Combining Higher Order Abstract Syntax with Tactical Theorem Proving and (Co)Induction[⋆]

S. J. Ambler (`S.Ambler@mcs.le.ac.uk`)
R. L. Crole (`R.Crole@mcs.le.ac.uk`) &
A. Momigliano (`A.Momigliano@mcs.le.ac.uk`)

Department of Mathematics and Computer Science, University of Leicester, Leicester, LE1 7RH, U.K.

Abstract. Combining Higher Order Abstract Syntax (HOAS) and induction is well known to be problematic. We have implemented a tool called Hybrid, within Isabelle HOL, which does allow object logics to be represented using HOAS, and reasoned about using tactical theorem proving in general and principles of (co)induction in particular. In this paper we describe Hybrid, and illustrate its use with case studies. We also provide some theoretical adequacy results which underpin our practical work.

1 Introduction

Many people are concerned with the development of computing systems which can be used to reason about and prove properties of programming languages. However, developing such systems is not easy. Difficulties abound in both practical implementation and underpinning theory. Our paper makes both a theoretical and practical contribution to this research area. More precisely, this paper concerns *how to reason about* object level logics with syntax involving variable binding—note that a programming language can be presented as an example of such an object logic. Our contribution is the provision of a mechanized tool, Hybrid, which has been coded within Isabelle HOL, and

- provides a form of *logical framework* within which the syntax of an object level logic can be adequately represented by *higher order abstract syntax* (HOAS);
- is consistent with *tactical theorem proving* in general, and principles of *induction and coinduction* in particular; and
- is *definitional* which guarantees consistency *within a classical type theory.*

We proceed as follows. In the introduction we review the idea of simple logical frameworks and HOAS, and the problems in combining HOAS and induction. In Section 2 we introduce our tool Hybrid. In Section 3 we provide some key technical definitions in Hybrid. In Section 4 we explain how Hybrid is used to

⋆ This work was supported by EPSRC grant number GR/M98555.

represent and reason about object logics, by giving some case studies. In Section 5 we give a mathematical description of Hybrid together with some underpinning theory. In Section 6 we review the articles related to our work. In Section 7 we comment on future plans.

Hybrid provides a form of logical framework. **Here we briefly recall some fundamental technical details of a basic logical framework, by using the vehicle of quantified propositional logic (*QPL*) as an object level logic—the notation will be used in Sections 4 and 5.** While this is a very small logic, it comes with a single binding construct which exemplifies the problems we are tackling. We let $V_1, V_2, \ldots$ be a countable set of (object level) variables. The *QPL* **formulae** are given by $Q ::= V_i \mid Q \supset Q \mid \forall V_i.\, Q$. Recall how to represent *QPL* in a simple logical framework—the framework here provides a form of HOAS, following [25]. A framework theory is specified by a signature of ground types which generate function types, and constants. The objects of the theory are given by $e ::= name \ \mid v_i \mid e\ e \mid \lambda\, v_i.\, e$ where *name* ranges over constants, and i ranges over $\mathbb{N}$ giving a countable set of (meta) variables v_i. From this, a standard type assignment system can be formulated, leading to a notion of canonical form. We refer the reader to [25] for the full definition. To represent *QPL* we take a ground type *oo* of *formulae*. We also give the constants of the theory, each of which corresponds to a *QPL* constructor. For *QPL* we specify $\mathsf{Imp} \ :: oo \Rightarrow oo \Rightarrow oo$ and $\mathsf{All} \ :: (oo \Rightarrow oo) \Rightarrow oo$. One can define a translation function $\ulcorner - \urcorner$ by the clauses

$$\ulcorner V_i \urcorner \stackrel{\text{def}}{=} v_i \qquad \ulcorner Q_1 \supset Q_2 \urcorner \stackrel{\text{def}}{=} \mathsf{Imp}\ \ulcorner Q_1 \urcorner \ulcorner Q_2 \urcorner \qquad \ulcorner \forall V_i.\, Q \urcorner \stackrel{\text{def}}{=} \mathsf{All}\ (\lambda\, v_i.\, \ulcorner Q \urcorner)$$

One can then show that this translation gives a "sensible" representation of *QPL* in the framework, meaning that the function $\ulcorner - \urcorner$ provides a bijection between *QPL* and canonical objects, and further $\ulcorner - \urcorner$ is compositional on substitution.

Although there are well known benefits in working with such higher order abstract syntax, there are also difficulties. In general, it is not immediately clear how to obtain a principle of induction over expressions or how to define functions on them by primitive recursion, although such principles do exist [27]. Worse still, apparently promising approaches can lead to inconsistencies [16]. The axiom of choice leads to loss of consistency, and exotic terms may lead to loss of adequacy. In our example, one would like to view the constants above as the constructors of a datatype $oo ::= var \mid \mathsf{Imp}\ (oo * oo) \mid \mathsf{All}\ (oo \Rightarrow oo)$ so that an induction principle is "immediate". Such datatype declarations would be perfectly legal in functional programming languages. In a theorem prover such as Isabelle HOL the constructors of a datatype are required to be injective [13]. However, the function $\mathsf{All}\ :: (oo \Rightarrow oo) \Rightarrow oo$ cannot be injective for cardinality reasons (Cantor's proof can be formalized within HOL) as Isabelle HOL provides a classical set theory. Moreover, the function space $oo \Rightarrow oo$ yields Isabelle HOL definable functions which do not "correspond" to terms of *QPL*. Such functions give rise to terms in *oo* which are unwanted, so-called *exotic* terms such as

$$\mathsf{All}\ (\lambda\, x.\, \mathsf{if}\ x = u\ \mathsf{then}\ u\ \mathsf{else}\ \mathsf{All}\ (\lambda\, z.\, z))$$

We show that it *is possible* to define a logical framework in which All is *injective on a subset* of $oo \Rightarrow oo$. The subset is sufficiently large to give an adequate representation of syntax—see Section 4.1.

2 Introducing Hybrid

Within Isabelle HOL, our goal is to **define a datatype for λ-calculus with constants over which we can deploy (co)induction principles, while representing variable binding through Isabelle's HOAS. We do this in a tool called Hybrid, which we introduce in this section.** Our starting point is the work [1] of Andrew Gordon, which we briefly review. It is well known (though rarely proved—see Section 5) that λ-calculus expressions are in bijection with (a subset of) de Bruijn expressions. Gordon defines a de Bruijn notation in which expressions have *named free variables* given by *strings*. He can write $T = \mathsf{dLAMBDA}\,\mathtt{v}\,t$ (where $\mathtt{v}$ is a string) which corresponds to an abstraction in which $\mathtt{v}$ is bound in t. The function $\mathsf{dLAMBDA}$ has a *definition* which converts T to the corresponding de Bruijn term which has an outer abstraction, and a subterm which is t in de Bruijn form, in which (free) occurrences of $\mathtt{v}$ are converted to bound de Bruijn indices. For example,

$$\mathsf{dLAMBDA}\,\mathtt{v}\,(\mathsf{dAPP}\;(\mathsf{dVAR}\;\mathtt{v})\;(\mathsf{dVAR}\;\mathtt{u})) = \mathsf{dABS}\;(\mathsf{dAPP}\;(\mathsf{dBND}\;0)\;(\mathsf{dVAR}\;\mathtt{u}))$$

Gordon demonstrates the utility of this approach. It provides a good mechanism through which one may work with named bound variables, but it does not exploit the built in HOAS which Isabelle HOL itself uses to represent syntax. The novelty of our approach is that we *do* exploit the HOAS at the meta (machine) level.

We introduce Hybrid by example. First, some basics. Of central importance is a Isabelle HOL datatype of de Bruijn expressions, where *bnd* and *var* are the natural numbers, and *con* provides names for constants

$$\boxed{expr ::= \mathsf{CON}\; con \mid \mathsf{VAR}\; var \mid \mathsf{BND}\; bnd \mid expr\;\$\$\;expr \mid \mathsf{ABS}\; expr}$$

Let $T_O = \Lambda\, V_1.\, \Lambda\, V_2.\, V_1\; V_3$ be a genuine, honest to goodness (object level) syntax[1] tree. Gordon would represent this by

$$T_G = \mathsf{dLAMBDA}\,\mathtt{v1}\,(\mathsf{dLAMBDA}\,\mathtt{v2}\,(\mathsf{dAPP}\;(\mathsf{dVAR}\,\mathtt{v1})\;(\mathsf{dVAR}\,\mathtt{v3})))$$

which equals

$$\mathsf{dABS}\;(\mathsf{dABS}\;(\mathsf{dAPP}\;(\mathsf{dBND}\;1)\;(\mathsf{dVAR}\;\mathtt{v3})))$$

Hybrid provides a binding mechanism with similarities to $\mathsf{dLAMBDA}$. Gordon's T would be written as $\mathsf{LAM}\,v.\,t$ in Hybrid. This is simply a *definition for* a de Bruijn term. A *crucial difference* in our approach is that *bound variables in the object logic* are *bound variables in Isabelle HOL*. Thus the v in $\mathsf{LAM}\,v.\,t$

[1] We use a capital Λ and capital V to avoid confusion with meta variables v and meta abstraction λ.

is a metavariable (and not a string as in Gordon's approach). In Hybrid we also choose to denote object level free variables by terms of the form $\mathsf{VAR}\,i$; however, this has essentially no impact on the technical details—the important thing is the countability of free variables. In Hybrid the T_O above is rendered as $T_H = \mathsf{LAM}\,v_1.\,(\mathsf{LAM}\,v_2.\,(v_1 \;\$\$\; \mathsf{VAR}\;3))$. The LAM is an Isabelle HOL binder, and this expression is by *definition*

$$\mathsf{lambda}\;(\lambda\,v_1.\,(\mathsf{lambda}\;(\lambda\,v_2.\,(v_1 \;\$\$\; \mathsf{VAR}\;3))))$$

where λv_i is meta abstraction and one can see that the object level term is rendered in the usual HOAS format, where $\mathsf{lambda} :: (\mathit{expr} \Rightarrow \mathit{expr}) \Rightarrow \mathit{expr}$ is a defined function. Then Hybrid will reduce T_H to the de Bruijn term

$$\mathsf{ABS}\;(\mathsf{ABS}\;(\mathsf{BND}\;1 \;\$\$\; \mathsf{VAR}\;3))$$

as in Gordon's approach. The key to this is, of course, the definition of lambda, which relies crucially on *higher order pattern matching.* We return to its definition in Section 3. In summary, Hybrid provides a form of HOAS where object level

- free variables correspond to Hybrid expressions of the form $\mathsf{VAR}\,i$;
- bound variables correspond to (bound) meta variables;
- abstractions $\Lambda V.\,E$ correspond to expressions $\mathsf{LAM}\,v.\,e = \mathsf{lambda}\;(\lambda\,v.\,e)$;
- applications $E_1\;E_2$ correspond to expressions $e_1 \;\$\$\; e_2$.

3 Definition of Hybrid in Isabelle HOL

Hybrid consists of a small number of Isabelle HOL theories. One of these provides the dataype of de Bruijn expressions given in Section 2. The theories also contain definitions of various key functions and inductive sets. **In this section we outline the definitions, and give some examples, where e ranges over Isabelle HOL expressions. We show how Hybrid provides a form of HOAS, and how this can be married with induction.**

We are going to use a pretty-printed version of Isabelle HOL concrete syntax; a rule

$$\frac{H_1 \ldots H_n}{C}$$

will be represented as $[\![\; H_1; \ldots; H_n \;]\!] \Longrightarrow C$. An Isabelle HOL type declaration has the form $s\;\;::\;[\,t_1, \ldots t_n\,] \Rightarrow t$. The Isabelle metauniversal quantifier is $\bigwedge$, while Isabelle HOL connectives are represented via the usual logical notation. Free variables are implicitly universally quantified. Datatypes will be introduced using BNF grammars. The sign $==$ (Isabelle metaequality) will be used for *equality by definition.*

Note that all of the infrastructure of Hybrid, which we now give, is specified *definitionally.* We do *not* postulate axioms, as in some approaches reviewed in Section 6, which require validation.

$\boxed{\mathsf{level} :: [\,bnd, expr\,] \Rightarrow bool}$ Recall that $\mathsf{BND}\ i$ corresponds to a bound variable in the λ-calculus, and $\mathsf{VAR}\ i$ to a free variable; we refer to **bound** and **free indices** respectively. We call a bound index i **dangling** if i or less Abs labels occur between the index i and the root of the expression tree. e is said to be at **level** $l \geq 1$, if enclosing e inside l Abs nodes ensures that the resulting expression has no dangling indices. These ideas are standard, as is the implementation of level.

$\boxed{\mathsf{proper} :: expr \Rightarrow bool}$ One has $\mathsf{proper}\ e == \mathsf{level}\, 0\, e$. A proper expression is one that has no dangling indices and corresponds to a λ-calculus expression.

$\boxed{\mathsf{insts} :: bnd \Rightarrow (expr)list \Rightarrow expr \Rightarrow expr}$ We explain this function by example. Suppose that j is a bound index occurring in e, and $v_0, \ldots, v_m$ a list of metavariables of type $expr$. Let $\mathsf{BND}\ j$ be enclosed by a ABS nodes. If $a \leq j$ (so that j dangles) then insts replaces $\mathsf{BND}\ j$ by v_{j-a}. If j does not dangle, then insts leaves $\mathsf{BND}\ j$ alone. For example, noting $5 - 2 = 3$,

$$\mathsf{insts}\ 0\ v_0, \ldots, v_m\ \mathsf{ABS}\ (\mathsf{ABS}\ (\mathsf{BND}\ 0\ \$\$\ \mathsf{BND}\ 5)) = \mathsf{ABS}\ (\mathsf{ABS}\ (\mathsf{BND}\ 0\ \$\$\ v_3))$$

$\boxed{\mathsf{abst} :: [\,bnd, expr \Rightarrow expr\,] \Rightarrow bool}$ This predicate is defined by induction as a subset of $bnd * (expr \Rightarrow expr)$. The inductive definition is

$$\begin{aligned}
&\Longrightarrow \mathsf{abst}\ i\ (\lambda\, v.\, v)\\
&\Longrightarrow \mathsf{abst}\ i\ (\lambda\, v.\, \mathsf{VAR}\ n)\\
j < i &\Longrightarrow \mathsf{abst}\ i\ (\lambda\, v.\, \mathsf{BND}\ j)\\
[\![\, \mathsf{abst}\ i\ f; \mathsf{abst}\ i\ g\,]\!] &\Longrightarrow \mathsf{abst}\ i\ (\lambda\, v.\, f\, v\ \$\$\ g\, v)\\
\mathsf{abst}\ (\mathsf{Suc}\ i)\ f &\Longrightarrow \mathsf{abst}\ i\ (\lambda\, v.\, \mathsf{ABS}\ (f\, v))
\end{aligned}$$

This definition is best explained in terms of the next function.

$\boxed{\mathsf{abstr} :: [\,expr \Rightarrow expr\,] \Rightarrow bool}$ We set $\mathsf{abstr}\ e == \mathsf{abst}\ 0\ e$. This function determines when an expression e of type $expr \Rightarrow expr$ is an **abstraction**. This is a key idea, and the notion of an abstraction is central to the formulation of induction principles. We illustrate the notion by example. Suppose that $\mathsf{ABS}\ e$ is proper; for example let $e = \mathsf{ABS}\ (\mathsf{BND}\ 0\ \$\$\ \mathsf{BND}\ 1)$. Then e is of level 1, and in particular there may be some bound indices which now dangle; for example $\mathsf{BND}\ 1$ in $\mathsf{ABS}\ (\mathsf{BND}\ 0\ \$\$\ \mathsf{BND}\ 1)$. An abstraction is produced by replacing each occurrence of a dangling index with a metavariable (which can be automated with insts) and then abstracting the meta variable. Our example yields the abstraction $\lambda\, v.\, \mathsf{ABS}\ (\mathsf{BND}\ 0\ \$\$\ v)$.

$\boxed{\mathsf{lbnd} :: [\,bnd, expr \Rightarrow expr, expr\,] \Rightarrow bool}$ This predicate is defined as an inductive subset of $S \overset{\text{def}}{=} bnd * (expr \Rightarrow expr) * expr$. The inductive definition is

$$\begin{aligned}
&\Longrightarrow \mathsf{lbnd}\ i\ (\lambda\, v.\, v)\ (\mathsf{BND}\ i)\\
&\Longrightarrow \mathsf{lbnd}\ i\ (\lambda\, v.\, \mathsf{VAR}\ n)\ (\mathsf{VAR}\ n)\\
&\Longrightarrow \mathsf{lbnd}\ i\ (\lambda\, v.\, \mathsf{BND}\ j)\ (\mathsf{BND}\ j)\\
[\![\, \mathsf{lbnd}\ i\ f\ s; \mathsf{lbnd}\ i\ g\ t\,]\!] &\Longrightarrow \mathsf{lbnd}\ i\ (\lambda\, v.\, f\, v\ \$\$\ g\, v)\ (s\ \$\$\ t)\\
\mathsf{lbnd}\ (\mathsf{Suc}\ i)\ f\ s &\Longrightarrow \mathsf{lbnd}\ i\ (\lambda\, v.\, \mathsf{ABS}\ (f\, v))\ (\mathsf{ABS}\ s)
\end{aligned}$$

There is a default case (omitted above) which is called when the second argument does not match any of the given patterns. It is a theorem that this defines a function. The proof is by induction on the *rank* of the function where $\mathsf{rank}\, f = = \mathsf{size}\,(f\,(\mathsf{VAR}\ 0))$.

$\boxed{\mathsf{lbind} :: [\, bnd, expr \Rightarrow expr\,] \Rightarrow expr}$ Set $\mathsf{lbind}\ i\ e == \epsilon\, s.\, \mathsf{lbnd}\ i\ e\ s$ where ϵ is the description operator. Consider the abstraction $\lambda\, v.\, \mathsf{ABS}\ (\mathsf{BND}\ 0\ \$\$\ v)$. The arguments to lbind consist of a bound index, and an abstraction. The intuitive action of this function is that it replaces each bound occurrence of a binding variable in the body of an abstraction, with a bound index, so that a level 1 expression results. This is the reverse of the procedure defined in the paragraph concerning abstractions, where dangling indices were instantiated to metavariables using insts. In practice, lbind will be called on 0 at the top level. Thus one has

$$\mathsf{lbind}\ 0\ (\lambda\, v.\, \mathsf{ABS}\ (\mathsf{BND}\ 0\ \$\$\ v)) = \ldots = \mathsf{ABS}\ (\mathsf{BND}\ 0\ \$\$\ \mathsf{BND}\ 1)$$

$\boxed{\mathsf{lambda} :: (expr \Rightarrow expr) \Rightarrow expr}$ Set $\mathsf{lambda}\ e == \mathsf{ABS}\ (\mathsf{lbind}\ 0\ e)$. Its purpose is to transform an abstraction into the "corresponding" proper de Bruijn expression. Our running example yields

$$\mathsf{lambda}\ (\lambda\, v.\, \mathsf{ABS}\ (\mathsf{BND}\ 0\ \$\$\ v)) = \mathsf{ABS}\ (\mathsf{ABS}\ (\mathsf{BND}\ 0\ \$\$\ \mathsf{BND}\ 1))$$

It is easy to perform induction over a datatype of de Bruijn terms. However, we wish to be able to perform induction over the Hybrid expressions which we have just given. In order to do this, we want to view the functions CON, VAR, $$, and lambda as datatype constructors, that is, they should be injective, with disjoint images. In fact, we identify subsets of *expr* and $expr \Rightarrow expr$ for which these properties hold. The subset of *expr* consists of those expressions which are *proper*. The subset of $expr \Rightarrow expr$ consists of all those e for which $\mathsf{LAM}\, v.\, e\, v$ is proper. In fact, this means e is an abstraction, which is intuitive but requires proof—it is a Hybrid theorem. We can then prove that

$$[\![\ \mathsf{abstr}\ e;\ \mathsf{abstr}\ f\ ]\!] \Longrightarrow (\mathsf{Lam}\, x.\ e\ x = \mathsf{Lam}\, y.\ f\ y) = (e = f) \qquad \mathit{INJ}$$

which says that lambda is injective on the set of abstractions. This is crucial for the proof of an induction principle for Hybrid, which is omitted for reasons of space, but will appear in a journal version of this paper.

4 Hybrid as a Logical Framework

Recall that in Section 2 we showed that Hybrid supports HOAS. **In this section we show how Hybrid can be used as a logical framework to represent object logics, and further how we can perform tactical theorem proving.**

The system provides:

– A number of automatic tactics: for example proper_tac (resp. abstr_tac) will recognize whether a given term is indeed proper (resp. an abstraction).

– A suite of theorems: for example, the injectivity and distinctness properties of Hybrid constants, and induction principles over *expr* and $expr \Rightarrow expr$, as discussed in Section 3.

Note that the adequacy of our representations will be proved in a forthcoming paper.

4.1 Quantified Propositional Logic

We begin with an encoding of the quantified propositional logic introduced in Section 1. While the fragment presented there is functionally complete, we choose to work with the following syntax

$$Q ::= V_i \mid \neg Q \mid Q \wedge Q' \mid Q \vee Q' \mid Q \supset Q' \mid \forall V_i.\, Q \mid \exists V_i.\, Q$$

This will allow us to demonstrate the representation of object level syntax in detail, and show some properties of an algorithm to produce negation normal forms.

So far we have written *expr* for the type of Hybrid expressions. This was to simplify the exposition, and does not correspond directly to our code. There one sees that Hybrid actually provides a type *con expr* of de Bruijn expressions, where *con* is a type of names of constants. Typically, such names are for object logic constructors. Thus we can define a different type *con* for each object logic we are dealing with. In the case of *QPL*, we declare

$$con ::= cNOT \mid cIMP \mid cAND \mid cOR \mid cALL \mid cEX$$

followed by a Isabelle HOL type whose elements represent the object level formulae, namely $oo == con\ expr$. Readers should pause to recall the logical framework (and HOAS) of Section 1.

Now we show how to represent the object level formulae, as Hybrid expressions of type *oo*. The table below shows analogous constructs in $\mathcal{LF}$ and *Hybrid*.

	Constants	**Application**	**Abstraction**
$\mathcal{LF}$	name	$e_1\ e_2$	$\lambda\, v_i.\, e$
Hybrid	CON $cNAME$	e_1 \$\$ e_2	LAM $v_i.\, e$

In the next table, we show the representation of the object level formulae $Q \supset Q'$ and $\forall V_i.\, Q$ in $\mathcal{LF}$ and Hybrid, where $\ulcorner - \urcorner$ is a translation function

$\mathcal{LF}$	Imp $\ulcorner Q \urcorner \ulcorner Q' \urcorner$	All $(\lambda\, v.\ulcorner Q \urcorner)$
Hybrid	CON $cIMP$ \$\$ $\ulcorner Q \urcorner$ \$\$ $\ulcorner Q' \urcorner$	CON $cALL$ \$\$ LAM $v.\ulcorner Q \urcorner$
Abbrevs	$\ulcorner Q \urcorner$ Imp $\ulcorner Q' \urcorner$	All $v.\ulcorner Q \urcorner$

The bottom row introduces some Isabelle HOL binders as abbreviations for the middle row, where the Isabelle HOL definitions are

$$q \ \mathsf{Imp}\ q' == \mathsf{CON}\ cIMP\ \$\$\ q\ \$\$\ q' \ \text{ where }\ \mathsf{Imp} :: [\, oo, oo\,] \Rightarrow oo$$

and

$$\mathsf{All}\, v.\, q\, v == \mathsf{CON}\ cALL\ \$\$\ \mathsf{LAM}\, v.\, q\, v \text{ where } \mathsf{All} :: (oo \Rightarrow oo) \Rightarrow oo$$

The code for the representation of the remainder of *QPL* is similar:

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{And} & :: & [\, oo, oo\,] \Rightarrow oo & \mathsf{Or} & :: & [\, oo, oo\,] \Rightarrow oo \\
q \ \mathsf{And}\ q' & == & \mathsf{CON}\ cAND\ \$\$\ q\ \$\$\ q' & q\ \mathsf{Or}\ q' & == & \mathsf{CON}\ cOR\ \$\$\ q\ \$\$\ q' \\
\mathsf{Not} & :: & oo \Rightarrow oo & \mathsf{Ex} & :: & (oo \Rightarrow oo) \Rightarrow oo \\
\mathsf{Not}\ q & == & \mathsf{CON}\ cNOT\ \$\$\ q & \mathsf{Ex}\, v.\, q\ v & == & \mathsf{CON}\ cEX\ \$\$\ \mathsf{LAM}\, v.\, q\ v
\end{array}$$

The *QPL* formula $\forall V_1.\, \forall V_2.\, V_1 \supset V_2$ is represented by $\mathsf{All}\, v_1.\, \mathsf{All}\, v_2.\ v_1\ \mathsf{Imp}\ v_2$, although the "real" underlying form is

$$\mathsf{CON}\ cALL\ \$\$\ (\mathsf{LAM}\, v_1.\, \mathsf{CON}\ cALL\ \$\$\ \mathsf{LAM}\, v_2.\, (\mathsf{CON}\ cIMP\ \$\$\ v_1\ \$\$\ v_2)$$

These declarations almost induce a data-type, in the sense the above defined constants enjoy certain freeness properties, much as they would if they were datatype constructors. We can prove that they define *distinct* values; for example $\mathsf{All}\, v.\, q\ v \neq \mathsf{Ex}\, v.\, q\ v$. This is achieved by straightforward simplification of their definitions to the underlying representation. Injectivity of higher-order constructors (recall the end of Section 1) holds conditionally on their bodies being abstractions. In particular, recall from Section 3 the result *INJ* that the LAM binder is injective on the set of abstractions. Simplification will yield $[\![\ \mathsf{abstr}\ e; \mathsf{abstr}\ f; \mathsf{All}\, v.\, e\ v\ =\ \mathsf{All}\, v.\, f\ v\ ]\!] \Longrightarrow e = f$. Since the type oo of legal formulae is merely an abbreviation for Hybrid expressions, we need to introduce a "well-formedness" predicate, such that $\mathsf{isForm}\ \ulcorner Q \urcorner$ holds iff Q is a legal object level formula.[2] The inductive definition of isForm in Isabelle HOL is immediate as far as the propositional part of *QPL* is concerned, for example $[\![\ \mathsf{isForm}\ p; \mathsf{isForm}\ q\ ]\!] \Longrightarrow \mathsf{isForm}\ (\ p\ \mathsf{Imp}\ q)$. For the quantified part, we first remark that in a framework such as $\mathcal{LF}$ one would write

$$[\![\ \forall y.\, \mathsf{isForm}\ y \to \mathsf{isForm}\ (p\ y)\ ]\!] \Longrightarrow \mathsf{isForm}\ (\mathsf{All}\, v.\, p\ v)$$

This is not possible in Isabelle HOL, since the above clause, if taken as primitive, would induce a (set-theoretic) non-monotone operator, and cannot be part of an introduction rule in an inductive definition. Therefore, we instead descend into the scope of the quantifier replacing it with a fresh free variable and add the corresponding base case:

$$\begin{array}{r}
\mathsf{isForm}\ (\mathsf{VAR}\ i) \\
[\![\ \mathsf{abstr}\ p; \forall i.\, \mathsf{isForm}\ (p\ (\mathsf{VAR}\ i))\ ]\!] \Longrightarrow \mathsf{isForm}\ (\mathsf{All}\, v.\, p\ v)
\end{array}$$

We can now proceed to an encoding of an algorithm for negation normal form as an inductive relation, where we skip some of the propositional clauses,

[2] Please see remarks in Section 7 concerning internalizing such predicates as types.

and Φ abbreviates $\mathsf{abstr}\ p; \mathsf{abstr}\ q$;

$$\begin{array}{rl}
 & \mathsf{nnf}\ (\mathsf{VAR}\ i)\ (\mathsf{VAR}\ i) \\
 & \mathsf{nnf}\ (\mathsf{Not}\ (\mathsf{VAR}\ i))\ (\mathsf{Not}\ (\mathsf{VAR}\ i)) \\
\mathsf{nnf}\ b\ d \Longrightarrow & \mathsf{nnf}\ (\mathsf{Not}\ (\mathsf{Not}\ b))\ d \\
[\![\ \mathsf{nnf}\ (\mathsf{Not}\ p)\ d; \mathsf{nnf}\ (\mathsf{Not}\ q) e\ ]\!] \Longrightarrow & \mathsf{nnf}\ (\mathsf{Not}\ (\ p\ \mathsf{And}\ q))\ (\ d\ \mathsf{Or}\ e) \\
 & \ldots \\
[\![\ \Phi;\ \forall i.\, \mathsf{nnf}\ (\mathsf{Not}\ (p\ (\mathsf{VAR}\ i)))(q\ (\mathsf{VAR}\ i))\ ]\!] \Longrightarrow & \mathsf{nnf}\ (\mathsf{Not}\ (\mathsf{Ex}\, v.\, p\ v))\ (\mathsf{All}\, v.\, q\ v) \\
[\![\ \Phi;\ \forall i.\, \mathsf{nnf}\ (\mathsf{Not}\ (p\ (\mathsf{VAR}\ i)))(q\ (\mathsf{VAR}\ i))\ ]\!] \Longrightarrow & \mathsf{nnf}\ (\mathsf{Not}\ (\mathsf{All}\, v.\, p\ v))\ (\mathsf{Ex}\, v.\, q\ v) \\
[\![\ \Phi;\ \forall i.\, \mathsf{nnf}\ (\mathsf{Not}\ (p\ (\mathsf{VAR}\ i)))(q\ (\mathsf{VAR}\ i))\ ]\!] \Longrightarrow & \mathsf{nnf}\ (\mathsf{All}\, v.\, p\ v)\ (\mathsf{All}\, v.\, q\ v) \\
[\![\ \Phi;\ \forall i.\, \mathsf{nnf}\ (\mathsf{Not}\ (p\ (\mathsf{VAR}\ i)))(q\ (\mathsf{VAR}\ i))\ ]\!] \Longrightarrow & \mathsf{nnf}\ (\mathsf{Ex}\, v.\, p\ v)\ (\mathsf{Ex}\, v.\, q\ v)
\end{array}$$

Note how, in the binding cases, we explicitly state in Φ which second-order terms are abstractions; this allows us to exploit the injectivity of abstractions to derive the appropriate elimination rules.

It is possible to show in a fully automatic way that the algorithm yields negation normal forms (whose definition is omitted), that is $\mathsf{nnf}\ q\ q' \Longrightarrow \mathsf{isNnf}\ q'$. Moreover it is a functional relation: $\mathsf{nnf}\ q\ q_1 \Longrightarrow \forall q_2.\, \mathsf{nnf}\ q\ q_2 \to q_1 = q_2$. The latter proof exploits a theorem regarding extensionality of abstractions, namely:

$$[\![\ \mathsf{abstr}\ e; \mathsf{abstr}\ f; \forall i.\, e\ (\mathsf{VAR}\ i) = f\ (\mathsf{VAR}\ i)\ ]\!] \Longrightarrow e = f$$

Note that the above is taken as an *axiom* in the *Theory of Contexts* [17].

4.2 Operational Semantics in the Lazy Lambda Calculus

The object logic in this section is yet another λ-calculus, Abramsky's lazy one [2]. We describe some properties of its operational semantics. In particular, we give HOAS encodings of some notions such as divergence and simulation which are naturally rendered *coinductively*—this can only be approximated in other approaches, as we discuss in Section 6.

To represent the lazy λ-calculus, the type *con* will contain the names *cAPP* and *cABS*, used to represent object level application and abstraction. We then define the constants below from these names, where $lexp == con\ expr$.

$$\begin{array}{rlrl}
@ & ::\ [\,lexp, lexp\,] \Rightarrow lexp & \mathsf{Fun}\,. & ::\ (lexp \Rightarrow lexp) \Rightarrow lexp \\
p\ @\ q & == \mathsf{CON}\ cAPP\ \$\$\ p\ \$\$\ q & \mathsf{Fun}\, x.\, f\ x & == \mathsf{CON}\ cABS\ \$\$\ \mathsf{LAM}\, x.\, f\ x
\end{array}$$

The definition of the well-formedness predicate isExp is analogous to the one in Section 4 and is omitted.

The benefits of obtaining object-level substitution via metalevel β-conversion are exemplified in the encoding of call-by-name evaluation (on closed terms) via the inductive definition of $\gg\ ::\ [\,lexp, lexp\,] \Rightarrow bool$.

$$\begin{array}{rl}
[\![\ \mathsf{abstr}\ e;\ \forall i.\, \mathsf{isExp}\ (e\ (\mathsf{VAR}\ i))\ ]\!] \Longrightarrow & \mathsf{Fun}\, x.\, e\ x \gg \mathsf{Fun}\, x.\, e\ x \\
[\![\ e1 \gg \mathsf{Fun}\, x.\, e\ x;\ \mathsf{abstr}\ e;\ \mathsf{isExp}\ e2;\ (e\ e2) \gg v\ ]\!] \Longrightarrow & (e1\ @\ e2) \gg v
\end{array}$$

Standard properties such as uniqueness of evaluation and value soundness have direct proofs based only on structural induction and the introduction and elimination rules.

Divergence can be defined co-inductively as the predicate $\mathsf{divrg} :: \mathit{lexp} \Rightarrow \mathit{bool}$:

$$\begin{array}{r} [\![\ \mathsf{isExp}\ e1;\ \mathsf{isExp}\ e2;\ \mathsf{divrg}\ e1\ ]\!] \Longrightarrow \mathsf{divrg}\ (e1\ @\ e2) \\ [\![\ e1 \gg\ \mathsf{Fun}\, x.\, e\ x;\ \mathsf{abstr}\ e;\ \mathsf{isExp}\ e2;\ \mathsf{divrg}\ (e\ e2)\ ]\!] \Longrightarrow \mathsf{divrg}\ (e1\ @\ e2) \end{array}$$

We can give a fully automated co-inductive proof of the divergence of combinators such as $\Omega \stackrel{\text{def}}{=} (\mathsf{Fun}\, x.\, x\ @\ x)\ @\ (\mathsf{Fun}\, x.\, x\ @\ x)$, once we have added the abstr_tac tactic to the built-in simplifier. Moreover, there is a direct proof that convergence and divergence are exclusive and exhaustive.

Applicative (bi)simulation $\leq\ :: [\,\mathit{lexp}, \mathit{lexp}\,] \Rightarrow \mathit{bool}$ is another interesting (co-inductive) predicate with the single introduction rule

$$\begin{array}{l} [\![\ \forall t.\ r \gg\ \mathsf{Fun}\, x.\, t\ x \wedge \mathsf{abstr}\ t \rightarrow (\exists u.\ s \gg\ \mathsf{Fun}\, x.\, u\ x \wedge \mathsf{abstr}\ u\ \wedge \\ \qquad\qquad\qquad\qquad\qquad (\forall p. \mathsf{isExp}\ p \rightarrow\ (t\ p) \leq\ (u\ p)))\ ]\!] \Longrightarrow\ r \leq\ s \end{array}$$

The HOAS style here greatly simplifies the presentation and correspondingly the metatheory. Indeed, with the appropriate instantiation of the coinductive relation, the proofs that simulation is a pre-order and bisimulation an equivalence relation are immediate. Other verified properties include Kleene equivalence is a simulation, and divergent terms are the least element in this order.

4.3 The Higher-Order π-Calculus

We present an encoding of the monadic higher-order π-calculus [24], with structural congruence and reaction rules; see [11] for a review of other styles of encodings for the first-order case. The syntax is given by the following, where $a, \overline{a}$ ranges over names, X over agent variables:

$$\begin{array}{l} \alpha ::= \tau \mid a(X) \mid \overline{a}\langle P\rangle \\ P ::= X \mid \Sigma_{i\in I}\alpha_i.\, P \mid (P_1 \mid P_2) \mid (\nu a)P \end{array}$$

The encoding introduces appropriate type abbreviations (namely pi and name) and constants; we concentrate on restriction, input and output:

$$\begin{array}{rcl} \mathsf{New} & :: & (\mathit{name} \Rightarrow \mathit{pi}) \Rightarrow \mathit{pi} \\ (\mathsf{New}\ a)p\ a & == & \mathsf{CON}\ \mathit{cNU}\ \$\$\ \mathsf{LAM}\, a.\, p\ a \\ \mathsf{In} & :: & [\,\mathit{name}, (\mathit{pi} \Rightarrow \mathit{pi})\,] \Rightarrow \mathit{pi} \\ \mathsf{In}\ a\ (p) & == & \mathsf{CON}\ \mathit{cIN}\ \$\$\ \mathsf{lambda}\ p \\ \mathsf{Out} & :: & [\,\mathit{name}, \mathit{pi}, \mathit{pi}\,] \Rightarrow \mathit{pi} \\ \mathsf{Out}\ \overline{a}\langle p\rangle q & == & \mathsf{CON}\ \mathit{cOUT}\ \$\$\ a\ \$\$\ p\ \$\$\ q \end{array}$$

Replication is defined from these constants and need not be primitive:

$$!\,p == (\mathsf{New}\ a)(D|\mathsf{Out}\ \overline{a}\langle p|D\rangle)\ \text{ where } D == \mathsf{In}\ a\ (\lambda\, x.\, (x|\mathsf{Out}\ \overline{a}\langle x\rangle))$$

We then inductively define well-formed processes $\mathsf{isProc}\, p$, whose introduction rules are, omitting non-binding cases,

$$\begin{array}{r} \mathsf{isProc}\ (\mathsf{VAR}\ i) \\ [\![\ \mathsf{abstr}\ p;\ \forall a.\, \mathsf{isName}\ a \rightarrow \mathsf{isProc}\ (p\ a)\ ]\!] \Longrightarrow \mathsf{isProc}\ (\mathsf{New}\ a)p\ a \\ [\![\ \mathsf{abstr}\ p;\ \mathsf{isName}\ a;\ \forall i.\, \mathsf{isProc}\ (p\ (\mathsf{VAR}\ i))\ ]\!] \Longrightarrow \mathsf{isProc}\ \mathsf{In}\ a\ (p) \end{array}$$

Restriction, being represented as a function of type $name \Rightarrow pi$ is well-formed if its body $(p\ a)$ is, under the assumption $\mathsf{isName}\ a$; this differs from input, which contains a function of type $pi \Rightarrow pi$.

Process abstraction is defined by $\mathsf{procAbstr}\ p \ == \ \mathsf{abstr}\ p \wedge (\forall q.\, \mathsf{isProc}\ q \to \mathsf{isProc}\ (p\ q))$. Next, structural congruence is introduced and we proceed to encode reaction rules as the inductive definition $\longmapsto :: [\, pi \Rightarrow pi\,] \Rightarrow bool$. Note that the presence of the structural rule make this unsuitable for search. Here is a sample:

$$\llbracket\ \mathsf{procAbstr}\ p; \mathsf{isProc}\ q; \mathsf{isProc}\ p'; \mathsf{isName}\ a\ \rrbracket \Longrightarrow \mathsf{In\ a\ (p)|Out\ \overline{a}\langle q\rangle p' \longmapsto (p\ q)|p'}$$
$$\llbracket\ \mathsf{abstr}\ p; \mathsf{abstr}\ p'; (\forall a.\, \mathsf{isName}\ a \to \mathsf{(p\ a)} \longmapsto \mathsf{(p'\ a)})\ \rrbracket \Longrightarrow \mathsf{(New\ a)p\ a} \longmapsto \mathsf{(New\ a)p'\ a}$$

A formalization of late operational semantics following [18] is possible and will be treated within a separate paper.

5 A Theory of Hybrid

The goal of this section is to describe a mathematical model of Hybrid and then show that the model provides an adequate representation of the λ-calculus. The work here serves to illustrate and under-pin *Hybrid*. We proceed as follows. In Section 5.1 we set up an *explicit bijection* between the set of alpha equivalence classes of lambda expressions, and the set of proper de Bruijn terms. This section also allows us to introduce notation. In Section 5.2 we prove adequacy for an object level λ-calculus by proving an equivalent result for (proper) object level de Bruijn expressions, and appealing to the bijection.

First, some notation *for the object level*. λ-calculus expressions are inductively defined by $E ::= V_i \mid E\ E \mid \Lambda V_i.E$. We write $E[E'/V_i]$ for capture avoiding substitution, $E \sim_\alpha E'$ for α-equivalence, and $[E]_\alpha$ for alpha equivalence classes. We write $\mathcal{LE}$ for the set of expressions, and $\mathcal{LE}/{\sim_\alpha}$ for the set of all alpha equivalence classes. The set of de Bruijn expressions is denoted by $\mathcal{DB}$, and generated by $D ::= \bar{i} \mid i \mid D\ \$\ D \mid A(D)$ where $\bar{i}$ is a free index. We write $\mathcal{DB}(l)$ for the set of expressions at level l, and $\mathcal{PDB}$ for the set of proper expressions. Note that $\mathcal{PDB} = \mathcal{DB}(0) \subset \mathcal{DB}(1) \ldots \subset \mathcal{DB}(l) \ldots \subset \mathcal{DB}$ and $\mathcal{DB} = \bigcup_{l<\omega} \mathcal{DB}(l)$.

5.1 A Bijection Between λ-calculus and Proper de Bruijn

Theorem 1. *There is a bijection $\theta\colon \mathcal{LE}/{\sim_\alpha} \rightleftarrows \mathcal{PDB}\colon \phi$ between the set $\mathcal{LE}/{\sim_\alpha}$ of alpha equivalence classes of λ-calculus expressions, and the set $\mathcal{PDB}$ of proper de Bruijn expressions,*

In order to prove the theorem, we first establish in Lemma 1 and Lemma 2 the existence of a certain family of pairs of functions $\llbracket - \rrbracket_L \colon \mathcal{LE} \rightleftarrows \mathcal{DB}(|L|)\colon (\!|-|\!)_L$. Here, **list** L of length $|L|$ is one whose elements are object level variables V_i. We say that L is **ordered** if it has an order reflected by the indices i, and no repeated elements. The first element has the largest index. We write ϵ for the empty list.

Lemma 1. *For any L, there exists a function $\llbracket - \rrbracket_L \colon \mathcal{LE} \to \mathcal{DB}(|L|)$ given recursively by*

- $[\![V_i]\!]_L \stackrel{\text{def}}{=}$ *if* $V_i \notin L$ *then* $\bar{i}$ *else posn* $V_i\, L$ *where posn* $V_i\, L$ *is the position of* V_i *in* L*, counting from the head, which has position* 0.
- $[\![E_1\ E_2]\!]_L \stackrel{\text{def}}{=} [\![E_1]\!]_L\ \$\ [\![E_2]\!]_L$
- $[\![\Lambda\, V_i.\, E]\!]_L \stackrel{\text{def}}{=} A([\![E]\!]_{V_i,L})$

Lemma 2. *For any ordered* L*, there exists a function* $(\!|-|\!)_L : \mathcal{DB}(|L|) \to \mathcal{LE}$ *given recursively by*

- $(\!|\bar{i}|\!)_L \stackrel{\text{def}}{=} V_i$
- $(\!|j|\!)_L \stackrel{\text{def}}{=} elem\, j\, L$ *the* j*th element of* L
- $(\!|D_1\ \$\ D_2|\!)_L \stackrel{\text{def}}{=} (\!|D_1|\!)_L\ (\!|D_2|\!)_L$
- $(\!|A(D)|\!)_L \stackrel{\text{def}}{=} \Lambda\, V_{M+1}.\, (\!|D|\!)_{V_{M+1},L}$ *where* $M \stackrel{\text{def}}{=} Max(D;L)$

$Max(D;L)$ denotes the maximum of the free indices which occur in D and the index j, where V_j is the head of L.

The remainder of the proof involves establishing facts about these functions. It takes great care to ensure all details are correct, and many sub-lemmas are required! We prove that each pair of functions "almost" gives rise to an isomorphism, namely that $[\![(\!|D|\!)_L]\!]_L = D$ and that $(\!|[\![E]\!]_L|\!)_{L'} \sim_\alpha E[L'/L]$. Let $q : \mathcal{LE} \to \mathcal{LE}/{\sim_\alpha}$ be the quotient function. We define $\theta([E]_\alpha) \stackrel{\text{def}}{=} [\![E]\!]_\epsilon$ and $\phi \stackrel{\text{def}}{=} q \circ (\!|-|\!)_\epsilon$, and the theorem follows by simple calculation. Although our proofs take great care to distinguish α-equivalence classes from expressions, we now write simply E instead of $[E]_\alpha$, which will allow us to write $[\![-]\!]_L : \mathcal{LE}/{\sim_\alpha} \leftrightarrows \mathcal{DB}(|L|) : (\!|-|\!)_L$

5.2 Adequacy of Hybrid for the λ-calculus

It would not be possible to provide a "complete" proof of the adequacy of the Hybrid machine tool for λ-calculus. A compromise would be to work with a mathematical description of Hybrid, but even that would lead to extremely long and complex proofs. Here we take a more narrow approach—we work with what amounts to a description of a fragment of Hybrid as a simply typed lambda calculus—presented as a theory in a logical framework as described in Section 1. Before we can state the adequacy theorem, we need a theory in the logical framework which we regard as a "model" of Hybrid. The theory has ground types *expr*, *var* and *bnd*. The types are generated by $\sigma ::= expr \mid var \mid bnd \mid \sigma \Rightarrow \sigma$. We declare constants

$$\begin{array}{rlrl} i :: & var & \mathsf{BND} :: & bnd \Rightarrow expr \\ i :: & bnd & \$\$ :: & expr \Rightarrow expr \Rightarrow expr \\ \mathsf{VAR} :: & var \Rightarrow expr & \mathsf{ABS} :: & expr \Rightarrow expr \end{array}$$

where i ranges over the natural numbers. The objects are generated as in Section 1. Shortage of space means the definitions are omitted—here we aim to give a flavour of our results. Our aim is to prove

Theorem 2 (Representational Adequacy).

- *There is an injective function* $|| - ||_\epsilon : \mathcal{LE}/{\sim_\alpha} \to \mathcal{CLF}_{expr}(\varnothing)$ *which is*
- *compositional on substitution, that is*

$$||E[E'/V_i]||_\epsilon = \mathsf{subst}\ ||E'||_\epsilon\ i\ ||E||_\epsilon$$

where $\mathsf{subst}\ e'\ i\ e$ *is the function on canonical expressions in which* e' *replaces occurrences of* $\mathsf{VAR}\,i$ *in* e*. The definition of meta substitution, and the set* $\mathcal{CLF}_{expr}(\Gamma)$ *of canonical objects in typing environment* Γ *is omitted.*

To show adequacy we establish the existence of the following functions

$$\begin{tikzcd}\mathcal{LE}/{\sim_\alpha} \arrow[r, "|| - ||_L"] & \mathcal{CLF}_{expr}(\beta(L)) & \mathcal{DB}(|L|) \arrow[l, "\Sigma_L"']\end{tikzcd}$$

The function β is a bijection between object level variables V_i and meta variables v_i. Thus $\beta(V_i) = v_i$. The notation $\beta(L)$ means "map" β along L. We write $\beta(L)$ for a typing environment in the framework with all variables of type expression. From now on, all lists L are ordered. The function Σ_L is defined by the clauses

- $\Sigma_L\,\bar{i} \overset{\text{def}}{=} \mathsf{VAR}\,i$
- $\Sigma_L\,i \overset{\text{def}}{=} elem\,i\,\beta(L)$
- $\Sigma_L\,(D_1 \ \$\ D_2) \overset{\text{def}}{=} \Sigma_L\,D_1 \ \$\$\ \Sigma_L\,D_2$
- $\Sigma_L\,A(D) \overset{\text{def}}{=} \mathsf{lambda}\,(\lambda\,v_{M+1}.\,\Sigma_{V_{M+1},L}\,D) = \mathsf{LAM}\,v_{M+1}.\,\Sigma_{V_{M+1},L}\,D$ where $M \overset{\text{def}}{=} Max(D;L)$ (see Lemma 2).

The definition of $||-||_L$ is $\Sigma_L \circ [\![-]\!]_L$. Note that this is a well defined function on an α-equivalence class because $[\![-]\!]_L$ is. In view of Theorem 1, and the definition of $|| - ||_\epsilon$, it will be enough to show the analogous result for the function Σ_ϵ. The proof strategy is as follows. We show that there is a function insts which maps de Bruijn expressions to canonical objects, possesses a left inverse, and is compositional with respect to substitution. Then we prove that the action of Σ_L is simply that of the insts function. Proofs of results will appear in a journal version of this paper.

6 Related Work

Our work could not have come about without the contributions of others. Here, following [28] we classify some of these contributions according to the mathematical construct chosen to model abstraction. The choice has dramatic consequences on the associated notions of recursion and proof by induction.

- De Bruijn syntax [4]: we do not review this further.
- Name-carrying syntax: here abstractions are pairs "(name, expression)" and the mechanization works directly on parse trees, which are quotiented by α-conversion [22, 29, 9]. While recursion/induction is well-supported, the detail

that needs to be taken care of on a case-by-case basis tends to be overwhelming. To partially alleviate this [1] *defines* name-carrying syntax in terms of an underlying type of De Bruijn λ-expressions, which is then used as a metalogic where equality is α-convertibility. Very recently, in [10, 28] Gabbay and Pitts have introduced a novel approach, based on the remarkable observation that a first-order theory of α-conversion and binding is better founded on the notion of name *swapping* rather than renaming. The theory can either be presented axiomatically as formalizing a primitive notion of swapping and *freshness* of names from which binding can be derived, or as a non-classical set-theory with an internal notion of *permutation* of atoms. Such a set-theory yields a natural notion of structural induction and recursion over α-equivalence classes of expressions, but it is incompatible with the axiom of choice. The aim of Pitts and Gabbay's approach is to give a satisfying foundation of the informal practice of reasoning modulo renaming, more than formulate an alternative logical framework for metareasoning (although this too is possible). An ML-like programming language, *FreshML*, is under construction geared towards metaprogramming applications.

– Abstractions as functions from *names* to expressions: mentioned in [1] (and developed in [12]) it was first proposed in [5], as a way to have binders as functions on inductive data-types, while coping with the issue of *exotic* expressions stemming from an inductive characterization of the set of names. The most mature development is Honsell et al.'s framework [17], which explicitly embraces an *axiomatic* approach to metareasoning with HOAS. It consists of a higher-order logic inconsistent with unique choice, but extended with a set of axioms, called the *Theory of Contexts,* parametric to a HOAS signature. Those axioms include the reification of key properties of names akin to *freshness*. More crucially, higher-order induction and recursion schemata on expressions are also assumed. The consistency of such axioms with respect to functor categories is left to a forthcoming paper. The application of this approach to object logics such as the π-calculus [18] succeeds not only because the possibility to "reflect" on names is crucial for the metatheory of operations such as mismatch, but also because here hypothetical judgments, which are only partially supported in such a style [5] are typically not needed. Moreover β-conversion can implement object-level substitution, which is in this case simply "name" for bound variable in a process. The latter may not be possible in other applications such as the ambient calculus. This is also the case for another case-study [23], where these axioms seem less successful. In particular, co-induction is available, but the need to code substitution explicitly makes some of the encoding fairly awkward.
– Abstractions as functions from *expressions* to expressions [26, 15]. We can distinguish here two main themes to the integration of HOAS and induction: one where they coexist in the same language and the other where inductive reasoning is conducted at an additional metalevel. In the first one, the emphasis is on trying to allow (primitive) recursive definitions on functions of higher type while preserving adequacy of representations; this has been real-

ized for the simply-typed case in [7] and more recently for the dependently-typed case in [6]. The idea is to separate at the type-theoretic level, via an S4 modal operator, the *primitive* recursive space (which encompasses functions defined via case analysis and iteration) from the *parametric* function space (whose members are those convertible to expressions built only via the constructors).

On the other side, the *Twelf* project [27] is built on the idea of devising an explicit (meta)metalogic for reasoning (inductively) about logical frameworks, in a fully automated way. $\mathcal{M}_2$ is a constructive first-order logic, whose quantifiers range over possibly open LF object over a signature. In the metalogic it is possible to express and inductively prove metalogical properties of an object logic. By the adequacy of the encoding, the proof of the existence of the appropriate LF object(s) guarantees the proof of the corresponding object-level property. It must be remarked that *Twelf* usage model is explicitly non-interactive (i.e. not programmable by tactics). Moreover, co-inductive definitions have to be rewritten in an inductive fashion, exploiting the co-continuity of the said notion, when possible.

Miller & McDowell [20] introduce a metameta logic, $FO\lambda^{\Delta I\!N}$, that is based on intuitionistic logic augmented with definitional reflection [14] and induction on natural numbers. Other inductive principles are *derived* via the use of appropriate measures. At the metameta level, they reason about object-level judgments formulated in second-order logic. They prove the consistency of the method by showing that $FO\lambda^{\Delta I\!N}$ enjoys cut-elimination [19]. $FO\lambda^{\Delta I\!N}$ approach [20] is interactive; a tactic-based proof editor is under development, but the above remark on co-induction applies.

7 Conclusions and Future Work

The induction principles of Hybrid involve universal quantifications over free variables when instantiating abstractions. It remains future work to determine the real utility of our principles, and how they compare to more standard treatments. Informal practice utilises the some/any quantifier that has emerged formally in [10]. McKinna and Pollack discuss some of these issues in [21].

Several improvements are possible:

We will internalize the well-formedness predicates as abstract types in Isabelle HOL, significantly simplifying judgments over object logics. For example, the subset of lazy λ-calculus expressions identified by predicate isExp will become a type, say *tExp*, so that evaluation will be typed as $\gg\ ::[\,tExp, tExp\,] \Rightarrow bool$. We will specialize the abstr predicate to the defined logic so that it will have type $(tExp \Rightarrow tExp) \Rightarrow bool$.

We envisage eventually having a system, similar in spirit to Isabelle HOL's datatype package, where the user is only required to enter a binding signature for a given object logic; the system will provide an abstract type characterizing the logic, plus a series of theorems expressing freeness of the constructors of such a type and an induction principle on the shape of expressions analogous to the one mentioned in Section 3.

We are in the process of further validating our approach by applying our methods to the compiler optimization transformations for Benton & Kennedy's MIL-lite language [3].

We intend to develop further the theory of Hybrid. Part of this concerns presenting the full details of the material summarized here. There are also additional results, which serve to show how Hybrid relates to λ-calculus. For example, we can prove that if $\mathsf{abstr}\ e$, then there exists $[\Lambda\, V_i.\, E]_\alpha \in \mathcal{LE}/{\sim_\alpha}$ such that $||\Lambda\, V_i.\, E||_\epsilon = \mathsf{LAM}\, v_i.\, e\, v_i$. On a deeper level, we are looking at obtaining categorical characterisations of some of the notions described in this paper, based on the work of Fiore, Plotkin and Turi in [8].

A full journal version of this paper is currently in preparation, which in particular will contain a greatly expanded section on the theory of Hybrid, and provide full details of the case studies.

Acknowledgments We would like to thank Andy Gordon for useful discussions and having provided the HOL script from [1]. We thank Simon Gay for discussions and ideas concerning the π-calculus. Finally, we are very grateful for the financial support of the UK EPSRC.

References

1. A. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In J.J. Joyce and C.-J.H. Seger, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 414–427, Vancouver, Canada, Aug. 1993. University of British Columbia, Springer-Verlag, published 1994.
2. S. Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.
3. N. Benton and A. Kennedy. Monads, effects and transformations. In *Proceedings of the 3rd International Workshop in Higher Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.
4. N. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.
5. J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 124–138, Edinburgh, Scotland, Apr. 1995. Springer-Verlag LNCS 902.
6. J. Despeyroux and P. Leleu. Metatheoretic results for a modal λ-calculus. *Journal of Functional and Logic Programming*, 2000(1), 2000.
7. J. Despeyroux, F. Pfenning, and C. Schürmann. Primitive recursion for higher-order abstract syntax. In R. Hindley, editor, *Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, pages 147–163, Nancy, France, Apr. 1997. Springer-Verlag LNCS.
8. M. Fiore and G. D. Plotkin and D. Turi. Abstract Syntax and Variable Binding. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 193–202, Trento, Italy, 1999. IEEE Computer Society Press.

9. J. Ford and I. A. Mason. Operational Techniques in PVS – A Preliminary Evaluation. In *Proceedings of the Australasian Theory Symposium, CATS '01*, 2001.
10. M. Gabbay and A. Pitts. A new approach to abstract syntax involving binders. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 214–224, Trento, Italy, 1999. IEEE Computer Society Press.
11. S. Gay. A framework for the formalisation of pi-calculus type systems in Isabelle/HOL. In *Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2001*, LNCS. Springer-Verlag, 2001.
12. A. D. Gordon and T. Melham. Five axioms of alpha-conversion. In J. von Wright, J. Grundy, and J. Harrison, editors, *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *Lecture Notes in Computer Science*, pages 173–190, Turku, Finland, August 1996. Springer-Verlag.
13. E. L. Gunter. Why we can't have SML style `datatype` declarations in HOL. In L. J. M. Claese and M. J. C. Gordon, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume A–20 of *IFIP Transactions*, pages 561–568. North-Holland Press, Sept. 1992.
14. L. Hallnas. Partial inductive definitions. *Theoretical Computer Science*, 87(1):115–147, July 1991.
15. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, Jan. 1993.
16. M. Hofmann. Semantical analysis for higher-order abstract syntax. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 204–213, Trento, Italy, July 1999. IEEE Computer Society Press.
17. F. Honsell, M. Miculan, and I. Scagnetto. An axiomatic approach to metareasoning on systems in higher-order abstract syntax. In *Proc. ICALP'01*, number 2076 in LNCS, pages 963–978. Springer-Verlag, 2001.
18. F. Honsell, M. Miculan, and I. Scagnetto. π-calculus in (co)inductive type theories. *Theoretical Computer Science*, 2(253):239–285, 2001.
19. R. McDowell. *Reasoning in a Logic with Definitions and Induction.* PhD thesis, University of Pennsylvania, 1997.
20. R. McDowell and D. Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Transaction in Computational Logic*, 2001. To appear.
21. J. McKinna and R. Pollack. Some Type Theory and Lambda Calculus Formalised. To appear in Journal of Automated Reasoning, Special Issue on Formalised Mathematical Theories (F. Pfenning, Ed.),
22. T. F. Melham. A mechanized theory of the π-calculus in HOL. *Nordic Journal of Computing*, 1(1):50–76, Spring 1994.
23. M. Miculan. Developing (meta)theory of lambda-calculus in the theory of contexts. In S. Ambler, R. Crole, and A. Momigliano, editors, *MERLIN 2001: Proceedings of the Workshop on MEchanized Reasoning about Languages with variable bINding*, volume 58 of *Electronic Notes in Theoretical Computer Scienc*, pages 1–22, November 2001.
24. J. Parrow. An introduction to the pi-calculus. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, pages 479–543. Elsevier Science, 2001.
25. F. Pfenning. Computation and deduction. Lecture notes, 277 pp. Revised 1994, 1996, to be published by Cambridge University Press, 1992.

26. F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.
27. F. Pfenning and C. Schürmann. System description: Twelf — a metalogical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.
28. A. M. Pitts. Nominal logic: A first order theory of names and binding. In N. Kobayashi and B. C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001, Sendai, Japan, October 29-31, 2001, Proceedings*, volume 2215 of *Lecture Notes in Computer Science*, pages 219–242. Springer-Verlag, Berlin, 2001.
29. R. Vestergaard and J. Brotherson. A formalized first-order conflence proof for the λ-calculus using one sorted variable names. In A. Middelrop, editor, *Proceedings of RTA'12*, volume 2051 of *LNCS*, pages 306–321. Springer-Verlag, 2001.