A Case Study of Co-induction in Isabelle HOL *

Jacob Frost †
Computer Laboratory
University of Cambridge
e-mail:Jacob.Frost@cl.cam.ac.uk

August 1993

Abstract

The consistency of the dynamic and static semantics for a small functional programming language was informally proved by R.Milner and M.Tofte. The notions of co-inductive definitions and the associated principle of co-induction played a pivotal role in the proof. With emphasis on co-induction, the work presented here deals with the formalisation of this result in the higher-order logic of the generic theorem prover Isabelle.

1 Introduction

In the paper Co-induction in Relational Semantics [1], R.Milner and M.Tofte prove the dynamic and static semantics for a small functional programming language consistent. The dynamic semantics associates a value to an expression of the language, while the static semantics associates a type. A value has a type. Consistency requires that the value of an expression has the type of the expression. Values can be infinite or non-well-founded because the language contains recursive functions. Non-well-founded values are handled using co-inductive definitions and the corresponding proof principle of co-induction. The notion of greatest fixed points are used to deal with co-inductive definitions. The aim of their paper is to direct attention to the principle of co-induction, by giving an example of its application to computer science.

The purpose of this paper is to investigate how the same consistency result can be proved formally in the higher-order logic (HOL) of the generic theorem prover Isabelle. There is little doubt that it is possible to prove the same or at least a very similar result in Isabelle HOL. A more interesting question is how easy and natural this can be done, in particular how well Isabelle HOL can handle the notions of

*Supported by ESPRIT Basic Research Project 6453, Types for Proofs and Programs.

†From October 93: Department of Computer Science, Building 344, The Technical University of Denmark, DK-2800 Lyngby, Denmark, e-mail:jf@id.dth.dk

co-inductive definitions and co-induction. It should be just as easy and natural to do a formal proof as doing a detailed informal proof, preferably easier. To answer the above question, and thereby unveiling strong and weak points of the Isabelle system, is therefore also a purpose of this paper. In order to come up with an answer, the formal proof in Isabelle HOL will be compared to its more informal counterpart throughout this paper. Besides the above, doing a formal proof of consistency in Isabelle HOL is a good opportunity to learn about both Isabelle HOL and co-induction in general.

This paper is meant to be largely self contained. As a consequence a survey of the original paper [1] is given first. For the same reason, it is followed by an overview of the Isabelle system and its implementation of HOL. Finally the formalisation of consistency in Isabelle HOL is described and discussed.

2 Co-induction in Relation Semantics

The aim of this section, is to give an overview of the parts of the paper [1] that must be formalised in order to prove the consistency result. For a more thorough treatment please refer to the original paper [1].

The original paper is concerned with proving the consistency of the dynamic and static semantics of a small functional programming language. As a consequence they first define such a language. Then they define the dynamic semantic, which associates values to expressions and the static semantics which associates types to expressions. Finally they state and prove consistency.

The first part of this section is concerned with notation. After that the rest of this section will follow the structure of the original paper.

2.1 Notation

The notation used here is quite similar to that of the original paper. It differs slightly in order to make the notation of this paper more homogeneous.

Let A and B be two sets. In the following, the disjoint union is written $A + B$ and the set of finite maps from A to B as $A \xrightarrow{\text{fin}} B$. A finite map is written on the form $\{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$. For $f \in A \xrightarrow{\text{fin}} B$, $\mathrm{dom}(f)$ denotes the domain and $\mathrm{rng}(f)$ the range of the map. If $f, g \in A \xrightarrow{\text{fin}} B$ then $f + g$ means f modified by g.

2.2 The Language

The language of expressions is defined by the BNF shown in figure 1. There are five different kinds of expressions: constants including constant functions, variables, abstractions, recursive functions and applications.

A key point here, is the existence of recursive functions. Without those there would be no need for non-well-founded values and consequently no need for co-induction in the proof of consistency.

$c \in \mathrm{Const}$
$v \in \mathrm{Var}$
$e \in \mathrm{Ex} ::= \mathrm{Const} \mid \mathrm{Var} \mid \textbf{fn}\ \mathrm{Var} \Rightarrow \mathrm{Ex} \mid \textbf{fix}\ \mathrm{Var}(\mathrm{Var}) = \mathrm{Ex} \mid \mathrm{Ex}\ \mathrm{Ex}$

Figure 1: The Language

2.3 Dynamic Semantics

An expression evaluates to a value in some environment. The purpose of the dynamic semantics is to relate environments and expressions to values. The first task is therefore to explain the notions of environments and values. Furthermore it must be explained how constant functions are applied to constants.

$$\begin{aligned}
v \in \mathrm{Val} &= \mathrm{Const} + \mathrm{Clos} && (1)\\
ve \in \mathrm{ValEnv} &= \mathrm{Var} \xrightarrow{\mathrm{fin}} \mathrm{Val} && (2)\\
cl \text{ or } \langle x, e, ve\rangle \in \mathrm{Clos} &= \mathrm{Var} \times \mathrm{Ex} \times \mathrm{ValEnv} && (3)\\
\text{Exists unique } cl_\infty \text{ solving: } cl_\infty &= \langle x, e, ve + \{f \mapsto cl_\infty\}\rangle && (4)\\
\mathrm{apply} &\in \mathrm{Const} \times \mathrm{Const} \to \mathrm{Const} && (5)
\end{aligned}$$

Figure 2: Values, Environments, Closures, ...

Values (1) are either constants or closures. Constant expressions always evaluate to constant values, while abstractions and recursively defined functions always evaluate to closures. Applications can evaluate to either. Environments (2) maps variables to values. Closures (3) represent functions and are triples consisting of the parameter to the function, the function body and the environment in which the body should be evaluated. Closures can be infinite or non-well-founded due to the existence of recursive functions. In case of a recursive function, the environment of the closure will map the name of the function to the closure itself. The requirement (4) expresses that the three mutual recursive equations (1)- (3) must be solved such that the set of closures contains the non-well-founded closures. Finally, the function apply (5) is supposed to capture how constant functions are applied to constants.

The dynamic semantics is a relational semantics often called a natural semantics. It is formulated as an inference system consisting of six rules. All the rules have conclusions of the form $ve \vdash e \longrightarrow v$, read e evaluates to v in ve. The inference system appears in figure 3.

It is worth noting that the only purpose of (4) is to ensure that the dynamic semantics always relates a unique value to a recursive function. It is never used directly in the proof of consistency.

$$\frac{}{ve \vdash c \longrightarrow c}$$

$$\frac{x \in \mathrm{dom}(ve)}{ve \vdash x \longrightarrow ve(x)}$$

$$\frac{}{ve \vdash \mathbf{fn}\ x \Rightarrow e \longrightarrow \langle x, e, ve \rangle}$$

$$\frac{cl_\infty = \langle x, e, ve + \{f \mapsto cl_\infty\} \rangle}{ve \vdash \mathbf{fix}\ f(x) = e \longrightarrow cl_\infty}$$

$$\frac{ve \vdash e_1 \longrightarrow c_1 \quad ve \vdash e_2 \longrightarrow c_2 \quad c = \mathrm{apply}(c_1, c_2)}{ve \vdash e_1\ e_2 \longrightarrow c}$$

$$\frac{\begin{array}{r} ve \vdash e_1 \longrightarrow \langle x', e', ve' \rangle \\ ve \vdash e_2 \longrightarrow v_2 \\ ve' + \{x' \mapsto v_2\} \vdash e' \longrightarrow v \end{array}}{ve \vdash e_1\ e_2 \longrightarrow v}$$

Figure 3: Dynamic Semantics

2.4 Static Semantics

An expression elaborates to a type in some type environment. The purpose of the static semantics is to relate type environments and expressions to types. Before this can be done, the notion of type and type environments must be explained. It must also be explained what type a constant has.

$$\begin{array}{rcll} \tau & \in & \mathrm{Ty} ::= \{\mathrm{int}, \mathrm{bool}, \ldots\} \mid \mathrm{Ty} \to \mathrm{Ty} & (6) \\ te & \in & \mathrm{TyEnv} = \mathrm{Var} \xrightarrow{\mathrm{fin}} \mathrm{Ty} & (7) \\ \mathrm{isof} & \subseteq & \mathrm{Const} \times \mathrm{Ty} & (8) \\ \multicolumn{3}{l}{\text{If } c_1 \text{ isof } \tau_1 \to \tau_2 \text{ and } c_2 \text{ isof } \tau_1 \text{ then apply}(c_1, c_2) \text{ isof } \tau_2} & (9) \end{array}$$

Figure 4: Types, Type Environments, ...

Types are primitive types, such as int, bool or function types (6). Type environments map variables to types (7). The correspondence relation isof (8) relate a constant to its type. The idea is that it should relate for example 3 to int and true to bool. It must be consistent with the function apply (9). The relation isof is extended pointwise to relate environments and type environments.

The static semantics is again a relational semantics, formulated as an inference system and consisting of five rules. All the rules have conclusions of the form $te \vdash e \Longrightarrow \tau$, read e elaborates to τ in te. The inference system is shown in figure 5.

$$\frac{c \text{ isof } \tau}{te \vdash c \Longrightarrow \tau}$$

$$\frac{x \in \text{dom}(te)}{te \vdash x \Longrightarrow te(x)}$$

$$\frac{te + \{x \mapsto \tau_1\} \vdash e \Longrightarrow \tau_2}{te \vdash \textbf{fn } x \Rightarrow e \Longrightarrow \tau_1 \to \tau_2}$$

$$\frac{te + \{f \mapsto \tau_1 \to \tau_2\} + \{x \mapsto \tau_1\} \vdash e \Longrightarrow \tau_2}{te \vdash \textbf{fix } f(x) = e \Longrightarrow \tau_1 \to \tau_2}$$

$$\frac{te \vdash e_1 \Longrightarrow \tau_1 \to \tau_2 \quad te \vdash e_2 \Longrightarrow \tau_1}{te \vdash e_1\ e_2 \Longrightarrow \tau_2}$$

Figure 5: Static Semantics

2.5 Consistency

The original paper is concerned with proving what is called basic consistency (10). Basic consistency expresses that in corresponding environments, expressions evaluating to constants must elaborate to the type of the constant. At first it might seem strange only to consider constant values. The reason is that functions, represented as closures, only are of interest because they can be applied and in the end yield some constant value.

Basic consistency cannot be proved directly by induction on the structure of evaluations. The reason is that an evaluation resulting in a constant might require evaluations resulting in closures. Attempting to do a proof, it shows up as too weak induction hypothesises.

Basic Consistency

$$\text{If } ve \text{ isof } te \text{ and } ve \vdash e \longrightarrow c \text{ and } te \vdash e \Longrightarrow \tau \text{ then } c \text{ isof } \tau \tag{10}$$

Consistency

$$\text{If } ve : te \text{ and } ve \vdash e \longrightarrow v \text{ and } te \vdash e \Longrightarrow \tau \text{ then } v : \tau \tag{11}$$

Figure 6: Basic Consistency and Consistency

It is necessary to prove a stronger result, called consistency (11). Consistency is formulated by extending the correspondence relation isof. The extended correspondence relation, :, also expresses what it means for a closure to have a type. It is defined as the greatest fixed point of the function in figure 7. See [1] for a discussion of why this particular function was chosen.

The notion of greatest fixed point is defined in figure 8. The corresponding

$$
\begin{array}{l}
\mathrm{f}(s) \equiv \{\ \langle v, \tau \rangle \mid \\
\qquad\qquad \text{if } v = c \text{ then } c \text{ isof } \tau; \\
\qquad\qquad \text{if } v = \langle x, e, ve \rangle \\
\qquad\qquad \text{then there exist a } te \text{ such that} \\
\qquad\qquad\qquad te \vdash \mathbf{fn}\ x \Rightarrow e \Longrightarrow \tau \text{ and} \\
\qquad\qquad\qquad \mathrm{dom}(ve) = \mathrm{dom}(te) \text{ and} \\
\qquad\qquad\qquad \langle ve(x), te(x) \rangle \in s \text{ for all } x \in \mathrm{dom}(ve) \\
\qquad\quad \} \\
v : \tau \equiv \langle v, \tau \rangle \in \mathrm{gfp}(\mathrm{Val} \times \mathrm{Ty}, \mathrm{f})
\end{array}
$$

Figure 7: The Extended Correspondence Relation

principle of co-induction expresses, that in order prove that a set s is included in the greatest fixed point of some function f, it is enough to prove that it is f-consistent, i.e. $s \subseteq f(s)$.

Greatest Fixed Points

$$\mathrm{gfp}(u, f) \equiv \bigcup \{ s \subseteq u \mid s \subseteq f(s) \}$$

Co-induction

$$\frac{s \subseteq f(s)}{s \subseteq \mathrm{gfp}(u, f)}$$

Figure 8: Greatest Fixed Points and Co-induction

It is interesting to consider what would happen if : was defined using the least fixed point instead of the greatest. The function does not require non-well-founded closures to be related to types. As a consequence taking the least fixed point, only the well-founded closures would be related to types. This would make it impossible to prove the result because closures might be non-well-founded. On the other hand, the function does not prevent non-well-founded closure from being related to types. Therefore taking the greatest fixed point causes non-well-founded closures to be related as well.

Consistency is proved by induction on the structure of evaluations or as they expressed in [1], on the depth of the inference. Applying induction, results in six cases, one for each of the rules of the dynamic semantics. The case for recursive functions is the most interesting in that it uses co-induction. The reason is that the non-well-founded closures are introduced by recursive functions.

3 Isabelle HOL

Isabelle is a generic theorem prover. It can be instantiated to support reasoning in an object-logic by extending its meta-logic. All the symbols of the object-logic are declared using typed lambda calculus, while the rules are expressed as axioms in the meta-logic.

Having explained the notation used, the typed lambda calculus used by Isabelle is first described. Then the pure Isabelle system is described and it is explained how the pure Isabelle system is extended to support reasoning in HOL. Finally an implementation of set theory in HOL is described. The following can only give an overview, but there are several papers describing Isabelle and its object-logics in detail, for example [3, 5, 4].

3.1 Notation

Here and in the rest of the paper, an Isabelle-like notation will be used. The Isabelle system uses an ASCII-notation. When working in Isabelle it is often necessary to supply information about the syntax, such as where arguments should be placed when using mix-fix notation, precedence etc. In order to improve readability most such information is left out here and a more mathematical notation is adapted, allowing the use of mathematical symbols etc.

3.2 Typed Lambda Calculus

Isabelle represents syntax using the typed λ-calculus. Lambda abstraction is written $\lambda x.t$ and application $t_1(t_2)$, where x is a variable and t, t_1, t_2 are terms.

Types in Isabelle can be polymorphic, ie. contain type variables such as α in the type α `list`. Function types are written $\sigma_1 \Rightarrow \sigma_2$, where σ_1 and σ_2 are types. New constants are declared by giving their type, for example: `succ` :: `nat` $\Rightarrow$ `nat`.

Isabelle uses a notion of classes to control polymorphism. Each type belong to a class. A class can be a subset of another class. Isabelle contains the built-in class `logic` of logical types. A new class is declared as a subclass of another class, for example the class `term` of terms which is included in the class `logic`. New types and type constructors can be declared by giving the class of the arguments and the result, for example `list` :: (`term`)`term`.

Curried abstraction $\lambda x_1 . \ldots \lambda x_n . t$ is abbreviated $\lambda x_1 \ldots x_n . t$, and curried application $t(t_1) \ldots (t_n)$ as $t(t_1, \ldots, t_n)$. Similar curried function types $\sigma_1 \Rightarrow (\ldots \sigma_n \Rightarrow \sigma \ldots)$ are abbreviated $[\sigma_1, \ldots, \sigma_n] \Rightarrow \sigma$.

3.3 Pure Isabelle

Object-logics are implemented by extending pure Isabelle which is described here. Pure Isabelle consist of the meta-logic and has support for doing proofs in this meta-logic and its possible extensions.

3.3.1 Syntax

Isabelle's meta-logic is a fragment of intuitionistic higher order logic. The symbols of the meta-logic is declared exactly the same way as symbols of an object-logic, by using typed lambda calculus. There is a built-in subclass of `logic` called `prop` of meta-level truth values. The symbols of the meta-logic are the three connectives, shown in figure 9, corresponding to implication, universal quantification and equality.

INFIXES

$$
\begin{aligned}
\Longrightarrow &:: [\texttt{prop}, \texttt{prop}] \Rightarrow \texttt{prop} \\
\bigwedge &:: (\alpha :: \texttt{logic} \Rightarrow \texttt{prop}) \Rightarrow \texttt{prop} \\
\equiv &:: [\alpha :: \texttt{logic}, \alpha] \Rightarrow \texttt{prop}
\end{aligned}
$$

Figure 9: Meta-level connectives

Nested implication $\phi_1 \Longrightarrow (\ldots \phi_n \Longrightarrow \phi \ldots)$ may be abbreviated $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$ and outer quantifiers can be dropped. Here $\neg b_1 = b_2$ is also written $b_1 \neq b_2$.

3.3.2 Inferences

The meta-logic is defined by a set of primitive axioms and inference rules. Proofs are seldom constructed using these rules. Usually a derived rule, the resolution rule, is used:

$$\frac{[\![\psi_1; \ldots; \psi_m]\!] \Longrightarrow \psi \quad [\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi}{([\![\phi_1; \ldots; \phi_{i-1}; \psi_1; \ldots \psi_m; \phi_{i+1}; \ldots; \phi_n]\!] \Longrightarrow \phi)s}(\psi s \equiv \phi_i s)$$

Here $1 \leq i \leq n$ and s is a higher order unifier of ψ and ϕ. A big machinery is connected with resolution and higher order unification. This includes schematic variables, lifting over formulae and variables etc. For the details refer to [3, 5].

3.3.3 Proofs

It is possible to construct proofs both in a forward and backward fashion in Isabelle. Bigger proofs are however usually constructed backward.

In Isabelle, a backwards proof is done by refining a proof state, until the desired result is proved. A proof state is simply a meta-level theorem of the form $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$, where $\phi_1, \ldots, \phi_n$ can be seen as subgoals and ϕ as the main goal. Repeatedly refining such a proof state, by resolving it with suitable rules, corresponds to applying rules to the subgoals until they all are proved.

In order to manage backward proofs, Isabelle has a subgoal module. It keeps track of the current and previous proof states. This make it possible, for example, to undo proof steps.

PREFIXES

$\neg :: \texttt{bool} \Rightarrow \texttt{bool}$	negation
$= :: [\alpha :: \texttt{term}, \alpha] \Rightarrow \texttt{bool}$	equality

INFIXES

$\wedge :: [\texttt{bool}, \texttt{bool}] \Rightarrow \texttt{bool}$	conjunction
$\vee :: [\texttt{bool}, \texttt{bool}] \Rightarrow \texttt{bool}$	disjunction
$\rightarrow :: [\texttt{bool}, \texttt{bool}] \Rightarrow \texttt{bool}$	implication

BINDERS

$\forall :: [\alpha :: \texttt{term} \Rightarrow \texttt{bool}] \Rightarrow \texttt{bool}$	universal quantification
$\exists :: [\alpha :: \texttt{term} \Rightarrow \texttt{bool}] \Rightarrow \texttt{bool}$	existential quantification

Figure 10: Symbols of Isabelle HOL

3.3.4 Tactics and Tacticals

Tactics perform backward proofs. They are applied to a proof state and may change several of the subgoals.

Isabelle has many different tactics. There are tactics for proving a subgoal by assumption, different forms of resolution for applying rules to subgoals etc. These will work in all logics.

Isabelle also have a number of generic packages, which depend on properties of the logic in question. To mention two, a classical reasoning package and a simplifier package. Each contain a number of tactics. To get access to these, the packages must be successfully instantiated for the actual logic. Then the classical reasoning package, for example, will provide a suite of tactics for doing proofs, using classical proof procedures. The tactic `fast_tac` for example will try to solve a subgoal, by applying the rules in a supplied set of rules in a depth first manner.

Tactics can be combined to new tactics using tacticals. There are tacticals for doing depth-first, best-first search etc., but also simpler tacticals for sequential composition, choice, repetition etc.

3.4 HOL

A number of logics have been implemented in Isabelle. They are described in [4]. Among these is HOL. The description of HOL given here will be brief and only cover parts relevant to the rest of the presentation. For a thorough treatment refer to [4].

There is a subclass of `logic`, called `term` of higher order terms and a type belonging to this, `bool` of object-level truth values. There is an implicit coercion to meta-level truth values called `Trueprop`. The symbols needed for this paper is declared in figure 10.

The formulation of HOL in Isabelle, identifies meta-level and object-level types.

TYPES

$\texttt{set} :: (\texttt{term})\texttt{term}$

CONSTANTS

$(\{\_\}) :: \alpha \Rightarrow \alpha\ \texttt{set}$	singleton

BINDERS

$(\{\_.\_\}) :: [\alpha \Rightarrow \texttt{bool}] \Rightarrow \alpha\ \texttt{set}$	comprehension

INFIXES

$\in :: [\alpha, \alpha\ \texttt{set}] \Rightarrow \texttt{bool}$	membership
$\cup :: [\alpha\ \texttt{set}, \alpha\ \texttt{set}] \Rightarrow \alpha\ \texttt{set}$	union
$\cap :: [\alpha\ \texttt{set}, \alpha\ \texttt{set}] \Rightarrow \alpha\ \texttt{set}$	intersection

PREFIXES

$\bigcup :: ((\alpha\ \texttt{set})\ \texttt{set}) \Rightarrow \alpha\ \texttt{set}$	general union
$\bigcap :: ((\alpha\ \texttt{set})\ \texttt{set}) \Rightarrow \alpha\ \texttt{set}$	general intersection

Figure 11: Symbols of the Set Theory

This makes it possible to take advantage of Isabelle's type system. Type checking is done automatically and most type constraints are implicit.

Using Isabelle HOL one often wants to define new types. Isabelle does not support type definitions, but they can be mimicked by explicit definition of isomorphism functions. See [2].

The meaning of the symbols is defined by a number of rules. They are usually formulated as introduction or elimination rules. Taking $\vee$ as an example, one of its introduction rules is $P \Longrightarrow P \vee Q$ and the elimination rule is $[\![P \vee Q; P \Longrightarrow R; Q \Longrightarrow R]\!] \Longrightarrow R$.

Most of the generic reasoning packages are instantiated to support reasoning in HOL. This includes the simplifier and the classical reasoning package.

3.5 Set Theory

A formulation of set theory has been given within Isabelle HOL. Again only the relevant part of the theory is covered here, but a detailed description of the full theory can be found in [4].

In order to formulate the set theory a new type constructor `set` is declared. Then the symbols of the set theory are declared. The symbols necessary for this presentation are shown in figure 11.

Just as before the meaning of the symbols is defined by a number of rules. The set theory also contains a large number of derived rules. Most of the generic reasoning packages are also instantiated to support reasoning in the set theory.

The set theory is used to define a number of new types and type constructors, using the technique described in [2]. Examples include natural numbers `nat`, disjoint unions $+$ and products $*$.

4 Formalisation in Isabelle HOL

This section describes how the contents of the original paper has been formalised in Isabelle HOL.

The formalisation rests on a theory of least and greatest fixed points. This theory is described first. After that the structure follows that of §2, describing the formalisation of each of the necessary constructs in turn. Finally, some aspects of the formalisation are discussed.

4.1 Least and Greatest Fixed Points

A theory of least and greatest fixed points has been developed in Isabelle HOL [2]. The theory of least fixed points can be used to deal with the formalisation of inductive definitions in Isabelle HOL. Examples are inductively defined datatypes and relations, such as the ones found in the original paper, from now on just called inductive datatypes and relations. Similarly the theory of greatest fixed points can be used to deal with the formalisation of co-inductive definitions of for example datatypes and relations. These will be called co-inductive datatypes and relations. The extended correspondence relation in the original paper can be seen as a co-inductive relation.

The theory of least and greatest fixed points is based on the Isabelle HOL set theory. The definitions of least and greatest fixed points, which appear in figure 12 resemble usual set theoretic definitions.

	LEAST FIXED POINTS	GREATEST FIXED POINTS
CONSTANT	$\mathtt{lfp} :: [\alpha\ \mathtt{set} \Rightarrow \alpha\ \mathtt{set}] \Rightarrow \alpha\ \mathtt{set}$	$\mathtt{gfp} :: [\alpha\ \mathtt{set} \Rightarrow \alpha\ \mathtt{set}] \Rightarrow \alpha\ \mathtt{set}$
AXIOM	$\mathtt{lfp}(f) \equiv \bigcap\{s.f(s) \subseteq s\};$	$\mathtt{gfp}(f) \equiv \bigcup\{s.s \subseteq f(s)\};$

Figure 12: Least and Greatest Fixed Points

Most of the properties, derived from the definitions, can be characterised as either introduction or elimination rules. In figure 13, the introduction rules can be used to conclude that some set is included in the least or greatest fixed point, the elimination rules that some set contains the least or greatest fixed point.

Induction is elimination. Co-induction is introduction. Both follow directly from the definitions. Taking the intersection, the least fixed point must be included in all s such that $f(s) \subseteq s$. Similar taking the union the greatest fixed point include all s such that $s \subseteq f(s)$.

	Least Fixed Points	Greatest Fixed Points
Co-induction		$s \subseteq f(s) \Longrightarrow s \subseteq \mathtt{gfp}(f);$
Introduction	$\mathtt{mono}(f) \Longrightarrow$ $f(\mathtt{lfp}(f)) \subseteq \mathtt{lfp}(f);$	$\mathtt{mono}(f) \Longrightarrow$ $f(\mathtt{gfp}(f)) \subseteq \mathtt{gfp}(f);$
Induction	$f(s) \subseteq s \Longrightarrow \mathtt{lfp}(f) \subseteq s;$	
Elimination	$\mathtt{mono}(f) \Longrightarrow$ $\mathtt{lfp}(f) \subseteq f(\mathtt{lfp}(f));$	$\mathtt{mono}(f) \Longrightarrow$ $\mathtt{gfp}(f) \subseteq f(\mathtt{gfp}(f));$
Fixed Point	$\mathtt{mono}(f) \Longrightarrow$ $\mathtt{lfp}(f) = f(\mathtt{lfp}(f));$	$\mathtt{mono}(f) \Longrightarrow$ $\mathtt{lfp}(f) = f(\mathtt{lfp}(f));$

Figure 13: Properties of Least and Greatest Fixed Points

In order to derive the introduction rule for least fixed points and the elimination rule for greatest fixed points it is necessary to assume that the function is monotone. The intersection or the union of a set of sets satisfying some condition, does not necessarily satisfy the condition themselves.

Not very surprisingly, least fixed points enjoy an elimination rule corresponding to the one for greatest fixed points. Similarly greatest fixed points enjoy an introduction rule corresponding to the one for least fixed points. They can be derived from the induction respectively co-induction rule, by assuming that f is monotone.

Deriving the fixed point property from the introduction and elimination rules is trivial.

Notice that the definitions and properties are completely symmetric. It is possible to go from least to greatest fixed points and back, by swapping `lfp` with `gfp`, the arguments of $\subseteq$ and intersections with unions. Doing this, introduction rules becomes elimination rules and vice versa.

The induction and co-induction rule in figure 13 are the only rules that do not assume that f is monotone. It is difficult not to ask which rules can be derived, assuming that f is monotone. In fact a lot of different rules can be derived, but so far I have found the two rules in figure 14 the most useful. Again the properties are symmetric.

In practice, introduction rules are used to prove that some element belongs to a fixed point, elimination rules that some property holds for all elements of a fixed point. All the rules are therefore put into a form that supports this kind of reasoning.

The only rules shown in figure 15 are the induction and co-induction rules, but the other rules can of course be reformulated in the same way.

Most of the rules described above come as part of a Isabelle HOL theory of least and greatest fixed points. It was however necessary to derive the form of the co-induction rule shown in figure 14 and 15.

	LEAST FIXED POINTS	GREATEST FIXED POINTS
CO-INDUCTION		$[\![\, \mathtt{mono}(f);$ $\quad s \subseteq f(s \cup \mathtt{gfp}(f))$ $]\!] \Longrightarrow$ $s \subseteq \mathtt{gfp}(f);$
INDUCTION	$[\![\, \mathtt{mono}(f);$ $\quad f(s \cap \mathtt{lfp}(f)) \subseteq s$ $]\!] \Longrightarrow$ $\mathtt{lfp}(f) \subseteq s;$	

Figure 14: Properties of Least and Greatest Fixed Points

	LEAST FIXED POINTS	GREATEST FIXED POINTS
CO-INDUCTION		$[\![\, \mathtt{mono}(f);$ $\quad x \in f(\{x\} \cup \mathtt{gfp}(f))$ $]\!] \Longrightarrow$ $x \in \mathtt{gfp}(f);$
INDUCTION	$[\![\, \mathtt{mono}(f);$ $\quad x \in \mathtt{lfp}(f);$ $\quad \bigwedge y.$ $\qquad y \in f(\mathtt{lfp}(f) \cap \{z.p(z)\}) \Longrightarrow$ $\qquad p(y)$ $]\!] \Longrightarrow$ $p(x);$	

Figure 15: Properties of Least and Greatest Fixed Points

4.2 The Language

Expressions, defined by the BNF in figure 1, can be formally expressed in Isabelle HOL as an inductive datatype. It can be done using the theory of least fixed points, but require quite a lot of tedious work. First the set of expressions must be defined as the least fixed point of a suitable monotone function. Then, in order to make expressions distinct from members of other types and to take advantage of Isabelle's type system, the set of expressions should be related to an abstract meta-level type of expressions. It can be done by declaring two isomorphism functions, an abstraction and a representation function. Using these, operations and properties should be lifted to the abstract level. From then on, all reasoning should take place at the abstract level. A description of the above method can be found in [2]

The solution adapted here and in the following is to give an axiomatic specification of inductive datatypes. Of course there is a greater risk of introducing errors, but given the amount of work otherwise required, that the above method for formalising inductive datatypes has been investigated elsewhere and that axiomatisation of inductive datatypes is well understood, it seems a sensible choice.

A standard axiomatisation of expressions is shown in figure 16. A type of expressions `Ex` is declared together with constants corresponding to the constructors of the datatype. There are rules stating that the constructors are distinct and injective. Because `Ex` is an inductive datatype there is also an induction rule. There is no need for introduction rules, because the constructors have been declared using the typed lambda calculus. It also simplifies for example the induction rule, because no type constraints have to appear explicitly.

Neither of the above solutions are very satisfactory. The only reasonable solution from a practical point of view, would be to give the constructors and their types to Isabelle and then let Isabelle generate all the necessary definitions and derive all the necessary properties automatically. Such a package should also set up tactics to support reasoning about inductive datatypes and relations.

4.3 Dynamic Semantics

Before the dynamic semantics can be formalised it is necessary to formalise the notion of values, environments and closures. It might seem a relatively hard task if it had to be done using greatest fixed points. Furthermore only a few obvious properties are needed in order to prove consistency. All these properties hold for every solution to the three equations (1)-(3) in figure 2. The requirement (4) that the set of closures must contain all non-well-founded closures is not directly relevant to the proof of consistency. In this light, it seems acceptable simply to state the few obvious properties needed. These appear in figure 17. It must however be considered a lacking feature of Isabelle that there is no automatic support for mutually recursive co-inductive datatypes.

The inference system in figure 3 can be seen as an inductive definition of a relation, in this case relating environments, expressions and values. Although this is not stated explicitly, it is obviously a correct interpretation because consistency is proved by induction on the depth of the inference in the original paper.

An inductive relation such as in figure 3 can be represented as a set of triples. Here a triple consist of an environment, an expression and a value. Because the relation is defined inductively, the corresponding set can be defined as the least fixed point of a function derived from the rules of the inference system. The formalisation in Isabelle HOL is based on this idea and appear in figure 18.

The function `eval_fun` is obtained directly from the rules of the inference system. For each rule all free variables are existentially quantified. The triple corresponding to the conclusion is claimed equal to pp. Every occurrence of the relation as a premise is translated to a corresponding triple and claimed to belong to the argument s of the function. Every other premise is translated directly into a corresponding Isabelle HOL formula. Finally all the pieces are combined by conjunctions and disjunctions and packed into a set comprehension.

Types

$\texttt{Const} :: \texttt{term} \quad \texttt{ExVar} :: \texttt{term} \quad \texttt{Ex} :: \texttt{term}$

Constants

$\texttt{e\_const} :: \texttt{Const} \Rightarrow \texttt{Ex}$
$\texttt{e\_var} :: \texttt{ExVar} \Rightarrow \texttt{Ex}$
$(\texttt{fn}\ \_ \ \Rightarrow\ \ \_) :: [\texttt{ExVar}, \texttt{Ex}] \Rightarrow \texttt{Ex}$
$(\texttt{fix}\ \_(\_)\ =\ \ \_) :: [\texttt{ExVar}, \texttt{ExVar}, \texttt{Ex}] \Rightarrow \texttt{Ex}$
$(\_@\_) :: [\texttt{Ex}, \texttt{Ex}] \Rightarrow \texttt{Ex}$

Injectiveness Axioms

$$
\begin{array}{l}
\texttt{e\_const}(c_1) = \texttt{e\_const}(c_2) \Longrightarrow c_1 = c_2; \\
\qquad\qquad \vdots \\
e_{11}@e_{12} = e_{21}@e_{22} \Longrightarrow e_{11} = e_{21} \wedge e_{12} = e_{22};
\end{array}
$$

Distinctness Axioms

$$
\begin{array}{l}
\texttt{e\_const}(c) \neq \texttt{e\_var}(x); \ldots \texttt{e\_const}(c) \neq e_1@e_2; \\
\qquad\qquad \vdots \\
\texttt{fix}\ f(x) = e_1 \neq e_1@e_2;
\end{array}
$$

Induction Axiom

$$
\begin{array}{l}
[\![\bigwedge x.p(\texttt{e\_var}(x)); \\
\ \ \bigwedge c.\ p(\texttt{e\_const}(c)); \\
\ \ \bigwedge x\ e.\ p(e) \Longrightarrow p(\texttt{fn}\ x \Rightarrow e); \\
\ \ \bigwedge f\ x\ e.\ p(e) \Longrightarrow p(\texttt{fix}\ f(x)\ =\ e); \\
\ \ \bigwedge e_1\ e_2.\ p(e_1) \Longrightarrow p(e_2) \Longrightarrow p(e_1@e_2) \\
]\!] \Longrightarrow \\
p(e);
\end{array}
$$

Figure 16: Expressions

Types

$\texttt{Val} :: \texttt{term} \quad \texttt{ValEnv} :: \texttt{term} \quad \texttt{Clos} :: \texttt{term}$

Constants

$\texttt{v\_const} :: \texttt{Const} \Rightarrow \texttt{Val}$
$\texttt{v\_clos} :: \texttt{Clos} \Rightarrow \texttt{Val}$

$\texttt{ve\_emp} :: \texttt{ValEnv}$
$(\_ + \{\_ \mapsto \_\}) :: [\texttt{ValEnv}, \texttt{ExVar}, \texttt{Val}] \Rightarrow \texttt{ValEnv}$
$\texttt{ve\_dom} :: \texttt{ValEnv} \Rightarrow \texttt{ExVar set}$
$\texttt{ve\_app} :: [\texttt{ValEnv}, \texttt{ExVar}] \Rightarrow \texttt{Val}$

$(\langle \_, \_, \_ \rangle) :: [\texttt{ExVar}, \texttt{Ex}, \texttt{ValEnv}] \Rightarrow \texttt{Clos}$

$\texttt{c\_app} :: [\texttt{Const}, \texttt{Const}] \Rightarrow \texttt{Const}$

Axioms

$\texttt{v\_const}(c_1) = \texttt{v\_const}(c_2) \Longrightarrow c_1 = c_2;$
$\texttt{v\_clos}(\langle x_1, e_1, ve_1 \rangle) = \texttt{v\_clos}(\langle x_2, e_2, ve_2 \rangle) \Longrightarrow$
$x_1 = x_2 \wedge e_1 = e_2 \wedge ve_1 = ve_2;$
$\texttt{v\_const}(c) \neq \texttt{v\_clos}(cl);$

$\texttt{ve\_dom}(ve + \{x \mapsto v\}) = \texttt{ve\_dom}(ve) \cup \{x\};$
$\texttt{ve\_app}(ve + \{x \mapsto v\}, x) = v;$
$x_1 \neq x_2 \Longrightarrow \texttt{ve\_app}(ve + \{x_1 \mapsto v\}, x_2) = \texttt{ve\_app}(ve, x_2);$

Figure 17: Values, Environments, Closures,...

Constants

$\texttt{eval\_fun} :: (\texttt{ValEnv} * \texttt{Ex} * \texttt{Val})\ \texttt{set} \Rightarrow (\texttt{ValEnv} * \texttt{Ex} * \texttt{Val})\ \texttt{set}$
$\texttt{eval\_rel} :: (\texttt{ValEnv} * \texttt{Ex} * \texttt{Val})\ \texttt{set}$
$(\_ \vdash \_ \Longrightarrow \_) :: [\texttt{ValEnv}, \texttt{Ex}, \texttt{Val}] \Rightarrow \texttt{bool}$

Axioms

$$
\begin{array}{l}
\texttt{eval\_fun}(s) \equiv \\
\{\ pp. \\
\quad (\exists ve\ c.pp = \langle\langle ve, \texttt{e\_const}(c)\rangle, \texttt{v\_const}(c)\rangle)\vee \\
\quad (\exists ve\ x.pp = \langle\langle ve, \texttt{e\_var}(x)\rangle, \texttt{ve\_app}(ve, x)\rangle \wedge x \in \texttt{ve\_dom}(ve))\vee \\
\quad (\exists ve\ e\ x.pp = \langle\langle ve, \texttt{fn}\ x \Rightarrow e\rangle, \texttt{v\_clos}(\langle x, e, ve\rangle)\rangle)\vee \\
\quad (\ \exists ve\ e\ x\ f\ cl. \\
\qquad pp = \langle\langle ve, \texttt{fix}\ f(x) = e\rangle, \texttt{v\_clos}(cl_\infty)\rangle\wedge \\
\qquad cl_\infty = \langle x, e, ve + \{f \mapsto \texttt{v\_clos}(cl_\infty)\}\rangle \\
\quad)\vee \\
\quad (\ \exists ve\ e_1\ e_2\ c_1\ c_2. \\
\qquad pp = \langle\langle ve, e_1 @ e_2\rangle, \texttt{v\_const}(\texttt{c\_app}(c_1, c_2))\rangle\wedge \\
\qquad \langle\langle ve, e_1\rangle, \texttt{v\_const}(c_1)\rangle \in s \wedge \langle\langle ve, e_2\rangle, \texttt{v\_const}(c_2)\rangle \in s \\
\quad)\vee \\
\quad (\ \exists ve\ ve'\ e_1\ e_2\ e'\ x'\ v\ v_2. \\
\qquad pp = \langle\langle ve, e_1 @ e_2\rangle, v\rangle\wedge \\
\qquad \langle\langle ve, e_1\rangle, \texttt{v\_clos}(\langle x', e', ve'\rangle)\rangle \in s\wedge \\
\qquad \langle\langle ve, e_2\rangle, v_2\rangle \in s\wedge \\
\qquad \langle\langle ve' + x' \mapsto v_2, e'\rangle, v\rangle \in s \\
\quad) \\
\}; \\
\texttt{eval\_rel} \equiv \texttt{lfp}(\texttt{eval\_fun}); \\
ve \vdash e \longrightarrow v \equiv \langle\langle ve, e\rangle, v\rangle \in \texttt{eval\_rel};
\end{array}
$$

Figure 18: Dynamic Semantics

The formalisation of the dynamic semantics must be correct. Given the definitions figure 18, introduction rules very similar to the inference system can be derived. More importantly it is possible to derive an induction rule. Big induction rules are notoriously difficult to write. The advantage of the approach used here, compared to an axiomatic approach is that it is possible to derive the correct induction rule. Some of the introduction rules and the induction rule appear in figure 19.

Although not very difficult, it is time consuming to define relations and derive properties as described above. Automating the process would save a lot of work. Isabelle should have inductive definitions.

4.4 Static Semantics

Before formalising the static semantics, it is necessary to formalise the notions of types, type environments etc. This is done in figure 20.

The type of types is another example of a construct that could be formalised as an inductive datatype using the theory of least fixed points. But as before, and for the same reasons, this is not done. Instead it is specified axiomatically. In fact only the properties needed for this paper are stated. Similar for the notion of type environments.

Just as it was the case in the original paper, the existence of a correspondence relation, relating constants to their type is claimed. The requirement that this should be consistent with application of constants is taken directly from the original paper.

The actual inference system can again be seen as an inductive definition of a relation. Again, it is formalised in Isabelle HOL, using the theory of least fixed points. The formalisation appears in figure 21.

Surprisingly, the fact that the relation is defined as the least fixed point and therefore enjoys an induction rule is never used in the proof of consistency. It is only necessary to use ordinary elimination and it would have been possible to use the greatest fixed point for the definition instead.

The inference system has an interesting and very useful property. In a derivation of a statement $ve \vdash e \Longrightarrow \tau$, only one rule can have been used for the last inference. The reason is that there is exactly one rule for each kind of expression. As a consequence, knowing that $ve \vdash e \Longrightarrow \tau$ hold it possible to conclude that the premises of the corresponding rule hold. In other words it is possible to use each of the rules backward. This kind of reasoning is used in the proof of consistency. The ordinary elimination rule and derived elimination rules allowing the kind of reasoning just described, are shown in figure 22.

To derive the last rules it is of course necessary to use properties of expressions. They are proved in a few lines by invoking a classical reasoning tactic with a proper set of rules. Similar for the rest of the rules, it cannot be claimed that they are difficult to derive. It is, however, very time consuming.

Introduction

$$ve \vdash \texttt{e\_const}(c) \longrightarrow \texttt{v\_const}(c);$$
$$x \in \texttt{ve\_dom}(ve) \Longrightarrow e \vdash \texttt{e\_var}(x) \longrightarrow \texttt{ve\_app}(ve, x);$$
$$\vdots$$
$$\begin{array}{l} [\![\ ve \vdash e_1 \longrightarrow \texttt{v\_clos}(\langle x', e', ve'\rangle); \\ \quad ve \vdash e_2 \longrightarrow v_2; \\ \quad ve' + \{x' \mapsto v_2\} \vdash e' \longrightarrow v \\ ]\!] \Longrightarrow \\ ve \vdash e_1 @ e_2 \longrightarrow v; \end{array}$$

Induction

$$\begin{array}{l} [\![\ ve \vdash e \longrightarrow v; \\ \quad \bigwedge ve\ c.\ p(ve, \texttt{e\_const}(c), \texttt{v\_const}(c)); \\ \quad \bigwedge x\ ve.\ x \in \texttt{ve\_dom}(ve) \longrightarrow p(ve, \texttt{e\_var}(x), \texttt{ve\_app}(ve, x)); \\ \quad \bigwedge x\ ve\ e.\ p(ve, \texttt{fn}\ x \Rightarrow e, \texttt{v\_clos}(\langle x, e, ve\rangle)); \\ \quad \bigwedge x\ f\ ve\ cl_\infty\ e. \\ \qquad cl_\infty = \langle x, e, ve + \{f \mapsto \texttt{v\_clos}(cl_\infty)\}\rangle \Longrightarrow \\ \qquad p(ve, \texttt{fix}\ f(x) = e, \texttt{v\_clos}(cl_\infty)); \\ \quad \bigwedge ve\ c_1\ c_2\ e_1\ e_2. \\ \qquad [\![\ p(ve, e_1, \texttt{v\_const}(c_1)); p(ve, e_2, \texttt{v\_const}(c_2))\ ]\!] \Longrightarrow \\ \qquad p(ve, e_1 @ e_2, \texttt{v\_const}(\texttt{c\_app}(c_1, c_2))); \\ \quad \bigwedge ve\ ve'\ x'\ e_1\ e_2\ e'\ v\ v_2. \\ \qquad [\![\ p(ve, e_1, \texttt{v\_clos}(\langle x', e', ve'\rangle)); \\ \qquad\ \ p(ve, e_2\rangle, v_2\rangle); \\ \qquad\ \ p(ve' + \{x' \mapsto v_2\}, e', v) \\ \qquad]\!] \Longrightarrow \\ \qquad p(ve, e_1 @ e_2, v) \\ ]\!] \Longrightarrow \\ p(ve, e, v); \end{array}$$

Figure 19: Derived Properties

Types

$\texttt{TyConst} :: \texttt{term} \quad \texttt{Ty} :: \texttt{term} \quad \texttt{TyEnv} :: \texttt{term}$

Constants

$\texttt{t\_const} :: \texttt{TyConst} \Rightarrow \texttt{Ty}$
$(\_ \to \_) :: [\texttt{Ty}, \texttt{Ty}] \Rightarrow \texttt{Ty}$

$\texttt{te\_emp} :: \texttt{TyEnv}$
$(\_ + \{\_ \mapsto \_\}) :: [\texttt{TyEnv}, \texttt{ExVar}, \texttt{Ty}] \Rightarrow \texttt{TyEnv}$
$\texttt{te\_app} :: [\texttt{TyEnv}, \texttt{ExVar}] \Rightarrow \texttt{Ty}$
$\texttt{te\_dom} :: \texttt{TyEnv} \Rightarrow \texttt{ExVar set}$

$(\_ \texttt{ isof } \_) :: [\texttt{Const}, \texttt{Ty}] \Rightarrow \texttt{bool}$
$(\_ \texttt{ isof\_env } \_) :: [\texttt{ValEnv}, \texttt{TyEnv}] \Rightarrow \texttt{bool}$

Axioms

$\texttt{t\_const}(c_1) = \texttt{t\_const}(c_2) \Longrightarrow c_1 = c_2;$
$\tau_{11} \to \tau_{12} = \tau_{21} \to \tau_{22} \Longrightarrow \tau_{11} = \tau_{21} \wedge \tau_{12} = \tau_{22};$
$[\![\bigwedge p.\ p(\texttt{t\_const}(p));$
$\quad \bigwedge \tau_1\ \tau_2.\ p(\tau_1) \Longrightarrow p(\tau_2) \Longrightarrow p(\tau_1 \to \tau_2)$
$]\!] \Longrightarrow$
$p(\tau);$

$\texttt{te\_dom}(te + \{x \mapsto \tau\}) = \texttt{te\_dom}(te) \cup \{x\};$
$\texttt{te\_app}(te + \{x \mapsto \tau\}, x) = \tau;$
$x_1 \neq x_2 \Longrightarrow \texttt{te\_app}(te + \{x_1 \mapsto \tau\}, x_2) = \texttt{te\_app}(te, x_2);$

$ve \texttt{ isof\_env } te \equiv$
$\texttt{ve\_dom}(ve) = \texttt{te\_dom}(te) \wedge$
$(\ \forall x.$
$\quad x \in \texttt{ve\_dom}(ve) \to$
$\quad (\exists c.\ \texttt{ve\_app}(ve, x) = \texttt{v\_const}(c) \wedge c \texttt{ isof } \texttt{te\_app}(te, x))$
$);$
$[\![c_1 \texttt{ isof } \tau_1 \to \tau_2; c_2 \texttt{ isof } \tau_1]\!] \Longrightarrow \texttt{c\_app}(c_1, c_2) \texttt{ isof } \tau_2;$

Figure 20: Types, Type Environments,...

CONSTANTS

$\texttt{elab\_fun} :: (\texttt{TyEnv} * \texttt{Ex} * \texttt{Ty})\ \texttt{set} \Rightarrow (\texttt{TyEnv} * \texttt{Ex} * \texttt{Ty})\ \texttt{set}$
$\texttt{elab\_rel} :: (\texttt{TyEnv} * \texttt{Ex} * \texttt{Ty})\ \texttt{set}$
$(\_ \vdash \_ \Longrightarrow \_) :: [\texttt{TyEnv}, \texttt{Ex}, \texttt{Ty}] \Rightarrow \texttt{bool}$

AXIOMS

$$
\begin{array}{l}
\texttt{elab\_fun}(s) \equiv \\
\{\ pp. \\
\quad (\exists te\ c\ \tau . pp = \langle\langle te, \texttt{e\_const}(c)\rangle, t\rangle \wedge c\ \texttt{isof}\ \tau)\vee \\
\quad (\exists te\ x . pp = \langle\langle te, \texttt{e\_var}(x)\rangle, \texttt{te\_app}(te, x)\rangle \wedge x \in \texttt{te\_dom}(te))\vee \\
\quad (\exists te\ x\ e\ \tau_1\ \tau_2 . pp = \langle\langle te, \texttt{fn}\ x \Rightarrow e\rangle, \tau_1 \to \tau_2\rangle \wedge \langle\langle te + \{x \mapsto \tau_1\}, e\rangle, \tau_2\rangle \in s)\vee \\
\quad (\ \exists te\ f\ x\ e\ \tau_1\ \tau_2 . \\
\qquad pp = \langle\langle te, \texttt{fix}\ f(x) = e\rangle, \tau_1 \to \tau_2\rangle \wedge \\
\qquad \langle\langle te + \{f \mapsto \tau_1 \to \tau_2\} + \{x \mapsto \tau_1\}, e\rangle, \tau_2\rangle \in s \\
\quad)\vee \\
\quad (\ \exists te\ e_1\ e_2\ \tau_1\ \tau_2\ . \\
\qquad pp = \langle\langle te, e_1 @ e_2\rangle, \tau_2\rangle \wedge \langle\langle te, e_1\rangle, \tau_1 \to \tau_2\rangle \in s \wedge \langle\langle te, e_2\rangle, \tau_1\rangle \in s \\
\quad) \\
\}; \\
\texttt{elab\_rel} \equiv \texttt{lfp}(\texttt{elab\_fun}); \\
te \vdash e \Longrightarrow \tau \equiv \langle\langle te, e\rangle, \tau\rangle \in \texttt{elab\_rel};
\end{array}
$$

Figure 21: Static Semantics

4.5 Consistency

The formalisation of consistency is divided into two parts. First it is considered how to state consistency in Isabelle HOL, then how to prove it.

4.5.1 Stating Consistency

Stating consistency proceeds just as in the original paper. The real interest is on proving basic consistency. In order to do that, is necessary to state and prove the stronger consistency result. This result is stated using an extended version of the correspondence relation `isof`.

The effort is concentrated on defining the extended correspondence relation and proving some properties about it. In the original paper it is defined as the greatest fixed point of a function. The formal definition is very similar. The only real difference is the style used to write the function. Here the style used is the same as was used to formalise the inference systems. In other words the new correspondence relation is a co-inductive relation defined by two rules. Making the formalisation consistent with the previous formalisations of inference systems, allows one to prove properties in a uniform way. It is for example easy to prove the function monotone using the same tactic as earlier. Worries that errors might have been introduced in

Ordinary Elimination

$$
\begin{array}{l}
[\![\ te \vdash e \Longrightarrow \tau; \\
\quad \bigwedge te\ c\ \tau.\ c\ \mathtt{isof}\ t \Longrightarrow p(te, \mathtt{e\_const}(c), \tau); \\
\quad \bigwedge te\ x.\ x \in \mathtt{te\_dom}(te) \Longrightarrow p(te, \mathtt{e\_var}(x), \mathtt{te\_app}(te, x)); \\
\quad \bigwedge te\ x\ e\ \tau_1\ \tau_2.\ te + \{x \mapsto \tau_1\} \vdash e \Longrightarrow \tau_2 \Longrightarrow p(te, \mathtt{fn}\ x \Rightarrow e, \tau_1 \to \tau_2); \\
\quad \bigwedge te\ f\ x\ e\ \tau_1\ \tau_2. \\
\qquad te + \{f \mapsto \tau_1 \to \tau_2\} + \{x \mapsto \tau_1\} \vdash e \Longrightarrow \tau_2 \Longrightarrow p(te, \mathtt{fix}\ f(x) = e, \tau_1 \to \tau_2); \\
\quad \bigwedge te\ e_1\ e_2\ \tau_1\ \tau_2. \\
\qquad [\![te \vdash e_1 \Longrightarrow \tau_1 \to \tau_2; te \vdash e_2 \Longrightarrow \tau_1]\!] \Longrightarrow p(te, e_1@e_2, \tau_2) \\
]\!] \Longrightarrow \\
p(te, e, t);
\end{array}
$$

Elimination for Each Expression

$$
\begin{array}{l}
te \vdash \mathtt{e\_const}(c) \Longrightarrow \tau \Longrightarrow c\ \mathtt{isof}\ \tau; \\
te \vdash \mathtt{e\_var}(x) \Longrightarrow \tau \Longrightarrow \tau = \mathtt{te\_app}(te, x) \wedge x \in \mathtt{te\_dom}(te); \\
\quad \vdots \\
te \vdash e_1@e_2 \Longrightarrow \tau_2 \Longrightarrow (\exists \tau_1.\ te \vdash e_1 \Longrightarrow \tau_1 \to \tau_2 \wedge te \vdash e_2 \Longrightarrow \tau_1);
\end{array}
$$

Figure 22: Elimination Rules

the reformulation is not important as long as basic consistency can be proved. The only purpose of the extended correspondence relation and consistency is to prove basic consistency. Basic consistency does not refer to the extended correspondence relation and does therefore not depend on the formulation of this relation. The Isabelle HOL formalisation is shown in figure 23.

From these definitions it is straightforward to derive introduction rules and elimination rules as it has been done earlier. More interestingly it is possible to derive the co-induction rules shown in figure 24.

Because co-induction is introduction there are of course two co-induction rules. These are based on the strong form of co-induction shown in figure 15. It is different from the form of co-induction used in [1], which is the weak form shown earlier. It turns out that the use of the strong form of co-induction shortens the proof, further backing the claim that this a useful formulation of co-induction.

Formalising the new correspondence relation is similar to formalising the inference systems and just as time consuming. The conclusion is of course that Isabelle should have automatic support for co-inductive definitions.

Now it is possible to state consistency in Isabelle HOL. The formulation of consistency given here differs from the original. The reason is that the formulation of consistency in the original is not suitable for doing a formal proof. For the proof to proceed smoothly it is necessary to reformulate it slightly as in figure 25. Basic consistency in figure 25 is translated directly from the original paper.

CONSTANTS

$\texttt{hasty\_fun} :: (\texttt{Val} * \texttt{Ty})\ \texttt{set} \Rightarrow (\texttt{Val} * \texttt{Ty})\ \texttt{set}$
$\texttt{hasty\_rel} :: "(\texttt{Val} * \texttt{Ty})\ \texttt{set}$
$(\_\ \texttt{hasty}\ \_) :: [\texttt{Val}, \texttt{Ty}] \Rightarrow \texttt{bool}$
$(\_\ \texttt{hasty\_env}\ \_) :: [\texttt{ValEnv}, \texttt{TyEnv}] \Rightarrow \texttt{bool}$

AXIOMS

$$
\begin{array}{l}
\texttt{hasty\_fun}(s) \equiv \\
\{\ p. \\
\quad (\exists c\ \tau.\ p = \langle \texttt{v\_const}(c), \tau \rangle \wedge c\ \texttt{isof}\ \tau) \vee \\
\quad (\ \exists x\ e\ ve\ \tau\ te. \\
\qquad p = \langle \texttt{v\_clos}(\langle x, e, ve \rangle), \tau \rangle \wedge \\
\qquad te \vdash \texttt{fn}\ x \Rightarrow e \Longrightarrow \tau \wedge \\
\qquad \texttt{ve\_dom}(ve) = \texttt{te\_dom}(te) \wedge \\
\qquad (\forall x_1 . x_1 \in \texttt{ve\_dom}(ve) \Rightarrow \langle \texttt{ve\_app}(ve, x_1), \texttt{te\_app}(te, x_1) \rangle \in s) \\
\quad) \\
\}; \\
\texttt{hasty\_rel} \equiv \texttt{gfp}(\texttt{hasty\_fun}); \\
v\ \texttt{hasty}\ \tau \equiv \langle v, \tau \rangle \in \texttt{hasty\_rel}; \\
ve\ \texttt{hasty\_env}\ te \equiv \\
\texttt{ve\_dom}(ve) = \texttt{te\_dom}(te) \wedge \\
(\forall x.\ x \in \texttt{ve\_dom}(ve) \Rightarrow \texttt{ve\_app}(ve, x)\ \texttt{hasty}\ \texttt{te\_app}(te, x));
\end{array}
$$

Figure 23: Extended Correspondence Relation

4.5.2 Proving Consistency

It turned out to be surprisingly easy to prove the consistency result. The proof proceeds more or less as the original proof.

The first step in the original proof was to use induction on the depth of the inference of evaluations. Here consistency is proved by induction on the structure of evaluations which is basically the same.

It is in connection with the application of induction that the only real difficulty of formalising the proof occur. Exactly what should the induction rule be applied to ? This is not obvious because the induction rule can be applied to almost everything.

The original formulation of consistency suggests to use the induction rule on $te \vdash e \Longrightarrow \tau \rightarrow v\ \texttt{hasty}\ \tau$. Attempting to prove consistency this way fails, because the induction hypothesises are too weak. This is the reason why consistency has been reformulated here. Besides rearranging the premises, τ and te have been explicitly quantified. Using the new formulation, consistency is proved by using the induction rule on $\forall \tau\ te.\ ve\ \texttt{hasty\_env}\ te \rightarrow te \vdash e \Longrightarrow \tau \rightarrow v\ \texttt{hasty}\ \tau$.

The above should not be seen as a problem of formalisation, but rather as a problem of proof. The original paper should state exactly what formula induction

$$
\begin{array}{l}
c\ \texttt{isof}\ \tau \Longrightarrow \langle \texttt{v\_const}(c), \tau\rangle \in \texttt{hasty\_rel};\\
[\![\ te \vdash \texttt{fn}\ x \Rightarrow e \Longrightarrow \tau;\\
\quad \texttt{ve\_dom}(ve) = \texttt{te\_dom}(te);\\
\quad \forall x_1.\\
\qquad x_1 \in \texttt{ve\_dom}(ve) \to\\
\qquad \langle \texttt{ve\_app}(ve, x_1), \texttt{te\_app}(te, x_1)\rangle \in \{\langle \texttt{v\_clos}(\langle x, e, ve\rangle), \tau\rangle\} \cup \texttt{hasty\_rel}\\
]\!] \Longrightarrow\\
\langle \texttt{v\_clos}(\langle x, e, ve\rangle), \tau\rangle \in \texttt{hasty\_rel};
\end{array}
$$

Figure 24: Co-induction Rules

Consistency

$$ve \vdash e \longrightarrow v \Longrightarrow (\forall \tau\ te.\ ve\ \texttt{hasty\_env}\ te \to te \vdash e \Longrightarrow \tau \longrightarrow v\ \texttt{hasty}\ \tau);$$

Basic Consistency

$$[\![ve\ \texttt{isof\_env}\ te;\ ve \vdash e \longrightarrow \texttt{v\_const}(c);\ te \vdash e \Longrightarrow \tau]\!] \Longrightarrow c\ \texttt{isof}\ \tau;$$

Figure 25: Consistency and Basic Consistency

should be applied to.

Having applied induction six cases remain to be proved, one for each of the rules of the dynamic semantics.

The first two cases, the ones for constants and variables, are claimed to be trivial in the original paper. Here they both have three lines proofs, of which only two lines are interesting. Both are proved by first using one of the elimination rules for elaborations and then an introduction rule for the extended correspondence relation or a call of a classical reasoning tactic.

In the original paper, they spend a little space on the third case, the one for abstraction. Here it seems just as trivial to prove as the first two. First an introduction rule for the extended correspondence relation is used, then a classical reasoning tactic.

The fourth case, the one for recursive functions, is the most interesting in that it uses co-induction. In the paper the proof takes up a little more than half a page. The formal proof is about 25 lines. The proof uses elimination on elaborations, some set theoretic reasoning, classical reasoning tactics etc. and of course co-induction. The stronger co-induction rule used here simplifies the proof, backing the claim that it is a useful formulation of co-induction.

The fifth case, the one for application of constants, is one of those claimed to be trivial in the original paper. Here it is however more complicated than the first three cases. It uses elimination on both elaborations and on the extended correspondence relation, as well as several calls of classical reasoning tactics. It also uses the requirement that the relation `isof` must be consistent with the function `apply`. Still the proof consists of less than 10 lines.

The last case, the one for application of closures, is the case that takes up most space in the original paper. Here it is shorter than the one for recursive functions. The proof uses elimination on elaborations and on `hasty`, classical reasoning tactics etc.

With the original proof guiding the formal proof, it was straightforward to carry out in Isabelle HOL. Filling in the necessary details required surprisingly little knowledge of how consistency actually was proved. It was a very positive experience.

4.6 Discussion

4.6.1 Inductive and Co-inductive Definitions

The case study considered in this paper illustrates in no uncertain manner how useful, especially inductive, but also co-inductive definitions are in computer science.

An estimated 4/5 of the work presented here is related to the formalisation of inductive and co-inductive definitions of relations and datatypes. In the case of relations, the Isabelle theory of least and greatest fixed points were used, while the datatypes were specified axiomatically. Even more work would have been required, if the formalisation of datatypes, had been based on the fixed point theory of Isabelle.

There is little doubt that it would be possible to mechanize the formalisation of inductive and co-inductive definitions in Isabelle. From a practical point of view, it is of course highly unsatisfactory that the bulk of work is concentrated on tedious and time consuming tasks, that could just as well be done automatically.

It can however be seen as a positive result, in the sense that it is very obvious how a huge improvement of the Isabelle system can be obtained. It is not every day one gets a chance to reduce the work load to approximately 1/5 by relatively simple means.

No matter how inductive and co-inductive definitions is automated it is not sufficient just to get the abstract properties of the inductive or co-inductive definition. There should also be support for doing proofs about inductively or co-inductively defined objects. Consider for example the present case study. It would have been useful if sets of rules for doing simple classical reasoning about inductive datatypes had been defined automatically.

An package for doing inductive and co-inductive definitions already exists for another of Isabelle's object-logics, the Zermelo-Fraenkel set theory. It is based on a theory of fixed points as above. It is planned to develop a similar package for Isabelle HOL.

4.6.2 Avoiding Co-induction

Although not really the subject here, it could be argued that there is no need for using co-induction to prove consistency.

It seems to be perfectly possible to do the consistency proof without using co-induction. One could work with a finite representation of the non-well-founded closures. At first the notion of co-inductive definitions and proofs might be overwhelming and this solution therefore seem compelling. Co-inductive definitions and proofs are however, perfectly natural and mechanically tractable notions. I therefore see no practical justification for using finite representations, if the possibility of using the more abstract notions of co-inductive definitions and proofs are present.

4.6.3 Working with Isabelle HOL

Disregarding the lack of inductive and co-inductive definitions, working with Isabelle HOL was a positive experience.

As already mentioned, doing the actual proof of consistency turned out to be surprisingly easy. The original proof could be used as an outline. From there on it was just a question of filling in a few details, a task which hardly required any knowledge of what was going on. Difficulties were only encountered when the original proof was not as clear as one could have wished, for example with respect to exactly what induction should be applied to. This must however be considered a problem of proof, not formalisation. Another remarkable fact is that the formal proof only takes up about the same space as the original proof. This contradicts, what seems to be the common conception, that formal proofs necessarily are long and much harder to do than corresponding informal proofs.

A nice feature of Isabelle is its tactics and tacticals. The classical reasoning tactics proved especially useful. The possibility of defining new tactics was only really exploited once, to write a tactic for proving functions corresponding to inference systems monotone. Writing good tactics is a difficult and time consuming job and instead of writing new tactics, one tends to use tactics already available. Their real potential seems to be when developing new theories which are intended to be used by others. Such theories should come with tactics for reasoning in the new theory.

5 Conclusion

The main result of the paper [1], consistency, has been proved formally in Isabelle HOL.

The notions of especially inductive but also co-inductive definitions turned out to be central in the formal treatment that leads to the consistency result. Inductive and co-inductive definitions of relations were formalised using a theory of least and greatest fixed points in Isabelle HOL, while an axiomatic specification was given of inductively and co-inductively defined datatypes.

An estimated 4/5 of the work was related to the formal treatment of inductive and co-inductive definitions. Even more work would have been required if the datatypes had been defined using the fixed point theory. From a practical point of view, it is of course unacceptable that far the most work is concentrated on a task which could be mechanized. It can however be seen as a positive result in the sense that it is obvious how a huge improvement of the Isabelle system can be obtained. Inductive and co-inductive definitions must be automated. It is planned to do so.

Doing the actual consistency proof was a very positive experience. It proceeded more or less as the original proof and hardly required any knowledge of how consistency originally was proved. Difficulties only arose, when the original proof was not as clear as one could wish. It seems that the hard part is to do the actual proof, not to formalise it. Furthermore it did only require about the same space as its more informal counterpart. This contradicts what seems to be the common conception, that formal proofs necessarily are long and much harder to do than the corresponding informal ones.

Because of the lack of automatic support for inductive and co-inductive definitions, the formalisation was not as easy and natural as one could wish. On the other hand, the formalisation of the actual consistency proof clearly demonstrated, that the Isabelle system has the potential.

Acknowledgements. Søren T. Heilmann and Niels B. Maretti commented on a draft of this paper. Lawrence C. Paulson suggested the project and commented on a draft of this paper. Mads Tofte answered questions about his and Robin Milner's paper.

References

[1] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.

[2] Lawrence C. Paulson. Co-induction and co-recursion in higher-order logic. Technical Report 304, University of Cambridge, Computer Laboratory, July 1993.

[3] Lawrence C. Paulson. Introduction to Isabelle. Technical Report 280, University of Cambridge, Computer Laboratory, January 1993.

[4] Lawrence C. Paulson. Isabelle's object-logics. Technical Report 286, University of Cambridge, Computer Laboratory, February 1993.

[5] Lawrence C. Paulson. The Isabelle reference manual. Technical Report 283, University of Cambridge, Computer Laboratory, February 1993.