{FDom}\ \bar{s}\ x \supset\ ((\mathsf{Closed\ ftv}\ (\bar{s}\ x))\ \vee\ ((\bar{s}\ x)\ =\ \mathsf{Var}\ x)))$$
$$\mathsf{Exp\_FClosed\_Id}\ \bar{s}\ =\ (\forall x.\ \mathsf{FDom}\ \bar{s}\ x \supset\ ((\mathsf{Closed\ fv}\ (\bar{s}\ x))\ \vee\ ((\bar{s}\ x)\ =\ \mathsf{var}\ x)))$$

In the rest of this section only the function for types, Ty_FClosed_Id, will be considered. The results all hold for the equivalent version for expressions. Some simple properties are:

Theorem 5.39

$$\forall \bar{s}.\ \mathsf{Ty\_FClosed\_Id}\ \bar{s}\ \supset\ (\forall x.\ \mathsf{Ty\_FClosed\_Id}\ (\bar{s}[x \mapsto \mathsf{Var}\ x]))$$
$$\forall \bar{s}\ \alpha.\ \mathsf{Ty\_FClosed\_Id}\ \bar{s}\ \wedge\ \mathsf{Closed\ ftv}\ \alpha\ \supset\ (\forall x.\ \mathsf{Ty\_FClosed\_Id}\ (\bar{s}[x \mapsto \alpha]))$$

Proof. Both results are proved by simplifying with the definitions above and simple properties of finite maps.

By replacing FClosed by Ty_FClosed_Id in the inductive proof above the inductive hypothesis will apply. While some theorems can be proved this way, the problem can be tackled more easily using two more results. The first states that any substitution satisfying Ty_FClosed_Id can be transformed into a closed substitution by removing the identity mappings.

Theorem 5.40

$$\forall \bar{s}.\ \mathsf{Ty\_FClosed\_Id}\ \bar{s}\ \supset\ \mathsf{FClosed\ ftv}\ (\mathsf{DRestrict}\ \bar{s}\ (\lambda x.\ \bar{s}\ x \neq \mathsf{Var}\ x))$$

Proof. By induction over the finite map $\bar{s}$ and simplifying with properies of FClosed, Ty_FClosed_Id and finite maps.

This closed substitution generated from a substitution satisfying Ty_FClosed_Id has the same effect as the original substitution.

Theorem 5.41

$$\forall \alpha\ \bar{s}.\ \mathsf{Ty\_FClosed\_Id}\ \bar{s}\ \supset\ [\alpha]_{\bar{s}}\ =\ [\alpha]_{\mathsf{DRestrict}\ \bar{s}\ (\lambda x.\ \bar{s}\ x\ \neq\ \mathsf{Var}\ x)}$$

Proof. By structural induction over the type α and simplifying with properies of Ty_FClosed_Id, FClosed and finite maps.

These results are sufficient to apply the inductive hypothesis in the example goal (5.30). This technique is used in the induction step of the proof of the following theorem.

Proofs about the semantics of the language will often proceed by rule induction [Win93] over the appropriate semantic relation. Some rules, including those for type and function application, generate a single substitution. In the inductive step it is often necessary to rearrange the order in which this single substitution and a simultaneous substitution of a finite map takes place. The key theorem is:

Theorem 5.42 *If f is some free variable function then*

$$\forall t\ \bar{s}\ t_1.\ \mathsf{FClosed}\ f\ \bar{s}\ \wedge\ \mathsf{Closed}\ f\ t_1\ \supset\ (\forall x.\ [t[t_1/x]]_{\bar{s}} = [t]_{\bar{s}[x \mapsto \mathsf{Var}\ x]}[t_1/x])$$

Proof. By structural induction over t.

While the version of this theorem for expressions is sufficient to prove the required results later in this thesis, the version for substitutions of types is not. In particular the condition that the type t_1 is closed will not hold. In the next section we prove a variant of this theorem without the condition that t_1 is closed and with a weaker conclusion.

5.5 Equality for expressions and types

The definitions of the types of SDT types and expressions in HOL gives rise to types for which the standard logical equality is not the equality that is required. In many presentations of lambda calculi and other such languages the syntax of binding constructs, such as type abstraction, is normally assumed to be equal up to alpha conversion. That is, two expressions or types will be equal if they differ only in the names of bound variables. For the types and expressions introduced in the last section this does not hold. The types

$$\forall x.\mathsf{Var}\ x$$

and

$$\forall y.\mathsf{Var}\ y$$

are not equal unless the strings x and y are equal.

To solve this problem new relations are introduced to define equivalence for both expressions and types. Very different approaches are taken for each. For types, a relation is introduced which captures precisely equivalence up to renaming of bound variable. For

$$\frac{}{\mathsf{Num} =_\alpha \mathsf{Num}}$$

$$\frac{}{\mathsf{Var}\ v =_\alpha \mathsf{Var}\ v}$$

$$\frac{t_1 =_\alpha t_3 \qquad t_2 =_\alpha t_4}{t_1 \rightarrow t_2 =_\alpha t_3 \rightarrow t_4}$$

$$\frac{t_1 =_\alpha t_2[\mathsf{Var}\ x/y]}{\forall x.t_1 =_\alpha \forall y.t_2} \quad \neg(\mathsf{ftv}\ (\forall y.t_2)\ x)$$

$$\frac{\begin{array}{l}\forall c.\ \mathsf{FDom}\ \overline{xs}\ c \supset \\ \quad \mathsf{all2}\ (=_\alpha)\ (\overline{xs}\ c)\ (\mathsf{Map}\ (\_[\mathsf{Var}\ x/y])(\overline{ys}\ c))\end{array}}{\mathsf{Data}\ x\ \overline{xs} =_\alpha \mathsf{Data}\ y\ \overline{ys}} \quad \begin{array}{l}\neg(\mathsf{ftv}\ (\mathsf{Data}\ y\ ys)\ x) \\ (\mathsf{FDom}\ \overline{xs}\ =\ \mathsf{FDom}\ \overline{ys})\end{array}$$

Figure 5.6: The defintion of $=_\alpha$

expressions, a new relation will be introduced later which will have equivalence up to renaming of bound variables as a property. This will be a co-inductively defined equality. The definition of this relation for expressions, and the proof of the correct properties, form a major part of this work and are given in chapter 7.

5.5.1 Alpha equivalence for types

The equivalence relation for types, $=_\alpha$, is defined as the inductive relation given in figure 5.6. The function all2 takes a binary relation and two lists and returns true if the lists are the same length and corresponding elements of the two lists are related by the binary relation. This definition was introduced in HOL using Harrison's inductive definitions package [Har95]. The definition has a side condition stating that the relation is monotonic. This is easily proved.

The definition also produces an induction theorem.

Theorem 5.43 (Rule induction for alpha equivalence)

$$
\begin{array}{l}
\forall R. \\
\quad R\ \mathsf{Num}\ \mathsf{Num}\ \wedge \\
\quad (\forall v.\ R\ (\mathsf{Var}\ v)(\mathsf{Var}\ v))\ \wedge \\
\quad (\forall t_1\ t_2\ t_3\ t_4. \\
\qquad R\ t_1\ t_3\ \wedge\ R\ t_2\ t_4\ \supset\ R\ (t_1 \rightarrow t_2)\ (t_3 \rightarrow t_4))\ \wedge \\
\quad (\forall x\ y\ t_1\ t_2. \\
\qquad R\ t_1\ (t_2[\mathsf{Var}\ x/y])\ \wedge\ \neg(\mathsf{ftv}\ (\forall y.t_2)x)\ \supset\ R\ (\forall x.t_1)\ (\forall y.t_2))\ \wedge \\
\quad (\forall x\ \bar{m}\ y\ \bar{n}. \\
\qquad (\forall c.\ \mathsf{FDom}\ \bar{m}\ c\ \supset \\
\qquad\quad \mathsf{all2}\ R\ (\bar{m}\ c)\ (\mathsf{Map}\ (\lambda z.\ z[\mathsf{Var}\ x/y])\ (\bar{n}\ c)))\ \wedge \\
\qquad \neg(\mathsf{ftv}(\mathsf{Data}\ y\ \bar{n})\ x)\ \wedge \\
\qquad (\mathsf{FDom}\ \bar{m}\ =\ \mathsf{FDom}\ \bar{n})\ \supset \\
\qquad\ R\ (\mathsf{Data}\ x\ \bar{m})\ (\mathsf{Data}\ y\ \bar{n})) \\
\qquad \supset \\
\quad (\forall a_1\ a_2.\ a_1 =_\alpha a_2\ \supset\ R\ a_1\ a_2)
\end{array}
$$

A more useful version of the induction theorem can be derived, and is referred to as strong rule induction [Gor95a]. For the rest of this thesis, where induction theorems are derived from a relation, only the strong rule induction version will be given.

Theorem 5.44 (Strong rule induction)

$$
\begin{array}{l}
\forall R. \\
\quad R\ \mathsf{Num}\ \mathsf{Num}\ \wedge \\
\quad (\forall v.\ R\ (\mathsf{Var}\ v)(\mathsf{Var}\ v))\ \wedge \\
\quad (\forall t_1\ t_2\ t_3\ t_4. \\
\quad\quad R\ t_1\ t_3\ \wedge\ R\ t_2\ t_4\ \wedge\ t_1 =_\alpha t_3\ \wedge\ t_2 =_\alpha t_4\ \supset\ R\ (t_1 \rightarrow t_2)\ (t_3 \rightarrow t_4))\ \wedge \\
\quad (\forall x\ y\ t_1\ t_2. \\
\quad\quad R\ t_1\ (t_2[\mathsf{Var}\ x/y])\ \wedge\ t_1 =_\alpha (t_2[\mathsf{Var}\ x/y])\ \wedge\ \neg(\mathsf{ftv}\ (\forall y.t_2)x)\ \supset\ R\ (\forall x.t_1)\ (\forall y.t_2))\ \wedge \\
\quad (\forall x\ \bar{m}\ y\ \bar{n}. \\
\quad\quad (\forall c.\ \mathsf{FDom}\ \bar{m}\ c\ \supset \\
\quad\quad\quad \mathsf{all2}\ R\ (\bar{m}\ c)\ (\mathsf{Map}\ (\lambda z.\ z[\mathsf{Var}\ x/y])\ (\bar{n}\ c)))\ \wedge \\
\quad\quad\quad \mathsf{all2}\ (=_\alpha)\ (\bar{m}\ c)\ (\mathsf{Map}\ (\lambda z.\ z[\mathsf{Var}\ x/y])\ (\bar{n}\ c)))\ \wedge \\
\quad\quad \neg(\mathsf{ftv}(\mathsf{Data}\ y\ \bar{n})\ x)\ \wedge \\
\quad\quad (\mathsf{FDom}\ \bar{m}\ =\ \mathsf{FDom}\ \bar{n})\ \supset \\
\quad\quad\ R\ (\mathsf{Data}\ x\ \bar{m})\ (\mathsf{Data}\ y\ \bar{n})) \\
\quad\quad \supset \\
\quad (\forall a_1\ a_2.\ a_1 =_\alpha a_2\ \supset\ R\ a_1\ a_2)
\end{array}
$$

Proof. Follows from theorem 5.43 by instantiating R in that theorem with

$$\lambda\alpha\ \beta.\alpha =_\alpha \beta\ \wedge\ R\ \alpha\ \beta$$

and simplifying.

The relation is an equivalence relation.

Theorem 5.45 *For any types* α, β, γ

$$\alpha =_\alpha \alpha$$

$$\alpha =_\alpha \beta\ \supset \beta =_\alpha \alpha$$

$$\alpha =_\alpha \beta\ \wedge\ \beta =_\alpha \gamma\ \supset \alpha =_\alpha \gamma$$

Proof. Reflexivity follows by a simple structural induction over the type α. Symmetry and transitivity follow by rule inductions with theorem 5.44 and some tedious but routine reasoning about free variables and substitutions.

Alpha equivalence relates two terms up to renaming of bound variables.

Theorem 5.46

$$\forall x\ y\ \alpha.\ \neg(\mathsf{ftv}\ (\forall x.\alpha)\ y)\ \supset\ (\forall x.\alpha) =_\alpha (\forall y.\alpha[\mathsf{Var}\ y/x])$$

Proof. Follows from the definiton of $=_\alpha$ and the reflexive and symetric properties.

We can now return to the variant of theorem 5.42 mentioned in the previous section.

Theorem 5.47

$$\forall t\ \bar{T}\ x\ t_1.\ \mathsf{FClosed\ ftv}\ \bar{T}\ \supset\ [t[t_1/x]]_{\bar{T}} =_\alpha [t]_{\bar{T}[x \mapsto \mathsf{Var}\ x]}[t_1/x]$$

Proof. By structural induction over t.

5.6 Static semantics

The static semantics are formalised by an inductively defined relation [Mel92]

$$\mathsf{Type} : (\mathit{string}, \mathit{ty})\mathit{fmap} \to \mathit{exp} \to \mathit{ty} \to \mathit{bool}$$

which takes a context, an expression and a type and returns true if the expression has the given type in the context. We will normally write the typing judgement

$$\mathsf{Type}\ \Gamma\ e\ \alpha$$

as

$$\Gamma \vdash e : \alpha$$

The Type relation is given in figure 5.7. The definition makes use of function, makefun, which takes a result type of a case expression, α, and a list of types representing the arguments of a constructor, $[\beta_1, \beta_2, \ldots, \beta_n]$, and returns the syntax of a function type which takes the same arguments as the constructor:

$$\beta_1 \to \beta_2 \to \ldots \to \beta_n \to \alpha$$

Definition 5.48

$$\begin{aligned} \mathsf{makefun}\ \alpha\ [\,] &= \alpha \\ \mathsf{makefun}\ \alpha\ (\mathsf{Cons}\ x\ xs) &= (x \to (\mathsf{makefun}\ \alpha\ xs)) \end{aligned}$$

Alpha equivalence for types is used in a number of places in the definition, but not everywhere it could be used. For example, the rule for function application could have been written as:

$$\frac{\Gamma \vdash e_1 : (\gamma \to \delta) \qquad \Gamma \vdash e_2 : \alpha \qquad \alpha =_\alpha \gamma \qquad \beta =_\alpha \delta}{\Gamma \vdash (e_1\ e_2) : \beta}$$

This is unnecessary since the reason for weakening equality to alpha equivalence is so that the following statement will be true.

$$\forall \Gamma\ e\ \alpha.\ \Gamma \vdash e : \alpha\ \supset\ (\forall \beta.\ \alpha =_\alpha \beta\ \supset\ \Gamma \vdash e : \beta)$$

$$\frac{}{\Gamma \vdash \mathsf{num}\ n : \mathsf{Num}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Num} \qquad \Gamma \vdash e_2 : \mathsf{Num}}{\Gamma \vdash \mathsf{nop}\ op\ e_1\ e_2 : \mathsf{Num}}$$

$$\frac{\beta =_\alpha \gamma}{\Gamma[x \mapsto \beta] \vdash \mathsf{var}\ x : \gamma} \qquad \frac{C \vdash e_1 : \mathsf{Num} \qquad C \vdash e_2 : \mathsf{Num}}{\Gamma \vdash \mathsf{bop}\ op\ e_1\ e_2 : \mathsf{Bool}}$$

$$\frac{\Gamma[x \mapsto \beta] \vdash e : \delta \qquad \beta =_\alpha \gamma}{\Gamma \vdash (\lambda x : \beta.\ e) : \gamma \rightarrow \delta}$$

$$\frac{\Gamma \vdash e_1 : (\beta \rightarrow \gamma) \qquad \Gamma \vdash e_2 : \beta}{\Gamma \vdash (e_1\ e_2) : \gamma}$$

$$\frac{\Gamma \vdash e : \beta \quad x \text{ not free in } \Gamma \qquad \gamma =_\alpha \forall x.\beta}{\Gamma \vdash \Lambda x.\ e : \gamma}$$

$$\frac{\Gamma \vdash e : \forall x.\beta \qquad \delta =_\alpha \beta[\gamma/x]}{\Gamma \vdash e_\gamma : \delta}$$

$$\frac{\Gamma[x \mapsto \beta] \vdash e : \beta \qquad \gamma =_\alpha \beta}{\Gamma \vdash \mathsf{rec}\ x_\beta\ e : \gamma}$$

$$\frac{\mathsf{all2}\ (\mathsf{Type}\ \Gamma)\ xs\ (\mathsf{Map}\ (\lambda y.\ y[\mathsf{Data}\ x\ \overline{m}/x])\ ts)}{\Gamma \vdash \mathsf{con}\ c_{\mathsf{Data}\ x\ \overline{m}}\ xs :\ \beta} \quad \begin{array}{l} \mathsf{FDom}\ \overline{m}\ c \\ ts\ =\ \overline{m}\ c) \\ \beta =_\alpha (\mathsf{Data}\ x\ \overline{m}) \end{array}$$

$$\frac{\begin{array}{l} \Gamma \vdash e : \mathsf{Data}\ x\ \overline{m} \\ \forall s.\mathsf{FDom}\ \overline{c}\ s\ \supset\ \mathsf{FDom}\ \overline{m}\ s\ \wedge \\ \qquad \Gamma \vdash (\overline{c}\ s) : (\mathsf{makefun}\ \beta\ (\overline{m}\ s))[\mathsf{data}\ x\ \overline{m}/x] \end{array}}{\Gamma \vdash \mathsf{case}\ e\ \overline{c} : \beta}$$

Figure 5.7: The typing rules

In order to prove this, it is necessary to weaken the equality of only the types which occur, or have components which occur, on both sides of the ':' in the type assignment. Thus the weakening is unnecessary for the rule for function application, but necessary for type application where β and γ occur in different places in the conclusion $\Gamma \vdash e_\beta : \gamma$. Similarly, the equivalence, $\beta =_\alpha \forall x.\gamma$ is necessary in the rule for type abstraction since x and β occur separately in the conclusion $\Gamma \vdash \Lambda x.\ e : \beta$.

The typing relation is again introduced using the inductive definitions package and the rules are proved to be monotonic. This gives rise to an induction theorem. A stronger version of this theorem, strong rule induction, can also be proved. The statement of the theorem is given in figure 5.8.

Typically we consider only well typed programs, so it is useful to introduce a second relation, Prog, which holds of a type α and an expression e only if e has type α in the empty context.

Definition 5.49

$$\mathsf{Prog}\ e\ \alpha \ = \ \mathsf{Type\ FEmpty}\ e\ \alpha$$

For clarity, Prog e α will often be written $e : \alpha$.

A major property of expressions that can be typed in the empty context is that they cannot contain free expression variables.

Theorem 5.50

$$\forall e\ \alpha.\ e : \alpha \ \supset\ \mathsf{Closed\ fv}\ e$$

The significance of this theorem, in conjunction with theorem 5.37, is that when a closed expression is applied to another closed expression, only closed substitutions are formed and so no renaming will occur.

5.7 Properties of the typing relation

There are some important properties of the relationship between typing judgements and alpha conversion. The two crucial theorems are given below.

Theorem 5.51 $\forall\, \Gamma\ e\ \alpha.\ \Gamma \vdash e : \alpha\ \supset\ (\forall \beta.\ \alpha =_\alpha \beta\ \supset\ \Gamma \vdash e : \beta)$

Proof. Follows by a straightforward rule induction with theorem 5.8.

The next theorem of this section expresses the fact a term has a unique type, when the types are considered equal up to renaming of the bound variables.

$$
\begin{array}{l}
\forall T.\\
\quad (\forall n\ \Gamma.\ T\ \Gamma\ (\mathsf{num}\ n)\ \mathsf{Num})\ \wedge\\
\quad (\forall n\ e_2\ e_1\ \Gamma.\\
\qquad T\ \Gamma\ e_1\ \mathsf{Num}\ \wedge\ T\ \Gamma\ e_2\ \mathsf{Num}\ \wedge\ \Gamma \vdash e_1 : \mathsf{Num}\ \wedge\ \Gamma \vdash e_2 : \mathsf{Num}\ \supset\\
\qquad\quad T\ \Gamma\ (\mathsf{nop}\ n\ e_1\ e_2)\ \mathsf{Num})\ \wedge\\
\quad (\forall n\ e_2\ e_1\ \Gamma.\\
\qquad T\ \Gamma\ e_1\ \mathsf{Num}\ \wedge\ T\ \Gamma\ e_2\ \mathsf{Num}\ \wedge\ \Gamma \vdash e_1 : \mathsf{Num}\ \wedge\ \Gamma \vdash e_2 : \mathsf{Num}\ \supset\\
\qquad\quad T\ \Gamma\ (\mathsf{bop}\ n\ e_1\ e_2)\ \mathsf{Bool})\ \wedge\\
\quad (\forall v\ \Gamma\ \alpha\ \beta.\ \alpha =_\alpha \beta\ \supset\ T\ \Gamma[v \mapsto t]\ (\mathsf{var}\ v)\ \beta)\ \wedge\\
\quad (\forall \gamma\ \alpha\ e\ \beta\ y\ \Gamma.\\
\qquad T\ \Gamma[y \mapsto \beta]\ e\ \alpha\ \wedge\ \Gamma[y \mapsto \beta] \vdash : e\ \alpha\ \wedge\ \beta =_\alpha \gamma\ \supset\\
\qquad\quad T\ \Gamma\ (\lambda y : \beta.\ e)\ (\gamma \rightarrow \alpha)\ \wedge\\
\quad (\forall e_2\ \alpha\ \beta\ e_1 \Gamma.\\
\qquad T\ \Gamma\ e_1\ (\beta \rightarrow \alpha)\ \wedge\ T\ \Gamma\ e_2\ \beta\ \wedge\ \Gamma \vdash e_1 : (\beta \rightarrow \alpha)\ \wedge\ \Gamma \vdash e_2 : \beta\ \supset\\
\qquad\quad T\ \Gamma\ (e_1\ e_2)\ \alpha)\ \wedge\\
\quad (\forall \beta\ x\ \alpha\ e\ \Gamma.\\
\qquad T\,\Gamma\,e\,\alpha\ \wedge\ \Gamma \vdash e : \alpha\ \wedge\\
\qquad (\forall y.\ \neg(\mathsf{FDom}\ \Gamma\ y\ \wedge\ \mathsf{ftv}\ (\Gamma\ y)\ x))\ \wedge\ \beta =_\alpha (\forall x.\alpha)\ \supset\\
\qquad\quad T\ \Gamma\ (\Lambda x.\ e)\ \beta)\ \wedge\\
\quad (\forall \beta\ \alpha\ \gamma\ x\ e\ \Gamma.\\
\qquad T\ \Gamma\ e\ (\forall x.\gamma)\ \wedge\ \Gamma \vdash e : \forall x.\gamma)\ \wedge\ \alpha =_\alpha (\gamma[\beta/x])\ \supset\\
\qquad\quad T\ \Gamma\ (e_\beta)\ \alpha)\ \wedge\\
\quad (\forall \beta\ e\ \alpha\ y\ \Gamma.\ T\ (\Gamma[y \mapsto \beta])\ e\ \alpha\ \wedge\ \Gamma[y \mapsto \beta] \vdash e : \alpha\ \wedge\ \beta =_\alpha \alpha\ \supset\\
\qquad\quad T\ \Gamma\ (\mathsf{rec}\ y_\alpha\ e)\ \beta)\ \wedge\\
\quad (\forall \alpha\ c\ ts\ \bar{m}\ x\ xs\ \Gamma.\\
\qquad \mathsf{all2}\ (T\ \Gamma)\ xs\ (\mathsf{Map}\ (\_[\mathsf{Data}\ x\ \bar{m}/x])\ ts)\ \wedge\\
\qquad \mathsf{all2}\ (\mathsf{Type}\ \Gamma)\ xs\ (\mathsf{Map}\ (\_[\mathsf{Data}\ x\ \bar{m}/x])\ ts)\ \wedge\\
\qquad \mathsf{FDom}\ \bar{m}\ c\ \wedge\ (ts\ =\ \bar{m}\ c)\ \wedge\ \alpha =_\alpha \mathsf{Data}\ x\ \bar{m}\ \supset\\
\qquad\quad T\ \Gamma\ (\mathsf{con}\ c_{(\mathsf{Data}\ x\ \bar{m})}\ xs)\ \alpha)\ \wedge\\
\quad (\forall \alpha\ \bar{n}\ \bar{m}\ x\ e\ \Gamma.\\
\qquad T\ \Gamma\ e\ (\mathsf{Data}\ x\ \bar{m})\ \wedge\ \Gamma \vdash e : \mathsf{Data}\ x\ \bar{m}\ \wedge\\
\qquad (\forall s.\\
\qquad\quad \mathsf{FDom}\ \bar{n}\ s\ \supset\ \mathsf{FDom}\ \bar{m}\ s\ \wedge\\
\qquad\quad T\ \Gamma\ (\bar{n}\ s)\ (\mathsf{makefun}\ \alpha\ (\mathsf{Map}\ (\_\ [\mathsf{Data}\ x\ \bar{m}/x])\ (\bar{m}\ s))\ \supset\\
\qquad\quad T\ \Gamma\ (\mathsf{case}\ e\ \bar{n})\ \alpha)\\
\qquad \supset\\
\quad (\forall \Gamma\ e\ \alpha.\ \Gamma \vdash e : \alpha\ \supset\ T\ \Gamma\ e\ \alpha)
\end{array}
$$

Figure 5.8: Strong rule induction for Types

Theorem 5.52 *Uniqueness of types up to alpha equivalence*

$$\forall \Gamma\ e\ \alpha.\ \Gamma \vdash e : \alpha\ \supset\ (\forall \beta.\ \Gamma \vdash e : \beta\ \supset\ \alpha\ =_\alpha\ \beta)$$

Proof. Follows by rule induction over the typing judgement $\Gamma \vdash e : \alpha$.

It is useful to know that a term has only one type. Much of theory that follows depends on the analysis of the types of terms and this analysis is easier if there is only one type to consider for each term.

This chapter has introduced closed substitutions and expressions that have a type in a context and in the empty context. The following results express the relationship between the type of expressions with free variables and the type of expressions with closed terms substituted for the free variables.

If an expression e is typeable in a context Γ then a substitution that maps variables to expressions with the type corresponding to that variable in the context will be known as a *closure*. This is defined as:

Definition 5.53

$$\mathsf{Closure}\ \Gamma\ \bar{s}\ =\ (\mathsf{FDom}\ \Gamma\ =\ \mathsf{FDom}\ \bar{s})\ \wedge\ (\forall x.\ \mathsf{FDom}\ \Gamma\ x\ \supset\ (\bar{s}\ x) : (\Gamma\ x))$$

One useful property of closures is that the substitution is itself closed.

Theorem 5.54

$$\forall \Gamma\ \bar{s}.\ \mathsf{Closure}\ \Gamma\ \bar{s}\ \supset\ \mathsf{FClosed}\ \mathsf{fv}\ \bar{s}$$

Proof. From definition of Closure we know that every expression in the range of the substitution is well-typed in the empty context and hence, from theorem 5.50, contains no free expression variables. The result follows from the definition of FClosed.

An important property of closures is that the substitution for any term typeable in the empty context is itself empty.

Theorem 5.55

$$\forall \bar{s}.\ \mathsf{Closure}\ \mathsf{FEmpty}\ \bar{s}\ =\ (\bar{s}\ =\ \mathsf{FEmpty})$$

Proof. Follows immediately from the definition of Closure and properties of finite maps.

The key theorem, which will be used extensively in Chapter 7, relates the type of an expression in a context to the type of the closed term produced by a closure for that context.

Theorem 5.56

$$\forall \Gamma\ e\ \alpha.\ \Gamma \vdash e : \alpha\ \supset\ (\forall \bar{s}.\ \mathsf{Closure}\ \Gamma\ \bar{s}\ \supset\ [e]_{\bar{s}} : \alpha)$$

Proof. By rule induction over the typing judgement and the properties of substitution, typing and closures.

A variant of this theorem can be proved that takes account of type substitutions as well. This theorem will be used in the theory for parametricity in Chapter 9. This extends theorem 5.56 by relating the type of an expression in a context to the type of that expression under both expression and typing substitutions.

Theorem 5.57

$$\begin{array}{l}
\forall\ \Gamma\ e\ \alpha. \\
\quad \Gamma \vdash e : \alpha\ \supset \\
\quad\quad (\ \forall \bar{s}\ \bar{T}.\ \mathsf{FClosed\ fv}\ \bar{s}\ \wedge\ \mathsf{FClosed\ ftve}\ \bar{s}\ \wedge\ \mathsf{FClosed\ ftv}\ \bar{T}\ \wedge \\
\quad\quad\ (\forall x.\ \mathsf{FDom}\ \Gamma\ x\ \supset\ (\mathsf{FDom}\ \bar{s}\ x\ \wedge\ \bar{s}\ x : [\Gamma x]_{\bar{T}}))\ \wedge \\
\quad\quad\ (\forall x.\ \mathsf{ftve}\ e\ x\ \supset\ \mathsf{FDom}\ \bar{T}\ x)\ \supset \\
\quad\quad\quad [[e]_{\bar{s}}]_{\bar{T}}\ :\ [\alpha]_{\bar{T}}\ )
\end{array}$$

Proof. By strong rule induction on the typing judgement $\Gamma \vdash e : \alpha$ and tedious reasoning about the substitutions.

5.8 Dynamic semantics

The dynamic semantics introduces a family of relations based on one relation, $\longrightarrow$. From this, many step reduction, $\longrightarrow^*$, and reduction to normal form or evaluation, $\Downarrow$, can be defined.

The reduction relation is defined inductively in the same way as the static semantics. The rules are given in figures 5.9 and 5.10. The choice of rules for the semantics has been discussed previously. The definition gives rise to the usual rule induction and strong rule induction theorems. In what follows, we consider the semantics of numbers separately to the semantics of the other syntactic constructs. Figure 5.10 gives the semantics for numbers and figure 5.9 gives the semantics for the other constructs.

5.8.1 The dynamic semantics of numbers

The dynamic semantics for numbers is given in full in figure 5.10. Here only the numbers and the binary operations are embedded, although more could easily be added by extending the syntax introduced in figure 5.2. There is an important difference between

$$\frac{}{(\lambda y : \alpha.\, e_1)\, e_2 \longrightarrow e_1[e_2/y]}$$

$$\frac{e_1 \longrightarrow e_2}{e_1\, e_3 \longrightarrow e_2\, e_3}$$

$$\frac{}{\mathsf{rec}\; y_\alpha\; e \longrightarrow e[\mathsf{rec}\; y_\alpha\; e/y]}$$

$$\frac{}{(\Lambda y.\, e)_\alpha \longrightarrow e[\alpha/y]}$$

$$\frac{e_1 \longrightarrow e_2}{e_{1\alpha} \longrightarrow e_{2\alpha}}$$

$$\frac{}{\mathsf{case}\;(\mathsf{con}\; x_\alpha\; xs)\; \bar{m} \longrightarrow \mathsf{applyfun}(\bar{m}\; x)\; xs}$$

$$\frac{e_1 \longrightarrow e_2}{\mathsf{case}\; e_1\; \bar{m} \longrightarrow \mathsf{case}\; e_2\; \bar{m}}$$

Figure 5.9: The dynamic semantics

the type of numbers in SDT and the HOL logic. In the logic all expressions with the type of numbers, including arbitrarily complex expressions, denote some unique number. In SDT an expression of number type may be non-terminating or contain a partial function which does not return a result. When moving between the two representation of numbers it is important that these differences are handled correctly. In this section a specific example of the behaviour of the binary relation "less than or equal" is considered.

Suppose e_1,e_2 and e_3 are expressions with type Num, n_1 and n_2 are numbers in the HOL logic, and $\leq$ is the "less-than-or-equal" function in the logic. The rules for this binary relation on numbers are:

$$\frac{e_1 \longrightarrow e_2}{\mathsf{bop} \leq e_1\; e_3 \longrightarrow \mathsf{bop} \leq e_2\; e_3}$$

$$\frac{e_2 \longrightarrow e_3}{\mathsf{bop} \leq e_1\; e_2 \longrightarrow \mathsf{bop} \leq e_1\; e_3}$$

$$\frac{}{\mathsf{bop} \leq (\mathsf{num}\; n_1)\; (\mathsf{num}\; n_2) \longrightarrow \mathsf{num}\; (n_1 \;\leq\; n_2)}$$

If $\leq$ is applied to numbers which reduce to a value by repeated application of the first two rules, then the third rule will apply and the application of $\leq$ can be pushed down into the HOL numbers and the result lifted. But if one of the arguments does not reduce to a value, then one of the first two rules can be applied indefinitely, or the argument contains a partial function and at some point no rule applies. In either case the third rule will not apply and a result of the form $\mathsf{num}\; n$ will never be returned. This means that, unlike the type of numbers in HOL, the number expressions in SDT may not represent any numeric literal.

$$\frac{}{\mathsf{nop}\ f\ (\mathsf{num}\ n_1)\ (\mathsf{num}\ n_2) \longrightarrow \mathsf{num}\ (f\ n_1\ n_2)}$$

$$\frac{e_1 \longrightarrow e_2}{\mathsf{nop}\ f\ e_1\ e_3 \longrightarrow \mathsf{nop}\ f\ e_2\ e_3}$$

$$\frac{e_1 \longrightarrow e_2}{\mathsf{nop}\ f\ \mathsf{num}\ n\ e_1 \longrightarrow \mathsf{nop}\ f\ \mathsf{num}\ n\ e_2}$$

$$\frac{}{\mathsf{bop}\ f\ (\mathsf{num}\ n_1)\ (\mathsf{num}\ n_2) \longrightarrow \mathsf{num}\ (f\ n_1\ n_2)}$$

$$\frac{e_1 \longrightarrow e_2}{\mathsf{bop}\ f\ e_1\ e_3 \longrightarrow \mathsf{bop}\ f\ e_2\ e_3}$$

$$\frac{e_1 \longrightarrow e_2}{\mathsf{bop}\ f\ \mathsf{num}\ n\ e_1 \longrightarrow \mathsf{bop}\ f\ \mathsf{num}\ n\ e_2}$$

Figure 5.10: The dynamic semantics of numbers

5.8.2 Derived relations

The reduction relation is a small step evaluation relation, where a term can be reduced repeatedly until a value is returned. It is useful to define a second relation, the relexive transitive closure of reduction, which can express many reduction steps in one relation. The relation $\longrightarrow^*$ is an inductively defined many step reduction relation.

Definition 5.58

$$\frac{}{e \longrightarrow^* e}$$

$$\frac{e_1 \longrightarrow^* e_2 \qquad e_2 \longrightarrow e_3}{e_1 \longrightarrow^* e_3}$$

If an expression has been reduced to a value (function, type abstraction, number or constructor) then it cannot be reduced any further. These normal forms of the reduction function can be defined by

Definition 5.59

$$\begin{aligned}\mathsf{NF}\ c = \ & \exists x\ \alpha\ as.\ c = \mathsf{con}\ x_\alpha\ as\ \vee \\ & \exists n.\ c = \mathsf{num}\ n\ \vee \\ & \exists x\ \alpha\ e.\ c = \lambda x : \alpha.\ e\ \vee \\ & \exists x\ e.\ c = \Lambda x.\ e\end{aligned}$$

Reduction to normal form, $\Downarrow$, can then be defined as

Definition 5.60

$$e_1 \Downarrow e_2 = (e_1 \longrightarrow^* e_2 \wedge \mathsf{NF}\ e_2)$$

Rules can be proved for this relation which are identical to the rules that would have been defined for a large step reduction relation, as discussed in chapter 3. One example is the rule for function application:

$$\frac{e_1 \Downarrow (\lambda y : \alpha.\ e) \qquad (e[e_2/y]) \Downarrow c}{(e_1\ e_2) \Downarrow c}$$

Later, it will be useful to state that an expression e evaluates to some normal form without needing to worry about the form of that result.

$$\exists c.\ e \Downarrow c$$

This statement will be abbreviated by $e \Downarrow$.

5.9 Properties of reduction

The reduction relation has been written in such a way as to specify the reduction order. At no point can more that one rule apply to any expression. It follows from this that the relation is deterministic.

Theorem 5.61

$$\forall e_1\ e_2.\ e_1 \longrightarrow e_2 \supset (\forall e_3.\ e_1 \longrightarrow e_3 \supset (e_2 = e_3))$$

Proof. By a rule induction on the reduction $e_1 \longrightarrow e_2$.

The reduction relation is untyped and so the rules may also be applied to untyped expressions. Whenever the rules are applied to well-typed expressions it is important that the type is preserved.

Theorem 5.62 (Subject Reduction)

$$\vdash \forall e_1\ e_2.\ e_1 \longrightarrow e_2 \supset (\forall \alpha.\ e_1 : \alpha \supset e_2 : \alpha)$$

Proof. By a rule induction on the reduction $e_1 \longrightarrow e_2$.

5.10 Related work

In this chapter the syntax of the SDT language was represented by types in the logic with a structural equality between terms of the type. For SDT types, a new equivalence was defined to identify types up to a renaming of bound variables. An alternative approach, which has been applied to simple languages with binding constructs, would be to define the types in a different way so that the standard equality has this property [GM96]. While this would simplify the problems of alpha equivalence and single substitutions it would not remove the need to reason about simultaneous substitutions later in this thesis and so would not allow any of the work on substitutions to be omitted.

The work in this chapter is similar in some ways to the deep embeddings of functional languages discussed in chapter 3. These embeddings also introduced new types for the syntax and new relations for the semantics. The language that has been most commonly embedded in theorem provers is the semantics of Standard ML, due to the availability of the a complete operational semantics for the language [MTH90, MT91, MTHM97]. Donald Syme embedded core ML in HOL [Sym93, Sym92]. The HOL-ML project developed an embedding of the core language [VG93, Van94] and looked at the module language and variants of the module language with the aim of comparing features [MG94].

These embeddings of ML differ from the embedding here. With all the ML embeddings, the aim was to reason about the language semantics, not to produce a reasoning system for programs in the language. The style in which the semantics is formalised is very different. The meaning of a function is defined relative to a complex environment containing state information. Function application is formalised not by substitution but by adding function closures to the environment. This avoids the complex reasoning about substitutions described here, but similar issues would arise in an attempt to compare different programs for equality. The environments would have to be compared which would involve renaming of variables in the environments.

Chapter 6

Automation of low level inference

The work described in the previous chapter establishes the theoretical foundations for reasoning about the syntax and semantics of SDT programs. This chapter deals with the practical aspects of reasoning about the typing and reduction of programs.

Results such as a proof that an expression evaluates to a specific value can be obtained by working out which rules to apply by hand or by conducting a long interactive goal-directed proof. The number of rules to be applied may be very large and applying all the rules by hand may not be practical. As type judgements may be in the side conditions for the application of many other theorems it is important that their proofs are as automatic as possible. Similar problems occur when proving many results about programs. There are often a large number of obvious or trivial proof steps to be carried out.

Many of the small steps in a proof will arise from calculating the type or value of a program and so tools have been developed to automate these proofs. The typing and reduction relations can be thought of as specifications of how to type or reduce expressions on an abstract machine. It is possible to write, in ML, a program that implements this specification.

For the reduction and typing relations, these programs and the relations will both be deterministic. The way in which the programs calculate the types or values will correspond exactly to the way in which the rules need to be applied to prove the same result. Because of this the programs can be used to return information about the rules applied and this can be used to generate the proof. This method provides a structured proof, following the definition precisely, rather than trying to solve a search problem or attempting the exhaustive application of rewrite rules.

Although HOL and the tools are both implemented in ML, they are treated as separate systems with an interface between them. A translator converts from the HOL types for the syntax of expressions and types to the ML types used in the tools. This allows the tools to be developed and tested separately from the rest of the system. The ML types contain additional type constructors. For example we represent HOL variables and HOL constants with separate type constructors as discussed in the next section.

One advantage of developing an external system to find the proof and then using a theorem prover such as HOL to check the proof and manipulate the results is that checking a proof may be much more efficient than searching for a proof. The resulting system is still guaranteed sound; if the interpreter is not correct then an incorrect proof will fail when checked. There are also applications, such as finding the witness for solving existential goals, for which discovering the type of the term may be enough without completing the proof.

The reducer and type checker also provide useful tools for experimenting with programs before deciding which direction a proof should take. In this case the reducer and type checker can be run without performing any proof at all and invoked only to provide a proof when the overall strategy has been established.

For the type checker, the results returned are always of the form

$$\Gamma \vdash e : \alpha$$

There are two variations of the reducer. The first, simple version returns a result of the form

$$e_1 \longrightarrow e_2$$

while the more general version returns results of the form

$$e_1 \longrightarrow^* e_2$$

For any expression e_1 there are many such results depending on how many reductions are performed. Parameters supplied to the reducer determine how many reduction steps are applied. For any specific parameter values, the result will be deterministic.

6.1 Translation of syntax

There is a translation from the HOL types for the syntax of expression to ML datatypes representing the syntax. This is not a one-to-one translation, as the ML types contain additional constructors. For example, we represent HOL variables and HOL constants with separate constructors and add constructors representing the substitution functions. This allows easier manipulation of these aspects of a term.

The ML types for the syntax could be extended in the future to handle heuristics such as rippling [BSvH+93] that can be used to control the application of rewrite rules in an equational reasoning system to rewrite the step case of an induction to the induction hypothesis.

6.2 Definition of new types and functions

In general, most programs entered into the system will not consist only of primitive syntax. Most will consist mainly of previously-defined functions and types. This section discusses how the tools handle the definition of these constants and how this information will be made available to other tools.

6.2.1 Datatype definition

The definition of lists in terms of the primitive syntax of the language was given in section 4.8. Some care was needed to ensure the definition behaved correctly with respect to the renaming of type variables. The differences between the naive definition and the correct one are predictable and the correct definition can be automatically deduced from the naive one. In addition to defining the type it is also necessary to define the logical constants representing the type and its constructors. The ML function `define_datatype` takes a specification of a type and produces a definition of the type and the logical constants and proves and stores some useful theorems about them. For lists `define_datatype` is called as follows:

```
define_datatype
    {Name="List",
      Def= Data ("List",[("Nil",[]),("Cons",[Var "a", Var "List"])])}
```

This function call defines a new constant in the logic, List, and two constants, Nil and Cons. These constants have the following definitions:

$$\mathsf{list}\ \alpha \ = \ \left(\begin{array}{l} \mathsf{Data}\ \mathit{list}\ [\mathit{nil} \mapsto [], \\ \qquad\qquad \mathit{cons} \mapsto [\mathsf{Var}\ a, \mathsf{Var}\ \mathit{list}]] \end{array} \right) [\alpha/a]$$

$$\begin{aligned} \mathsf{nil}_\alpha &= \mathsf{con}\ \mathit{nil}_{\mathsf{list}\ \alpha}\ [] \\ \mathsf{cons}_\alpha\ h\ t &= \mathsf{con}\ \mathit{cons}_{\mathsf{list}\ \alpha}\ [h\ , t] \end{aligned}$$

The typing rules for the constructors will normally be defined automatically using the tools presented later in this section. For the constructors for lists the following theorems will be proved:

$$C \vdash \mathsf{Nil}_\alpha : \mathsf{List}\ \alpha$$

```
datatype ty = HOLvar_ty of string
            | HOLconst_ty of string * ty list
            | Var of string
            | Num
            | Data of string * (string*ty list) list
            | Fun of  ty * ty
            | All of string * ty
            | HOLty of term
            | ttsub of ty * string * ty


and exp = HOLvar_exp of string
            | HOLconst_exp of string * arg list
            | num of term
            | var of string
            | con of string * ty * exp list
            | lambda of string * ty * exp
            | app of exp * exp
            | Lambda of string * exp
            | App of exp * ty
            | Rec of string * ty *  exp
            | Case of exp  * (string * exp) list
            | eesub of exp * string * exp
            | tesub of exp * string * ty
            | HOLexp of term
            | nnnop of term * exp * exp
            | nnbop of term * exp * exp

and arg = Ty of ty | Exp of exp
```

Figure 6.1: The ML types for the abstract syntax of expressions

$C \vdash h : \alpha \ \wedge \ C \vdash t : \mathsf{List}\,\alpha \ \supset \ C \vdash \mathsf{Cons}_\alpha\, \mathsf{h}\ \mathsf{t} : \mathsf{List}\,\alpha$

To make these definitions and typing rules available to the other tools in a uniform and efficient way they are stored in a global state. The ML records that are stored for each constructor are:

```
type Tyconinfo = {Name:string,      (*The name of the constuctor*)
                  Def:thm,          (*The theorem defining the constant*)
                 }


type Coninfo = {Name:string,      (*The name of the constuctor*)
                Def:thm,            (*The theorem defining the constant*)
                Ty:thm option,      (*The theorem giving the type*)
               }
```

The theorem for the type of the constructor is an option type that can return either the value `NONE` indicating there is no type stored or return `SOME t` where `t` is the theorem required. This is to allow the function **define_datatype** to fail gracefully if, for any reason, it cannot prove the typing rule for the constructor. For constructors the automatic tools should always prove the type theorem but this is adopted as a design decision for all stored theorems. Future chapters will extend the range of theorems that will be stored and some of these may not be able to be proved automatically. A set of functions is provided which, given the name of a constant, look up that constant in the state and return the corresponding record.

6.2.2 Expression definition

The tool for expression definition takes the specification of an expression and defines the new logical constant. It also produces theorems about evaluation and typing. The function map used can be introduced in this way. The definition of map is given in figure 6.2.

The theorems produced by this are the theorem storing the definition of map and the evaluation and typing theorems in figure 6.3. The evaluation theorem is just one unwinding of the recursive function. The theorems produced by the definition are stored in the system's state and can be accessed in a similar way to the information about type definitions.

The type of the record storing this information is:

```
type Expinfo = {Def:thm,
                Eval:thm option,
```

$$
\begin{array}{l}
\mathsf{map} \;=\; \mathsf{rec}\; \mathit{mapf} : (\forall\alpha\;\beta.\; (\alpha \to \beta) \to \mathsf{List}\;\alpha \to \mathsf{List}\;\beta) \\
\qquad\qquad \Lambda\alpha.\; \Lambda\beta. \\
\qquad\qquad \lambda\, f : \alpha \to \beta.\; \lambda\, x : \mathsf{List}\;\alpha. \\
\qquad\qquad\quad \mathsf{case}\; x \;\mathsf{of} \\
\qquad\qquad\qquad \mathsf{nil} \mapsto \mathsf{nil}_\beta \;\mid \\
\qquad\qquad\qquad \mathsf{cons} \mapsto (\lambda\, hd : \alpha.\; \lambda\, tl\; : \mathsf{List}\;\alpha. \\
\qquad\qquad\qquad\qquad (\mathsf{cons}_\beta\; (f\; hd)\; (\mathit{mapf}_{\alpha\;\beta}\; f\; tl)))
\end{array}
$$

Figure 6.2: The definition of map

$$
\begin{array}{l}
\vdash \forall\alpha\;\beta.\; \mathsf{map}_{\alpha\;\beta} \;\longrightarrow^{*}\; \lambda\, f : \alpha \to \beta.\; \lambda\, x : \mathsf{List}\;\alpha. \\
\qquad\qquad\qquad\qquad \mathsf{case}\; x \;\mathsf{of} \\
\qquad\qquad\qquad\qquad\quad \mathsf{nil} \mapsto \mathsf{nil}_\beta \;\mid \\
\qquad\qquad\qquad\qquad\quad \mathsf{cons} \mapsto (\lambda\, hd : \alpha. \lambda\, tl\; : \mathsf{List}\;\alpha. \\
\qquad\qquad\qquad\qquad\qquad (\mathsf{cons}_\beta\; (f\; hd)\; (\mathsf{map}_{\alpha\;\beta}\; f\; tl))) \\
\\
\vdash \forall\alpha\;\beta.\; \mathsf{map}_{\alpha\;\beta} \;:\; ((\alpha \to \beta) \to \mathsf{List}\;\alpha \to \mathsf{List}\;\beta)
\end{array}
$$

Figure 6.3: The reduction and typing theorems for map

```
Name:string,
Ty:thm option}
```

6.3 Automating type inference and reduction

The tools for type inference and reduction are similar and this section concentrates on a description of the type inference tools. As an example typing the application of the identity function to a number is discussed. The expression $(\Lambda\alpha.\;\lambda x : \alpha.\; x)_{\mathsf{Num}}\; 1$ has type Num. This can be proved by the following sequence of inferences.

$$
\dfrac{\dfrac{\dfrac{\dfrac{[x \mapsto \alpha] \vdash x : \alpha}{\vdash \lambda x : \alpha.\; x : \alpha \to \alpha}}{\vdash \Lambda\alpha.\; \lambda x : \alpha.\; x : \forall\alpha.\alpha \to \alpha}}{\vdash (\Lambda\alpha.\; \lambda x : \alpha.\; x)_{\mathsf{Num}} : \mathsf{Num} \to \mathsf{Num}} \qquad \vdash 1 : \mathsf{Num}}{\vdash (\Lambda\alpha.\; \lambda x : \alpha.\; x)_{\mathsf{Num}}\; 1 : \mathsf{Num}}
$$

Type inference is done by recursively decomposing the expression, calculating the type of the expressions at the leaves of the tree and using these to derive the type of the initial expression. The types of constants are found by looking up the type in the global state.

The algorithm for doing this is much simpler than type inference in languages such as Haskel and ML. This is because SDT is explicitly typed and no unification is necessary to determine the type of expressions.

While a conventional implementation would simply return the type of the expression, the tool devised here returns a derivation tree that can be used to reconstruct the proof in HOL. The type of the result of type inference is defined in terms of

```
datatype 'a Result = Node of 'a *  'a Result list;
```

This type is used for the result of both the typing and reduction tools. For type inference the result type is

```
((string * ty) list * exp * ty) Result
```

The three elements in the type correspond to the context, expression and type in the typing judgement. This encodes the derivation tree corresponding to the inferences shown in the diagram above. The tree can then be traversed from the nodes to the root, building a forward proof about the type of a term using the rules in figure 5.7.

The tools for reducing a term are similar. For single step reduction, the expression is broken down and the proof built up according to dynamic semantics given in the previous chapter.

For the many step reduction there are different strategies possible. The simplest would be to apply the tools for single step reduction repeatedly until they failed. This is inefficient since it involves repeatedly traversing the expression. Instead a new set of rules is proved and used to gain a more efficient proof. These rules are designed to resemble a big step semantics. For example, the rule for function application is

$$\frac{e_1 \longrightarrow^* (\lambda y : \alpha.\, e) \qquad (e[e_2/y]) \longrightarrow^* c}{(e_1\ e_2) \longrightarrow^* c}$$

The advantage of using many step reduction, $\longrightarrow^*$, instead of reduction to a value, $\Downarrow$, is that if any part of the term cannot be reduced then the rule

$$\frac{}{e \longrightarrow^* e}$$

can always be invoked to terminate the proof of some reduction property. This may not be reduction to a normal form but will allow the term to be reduced as far as possible. Later in this section the use of HOL variables as meta variables and how the reducer can reason about these is considered.

Two choices to be made when developing the tools are how to handle substitution and how to deal with variables.

6.3.1 Substitution

There are a number of different options for handling substitution when reducing and type checking expressions. The current system expands out all substitutions as they arise. This is the simplest approach, but is inefficient for a number of reasons.

- Evaluation of expressions before substitution can simplify the expression and reduce the number of substitutions.
- Syntactic checks can remove substitutions. If an expression contains no free variables then any substitution will leave the expression unchanged and so the substitution need not be applied.

An alternative method would be to only evaluate the substitution function when necessary. This approach could lead to a more efficient system and could form the basis for further work

6.3.2 Variables

While the mechanism for dealing with HOL constants in SDT expressions is straightforward, HOL variables raise a more complex problem. This problem arises because instead of always reasoning about fully expanded SDT expressions, we often reason about expressions contain a HOL variable representing as arbitrary expression. When the proof tools encounter such a variable they should attempt to deduce as much as possible automatically but should raise appropriate proof obligations or fail sensibly if they cannot. The need to fail with a sensible result is important. Such failures may indicate an error in the program and a sensible error message or a proof obligation that cannot be proved can help find the error.

A goal of the form

$$a : \mathsf{Num}$$

where the theorem $a = \mathsf{num}\ 2$ occurs on the assumption list could be solved by first rewriting with the assumption and then calling the type-checker. This is not satisfactory because goals such as this should be solved automatically and rewriting unnecessarily with the assumption list could complicate the goal.

The solution adopted is to write conversions [GM93] that take a list of terms representing known facts about the variables of the current term as an argument. These conversions have the type `[term] -> term -> thm`. The theorem produced will have the form

$$A_1, A_2, \ldots, A_n \vdash e_1 = e_2$$

where $A_1, A_2, ..., A_n$ are the assumptions made to prove $e_1 = e_2$. These assumptions can then be discharged from the assumption list or turned into new proof obligations by tactics provided to handle these assumptions. In normal use the assumptions of the theorem made will be a subset of the assumption list and can be discharged automatically. If the evaluator or type checker is unable to prove some theorem, this may also be added to the assumptions of the theorem and eventually be turned into a new proof goal by the tactic applying the conversion. This goal may not be able to be proved and hence indicate an error in the proof attempt.

As an example of the use of these conversions, suppose we wished to prove the goal

$$(f\ x) : \beta$$

under the assumptions

$$\vdash\ f : \alpha \rightarrow \beta$$
$$\vdash\ x : \alpha$$

The assumptions are essential because they are the only source of the information that the type of x is α. If TYPE_CONV is the function written to call the type checker and it is called with the arguments $[f : \alpha \rightarrow \beta,\ x : \alpha]$ and applied to $(f\ x) : \beta$ it will return

$$f : \alpha \rightarrow \beta,\ x : \alpha\ \vdash\ (f\ x) : \beta\ =\ \mathsf{T}$$

The type checking will have occurred by looking up the type of f and x from the assumptions.

This tool cannot make use of arbitrary facts from the assumption list. In the current system only theorems about the definition of expressions, equality of expressions, evaluation of expressions and the type of closed expressions will be used. This allows a large class of problems to be simplified automatically while not generating too large a proof search. Tactics are provided to search the assumption list for suitable assumptions, call the conversions, and discharge the assumptions.

The tactics and conversions also take a list of statements that may be supplied by the user. These will be searched as if they were in the assumption list and, if used, will give rise to new subgoals. These goals would then need to be proved separately. This can be useful if the user knows that a fact is easily proved and believes that it will be useful in an automatic proof but it will not be proved automatically.

Support for variables in terms that are to be reduced instead of type checked is similar. For a variable x, assumptions of the form $x \longrightarrow e$, $x \longrightarrow^* e$ and $x = e$ will be used in the proof about the reduction of a term.

The collection of techniques used to deal with variables and constants allows automatic reduction of a term as far as possible using information about the constants and variables in the assumption list. In contrast to the mechanisms for pushing through reduction as far as possible it is sometimes desirable to reduce the number of reduction steps. In particular, it may only be necessary to reduce a term by expanding a few variables and constants. The reducer can be supplied with a list of constants and variables and will only expand the variables or constants in that list.

6.4 Related work

The use of an SML interpreter, the Kit Compiler [BRTT93], with HOL-ML [VG93] to perform a similar task was investigated in [Col94] and summarised in [CG94]. This work did not deal with the symbolic evaluation of expressions. The symbolic evaluation of programs that are partially made up of HOL variables was investigated by Camilleri and Zammit[CZ94].

Richard Boulton's Claret system [Bou97, Bou98] can automatically generate many similar tools to those in this chapter. Claret takes a specification of the syntax and a specification of the semantics in a denotational style and returns the syntax of the language both as ML types and types in the HOL logic along with pretty printers and parsers. The specification of the denotational semantics is used to generate rules in the logic and to generate tools for mechanizing those rules. While the Claret tools could have been used to automate some of the work discussed here if they had been available at the time, the style of semantics used is incompatible with that used here.

Donald Syme has developed a theorem proving environment, Declare, which is designed for reasoning about operational semantics [Sym97, Sym98]. Declare uses a declarative style of theorem proving. Many of the results and tools in this chapter may be able to be reproduced more easily using Declare as it contains automation specific to the task of reasoning about semantics relations similar to those here. Declare was not available when the work described here was carried out.

Chapter 7

Equivalence

The previous chapters have defined the semantics of SDT and tools for reasoning about the semantics. This chapter considers what it means for two programs to be equal. As the main relations are defined co-inductively this chapter begins by formalising co-inductive relations and the labelled transition system which, together with typing and reduction, form the components used in the definition of equivalence. The chapter concludes with the proofs that the relation introduced satisfies the required properties of equivalence. The proofs of some of these properties involve some long and complex theory development and proofs. These are necessary to prove that equivalence is a congruence but are not used in later chapters.

7.1 Co-induction

Chapter 2 introduced co-induction with familiar but informal set notation. While the notation suggests a representation of relations as sets of pairs, we choose instead to represent a binary relation as a function from two arguments to a boolean. Thus a relation R between two expressions will have type

$$R : \mathit{exp} \rightarrow \mathit{exp} \rightarrow \mathit{bool}$$

and the membership for the relationship, $(x, y) \in R$ is written $R\ x\ y$. In a later section we show how this theory can be reworked for unary relations and how it could be generalised. This theory will be used to define equivalence from the labelled transition system. Subset and union for relations expressed as functions can be defined as follows:

Definition 7.1 *If R and S are binary relations, represented as functions, the subset, $\subseteq$, and union, $\cup$, are defined to be:*

$$R \subseteq S \;=\; \forall x\, y.\, R\ x\ y \supset S\ x\ y$$

$$R \cup S = \forall x\, y.\ R\, x\, y \vee S\, x\, y$$

The usual theorems about subset and union can be easily proved. Some examples are given in the following theorem.

Theorem 7.2

$$\forall x\, y.\ x \subseteq y \wedge y \subseteq x \supset (x = y)$$

$$\forall x\, y.\ x \subseteq (x \cup y)$$

$$\forall x\, y.\ y \subseteq (x \cup y)$$

$$\forall x\, y\, z.\ x \subseteq z \wedge y \subseteq z \supset (x \cup y) \subseteq z$$

$$\forall x\, y.\ y \subseteq x \supset (x \cup y = x)$$

Proof. All the results follow directly from the definitions and some simplification.

The important definitions for the development of the theory for co-induction are

Definition 7.3 *We define what it means for f from binary relations to binary relations to be monotonic by*

$$\mathsf{Monotone}\ f = \forall x\, y.\ x \subseteq y \supset f\, x \subseteq f\, y$$

Definition 7.4 *If f is a function from binary relations to binary relations and x is a binary relation then we define what it means for x to be f-Dense by*

$$\mathsf{Dense}\ f\ x = x \subseteq (f\ x)$$

Definition 7.5 *If f is a function from binary relations to binary relations then* **gfp** f *is defined to be the function*

$$\mathsf{gfp}\ f = \lambda a\, b.\ \exists x.\ \mathsf{Dense}\ f\ x \wedge x\ a\ b$$

This definition does not define **gfp** f to be the greatest fix point of f but instead provides a way of constructing the greatest fix point. The following results show that this is the greatest fix point.

If f is a function from binary relations to binary relations then **gfp** f contains any relation which is f-dense.

Theorem 7.6

$$\forall f.\ \mathsf{Dense}\ f\ x \supset x \subseteq \mathsf{gfp}\ f$$

Proof. Follows easily from the definitions of gfp and Dense.

The greatest fixpoint is itself an f-dense relation

Theorem 7.7

$$\forall f.\ \mathsf{Monotone}\ f \ \supset\ \mathsf{Dense}\ f\ (\mathsf{gfp}\ f)$$

Proof. Follows from the definitions of Monotone, Dense and gfp, along with theorem 7.6.

Theorems 7.6 and 7.7 show that the greatest fixpoint is the largest f-dense relation associated with a monotonic function f. It is also a fix point

Theorem 7.8

$$\forall f.\ \mathsf{Monotone}\ f \ \supset\ (\mathsf{gfp}\ f \ =\ f\ (\mathsf{gfp}\ f))$$

Proof. Follows from theorem 7.6 and theorem 7.7 and the definitions of the constants.

The priniciple of co-induction can now be stated and proved.

Theorem 7.9 (Co-induction)

$$\forall f.\ (\mathsf{Monotone}\ f\ \wedge\ \mathsf{Dense}\ f\ x\ ) \supset\ x\ \subseteq\ (\mathsf{gfp}\ f)$$

Proof. Follows immediately from theorems 7.7 and 7.6.

This theorem will be used in defining a new relation, R say, as the greatest fixpoint of some monotonic function f. With these assumptions theorem 7.9 simplifies to:

$$\forall f.\ \mathsf{Dense}\ f\ x\ \supset\ x\ \subseteq\ R$$

To show that some pair of values a and b are related by R ($R\ a\ b$) we only need to find a relation S such that

$$S\ a\ b \tag{7.1}$$

and

$$\mathsf{Dense}\ f\ S \tag{7.2}$$

From equation 7.2 and theorem 7.9 we get that $S\ \subseteq\ R$ and so from the definition of subset we get

$$S\ a\ b\ \subseteq\ R\ a\ b$$

A stronger version of theorem 7.9 can also be derived.

Theorem 7.10 (Strong co-induction)

$$\vdash \forall f.\ \mathsf{Monotone}\ f\ \supset\ (x\ \subseteq\ (f\ (x\ \cup\ (\mathsf{gfp}\ f)))\ \supset\ x\ \subseteq\ (\mathsf{gfp}\ f))$$

Proof. Follows from the definitions and basic properties of the constants.

All the above results are for binary relations. An identical theory can be developed for relations taking any number of arguments. In chapter 9 a theory of co-induction for unary functions (predicate sets) is used. The same theory can be proved for a set of definitions in this form. For example, the definition of subset and union would be:

Definition 7.11 *Sunset and union for unary predicates.*

$$x\ \subseteq\ y\ =\ \forall a.\ x\ a\ \supset\ y\ a$$

$$x\ \cup\ y\ =\ \lambda a.\ x\ a\ \vee\ y\ a$$

These definitions are identical to those in the HOL predicate sets library. The proofs of the theorems corresponding to theorems 7.1 to 7.5 are a simple adaptation of the proofs of these theorems. It would be possible to generate these theorems automatically for predicates with any number of arguments to get a package with the same functionality as John Harrison's induction package [Har95].

7.2 Labelled transition system

The labelled transition system is used to represent the observable properties of terms. If an expression evaluates to a number or to a datatype constructor with no arguments then we can observe the value of the number or constructor. If an expression evaluates to a constructor for a datatype with one or more arguments then we can make further observations of each argument. If an expression does not evaluate to a literal then it must be either a type abstraction, a function abstraction or an undefined expression. If the expression is undefined then we can never make an observation about the expression while if it is an abstraction we can apply it to an arbitrary term of the right type and then make observations. This process must eventually lead to the observation of literals or constructors or to undefined expressions. The possible observations will give rise to a tree of possibly infinite depth, with the observation of numbers and nullary constructors at the leaf nodes.

The labelled transition system is introduced in two stages, the definition of the labels and the definition of the transitions relation.

$$
\begin{array}{rcl}
\textit{label} & ::= & \mathsf{numL}\ \textit{num} \\
 & | & \mathsf{applL}\ \textit{exp} \\
 & | & \mathsf{ApplL}\ \textit{exp} \\
 & | & \mathsf{destL}\ \textit{string}\ \textit{ty}\ \textit{num}
\end{array}
$$

Figure 7.1: Labels for labelled transition system.

7.2.1 Labels

The labels for the transition system are introduced as a new type in HOL using the recursive type definition package [Mel89]. The syntax is given in figure 7.1. In some settings, such as CCS [Mil89], these labels are known as actions.

The argument of the numL label is simply the value observed. The arguments to the applL and ApplL labels are the terms to which the abstraction is applied. The arguments to the destructor label destL are the name of the constructor, the datatype which it is a constructor for and the argument being observed. For a constructor with no arguments the only possible observation will have this number being 0. For a constructor with n arguments there will be n observations with numbers 1 to n. The possible observations for each expression are formalised by the transition relation.

As with all syntactic types introduced, a series of characterising theorems can be derived. The important facts about the labels are that they are distinct and their constructors are one to one.

7.2.2 Transition relation

The labelled transition system is introduced as a relation

$$\mathsf{LTS} : \textit{exp} \rightarrow \textit{exp} \rightarrow \textit{label} \rightarrow \textit{bool}$$

where *label* is the type of labels. LTS e_1 e_2 a means that under the rules for LTS the expression e_1 can make a transition to e_2 with label a. In the rest of this thesis this will be represented by the notation

$$e_1 \xrightarrow{a} e_2$$

The rules are given in figure 7.2. The expression **0** is a non-terminating, or bottom, element of type Num. This type is unimportant, as the only purpose of **0** is to give an element of which no observations can made. **0** is defined as the function

$$\mathsf{rec}\ x_{\mathsf{Num}}\ x$$

$$\frac{}{\mathsf{num}\ n \xrightarrow{\mathsf{numL}\ n} 0} \qquad \frac{e_1 \longrightarrow e_2 \quad e_1 \xrightarrow{l} e_3 \quad \vdash a : \mathsf{Num}}{e_1 \xrightarrow{l} e_3}$$

$$\frac{\vdash e_1 : \alpha \to \beta \quad \vdash e_2 : \alpha}{e_1 \xrightarrow{\mathsf{appL}\ e_2} e_1\ e_2} \qquad \frac{\vdash e_1 : \forall x.\alpha}{e_1 \xrightarrow{\mathsf{AppL}\ \beta} e_{1\beta}}$$

$$\frac{\vdash \mathsf{con}\ c_\alpha\ [] : \mathsf{Data}\ x\ \overline{m}}{\mathsf{con}\ c_\alpha\ [] \xrightarrow{\mathsf{dest}\ c\ \alpha\ 0} 0} \qquad \frac{\vdash \mathsf{con}\ c_\alpha\ [e_1\ ..\ e_n] : \mathsf{Data}\ x\ \overline{m}}{\mathsf{con}\ c_\alpha\ [e_1\ ..\ e_n] \xrightarrow{\mathsf{dest}\ c\ \alpha\ i} e_i}$$

$$\frac{e_1 \longrightarrow e_2 \quad e_2 \xrightarrow{l} e_3 \quad \vdash e_1 : \mathsf{Data}\ x\ \overline{m}}{e_1 \xrightarrow{l} e_3}$$

Figure 7.2: Rules for labelled transition system.

The bottom element of any type can be generated by the function

$$\bot \ = \ \Lambda\alpha.\ \mathsf{rec}\ x_\alpha\ x$$

so that

$$0 \ = \ \bot_{\mathsf{Num}}$$

Theorem 7.12 *Rule induction for labelled transition system.*

$$\begin{array}{l}
\forall L. \\
\quad (\forall n.\ L\ (\mathsf{num}\ n)\ \mathsf{Zero}\ (\mathsf{numL}\ n)) \wedge \\
\quad (\forall e_1\ e_2\ \alpha\ \beta.\ e_1 : \alpha \to \beta\ \wedge\ e_2 : \beta\ \supset\ L\ e_1\ (e_1\ e_2)\ (\mathsf{appL}\ b))\ \wedge \\
\quad (\forall e_1\ \alpha\ \beta\ x.\ e_1 : (\forall x.\beta)\ \supset\ L\ e_1\ (e_{1\beta})\ (\mathsf{AppL}\ \beta))\ \wedge \\
\quad (\forall e_1\ e_2\ e_3\ \alpha.\ L\ e_2\ e_3\ \alpha\ \wedge\ e_1 : \mathsf{Num}\ \wedge e_1 \longrightarrow e_2\ \supset\ L\ e_1\ e_3\ \alpha)\ \wedge \\
\quad (\forall e_1\ e_2\ e_3\ x\ \overline{m}\ \alpha.\ L\ e_2\ e_3\ \alpha\ \wedge\ e_1 : \mathsf{Data}\ x\ \overline{m}\ \wedge e_1 \longrightarrow e_2\ \supset\ L\ e_1\ e_3\ \alpha)\ \wedge \\
\quad (\forall c\ \alpha.\ L\ (\mathsf{con}\ c_\alpha\ [\,])\ 0\ (\mathsf{destL}\ c\ \alpha\ 0))\ \wedge \\
\quad (\forall c\ \alpha\ xs\ i.\ 0 < i\ \wedge\ i \leq \mathsf{LENGTH}\ xs\ \supset \\
\qquad L\ (\mathsf{con}\ c_\alpha\ xs)\ (\mathsf{EL}\ (\mathsf{PRE}\ i)\ xs)\ (\mathsf{destL}\ c\ \alpha\ i)) \\
\qquad \supset \\
\quad (\forall e_1\ e_2\ \alpha.\ e_1 \xrightarrow{e_2} \alpha\ \supset\ L\ e_1\ e_2\ \alpha)
\end{array}$$

We can show by rule induction over the transition relation that if there is a transition from an expression e_1 to an expression e_2 then e_1 and e_2 are both well typed.

$$\forall e_1\ e_2\ l.\ e_1 \xrightarrow{l} e_2\ \supset\ (\exists\alpha.\ e_1 : \alpha)$$

$$\forall e_1\ e_2\ l.\ e_1 \xrightarrow{l} e_2\ \supset\ (\exists\alpha.\ e_2 : \alpha)$$

In general e_1 and e_2 may not have the same type.

7.2.3 Passive and active types

The labelled transition system makes a distinction between two classes of types. The observable behaviours of functions and type abstractions depend only on their types while the behaviours of all the other types depend on their values. Using the same terminology as Gordon [Gor95a], these classes are referred to as passive and active types respectively. The active types are the only types where we are interested in observing the their value. For expressions of passive types we only make observations of the values created by applying them to other expressions. Two predicates over the syntax of the types, Passive and Active, are defined to test which class a type belongs to.

Definition 7.13 $\mathsf{Active}\ \alpha = (\alpha = \mathsf{Num}) \vee (\exists x\ \vec{m}.\ \alpha = \mathsf{Data}\ x\ \vec{m})$

Definition 7.14 $\mathsf{Passive}\ \alpha = (\exists \beta\ \gamma.\ \alpha = \beta \rightarrow \gamma) \vee (\exists x\ \beta.\ \alpha = \forall x.\beta)$

This distinction is important in considering the behaviour of the relations defined from the equality to be defined over terms. The proof that two expressions are equivalent will involve applying passive types to types and expressions as appropriate until an active type is produced. The main work of the proof will involve considering the possible transitions of the expression of active types.

This has important consequences for deciding the meaning of equality. If $\perp_\alpha$ is a non-terminating element of type α then it is necessary to decide if the expressions

$$\lambda x : \alpha.\ \perp_\alpha \qquad \perp_{\alpha \rightarrow \alpha}$$

are equivalent. They clearly have different behaviour with respect to the given dynamic semantics. The first expression can make no reductions and evaluates to itself while the second can make an infinite series of reductions and hence evaluation will never terminate. With the choice of active and passive types given above these terms can both make the same transitions. Both can be applied to another expression and then can make no more transitions. This reflects the decision that it is not possible to observe the behaviour of a function without applying it to something.

Other choices could have been made in defining active and passive types. The function type could have been made active by including the following two rules instead of the one given above

$$\frac{\vdash (\lambda x : \alpha.\ e_1) : \alpha \rightarrow \beta \qquad \vdash b : \alpha}{\lambda x : \alpha.\ e_1 \xrightarrow{\mathsf{appL}\ e_2} \lambda x : \alpha.\ e_1\ e_2}$$

$$\frac{\vdash e_1 : \alpha \rightarrow \beta \qquad e_1 \longrightarrow e_2 \qquad e_2 \xrightarrow{l} e_3}{e_1 \xrightarrow{l} e_3}$$

then the expressions $\lambda x : \alpha.\ \bot_\alpha$ and $\bot_{\alpha \to \alpha}$ would have different transition graphs.

One useful property, which is true of active types but not passive types, is that reduction has no effect on the possible transitions. If an expression e_1 reduces to e_2 then e_1 can make a particular transition if and only if e_2 can make the same transition.

Theorem 7.15

$$\forall e_1\ e_2\ e_3\ l.\ (\exists \alpha.\ e_1 : \alpha\ \wedge\ \mathsf{Active}\ \alpha)\ \wedge\ e_1 \longrightarrow e_2\ \supset\ (e_1 \xrightarrow{l} e_3\ =\ e_2 \xrightarrow{l} e_3)$$

Proof. From the definition of active types, the rules for the labelled transition system and theorem 5.61.

7.3 Equivalence relation

This section considers the formal definition of the equivalence relation between expressions in the language. The definition is guided by the intended co-induction property discussed earlier. For any two programs, x and y we should be able to prove their equivalence by finding a relation S which contains the pair x and y and is a bisimulation. That is, it has the property that for any $(a, b) \in S$

$$\begin{array}{l}(\forall a'.\ \forall l.\ a \xrightarrow{l} a' \supset (\exists b'.\ b \xrightarrow{l} b'\ \wedge\ (a', b') \in S\ \vee\ a' == b'))\ \wedge \\ (\forall b'.\ \forall l.\ b \xrightarrow{l} b' \supset (\exists a'.\ a \xrightarrow{l} a'\ \wedge\ (a', b') \in S\ \vee\ a' == b'))\end{array}$$

This is the usual property given in other treatments of the theory. But because we do not have a type of well-typed expressions it is necessary to either introduce this type or to define the behaviour of equivalence for terms that are not well-typed. As in the rest of this work we choose not to introduce a new type and instead modify the equivalence relation to deal with this issue.

Given that only the equivalence of well-typed terms is of interest the choice of how to handle terms that are not well-typed is not crucial. If every theorem involving the equivalence relation also includes the assumptions that all expressions involved are well-typed, then the issue will not arise in practice. But in order to simplify theorems we would prefer that such side conditions were not always necessary. Thus some consideration must be given to the various possible interpretations of the equality function for non well-typed expressions. There are three main possibilities.

- Consider all terms which are not well typed to be equal. This could thought of as modelling some universal error value.
- Define equivalence such that terms that are not well typed are equal to themselves and not to any other.

- Define equivalence such that terms that are not well typed are not equivalent to any term, including themselves.

For pragmatic reasons the third possibility is chosen. This means any theorem of the form

$$e_1 == e_2$$

carries more meaning than the other two possibilities. With the approach here the theorem states the equivalence of two terms and the fact that they are well typed. The other approaches would require a proof that the terms were well typed before any information about the structure or semantics of the terms could be deduced.

This approach distils the disadvantage of not creating a new type of well-typed expressions into one weakness. The equivalence relation will not be able to be proved to be reflexive without an assumption that the expression is well typed. That is, the statement

$$\forall e.\ e == e$$

is false while the statement

$$\forall e.\ (\exists \alpha.\ e : \alpha) \supset e == e$$

is true.

In practice most of the assumptions of the form $e : \alpha$ will already have been proved or be easy to prove. The interaction of the need to prove this assumption and the automatic proof tools designed for the system is discussed in the next chapter.

To capture the effect on untyped terms in the definition of equivalence we change the meaning of a bisimulation to

$$\begin{aligned}&\forall e_1\ e_1.\ (e_1, e_2) \in S \supset \\ &\quad (\exists \alpha.\ e_1 : \alpha \wedge e_2 : \alpha) \wedge \\ &\quad (\forall e_3.\ \forall l.\ e_1 \xrightarrow{l} e_3 \supset (\exists e_4.\ e_2 \xrightarrow{l} e_4 \wedge (e_3, e_4) \in S \vee e_3 == e_4)) \wedge \\ &\quad (\forall e_4.\ \forall l.\ e_2 \xrightarrow{l} e_4 \supset (\exists e_3.\ e_1 \xrightarrow{l} e_3 \wedge (e_3, e_4) \in S \vee e_3 == e_4))\end{aligned} \tag{7.3}$$

This property could be derived by defining the equivalence relation, $==$, as the greatest fixpoint of a function, F, with the property

$$\begin{aligned}&\forall S\ e_1\ e_2.\ (F\ S)\ e_1\ e_2 = \\ &\qquad (\exists \alpha.\ e_1 : \alpha \wedge e_2 : \alpha) \wedge \\ &\qquad (\forall e_3\ l.\ e_1 \xrightarrow{l} e_3 \supset (\exists e_4.\ e_2 \xrightarrow{l} e_4 \wedge S\ e_3\ e_4)) \wedge \\ &\qquad (\forall e_4\ l.\ e_2 \xrightarrow{l} e_4 \supset (\exists e_3.\ e_1 \xrightarrow{l} e_3 \wedge S\ e_3\ e_4))\end{aligned} \tag{7.4}$$

The property given in equation 7.3 is the property required for a proof using strong co-induction (theorem 7.10) with this definition of ==. Instead of following this approach, all the definitions are made in terms of simulation. This will allow the simplification of the definitions and proofs by exploiting the symmetry of the definition of bisimulation. The property of a simulation relation will be

$$\begin{aligned}&\forall e_1\ e_2.\ (e_1,\ e_2)\ \in\ S\ \supset\\ &\quad(\exists \alpha.\ e_1 : \alpha\ \wedge\ e_2 : \alpha)\wedge\\ &\quad(\forall e_3.\ \forall l.\ e_1 \xrightarrow{l} e_3 \supset\ (\exists e_4.\ e_2 \xrightarrow{l} e_4\ \wedge\ (e_3,\ e_4)\ \in\ S))\end{aligned} \tag{7.5}$$

An equivalence relation, ==, with the required properties will be defined in terms of the this relation.

7.3.1 Simulation

Simulation is defined in terms of the function F_{SIM} with property

Definition 7.16

$$\begin{aligned}&(F_{\mathsf{SIM}}\ S)\ e_1\ e_2\ =\\ &\quad(\exists \alpha.\ e_1 : \alpha\ \wedge\ e_2 : \alpha)\wedge\\ &\quad(\forall e_3.\ \forall l.\ e_1 \xrightarrow{l} e_3 \supset\ (\exists e_4.\ e_2 \xrightarrow{l} e_4\ \wedge\ S\ e_3\ e_4\ ))\end{aligned}$$

This function is a monotone function.

Theorem 7.17

$$\mathsf{Monotone}\ F_{\mathsf{SIM}}$$

Proof. By the definition of Monotone and simplification.

Simulation is defined as the greatest fixpoint of F_{SIM}.

Definition 7.18

$$\mathsf{SIM}\ =\ \mathsf{gfp}\ F_{\mathsf{SIM}}$$

A principle of coinduction follows from theorem 7.9.

Theorem 7.19

$$\forall x\ y.\ (\exists S.\ \mathsf{Dense}\ F_{\mathsf{SIM}}\ S\ \wedge\ S\ x\ y)\ \supset\ \mathsf{SIM}\ x\ y$$

If the definitions of Dense and F_{SIM} are expanded out, then this theorem simplifies to

$$\begin{array}{l}
\forall x\ y. \\
\quad (\exists S. \\
\quad\quad (\forall e_1\ e_2. \\
\quad\quad\quad S\ e_1\ e_2 \supset \\
\quad\quad\quad\quad (\exists \alpha.\ e_1 : \alpha\ \wedge\ e_2 : \alpha)\ \wedge \\
\quad\quad\quad\quad (\forall e_3.\ \forall l.\ e_1 \xrightarrow{l} e_3 \supset\ (\exists e_4.\ e_2 \xrightarrow{l} e_4\ \wedge\ S\ e_3\ e_4\ ))) \\
\quad\quad \wedge\ S\ x\ y) \\
\quad\quad\quad \supset \\
\quad \mathsf{SIM}\ x\ y
\end{array}$$

This theorem can be used to prove other properties of the simulation function. First, it is reflexive for well typed programs and transitive for all programs.

Theorem 7.20

$$\forall x\ \alpha.\ x : \alpha \supset \mathsf{SIM}\ x\ x$$

$$\forall x\ y\ z.\ \mathsf{SIM}\ x\ y\ \wedge\ \mathsf{SIM}\ y\ z\ \supset\ \mathsf{SIM}\ x\ z$$

Proof. For reflexivity the proof is a simple coinductive proof using the relation S_R where

$$S_R\ x\ y\ =\ (\exists \alpha.\ x : \alpha)\ \wedge\ (x = y)$$

For transitivity the proof is again by coinduction using the relation S_T where

$$S_T\ x\ z\ =\ \exists y.\ \mathsf{SIM}\ x\ y\ \wedge\ \mathsf{SIM}\ y\ z$$

7.3.2 Bisimulation

Bisimulation can now be defined in terms of simulation. The function op that reverses the order of elements in a relation given as

Definition 7.21

$$(\mathsf{op}\ S)\ x\ y\ =\ S\ y\ x$$

The definiton of the function on which bisimulation is based is

Definition 7.22

$$F_{==}\ S\ x\ y\ =\ (F_{\mathsf{SIM}}\ S\ x\ y)\ \wedge\ (\mathsf{op}\ (F_{\mathsf{SIM}}\ (\mathsf{op}\ S))\ x\ y)$$

This is a monotone function and has the same property as the function described in equation 7.4.

$$\begin{array}{l} \forall S\ e_1\ e_2.\ (F\ S)\ e_1\ e_2\ = \\ \qquad (\exists \alpha.\ e_1 : \alpha\ \wedge\ e_2 : \alpha)\ \wedge \\ \qquad (\forall e_3\ l.\ e_1 \xrightarrow{l} e_3\ \supset\ (\exists e_4.\ e_2 \xrightarrow{l} e_4\ \wedge\ S\ e_3\ e_4))\ \wedge \\ \qquad (\forall e_4\ l.\ e_2 \xrightarrow{l} e_4\ \supset\ (\exists e_3.\ e_1 \xrightarrow{l} e_3\ \wedge\ S\ e_3\ e_4)) \end{array}$$

The equivalence relation between expressions that is central to this work can now be defined as follows

Definition 7.23

$$== \ = \ (\text{gfp}\ F_{==})$$

The principles of co-induction and strong co-induction follow easily from the definiton and theorems 7.9 and 7.10.

Theorem 7.24 (A co-induction principle for ==) *If there is a relation S such that $S\ x\ y$ and for any a, b for which $S\ a\ b$*

$$\begin{array}{l} (\forall a'.\ \forall \alpha.\ a \xrightarrow{\alpha} a'\ \supset\ (\exists b'.\ b \xrightarrow{\alpha} b'\ \wedge\ S\ a'\ b'))\ \wedge \\ (\forall b'.\ \forall \alpha.\ b \xrightarrow{\alpha} b'\ \supset\ (\exists a'.\ a \xrightarrow{\alpha} a'\ \wedge\ S\ a'\ b')) \end{array}$$

then $x == y$.

Theorem 7.25 (A strong co-induction principle for ==) *If there is a relation S such that $S\ x\ y$ and for any a, b for which $S\ a\ b$*

$$\begin{array}{l} (\forall a'.\ \forall \alpha.\ a \xrightarrow{\alpha} a'\ \supset\ (\exists b'.\ b \xrightarrow{\alpha} b'\ \wedge\ S\ a'\ b'\ \vee\ a' == b'))\ \wedge \\ (\forall b'.\ \forall \alpha.\ b \xrightarrow{\alpha} b'\ \supset\ (\exists a'.\ a \xrightarrow{\alpha} a'\ \wedge\ S\ a'\ b'\ \vee\ a' == b')) \end{array}$$

then $x == y$.

This relation is an equivalence relation.

Theorem 7.26

$$\forall x\ \alpha.\ x : \alpha\ \supset\ x == x$$

$$\forall x\ y.\ x == y \supset\ y == x$$

$$\forall x\ y\ z.\ x == y\ \wedge\ y == z\ \supset\ x == z$$

Proof. The proofs of the three theorems are by co-induction using the relation S_R, S_S and S_T respectively:

$$\begin{array}{lcl} S_R\ x\ y & = & (\exists \alpha.\ x : \alpha) \wedge (x = y) \\ S_S\ x\ y & = & \mathsf{op} == \\ S_T\ x\ z & = & \exists y.\ x == y \wedge y == z \end{array}$$

Many other properties can be proved for the relation. The first that is investigated is the proof that the relation is a congruence.

7.4 Congruence

This section presents the proof that the equivalence relation, ==, is a congruence. This will form the basis of the equational reasoning system developed in the next section. The congruence rules are not proved directly but are derived from a more general theorem depending on a concept of *contexts*, closing substitutions for these contexts (*closures*), and the properties of extensions of a relation between closed terms to a relation between open terms (*open extensions*) in conjunction with closures of the terms.

The proof, using Howe's Method [How89], works by proving properties of an open extension of ==. The proof of congruence is long and involves the introduction of an additional inductively defined relation that is easily proved to be a congruence, and then proving that the two relations are equal. The mechanised proof mirrors very closely a proof on paper [Gor95a].

A special case of the result will be that for an expression C with one free variable x, any two substitutions $\mathfrak{s}_1$ and $\mathfrak{s}_2$ with the property that

$$\mathfrak{s}_1\ x \ == \ \mathfrak{s}_2\ x$$

give

$$[C]_{\mathfrak{s}_1} \ == \ [C]_{\mathfrak{s}_2}$$

The congruence rules are then obtained by specialising the context C.

For the rest of this section we discuss simulation instead of bisimulation, as the results for simulation are simpler and the bisimulation results follow easily.

7.4.1 Compatible refinement

The concept of congruence is formalised using a new relation, Compref, called the compatible refinement of a relation. The rules defining Compref R for some relation R between expressions are given in figure 7.3. A relation is a precongruence if it contains

$$\frac{}{\mathsf{Compref}\ R\ \Gamma\ \alpha(\mathsf{var}\ e)(\mathsf{var}\ e)}$$

$$\frac{}{\mathsf{Compref}\ R\ \Gamma\ \mathsf{Num}\ (\mathsf{num}\ n)\ (\mathsf{num}\ n)}$$

$$\frac{\mathsf{Compref}\ R\ \Gamma\ \mathsf{Num}\ e_1\ e_2 \qquad \mathsf{Compref}\ R\ \Gamma\ \mathsf{Num}\ f_1\ f_2}{\mathsf{Compref}\ R\ \Gamma\ \mathsf{Num}\ (\mathsf{nop}\ n\ e_1\ e_2)\ (\mathsf{nop}\ n\ f_1\ f_2)}$$

$$\frac{\mathsf{Compref}\ R\ \Gamma\ \mathsf{Num}\ e_1\ e_2 \qquad \mathsf{Compref}\ R\ \Gamma\ \mathsf{Num}\ f_1\ f_2}{\mathsf{Compref}\ R\ \Gamma\ \mathsf{Bool}\ (\mathsf{bop}\ n\ e_1\ e_2)\ (\mathsf{bop}\ n\ f_1\ f_2)}$$

$$\frac{R\ \Gamma\ (\alpha \rightarrow \beta)\ e_1 f_1 \quad R\ \Gamma\ \alpha\ e_2 f_2}{\mathsf{Compref}\ R\ \Gamma\ \beta\ (e_1\ e_2)(f_1\ f_2)}$$

$$\frac{R\ (\Gamma[x \mapsto \alpha])\ \beta\ e_1\ e_2}{\mathsf{Compref}\ R\ \Gamma\ (\gamma \rightarrow \beta)\ (\lambda x : \alpha .\ e_1)\ (\lambda x : \alpha .\ e_2)}\ \gamma =_\alpha \alpha$$

$$\frac{R\ \Gamma\ (\forall x.\alpha)\ e_1\ f_1}{\mathsf{Compref}\ R\ \Gamma\ \gamma\ (e_{1\beta})\ (f_{1\beta})}\ \gamma =_\alpha x[\alpha/\beta]$$

$$\frac{R\ \Gamma\ \alpha\ e_1\ e_2}{\mathsf{Compref}\ R\ \Gamma\ \gamma\ (\Lambda x.\ e_1)(\Lambda x.\ e_2)}\ \gamma =_\alpha \forall x.\alpha$$

$$\frac{R\ (\Gamma[x \mapsto \alpha])\alpha\ e_1\ e_2}{\mathsf{Compref}\ R\ \Gamma\ \beta\ (\mathsf{rec}\ x_\alpha\ e_1)(\mathsf{rec}\ x_\alpha\ e_2)}\ \beta =_\alpha \alpha$$

$$\frac{\mathsf{all3}\ (R\ \Gamma)\ (\mathsf{Map}\ (\lambda y.\ y[\mathsf{Data}\ x\ \bar{m}/x])\ ts)\ es_1\ es_2}{\mathsf{Compref}\ R\ \Gamma\ \beta\ (\mathsf{con}\ c_\alpha\ es_1)(\mathsf{con}\ c_\alpha\ es_2)}\ \begin{array}{l}\mathsf{FDom}\ \bar{m}\ c\\ ts = \bar{m}\ c\\ \beta =_\alpha (\mathsf{Data}\ x\ \bar{m})\end{array}$$

$$\frac{\begin{array}{l} R\ \Gamma\ (\mathsf{Data}\ x\ \bar{d})\ e_1\ e_2\\ \mathsf{FDom}\ \bar{c}_1\ s = \mathsf{FDom}\ \bar{c}_2\ s\\ \forall s.\ \mathsf{FDom}\ \bar{c}_1\ s \supset\\ \quad \mathsf{FDom}\ \bar{d}\ s \land\\ \quad R\ \Gamma\ (\mathsf{makefun}\ \alpha\ (\mathsf{Map}\ (\lambda y.\ y[\mathsf{Data}\ x\ \bar{d}/x])\ (\bar{d}\ s))\ (\bar{c}_1\ s)\ (\bar{c}_2\ s)) \end{array}}{\mathsf{Compref} R\ \Gamma\ \alpha\ (\mathsf{case}\ e_1\ \bar{c}_1)\ (\mathsf{case}\ e_2\ \bar{c}_2)}$$

Figure 7.3: The definition of compatible refinement

its own compatible refinement and a congruence if it is also an equivalence relation. Compref is easily proved to be a monotone function.

7.4.2 Open extensions

An open extension of a relation is a relation which takes the same argument as the original along with closures of those arguments.

Definition 7.27 *For any relation R the open extension* Open R *is defined by*

$$\text{Open } R\ \Gamma\ \alpha\ e_1\ e_2\ =\ \Gamma \vdash e_1 : \alpha\ \wedge\ \Gamma \vdash e_2 : \alpha\ \wedge (\forall \bar{s}.\ \text{Closure } \Gamma\ \bar{s}\ \supset\ R[e_1]_{\bar{s}}[e_2]_{\bar{s}})$$

The useful properties of this definition include the reflexivity of the open extension of simulation.

Theorem 7.28

$$\forall \Gamma\ e\ \alpha.\ \Gamma \vdash e : \supset\ \text{Open SIM } \Gamma\ \alpha\ e\ e$$

Proof. Using definiton 7.27 this simplifies to the goal

$$\text{SIM } (x_{\bar{s}})\ (x_{\bar{s}})$$

with the assumptions Closure $\Gamma\ \bar{s}$ and $\Gamma \vdash e$:. These assumptions and theorem 5.56 give

$$[x]_s : \alpha$$

The result follows from theorem 7.20.

The open extension of simulation does not distinguish between alpha-equivalent types.

Theorem 7.29

$$\forall \Gamma\ \alpha\ \beta\ e_1\ e_2.\ \alpha\ ==_\alpha\ \beta\ \supset\ \text{Open SIM } \Gamma\ \alpha\ e_1\ e_2\ =\ \text{Open SIM } \Gamma\ \beta\ e_1\ e_2$$

Proof. From the definition of open (definition 7.27) and the theorem relating typing judgements with alpha-equivalent types (theorem 5.51).

7.4.3 Precongruence and congruence

Instead of directly proving that simulation is a precongruence, a new relation which can be easily proved to be a precongruence is introduced. The fact that simulation is a precongruence will be proved by showing that the open extension of simulation is equal to this new relation. This relation, which will be referred to as the candidate relation, is defined as follows:

Definition 7.30

$$\frac{\mathsf{Compref\ Cand}\ \Gamma\ \alpha\ e_1\ e_2 \quad \mathsf{Open\ SIM}\ \Gamma\ \alpha\ e_2\ e_3}{\mathsf{Cand}\ \Gamma\ \alpha\ e_1\ e_3}$$

This relation and the structure of the proof that it is equal to the open extension of simulation is very close in form to the presentation in Gordon's report [Gor95a]. A range of simple properties can again be proved including reflexivity results.

Theorem 7.31

$$\forall \Gamma\ e\ \alpha.\ \Gamma \vdash e : \alpha\ \supset\ \mathsf{Cand}\ \Gamma\ \alpha\ e\ e$$

Proof. By rule induction over the typing judgement. Each case follows from definition 7.30, the reflexivity of open simulation (theorem 7.28), the definition of compatible refinement (figure 7.3), and the typing rules.

Theorem 7.32

$$\forall \Gamma\ e\ \alpha.\ \Gamma \vdash e : \alpha\ \supset\ \mathsf{Compref\ Cand}\ \Gamma\ \alpha\ e\ e$$

Proof. By rule induction over the typing judgement. Each case follows easily the rules for compatible refinement (figure 7.3) and from theorem 7.31.

From the definition it is easy to prove that the relation, Cand, is a precongruence.

Theorem 7.33

$$\forall \Gamma\ \alpha\ e_1\ e_2.\ \Gamma \vdash e_2 : \alpha\ \supset\ \mathsf{Compref\ Cand}\ \Gamma\ \alpha\ e_1\ e_2\ \supset\ \mathsf{Cand}\Gamma\ \alpha\ e_1\ e_2$$

Proof. Follows immediately from the definition of Cand and the reflexivity of the open extension of simulation (theorem 7.28).

The proof that the relation, Cand, and the open extension of simulation are equal is done by showing that both relations contain the other. The first direction is straightforward.

Theorem 7.34

$$\forall \Gamma\ \alpha\ e_1\ e_2.\ \mathsf{Open\ Sim}\ \Gamma\ \alpha\ e_1\ e_2\ \supset\ \mathsf{Cand}\ \Gamma\ \alpha\ e_1\ e_2$$

Proof. For any Γ,α,e_1 and e_2 we have Open Sim $\Gamma\ \alpha\ e_1\ e_2$. This and definition 7.27 give $\Gamma \vdash e_1 : \alpha$. From this and theorem 7.32 we get Compref Cand $\Gamma\ \alpha\ e_1\ e_2$. The result follows from using the definition of Cand.

The proof in the other direction requires a number of lemmas expressing the relationship between the candidate relation and the open extension of simulation.

Lemma 7.35

$$\forall \Gamma\ \alpha\ e_1\ e_2\ e_3.\ \mathsf{Cand}\ \Gamma\ \alpha\ e_1\ e_3\ \wedge\ \mathsf{Open}\ \Gamma\ \alpha\ \mathsf{Sim}\ e_3\ e_2\ \supset\ \mathsf{Cand}\ \Gamma\ \alpha\ e_1\ e_2$$

Proof. Follows from the definition of Cand and the transitivity of simulation and open extentensions.

Cand is the smallest relation satifying theorems 7.33 and 7.35.

Lemma 7.36

$$\begin{array}{l}\forall R.\\ \quad ((\forall \Gamma\ \alpha\ e_1\ e_2.\ \Gamma \vdash e_2 : \alpha\ \supset\ \mathsf{Compref}\ R\ \Gamma\ \alpha\ e_1\ e_2\ \supset\ R\ \Gamma\ \alpha\ e_1 e_2) \wedge\\ \quad (\forall \Gamma\ \alpha\ e\ e_1\ e_2.\ R\ \Gamma\ \alpha\ e\ e_2\ \wedge \mathsf{Open\ SIM}\ \Gamma\ \alpha\ e_2\ e_1\ \supset\ R\ \Gamma\ \alpha\ e\ e_1))\\ \qquad \supset\\ \quad (\forall \Gamma\ \alpha\ e_1\ e_2.\ \mathsf{Cand}\ \Gamma\ \alpha\ e_1\ e_2\ \supset\ R\ \Gamma\ \alpha\ e_1\ e_2)\end{array}$$

Proof. By rule induction using the induction theorem arising from the definition of Cand.

If two expressions are related by the candidate relation and two further expressions related by the candidate relation are substituted into these expressions, then the resulting expressions are also related by the candidate relation.

Lemma 7.37

$$\begin{array}{l}\forall \Gamma\ \alpha\ e_1\ e_2.\ \mathsf{Cand}\ \Gamma\ \alpha\ e_1\ e_2\ \supset\\ \quad (\forall \Gamma_1\ x\ \beta\ f_1\ f_2.\ (\Gamma\ =\ (\Gamma_1[x \mapsto \beta]))\ \supset\\ \qquad \mathsf{Cand}\ \Gamma_1\ \beta\ f_1\ f_2\ \supset\ \mathsf{Cand}\ \Gamma_1\ \alpha\ (e_1[f_1/x])\ (e_2[f_2/x]))\end{array}$$

A related theorem states that if a substitution is made for all the free variables in the expressions related by the candidate relation, then the resulting expressions are related by the candidate relation with an empty context.

Lemma 7.38

$$\begin{array}{l}\forall \Gamma\ \alpha\ e_1\ e_2.\ \mathsf{Cand}\ \Gamma\ \alpha\ e_1\ e_2\ \supset\\ \quad (\forall \bar{s}.\ \mathsf{Closure}\ \Gamma\ \bar{s}\ \supset\ \mathsf{Cand\ FEmpty}\ \alpha\ [e_1]_{\bar{s}}\ [e_2]_{\bar{s}})\end{array}$$

Proof. Both the results are proved by a rule induction over

$$\mathsf{Cand}\ \Gamma\ \alpha\ e_1\ e_2$$

then reasoning about the compatible refinement relation and substitution.

The final collection of results needed relate to the relation S defined by

$$S\ e_1\ e_2\ =\ (\exists \alpha.\ \mathsf{Cand\ FEmpty}\ \alpha\ e_1\ e_2)$$

with the properties

Lemma 7.39

$$\forall e_1\ e_2.\ e_1 \longrightarrow e_2 \ \supset\ \forall e_3.\ S\ e_1\ e_3 \ \supset\ S\ e_2\ e_3$$

Proof. By rule induction over the reduction relation.

Lemma 7.40

$$\forall e_1\ e_2\ l.e_1 \xrightarrow{l} e_2 \supset (\forall e_3.\ S\ e_1\ e_3 \ \supset\ (\exists e_4.\ e_3 \xrightarrow{l}_4 \ \wedge\ S\ e_2\ e_4))$$

Proof. By rule induction over the transition relation and using lemma 7.40.

It follows from this that the relation S is contained in the simulation relation

Lemma 7.41

$$\forall e_1\ e_2.\ S\ e_1\ e_2 \ \supset\ \mathsf{SIM}\ e_1\ e_2$$

Proof. By co-induction using lemma 7.40 to show that S is a simulation.

Lemma 7.42

$$\forall\ \Gamma\ \alpha\ e_1\ e_2.\ \mathsf{Open}\ S\ \Gamma\ \alpha\ e_1\ e_2 \ \supset\ \mathsf{Open\ SIM}\ \Gamma\ \alpha\ e_1\ e_2$$

Proof. Follows immediately from the definition of open extension and lemma 7.41.

Lemma 7.43

$$\forall\ \Gamma\ \alpha\ e_1\ e_2.\ \mathsf{Cand}\ \Gamma\ \alpha\ e_1\ e_2 \ \supset\ \mathsf{Open}\ S\ \Gamma\ \alpha\ e_1\ e_2$$

Proof. Follows from the definition of Open and lemma 7.38.

From these properties of Cand the following theorem can be proved.

Theorem 7.44

$$\mathsf{Open\ Sim}\ = \mathsf{Cand}$$

Proof. From lemmas 7.42 and 7.43 we get the result

$$\forall\Gamma\alpha\ e_1\ e_2.\ \mathsf{Cand}\ \Gamma\ \alpha\ e_1\ e_2 \ \supset\ \mathsf{Open\ SIM}\ \Gamma\ \alpha\ e_1\ e_2$$

and theorem 7.34 gives the result in the other direction.

From this it follows that Open Sim is a precongruence since Cand is a precongruence (theorem 7.33).

To relate the results for simulation to bisimulation it is necessary to show the relationship between the open extension of simulation and the open extension of bisimulation.

Theorem 7.45

$$\begin{array}{l} \forall \Gamma\ \alpha\ e_1\ e_2. \\ \quad (\mathsf{Open\ SIM}\ \Gamma\ \alpha\ e_1\ e_2\ \wedge\ \mathsf{Open\ (op\ SIM)}\Gamma\ \alpha\ e_1\ e_2)\ \supset \\ \qquad\qquad (\mathsf{Open\ (==)})\Gamma\ \alpha\ e_1\ e_2 \end{array}$$

Proof. Follows from the definitions of Open and op and the relationship between bisimulation and simulation.

Using this and the congruence of the open extension of simulation it is easy to prove that the open extension of bisimulation is also a congruence.

Theorem 7.46

$$\begin{array}{l} \forall \Gamma\ \alpha\ e_1\ e_2. \\ \quad \Gamma \vdash e_2 : \alpha\ \supset\ \mathsf{Compref\ (Open\ (==)}\ \Gamma\ \alpha\ e_1\ e_2)\ \supset\ (\mathsf{Open(==)}\Gamma\ \alpha\ e_1\ e_2) \end{array}$$

To relate this theorem to the congruence rules for closed programs we need one further result.

Theorem 7.47

$$\forall e_1\ e_2.\ e_1\ ==\ e_2\ =\ (\exists\alpha.\ \mathsf{Open\ (==)\ FEmpty}\ \alpha\ e_1\ e_2)$$

Proof. The proof is straightforward using the definition of Open and the fact that the substitutions generated by expanding the definition of Open must be empty and hence have no effect of the terms.

A set of congruence rules for expressions are easily proved from the last two theorems and the definition of compatible refinement. For example, the rule for application is:

$$\begin{array}{l} \forall e_1\ e_2\ e_3\ e_4\ \alpha\ \beta. \\ \quad e_1 : \alpha \rightarrow \beta\ \wedge e_2 : \alpha\ \wedge\ e_1\ ==\ e_3\ \wedge\ e_2\ ==\ e_4\ \supset\ (e_1\ e_2\ ==\ e_3\ e_4) \end{array}$$

7.5 Properties of equivalence

This section describes some useful and important results including the fact that equivalence is preserved by reduction. This proof will make use of the following result about the possible transitions for active types.

Theorem 7.48

$$\begin{array}{l} \forall e_1\ e_2.\ e_1 \longrightarrow e_2\ \supset \\ \quad (\exists\alpha.\ \mathsf{Active}\ \alpha \wedge\ e_1 : \alpha)\ \supset (\forall e_3\ l.\ e_1 \xrightarrow{l} e_3\ =\ e_2 \xrightarrow{l} e_3) \end{array}$$

Proof. By rule induction over the reduction relation.

Theorem 7.49

$$\forall e_1\ e_2.\ e_1 \longrightarrow^* e_2 \supset$$
$$(\exists \alpha.\ \mathsf{Active}\ \alpha \wedge e_1 : \alpha) \supset (\forall e_3\ l.\ e_1 \xrightarrow{l} e_3 = e_2 \xrightarrow{l} e_3)$$

Proof. By induction over the definition of many step reduction.

The key theorem is

Theorem 7.50

$$\forall a\ b.\ a \longrightarrow b \supset a == b$$

Proof. This is proved by co-induction using the bisimulation $\lambda a\ b.\ a \longrightarrow b$ and a case analysis over all possible transitions. If the type of a is active then the result follows easily from theorem 7.48.

Results relating the transition relation with evaluation can also be proved. These are used in the proof of contextual equivalence and in the tools described in the next chapter.

Theorem 7.51

$$\forall e_1\ e_2\ l.\ e_1 \xrightarrow{l} e_2 \supset (\exists \alpha.\ e_1 : \alpha \wedge \mathsf{Active}\ \alpha \supset e_1 \Downarrow)$$

Proof. By rule induction over the transition $e_1 \xrightarrow{l} e_2$.

Although it has been proved that the defined equivalence relation, $==$, is a congruence it is not true that equivalence equals equality. The theorem

$$\forall e_1\ e_2 : \mathsf{exp}.\ (e_1 == e_2) = (e_1 = e_2)$$

does not hold. The two expressions

$$(\lambda x : \alpha.x)(\lambda x : \alpha.x)$$

and

$$\lambda x : \alpha.x$$

are equivalent, since the first evaluates to the second, but are not equal, since the first is an application and the second is a lambda abstraction.

In practise this does not present a problem, unless we are reasoning about a predicate over expressions that does not respect equivalence. This issue arises in chapter 9 and some solutions are discussed there.

It would be possible to define a new type by taking the quotient of the expression type with the equivalence relation. This would effectively disallow the definition of predicates over the new type that do not respect equivalence. Instead we take the approach of restricting the predicates allowed in certain situations.

7.6 Contextual equivalence

Contextual equivalence was introduced in section 3.3. The informal definition given there stated that two expressions are equal when substituted into a larger expression (the context) and the convergence behaviour of that larger expression is the same for both expressions. In formalising the definition there are number of considerations, particularly relating to the possible types of the context. In section 7.2.3 active and passive types were introduced. Equivalent expressions of active types will have the same evaluation behaviour while equivalent expressions of passive types may not. For this reason we restrict the definition of contextual equivalence to contexts of active type. A context is defined to be

Definition 7.52

$$\mathsf{Context}\ x\ \alpha\ e \ = \ (\exists \beta.\ \mathsf{Active}\ \beta \ \wedge \ [x \mapsto \alpha] \vdash e : \beta)$$

As with the other results in this section we do not prove that applicative bisimulation and contextual equivalence coincide directly but instead prove the equivalence of simulation and contextual order. Contextual order can be defined as follows.

Definition 7.53

$$\begin{aligned}\mathsf{CO}\ e_1\ e_2 \ = \ & \exists \alpha.\ e_1 : \alpha \ \wedge \ e_2 : \alpha \ \wedge \\ & (\forall x\ e.\ \mathsf{Context}\ x\ \alpha\ e \ \supset ((e[e_1/x]) \Downarrow \ \supset \ (e[e_2/x]) \Downarrow))\end{aligned}$$

The proof that contextual order includes simulation is relatively simple.

Theorem 7.54

$$\forall e_1\ e_2.\ \mathsf{SIM}\ e_1\ e_2 \ \supset \ \mathsf{CO}\ e_1\ e_2$$

Proof. For any context e we need to show that if $\mathsf{SIM}\ e_1\ e_2$ then

$$(e[e_1/x]) \Downarrow \ \supset \ (e[e_2/x]) \Downarrow$$

Since simulation is a precongruence we know that $\mathsf{SIM}\ (e[e_1/x])\ (e[e_2/x])$. Now, if $(e[e_1/x]) \Downarrow$ then it reduces to normal form and can make a transition. From this we know that $e[e_2/x]$ makes the same transition and, from theorem 7.51, that $(e[e_2/x]) \Downarrow$.

It can also be proved that simulation includes contextual order.

Theorem 7.55

$$\forall e_1\ e_2.\ \mathsf{CO}\ e_1\ e_2 \supset \ \mathsf{SIM}\ e_1\ e_2$$

Proof. The result follows immediately from a proof that contextual order is a simulation.

$$\begin{aligned} \mathsf{CO}\ e_1\ e_2 \supset\ & (\exists \alpha.\, e_1 : \alpha \wedge e_2 : \alpha) \wedge \\ & (\forall e_3.\, \forall l.\, e_1 \xrightarrow{l} e_3 \supset (\exists e_4.\, e_2 \xrightarrow{l} e_4 \wedge \mathsf{CO}\ e_3\ e_4\,)) \end{aligned}$$

This proof uses a case analysis of all the possible transitions and properties of the transition and evaluation relations. In the following chapters only the co-inductively defined equivalence is used. It would be possible to develop other proofs using contextual equivalence but this is not investigated here.

7.7 Related work

All the theory in this section is based on Gordons work on operational theories for functional programming language [Gor93b, Gor94, Gor93a, Gor95a, Gor95b]. The general structure of the theory and the structure of many individual proofs follows this work closely. The main differences are the details of the language.

Other approaches can be used for developing a co-inductively defined equivalence for a functional programming language without introducing a labelled transition system. This can be done by defining equivalence in terms of a large step evaluation relation [Gor94, Pit97]. Expressions are equivalent if they have the same termination behaviour and evaluate to terms that are equivalent. This approach has been used by Ambler and Crole [AC99] as a basis of a mechanised theory similar to that presented in this chapter.

The small step semantics and labelled transition system was used here because it is finer grained and provides more information about the way an expression is decomposed. This can be useful when reasoning about infinite structures and for finding errors in expressions during a proof attempt.

Chapter 8

Supporting formal reasoning

This chapter discusses various reasoning principles that can be developed from the theory of equivalence derived in the previous chapter. These are used to develop the tools that are necessary to make reasoning practical.

The tools described in chapter 6 can automate the proofs of many facts about the evaluation and typing relations. Although this allows the automation of many of the trivial steps in a proof, the tools described are not sufficient to raise the level of interaction to a high level. The typing and evaluation conversions must still be applied by hand, as must the theorems we need about labelled transition systems. This section gives some examples of how the tools can be combined with commonly used meta-theorems to produce the higher level proof tools that give the desired level of interaction. This chapter then gives a series of examples of the use of the tools.

8.1 Constants, equivalence and transitions

Chapter 6 introduced ML programs that simplified the definition and use of SDT datatypes and functions. These tools stored theorems about the typing and reduction for the new constants introduced. In the last chapter the transition system and equivalence relation were added. The tools can be extended to store results about these for the introduced constants.

The data stored for constructors is updated with two new fields. One stores a theorem stating the possible transitions for a given constructor and the other stores a congruence rule. The expanded record is:

```
type Coninfo = { Name:string,      (*The name of the constuctor*)
                 Def:thm,          (*The definition of the constructor*)
                 Ty:thm option,    (*The theorem giving the type*)
```

```
        Lts:thm option,   (*The theorem giving the transitions*)
        Cong:thm option   (*The congruence rule*)
      }
```

For lists the new theorems for the possible transitions would be:

$$\forall e\ \alpha\ l.\ \mathsf{Nil}_\alpha \xrightarrow{l} e \ = \ (e \ = \ \mathsf{Zero}) \wedge (l \ = \ \mathsf{destL}\ \mathit{Nil}\ (\mathsf{List}\ \alpha)\ 0)$$

$$\begin{aligned}&\forall e\ e_1\ e_2\ l\ \alpha.\ \mathsf{Cons}_\alpha\ e_1\ e_2 \xrightarrow{l} e \ = \\ &\qquad (e \ = \ e_1) \ \wedge \ (l \ = \ \mathsf{destL}\ \mathit{Cons}\ (\mathsf{List}\ \alpha)\ 1) \vee \\ &\qquad (e \ = \ e_2) \ \wedge \ (l \ = \ \mathsf{destL}\ \mathit{Cons}\ (\mathsf{List}\ \alpha)\ 2)\end{aligned}$$

The congruence rule for Cons is

$$\begin{aligned}&\forall x\ y\ xs\ ys\ \alpha. \\ &\quad x : \alpha \ \wedge \ xs : \mathsf{List}\ \alpha \ \wedge \ x == y \ \wedge \ xs == ys \ \supset \ \mathsf{Cons}_\alpha\ x\ xs == \mathsf{Cons}_\alpha\ y\ ys\end{aligned}$$

The data stored for a datatype is updated by adding one new field. This stores the possible transitions that any expression of that type can make. This is simply the sum of the possible transitions for the constructors of the type.

The definition and reduction rules for map given earlier differ from the normal rules that would be used to define map. The use of the case split would normally be replaced by pattern matching. The rules would be

$$\begin{aligned}\mathsf{map}_\alpha\ f\ [\,] \quad &== \ [\,] \\ \mathsf{map}_\alpha\ f\ (\mathsf{Cons}_\alpha\ x\ xs) \quad &== \ \mathsf{Cons}_\alpha\ (f\ x)\ (\mathsf{map}_\alpha\ f\ xs)\end{aligned}$$

for correctly typed arguments. These rules can be easily proved from the definition of map. The next section introduces tools for rewriting with equivalence and it is useful to have the rules representing these pseudo-definitions stored to be used for rewriting. A new field is added to the functions record to store this information although at the time of definition no theorem is stored. These can be added later if they are proved. In principle an automated tool could be added to take a specification in this more natural form, work out the necessary underlying function and then prove these rules. This could form the basis of future work.

8.2 Equational reasoning

This section introduces the equational reasoning system that is based on the congruence results in the previous chapter. The rewriter simply takes a list of equivalences and an expression and traverses the expression substituting terms from the list where they match sub-expressions.

The rewriter is based on the fact that the relation $==$ is a congruence. An ML function is defined to prove the appropriate congruence results when needed. Such a function is called a conversion. The type of a conversion is **conv** which is equal to the type `term -> thm`. A conversion takes a term t and returns a theorem of the form $\vdash t = t'$. Thus a conversion proves the equality of a term to some other term and returns a theorem stating this equality. Following Paulson, conversions are used as the basis for implementing rewriting [Pau83].

The function used here to prove congruence results, `cong_CONV:thm -> conv`, is a slight variation on this general pattern. It takes a theorem of the form $\vdash e_1 == e_2$ and returns a conversion that proves that any expression e is equivalent to the expression formed by replacing all occurrences of e_1 in e by e_2. If the new expression is denoted by e' then the theorem returned is of the form $e == e'$. Note that it does not return $e = e'$.

This conversion can then be used as a basis for a tactic

```
EQUIV_REWRITE_TAC : thm list -> tactic
```

which takes a list of theorems of the form $e_1 == e_2$. The tactic reduces a goal of the form $e == e'$ by rewriting all sub-terms of this goal matching the left hand sides of one of the list of theorems to the right hand side. A variation on this is the tactic `ASM_EQUIV_REWRITE_TAC : thm list -> tactic` which also rewrites with any assumptions of the form $e_1 == e_2$.

Type checking may be required to complete the rewriting process, since the reflexive property and congruence rules must be used and these contain typing judgements as side conditions. The rewriting conversions will use the type checker to automate these proofs where possible. If any of these proofs fail then the side condition will be turned into an assumption of the resulting theorem in a similar way to the treatment of variables in chapter 6.

8.3 Variables

The type-checker and reducer introduced in chapter 6 contain support for handling logical variables in the expressions they are applied to. This section discusses the extension of these tools to also make use of facts about the equivalence of expressions.

For type checking there are several additional types of assumption which are useful for deciding the type of an expression. These can be searched efficiently for additional information. The new assumptions that are handled include statements of the form $x == e$ where x is a variable. If the type checker is applied to an expression containing the variable x then it will try to type check e and use that result in the proof. This is often preferable to rewriting the goal to replace x with e. If e is a large term then such a rewrite may increase the size of the goal considerably, particularly if x appears more than once.

This approach could be extended to include a wider variety of statements, such as an equivalence consisting of a term other than a variable on the left hand side. This would require more to find a proof automatically. In most cases, adding a new term to the list of facts used by the tools can be used instead and the resulting subgoal proved.

The same extensions cannot be made to the reducer. When trying to prove that a term e_1 reduces to a term e_2, the fact that e_1 contains a variable x and $x == e$ cannot be used in reducing e_1. The central problem is that, while the statement

$$e_1 \longrightarrow^* e_2 \supset e_1 == e_2 \tag{8.1}$$

is true, the statement

$$e_1 == e_2 \supset e_1 \longrightarrow^* e_2$$

is not in general

8.4 Coinduction

The tactics for co-induction and strong co-induction work by manipulating the goal, applying a theorem and then simplifying the resulting subgoals. The tactics take a relation S and manipulate the goal, such as stripping away some universal quantifiers and assumptions, so that it is in the form $a == b$ suitable for the application of one of the principles of co-induction. The tactics then tidy up the resulting subgoals and attempt to solve any subgoals involving only type checking. For some simple cases they also prove that $S\ a\ b$.

In general the work here will not try to determine what the bisimulation relation S is. One tactic GUESS_COINDUCT_TAC will try the simplest relation. For an equivalence $e_1 == e_2$ this will be the relation containing pairs of the form $(e_1,\ e_2)$ generalised over any HOL variables in the expressions. Dennis [Den99] has investigated the use of proof planning to work out the bisimulation relations for functional programming languages.

8.4.1 Labelled transition system

There are a number of theorems about the labelled transition system that depend on the evaluation and typing relations. Rather than force the user to apply the evaluation and type tactics explicitly, higher level tools are provided to apply the lower level tools automatically.

For example, a tool to apply a result about the labelled transition system is the conversion that applies theorem 7.49. The theorem is

$$\begin{aligned}&\forall e_1\ e_2.\ e_1 \longrightarrow^* e_2 \supset \\ &\qquad (\exists \alpha.\ \mathsf{Active}\ \alpha \wedge e_1 : \alpha) \supset (\forall e_3\ l.\ e_1 \xrightarrow{l} e_3 \;=\; e_2 \xrightarrow{l} e_3)\end{aligned}$$

The conversion **LTS_REDUCE_CONV : term list -> conv** takes a list of terms, typically derived from the assumption list as in chapter 6, and tries to prove the equation

$$e_1 \xrightarrow{l} e_3 \;=\; e_2 \xrightarrow{l} e_3$$

by evaluating e_1, then instantiating the theorem above to the appropriate terms and using the reduction theorem to remove the antecedent of the implication. The condition that the type is active is proved by using the type checker to decide the type of e_1 or raising this as a separate proof obligation if that fails. A tactic, **LTS_REDUCE_TAC**, applies this conversion to any transitions in the goal.

For example, the function map was introduced earlier and the function iterate can be defined easily. The following equations specify the behaviour of these functions.

$$\begin{aligned}&\mathsf{map}_{\alpha\ \beta}\ f\ \mathsf{Nil}_\alpha \;==\; \mathsf{Nil}_\beta \\ &\mathsf{map}_{\alpha\ \beta}\ f\ (\mathsf{Cons}_\alpha\ x\ xs) \;==\; \mathsf{Cons}\ (f\ x)\ (\mathsf{map}\ f\ xs) \\ &\mathsf{iterate}_\alpha\ f\ x \;==\; \mathsf{Cons}_\alpha\ x\ (\mathsf{iterate}\ f\ (f\ x))\end{aligned}$$

The following statement can be proved by co-induction using the tools described here.

$$\begin{aligned}&\forall f\ \alpha\ x.\ f : \alpha \rightarrow \alpha \ \wedge\ x : \alpha \supset \\ &\qquad \mathsf{iterate}_\alpha\ f\ (f\ x) \;==\; \mathsf{map}_{\alpha\ \alpha}\ f\ (\mathsf{iterate}_\alpha\ f\ x)\end{aligned}$$

First the tactic **GUESS_COINDUCT_TAC** is applied. The relation that this tactic chooses to use is the one relating any expression a and b where

$$\begin{aligned}a &= \mathsf{iterate}_\alpha f'\ (f'\ x') \\ b &= \mathsf{map}_{\alpha\ \alpha}\ f'\ (\mathsf{iterate}_\alpha\ f'\ x')\end{aligned}$$

for some f' and x'. The tactic also automatically proves that a and b have the correct type and that the left and right hand sides of the original goal are in the relation. It only remains to show that the relation is a bisimulation. There are two subgoals generated, the first of which is

$$\frac{\begin{array}{l} a \xrightarrow{l} a' \supset (\exists b'.\, b \xrightarrow{l} b' \wedge \\ \qquad ((\exists f\ x. \\ \qquad\quad (a' = \mathsf{iterate}_\alpha\ f\ (f\ x)) \wedge \\ \qquad\quad (b' = \mathsf{map}_{\alpha\ \alpha}\ f\ (\mathsf{iterate}_\alpha f\ x)) \wedge \\ \qquad\quad f : \alpha \rightarrow \alpha \wedge \\ \qquad\quad x : \alpha) \vee \\ \qquad a' == b')) \end{array}}{\begin{array}{l} a = \mathsf{iterate}_\alpha\ f'(f'\ x') \\ b = \mathsf{map}_{\alpha\ \alpha}\ f'\ (\mathsf{iterate}_\alpha\ f'\ x') \\ f' : \alpha \rightarrow \alpha \\ x' : \alpha \end{array}}$$

The second goal is similar and relates to matching any transitions for b with transitions for a. The proofs of the two goals are identical so only the first is considered here. If LTS_REDUCE_TAC is applied to the goal, then the expressions on the left hand side of the transition are reduced to give a new goal.

$$\begin{array}{l} \mathsf{Cons}_\alpha\ (f'\ x')\ (\mathsf{iterate}_\alpha\ f'\ \ (f'(f'\ x'))) \xrightarrow{l} a' \supset \\ \quad (\exists b'.\, \mathsf{Cons}_\alpha\ (f'\ x')\ (\mathsf{map}_{\alpha\ \alpha}\ f'\ (\mathsf{iterate}\ f'\ (f'\ x'))) \xrightarrow{l} b' \wedge \\ \qquad ((\exists f\ x. \\ \qquad\quad (a' = \mathsf{iterate}_\alpha\ f\ (f\ x)) \wedge \\ \qquad\quad (b' = \mathsf{map}_{\alpha\ \alpha}\ f\ (\mathsf{iterate}_\alpha f\ x)) \wedge \\ \qquad\quad f : \alpha \rightarrow \alpha \wedge \\ \qquad\quad x : \alpha) \vee \\ \qquad a' == b')) \end{array}$$

The next step in the proof is to analyse the possible values of the label and right hand sides of the transition.

This is done using a second conversion, LTS_CASE_CONV, or associated tactic, LTS_CASE_TAC, which performs a case analysis on the structure of an expression to determine the possible transitions. For the example above the transistion

$$\mathsf{Cons}_\alpha\ (f'\ x')\ (\mathsf{iterate}_\alpha\ f'\ \ (f'(f'\ x'))) \xrightarrow{l} a'$$

can be simplifed using the theorem about the possible transition for Cons that is stored in the global state. The theorem is

$$\begin{array}{l} \forall e\ e_1\ e_2\ l\ \alpha.\ \mathsf{Cons}_\alpha\ e_1\ e_2 \xrightarrow{l} e \ = \\ \qquad (e = e_1) \wedge (l = \mathsf{destL}\ \mathit{Cons}\ (\mathsf{List}\ \alpha)\ 1) \vee \\ \qquad (e = e_2) \wedge (l = \mathsf{destL}\ \mathit{Cons}\ (\mathsf{List}\ \alpha)\ 2) \end{array}$$

In the example this means that either

$$a' = f'\ x' \text{ and } l = \mathsf{destL}\ \mathit{Cons}\ (\mathsf{List}\ \alpha)\ 1$$

or

$$a' = \mathsf{iterate}_\alpha\ f'\ (f'(f'\ x')) \text{ and } l = \mathsf{destL}\ \mathit{Cons}\ (\mathsf{List}\ \alpha)\ 2$$

The first case is solved by letting $b' = (f'\ x')$, since $(f'\ x') == (f'\ x')$ by reflexivity. The remaining case is solved by letting $b' = (\mathsf{map}_{\alpha\ \alpha}\ f'\ ((\mathsf{iterate}_\alpha\ f')\ (f'\ x')))$. The result follows since the values for a' and b' are in the bisimulation. This is shown by proving that

$$\begin{aligned}
&\exists f\ x. \\
&\quad (a' = \mathsf{iterate}_\alpha\ f\ (f\ x)) \wedge \\
&\quad (b' = \mathsf{map}_{\alpha\ \alpha}\ f\ (\mathsf{iterate}_\alpha f\ x)) \wedge \\
&\quad f : \alpha \rightarrow \alpha \wedge \\
&\quad x : \alpha
\end{aligned}$$

This can be done by choosing f' as the witness for f and $f'\ x'$ as the witness for x.

This proof is typical of examples where the programs are generating lists. In this case we do not have to reason about undefined lists. There are added complications when this simplification cannot be made and these are discussed in a later example.

8.5 Strictness of functions

There are some programs that cannot be reduced sufficiently to allow case analysis of the possible transition and additional information must be derived before the tools described above can be applied. For example, suppose the map function is applied to some unknown list xs

$$\mathsf{map}_{\alpha\ \beta}\ f\ xs$$

If xs is the empty list then the program will evaluate to Nil and if xs is a Cons then the program will evaluate to a Cons. But xs may be undefined, in which case we cannot evaluate the application of map. This is because map is strict in its second argument; if the second argument of an application of map cannot be evaluated then neither can the application of map. In many cases it is possible to make use of this strictness information in co-inductive proofs.

Consider a goal of the form

$$(\mathsf{map}_{\alpha\ \beta}\ f\ xs \xrightarrow{l} e) \supset P$$

Although xs is a variable, this does give sufficient information about the reduction behaviour of xs to perform the case analysis described above. We can prove that if any program of List type makes a transition, then it must also evaluate to some value. So $\mathsf{map}_{\alpha\,\beta}\ f\ xs$ can be evaluated and because map is strict in its second argument we know that xs evaluates. Inspection of the type of x will give the possible values of x and then LTS_REDUCE_CONV and LTS_CASE_CONV can be applied.

8.5.1 Evaluation to normal forms and strictness

The argument in the example above was phrased in terms of the strictness of map in its second argument. There are different ways to handle the propagation of such information. For example, a predicate Strict could be defined to test if any function is strict in an argument

$$\mathsf{Strict}\ f \;=\; (\exists \alpha\, \beta.\ f : \alpha \to \beta \;\wedge\; f\ \bot_\alpha \;==\; \bot_\beta)$$

and a theory built up from this to allow reasoning about the strictness of functions.

The approach taken here is not to reason directly about the strictness of functions but instead to reason about the evaluation behaviour of the functions. An alternative, equivalent definition of Strict would be that

$$\mathsf{Strict}\ f \;=\; (\exists \alpha\, \beta.\ f : \alpha \to \beta) \;\wedge\; (\forall e.\ (f\ e) \Downarrow \supset\ e \Downarrow)$$

although this is not used explicitly. The results about evaluation behaviour are defined when needed and not stored as theorems involving the constant Strict.

The main results in this section relate to the mechanism for propagating the information about the evaluation of an expression and its sub-expressions through a goal. For example, a common goal will be of the form

$$\frac{F[e] \xrightarrow{\alpha} z}{G[e] \xrightarrow{\beta} y}$$

where F and G are expressions containing the expression e as a sub-expression. If G is strict in e then we can make use of the assumption that $G[e]$ makes transition and hence has a normal form to show that e must reduce to some value. Theorem 7.51 is used to get the initial evaluation result that forms the basis for the proof. This theorem states that

$$\forall e_1\ e_2\ l.\ e_1 \xrightarrow{l} e_2 \;\subset\; (\exists \alpha.\ e_1 : \alpha \;\wedge\; \mathsf{Active}\ \alpha \supset\ e_1 \Downarrow)$$

If we can propagate the fact that $G[e]$ has this evaluation behaviour in the above goal to a result about e, then we will be able to deduce the possible transitions for $F[e]$. Some results which show this propagation are:

Theorem 8.1 *Propagation of normal form results into expressions.*

$$\forall n.\ \mathsf{num}\ n \Downarrow$$

$$\forall c\ es.\ \mathsf{case}\ c\ es \Downarrow\ \supset\ c \Downarrow$$

$$\forall op\ a\ b.\ (\mathsf{nop}\ op\ a\ b) \Downarrow\ \supset\ a \Downarrow$$

$$\forall op\ a\ b.\ (\mathsf{nop}\ op\ a\ b) \Downarrow\ \supset\ b \Downarrow$$

From these theorems a series of conversions and tactics are built that can prove results about the propagation of evaluation information. In the above example these will prove that $e \Downarrow$ since $G[e] \Downarrow$. Since the type of x is known, and it is known that it reduces to normal form, all the possible values of x can be determined and a case split made on these values. This is discussed in more detail for a specific example in the next section.

8.6 Application of the tools

This section presents a series of small examples showing the different ways of using the tools to reason about the equivalence of expressions.

8.6.1 map-compose

This example involves the interaction of two functions, map and compose. The theorem we aim to prove is

$$\begin{array}{l} \forall f\ g\ x\ t\ t_1\ t_2. \\ \quad f : t_1 \rightarrow t_2\ \wedge\ g : t \rightarrow t_1\ \wedge\ x : \mathsf{List}\ x\ \supset \\ \quad\quad \mathsf{map}_{t\,t_2}\ (\mathsf{compose}_{t\,t_1\,t_2}\ f\ g)\ x\ ==\ \mathsf{map}_{t_1\,t_2}\ f\ (\mathsf{map}_{t\,t_1}\ g\ x) \end{array}$$

The definition of map is given in figure 6.2 and the definition of compose is

Definition 8.2

$$\mathsf{compose}_{t\,t_1\,t_2}\ =\ \lambda f : t_1 \rightarrow t_2.\ (\lambda g : t \rightarrow t_1.\ (\lambda x : t.\ (f\ (g\ x))))$$

A natural equational definition can be derived from this definition. An equational theorem for compose is

$$(\mathsf{compose}_{t\,t_1\,t_2}\ f\ g)\ x\ ==\ f\ (g\ x)$$

for all appropriately typed f, g, and x.

As with the map-iterate example, the proof is by strong co-induction, using a relation S with definition

$$\begin{aligned} S\,a\,b \;=\; & \exists f\,g\,x. \\ & (a \;=\; \mathsf{map}_{t\,t_2}(\mathsf{compose}_{t\,t_1\,t_2}\ f\ g)\ x) \wedge \\ & (b \;=\; \mathsf{map}_{t_1\,t_2}\ f\ (\mathsf{map}_{t\,t_1}\ g\ x)) \wedge \\ & \mathsf{Prog}\ (t_1 \rightarrow t_2)\ f\ \wedge\ \mathsf{Prog}\ (t \rightarrow t_1)\ g\ \wedge\ \mathsf{Prog}\ (\mathsf{List}\ t)\ x \end{aligned}$$

The proof begins by applying the tactic for strong co-induction. In addition to applying co-induction this performs some automatic proof about the types of the terms and proves the theorem

$$\vdash\ S\ (\mathsf{map}_{t\,t_2}(\mathsf{compose}_{t\,t_1\,t_2}\ f\ g)\ x)\ (\mathsf{map}_{t_1\,t_2}\ f\ (\mathsf{map}_{t\,t_1}\ g\ x))$$

which states that the left and right hand sides of the original goal are included in the relation S. It remains to be shown that S is included in $F_{==}(S \cup {==})$. Two goals are generated.

1. $\forall a'\ act.\ \mathsf{LTS}\ a\ a'\ act\ \supset (\exists b'.\ (\mathsf{LTS}\ b\ b'\ act)\ \wedge\ (S\ a'\ b'\ \vee\ a' == b'))$
2. $\forall b'\ act.\ \mathsf{LTS}\ b\ b'\ act\ \supset (\exists a'.(\mathsf{LTS}\ a\ a'\ act)\ \wedge\ (S\ a'\ b'\ \vee\ a' == b'))$

where we have assumptions

$$\begin{aligned} a &= \mathsf{map}_{t\,t_2}(\mathsf{compose}_{t\,t_1\,t_2}\ f'\ g')\ x' \\ b &= \mathsf{map}_{t_1\,t_2}\ f'\ (\mathsf{map}_{t\,t_1}\ g'\ x') \end{aligned}$$

for some f', g', and x'. We cannot proceed as in the first example because we cannot evaluate either a or b unless we can evaluate x'. But, from the strictness of map, the assumption that a makes some transition and the type of x', the system can deduce that x' must evaluate to nil or some cons cell. Two goals are generated, with the new assumptions

$$\begin{aligned} &\mathsf{Eval}\ x'\ \mathsf{nil}_t \\ &\mathsf{Eval}\ x'\ (\mathsf{cons}\ h'\ t') \end{aligned}$$

for some h' and t' with the correct types.

With the possible values for x' known, a and b can be evaluated and the goal simplified using LTS_EVAL_TAC and LTS_DISCH_TAC as in the previous example. If x evaluates to nil then a and b evaluate to nil and the transition must be the Nil transition. If x evaluates to cons h' t' then two goals corresponding to the Hd and Tl transitions are generated. Each of the goals is solved by choosing a witness for b' and followed by some simple equational reasoning.

8.6.2 rotate

If we introduce the tree type into SDT and define the function rotate with the properties

$$\vdash \forall \alpha\, a.\ a : \alpha \supset \mathsf{rotate}_\alpha\ (\mathsf{Leaf}\ a)\ ==\ (\mathsf{Leaf}\ a)$$
$$\vdash \forall \alpha\, l\, r.\ l : \mathsf{tree}\ \alpha\ \wedge\ r : \mathsf{tree}\ \alpha\ \supset$$
$$\mathsf{rotate}_\alpha\ (\mathsf{Node}\ l\ r)\ ==\ \mathsf{Node}\ (\mathsf{rotate}_\alpha\ r)\ (\mathsf{rotate}_\alpha\ l)$$

we can prove the following theorem

$$\vdash \forall t\, \alpha.\ t : \mathsf{tree}\ \alpha\ \supset\ \mathsf{rotate}_\alpha(\mathsf{rotate}_\alpha\ t)\ ==\ t$$

The trees t for which this theorem hold may be finite, infinite or undefined. The proof is by co-induction using the trivial relation

$$\{(\mathsf{rotate}_\alpha(\mathsf{rotate}_\alpha\ t'), t')\}$$

The proof involves a simple analysis of the transition system. It requires some strictness analysis to start the proof. In order to analyse the transitions we need to know more about the tree t'. Because we know that rotate is strict in its first argument we know that the whole expression is dependent on the value of t'. We can then perform a case analysis over the possible constructors of tree type (Node or Leaf) and the possible transitions. In our system the whole proof is fully automated, including the strictness analysis. The actual proof script used is

```
GUESS_CO_INDUCT_TAC  THEN
LTS_STRICT_TAC THEN
LTS_SIMP_TAC
```

This script is very general and will prove a wide variety of goals for a range of different datatypes.

It is for proving results like this that co-induction is particularly useful. An alternative inductive proof would require a side condition that the tree was finite. This side condition would then need to be proved before using the theorem in subsequent rewriting.

8.6.3 Extending the bisimulation

In the earlier examples in this section the bisimulations have been trivial and consist of only the original pair of terms generalised over some of the variables. While a large class of problems can be solved by these simple bisimulations, there are problems where a more complex relation is needed. There are two common methods for adding more elements to the bisimulation. The first is to add additional pairs of terms with a different pattern. Section 2.3.2 included an example of a problem where there are two pairs in bisimulation. The theorem that can be proved is

$$\vdash \mathsf{tflist} == \mathsf{merge}_{\mathsf{Bool}}\ \mathsf{tlist}\ \mathsf{flist}$$

This can easily be proved using co-induction with the bisimulation

$$\begin{aligned}\exists e_1\ e_2.\ & (e_1 == \mathsf{tflist} \wedge e_2 == \mathsf{merge\ tlist\ flist}) \vee \\ & (e_1 == \mathsf{False::tflist} \wedge e_2 == \mathsf{False::merge\ tlist\ flist})\end{aligned}$$

The second method used to the extend the bisimulation is to pick new functions which evaluate to the function in the initial goal for some inputs. The following example, used by Dennis [DG97], illustrates this generalisation.

$$\begin{aligned}\mathsf{h}\ =\ & \mathsf{rec}\ h : (\forall\alpha.\ (\alpha \to \alpha) \to \alpha \to \mathsf{List}\ \alpha). \\ & \Lambda\alpha.\ \lambda f : \alpha \to \alpha.\ \lambda x : \alpha.\ \mathsf{Cons}_\alpha\ x\ (\mathsf{map}\ \alpha\ \alpha\ f\ (h_\alpha\ f\ x))\end{aligned}$$

This function is another way of defining the iterate function. The theorem

$$\forall f\ x\ \alpha.\ f : \alpha \to \alpha \wedge x : \alpha \supset h_\alpha\ f\ x == \mathsf{iterate}_\alpha\ f\ x$$

should hold. This is not provable using the simple bisimulation consisting of the pairs of the form

$$(h_\alpha\ f\ x,\ \mathsf{iterate}_\alpha\ f\ x)$$

because each step in the production of the list from the application of h introduces a new application of the map function into the result. The solution is to introduce a new function which will be used to capture the repeated applications of map or any other function.

$$\begin{aligned}\mathsf{fexp} =\ & \mathsf{rec}\ fexp : (\forall\alpha..\ (\alpha \to \alpha) \to \mathsf{Num} \to \alpha \to \alpha). \\ & \quad \Lambda\alpha.\lambda f : \alpha \to \alpha.\ \lambda x : \mathsf{Num}.\ \lambda y : \alpha. \\ & \qquad \mathsf{case}\ (x = 0)\ \mathsf{of} \\ & \qquad\quad \mathsf{True} \mapsto y\ \mid \\ & \qquad\quad \mathsf{False} \mapsto f\ (fexp_\alpha\ f\ (x-1)\ y)\end{aligned}$$

This allows the bisimulation for the proof to be written down as a generalisation of the original goal

$$\begin{aligned}\forall e_1\ e_2.\ (\exists n.\ & e_1 == \mathsf{fexp}_{(\mathsf{List}\ \alpha)}\ (\mathsf{map}_{\alpha\ \alpha}\ f)\ n\ (h_\alpha\ f\ x) \wedge \\ & e_2 == \mathsf{iterate}_\alpha\ f\ (\mathsf{fexp}_\alpha\ f\ n\ x))\end{aligned}$$

The equivalence of h and iterate can then be proved by co-induction using this relation.

8.6.4 An example using filter

One function that is simple to define but is hard to reason about is filter. It can be easily defined in SDT and proved to have the properties

$$\begin{array}{lcl} \mathsf{filter}_\alpha\ p\ [] & == & [] \\ \mathsf{filter}_\alpha\ p\ (\mathsf{Cons}_\alpha\ x\ xs) & == & \mathsf{case}\ (p\ x)\ \mathsf{of} \\ & & \quad \mathsf{True} \mapsto \mathsf{Cons}_\alpha\ x\ (\mathsf{filter}_\alpha\ p\ xs) \\ & & \quad \mathsf{False} \mapsto \mathsf{filter}_\alpha\ p\ xs \end{array}$$

If filter is applied to an infinite list and the predicate p is false for all elements of the list then the function will never return a value. This means that the termination behaviour depends on the value of its inputs and not on the structure of the input as with functions like map. It is possible to reason about the function, by co-induction, with arbitrary arguments but this requires a rule induction over the reduction relation. There are ways of performing this rule induction once and deriving a new proof principle that is similar but more complex than the proof based on bisimulation [Gor95a] but this has not yet been mechanised.

With the tools described here the function can still be entered into the system and properties proved of programs using filter without proving general properties of filter. For example the goal

$$\mathsf{filter}_{\mathsf{Bool}}\ \mathsf{istrue}\ \mathsf{tflist}\ ==\ \mathsf{tlist}$$

can be proved with a trivial co-induction argument.

8.7 A model of circuits

This sections discusses some experiments with using SDT to investigate a translation of Ruby [SJ90], a relational hardware description language, into Haskell [H$^+$92]. The purpose of the translation is to allow the execution of specifications in Ruby. This new functional model of circuits, known as the Slack Circuit model, was developed by Jonathan Hogg. This section does not discuss the details of the model or all the results shown using SDT. These are covered elsewhere [CH97].

8.7.1 The Slack-Circuit Model

The problem with embedding relation languages, such as Ruby, in a functional language is the need to consider dataflow with functions. But, the problem is not in the presence of dataflow as such, since most circuits have well-defined dataflow implicit in them; the problem is in the need to explicitly specify the directions in the combinators.

The Slack-Circuit model takes a different approach to other translations by modelling relations, not as functions from inputs to outputs, but as functions from a domain/range pair to a new domain/range pair. The new domain and range represent the state of the circuit's signals after the circuit has notionally executed. Consider an inverting circuit described as a relation:

$$a \ \mathbf{inv} \ b \iff a = \bar{b}$$

This specifies that the domain, a, is the logical opposite of the range, b. Although the directions are not specified here, we can implicitly determine two possible dataflows: a as input and b as output, or b as input and a as output. The following function, `inV`, gives an interpretation of this relation.

```
inV :: ([Bool],[Bool]) -> ([Bool],[Bool])
inV (as,bs) = (map not bs, map not as)
```

The domain and range types are streams of booleans. Because Haskell is a lazily evaluated functional language, these streams can be infinite. The question is which way is this circuit executed? Here we take advantage of another property of laziness to determine the appropriate order of execution. Consider the following Haskell expression:

```
ys = snd (inV (xs,ws))
```

If we evaluate `ys` then the second `map` in the definition of `inV` will be evaluated such that `ys = map not xs`. If we consider the evaluation of `ys` to be the act of *observing* the `inV` circuit, then the direction of the circuit can be said to be determined observationally. `ws` could be any value without affecting the result.

Circuit combinators can be specified in a similar manner. The standard Ruby circuit combinator is the serial composition operator which connects the range and domain of two respective circuits:

$$x \ (R; S) \ z \iff \exists y. \ x \ R \ y \ \& \ y \ S \ z$$

The encoding for the Ruby serial combinator into Haskell is shown below using the symbol `<->`.

```
(<->) :: Circuit a b -> Circuit b c -> Circuit a c
(<->) r s (a,c) = (a',c')
                  where
                    (a',b1) = r (a,b2)
                    (b2,c') = s (b1,c)
```

where `Circuit a b` is the type `(a,b) -> (a,b)`. The way in which this functions works is not clear, depends heavily on laziness and polymorphism, and contains mutually recursive definitions of a and b. The function does behave correctly when tested on inputs representing well-formed circuits and it can be encoded in SDT. This function, and many others in the model, could not be expressed directly in the logic of theorem provers such as HOL. The rest of this section discusses some experiments in using SDT to investigate the behaviour of these functions.

8.7.2 Formal execution

The first, and one of the most useful, attempts at reasoning about the model was to formally execute some circuits to investigate their behaviour. SDT allows symbolic evaluation and fine control over how far the evaluation proceeds. In particular this allows reasoning about circuits that are not behaving correctly to investigate why. Typically problems with circuits were caused by a subtle mistake in the strictness properties of the circuit. While not yielding interesting proofs, this process did prove an effective form of debugging.

8.7.3 Simple circuits

The next level of check on the behaviour of the model is to attempt to prove correctness of some simple circuits. We prove that the composition of two inverters is just the identity circuit. The theorem we want to prove is:

$$\vdash \forall d\, r.\ d : [\mathit{Bool}] \wedge r : [\mathsf{Bool}] \supset (\mathsf{inV{<}{-}{>}inV})\,(d, r) == \mathsf{iD}_{[\mathsf{Bool}]}(d, r)$$

The inverter was defined by mapping the not function over the inputs. In order to prove this theorem we prove the following result about map.

$$\vdash \forall xs.\ xs : [\mathsf{Bool}] \supset \mathsf{map\ not\ (map\ not\ }xs) == xs$$

The proof follows the usual pattern for simple co-inductive proofs described earlier in this chapter. We use the obvious bisimulation relation and use the fact that map is strict in its second argument. The proof concludes with some simple equational reasoning about the not function. The proof of the theorem about the inverter follows by some simple evaluation and rewriting. Similar results can be easily proven about other simple gates.

8.7.4 Combinator proofs

The most important proofs about the correctness of the model are the proofs of the properties of the combinators. We begin with a discussion of the relatively simple converse combinatory, which "reverses" the direction of a circuit.

```
converse =
    $\Lambda\alpha.\Lambda\beta.\lambda R : (\mathsf{Pair}\ \alpha\ \beta) \rightarrow (\mathsf{Pair}\ \alpha\ \beta).\lambda s : \mathsf{Pair}\ \beta\ \alpha.$
      case $s$ ofpair $\rightarrow\ \lambda b : \beta.\ \lambda a : \alpha.$
      (case $(R\ (a,\ b))$of
        pair $\rightarrow\ \lambda a_1 : \beta.\lambda b_1 : \beta.\ (b_1,\ a_1))$
```

As pattern matching is not part of SDT it is necessary to use the case expression to decompose the pair.

This function illustrates one of the subtle differences introduced into the embedding of the Slack-Circuit Model in Haskell. There are a number of rules in Ruby for this combinator. One of the rules is

$$\vdash \forall \alpha\ \beta\ R\ s.\ R : \mathsf{circuit}\ \alpha\ \beta\ \wedge\ s : (\alpha,\beta)\ \supset$$
$$\mathsf{converse}(\mathsf{converse}\ R)\ s == R\ s$$

This law is not provable for the Slack-Circuit Model. The provable version is

$$\vdash \forall \alpha\ \beta\ R\ d\ r.\ R : \mathsf{circuit}\ \alpha\ \beta\ \wedge\ d : \alpha\ \wedge\ r : \beta\ \supset$$
$$\mathsf{converse}(\mathsf{converse}\ R)\ (d,r) == R\ (d,r)$$

The difference between these two rules is in the conditions on the structure of the signal. Regardless of the definition of converse it is always possible to find a circuit and signal that will behave differently when reversed twice. For our definition of converse consider the circuit with behaviour $\forall x.\ R\ x\ =\ ([],[])$ and the signal $\perp$, the undefined value. The circuit returns $([],[])$ while the circuit after reversing returns $\perp$. This can be fixed by making converse less strict, but another pair of circuit and signals can be found to cause a similar problem. This is caused because the definition of converse decides whether the converse of a circuit is strict on its input and not the strictness property of the circuit. This was only noticed while attempting the proof of the above result.

The next combinator we consider is serial composition. As the purpose of the translation from a relational language to a functional one is to execute circuit specifications, the most important property is that serial composition decomposes to function composition if the directions of data flow can be resolved. What this illustrates is that for any concrete circuit, the translation from Ruby to the Slack-Circuit Model does have an advantage over a translation to a functional style. We can defer resolving the directions in the relational descriptions until after the translation and the directions will be correctly resolved by evaluation. We do not prove this for general circuits but give an example using the following function which converts a function into a circuit with a left to right data flow.

$$\mathsf{wraplr}\ =\ \Lambda\alpha.\Lambda\beta.\lambda f : \alpha \rightarrow \beta.\ \lambda s : (\alpha,\ \beta).\ (\mathsf{zz}_\alpha,\ f\ (\mathsf{fst}\ s))$$

Using serial composition to compose two functions converted to circuits decomposes into function composition as expected.

$$\vdash \forall f\ g\ d\ r.\ f : \alpha \rightarrow \beta \wedge g : \beta \rightarrow \gamma \wedge d : \alpha \wedge r : \gamma \supset$$
$$\mathsf{snd}\ ((\mathsf{wraplr}_{\alpha\,\beta} f)\texttt{<->}(\mathsf{wraplr}_{\beta\,\gamma}\ g)(d,\ r)) \ == \ g\ (f\ d)$$

A similar result holds for a right to left data flow. We have also proved the decomposition for the parallel and serial compositions of functions with data flows in different directions.

While not a general proof of correctness these proofs were useful for checking the correctness of the composition operators. We can speculate that this can be extended to all circuits that conform to some notion of being well-formed. Such conditions would not apply to Ruby and this illustrates one difference in moving between the two models. This is consistent with having to determine the directionality in the circuit before making a translation to a more conventional functional model. In the Slack-Circuit model we are able to defer such reasoning about directionality to the point where it is necessary to complete a proof.

The final result we look at for serial composition is associativity. In Ruby the theorem

$$\vdash R\texttt{<->}(S\texttt{<->}T) == (R\texttt{<->}S)\texttt{<->}T$$

hold for for any circuits R, S and T. We would hope to be able to prove a similar theorem for the Slack-Circuit Model. Unfortunately, the proof requires reasoning about the directions of the data-flows. The circuits may have arbitrarily complex data-flows and as this is determined by the circuit definitions and not just their types this would require us to formalise a directional type system for the language. This formalisation has not yet been attempted.

We can prove the result for any specific functions. For example, we can express the fact that a circuit f has no right to left data flow by assuming that

$$\exists f_1\ f_2.\ R\ (d,r) \ == \ (f_1\ d,\ f_2\ d)$$

This says that the circuit R returns a well formed pair and is only a function of it's domain. The associativity theorem can be easily proved for circuits of this form and for any specific combination of such circuits.

A number of other laws for the Ruby combinators, not involving serial composition, have been proved. The results in this section are all consistent with the expectation that the Slack-Circuit Model gives a correct translation of well-formed Ruby specifications.

8.8 Related work

The main focus of the work here has been to formalise proofs by co-induction where the bisimulation to be used is known. Other that the trivial bisimulation relations that can be found using GUESS_COINDUCT_TAC, and are sufficient to solve a large class of problems, this work does not consider any automatic means of finding the relations to be used in a co-inductive proof. Louise Dennis has investigated how to use proof planning to do this.

Chapter 9

Styles of Reasoning

The previous chapters have developed a system based on a theory of a co-inductively defined equivalence. Using the theory, we can do proofs using evaluation, equational reasoning and co-induction. In practise this is not sufficient. A stated aim of this work is to have an extensible system for which new types of reasoning can be added. This is done by proving some new results from the semantics of the language and the definition of equivalence. This chapter gives some examples of the extension of the system with new styles of reasoning. It begins with a discussion of induction for finite data structures.

9.1 Induction over finite data

Co-induction provides a proof principle for reasoning about the equality of infinite data structures. When proving properties of finite data structures, structural induction is both adequate and easier to apply. This section presents a theory of induction for finite lists derivable in the setting presented in previous chapters. None of the techniques in this section are specific to lists and could be applied to any datatypes.

The conventional form of induction for lists is

$$\begin{array}{l} \forall P. \\ \quad P\,[\,]\,\wedge \\ \quad (\forall h:\alpha\; t:\mathsf{List}\;\alpha.\; P\,t \supset P(\mathsf{Cons}_\alpha\; h\; t)) \\ \qquad \supset \\ \forall l:\mathsf{List}\;\alpha.\; P\,l \end{array}$$

As it stands this theorem is not true in our system. There are two problems. First we must restrict the lists quantified over to be finite. Second we must deal with the fact that the predicate P may not preserve equivalence. That is, P is a predicate over the syntax of lists, and could distinguish equivalent values. This issue was introduced in section 7.5.

The first problem is to restrict the induction theorem to apply only to finite lists. There are two ways to express this. The first is to say that every finite list is equivalent (==) to some list constructed only using nil and cons, rather than function applications or other expressions. A meta-level relation concrete is defined below that tests if a list is of this form.

$$\frac{}{\textsf{Concrete Nil}_\alpha\ \alpha}$$

$$\frac{\textsf{Concrete}\ xs\ \alpha}{\textsf{Concrete}\ (\textsf{Cons}_\alpha\ x\ xs)\ \alpha}$$

The second way to express this is to define an object language function length to calculate the length of a list and define a list l to be finite if (length l) $\longrightarrow^* n$ for some natural n.

Definition 9.1

$$\begin{aligned}\textsf{length} = \ & \textsf{rec}\ \textit{length} : \forall\alpha.\textsf{List}\ \alpha \to \textsf{Num}.\\ & \quad \Lambda\alpha.\ \lambda l : \textsf{List}\ \alpha.\\ & \quad\quad \textsf{case}\ l\ \textsf{of}\\ & \quad\quad\quad \textit{Nil} \mapsto 0\ \mid\\ & \quad\quad\quad \textit{Cons} \mapsto \lambda x : \alpha.\ \lambda xs : \textsf{List}\ \alpha.\ 1 + (\textit{length}_\alpha\ xs)\end{aligned}$$

A finite list can now be defined as a list that has a length. The length function will not terminate for infinite lists.

Definition 9.2 $\textsf{Finite}\ l\ \alpha = \exists n.\ \textsf{Length}_\alpha\ l \longrightarrow^* n$

The relationship between finite lists and concrete lists is given by the following theorem

Theorem 9.3

$$\forall l\ n\ \alpha.\ \textsf{Finite}\ l\ \alpha = (\exists\, l_1.\ l == l_1 \wedge \textsf{Concrete}\ l_1\ \alpha)$$

Proof. We can prove that a finite list is equal to some concrete list by induction over the length of the list and show that a concrete list is finite by rule induction.

The second difficulty with the statement of the induction theorem arises because a predicate P of type *exp* $\to$ *bool* is a predicate over the *syntax* of expressions. Two expressions which are equivalent may have different syntax and the predicate P may refer to the syntax. This problem was introduced in section 7.5. The equivalence relation == partitions all expressions into equivalence classes and it is necessary to either define the predicate P over only one member of each equivalence class or to ensure that the predicate does not distinguish between different members of the equivalence classes.

The first, unsatisfactory, solution is to define the induction theorem only over a subset of all finite lists, one representative for each equivalence class. The discussion of how to restrict induction to finite lists introduced the predicate Concrete and showed that every finite list will be equal to a list satisfying this predicate. Restricting reduction to lists satisfying the predicate Concrete restricts induction to one specific representative of each equivalence class.

Theorem 9.4

$$
\begin{array}{l}
\forall P. \\
\quad P\ \mathsf{Nil}_\alpha \wedge \\
\quad (\forall t : \alpha\ h : \mathsf{List}\ \alpha.\ P\ t\ \supset\ P\ (\mathsf{Cons}_\alpha\ h\ t)) \\
\qquad \supset \\
\quad \forall l : \mathsf{List}\ \alpha.\ \mathsf{Concrete}\ l\ \alpha \supset\ P\ l
\end{array}
$$

Proof. By rule induction for concrete lists.

The alternative restriction is to ensure the predicate P yields the same result for any two members of an equivalence class. This can easily be expressed by imposing the restriction $\forall l_1\ l_2 . l_1\ ==\ l_2\ \supset\ P\ l_1\ =\ P\ l_2$. The alternative induction theorem is:

Theorem 9.5

$$
\begin{array}{l}
\forall P. (\forall l\ l'.\ l\ ==\ l'\ \supset\ P\ l\ =\ P\ l')\ \supset \\
\quad P\ \mathsf{Nil}_\alpha \wedge \\
\quad (\forall t : \alpha\ \ h : \mathsf{List}\ \alpha. .\ P\ t\ \ \supset\ P\ (\mathsf{Cons}_\alpha\ h\ t)) \\
\qquad \supset \\
\quad \forall l : \mathsf{List}\ \alpha.\ \mathsf{Finite}\ l\ \supset\ P\ l
\end{array}
$$

Proof. By induction over the length of the list.

There are some simple syntactic conditions which are sufficient to allow the automatic proof of the precondition that the predicate preserves equivalence. In particular, many predicates will be of the form $C_1[l]\ ==\ C_2[l]$ where $C_1[l]$ and $C_2[l]$ are larger programs containing l. If $l == l'$ then $C_1[l']\ ==\ C_2[l']$ follows immediately by rewriting.

9.2 The take lemma

The take lemma [BW88] provides a simple means to prove many theorems about infinite lists. The theorem states that you can prove the equality of two infinite lists by proving that all the initial segments of the list are equal. This allows the proof to be reduced to a proof about finite lists. The function take is defined so that the following equations hold:

$$\begin{array}{lcl} \mathsf{take}_\alpha\ 0\ xs & == & [\,] \\ \mathsf{take}_\alpha\ (n+1)\ [\,] & == & [\,] \\ \mathsf{take}_\alpha\ (n+1)\ (\mathsf{Cons}_\alpha\ x\ xs) & == & \mathsf{Cons}_\alpha\ x\ (\mathsf{take}_\alpha\ n\ xs) \end{array}$$

The definiton in SDT which gives rise to these equations is:

Definition 9.6

$$\begin{array}{lll} \mathsf{take} & = & \mathsf{rec}\ \mathit{takef} : \forall\alpha.\mathsf{Num} \to \mathsf{List}\ \alpha \to \mathsf{List}\ \alpha. \\ & & \quad \Lambda\alpha.\ \lambda n : \mathsf{Num}.\ \lambda xs : \mathsf{List}\ \alpha. \\ & & \quad\quad \mathsf{case}\ (n\ =\ 0)\ \mathsf{of} \\ & & \quad\quad\quad \mathit{True} \mapsto \mathsf{Nil}_\alpha\ \mid \\ & & \quad\quad\quad \mathit{False} \mapsto\ \mathsf{case}\ xs\ \mathsf{of} \\ & & \quad\quad\quad\quad\quad \mathit{Nil} \mapsto \mathsf{Nil}_\alpha\ \mid \\ & & \quad\quad\quad\quad\quad \mathit{Cons} \mapsto \lambda x_1 : \alpha.\ \lambda xs_1 : \mathsf{List}\ \alpha. \\ & & \quad\quad\quad\quad\quad\quad (\mathsf{Cons}_\alpha\ x_1\ (\mathit{takef}_\alpha\ (n\ -\ 1)\ xs_1) \end{array}$$

The key theorem is

Theorem 9.7 (The take lemma)

$$\forall xy.\ (\forall n.\ \mathsf{take}_\alpha\ n\ x\ ==\ \mathsf{take}_\alpha\ n\ y)\ \wedge\ x : \mathsf{List}\ \alpha\ \wedge\ y : \mathsf{List}\ \alpha\ \ =\ (x\ ==\ y)$$

Proof. By co-induction using the relation

$$\lambda x\ y.\ (\forall n.\ \mathsf{take}_\alpha\ (n+1)\ x\ ==\ \mathsf{take}_\alpha\ (n+1)\ y)\ \wedge\ x : \mathsf{List}\ \alpha\ \wedge y : \mathsf{List}\ \alpha$$

and some reasoning about the evaluation behaviour of the expressions.

The reverse implication from that given in the take lemma is not interesting and is easily proved by equational reasoning.

The shape of a proof using the take lemma will be very similar to a proof using co-induction but is more restrictive. Using the take lemma an induction over the lengths of the initial segments is performed. For the base case $n = 0$ the result is trivial since $\mathsf{take}\ 0\ xs\ =\ []$ for any list xs. For the step case we prove that the heads are equal and the tails have a pattern (the inductive hypothesis).

9.3 Parametric polymorphism

Parameteric Polymorphism, or "theorems for free" as it is sometimes known [Wad89], allows the mechanical derivation of properties of expressions from their type alone. Most presentations of parametric polymorphism is based on a domain theoretic semantics. In this section a restricted version of parametric ploymorphism is considered based on the operational semantics discussed earlier.

There are two main restrictions which have been made to simplify the problem. These are to ignore both the recursion operator and datatypes. It is believed that these restrictions could be removed by additional work. Other, more recent, work on parametric polymorphism may provide a better approach to this proof [Pit98] than what follows.

The key idea is to define, for each type α, a relation $[\alpha]$. This relation will be referred to as an Action on the type. The main result of this section will be to prove that

$$(\alpha, \alpha) \in [\alpha]$$

Interesting results about functions of type α can be derived from this theorem and the definition of the relation corresponding to α.

For a function with the type of the identity function, $\forall\alpha.\alpha \rightarrow \alpha$, then the following holds.

$$\vdash (f, f) \in [\forall\alpha.\alpha \rightarrow \alpha]$$

One theorem which can be derived from this is:

$$\forall\alpha\ \beta\ x\ y.\ f_\beta\ (g\ x) \ = \ g\ (f_\alpha\ x)$$

The details are given below.

9.3.1 Admissible relations

Relations will be introduced in this sections which have several useful properties. These will be called admissible relations and are relations R between terms of two closed types such that:

- R relates only terms of the appropriate types.
- R respects equivalence.
- R relates a divergent terms ($\bot$) to another divergent term.

The formal definition is

Definition 9.8

$$\begin{aligned}
\mathsf{AdRel}\ R\ \alpha\ \beta \ = \ & \mathsf{Closed\ ftv}\ \alpha\ \wedge\ \mathsf{Closed\ ftv}\ \beta\ \wedge \\
& \forall x\ y.\ (R\ x\ y)\ \supset\ \ x : \alpha\ \wedge\ y : \beta\ \wedge \\
& \qquad\qquad\qquad\qquad (\forall x'\ y'.\ x == x'\ \wedge\ y == y' \supset\ R\ x'\ y')\ \wedge \\
& R\ \bot_\alpha\ \bot_\beta
\end{aligned}$$

In particular, consider a strict function a and the relation that relates the pairs $(x, a\ x)$ where x is any expression of the correct type. This is an admissible relation. Another useful result is that all admissible relations are non-empty. An admissible relation between types α and β contains relates $\bot_\alpha$ and $\bot_\beta$.

9.3.2 Actions on types

While the types we eventually consider will all be closed, the definition of the action on a type will need to consider free type variables. This is due to the clause in the definition for the type abstraction that involves defining a relation for the body of the abstraction, which is not closed. We use a new type context, mapping types to relations between expressions of particular types. In use these relations will be restricted to admissible relations. We write the actions on a type α as $\llbracket\alpha\rrbracket_{\overline{R}}$ where $\overline{R}$ is the mapping from types to relations.

Definition 9.9

$$\begin{array}{ll}
\llbracket\alpha\rrbracket_{\overline{R}} & = \overline{R}\ \alpha \\
\llbracket\mathsf{Num}\rrbracket_{\overline{R}}\ e\ e' & = e : \mathsf{Num} \wedge e' : \mathsf{Num} \wedge (e\ ==\ e')) \\
\llbracket\alpha \rightarrow \beta\rrbracket_{\overline{R}}\ f\ f' & = \forall e\ e'.\ \llbracket\alpha\rrbracket_{\overline{R}}\ e\ e' \supset \llbracket\beta\rrbracket_{\overline{R}}\ (f\ e)\ (f'\ e') \\
\llbracket\forall x.\alpha\rrbracket_{\overline{R}}\ e\ e' & = (\exists\overline{T_1}.\ \mathsf{FClosed\ ftv}\ \overline{T}_1\ \wedge e : [\forall x.\alpha]_{\overline{T_1}}) \wedge \\
 & \quad (\exists\overline{T_2}.\ \mathsf{FClosed\ ftv}\ \overline{T}_2\ \wedge e' : [\forall x.\alpha]_{\overline{T_2}}) \wedge \\
 & \quad (\forall A\ \beta\ \gamma.\ \mathsf{AdRel}\ A\ \beta\ \gamma\ \supset \llbracket\alpha\rrbracket_{\overline{R}[x \mapsto A]}\ e_\beta\ e'_\gamma)
\end{array}$$

The first result to be proved is that an action over a type is an admissible relation. The goal

$$\mathsf{AdRel}\ \llbracket\alpha\rrbracket_{\overline{R}}\ \alpha\ \alpha$$

is similar to the goal we want but is incorrect since t may not be a closed type. Instead we prove

$$\mathsf{AdRel}\ \llbracket\alpha\rrbracket_{\overline{R}}\ [\alpha]_{\overline{T_1}}\ [\alpha]_{\overline{T_2}}$$

for appropriate closing type substitutions $\overline{T_1}$ and $\overline{T_1}$ and relation context $\overline{R}$. The conditions for these maps is formalised by a relation RelProp defined as:

$$\begin{array}{l}
\mathsf{RelProp}\ \overline{R}\ \overline{T_1}\ \overline{T_2}\ f\ = \\
\qquad \mathsf{FClosed\ ftv}\ \overline{T_1}\ \wedge \\
\qquad \mathsf{FClosed\ ftv}\ \overline{T_2}\ \wedge \\
\qquad \forall x.\ f\ x\ \supset\ \mathsf{Dom}\ \overline{R}\ x \wedge \mathsf{Dom}\ \overline{T_1}\ x\ \wedge \mathsf{Dom}\ \overline{T_2}\ x \wedge \\
\qquad\qquad \mathsf{AdRel}\ \overline{R}x\ \overline{T_1}x\ \overline{T_2}x
\end{array}$$

The function f will normally by the function ftv α representing the list of free type variables in the type α. One important property of RelProp which shows how the maps can be extended is:

$$\begin{array}{l}\forall \overline{R}\ \overline{T_1}\ \overline{T_2}\ x\ f.\ \mathsf{RelProp}\ \overline{R}\ \overline{T_1}\ \overline{T_2}\ f\ \supset \\ \quad \forall A\ \alpha\ \beta.\ \mathsf{AdRel}\ A\ \alpha\ \beta\ \supset \\ \qquad \mathsf{RelProp}\ \overline{R}[x \mapsto A]\ \overline{T_1}[x \mapsto \alpha]\ \overline{T_2}[x \mapsto \beta]\ (\lambda x.\ (x = s)\ \vee\ f\ x)\end{array}$$

With this and similar lemmas proved we can prove that the action on a type is admissible.

Theorem 9.10

$$\forall \alpha\ \overline{R}\ \overline{T_1}\ \overline{T_2}.\ \mathsf{RelProp}\ \overline{R}\ \overline{T_1}\ \overline{T_2}\ (\mathsf{ftv}\ \alpha) \supset \mathsf{AdRel}\ \llbracket \alpha \rrbracket_{\overline{R}}\ [\alpha]_{\overline{T_1}}\ [\alpha]_{\overline{T_2}}$$

Proof. By induction over the type α.

9.3.3 The parametricity theorem

The result we are aiming at is

$$\forall e : \alpha.\ \llbracket \alpha \rrbracket_{[]}\ e\ e$$

The proof is by rule induction over the derivation of the type of e. We again need to generalise to non-empty relation enviroments and closing substitutions for the types to get the proof to work. The side conditions on the relations and substitutions are given by:

$$\begin{array}{l}\forall \overline{R}\ \Gamma\ \overline{T_1}\ \overline{T_2}\ \overline{s_1}\ \overline{s_2}\ f. \\ \quad \mathsf{respects}\ \overline{R}\ \Gamma\ \overline{T_1}\ \overline{T_1}\ \overline{s_1}\ \overline{s_2}\ f\ = \\ \qquad \mathsf{FClosed}\ \mathsf{fv}\ \overline{s_1}\ \wedge\ \mathsf{FClosed}\ \mathsf{fv}\ \overline{s_2}\ \wedge \\ \qquad \mathsf{FClosed}\ \mathsf{ftve}\ \overline{s_1}\ \wedge\ \mathsf{FClosed}\ \mathsf{ftve}\ \overline{s_2}\ \wedge \\ \qquad (\forall x.a\mathsf{FDom}\ \Gamma\ x\ \supset\ \mathsf{FDom}\ \overline{s_1}\ x\ \wedge\ \mathsf{FDom}\ \overline{s_2}\ x\ \wedge \\ \qquad\qquad\qquad\qquad \overline{s_1}\ x : [\Gamma\ x]_{\overline{T_1}})\wedge\ \overline{s_2}\ x : [\Gamma\ x]_{\overline{T_2}}))\wedge \\ \qquad (\forall x.\ \mathsf{FDom}\ \Gamma\ x\ \supset\ \overline{R}\llbracket \Gamma\ x \rrbracket_{\overline{R}}\ [[x]_{\overline{s_1}}]_{\overline{T_1}}\ [[x]_{\overline{s_2}}]_{\overline{T_2}})\ \wedge \\ \qquad (\mathsf{RelProp}\ \overline{R}\ \overline{T_1}\ \overline{T_2}\ f)\end{array}$$

Theorem 9.11 (Parametricity)

$$\begin{array}{l}\forall \Gamma\ e\ \alpha.\ \Gamma \vdash e : \alpha\ \supset \\ \quad (\forall \overline{R}\ \overline{T_1}\ \overline{T_2}\ \overline{a_1}\ \overline{a_2}. \\ \qquad \mathsf{respects}\ \overline{R}\ \Gamma\ \overline{T_1}\ \overline{T_2}\ \overline{a_1}\ \overline{a_2}\ (\mathsf{ftv}\ \alpha)\ \supset\ \llbracket \alpha \rrbracket_{\overline{R}}\ [[e]_{\overline{a_1}}]_{\overline{T_1}}\ [[e]_{\overline{a_2}}]_{\overline{T_1}}\end{array}$$

Proof. By rule induction over the typing judgement and properties of substitution.

The version of the theorem for closed terms can be proved by setting all the substitutions and contexts to empty. The side condition is trivial to prove in this case.

Theorem 9.12

$$\forall e\alpha.\ e : \alpha.\ [\![\alpha]\!]_{[]}\ e\ e$$

9.3.4 Example: The identity functions

For any function f with type $\forall\alpha.\alpha \to \alpha$ the following holds.

$$[\![\forall\alpha.\alpha \to \alpha]\!]_{[]}\ f\ f$$

The useful theorem can be derived by expanding out the definition of actions on types. For any admissible relation A between types α and β

$$[\![\alpha \to \alpha]\!]_{[\alpha\mapsto A]}\ f_\alpha\ f_\beta$$

By expanding again

$$\forall x\ y.\ \ A\ x\ y\ \supset\ A\ (f_\alpha\ x)\ (f_\beta\ y)$$

Take A to be the relation such that $A\ x\ y$ if $y\ ==\ g\ x$ where g is a strict function of type $\alpha \to \beta$.

$$\forall x\ y.\ y\ ==\ g\ x\ \supset\ f_\beta\ y\ ==\ g\ f_\alpha\ x$$

Finally some rewriting gives the result.

$$\forall x\ y.\ f_\beta\ (g\ x)\ =\ g\ (f_\alpha\ x)$$

9.4 Invariants over infinite data

All the theory and tools presented so far have dealt with proving the equivalence of two programs. None have dealt with more general properties. This allows the proof of correctness only with respect to a specification written in the same language.

To prove a more arbitrary property of a program, such as the fact that a program produces a sorted list, we need some different machinery. Many such properties can be expressed as local properties extended to the whole list. For example, a sorted list is one in which each pair of adjacent elements of the list are ordered. These predicates can be expressed as the greatest fixpoint of a function capturing the local property and the proofs can proceed by co-induction.

This section considers some simple examples of sorted lists. The definition and proofs are very similar to the theory for bisimulation. We begin by defining a function that takes a set and produces the set of all lists that have the first two elements ordered and the tail in the given set.

Definition 9.13

$$\begin{aligned} \mathcal{F}_{\mathsf{ORD}}\ S\ l \ = \ & \exists\alpha.\ l : \mathsf{List}\ \alpha\ \wedge (\mathsf{iscons}_\alpha\ l\ == \ \mathsf{True})\ \supset \\ & ((\mathsf{head}_\alpha\ l\ <= \ \mathsf{head}_\alpha(\mathsf{tail}_\alpha\ l))\ == \ \mathsf{True})\ \wedge\ S(\mathsf{tail}_\alpha\ l) \end{aligned}$$

This function is monotone and the set of ordered is lists can be defined as its' the greatest fixpoint.

Definition 9.14

$$\mathsf{ORD}\ =\ \mathsf{gfp}\ \mathcal{F}_{\mathsf{ORD}}$$

A co-induction priniciple can be easily derived:

Theorem 9.15

$$\forall l.\ (\exists S.\ \mathsf{Dense}\ \mathcal{F}_{\mathsf{ORD}}\ S\ \wedge\ S\ l)\ \supset\ \mathsf{ORD}\ l$$

Infinite lists can now to be proved to be ordered by finding a set containing the list which satisfies the Dense property for $\mathcal{F}_{\mathsf{ORD}}$. That is:

$$\begin{aligned} \forall l. S\ l\ \supset\ & \exists\alpha.\ l : \mathsf{List}\ \alpha\ \wedge (\mathsf{iscons}_\alpha\ l\ == \ \mathsf{True})\ \supset \\ & ((\mathsf{head}_\alpha\ l\ <= \ \mathsf{head}_\alpha(\mathsf{tail}_\alpha\ l))\ == \ \mathsf{True})\ \wedge\ S(\mathsf{tail}_\alpha\ l) \end{aligned}$$

If ones is the infinite list containing only the number 1, then it can be proved to be ordered using the set characterised by the function:

$$\lambda x.\ x\ ==\ \mathsf{ones}$$

Finally, if plustwo is a function that takes any number n and returns the infinite list

$$[n,\ n+2,\ n+4,\ \ldots]$$

then for any n this can be proved to be ordered using the set

$$\lambda x.\ \exists n.\ x\ ==\ \mathsf{plustwo}\ n$$

9.5 Restricted principles of definition

It has been discussed in earlier chapters that both supporting reasoning and the reasoning itself are easier if range of functions that can be defined are restricted in some way. This is particularly true of the possible termination behaviour of programs. The work here does not insist on any restricted use of the programming language, but it does, in principle, allow theory and tools to be developed to allow a programmer to obtain easier proofs if a different method of programming is used.

This section discusses one example of such a style of programming. This involves defining recursive functions that consume finite lists in terms of the fold operator [Hut98]. While this may seem restrictive, all primitive recursive functions can be defined in this way. In this section only the fold operator for lists is discussed but similar functions can be defined for other types.

Fold is defined so that:

Definition 9.16

$$\begin{aligned}
\mathsf{fold}_{\alpha\ \beta}\ f\ v\ \mathsf{Nil}_\alpha &== v \\
\mathsf{fold}_{\alpha\ \beta}\ f\ v\ (\mathsf{Cons}_\alpha\ x\ xs) &== f\ x\ (\mathsf{fold}_{\alpha\ \beta}\ f\ v\ xs)
\end{aligned}$$

Properties of fold can be proved that allow some reasoning about recursive functions defined in terms of fold to be reduced to equational reasoning. One example of this is the fusion theorem, which is proved by induction.

Theorem 9.17 *For any finite list xs*

$$\begin{aligned}
&\forall f\ g\ h\ a\ b\ \alpha\ \beta. \\
&\quad f : \alpha \rightarrow \beta\ \wedge\ g : \beta \rightarrow \alpha \rightarrow \alpha\ \wedge\ h : \alpha \rightarrow \beta \rightarrow \beta\ \wedge\ a : \alpha \wedge\ b : \beta\ \wedge \\
&\quad (\forall x.\ (f\ x) \Downarrow\ \supset\ x \Downarrow)\ \wedge \\
&\quad f\ a\ ==\ b\ \wedge \\
&\quad (\forall x\ y.\ f\ (g\ x\ y)\ == (h\ x)\ (f\ y)) \supset \\
&\qquad f\ (\mathsf{fold}_{\beta\ \alpha}\ g\ a\ xs)\ ==\ (\mathsf{fold}_{\alpha\ \beta} h\ b\ xs)
\end{aligned}$$

Proof. This is proved for finite lists using structural induction.

It is not proved for infinite lists here due to the same problem as with the filter example (8.6.4). In fact filter can be defined in terms of fold.

Chapter 10

Conclusions

In this work a system has been developed to support formal reasoning about programs written in a small, non-strict functional programming language. The system has been developed in the HOL theorem prover, which provides security and basic reasoning technology. The semantics of the language are defined in an operational style with a co-inductively defined equivalence relation. This provides support for the two main styles of reasoning necessary for a useful, extensible system. The primary use is to support reasoning about functional programs, particularly those that cannot be expressed directly in the logic of theorem provers. In order to do this, and to allow the extension of the system with new proof rules, it also supports reasoning about the semantics of the language.

10.1 Reasoning about programs

A major feature of this system is the ability to enter programs and express properties of these programs with little initial overhead. This work differs from other work using theorem provers in not imposing any restrictions on the form of the programs; any syntactically correct program can be entered. There are no restrictions on how recursion is used or on the termination behaviour of the programs. The advantages gained from this depend on the programs entered and the reasoning to be attempted. If it is intended to prove termination properties of programs for all inputs, or if a proof depends on these properties, then the work of proving the termination properties of the program will need to be carried out in any case. If the program uses only finite data then other tools such as TFL [Sli96] may lead to less work, as they are designed to support only programs such as these and automate much of the work to prove the termination properties. In many cases, proofs will require considering such properties of a program at the time of writing to obtain an easy proof. It is recognised that the ability to enter any program does not make these proofs any easier.

The advantage over other systems is strongest when the programs of interest cannot be rewritten in a style that makes termination properties easier to prove. The circuit model given in Chapter 8 was written without any consideration for the possibility of proving properties of the programs and many of the functions have no clear termination properties at all. The functions could not be written into a more proof-friendly form without fundamentally changing the meaning of the functions. The ability to reason about such functions is important and can lead to finding errors and to a clearer understanding of the functions. Even when a proof cannot be obtained, theorem proving can lead to a significantly improved understanding of a problem. The reason behind a proof failing can be greatly informative. Many functions perform correctly only for a limited set of inputs and a better understanding of that set is a useful result. Such lightweight use of theorem proving is impossible if there are large overheads involved in entering the functions into the system.

A second example of the advantage of having a small overhead for entering functions into a theorem proving tool is where only part of a larger system is considered. If this part uses functions from the rest of the system then general properties of these other functions may not be important but only their performance over a limited range of inputs or for a specific input. These functions can still be entered and used and the necessary results proved for the reduced range of inputs only. The example in section 8.6.4 used the filter function although general properties of that function are not considered. This example could not be considered in any system where the termination properties of filter were required for it to be defined.

10.2 Reasoning about semantics

In this work, reasoning about the semantics of the language serves two main functions. The first, investigated in chapters 5 and 7, is to enable reasoning about the correctness of the semantics. A group of standard results were proved, including results stating that the typing and reduction rules were deterministic and that equivalence was correct with respect to contextual equivalence.

The second function of the ability to reason about the semantics is to add new proof rules to the system. Chapter 9 presented some derivations of such results. This kind of extension is not possible unless the semantics of the language can be reasoned about and allows the system to be augmented in a safe way with proof rules that were not envisaged in the original design. These rules extend the capabilities of the system for reasoning about programs and so can feed into reasoning about programs described in the previous section.

10.3 Further work

There are a number of ways in which the work presented here could be extended, adapted or improved. These changes can be divided into future work on the language, the proof tools, and the application of the work to different languages.

10.3.1 The language

There are two major changes that can be made to the language supported by the tools described here. First, the details of the semantics could be modified to bring the operational semantics of the language more in line with the behaviour of real interpreters and compilers for functional languages. The semantics given here is a call by name semantics. The interpreters and compilers for Haskell and similar languages use lazy evaluation. In both evaluation strategies the arguments to a function are not evaluated when the function is called but are only evaluated when needed, and only evaluated as far as is necessary at any given time. In the call by name semantics given here these arguments may be duplicated when they are substituted into a term and hence evaluated more than once. In a true lazy semantics references are passed to the arguments and they are not duplicated. This makes no difference to the meaning of a program, but it increases the efficiency where it avoids the duplication of an argument.

It would be interesting to base a formal reasoning system on a true lazy semantics. This would complicate the semantics but may lead to more efficient proofs. It is not clear if this efficiency would be worth the additional complexity.

The second, and more obvious, change to the language would be to extend the syntax to allow more programming constructs to be used. There are three ways to do this. The syntax and operational semantics of the language could be extended to support new programming constructs such as pattern matching or list comprehensions. The work required to add new constructs depends largely on whether any new bindings can occur. Much of the work on the existing system was dealing with the existing binding constructs. Non-binding syntax is relatively easy to introduce.

A second method would be to add more syntactic sugar to the language in a similar way to booleans were added. Many of the tools and meta-theory treat booleans as if they were primitive, so a user sees an expanded syntax compared to the underlying syntax.

The final method would be to provide an automatic translation from a more general language, such as Haskell, to SDT. Indeed, for Haskell such a translator would be easy to develop from the Glasgow Haskell Compiler, which contains a similar translation as one of the stages in compilation [Jon96]. The practical use of a mechanical translation may be limited because much of the information about the structure of the original term is lost in the mapping. To create a useful system it would be necessary to store enough information during the translation to allow the reverse translation when presenting the user with output from the system. This would be primarily an interface issue.

On a smaller scale, pattern matching could be incorporated into the system by taking a specification using pattern matching, deriving the corresponding SDT expression, defining this and then proving the rules that incorporate the pattern matching. The underlying term need never be visible to the user. The mechanism for applying these rules already exists and was discussed in Chapter 6. The translation would be similar to tools which already exist in the HOL theorem prover [Sli96].

10.3.2 Additional proof rules

The ability to extend the system by deriving new proof rules gives a clear avenue for further work. While a selection of proof rules has been presented here, there are still more that could be added. The parametricity theory could be reworked using more recent work [Pit98] to extend its applicability to the whole language. Examples, such as the behaviour of the filter function, could be tackled using variants of the current co-induction principle described by Gordon [Gor95a]. The fold function and its corresponding theory have been treated here but the related function, unfold, dealing with the creation, rather than consumption, of lists has not yet been treated. Finally, there are families of well known laws, such as the monad laws [Bir98] which could be proved in the system.

Most of these rules are aimed at capturing some pattern of recursion which is commonly used in programs and proving some result about all programs fitting that pattern. If a proof rule is developed which captures the necessary inductive argument then these programs can be reasoned about using only equational reasoning plus the appropriate rule.

10.3.3 Tool support

There are several ways that tools support for the reasoning in the system could be enhanced. One possibility would be the automatic generation of bisimulation relations for use in coinductive proofs. This could be done by integrating with a tools such as Dennis' Co-induction Critic [DG97], which uses proof planning [Bun88, BSvH$^+$93] to generate an outline proof and corresponding bisimulation but does not formally check the proof. A loose integration of the two systems could use only the bisimulation relation while a tighter integration could use the proof generated by the proof planning as the basis for a proof in HOL. Similar support could be developed for proofs by induction, the area to which poor planning was originally applied. Much of the technology to support such an integration has been developed by the Clam-HOL project [SBB98, SGBB98].

The second area where more tool support could be provided would be through a custom user interface. The current system uses the text based interface to the HOL theorem prover. This raises issues other than the usual issues relating to interfaces to theorem proving tools [AGMT98]. The main deficiency in the current interface is that the HOL logic itself contains functions, variables and constants, leading to the possible confusion between the syntax of the logic and the syntax of SDT programs. For example, function application is best represented in both languages by juxtaposition and variables by the name of the variables. The current interface uses the same syntax to pretty print both languages, but quotation and parenthesis to mark the blocks of SDT syntax. This can be confusing and a graphical interface would allow the use of colour to distinguish between the two.

Appendix A

Notation

Expressions	e, e', e_1, e_2
Expresions (functions)	f, f'
Variable	x, y
Types	t, t', t_1, t_2
Type variables	α, β
Relations	R
Finite maps (to Relations)	$\overline{R}$
Finite maps (to Types)	$\overline{T}, \Gamma$
Finite maps (to Expressions)	$\overline{s}, \overline{s_1}, \overline{m}$
Extension of finite map	$\overline{s}[x \mapsto y]$
Application of finite map	$\overline{s}x$
Substitution with finite map	$[e]_{\overline{s}}$
Substitution for single variable	$e[e_1/x]$

Appendix B

A Theory of Finite Maps

B.1 Introduction

Functions defined on only a finite domain occur frequently in computing science. One field in which these functions, commonly referred to as finite maps or finite partial functions, are used is in reasoning about the semantics of programming languages, where they can model semantic objects such as type contexts and environments.

A commonly-used representation for finite maps is simply the theory of lists; a finite map can be represented by a list of pairs, and functions to update and apply maps can be defined easily and will behave correctly when used. Unfortunately this simple use of lists is flawed because two lists that behave the same when used as finite maps may not be logically equal, a property that is essential where reasoning about the equality of finite maps is required. These issues are discussed in more detail later.

This paper presents a theory of finite maps that will be the basis for a finite maps library in the HOL theorem prover [GM93]. This work follows the HOL tradition of taking a purely definitional approach. We characterize the theory in terms of a small set of axioms that are sufficient to capture the intended meaning of finite maps. The choice of these axioms is discussed in section 2. A model for these properties is then constructed using types and constants that already exist in HOL. Section 3 describes the representation used for this model and how the characteristic theorems are proved. Section 4 describes how the theory can be enriched with more theorems and concepts using those already defined. Section 5 addresses the issue of defining recursive types containing finite maps. This potentially difficult problem has been the motivation for using lists to model finite maps in the past, as this provides a means to define such types. Section 6 describes some decision procedures we have implemented and section 7 gives an example that uses finite maps to represent contexts for the type system for a small language.

B.2 Finite maps

This section describes the choice of axioms for the theory of finite maps. These axioms are intended to fully characterise the type, and must of course be chosen so as to be consistent. We determine that this is the case by providing a model for the axioms in later sections. It is also necessary that the choice of axioms be complete; any property we wish to prove of finite maps should be provable from these axioms. A further result, interesting for theoretical reasons, is that the axioms should be independent of each other; if any axiom is removed or weakened then the set of axioms will fail to completely specify finite maps.

The type of finite maps will be introduced by a new binary type operator *fmap*. A finite map from type α to type β has type (α, β)*fmap*. An important concept is the domain of a finite map. This is the finite set of values over which the application of a finite map will be specified.

B.2.1 Axioms for the constants

Four constants are introduced, with informal definitions as follows:

- Empty : The finite map with no elements in its domain.
- Update f (x, y) : The basic operation to allow the extension of a finite map f with a new mapping from x to y. There should be no restriction on whether or not x is already in the domain of f. If x is in the domain then the value to which x is mapped will be updated to be y. Some other formalisms of finite maps restrict Update to extension of a finite map only with elements not in the domain.
- Apply f x : If x is in the domain of f then Apply f x denotes the value to which x is mapped.
- Domain f x : The function Domain tests whether an element x is a member of the domain of f. The domain is formulated in terms of a boolean function rather than a set so that the resulting theory does not depend on a particular variety of set theory. It is a relatively trivial task to construct the domain set from the definitions given below in the user's choice of set theory.

The above informal definitions still leave some ambiguities to be resolved. In particular nothing has been said about the outcome of applying a finite map to an element not in its domain. (Apply f) : $\alpha \rightarrow \beta$ should be a partial function, only defined on the domain of f. But all functions in HOL are total and so are defined for all elements of the correct type. The traditional solution in HOL is to leave a partial function unspecified for values not in the correct domain. Thus applying a finite map f of type $(\alpha,\beta)\mathit{fmap}$ to a value that is not in the domain of f will return a value of type β, but this value will be unspecified and it will not be possible to prove which member of the type β has been returned.

An alternative approach, similar to that used in [Gun93, Sym93], is to define Apply to return a result of type $\beta + \mathit{one}$ where *one* is the type with only one element, namely the value denoted by one. Returning one indicates that the finite map is undefined for that element. This has the advantage of reducing the number of constants that need to be "axiomatised" but the disadvantage of complicating the type of the value produced by Apply. This modified apply function can, however, be defined in terms of the constants Apply and Domain introduced above.

It is claimed that the intended meaning of the constants Empty, Apply, Update and Domain can be formalised by the six basic axioms

$$\vdash \forall f\, a\, b.\ \mathsf{Apply}\ (\mathsf{Update}\ f\ (a,b))\ a\ =\ b$$

$$\vdash \forall x\, a.\ (x \neq a)\ \supset\ \forall f\, b.(\mathsf{Apply}\ (\mathsf{Update}\ f\ (a,b))\ x\ =\ \mathsf{Apply}\ f\ x)$$

$$\begin{aligned}&\vdash \forall a\, c.\ (a \neq c) \supset\\ &\quad \forall f\, b\, d.\\ &\qquad (\mathsf{Update}\ (\mathsf{Update}\ f\ (a,b))\ (c,d)\ =\ \mathsf{Update}\ (\mathsf{Update}\ f\ (c,d))\ (a,b))\end{aligned}$$

$$\vdash \forall f\, a\, b\, c.\ \mathsf{Update}\ (\mathsf{Update}\ f\ (a,b))\ (a,c)\ =\ \mathsf{Update}\ f\ (a,c)$$

$$\vdash \forall a.\ \neg(\mathsf{Domain}\ \mathsf{Empty}\ a)$$

$$\vdash \forall f\, a\, b\, x.\ \mathsf{Domain}\ (\mathsf{Update}\ f\ (a,b))\ x\ =\ (x\ =\ a)\ \vee\ \mathsf{Domain}\ f\ x$$

together with a further induction axiom which is explained in the next section.

B.2.2 Induction

The axioms in the previous section do not express the fact that the partial functions being considered are finite. In addition to these axioms, an induction principle is needed:

$$\begin{aligned}&\vdash \forall P.\\ &\quad P\ \mathsf{Empty} \wedge (\forall f.\ P\ f \supset (\forall x\, y.\ P\ (\mathsf{Update}\ f\ (x,y))))\\ &\qquad \supset\\ &\quad \forall f.\ P\ f\end{aligned}$$

This gives us the property that any finite map can be formed by a finite number of updates of the empty map. It easily follows from this that the domain of a finite map can be enumerated by some initial fragment of the natural numbers.

This induction principle is not strong enough to derive the natural characterisation of equality for finite maps (see below). Indeed, it is possible to formulate a model in which the six basic axioms hold along with this induction axiom but the characterisation of equality shown below is false.

The following stronger induction principle is sufficient to derive the characterisation of equality for finite maps. In the step case induction, this stronger principle allows us to assume that the element being added is not in the domain of the finite map.

$$\begin{array}{l} \vdash \forall P. \\ \quad P\ \mathsf{Empty} \wedge \\ \quad (\forall f.\ P\ f \supset (\forall x.\ \neg(\mathsf{Domain}\ f\ x)\ \supset\ \forall y.\ P\ (\mathsf{Update}\ f\ (x,y)))) \\ \quad\ \supset \\ \quad \forall f.\ P\ f \end{array}$$

An alternative to adding to this stronger induction axiom is to add the weaker induction theorem and the equality theorem below as axioms and derive the strong induction theorem. This would also replace other basic axioms for the constants discussed earlier. It was felt that it was better to use the basic axioms and derive the equality theorem as the basic axioms capture the intended meaning of the various constants more precisely.

B.2.3 Equality

We now consider a theorem characterising when two finite maps are equal. If the axioms above provide a complete characterisation of finite maps then it should be possible to derive such a theorem from them. We begin by considering how to formulate equality. The naive formulation

$$\vdash \forall f\ g.\ (\forall x.\ \mathsf{Apply}\ f\ x\ =\ \mathsf{Apply}\ g\ x)\ =\ (f = g)$$

does not hold because of a problem with the application of finite maps to elements not in their domain. Consider the two finite maps

$$\begin{array}{l} f = \mathsf{Empty} \\ g = \mathsf{Update}\ \ \mathsf{Empty}\ \ (x,\ \mathsf{Apply}\ \mathsf{Empty}\ x) \end{array}$$

where x is some arbitrary value. Then f and g are different finite maps, with different domains; but for any y

$$\begin{aligned}
&\mathsf{Apply}\ g\ y\\
&= \mathsf{Apply}\ (\mathsf{Update}\ \mathsf{Empty}\ (x,\ \mathsf{Apply}\ \mathsf{Empty}\ x))\ y\\
&= \mathsf{Apply}\ \mathsf{Empty}\ y \qquad \text{by a case split on } y = x \text{ and the axioms for } \mathsf{Apply}\\
&= \mathsf{Apply}\ f\ y
\end{aligned}$$

So we have two different finite maps that agree on all elements to which they are applied and hence are equal by the naive formulation of equality. To fix this we modify the characterisation of equality to say that two finite maps are equal only if, in addition to agreeing on all elements, their domains are equal. The following theorems can be proved

$$\vdash \forall f\ g.\ ((\mathsf{Domain}\ f = \mathsf{Domain}\ g)\ \wedge\ (\mathsf{Apply}\ f = \mathsf{Apply}\ g))\ =\ (f = g)$$

$$\begin{aligned}
&\vdash \forall f\ g.\\
&\quad ((\mathsf{Domain}\ f = \mathsf{Domain}\ g)\ \wedge\ (\forall x.\ \mathsf{Domain}\ f\ x\ \supset\ (\mathsf{Apply}\ f = \mathsf{Apply}\ g))\\
&\qquad =\ (f = g)
\end{aligned}$$

B.3 The logical definition of finite maps

B.3.1 Possible representations

Having decided on the characteristic axioms we now must supply a model from which these can be derived. From the view of a programmer the obvious choice is a list of pairs. This has practical merits since lists are well supported in HOL. But we soon run into difficulties if we take this path. Consider the two lists $[(x, int), (x, int)]$ and $[(x, int)]$. These are clearly equal when considered as the finite maps mapping x to int, but are not logically equal lists.

In the theory for HOL-ML [MG94, VG93] this problem was overcome by defining the update function so that only ordered lists with every element appearing at most once in the domain can be constructed. Thus the lists representing two equal finite maps will also be equal. The disadvantage here is that an ordering over the domain of the map is needed, an ordering that should not be needed.

A variation is to define an equivalence relation relating all lists that are equal when considered as finite maps and then define the type of finite maps to be the quotient of lists with this relation. Each element in the defined type will be represented by an equivalence class of lists generated by the defined equivalence relation.

Another possible representation is sets of pairs. This solves some of the problems indicated above but forces us to ensure that no two elements of the set have the same first element. In set theory functions are represented as sets of pairs. This is a good representation of functions in terms of the fundamental object, namely sets. In HOL functions are the fundamental object and therefore sets offer us no advantage. The representation discussed in the next section therefore uses functions instead. This also avoids the need to restrict the theory to some particular set library.

B.3.2 The representation used

The representation used is a function from the type of the domain, α, to the type $\beta+one$. This function maps an element to one if it is not in the domain of the map and to the image of the element if it is in the domain.

What remains is to define a notion of finiteness for functions of this type. A predicate is_fmap can be defined inductively by the following rules:

$$\frac{}{\text{is\_fmap}\ (\lambda a.\ (\text{InR one}))}$$

$$\frac{\text{is\_fmap}\ f}{\text{is\_fmap}\ (\lambda x.\ (x = a) \Rightarrow \text{InL}\ b \mid f\ x)}$$

This gives rise to an induction principle that expresses the finiteness of the functions for which is_fmap holds.

$$\begin{array}{l}
\vdash \forall P. \\
\quad P\ (\lambda x.\ \text{InR one}) \land \\
\quad (\forall f.\ P\ f\ \supset\ (\forall a\, b.\ P(\lambda x.\ (x = a) \Rightarrow (\text{InL}\ b) \mid (f\ x)))) \\
\qquad \supset \\
\quad (\forall f.\ \text{is\_fmap}\ f\ \supset\ P\ f)
\end{array}$$

The type $(\alpha,\beta)fmap$ can then be defined to be the set of functions of type $\alpha \rightarrow (\beta + \text{one})$ for which is_fmap holds. The witness that this new type is non-empty is the function $\lambda x.$ InR one, which represents the empty finite map. A bijection between the the type $(\alpha,\beta)fmap$ and the representation is defined by the functions fmap_ABS, of type $(\alpha \rightarrow (\beta + one)) \rightarrow (\alpha,\beta)fmap$ and fmap_REP, of type $(\alpha,\beta)fmap \rightarrow (\alpha \rightarrow (\beta + one))$. This process of defining a new type and the bijection is described in [GM93].

The constants Empty, Update, Apply, and Domain can be defined in terms of the representation as follows:

$\vdash$ Empty = fmap_ABS ($\lambda a.$ InR one)

$\vdash$ Update f (a,b) = fmap_ABS ($\lambda x.$ $(x = a) \Rightarrow$ InL b | (fmap_REP f) x)

$\vdash$ Apply f x = OutL ((fmap_REP f) x)

$\vdash$ Domain f x = IsL ((fmap_REP f) x)

From these definitions we can derive all the axioms given in section 2. Having done so the representation is not used again; all other theorems in the theory can be proved from just these seven axioms.

The derivation of the characteristic axioms for each constant is straightforward. The axioms can be proved easily at the representation level and then "lifted" to the abstract level. The derivation for the stronger induction principle requires an induction over the size of the domain. This proof involves formalising the concept of the size of the domain. The HOL implementation of these proofs currently employs a set library but this dependency on sets will be removed for the final finite maps library.

B.3.3 Consistence, independence and completeness

The axioms listed in section 2 are consistent because they are derivable from the model just described. Completeness of the axioms with respect to the model can be shown by assuming that the axioms hold and showing that the type specified by the axioms is isomorphic to the model. We show that the function rep with the defining property

$$\vdash \forall f : (\alpha,\beta)\mathit{fmap}.\ \mathsf{rep}\ f \ = \ \lambda x.\,(\mathsf{Domain}\ f\ x \ \Rightarrow\ \mathsf{InL}\ (\mathsf{Apply}\ f\ x) \mid \mathsf{InR}\ \mathsf{one})$$

is a bijection from the type $(\alpha,\beta)\mathit{fmap}$ to the subset of the type $\alpha \to (\beta + \mathit{one})$ satisfying the predicate is_fmap, using only the axioms and not the underlying model. This gives us a means by which to reconstruct the model from the axioms.

This function rep is onto and one to one:

$$\vdash \forall f : \alpha \to (\beta + \mathit{one}).\ \mathsf{is\_fmap}\ f \ \supset\ (\exists g.\ \mathsf{rep}\ g \ = \ f)$$

$$\vdash \forall (f : (\alpha,\beta)\mathit{fmap})\ (g : (\alpha,\beta)\mathit{fmap}).\ (\mathsf{rep}\ f \ = \ \mathsf{rep}\ g) \ \supset\ (f \ = \ g)$$

and its image is contained in the subset of $\alpha \to (\beta + \mathit{one})$ defined by is_fmap:

$$\vdash \forall f : (\alpha,\beta)\mathit{fmap}.\ \mathsf{is\_fmap}\ (\mathsf{rep}\ f)$$

This is in effect an redefinition of the function fmap_REP using only the axioms and not referring to either fmap_ABS or fmap_REP.

We believe that the axioms are also independent but have not attempted a formal proof. That is, we have not shown that a model can be found for each possible set of axioms with one axiom replaced by its negation. While still important for theoretical reasons, this property is not as important in practice; it does not affect either what can be proved or the consistency of the system.

B.4 Enriching the theory

The seven theorems that characterise finite maps are sufficient to build a rich and useful theory. The most important theorems that can be proved are those characterising equality, as discussed above. Many more theorems can also be proved about the basic constants, but this section concentrates on how the theory can be extended with new concepts built up from the seven axioms.

The only method introduced so far for constructing finite maps is Update. In practice, functions are needed to update finite maps by extending them with other finite maps and to allow the domain over which a finite map is defined to be reduced.

The constant Extend is defined so that

$$\mathsf{Apply}\ (\mathsf{Extend}\ f\ g)\ x \ = \ \begin{cases} \mathsf{Apply}\ f\ x & \text{if Domain } f\ x \\ \mathsf{Apply}\ g\ x & \text{otherwise} \end{cases}$$

Formally, the defining property of Extend is:

$$\begin{aligned} &\vdash \forall f\ g. \\ &\quad (\forall x.\ \mathsf{Domain}\ (\mathsf{Extend}\ f\ g)\ x \ = \ \mathsf{Domain}\ f\ x \ \vee\ \mathsf{Domain}\ g\ x) \wedge \\ &\quad (\forall x.\ \mathsf{Apply}\ (\mathsf{Extend}\ f\ g)\ x \ = \ ((\mathsf{Domain}\ f\ x) \Rightarrow (\mathsf{Apply}\ f\ x) \mid (\mathsf{Apply}\ g\ x))) \end{aligned}$$

This definition is made by first proving, by a straightforward induction over f, that a function with this property exists and then using the principle of constant specification to define Extend. More useful theorems about Extend can be derived from this definition. Some examples are

$$\vdash \forall g.\ \mathsf{Extend}\ \mathsf{Empty}\ g \ = \ g$$

$$\vdash \forall f.\ \mathsf{Extend}\ f\ \mathsf{Empty} \ = \ f$$

$$\vdash \forall f\ g\ x\ y.\ \mathsf{Extend}\ (\mathsf{Update}\ f\ (x,y))\ g \ = \ \mathsf{Update}\ (\mathsf{Extend}\ f\ g)\ (x,y)$$

$$\begin{aligned} &\vdash \forall f\ g\ x\ y.\ \mathsf{Extend}\ f\ (\mathsf{Update}\ g\ (x,y)) = \\ &\qquad ((\mathsf{Domain}\ f\ x) \ \Rightarrow\ (\mathsf{Extend}\ f\ g) \mid (\mathsf{Update}\ (\mathsf{Extend}\ f\ g)\ (x,y))) \end{aligned}$$

$$\vdash \forall f\ g\ x.\ \mathsf{Domain}\ (\mathsf{Extend}\ f\ g)\ x \ = \ \mathsf{Domain}\ f\ x \ \vee\ \mathsf{Domain}\ g\ x$$

These results all follow by simple proofs using the basic axioms and the definition of Extend.

All the constants discussed above either increase or preserve the domain of a finite map. A constant DRestrict can be defined which reduces the domain of a finite map to those elements satisfying some predicate. DRestrict is again defined by proving the existence of a function with the appropriate properties and then using constant specification.

The function is characterised by the theorem

$$\begin{array}{l}\vdash \forall f\ p.\\ \quad (\forall x.\ \mathsf{Domain}\ (\mathsf{DRestrict}\ f\ p)\ x\ =\ \mathsf{Domain}\ f\ x\ \wedge\ p\ x)\ \wedge\\ \quad (\forall x.\ \mathsf{Domain}\ f\ x\ \wedge\ p\ x\ \supset\ (\mathsf{Apply}\ (\mathsf{DRestrict}\ f\ p)\ x\ =\ \mathsf{Apply}\ f\ x))\end{array}$$

Some useful properties that can be proved of DRestrict are

$$\vdash \forall p.\ \mathsf{Restrict}\ \mathsf{Empty}\ p\ =\ \mathsf{Empty}$$

$$\begin{array}{l}\vdash \forall f\ p\ a\ b.\\ \quad \mathsf{DRestrict}\ (\mathsf{Update}\ f\ (a, b))\ p\ =\\ \quad\quad ((p\ a)\ \Rightarrow\ (\mathsf{Update}\ (\mathsf{DRestrict}\ f\ p)\ (a, b))\ \mid\ (\mathsf{DRestrict}\ f\ p))\end{array}$$

$$\vdash \forall f\ p\ q.\ \mathsf{DRestrict}\ f\ (\lambda x.\ p\ x\ \vee\ q\ x)\ =\ \mathsf{Extend}\ (\mathsf{DRestrict}\ f\ p)\ (\mathsf{DRestrict}\ f\ q)$$

The proofs of these theorems are again straightforward. The function Delete, to remove a single element from a finite map, can be defined in terms of DRestrict.

A related concept to the domain of a finite map is the range. An element is in the range if there is some element in the domain which is mapped to it. The function Range and RRestrict are defined with the same functionality as Domain and DRestrict but relating to the the range rather the domain.

Another important concept is composition, either of two finite maps or a finite map and a function. Three infix composition functions are defined:

$$\begin{array}{ll}\mathsf{f\_o\_f} & : (\beta, \gamma)\mathit{fmap} \to (\alpha, \beta)\mathit{fmap} \to (\alpha, \gamma)\mathit{fmap}\\ \mathsf{o\_f} & : (\beta \to \gamma) \to (\alpha, \beta)\mathit{fmap} \to (\alpha, \gamma)\mathit{fmap}\\ \mathsf{f\_o} & : (\beta, \gamma)\mathit{fmap} \to (\alpha \to \beta) \to (\alpha, \gamma)\mathit{fmap}\end{array}$$

The notation is designed to show the link with composition of functions

$$\mathsf{o}\ : (\beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma)$$

Two of the other functions defined are Submap, a mapping that is defined on a subset of the domain of another map but maps the elements for which it is defined to the same values and EveryMap, a function that takes a predicate and tests whether it holds of every pair of (domain,range) elements inserted into the finite map.

B.5 Finite maps and recursive types

In this section we consider the problem of defining recursive types that include finite maps. It is a well known difficulty with the HOL system that types such as

$$\mathit{Val} = \mathsf{CONST}\ \mid\ \mathsf{RECORD}\ \mathit{num} \to \mathit{Val} \tag{B.1}$$

may not be introduced into the system easily by automated or manual techniques. This is because the presence of the function *num* → *Val* results in a type that is too large for the trees used in the system's automatic type definition package [Mel89]. The user must therefore carry out a tiresome type construction manually. A possible solution is described in [Gun93].

A related problem is that of defining types of the form

$$Val = \mathsf{CONST} \mid \mathsf{RECORD}\ (num, Val) fmap \tag{B.2}$$

Types such as this are often used to represent expression values in programming language formalisations such as the definition of Standard ML [MTH90]. This section describes how types of this form may be introduced into HOL manually, without needing a solution to the more general problem (B.1) above.

The method we will use is as follows. First, we shall define a mapping between the type $(\alpha,\beta)fmap$ and a subset of the type $(\alpha,\beta)list$. We then show that this mapping gives a unique list for each finite map, which we call its *canonical representation*. Next we manually introduce the concrete type

$$ListVal = \mathsf{List\_CONST} \mid \mathsf{List\_RECORD}\ (num \times ListVal) list \tag{B.3}$$

into HOL. We use a subset of this type as the representation for type (B.2). Finally we develop the necessary theorems which characterise type (B.2) independently of its representation.

B.5.1 Canonical representations for finite maps

We introduce two operators FFst and FRest which decompose a finite map into a single element and a remainder. This does not need an ordering of elements in the finite map, since we appeal to the HOL choice operator.

$$\begin{array}{l} \vdash \forall f. \\ \quad \mathsf{FFst}\ f = (\varepsilon p.\ \mathsf{Domain}\ f\ (\mathsf{FST}\ p) \wedge (\mathsf{SND}\ p = \mathsf{Apply}\ f\ (\mathsf{FST}\ p)))) \end{array}$$

$$\vdash \forall f.\ \mathsf{FRest}\ f = \mathsf{Delete}\ (\mathsf{FST}\ (\mathsf{FFst}\ f))\ f$$

A relation Canon_Rel between finite maps and paired lists can now be defined by primitive recursion on lists:

$$\begin{array}{l} \vdash (\forall f.\ \mathsf{Canon\_Rel}\ f\ [\,] = (f = \mathsf{Empty})) \wedge \\ \quad (\forall f\ h\ t. \\ \quad\ \mathsf{Canon\_Rel}\ f\ (\mathsf{Cons}\ h\ t) = \\ \quad\ \neg(f = \mathsf{Empty}) \wedge (h = \mathsf{FFst}\ f) \wedge \mathsf{Canon\_Rel}\ (\mathsf{FRest}\ f)\ t) \end{array}$$

Intuitively this definition ensures that f is related to l if and only if l has the form [FFst f, FFst (FRest f), FFst (FRest (FRest f)), ...]. Thus precisely one list is defined for each finite map. It is a lengthy process, but fairly straightforward, to prove that Canon_Rel defines a unique list for every map.

Based on Canon_Rel, two functions Canon_of_Fmap and Fmap_of_Canon can then be defined, giving an isomorphism between finite maps and their canonical representations.

$$\begin{array}{l}
\vdash (\forall f.\ \mathsf{Fmap\_of\_Canon}\ (\mathsf{Canon\_of\_Fmap}\ f)\ =\ f) \wedge \\
\quad (\forall l. \\
\quad\ \ \mathsf{Canon\_Rel}\ (\mathsf{Fmap\_of\_Canon}\ l)\ l\ \supset \\
\quad\quad (\mathsf{Canon\_of\_Fmap}\ (\mathsf{Fmap\_of\_Canon}\ l)\ =\ l))
\end{array}$$

B.5.2 Introducing the type *ListVal*

The type

$$\mathit{ListVal} = \mathsf{List\_CONST}\ \mid\ \mathsf{List\_RECORD}\ (\mathit{num} \times \mathit{ListVal})\mathit{list} \tag{B.4}$$

can be introduced manually using a technique similar to that for the type

$$\mathit{data} = \mathsf{List\_CONST}\ \mid\ \mathsf{List\_RECORD}\ (\mathit{data})\mathit{list} \tag{B.5}$$

The manual method for doing this was described on the `info-hol` mailing list [Mel91]. The only real complication is that we are defining a type where a recursive reference to the type occurs nested within a product type on the right-hand side. We derive the following characteristic theorem for the type, which states that a unique function exists for every primitive recursive specification over the type:

$$\begin{array}{l}
\vdash \forall e\ f.\ \exists!\ fn. \\
\quad (fn\ \mathsf{List\_CONST} = e) \wedge \\
\quad (\forall l.\ fn\ (\mathsf{List\_RECORD}\ l) = f\ (\mathsf{MAP}\ (fn \circ \mathsf{SND})\ l)\ l)
\end{array}$$

B.5.3 Introducing the type *Val*

A subset of *ListVal*, isomorphic to our required type *Val*, is defined by a predicate Is_Val introduced by a primitive recursive definition:

$$\begin{array}{l}
\vdash \mathsf{Is\_Val}\ \mathsf{List\_CONST} \wedge \\
\quad (\forall l.\ \mathsf{Is\_Val}\ (\mathsf{List\_RECORD}\ l)\ = \\
\quad\quad (\exists f.\ \mathsf{Canon\_Rel}\ f\ l)\ \wedge\ \mathsf{ALL\_EL}\ (\mathsf{Is\_Val} \circ \mathsf{SND})\ l))
\end{array}$$

where ALL_EL tests if a predicate is true of every element of a list. This definition ensures that each list within a *ListVal* value is a canonical representation of some finite map.

The new type *Val* is now introduced based on the subset defined by Is_Val. The constructors CONST and RECORD are defined in terms of List_CONST and List_RECORD . The characteristic theorem for the type is derived from the characteristic theorem for *ListVal*. The derivation is a lengthy forward proof, and relies on properties of FFst, FRest, EveryMap and the functions Canon_of_FMap and FMap_of_Canon. The resulting theorem is:

$$\begin{aligned}&\vdash \forall e\ f.\ \exists!\ fn.\\ &\quad (fn\ \mathsf{CONST} = e) \land\\ &\quad (\forall fmap.\ fn\ (\mathsf{RECORD}\ fmap) = f\ (fn\ \mathsf{o\_f}\ fmap)\ fmap)\end{aligned}$$

where o_f composes a function with a finite map. This gives a full characterisation of the type *Val*. As with other HOL recursive types, an induction principle for *Val* may be derived from this theorem, along with theorems proving that the constructors CONST and RECORD are one to one and distinct.

B.6 Decision procedures

HOL theories typically come with a set of tools for reasoning about the constructs defined in them. This section discusses some decision procedures for finite maps, and in particular a conversion for determining the result of applying a finite map to an element. The decision procedures for finite maps fall into two categories; those which simplify terms as far as possible and return a single theorem capturing the result of this simplification; and those which will perform case splits and make additional assumptions to return more information. We first discuss an example of the former.

B.6.1 Simplifying terms

Function evaluation conversions can be introduced for all the constants defined in the theory. For example, the conversion Extend_CONV : *conv* simplifies the addition of finite maps as far as possible. For the finite map,

$$\mathsf{Extend}\ \ (\mathsf{Update\ Empty}\ (1,\mathsf{T}))\ \ (\mathsf{Update\ Empty}\ (2,\mathsf{F}))$$

Extend_CONV will return the theorem

$$\begin{aligned}&\vdash \mathsf{Extend}\ \ (\mathsf{Update\ Empty}\ (1,\mathsf{T}))\ \ (\mathsf{Update\ Empty}\ (2,\mathsf{F}))\ =\\ &\quad \mathsf{Update}\ (\mathsf{Update\ Empty}\ (2,\mathsf{F}))\ (1,\mathsf{T})\end{aligned}$$

Note that in this case the conversion has been able to remove the Extend operator completely. The conversion Restrict_CONV : *conv* → *conv* simplifies terms of the form

Restrict f p as far as possible. The first argument must be a conversion to evaluate applications of the restriction function p to individual elements of the domain of f.

B.6.2 A reducer for Apply

The most common decision procedure needed for finite maps is one to determine the result of applying a finite map to an element. A similar conversion computes whether an element is in the domain of a map.

Finite maps are frequently used in formal descriptions of programming languages, and it is common to write *symbolic evaluators* for these languages once they have been embedded into HOL. Such a symbolic evaluator was constructed for the Standard ML Core language in [Sym92]. When writing a symbolic evaluator it is useful to have an application reducer capable of handling applications of finite maps containing HOL variables, as in the case Apply (Update E $(x, 200)$) y. This lookup may arise if the evaluator were reducing a program expression such as "`let x = 200 in y`" in an arbitrary variable environment E. It is useful if the symbolic evaluator can make the necessary assumption that the variable y has some value in this environment, which can later be proved by type inference or some other method. We rarely want the symbolic evaluator to fail just because it cannot determine exactly the result of applying an arbitrary finite map to an element.

The conversion described in this section does not halt when it cannot determine the result of an application. Instead it makes additional assumptions in order to compute a result. These are accumulated in the assumption list of the returned theorem. Sometimes the most important use of these assumptions is to help the users of the conversion find mistakes in their input.

The apply reduction conversion, Apply_CONV, has type *conv* $\rightarrow$ *convl*. The type *convl* is a function from a term to a list of theorems corresponding to the results of evaluation under different assumptions. [1] The first argument should be a conversion that decides equality between members of the domain of the finite map. For example, applying Apply_CONV to num_EQ_CONV and the term

$$\text{Apply (Update Empty } (2, \text{F})) \; x$$

returns the theorems

$$(2 = x) \vdash \text{Apply (Update Empty } (2, \text{F}))x = \text{F}$$

$$(2 \neq x) \vdash \text{Apply (Update Empty } (2, \text{F}))x = \text{Apply Empty } x$$

[1] In the current implementation a lazy list or sequence is used. since the function could potentially return a large list of theorems.

In this example the equality conversion num_EQ_CONV has been unable to determine whether $2 = x$. Thus, Apply_CONV has returned two theorems with different assumptions.

Applying Apply_CONV to num_EQ_CONV and the term

$$\mathsf{Apply}\ (\mathsf{Update}\ (\mathsf{Update}\ (E : (num, bool)fmap)\ (x, y))\ (1, \mathsf{T}))\ 3$$

returns the theorems

$$(x = 3) \vdash \mathsf{Apply}\ (\mathsf{Update}\ (\mathsf{Update}\ E\ (x, y))\ (1, \mathsf{T}))\ 3 \ = y$$
$$(x \neq 3) \vdash \mathsf{Apply}\ (\mathsf{Update}\ (\mathsf{Update}\ E\ (x, y))\ (1, \mathsf{T}))\ 3 \ = \mathsf{Apply}\ E\ 3$$

Here, the arbitrary finite map E extended with an arbitrary pair (x, y) and the pair $(1, \mathsf{T})$ has been applied to 3. Apply_CONV has reduced the application as far as possible, making assumptions about whether or not $x = 3$. In practice Apply_CONV would be used in conjunction with the conversion Domain_CONV.

Another kind of term we would like to be able to reduce are those of the form Apply (Extend f g) x. Adding two variable environments together is common in formal programming language descriptions, and hence the construct Extend f g will often arise. The conversion Apply_CONV is able to reduce applications of this form also, even if the Extend operators are nested arbitrarily.

B.7 An example

Much of the motivation for this work has come from research on embedding the semantics of programming languages in HOL. In this section we give an example of using finite maps to reason about a small language (a simply-typed λ-calculus). Finite maps can be used in two important places here. The first is in the type system, where a finite map can be used to store the context in which a typing judgement holds. Here the finite map used is a mapping from identifiers to types. Strictly speaking, there is no need for the mappings used here to be finite; for our purposes an infinite map would suffice. But there is also no need to allow infinite maps, and the restriction to finite maps provides an induction principle that is useful in proofs.

The second place where finite maps are useful is in the definition of the evaluation relation. Here substitution functions or environments mapping identifiers to expressions can be represented by finite maps.

For the purpose of this example we concentrate on the type system.

B.7.1 A small language

Our aim is to construct a small language that includes variable binding. We define a type, *ty*, to be either an atomic type or a function type:

$$\begin{array}{lll} ty & ::= & \textsf{Atom}\ \mathit{string} \\ & \mid & ty \rightarrow ty \end{array}$$

and an expression, *exp*, to be either an identifier, function abstraction, or function application:

$$\begin{array}{lll} exp & ::= & \textsf{Id}\ \mathit{string} \\ & \mid & \textsf{Lambda}\ \mathit{string}\ ty\ exp \\ & \mid & \textsf{App}\ exp\ exp \end{array}$$

The typing rules are defined as a relation of the form Type $C\ e\ t$ where Type has type *(string,ty)fmap* → *exp* → *ty* → *bool*. This denotes true if the expression e has type t in the context C. The rules for this relation are:

$$\frac{}{\textsf{Type}\ \ (\textsf{Update}\ C\ (v,t))\ \ (\textsf{Id}\ v)\ \ t}$$

$$\frac{\textsf{Type}\ \ (\textsf{Update}\ C\ (y,t_1))\ \ e\ \ t_2}{\textsf{Type}\ \ C\ \ (\textsf{Lambda}\ y\ t_1\ e)\ \ t_1 \rightarrow t_2}$$

$$\frac{\textsf{Type}\ \ C\ \ e_1\ \ (t_1 \rightarrow t_2) \qquad \textsf{Type}\ \ C\ \ e_2\ \ t_1}{\textsf{Type}\ \ C\ \ (\textsf{App}\ e_1\ e_2)\ \ t_2}$$

An important point to note about these rules is that the expression Update $C\ (v,t)$ is used to denote any finite map that maps v to t. The use of this expression does not imply that (v,t) must be the "last update" used to build the type context. This works in our theory because any finite map that maps v to t is equal to a finite map in which the last update was (v,t). This gives another illustration of the usefulness of a theory in which equality does not depend on the order of the updates.

B.7.2 Context extension

This section presents some theorems about how the context of a valid typing judgement can be extended. In semantics this is referred to as weakening the context, as we are adding surplus information.

The first theorem shows that the context can be extended by any mapping from an element not already present in the domain.

$$\begin{array}{l} \vdash \forall C\ e\ t. \\ \quad \mathsf{Type}\ C\ e\ t \supset \\ \quad (\forall x.\ \neg(\mathsf{Domain}\ C\ x) \supset (\forall y.\ \mathsf{Type}(\mathsf{Update}\ C\ (x,y))\ e\ t)) \end{array}$$

This is proved by an induction over the rules for the Type relation. The proof makes use of several of the theorems about finite maps, including the theorem that asserts the equality of maps with two elements inserted in a different order provided they are updating different elements of the domain.

Further theorems, which can be proved by a simple induction over one of the contexts, show under what conditions extending the context with another context will preserve typing judgements. The simplest such theorem is

$$\vdash \forall C\ e\ t.\ \mathsf{Type}\ C\ e\ t \supset \forall C'.\mathsf{Type}\ (\mathsf{Extend}\ C\ C')\ e\ t$$

This says the context can be weakened by extension with any context and the type judgement will still be preserved.

B.7.3 Restriction of the context

An important and practical theorem about this language is that type judgements are preserved by restricting the context to the free variables in the expression being typed. A function Fv can be defined to test if a variable is free in an expression

$$\begin{array}{l} \vdash (\forall i\ x.\ \mathsf{Fv}\ (\mathsf{Id}\ i)\ x \ = \ i = x) \ \wedge \\ \quad (\forall y\ t\ e\ x.\ \mathsf{Fv}\ (\mathsf{Lambda}\ y\ t\ e)\ x \ = \ (y \neq x) \wedge \mathsf{Fv}\ e\ x) \ \wedge \\ \quad (\forall e_1\ e_2\ x.\ \mathsf{Fv}\ (\mathsf{App}\ e_1\ e_2)\ x \ = \ \mathsf{Fv}\ e_1\ x \vee \mathsf{Fv}\ e_2\ x) \end{array}$$

The theorem that can then be proved is

$$\vdash \forall C\ e\ t.\ \mathsf{Type}\ C\ e\ t = \mathsf{Type}\ (\mathsf{DRestrict}\ C\ (\mathsf{Fv}\ e))\ e\ t$$

This theorem follows by using the two lemmas below:

$$\vdash \forall C\ e\ t.\ \mathsf{Type}\ (\mathsf{DRestrict}\ C\ (\mathsf{Fv}\ e))\ e\ t \supset \mathsf{Type}\ C\ e\ t$$

$$\vdash \forall C\ e\ t.\ \mathsf{Type}\ C\ e\ t \supset \mathsf{Type}\ (\mathsf{DRestrict}\ C\ (\mathsf{Fv}\ e))\ e\ t$$

The first of these follows by observing that the expression $\mathsf{Fv}\ e\ x \vee \neg(\mathsf{Fv}\ e\ x)$ is true for any e and x. This is used to show that

$$\begin{array}{l} \mathsf{Type}\ C\ e\ t \\ = \mathsf{Type}\ (\mathsf{DRestrict}\ C\ ((\lambda x.\ (\mathsf{Fv}\ e\ x) \vee ((\lambda y.\ \neg(\mathsf{Fv}\ e\ y))\ x)))\ e\ t \\ = \mathsf{Type}\ (\mathsf{Extend}\ (\mathsf{DRestrict}\ C\ (\mathsf{Fv}\ e)\ (\mathsf{DRestrict}\ C\ (\lambda y.\ \neg(\mathsf{Fv}\ e\ y))\ e\ t \end{array}$$

The result then follows from the theorem

$$\vdash \forall C\ e\ t.\ \mathsf{Type}\ C\ e\ t \supset \forall C'\ \mathsf{Type}\ (\mathsf{Extend}\ C\ C')\ e\ t$$

The other implication requires an induction over the rules for the relation Type. This decomposes the goal into three subgoals, each of which can be solved by manipulation of the contexts similar to that used above.

Both the free variables in a term and the restriction of a context can be computed easily using simple conversions. The last theorem then provides a means to reduce the problem of type checking in a large context.

The importance of this simple example is that much of the task of formalising and proving these results has been removed by the use of the finite maps library and the concepts and theorems developed there.

B.8 Conclusions

Any theory based on a set of axioms must satisfy two essential properties. First, it must be consistent, a fact guaranteed here by the derivation of the axioms from a representation in terms of functions. The axioms should also be complete with respect to the model. This has been proved and so the equivalent of any property provable in the model will be provable from our axioms.

The principal motivation for this work was the need for a practical tool to aid in the development, within HOL, of semantics for programming languages. The theory makes the task of meta-reasoning, such as that in the example, significantly easier than it would be otherwise. Enough theorems have been proved to allow conversions to be written to reduce a variety of expressions involving finite maps to simpler forms. This has practical benefits when building systems like partial evaluators or type checkers using HOL.

One reason that others have used lists to represent finite maps was the ability to define recursive types such as those discussed in section 5. We have shown here that such types can also be defined with the finite maps presented here, although this is not automated. The derivation of this type justifies the axiomatisation of a similar type used in [Sym92].

Acknowledgements

Thanks are due to Tom Melham for advice on all aspects of this work, from the theory to the presentation style. Thanks must also go to the Engineering and Physical Sciences Research Council for financial support of the first author.

Bibliography

[Abr90] Samson Abramsky. The Lazy Lambda Calculus. In David Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

[AC99] Simon A Ambler and Roy L Crole. Mechanised Operational Semantics via (Co)Induction. In Y. Berlot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'99), Nice, France*, volume 1690 of *Lecture Notes in Computer Science*, pages 221–238. Springer, September 1999.

[ACN90] L. Augustsson, Th. Coquand, and B. Nordstrom. A short description of another logical framework. In G. Huet and G. Plotkin, editors, *Preliminary Proceedings of Logical Frameworks*, 1990.

[Acz77] P Aczel. An introduction to inductive definition. In J Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North Holland, 1977.

[Age94] Sten Agerholm. A HOL Basis for Reasoning about Functional Programs. Technical Report RS-94-44, Basic Research in Computer Science, University of Aarhus, December 1994.

[AGMT98] J. S. Aitken, P. Gray, T. Melham, and M. Thomas. Interactive theorem proving: An empirical study of user activity. *Journal of Symbolic Computation*, 1998.

[AJM94] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF (extended abstract). In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software. International Symposium TACS'94*, number 789 in Lecture Notes in Computer Science, pages 1–15, Sendai, Japan, April 1994. Springer-Verlag.

[BGG+92] Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Theorem Provers in Circuit Design: Theory, Practice and Experience: Proceedings of the IFIP WG10.2 International Conference, Nijmegen*, pages 129–156. North-Holland, June 1992.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall Press, 2nd edition, 1998.

[Bou97] R. J. Boulton. A tool to support formal reasoning about computer languages. In E. Brinksma, editor, *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 81–95, Enschede, The Netherlands, April 1997. Springer.

[Bou98] R. J. Boulton. Generating embeddings from denotational descriptions. In J. Grundy and M. Newey, editors, *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'98)*, volume 1479 of *Lecture Notes in Computer Science*, pages 67–86, Canberra, Australia, September/October 1998. Springer.

[BRTT93] Lars Birkdal, Nick Rothwell, Mads Tofte, and David N. Turner. The ML Kit. Technical Report 93/14, Department of Computer Science, University of Copenhagen, March 1993.

[BSvH+93] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993.

[Bun88] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* International Series in Computer Science. Prentice Hall, 1988.

[CAB+86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knobloch, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the NuPRL Proof Development System.* Prentice-Hall, 1986.

[CG94] Graham Collins and Stephen Gilmore. Supporting Formal Reasoning about Standard ML. Technical Report ECS-LFCS-94-310, Laboratory for Foundations of Computer Science, University of Edinburgh, November 1994.

[CH97] Graham Collins and Jonathan Hogg. The circuit that was too lazy to fail. Draft Report (http://www.collins-peak.net/academic.html), 1997.

[CO92] A. Cant and M.A. Ozols. A verification environment for ML programs. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applicatins*, San Francisco, California, June 1992.

[Col94] Graham Collins. Supporting Formal Reasoning about Standard ML. MSc dissertation, University of Edinburgh, September 1994.

[Col96a] Graham Collins. A Proof Tool for Reasoning about Functional Programs. In J. von Wright, J. Grundy, and J Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 109–124. Springer-Verlag, 1996.

[Col96b] Graham Collins. Supporting Reasoning about Functional Programs : An Operational Approach. In *1995 Glasgow Workshop on Functional Programming*, Electroninc Workshops in Computer Science. Springer-Verlag, 1996.

[CS95] Graham Collins and Donald Syme. A Theory of Finite Maps. In E. Thomas Schubert, Phillip J. Windley, and Hames Alves-Foss, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 122–137. Springer-Verlag, 1995.

[CZ94] J. Camilleri and V. Zammit. Symbolic animation as a proof tool. In T. F. Melham and J. Camilleri, editors, *Higher Order Logic Theorem Proving and Its Applications: 7th International Workshop*, volume 859, pages 113–127. Springer-Verlag, sept 1994.

[Den99] Louise A Dennis. *Proof Planning Coinduction*. PhD dissertation, University of Edinburgh, 1999.

[DFH$^+$93] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The Coq Proof Assistant User's Guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.

[DG97] A. Dennis, L.A. Bundy and I Green. Using a generalisation critic to find bisimulations for coinductive proofs. In *CADE-14*, pages 276–290, 1997. Also available as Edinburgh DAI Research Report, 834.

[FT96] Tom Fukushima and Charles Tuckey. *Charity User Manual*, January 1996. (draft, `http://www.cpsc.ucalgary.ca/projects/charity/home.html`).

[GG89] Stephen J. Garland and John V. Guttag. An Overview of LP, The Larch Prover. In Nachum Dershowitz, editor, *Rewriting Techniques and Applications*, volume 355 of *LNCS*, pages 137–155. Springer-Verlag, 1989.

[Gil96] Andy Gill. A Graphical User Interface for an Equational Reasoning Assistant. In *1995 Glasgow Workshop on Functional Programming*, Electroninc Workshops in Computer Science. Springer-Verlag, 1996.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[GM96] A. D. Gordon and T. Melham. Five Axioms of Alpha-Conversion. In J. von Wright, J. Grundy, and J Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 173–190. Springer-Verlag, 1996.

[GMW79] M.J. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF : A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, 1979.

[Gor93a] Andrew D. Gordon. An Operational Semantics for I/O in a Lazy Functional Language. In *Conference on Functional Programming Languages and Computer Architecture, Copenhagen*, pages 136–145. ACM Press, June 1993.

[Gor93b] Andrew D. Gordon. Functional programming and input/output. Technical Report 285, University of Cambridge Computer Laboratory, February 1993.

[Gor94] Andrew D. Gordon. *Functional Programming and Input/Output.* Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[Gor95a] Andrew D. Gordon. Bisimilarity as a Theory of Functional Programming. Technical Report NS-95-3, Basic Research in Computer Science, University of Aarhus, July 1995.

[Gor95b] Andrew D. Gordon. A Tutorial on Co-induction and Functional Programming. In *1994 Glasgow Workshop on Functional Programming*, Workshops in Computer Science, pages 78–95. Springer-Verlag, 1995.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing. The MIT Press, 1992.

[Gun93] Elsa Gunter. A Broader Class of Trees for Recursive Type Definitions for HOL. In J. J. Joyce and C. J. H. Seger, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 141–154. Springer-Verlag, 1993.

[H+92] Paul Hudak et al. Report on the functional programming language Haskell, version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[Har95] John Harrison. Inductive definitions: automation and application. In E. Thomas Schubert, Phillip J. Windley, and Hames Alves-Foss, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 200–213. Springer-Verlag, 1995.

[HM94] Peter V. Homier and David F. Martin. Trustworthy Tools for Trustworthy Programs: A Verified Verification Condition Generator. In Thomas F. Melham and Juanito Camilleri, editors, *Theorem Proving in Higher Order Logics*, volume 859 of *Lecture Notes in Computer Science*, pages 269–284. Springer-Verlag, September 1994.

[How89] Douglas J. Howe. Equality in lazy computation systems. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, pages 198–203, 1989.

[Hut98] Graham Hutton. Theorems for free. In *3rd ACM SIGPLAN International Conference on Functional Programming.* ACM, September 1998.

[Jon96] Simon Peyton Jones. Compiling haskell by program transformation: a report from the trenches. In *Proceedings of the European Symposium on Programming (ESOP'96), Linkping, Sweden*, volume 1058 of *Lecture Notes in Computer Science.* Springer Verlag, January 1996.

[JR97] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.

[LP92] Z. Luo and R. Pollack. LEGO Proof Development System: User's Manual. Technical Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh, November 1992.

[Luo94] Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Number 11 in International Series of Monographs on Computer Science. Oxford University Press, 1994.

[McC98] Guy McCusker. *Games and Full Abstraction for a Functional Metalanguage with Recursive Types*. Distinguished Dissertations. Springer-Verlag, 1998.

[Mel89] Tom F. Melham. Automating Recursive Type Definitions in HOL. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer-Verlag, 1989.

[Mel91] Tom F. Melham. Recursive Data Types. Message on *info-hol* mailing list, 9th November 1991.

[Mel92] Tom F. Melham. A Package for Inductive Relation Definitions in HOL. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, Davis, August 1992*, pages 350–357. IEEE Computer Society Press, 1992.

[Mel94] Tom F. Melham. A mechanized theory of the π-calculus in HOL. *Nordic Journal of Computing*, 1:50–76, 1994.

[MG94] Savi Maharaj and Elsa Gunter. Studying the ML Module System in HOL. In Tom Melham and Juanito Camilleri, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 859 of *Lecture Notes in Computer Science*, pages 346–361. Springer-Verlag, September 1994.

[Mil89] Robin Milner. *Communication and concurrency*. International Series in Computer Science. Prentice Hall, New York, 1989.

[MPW92a] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I. *Information and Computation*, 100(1):1–40, September 1992.

[MPW92b] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, II. *Information and Computation*, 100(1):41–77, September 1992.

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML*. The MIT Press, 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[Nes92] M. Nesi. A Formalization of the Process Algebra CCS in Higher Order Logic. Technical Report 278, Computer Laboratory, University of Cambridge, December 1992.

[NPS90] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. International Series of Monographs on Computer Science 7. Oxford University Press, 1990.

[Pau83] Lawrence Paulson. A Higher Order Implementaion of Rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[Pau87] L. C. Paulson. *Logic and computation*. Cambridge University Press, 1987.

[Pau94] Lawrence C. Paulson. Co-induction and co-recursion in higher order logic. Technical report, University of Cambridge Computer Laboratory, 1994.

[Pau96] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

[Pit94] A. M. Pitts. A co-induction principle for recursively defined domains. *Theoretical Computer Science*, 124:195–219, 1994. (A preliminary version of this work appeared as Cambridge Univ. Computer Laboratory Tech. Rept. No. 252, April 1992.).

[Pit96] A. M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996. (A preliminary version of this work appeared as Cambridge Univ. Computer Laboratory Tech. Rept. No. 321, December 1993.).

[Pit97] A. M. Pitts. Operationally Based Theories of Program Equivalence. In D. Dybjer and A. M. Pitts, editors, *Semantics and Logic of Computation*. 1997.

[Pit98] A. M. Pitts. Parametric polymorphism and operational equivalence (preliminary version). *Electronic Notes in Theoretical Computer Science*, 10,

1998. Proceedings, 2nd Workshop on Higher Order Operational Techniques in Semantics, Stanford CA, December 1997.

[Plo91] Gordon D Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN - 19, Aarhus University, September 1981, Reprinted April 1991.

[Reg95] Franz Regensburger. HOLCF : Higher Order Logic of Computable Functions. In E. Thomas Schubert, Phillip J. Windley, and Hames Alves-Foss, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 293–307. Springer-Verlag, 1995.

[SBB98] Konrad Slind, Mike Gordonand Richard Boulton, and Alan Bundy. System description: An interface between CLAM and HOL. In C. Kirchner and H. Kirchner, editors, *Proceedings of the Fifteenth International Conference on Automated Deduction (CADE-15), Lindau, Germany*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 134–138. Springer, July 1998.

[SGBB98] Konrad Slind, Mike Gordon, Richard Boulton, and Alan Bundy. An interface between CLAM and HOL. In J. Grundy and M. Newey, editors, *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'98), Canberra, Australia*, volume 1479 of *Lecture Notes in Computer Science*, pages 87–104. Springer, September/October 1998.

[SJ90] Mary Sheeran and Geraint Jones. Circuit design in Ruby. North Holland, 1990.

[Sli96] Konrad Slind. Function definition in higher-order logic. In J. von Wright, J. Grundy, and J Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 381–397. Springer-Verlag, 1996.

[Sym92] Donald Syme. Supporting Formal Reasoning about Standard ML. Honours Thesis, Australian National University, 1992.

[Sym93] Donald Syme. Reasoning with the Formal Definition of Standard ML in HOL. In *Higher Order Logic Theorem Proving and Its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 43–60. Springer-Verlag, 1993.

[Sym97] Don Syme. Declare: A prototype declarative proof system for higher order logic. Technical Report 416, University of Cambridge Computer Laboratory, feb 1997.

[Sym98] Donald Syme. *Declarative Theorem Proving for Operational Semantics.* PhD dissertation, University of Campridge, 1998.

[Tho89] Simon J. Thompson. A logic for Miranda. *Formal Aspects of Computing*, 1, 1989.

[Tho91] Simon Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.

[Tho93] Simon J. Thompson. Formulating Haskell. In *Workshop on Functional Programming, Ayr, 1992*, Workshops in Computing. Springer-Verlag, 1993.

[Tho94] Simon J. Thompson. A logic for Miranda, revisited. Revised version of the 1989 article, 1994.

[Tho96] Simon Thompson. *Haskell: The Craft of Functional Programming.* Addison-Wesley, 1996.

[Tur95] David A Turner. Elementary strong functional programming. In *Functional Programming Languages in Education*, volume 1022 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, December 1995.

[Van94] Myra VanInwegen. The formal specification of programming languages. PhD Thesis Proposal, University of Pennsylvania, September 1994.

[VG93] Myra VanInwegen and Elsa Gunter. HOL-ML. In J. J. Joyce and C. J. H. Seger, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 61–74. Springer-Verlag, 1993.

[Wad89] Philip Wadler. Theorems for free. In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages.* The MIT Press, 1993.