ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

David N. Turner (Ed)

Functional Programming, Glasgow 1995

Proceedings of the 1995 Glasgow Workshop on Functional
Programming,
Ullapool, Scotland, 10-12 July 1995

Paper:

Supporting Reasoning about Functional Programs: An Operational Approach

Graham Collins

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

Supporting Reasoning about Functional Programs : An Operational Approach

Graham Collins

Department of Computing Science, University of Glasgow,
Glasgow, Scotland, G12 8QQ.
grmc@dcs.gla.ac.uk

Abstract

Some existing systems for supporting reasoning about functional programs have been constructed without first formalising the semantics of the language. This paper discusses how a reasoning system can be built, within the HOL theorem proving environment, based on an operational semantics for the language and using a fully definitional approach. The theoretical structure of the system is based on work by Andrew Gordon, where applicative bisimulation is used to define program equivalence. We discuss how this theory can be embedded in HOL and the type of tools which can be built on top of this theoretical framework to make reasoning possible in practice.

Introduction

It is often stated that one advantage of functional programming languages, in particular pure functional languages, is that they are suitable for formal reasoning. While such reasoning is indeed carried out by some, there is a need for tools to make formal reasoning more practical and usable by a wider community. A machine based tool can be used as a proof checker, ensuring that all steps in a proof are valid, or it can automate some steps in the proof. This may involve automating only trivial steps that would be ignored when carrying out a proof on paper or some automation of more significant proof steps.

One style used for reasoning about functional programs is equational reasoning. Typically a theorem of the form $a = b$ is proved by rewriting one or both sides of the equality with some algebraic identities until both sides are identical. Some systems to support such reasoning have been built without first formalising the semantics of the language [6, 15, 16]. Each function is defined by simple algebraic identities that are introduced as axioms.

Equational reasoning alone is not sufficient for any but the simplest of proofs. A range of other proof methods is required. For finite data and functions with finite behaviour, induction provides the common proof method. With lazy languages many programs exhibit infinite behaviour and a proof method such as co-induction is required. Such proofs method are not yet widely understood and theorem proving support can help introduce them to a wider community and give greater confidence to a programmer using an unfamiliar method.

The first stage in building such a proof system is to formalise and embed the semantics of a language within the theorem prover. Some of the methods by which this can be done are discussed in section 2 and an actual embedding is described in section 3.

A definition of equality between programs is more difficult. One approach is bisimulation. In section 4 we discuss some of the difficulties and give details of the construction within HOL of a theory of program equivalence for a small language. This is based on the semantics defined in section 3.

The language discussed here is PCF plus streams. The exact formulation of the syntax and semantics and the definition of program equality is taken from work by Andrew Gordon that provides a rigorous development, on paper, of a theory for a number of languages [7]. PCF plus streams is the simplest and the one chosen here to test how easily and effectively such an operational theory can be embedded in a theorem prover. This paper does not attempt to provide all the details of the system developed but concentrates on how a system can be structured.

1 The HOL theorem proving system

HOL is a theorem proving environment for classical higher order logic [8]. 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, 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 subset of an existing type, and prove the existence of a function with some chosen functionality before introducing such a function. This is guaranteed to preserve the consistency of the system, but leads to complex definitions. Packages are provided to perform the definition automatically from natural specifications of some important classes of types and functions.

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 results $\Gamma_1 \vdash t_1 \supset t_2$ and $\Gamma_2 \vdash t_1$ and yields the result $\Gamma_1 \cup \Gamma_2 \vdash t_2$ where Γ_1 and Γ_2 are the assumptions of the theorems.

Goal directed proof is supported by the subgoal package, which allows the goal to be repeatedly decomposed into subgoals which can eventually be proved. The current goal is a term together with a list of terms representing the assumptions that can be made when proving the goal. These assumptions may arise from manipulating known theorems or from an induction hypothesis. This decomposition of a goal is usually performed by tactics, functions which 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 modified 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 some proof search. A proof in HOL is an ML program. This is 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.

2 Semantics and meta-theoretic reasoning

Several researchers have investigated how to formalise the semantics of functional programming languages and embed the semantics in a theorem prover. The intention has usually been to prove meta-theoretic results about the language rather than developing a system to reason about programs.

2.1 Formalising the semantics

Two approaches to formalising the semantics for functional programming languages are to use either an operational or denotational semantics. Most of the literature about the semantics of specific languages uses some form of operational semantics. The most significant piece of work in this field is the Definition of Standard ML [12].

In the operational approach a static semantics, which deals with how to assign types to expressions, and a dynamic semantics, which defines what each expression evaluates to, are defined. This is done using a family of relations, one for each syntactic class in the language. The semantics then provides a series of rules specifying how these relations can be evaluated.

The approaches to formalising the static semantics are all relatively similar. Some form of type context or environment is used to map identifiers to their types, and a relation of the form $\text{Type } C \ e \ t$ is defined so as to be true if and only if the expression e can be assigned the type t in a context C .

A more varied approach is taken to the formalisation of the dynamic semantics. Some differences centre around how variable binding is formalised. This can be done by storing the mapping from variables to the expression or value they are bound to in an environment, or by substituting the value to which the variable is bound throughout the expression.

2.2 Embedding in theorem provers

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

A conditional operator would be represented as a function in the logic of type $bool \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$ that evaluates its first argument and returns the value of its second or third argument if the first argument evaluates to 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 [3].

A second method is to represent the syntax of the language by the values of one or more new types in the logic. Typically each type of 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 embedded types. This is referred to as a deep embedding [3].

One 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 [12] 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 investigated in the HOL-ML project [10, 17]. Here the dynamic semantics of the language is investigated, including the imperative features and the module system. The project is a deep embedding of the Definition of Standard ML and the HOL theorem prover is used. 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 [1, 14]. The LCF theorem provers [13] 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 like Higher Order Logic.

3 An embedding for PCF plus streams

In the work described here we seek not only to embed the semantics but to define an equivalence relation over programs. An equivalence relation is not defined in any of the systems based on an operational semantics in the previous section. It is not even clear how to define such a relation for Standard ML with state. For pure functional languages, such as the language to be discussed here, there are ways to define an equivalence relation from either an operational or denotational semantics.

The theory we describe below has a number of layers, each of which is a formalised theory in HOL. The first three, syntax, static semantics and dynamic semantics, along with the associated meta-theory, are similar to the work described in the previous section. The next layer, the definition of an equivalence relation, completes the theory underlying the reasoning system. Before explaining how these layers are defined we briefly introduce PCF plus streams.

3.1 Call by name PCF plus streams

The syntax of this language is given in figure 1. It is identical to the first language introduced in Andrew Gordon's report [7]. The important features include function abstraction, recursive functions, and lists. The inclusion of some form of lists allows induction and co-induction over lists to be investigated. A form of case analysis is provided for lists, which allows a simple variation on pattern matching over lists. The case analysis function takes a list on which to perform the case analysis, the list to be returned if the first list is empty, and a function which can be applied to the head and tail of the list if it is not empty. The syntax given in figure 1 is not exactly the syntax which is embedded in HOL where a slightly more abstract syntax is used. In particular, function application cannot be represented by juxtaposition

$ty ::=$	Num	$exp ::=$	n	(Natural number)
	Bool		true	
	List ty		false	
	$ty_1 \rightarrow ty_2$		nil ty	(Empty list of type t)
			cons $exp_1\ exp_2$	
			x	(Identifier)
			succ exp	
			pred exp	
			iszero exp	
			if exp_1 then exp_2 else exp_3	
			lambda $x\ ty\ exp$	
			$exp_1\ exp_2$	(function application)
			rec $x\ ty\ exp$	
			scase $exp_1\ exp_2\ exp_3$	

Figure 1: The syntax of PCF plus streams

as this is used for function application in the HOL logic. For the sake of clarity we use the concrete syntax shown here throughout the rest of this paper.

This small language omits many important features including polymorphism, pattern matching, type constructors, and user defined types. The absence of polymorphism means that functions such as `map` must be represented by a family of functions indexed by the type of the arguments they can be applied to.

3.2 Syntax

The abstract syntax is represented by values of some defined types in the logic. As with all new types in HOL the abstract syntax must be defined as subsets of existing types. The introduction of such syntactic objects into HOL is described in [11]. This takes a simple specification of the required type and performs all the inference necessary to define that type.

For PCF plus streams, two new types, ty and exp , are introduced. Meta-level functions can be defined over these types. For example we introduce a function, `FV`, to determine the set of identifiers in an expression that are not bound by some lambda abstraction. This is a simple recursive function that can be defined using a tool built into HOL from a simple specification.

3.3 Semantics

The next layer of the system is the operational semantics of the language. The static semantics are formalised by a relation

$$\text{Type} : \text{context} \rightarrow \text{exp} \rightarrow \text{ty} \rightarrow \text{bool}$$

An example of one of the defining rules for this relation is the rule for application:

$$\frac{\text{Type } C\ e_1\ (t_1 \rightarrow t_2) \quad \text{Type } C\ e_2\ t_1}{\text{Type } C\ (e_1\ e_2)\ t_2}$$

The only difficulty in defining the `Type` relation is in defining the notion of the type context, which is a mapping between identifier and types. This is formalised using finite maps, which have not been adequately defined in HOL before. This is described elsewhere [5].

Typically we only consider well typed programs so it is useful to introduce a new relation, `Prog`, which holds of an expression e and type t only if e has type t in the empty context. This is trivial to define in terms of `Type`. We use this

relation to perform a form of “sub-typing”. Often we will want to say that a property P holds for all well-typed terms. In the absence of a type of “well-typed” terms we can instead prove the theorem $\forall x. (\exists t. \text{Prog } t \ x) \supset P \ x$.

The dynamic semantics are formalised with a relation between syntactic objects. Before this relation can be introduced it is necessary to formalise substitution. A substitution function, **Sub**, is defined in such a way that it deals correctly with variable capture. If we consider substituting only well-typed terms then variable capture can be ignored and simpler properties of the substitution function can be used in proofs.

With substitution defined, a relation $\longrightarrow : \text{exp} \rightarrow \text{exp} \rightarrow \text{bool}$ between well typed expressions can be introduced. The proposition $e_1 \longrightarrow e_2$ states that under the rules defining the relation $\longrightarrow$ e_1 can reduce to e_2 . This reduction relation is a small step reduction. It may be possible to perform a series of reductions of an expression. This series may be infinite.

Two examples of the defining rules for this relation are.

$$\frac{e_1 \longrightarrow e_3}{(e_1 \ e_2) \longrightarrow (e_3 \ e_2)} \quad (1)$$

$$(\text{lambda } y \ t \ e) \ e_1 \longrightarrow (\text{Sub } e \ (y, \ e_1)) \quad (2)$$

An alternative large step evaluation relation, **Eval**, can be defined in terms of $\longrightarrow$. This is closer to the style of relation used to define the semantics of Standard ML [12]. This can be thought of as a specification of an interpreter for the language and is important for efficient mechanisation as discussed later. The rule for function application is:

$$\frac{\text{Eval } e_1 \ (\text{lambda } y \ t \ e) \quad \text{Eval } (\text{Sub } e \ (y, \ e_2)) \ c}{\text{Eval } (e_1 \ e_2) \ c}$$

This rule is obtained by repeated applications of rule 1 above until the function being applied is reduced to a lambda abstraction followed by a single application of rule 2.

A number of meta-theoretic results can be proved about the semantics. These include the fact that reduction and evaluation are deterministic. For the reduction relation described above, $e_1 \longrightarrow e_2$ states only that e_1 can reduce to e_2 and does not state that e_2 is the only value to which e_1 reduces. The required result can be written

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

One fundamental result, essential for reasoning about programs, is that reduction and evaluation preserve types. The result for reduction is:

$$\vdash \forall e_1 \ e_2. e_1 \longrightarrow e_2 \supset (\forall t. \text{Prog } t \ e_1 \supset \text{Prog } t \ e_2)$$

Other important meta-theoretic results include the induction theorem for lists.

4 Defining an equivalence relation

The reduction and evaluation relations are important for the development of a usable system. They give a strong semantic relation between terms. They are not, however, adequate for all purposes as they do not permit equational reasoning. In order to develop an equational reasoning system we also require a congruence relation which captures the meaning of equality of programs. In this section we develop a theory of program equivalence based on applicative bisimulation [7].

4.1 Co-induction

Applicative bisimulation is defined co-inductively. The definition depends on two concepts: Monotonic functions and F -Dense sets. A function F is monotonic if

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

and a set X is F -Dense if for the function F

$$X \subseteq F(X)$$

The greatest fixpoint of 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 then:

$$X \subseteq F(X) \supset X \subseteq \text{gfp } F$$

for any X where $\text{gfp } F$ is the greatest fixpoint of F . For the definition of equivalence of programs we find some monotonic function $F_{==}$ capturing the meaning of the equivalence and define $==$ to be $\text{gfp } F_{==}$. To prove that $x == y$ by co-induction we need to find some relation X such that $(x, y) \in X$ and show that X is $F_{==}$ -Dense.

A second principle of co-induction, sometimes referred to as strong co-induction [7], can be derived from co-induction:

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

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

Co-inductive definitions have not yet been implemented in HOL. Relations in HOL are normally represented by functions rather than sets. For the definition of applicative bisimulation only binary relations are required and so this is all that has been implemented. A more general co-inductive definition package could be developed. With this approach a binary relation is represented by a function of type $\alpha \rightarrow \alpha \rightarrow \text{bool}$ rather than a set of pairs. As an example the definition of the predicate formalising F -Dense relations is

$$\text{Dense } F \ X = (\forall a \ b. X \ a \ b \supset (F \ X) \ a \ b)$$

The fixpoint of a function F is defined not as the union of sets but by the definition

$$(\text{gfp } F) \ a \ b = \exists X. \text{Dense } F \ X \wedge X \ a \ b$$

This is the equivalent definition in our formalisation using functions.

In order to define applicative bisimulation co-inductively, as the greatest fixpoint of some function $F_{==}$, we must first formulate the function $F_{==}$ which captures the meaning of applicative bisimulation. The definition of this function is based on a labelled transition system for the language.

4.2 A labelled transition system

The labelled transition system captures the idea of observable properties of programs. It is defined by new relation

$$\text{LTS} : \text{exp} \rightarrow \text{exp} \rightarrow \text{act} \rightarrow \text{bool}$$

where act is the type of possible labels. $\text{LTS } e_1 \ e_2 \ a$ means that under the rules for **LTS** the expression e_1 can make a transition to e_2 with label a .

The rules capture the idea that, for a function, you can observe what the function is applied to and then observe the behaviour of the resulting program, and for any other type you can observe some facts about the value of the program if it has one. The rules for **cons** are:

$$\frac{}{\text{LTS } (\text{cons } a \ b) \ a \ (\text{Hd})} \text{Prog } t \ a \wedge \text{Prog } (\text{List } t) \ b$$

$$\frac{}{\text{LTS } (\text{cons } a \ b) \ b \ (\text{Tl})} \text{Prog } t \ a \wedge \text{Prog } (\text{List } t) \ b$$

These rules give the possible transitions for a program of list type with a cons cell as the outermost constructor. It is possible to carry out a transition to either the head or tail of the list. These transitions have labels **Hd** and **Tl**.

4.3 Applicative bisimulation

Informally, two programs e_1 and e_2 are bisimilar if they can make the same observable transitions to terms which are also bisimilar. More formally, bisimulation, $=$, is defined as the fixed point of a function $F_{=}$ that is defined to have the property

$$\begin{aligned} \forall S \ a \ b. (F_{=} \ S) \ a \ b = & \\ & (\exists t. \text{Prog } t \ a \ \wedge \ \text{Prog } t \ b) \wedge \\ & (\forall a' \ act. \text{LTS } a \ a' \ act \supset (\exists b'. (\text{LTS } b \ b' \ act) \wedge S \ a' \ b')) \wedge \\ & (\forall b' \ act. \text{LTS } b \ b' \ act \supset (\exists a'. (\text{LTS } a \ a' \ act) \wedge S \ a' \ b')) \end{aligned}$$

The first line of the definition has the effect of restricting equivalence to talk about well typed terms only.

The relation $=$ is defined to be $\text{gfp } F_{=}$. It is an equivalence relation and can also be proved to be a congruence using Howe's method [9]. These results are sufficient to develop an equational reasoning system for the language within HOL.

Because the relation is co-inductively defined it also allows the possibility of co-inductive proof. An example is given at the end of this paper.

5 Practical support for equational reasoning

The work described in the previous section establishes the theoretical foundations for a system to support reasoning about PCF programs. This section deals with the practical aspects of reasoning about actual programs.

There are practical difficulties with the work discussed in the previous section. 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 goal-directed proof. However, the number of rules to be applied may be very large and so applying all the rules by hand may not be practical. 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.

The solution is to take advantage of the fact that the meta-language for HOL, Standard ML, is a full programming language. This can be used to write higher level proof functions which can be applied to perform many proof steps at once.

The simplest such functions are tactics which perform some simple manipulation of goals before and after the application of theorems. One example is induction tactics which manipulate the goal into the form required to apply the induction theorem, use the theorem and then simplify the goal into the base and step cases.

In this section we concentrate on a more substantial piece of automation. Many of the small steps in a proof will arise from calculating the type or value of a program. It is essential and possible to automate these proof steps.

5.1 Interpreter

The **Type** and **Eval** relations can be thought of as specifications of how to type or evaluate expressions on an abstract machine. It is of course possible to write an interpreter which implements this specification. Standard ML provides a programming language in which to write such an interpreter. For any expression e the interpreter can give us the expression e' to which e evaluates.

Both the interpreter and the relation **Eval** are deterministic so the way in which the interpreter evaluates e to e' will correspond exactly to way in which the rules need to be applied to prove $\text{Eval } e \ e'$. The interpreter can therefore be used to both evaluate e and return the instructions for how to prove $\text{Eval } e \ e'$. Similar comments apply for the **Type** relation. 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.

In the work developed here, although HOL and the interpreter are both implemented in Standard ML, they are treated as separate systems with an interface between them. We provide a translation from the HOL types for the syntax of expression to the ML types used in the interpreter. This allows the interpreter to be developed and tested separately from rest of the system. The interpreter can be extended to behave as a symbolic evaluator to aid in more general

proofs. The translation between HOL types and ML types is not a 1-1 translation. The ML types contain additional type constructors. For example we represent HOL variables and HOL constants with separate type constructors and change the substitution function to a type constructor. This allows us to manipulate these aspects of a term more easily.

The use of an SML interpreter, the ML Kit [2], with HOL-ML [17] to perform a similar task is discussed elsewhere [4]. This work did not develop the idea of symbolic evaluation.

The general technique of developing an external system to find proofs and then using a theorem prover such as HOL to check the proof then store and manipulate the results has not been widely investigated before. One advantage is that checking a proof is much more efficient than searching for a proof. The resulting system is still consistent. If the interpreter is not correct then an incorrect proof will fail when checked.

5.2 Generation of proof

Given the ML types representing the abstract syntax it is a relatively easy task to construct an evaluator and type checker for the language. This just involves writing a recursive function with a clause for each of the rules for the relations `Type` and `Eval`.

A basic interpreter would only return the result of evaluating the expression. The interpreter described here returns the derivation tree instead of just the expression. This tree has a node for each stage in the evaluation. Each node is equivalent to some rule in the dynamic semantics and contains the result of the evaluation at that point plus the derivation tree for each subexpression which must be evaluated.

The type of this extended interpreter is `Exp -> Exp Result` where the type `Exp Result` is the derivation tree.

This derivation tree contains all the information required to prove what an expression evaluates to. The method is to first convert the expression to the equivalent expression in the ML syntax. This is then evaluated to return the derivation tree. The derivation tree is used to structure a forward proof that `Eval e e'` where `e'` is the result of evaluating `e`.

The advantage of this approach is that there is no need for any proof search. The evaluator can be an efficient functional program which will calculate the result and derivation tree quickly. This is then converted to a proof in HOL.

5.3 HOL variables and constants

The evaluator can be extended to handle HOL variables and constants. This extension involves making additional assumptions which may be discharged later. When evaluating a term the evaluator described above must check that certain subterms can be evaluated or are well typed. If the evaluator encounters a HOL variable representing an expression then there is no way to check this from the structure of the term alone. The evaluator simply adds the assumption, returning a theorem of the form

$$A_1, A_2, \dots, A_n \vdash \text{Eval } e_1 e_2$$

where $A_1, A_2, \dots, A_n$ are the assumptions made to prove `Eval e1 e2`. This ability to make assumptions is especially useful when carrying out proofs by induction where the step case can be reduced to induction hypothesis by the evaluator or at least part of this reduction will be automated.

In many cases the assumptions can be automatically discharged by higher level proof tools

6 Example

The purpose of this example is to show that a simple proof, already worked out formally on paper [7], can be replayed through the system described here with the potentially large number of trivial proof steps that are omitted on paper being handled by the automatic proof tools described in the previous section.

The example uses two functions, `map` and `iterate`. These functions are introduced by the definitions in figure 2. While these definitions look complex they could be generated from a simpler syntax using pattern matching. The definitions give families of functions indexed by the types of arguments to the functions. Any instance of these functions will not be polymorphic.

```

iterate (t : ty) =
  rec iterate ((t → t) → t → List t)
    (lambda f (t → t)
      (lambda x t
        (cons x (iterate f (f x))))))

map (t : ty) (t1 : ty) =
  rec map ((t → t1) → List t → List t1)
    lambda f (t → t1)
    lambda x (List t)
    scase x
      nilt1
      (lambda hd t
        lambda tl (List t)
          (cons (f hd) (map f tl)))
    
```

 Figure 2: The definitions of `iterate` and `map`

The types of these functions can be derived automatically to get the theorems

$$\vdash \forall t. \text{Prog } ((t \rightarrow t) \rightarrow t \rightarrow \text{List } t) \text{ (iterate } t)$$

$$\vdash \forall t_1. \text{Prog } ((t \rightarrow t_1) \rightarrow \text{List } t \rightarrow \text{List } t_1) \text{ (map } t \ t_1)$$

The theorem we want to prove is that, for all correctly typed programs, two infinite lists generated in different ways are equal.

$$\vdash \forall f \ t \ x. \text{Prog } (t \rightarrow t) \ f \ \wedge \ \text{Prog } t \ x \supset$$

$$(\text{iterate } t) \ f \ (f \ x) == (\text{map } t \ t) \ f ((\text{iterate } t) \ f \ x)$$

This is true because both sides are equal to the list

$$[f \ x, \ f \ (f \ x), \ f \ (f \ (f \ x)), \ \dots]$$

The proof is by strong co-induction using the relation S with definition

$$S \ a \ b \equiv \exists f \ x. \text{Prog } t \rightarrow t \ f \ \wedge \ \text{Prog } t \ x \wedge$$

$$(a = (\text{iterate } t) \ f \ (f \ x)) \wedge (b = (\text{map } t \ t) \ f \ ((\text{iterate } t) \ f \ x))$$

It is easy to prove that

$$S \ ((\text{iterate } t) \ f \ (f \ x)) \ ((\text{map } t \ t) \ f \ ((\text{iterate } t) \ f \ x))$$

It remains to prove that S is included in $==$. By strong co-induction we need only show that S is included in $F_{==} (S \cup ==)$. Expanding with the definitions we need to show that for any (a, b) such that $S \ a \ b$

1. $(\exists t. \text{Prog } t \ a \ \wedge \ \text{Prog } t \ b)$
2. $\forall a' \ act. \text{LTS } a \ a' \ act \supset (\exists b'. (\text{LTS } b \ b' \ act) \ \wedge \ (S \ a' \ b' \vee \ a' == b'))$
3. $\forall b' \ act. \text{LTS } b \ b' \ act \supset (\exists a'. (\text{LTS } a \ a' \ act) \ \wedge \ (S \ a' \ b' \vee \ a' == b'))$

(1) follows easily from the definition of S and use of the type checker. (2) and (3) are similar and we look only at (2). Since a and b are related by S we know something of their form. Taking arbitrary a' and act and replacing a and b with what we know about their form we get

$$\text{LTS } ((\text{iterate } t) f (f x)) a' act \supset \\ (\exists b'. (\text{LTS } ((\text{map } t t) ((\text{iterate } t) f) x) b' act) \wedge (S a' b' \vee a' == b'))$$

A useful property of the labelled transition system is that

$$\forall e_1 e_2. \text{Eval } e_1 e_2 \supset (\text{LTS } e_1 e a = \text{LTS } e_2 e a)$$

We can use this plus the evaluator to reduce the goal, in one proof step, to:

$$\text{LTS } (\text{cons } (f x) ((\text{iterate } t) f (f (f x)))) a' act \supset \\ (\exists b'. (\text{LTS } (\text{cons } (f x) ((\text{map } t t) ((\text{iterate } t) f) (f x))) b' act) \wedge \\ (S a' b' \vee a' == b'))$$

There are only two possible transitions for a **CONS** cell so we can reduce the goal in one automated proof step, by case analysis of the possible transitions, to two goals

$$(a' = (f x) \wedge act = \text{Hd}) \supset \\ (\exists b'. (((b' = (f x) \wedge act = \text{Hd}) \vee \\ (b' = ((\text{map } t t) ((\text{iterate } t) f) (f x)) \wedge act = \text{Tl}))) \wedge \\ (S a' b' \vee a' == b'))$$

$$(a' = ((\text{iterate } t) f (f (f x))) \wedge act = \text{Tl}) \supset \\ (\exists b'. (((b' = (f x) \wedge act = \text{Hd}) \vee \\ (b' = ((\text{map } t t) ((\text{iterate } t) f) (f x)) \wedge act = \text{Tl}))) \wedge \\ (S a' b' \vee a' == b'))$$

The first goal is solved by letting $b' = (f x)$, since $(f x) == (f x)$ by reflexivity. The remaining goal is solved by letting $b' = ((\text{map } t t) ((\text{iterate } t) f) (f x))$. The result follows since the values for a' and b' are related by S .

For this example the goal can be proved by interacting with the theorem prover at almost the same level as on paper without the need to evaluate the terms or perform the case split on possible actions.

This proof has also been carried out in LCF using fixed point induction [13]. This involves reasoning about the functions as the fixed point of other functions. The proof in the system presented here is of a similar length but the use of co-induction avoids the need to reason about the functions as fixed points.

7 Conclusions

This paper presents a reasoning system for a small language constructed using the HOL theorem proving system and a purely definitional approach. Assuming the original HOL system is consistent then the system described here is also consistent. This property does not automatically hold of systems built using axioms to define the meaning of functions. The system will support a variety of styles of reasoning including both inductive and co-inductive proof.

The system can help users to develop skill with new proof methods. Co-inductive proofs can often provide simple proofs of properties. Many people are unfamiliar with the proof style and a system like the one discussed here lets them experiment with the security that the final proof must be correct.

The architecture described here is harder to build than a system which does not formalise the semantics or use a definitional approach. This extra work is needed once only and leads to a more secure and powerful system. Due to the tools provided by the HOL system the effort required to produce the definitional system is not significantly more than would be required to simply axiomatise the semantics.

Most of the additional work comes from basing the system on the operational semantics rather than introducing the meaning of each function with a set of axioms. However, the semantics, along with the interpreter, gives the ability to

reduce terms to some form of normal form by evaluating. This is a powerful feature which can prove or partially prove many goals in one proof step with no proof search.

Another reason for working with the semantics is that it allows meta-theoretic reasoning in addition to equational reasoning. As well as giving confidence in the correctness of the semantics this allows the proof of important results such as inductions. It is believed that this architecture will provide a suitable structure for automatically proving the induction theorems for large classes of user defined datatypes.

Finally, the definition of equivalence is given in terms of the reduction relation. This definition gives rise to a principal of co-induction for proving the equality of programs.

The system inherits all the functionality of the HOL system. The proofs produced are ML programs which can be reused or modified. A well known problem with theorem proving is that a proof will often fail if the goal is modified even slightly. There is hope that the proofs generated here will be fairly robust due to the high level of the proof steps.

No mention has been made of interfaces to the system. The current implementation simply uses the HOL teletype interface. There are several research projects investigating graphical interfaces to HOL. A system based on HOL will be able to use the results of such research.

Future work will include extending both the functionality of this system and the features of the embedded programming language. We also intend to compare the use of operational semantics here with the use of denotational semantics in other systems.

Acknowledgements

Thanks are due to Tom Melham and Andy Gordon for comments on this work. Thanks must also go to the Engineering and Physical Sciences Research Council for financial support.

References

- [1] 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.
- [2] 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.
- [3] 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.
- [4] 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.
- [5] 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.
- [6] Andy Gill. A graphical user interface for an equational reasoning assistant. In *Glasgow Workshop on Functional Programming*, July 1995.
- [7] 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.
- [8] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

- [9] Douglas J. Howe. Equality in lazy computation systems. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, pages 198–203, 1989.
- [10] 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.
- [11] 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.
- [12] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.
- [13] L. C. Paulson. *Logic and computation*. Cambridge University Press, 1987.
- [14] 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.
- [15] Simon J. Thompson. A logic for Miranda. *Formal Aspects of Computing*, 1, 1989.
- [16] Simon J. Thompson. Formulating Haskell. In *Workshop on Functional Programming, Ayr, 1992*, Workshops in Computing. Springer-Verlag, 1993.
- [17] 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.