A Case Study of Co-induction in Isabelle [1]

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

February 1995

[1]Supported by ESPRIT Basic Research Project 6453, Types for Proofs and Programs and the Danish Natural Science Research Council

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 generic theorem prover Isabelle.

Contents

List of Figures

1 Introduction

In the paper Co-induction in Relational Semantics [2], 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 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. A more interesting question is how easy and natural this can be done, in particular how well Isabelle can handle the notions of co-inductive definitions and co-induction. 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, this paper describes the formalisation of the consistency result in two of Isabelles object logics: Higher Order Logic (HOL) and Zermelo-Frankel Set Theory (ZF). Throughout this paper, the formal treatments will be compared to each other and to their more informal counterpart. The formal development in HOL and ZF was carried out with more than one year in between. The comparison should therefore not only illustrate differences and similarities between formal/informal treatment, the object logics ZF/HOL, but also how Isabelle have evolved in that year.

This paper is an extension of an earlier paper [1] only describing the formalisation of the consistency result in Isabelle HOL. As its predecessor, this paper is meant to be largely self contained. As a consequence a survey of the original paper [2] by Robin Milner and Mads Tofte is given first. For the same reason, it is followed by an overview of the Isabelle system and its implementation of HOL and ZF. In the following two sections the formalisation in Isabelle HOL and Isabelle ZF is discussed. Finally a few conclusions. The sections describing the original paper [2] and the formalisation in HOL are kept almost unchanged from the first paper. The overview of Isabelle is extended to describe the implementation of ZF. The section on the formalisation in ZF is new and the conclusion is changed to reflect the extra information obtained.

2 Co-induction in Relation Semantics

The aim of this section, is to give an overview of the parts of the paper [2] that must be formalised in order to prove the consistency result. For a more thorough treatment please refer to the original paper [2].

The original paper is concerned with proving the consistency of the dynamic and static semantics of a small functional programming language. As a consequence it first defines such a language. Then it defines the dynamic semantics, which associates values to expressions and the static semantics which associates types to expressions. Finally it states and proves 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 $f \in A \xrightarrow{\text{fin}} B$, $\text{dom}(f)$ denotes the domain and $\text{rng}(f)$ 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.

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

Figure 1: Constants, variables and expressions

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.

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.

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

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.

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.

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

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

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

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

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

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

Figure 3: Evaluation

$$\begin{array}{rcll} \tau & \in & \mathrm{Ty} ::= \{\mathrm{int}, \mathrm{bool}, \ldots\} \mid \mathrm{Ty} \to \mathrm{Ty} & (6) \\ te & \in & \mathrm{TyEnv} = \mathrm{Var} \xrightarrow{\mathrm{fin}} \mathrm{Ty} & (7) \\ \mathrm{isof} & \subseteq & \mathrm{Const} \times \mathrm{Ty} & (8) \\ \multicolumn{4}{l}{\text{If } c_1 \text{ isof } \tau_1 \to \tau_2 \text{ and } c_2 \text{ isof } \tau_1 \text{ then } \mathrm{apply}(c_1, c_2) \text{ isof } \tau_2} & (9) \end{array}$$

Figure 4: Type constants, 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.

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

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

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

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

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

Figure 5: Elaboration

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 manifest itself as too weak induction hypothesises.

Basic Consistency

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

Consistency

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

Figure 6: 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 [2] for a discussion of why this particular function was chosen.

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

Figure 7: Extended correspondence relation

The notion of greatest fixed point is defined in Figure 8. The corresponding principle of co-induction expresses, that in order to 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 f 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 [2], 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

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.

This section attempt to give an overview over Isabelle. At first, a brief overview of the Isabelle documentation is given. Then the notation is explained, followed by a description of the typed lambda calculus used by Isabelle. Next the pure Isabelle system is described and it is explained how the pure Isabelle system is extended to support reasoning in HOL and ZF.

3.1 Documentation

The Isabelle system is extensively documented. The main reference is the Isabelle Book [8]. This book contains most the information found in the three technical reports [9, 5, 4]. References will only be made to these reports if the information cannot be found in the Isabelle book. A number of other papers and reports discuss Isabelle and related issues and will be mentioned when necessary. Some information can only be found online in which case a URL address will be provided.

3.2 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.

3.3 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 $[\sigma_1, \ldots, \sigma_n] \Rightarrow \sigma$.

3.4 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.

3.4.1 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.

3.4.2 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 [8].

3.4.3 Proofs

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

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.

3.4.4 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.

3.5 Higher Order Logic

A number of logics have been implemented in Isabelle. 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 [8].

PREFIXES

$\neg :: \texttt{bool} \Rightarrow \texttt{bool}$	negation

INFIXES

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

BINDERS

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

TRANSLATIONS

$a \neq b \equiv \neg(a = b)$	not equal

Figure 10: Logical symbols in HOL

3.5.1 Basic HOL

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 connectives needed for this paper is declared in Figure 10.

The formulation of HOL in Isabelle, identifies meta-level and object-level types. 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 [3].

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.

3.5.2 HOL 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

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 in HOL set theory

theory can be found in [8].

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 [3]. Examples include natural numbers `nat`, disjoint unions + and products $*$.

3.5.3 (Co)Inductive definitions in HOL

A package for doing inductive and co-inductive definitions has been developed in Isabelle HOL [10, 9]. Unfortunately this package was not available when the consistency proof by Mads Tofte and Robin Milner was formalised in HOL. Instead a basic theory of least and greatest fixed points was used. A description can be found in §4.1 and [3].

Prefixes	
$\neg :: \mathtt{o} \Rightarrow \mathtt{o}$	negation
Infixes	
$= :: [\alpha, \alpha] \Rightarrow \mathtt{o}$	equality
$\wedge :: [\mathtt{o}, \mathtt{o}] \Rightarrow \mathtt{o}$	conjunction
$\vee :: [\mathtt{o}, \mathtt{o}] \Rightarrow \mathtt{o}$	disjunction
$\rightarrow :: [\mathtt{o}, \mathtt{o}] \Rightarrow \mathtt{o}$	implication
Binders	
$\forall :: [\alpha \Rightarrow \mathtt{o}] \Rightarrow \mathtt{o}$	universal quantification
$\exists :: [\alpha \Rightarrow \mathtt{o}] \Rightarrow \mathtt{o}$	existential quantification
Translations	
$a \neq b \equiv \neg(a = b)$	not equal

Figure 12: Logical symbols in ZF

3.6 Zermelo-Frankel Set Theory

A large part of Zermelo-Frankel Set Theory (ZF) has been developed within Isabelle. This will only be a brief description, but details can be found in the Isabelle Book [8].

3.6.1 Basic ZF

Isabelle ZF is an extension of Isabelle First Order Logic (FOL). ZF inherits the meta-type of first-order formulae `o` and the usual connectives from FOL. `o` lies in class `logic`, and there is an implicit coercion from `o` to meta-level truth values `prop` called `Trueprop`. Some of the connectives and their types can be found in Figure 12.

The ZF theory declares a new meta-type `i` of individuals, which belongs to the class of first order terms `term`. The class `term` is a subclass of `logic`. The syntax of FOL is extended with symbols for the usual constructs in ZF. The symbols needed for this paper appear in Figure 13

Most of the generic reasoning packages, such as the simplifier and the classical reasoning package, are instantiated to support reasoning in ZF.

Constants

$0 :: \mathtt{i}$	empty set
$\mathtt{cons} :: [\mathtt{i}, \mathtt{i}] \Rightarrow \mathtt{i}$	finite set constructor
$\mathtt{domain} :: \mathtt{i} \Rightarrow \mathtt{i}$	domain of a relation
$\mathtt{if} :: [\mathtt{o}, \mathtt{i}, \mathtt{i}] \Rightarrow \mathtt{i}$	conditional

Prefixes

$\bigcup :: \mathtt{i} \Rightarrow \mathtt{i}$	set union
$\bigcap :: \mathtt{i} \Rightarrow \mathtt{i}$	set intersection

Infixes

$\in :: [\mathtt{i}, \mathtt{i}] \Rightarrow \mathtt{o}$	membership
$\text{“} :: [\mathtt{i}, \mathtt{i}] \Rightarrow \mathtt{i}$	image of a relation
$\cup :: [\mathtt{i}, \mathtt{i}] \Rightarrow \mathtt{i}$	union
$\cap :: [\mathtt{i}, \mathtt{i}] \Rightarrow \mathtt{i}$	intersection

Translations

$\{a_1, \ldots, a_n\} \equiv \mathtt{cons}(a_1, \ldots, \mathtt{cons}(a_n, 0))$	finite set
$\bigcup_{x \in A} B(x) \equiv \bigcup\{B(x).x \in A\}$	gerneral union
$\bigcap_{x \in A} B(x) \equiv \bigcap\{B(x).x \in A\}$	gerneral intersection

Figure 13: Symbols in ZF

3.6.2 (Co)Inductive definitions in ZF

A package for doing inductive and co-inductive definitions has been developed within ZF. A description of the package and its theoretical foundations can be found in [7]. Further instruction on how to use the package can be found in §5 of this paper, where the package is applied to a number of realistic examples.

4 Formalisation in Isabelle HOL

This section describes how the contents of the original paper was formalised in Isabelle HOL in the summer 93. Isabelle HOL has evolved considerably since then. A major addition is that of an (co)inductive package [10]. Such a package would have improved the formalisation and this should be taken into account when reading the following. The formalisation in HOL as described below can be seen a strong argument for the development of a (co)inductive package.

The formalisation rests on a theory of least and greatest fixed points. This theory is described first. After this description 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 A theory Least and Greatest Fixed Points

A theory of least and greatest fixed points has been developed in Isabelle HOL [3]. 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 datatypes and inductive 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-datatypes and co-inductive 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 14 resemble usual set theoretic definitions.

	Least Fixed Points	Greatest Fixed Points
Constant	$\texttt{lfp} :: [\alpha\ \texttt{set} \Rightarrow \alpha\ \texttt{set}] \Rightarrow \alpha\ \texttt{set}$	$\texttt{gfp} :: [\alpha\ \texttt{set} \Rightarrow \alpha\ \texttt{set}] \Rightarrow \alpha\ \texttt{set}$
Axiom	$\texttt{lfp}(f) \equiv \bigcap\{s.f(s) \subseteq s\};$	$\texttt{gfp}(f) \equiv \bigcup\{s.s \subseteq f(s)\};$

Figure 14: Least and greatest fixed points in HOL

Most of the properties, derived from the definitions, can be characterised as either introduction or elimination rules. In Figure 15, 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.

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

Figure 15: Properties of least and greatest fixed points in HOL

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 $s \subseteq f(s)$.

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 surprising, 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 15 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 16 the most useful. Again the properties are symmetric.

	Least Fixed Points	Greatest Fixed Points
Co-induction		$[\![\ \texttt{mono}(f);$ $\quad s \subseteq f(s \cup \texttt{gfp}(f))$ $]\!] \Longrightarrow$ $s \subseteq \texttt{gfp}(f);$
Induction	$[\![\ \texttt{mono}(f);$ $\quad f(s \cap \texttt{lfp}(f)) \subseteq s$ $]\!] \Longrightarrow$ $\texttt{lfp}(f) \subseteq s;$	

Figure 16: Properties of least and greatest fixed points in HOL

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.

	Least Fixed Points	Greatest Fixed Points
Co-induction		$[\![\ \texttt{mono}(f);$ $\quad x \in f(\{x\} \cup \texttt{gfp}(f))$ $]\!] \Longrightarrow$ $x \in \texttt{gfp}(f);$
Induction	$[\![\ \texttt{mono}(f);$ $\quad x \in \texttt{lfp}(f);$ $\quad \bigwedge y.$ $\quad\quad y \in f(\texttt{lfp}(f) \cap \{z.p(z)\}) \Longrightarrow$ $\quad\quad p(y)$ $]\!] \Longrightarrow$ $p(x);$	

Figure 17: Properties of least and greatest fixed points in HOL

The only rules shown in Figure 17 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 16 and 17.

4.2 The Language

Expressions, defined by the BNF in Figure 1, can be formally expressed in Isabelle HOL as a 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 [3].

The solution adapted here and in the following is to give an axiomatic specification of 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 datatypes has been investigated elsewhere and that axiomatisation of datatypes is well understood, it seems a sensible choice.

A standard axiomatisation of expressions is shown in Figure 18. 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 a 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. A much better solution from a practical point of view is to use a (co)inductive package like that available for Isabelle HOL today. It will automatically provide the necessary properties, given the constructors and their types. Unfortunately this package was not available at the time of formalisation.

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

TYPES

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

CONSTANTS

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

INJECTIVENESS AXIOMS

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

DISTINCTNESS AXIOMS

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

INDUCTION AXIOM

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

Figure 18: Constants, variables and expressions in HOL

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 19: Values, value environments and closures in HOL

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 19. It must however be considered a lacking feature of Isabelle HOL that there is no automatic support for mutually recursive co-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 20.

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

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

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

Figure 20: Evaluation in HOL

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.

The formalisation of the dynamic semantics must be correct. Given the definitions Figure 20, 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 21.

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.

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 22 and Figure 23.

The type of types is another example of a construct that could be formalised as a 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 24.

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 25.

INTRODUCTION

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

INDUCTION

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

Figure 21: Properties of evaluation in HOL

Types

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

Constants

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

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

Axioms

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

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

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

Figure 22: Type constants, types and type environments in HOL

CONSTANTS

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

AXIOMS

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

Figure 23: Basic correspondence relation in HOL

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.

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 the reformulation is not important as long as basic

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

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

Figure 24: Elaboration in HOL

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 26.

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 27.

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 17. It is different from the form of co-induction used in [2], 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 is 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, as provided

Ordinary Elimination

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

Elimination for Each Expression

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

Figure 25: Properties of elaboration in HOL

by the (co)inductive package which now exists.

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 28. Basic consistency in Figure 28 is translated directly from the original paper.

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 anything.

The original formulation of consistency suggests to use the induction rule on $te \vdash e \Longrightarrow \tau \to v\ \texttt{hasty}\ \tau$. Attempting to prove consistency this way

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

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

Figure 26: Extended correspondence relation in HOL

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 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.

$$c\ \texttt{isof}\ \tau \Longrightarrow \langle \texttt{v\_const}(c), \tau\rangle \in \texttt{hasty\_rel};$$

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

Figure 27: Co-induction rules for `hasty_rel` in HOL

Consistency

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

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 28: Consistency in HOL

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 here 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.

The above clearly shows the need for a (co)inductive package for Isabelle HOL. 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.

A (co)inductive package should not only provide the obvious abstract properties for (co)inductive definitions. It should also support reasoning about (co)inductively defined objects. Consider for example the present case study. It would have been useful if special elimination rules like those in Figure 25 had been derived automatically. It would also be useful if the package supported simple classical reasoning about the (co)inductively defined constructs.

Although not available at the time of formalisation, a (co)inductive package for Isabelle HOL now exists [10]. It is derived from the corresponding package for Isabelle ZF, but does not support co-datatypes. Using this package it should be possible to improve the formalisation described above considerably.

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, at the time of formalisation, 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 Formalisation in Isabelle ZF

At first sight it might seem a vain undertaking to carry of the same formal development in ZF as has already been carried out in HOL. There are however a number of reasons for doing it.

ZF and HOL are different logics. The development in ZF provide an opportunity to study some of the consequences of these differences, in particular the consequences with regard to non-well-founded objects.

The formalisation of the consistency result in ZF turned out to be one realistic (co)inductive definition after another. In other words it is a good example for testing the (co)inductive package in ZF. Testing should be understood in a broader sense, covering not only the practical aspect of testing for bugs, but also the actual design of the package.

The formal development in HOL was carried out without the use of a (co)inductive package. It was claimed that a (co)inductive package would be a huge improvement and estimates of the resulting reductions in workload was given. A

similar development in ZF make it possible to verify the claim and substantiate the estimates further, or possibly reject the claim.

The structure of this section follows that of §2 and §4.

5.1 The Language

It is easy to define the set of expressions in Isabelle ZF. The BNF in Figure 1, is formalised using the datatype declaration provided by the (co)inductive package. A datatype is one example of an inductively defined set in ZF. The formal definition appears in Figure 29. Although the definition look very much like a similar definition in SML, there are a few things to explain.

CONSTANTS

```
consts
   Const :: i
rules
   constU   c ∈ Const ⟹ c ∈ univ(A)
   constNEE   c ∈ Const ⟹ c ≠ 0
```

VARIABLES

```
consts
   ExVar :: i
rules
   exvarU   x ∈ ExVar ⟹ x ∈ univ(A)
```

EXPRESSIONS

```
consts
   Exp :: i
datatype
   Exp = e_const(c ∈ Const) |
         e_var(x ∈ ExVar) |
         e_fn(x ∈ ExVar, e ∈ Exp) |
         e_fix(x1 ∈ ExVar, x2 ∈ ExVar, e ∈ Exp) |
         e_app(e1 ∈ Exp, e2 ∈ Exp)
   type_intrs   [constU, exvarU]
```

Figure 29: Constants, variables and expressions in ZF

Russell's paradox shows that too big sets makes ZF inconsistent. The (co)inductive package must prove that the definition does not yield a set that is too big. This can be seen as type-checking the definition and is done by proving that all of the introduction rules are closed under some existing set. Such a set is called a bounding set. The set defined is a subset of the bounding set.

The standard bounding set for a datatype definition with parameters $A_1, \ldots, A_n$ is $\texttt{univ}(A_1 \cup \ldots \cup A_n)$. This set contains $A_1 \cup \ldots \cup A_n$, and is closed under paring, left and right injections; the constructs used to define the datatype. The bounding set for the definition of `Exp` is $\texttt{univ}(0)$ because the definition has no parameters. To type-check the definition of `Exp`, all of the rules must be closed under $\texttt{univ}(0)$. For example, to prove that the rule for function abstraction is closed, the package must prove:

$$[\![x \in \texttt{ExVar}; e \in \texttt{univ}(0)]\!] \Longrightarrow \texttt{e\_fn}(x, e) \in \texttt{univ}(0)$$

The extra rules needed by the package to type-check a definition is given in the `type_intrs` ... part of the definition. In order to type-check the definition of `Exp`, the unspecified sets `ExVar` and `Const` must both be subsets of the bounding set $\texttt{univ}(0)$. This explains the role of the two axioms `constU` and `exvarU` in Figure 29. The presence of the axiom `constNEE` is explained in §5.2.2.

Given the definition in Figure 29 the package defines the set `Exp` and all the constructors. It will also derive the usual rules: introduction, elimination, induction, injectiveness and distinctness. It does however not relate the defined set to an abstract set by two isomorphism functions, as discussed in section 4.2. As a consequence, elements of different datatypes may be identical.

An alternative approach is to parametrise the definition of `Exp` by two sets corresponding to `Const` and `ExVar`. They would then automatically be included in the bounding set. It is difficult to say which solution is the best, as the latter method would require other sets depending on `Exp` to be parametrised as well.

The amount of work done by the package is substantial. Because all datatypes was axiomatised in HOL, it is not possible to do a direct comparison. It is however clear that the package is a huge improvement; the reason for axiomatising datatypes in HOL was the amount of work required if they where to be defined by hand. Now the definition in ZF takes up noticeable less space than the axiomatic specification given in HOL and are closer to the original specification.

5.2 Dynamic Semantics

Before the notion of values are formalised, a notion of variant maps are defined. After that the section follows the usual pattern.

5.2.1 Variant Maps

Sets in ZF must be well-founded by the foundation axiom:

$$A = 0 \vee (\exists x \in A. \forall y \in x. y \notin A)$$

The foundation axiom outlaws infinite descents under the membership relation. Take, for example an equation like $a = \{a\}$. Any solution to this must be non-

well-founded, but it can clearly not have any solution in ZF as it would lead to an infinite decent:

$$\ldots \in \{\ldots\} \in \{\{\ldots\}\}$$

Another example is infinite lists encoded using the standard notion of pair in ZF:

$$\langle a, b\rangle \equiv \{\{a\}, \{a, b\}\}$$

The infinite list with elements $a_1, a_2 \ldots$ would be:

$$\langle a_1, \langle a_2, \ldots\rangle\rangle = \{\{a_1\}, \{a_1, \{\{a_2\}, \{a_2, \ldots\}\}\}\}$$

Such an infinite list cannot be a set in ZF because there exists an infinite decent:

$$\ldots \in \{a_2, \ldots\} \in \langle a_2, \ldots\rangle \in \{a_1, \langle a_2, \ldots\rangle\} \in \langle a_1, \langle a_2, \ldots\rangle\rangle$$

The above might give the impression that co-datatypes are of very limited use in ZF. If encoded using ordinary pairs the set of lazy lists would for example be exactly the same as the set of finite lists. Fortunately it is possible to circumvent the problem by defining a new notion of pairs; the variant pair [6].

Keeping in mind, that everything is a set in ZF, the variant notion of pair is defined as:

$$\langle a; b\rangle \equiv (\{0\} \times a) \cup (\{1\} \times b) = a + b$$

Using the new notion of pair to encode lists, the infinite list with elements $a_1 = \{a_{11}, \ldots, a_{1m}\}, a_2 = \{a_{21}, \ldots, a_{2n}\}, \ldots$ is:

$$\langle a_1; \langle a_2; \ldots\rangle\rangle = \{\langle 0, a_{11}\rangle, \ldots \langle 0, a_{1m}\rangle, \langle 1, \langle 0, a_{21}\rangle\rangle, \ldots \langle 1, \langle 0, a_{2n}\rangle\rangle, \ldots\}$$

As it appears there are no infinite descents, when using the variant representation of pair, although the list is infinite.

When doing (co)inductive definitions such as co-datatype definitions, it is crucial to use the right representations to avoid forcing sets to be well-founded. The (co)inductive package is therefore based on a theory of variant pairs, and uses them when necessary [7]. This does however not free the user of the package from being cautious.

In the next section value environments are going to be represented as what is basically functions. The standard representation of functions in ZF is as sets of pairs. Value environments will be defined as a part of a mutual co-datatype definition. Using the standard function space in ZF to represent value environments, would lead to a problem similar to the problem of infinite lists above. To overcome this problem the notion of variant maps (functions) is introduced.

A variant map is a generalisation of a variant pair. A variant pair is a sum (a+b) and a variant map is a general sum ($\Sigma_{x\in a}.b_x$) [6]. The set of all variant maps can easily be defined directly from the set of standard maps by converting

each standard map into a variant map. Unfortunately such a definition makes reasoning hard. An alternative approach, where a slightly too big set is restricted to the desired set, has therefore been adopted instead. The formal definition of the set $\texttt{TMap}(A, B)$ of total maps from A to B and the set of partial maps $\texttt{Map}(A, B)$ from A to B, as well as some of the associated operations can be found in Figure 30.

$$
\begin{array}{l}
\texttt{consts} \\
\quad \texttt{TMap} :: [\texttt{i}, \texttt{i}] \Rightarrow \texttt{i} \\
\quad \texttt{Map} :: [\texttt{i}, \texttt{i}] \Rightarrow \texttt{i} \\
\texttt{rules} \\
\quad \texttt{TMap\_def} \quad \texttt{TMap}(A, B) \equiv \{m \in \texttt{Pow}(A \times \bigcup B).\forall a \in A.m\text{“}\{a\} \in B\} \\
\quad \texttt{Map\_def} \quad \texttt{Map}(A, B) \equiv \texttt{TMap}(A, \texttt{cons}(0, B)) \\
\texttt{consts} \\
\quad \texttt{map\_emp} :: \texttt{i} \\
\quad \texttt{map\_owr} :: [\texttt{i}, \texttt{i}, \texttt{i}] \Rightarrow \texttt{i} \\
\quad \texttt{map\_app} :: [\texttt{i}, \texttt{i}] \Rightarrow \texttt{i} \\
\texttt{rules} \\
\quad \texttt{map\_emp\_def} \quad \texttt{map\_emp} \equiv 0 \\
\quad \texttt{map\_owr\_def} \quad \texttt{map\_owr}(m, a, b) \equiv \Sigma_{x \in \{a\} \cup \texttt{domain}(m)} \texttt{if}(x = a, b, m\text{“}\{x\}) \\
\quad \texttt{map\_app\_def} \quad \texttt{map\_app}(m, a) \equiv m\text{“}\{a\}
\end{array}
$$

Figure 30: Variant maps in ZF

The set $\texttt{Pow}(A \times \bigcup B)$ is too big to be $\texttt{TMap}(A, B)$ because an element of A could be mapped to a mix of elements from B. It is easy to see that this mix do not necessarily belong to B itself. If for example B is the set $\{\{b_1\}, \{b_2\}\}$, an element of A might be mapped to $\{b_1, b_2\}$, which clearly is not a member of B. The predicate $\forall a \in A.m\text{“}\{a\} \in B$ therefore requires that all elements in A is mapped to an element in B. To see this, simply view a map as a relation on A and $\bigcup B$. Application then becomes the image of a singleton set.

It is interesting to see what happens to $\texttt{TMap}(A, B)$ if B contains the empty set 0. In this case total maps effectively becomes partial maps, as it is impossible to tell whether an element of A is not in the domain of the map or just mapped to 0. This fact is exploited in defining the set of partial maps. The set of partial maps is the corresponding set of total maps, with the empty set added to the range. It is easy to define the usual operations on maps: domain, overwriting and application. All the definitions, except for the definition of domain appear in Figure 30. The domain of a map is simply the domain (`domain`) of the corresponding relation.

Proving the necessary properties about variant maps required some effort. This is not very surprising as the theory of variant maps is new. With a well

developed theory of variant notions, only a little effort would have been required.

The properties proved is similar to those one would expect for maps based on the ordinary notion of pairs. The main difference is that it is necessary to be careful about empty sets as elements. An example is the following theorem:

$$b \neq 0 \Longrightarrow \texttt{domain}(\texttt{map\_owr}(m, a, b)) = \{a\} \cup \texttt{domain}(m)$$

Here it is necessary to ensure that b is not the empty set, because if this where the case, the modification of m would have no effect.

5.2.2 Values and Value Environments

Closures can be non-well-founded. As a consequence, a co-datatype declaration is used to formalise the notion of values, value environments and closures. Using a datatype declaration would only allow well-founded closures. The formal definition appear in Figure 31.

In order to simplify matters, no separate set for closures is defined. The set of closures has been eliminated by substituting the right hand side of (3) for `Clos` in (1). The two remaining sets `Val` and `ValEnv` are defined as a mutual co-datatype. The default name for the combined set of values and value environments defined by the package is `Val_ValEnv`.

There are two constructors for values, one for constants and one for closures. A value environment is basically a variant map and consequently only one constructor exists. As mentioned in the previous section, it is not possible to use the ordinary function space to formalise the notion of value environments, as this would force all closures to be well-founded.

The default bounding set for a co-datatype with parameters $A_1, \ldots, A_n$ is $\texttt{quniv}(A_1 \cup \ldots \cup A_n)$, a superset of $\texttt{univ}(A_1 \cup \ldots \cup A_n)$, which is also closed under the variant constructions used by the package in (co)datatype definitions. The rules in the `type_intrs` ... part of the definition enable the package to prove that the rules are closed under $\texttt{quniv}(0)$. The rule for variant maps `mapQU` is however not as one might expect:

$$[\![m \in \texttt{PMap}(A, \texttt{quniv}(B)); \bigwedge x.x \in A \Longrightarrow x \in \texttt{univ}(B)]\!] \Longrightarrow m \in \texttt{quniv}(B)$$

In this case it requires `ExVar` to be a subset of $\texttt{univ}(0)$ instead of $\texttt{quniv}(0)$. It is therefore also necessary to include `exvarU` in the list of rules supplied to the package.

The operations on value environments are defined in terms of the case analysis operator provided by the package and the operators on maps. A separate case analysis operator for each of the mutually defined sets would make the definitions smaller and more readable.

The necessary properties about values and value environments are proved using properties of maps and the rules derived by the package. The (co)inductive

```
consts
   Val :: i    ValEnv :: i    Val_ValEnv :: i
codatatype
   Val = v_const(c ∈ Const) |
         v_clos(x ∈ ExVar, e ∈ Exp, ve ∈ ValEnv) and
   ValEnv = ve_mk(m ∈ Map(ExVar, Val))
   monos [map_mono]
   type_intrs [constQU, exvarQU, exvarU, expQU, mapQU]

consts
   ve_emp :: i
   ve_owr :: [i, i, i] ⇒ i
   ve_dom :: i ⇒ i
   ve_app :: [i, i] ⇒ i
rules
   ve_emp_def
      ve_emp ≡ ve_mk(map_emp)
   ve_owr_def
      ve_owr(ve, x, v) ≡
      ve_mk(Val_ValEnv_case(λx.0, λx y z.0, λm.map_owr(m, x, v), ve))
   ve_dom_def
      ve_dom(ve) ≡ Val_ValEnv_case(λx.0, λx y z.0, λm.domain(m), ve)
   ve_app_def
      ve_app(ve, a) ≡
      Val_ValEnv_case(λx.0, λxyz.0, λm.map_app(m, a), ve)
```

Figure 31: Values and value environments in ZF

package automatically proves most of the properties needed, but it would have been useful if the function `mk_cases` could have been used to derive specialised elimination rules for each of the two sets `Val` and `ValEnv`. The problem of having empty sets in the range of maps shows up here again. In order to prove consistency later it is necessary to prove that no value is the empty set. A consequence is that the set of constants must not contain the empty set as an element. This explain the role of the axiom `constNEE` in Figure 29.

Because the notion of values, value environments and closure was axiomatised in HOL it is again difficult to do a direct comparison. It is however clear that much more work would have been required in HOL, if the formalisation was done in the same way as in ZF, but without a (co)inductive package. If the formalisation was carried in HOL using the (co)inductive package, ZF would have a small handicap, because of the way it treats non-well-founded constructions and because type-constraints must be handled explicitly.

As an alternative to using variant maps, value environments could be lazy lists. Although the definition of the set of values and the set of value environments would be easier, it would be harder to define the operations on maps, and maybe also to reason about those. I cannot say which solution is the best.

5.2.3 Evaluation

The original inference system constitutes an inductive definition of a evaluation relation between environments, expressions and values; a set of triples. It is easily formalised in ZF as an inductive definition using the (co)inductive package. All it takes is changing the syntax and adding a few things such as type constraints and the rules needed for showing that the rules are closed under the bounding set.

In Figure 32, the `domains`... part of the definition state the name of the relation (`EvalRel`) and the bounding set (`ValEnv` $\times$ `Exp` $\times$ `Val`). The set defined must be a subset of the bounding set. The rules given in the `type_intrs`... part allow the package to prove that the rules are closed under the bounding set. In contrast to (co)datatypes these rules are no longer concerned with `univ`(...) and `quniv`(...), because the bounding set is no longer one of these.

The size of the definitions in HOL and ZF are similar. However, much less needs to be proved in ZF. The proofs needed to derive the necessary rules was around 125 lines in HOL, while none where needed in ZF.

5.3 Static Semantics

The (co)inductive package makes it very easy to formalise the static semantics. Before the notion of elaboration is formalised a notion of type, type environments and correspondence of constants and types are formalised.

```
consts
  EvalRel :: i
inductive
  domains EvalRel ⊆ ValEnv × Exp × Val
  intrs
    eval_constI
      ⟦ve ∈ ValEnv; c ∈ Const⟧ ⟹
      ⟨ve, e_const(c), v_const(c)⟩ ∈ EvalRel
    eval_varI
      ⟦ve ∈ ValEnv; x ∈ ExVar; x ∈ ve_dom(ve)⟧ ⟹
      ⟨ve, e_var(x), ve_app(ve, x)⟩ ∈ EvalRel
    eval_fnI
      ⟦ve ∈ ValEnv; x ∈ ExVar; e ∈ Exp⟧ ⟹
      ⟨ve, e_fn(x, e), v_clos(x, e, ve)⟩ ∈ EvalRel
    eval_fixI
      ⟦ve ∈ ValEnv; x ∈ ExVar; e ∈ Exp; f ∈ ExVar; cl ∈ Val;
       v_clos(x, e, ve_owr(ve, f, cl)) = cl
      ⟧ ⟹
      ⟨ve, e_fix(f, x, e), cl⟩ ∈ EvalRel
    eval_appI1
      ⟦ve ∈ ValEnv; e1 ∈ Exp; e2 ∈ Exp; c1 ∈ Const; c2 ∈ Const;
       ⟨ve, e1, v_const(c1)⟩ ∈ EvalRel;
       ⟨ve, e2, v_const(c2)⟩ ∈ EvalRel
      ⟧ ⟹
      ⟨ve, e_app(e1, e2), v_const(c_app(c1, c2))⟩ ∈ EvalRel
    eval_appI2
      ⟦ve ∈ ValEnv; ve' ∈ ValEnv; e1 ∈ Exp;
       e2 ∈ Exp; e' ∈ Exp; x' ∈ ExVar; v ∈ Val;
       ⟨ve, e1, v_clos(x', e', ve')⟩ ∈ EvalRel;
       ⟨ve, e2, v2⟩ ∈ EvalRel;
       ⟨ve_owr(ve', x', v2), e', v⟩ ∈ EvalRel
      ⟧ ⟹
      ⟨ve, e_app(e1, e2), v⟩ ∈ EvalRel
    type_intrs
      c_appI :: ve_appI :: ve_empI :: ve_owrI ::
      Exp.intrs@Val_ValEnv.intrs
```

Figure 32: Evaluation in ZF

5.3.1 Types and Type Environments

Types and type environments can be seen as datatypes and are formalised using the datatype declaration of the (co)inductive package. Type environments are formalised as lists of pairs of variables and types. Alternatively, they could have been defined as partial functions from variables to types. The definitions appear in Figure 33.

The purpose of `exvarU`, `tyconstU` and `dom_subset RS subsetD`. is to allow the package to type-check the definitions, just as it was the case for expressions. The latter of these `dom_subset RS subsetD`, is actually a proof of the theorem:

$$t \in \texttt{Ty} \Longrightarrow t \in \texttt{univ}(0)$$

The operations `te_dom` and `te_app` are defined by recursion on the structure of environments. A package for doing recursive definition might make the definitions smaller and more readable.

The (co)inductive package derives all the necessary rules, except for those concerned with `te_dom` and `te_app`. To prove the necessary theorems about these requires a little effort. Again a recursion package might be useful.

Compared to the axiomatic approach used in HOL, doing a proper datatype definition in ZF only requires some extra proof effort, related to the operators `te_dom` and `te_app`. Again, the development using the (co)inductive package must be expected to be far less time consuming than one done without the assistance of such a package.

5.3.2 Basic Correspondence Relation

The basic correspondence relation `isof` is treated just as it was in HOL. The necessary definitions can be found in Figure 34.

5.3.3 Elaboration

The inference system for the static semantics, can be seen as an inductive definition of an elaboration relation, ie. a set of tipples. As it was the case for evaluations, turning such an inference system, into an inductive definition in ZF using the (co)inductive package, is basically just a question of changing the syntax. The result can be seen in Figure 35.

The `domains`... part of the definition states what the bounding set is, in other words that the relation is a subset of $\texttt{ValEnv} \times \texttt{Exp} \times \texttt{Ty}$. The rules in the `type_intrs`... reflect this, in the sense that they allow the package to prove that all the rules are closed under the bounding set.

The package directly proves most of the rules needed: introduction rules, elimination rule and induction rule. Specialised elimination rules, similar to those

TYPE CONSTANTS

consts
  TyConst :: i
rules
  tyconstU $tc \in \texttt{TyConst} \Longrightarrow tc \in \texttt{univ}(A)$

TYPES

consts
  Ty :: i
datatype
  $\texttt{Ty} = \texttt{t\_const}(tc \in \texttt{TyConst}) \mid \texttt{t\_fun}(t_1 \in \texttt{Ty}, t_2 \in \texttt{Ty})$
  type_intrs [tyconstU]

TYPE ENVIRONMENTS

consts
  TyEnv :: i
datatype
  $\texttt{TyEnv} = \texttt{te\_emp} \mid \texttt{te\_owr}(te \in \texttt{TyEnv}, x \in \texttt{ExVar}, t \in \texttt{Ty})$
  type_intrs [exvarU, Ty.dom_subset RS subsetD]

consts
  te_rec :: [i, i, [i, i, i, i] ⇒ i] ⇒ i
rules
  te_rec_def
    $\texttt{te\_rec}(te, c, h) \equiv$
    $\texttt{Vrec}(te, \lambda te\ g.\texttt{TyEnv\_case}(c, \lambda te'\ x\ t.h(te', x, t, g`te'), te))$

consts
  te_dom :: i ⇒ i
  te_app :: [i, i] ⇒ i
rules
  te_dom_def $\texttt{te\_dom}(te) \equiv \texttt{te\_rec}(te, 0, \lambda te\ x\ t\ r.r \cup \{x\})$
  te_app_def $\texttt{te\_app}(te, x) \equiv \texttt{te\_rec}(te, 0, \lambda te\ y\ t\ r.\texttt{if}(x = y, t, r))$

Figure 33: Type constants, types and type environments in ZF

```
consts
   isof :: [i, i] ⇒ o
rules
   isof_app
      ⟦isof(c1, t_fun(t1, t2)); isof(c2, t1)⟧ ⟹ isof(c_app(c1, c2), t2)
   isofenv_def
      isofenv(ve, te) ≡
      ve_dom(ve) = te_dom(te)∧
      (∀x ∈ ve_dom(ve).
       (∃c. ve_app(ve, x) = v_const(c) ∧ isof(c, te_app(te, x)))
      )
```

Figure 34: Basic correspondence relation in ZF

mentioned in §4.4, are generated using the function `mk_cases` provided by the package. For example the elimination rule for constant expressions:

$$\begin{array}{l}\llbracket \langle te, \texttt{e\_const}(c), t\rangle \in \texttt{ElabRel};\\ \quad \llbracket \texttt{isof}(c, t); t \in \texttt{Ty}; c \in \texttt{Const}; te \in \texttt{TyEnv}\rrbracket \Longrightarrow Q\\ \rrbracket \Longrightarrow Q\end{array}$$

is produced by the following:

```
ElabRel.mk_cases Exp.con_defs ⟨te, e_const(c), t⟩ ∈ ElabRel
```

While the actual definitions in HOL and ZF takes up more or less the same amount of space, a huge improvement is fund when it comes to proving the necessary rules. The proofs in HOL was approximately 200 lines while they are down to approximately 15 lines in ZF.

5.4 Consistency

5.4.1 Stating Consistency

The basic setup is the same as in the original paper and HOL. In order to prove basic consistency a stronger consistency result must be proved. The stronger result is formulated using the extended correspondence relation.

It is in the treatment of the extended correspondence relation, that the only difference to the formalisation in HOL is to be found. In HOL the extended correspondence relation was defined directly as the greatest fixed point of a function. This time it is defined using the (co)inductive package. Figure 36 show the co-inductive definition.

The resulting set must be a subset of `Val` × `Ty` because the extended correspondence relation relates values and types. This set is defined co-inductively by

```
consts
   ElabRel :: i
inductive
   domains ElabRel ⊆ TyEnv × Exp × Ty
   intrs
      elab_constI
         ⟦te ∈ TyEnv; c ∈ Const; t ∈ Ty; isof(c, t)⟧ ⟹
         ⟨te, e_const(c), t⟩ ∈ ElabRel
      elab_varI
         ⟦te ∈ TyEnv; x ∈ ExVar; x ∈ te_dom(te)⟧ ⟹
         ⟨te, e_var(x), te_app(te, x)⟩ ∈ ElabRel
      elab_fnI
         ⟦te ∈ TyEnv; x ∈ ExVar; e ∈ Exp; t1 ∈ Ty; t2 ∈ Ty;
          ⟨te_owr(te, x, t1), e, t2⟩ ∈ ElabRel
         ⟧ ⟹
         ⟨te, e_fn(x, e), t_fun(t1, t2)⟩ ∈ ElabRel
      elab_fixI
         ⟦te ∈ TyEnv; f ∈ ExVar; x ∈ ExVar; t1 ∈ Ty; t2 ∈ Ty;
          ⟨te_owr(te_owr(te, f, t_fun(t1, t2)), x, t1), e, t2⟩ ∈ ElabRel
         ⟧ ⟹
         te, e_fix(f, x, e), t_fun(t1, t2)⟩ ∈ ElabRel
      elab_appI
         ⟦te ∈ TyEnv; e1 ∈ Exp; e2 ∈ Exp; t1 ∈ Ty; t2 ∈ Ty;
          ⟨te, e1, t_fun(t1, t2)⟩ ∈ ElabRel;
          ⟨te, e2, t1⟩ ∈ ElabRel
         ⟧ ⟹
         ⟨te, e_app(e1, e2), t2⟩ ∈ ElabRel
   type_intrs te_appI :: Exp.intrs@Ty.intrs
```

Figure 35: Elaboration in ZF

```
consts
   HasTyRel :: i
coinductive
   domains HasTyRel ⊆ Val × Ty
   intrs
      htr_constI
         ⟦c ∈ Const; t ∈ Ty; isof(c, t)⟧ ⟹
         ⟨v_const(c), t⟩ ∈ HasTyRel
      htr_closI
         ⟦x ∈ ExVar; e ∈ Exp; t ∈ Ty; ve ∈ ValEnv; te ∈ TyEnv;
          ⟨te, e_fn(x, e), t⟩ ∈ ElabRel;
          ve_dom(ve) = te_dom(te);
          {⟨ve_app(ve, y), te_app(te, y)⟩.y ∈ ve_dom(ve)} ∈ Pow(HasTyRel)
         ⟧ ⟹
         ⟨v_clos(x, e, ve), t⟩ ∈ HasTyRel
   monos [Pow_mono]
   type_intrs Val_ValEnv.intrs

consts
   hastyenv :: [i, i] ⇒ o
rules
   hasty_env_def
      hastyenv(ve, te) ≡
      ve_dom(ve) = te_dom(te)∧
      (∀x ∈ ve_dom(ve).⟨ve_app(ve, x), te_app(te, x)⟩ ∈ HasTyRel)
```

Figure 36: Extented correspondence relation in ZF

two rules derived directly from the definition given in the original paper. The last premise in the rule for closures (`htr_closI`) is just another way of writing:

$$\forall y \in \texttt{ve\_dom}(ve).\langle \texttt{ve\_app}(ve, y), \texttt{te\_app}(te, y)\rangle \in \texttt{HasTyRel}$$

It was necessary to rewrite the premise, because all premises that mention the recursive set must have the form $t \in M(R)$, where t is any term, M is a monotone operator on sets and R the set under construction. `Pow_mono` is included in the `monos`... part of the definition because of the monotonicity requirement on M. The rules needed to type-check the definition are the introduction rules for values and value environments: `Val_ValEnv.intrs`. The package derives all the necessary rules: introduction rules, elimination rules and the co-induction rule. Finally the pointwise extension to environments `hastyenv` is defined.

The formulation of consistency and basic consistency is shown in Figure 37 and is exactly the same as in HOL except for the syntax used and the explicit type constraints.

Consistency

$$\langle ve, e, v\rangle \in \texttt{EvalRel} \Longrightarrow (\forall t\ te.\texttt{hastyenv}(ve, te) \longrightarrow \langle te, e, t\rangle \in \texttt{ElabRel} \longrightarrow \langle v, t\rangle \in \texttt{HasTyRel})$$

Basic Consistency

$$[\![\ ve \in \texttt{ValEnv}; te \in \texttt{TyEnv};\ \texttt{isofenv}(ve, te);\ \langle ve, e, \texttt{v\_const}(c)\rangle \in \texttt{EvalRel};\ \langle te, e, t\rangle \in \texttt{ElabRel}\ ]\!] \Longrightarrow \texttt{isof}(c, t)$$

Figure 37: Consistency in ZF

5.4.2 Proving Consistency

The differences between the formal proof of consistency in HOL and ZF are small. In fact large parts of the proof was mainly done by cutting and pasting from the proof in HOL.

The structure of the proof in ZF is exactly the same as in HOL. Consistency is proved by induction on the structure of evaluations, by using the induction rule proved by the (co)inductive package. This leads to six cases, one for each rule of the inference system. These are all proved more or less the same way as in HOL, using equivalent rules.

Size-wise, the two proofs are also very similar. One would expect proofs to be a bit longer in ZF, because all type constraints must be proved by hand, in contrast to HOL where they are dealt with automatically by the type system of Isabelle. Even though, it turns out that parts of the proof is actually shorter in ZF, because of other improvements, which could also have been made in HOL. For example the first three cases effectively reduces to one line proofs, by using some suitable classical reasoning sets. The three remaining cases still requires some manual effort to get though.

5.5 Discussion

5.5.1 Non-well-founded objects in ZF

As demonstrated above, the choice of the well-founded set theory ZF over non-well-founded set theory or HOL clearly has consequences when dealing with non-well-founded objects. It is not surprising that it is possible to formalise the consistency result in ZF. At worst one could imagine that it would be necessary to adopt a finite encoding of non-well-founded closures. It however turned out far more positive. By using a variant encoding it is possible to work with genuine non-well-founded objects in ZF. Co-datatypes have a place in ZF.

On the negative side quite a lot of work might be required to define and prove the necessary properties about the variant constructions needed. The example of variant maps above clearly illustrates that. It can however largely be put down to lack of support. The theory for ordinary notions such as pairs, functions etc. are far more developed, than the theory for the corresponding variant notions. Even with such a well developed theory some oddities might remain. It is, for example, necessary to ensure that certain sets used in connection with variant maps, do not contain the empty set as an element.

There is one issue which have not been addressed here: how to construct non-well-founded elements. It has not been proved that a theorem similar to (4) actually holds for values. A typical way to prove the "exists" part of such a theorem would be to exhibit a witness and prove that it is a value. This should be done.

5.5.2 The (co)inductive package

There is no question that the (co)inductive package is a big improvement of Isabelle ZF. Regarding the design of the package, the formalisation of the consistency result only gave rise to a few minor points of criticism:

- Generally, definitions and proofs are written in different files in Isabelle. Definitions reside in theory files and proofs in ML files. Theory files and ML files belong together pairwise. Isabelle loads a ML file after the corresponding theory file. Because (co)inductive definitions can refer to rules

that requires proof, it is sometimes necessary to do proofs within a theory file, which looks rather ugly, or to split the theory file although this is not motivated by the structure of the development.

- In the case of mutually defined sets, it would be nice to have specialised elimination rules for each set, instead of only having an elimination for the combined set. In the case of (co)datatypes it would also be nice to have specialised case-analysis operators.

- It would be nice if the (co)inductive package could organise the derived rules into suitable classical reasoning sets. Such sets made the proof of consistency easier.

- Although not directly related to the design of the package, a further developed theory of variant functions, products, sums etc. would be very useful. This is especially true when dealing with co-datatypes as the definition of values and value environment above illustrates.

Most of the time the package did exactly what was needed. Without mentioning it all again, a particular useful feature was the function `mk_cases` for deriving special instances of the elimination rules.

The work with the package revealed several bugs. None of these were serious, in the sense that something wrong could be proved. The package does not merely state axioms, but proves all the necessary rules from definitions. As a consequence, the errors manifested themselves, as the package not being able to prove some of the rules or looping in the attempt.

6 Conclusion

The main result of the paper [2], consistency, has been proved formally in Isabelle HOL and Isabelle ZF.

The notions of especially inductive definitions and datatypes but also coinductive definitions and co-datatypes turned out to be central in the formal treatment that leads to the consistency result. In Isabelle HOL (co)inductive definitions were formalised using a theory of least and greatest fixed points, while an axiomatic specification was given of (co)datatypes. In Isabelle ZF all (co)inductive definitions and (co)datatypes were formalised using the (co)inductive package available.

The development in Isabelle HOL clearly demonstrated the need for automated support for (co)inductive definitions of sets and datatypes, as an estimated 4/5 of the work was related to such definitions. Even more work would have been required if the datatypes had been defined using the fixed point theory. The formalisation in Isabelle ZF confirmed this. The (co)inductive package in

ZF reduced the work required dramatically. In many cases the package proved all the rules needed for each (co)inductive definition.

The handling of non-well-founded objects in the well-founded set theory ZF, turned out to be particular interesting. It was possible to represent non-well-founded closures as genuine non-well-founded objects. It was done by using a representation based on a so called variant notion of pairs, as opposed to ordinary pairs.

In contrast to HOL, ZF is an untyped logic. In general it made theorems and proofs slightly bigger in HOL. It however did not have any real practical consequences. Quite the opposite, much of the development in ZF was done by cutting and pasting from the corresponding parts in the HOL development.

Doing the actual consistency proof in both Isabelle HOL and ZF was a very positive experience. In both cases 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 proof, not to formalise it. Quite remarkable it 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.

Acknowledgements. Lawrence C. Paulson suggested the project and gave much useful advice. Søren T. Heilmann, Niels B. Maretti and Ole Rasmussen commented on drafts of this paper or its predecessor. Mads Tofte answered questions about his and Robin Milner's paper.

References

[1] Jacob Frost. A case study of co-induction in Isabelle HOL. Technical Report 308, University of Cambridge, Computer Laboratory, August 1993. http://www.cl.cam.ac.uk/Research/Reports/TR308-jf10008-co-induction.dvi.gz.

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

[3] Lawrence C. Paulson. Co-induction and co-recursion in higher-order logic. Technical Report 304, University of Cambridge, Computer Laboratory, July 1993. http://www.cl.cam.ac.uk/Research/Reports/TR304-lcp-coinduction.ps.gz.

[4] Lawrence C. Paulson. Introduction to Isabelle. Technical Report 280, University of Cambridge, Computer Laboratory, January 1993. ftp://ftp.cl.cam.ac.uk/ml/intro.dvi.gz.

[5] Lawrence C. Paulson. The Isabelle reference manual. Technical Report 283, University of Cambridge, Computer Laboratory, February 1993. ftp://ftp.cl.cam.ac.uk/ml/ref.dvi.gz.

[6] Lawrence C. Paulson. A concrete final coalgebra theorem for ZF set theory. Technical Report 334, University of Cambridge, Computer Laboratory, May 1994. http://www.cl.cam.ac.uk/Research/Reports/TR334-lcp-final.coalgebra.dvi.gz.

[7] Lawrence C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. Technical Report 320, University of Cambridge, Computer Laboratory, November 1994. http://www.cl.cam.ac.uk/Research/Reports/TR320-lcp-isabelle-ind-defs.dvi.gz.

[8] Lawrence C. Paulson. *Isabelle, A Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1994.

[9] Lawrence C. Paulson. Isabelle's object-logics. Technical Report 286, University of Cambridge, Computer Laboratory, October 1994. ftp://ftp.Cl.cam.ac.uk/ml/logics.dvi.gz.

[10] Tobias Nipkow and Lawrence C. Paulson. Datatypes and (co)inductive definitions in Isabelle/HOL. ftp://ftp.cl.cam.ac.uk/ml/HOL-extensions.dvi.gz, 1994.