A Comparison of Formalizations of the Meta-Theory of a Language with Variable Bindings in Isabelle

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

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

Abstract. Theorem provers can be used to reason formally about programming languages and there are various general methods for the formalization of variable binding operators. Hence there are choices for the style of formalization of such languages, even within a single theorem prover. The choice of formalization can affect how easy or difficult it is to do automated reasoning. The aim of this paper is to compare and contrast three formalizations (termed *de Bruijn*, *weak HOAS* and *full HOAS*) of a typical functional programming language. Our contribution is a detailed report on our formalizations, a survey of related work, and a final comparative summary, in which we mention a novel approach to a hybrid de Bruijn/HOAS syntax.

1 Introduction

Theorem provers can be used to reason about programming languages. For example, Ambler and Crole have used Isabelle to prove results about a small functional language **PL** [2], by formalizing it within Isabelle/HOL. A key feature of **PL** is that variable binding is pervasive. Further, there are a number of methods which form the theoretical basis for the formalization of variable binding operators, whatever setting they occur in; in particular, there are choices for a formalization of **PL**. The choice of method can have a big effect on how easy or difficult it is to do automated reasoning about **PL**. In more detail we might hope that automated proofs mirror informal mathematical proofs, that we can minimize the difficulty of proofs and the level of infrastructure (such as "book-keeping" lemmas), and maximize genericity and scalability. The aim of this paper is to compare and contrast three formalizations of **PL**, by examining such requirements in each case. It is clearly extremely difficult to objectively compare such requirements between different formalizations of one object level language. Our contribution is to report our experiences as clearly as possible, and we hope our findings will be useful to those working in related areas who wish to make informed choices about the pros and cons of formalizing variable binding.

We proceed by giving three formalizations in Isabelle [25] of the (meta)theory of a small core functional programming language. The types and terms are given respectively by

$$A ::= \mathbf{Nat} \mid A_1 \to A_2$$
$$M ::= x \mid \mathbf{lam}\ x\,.\ M \mid M\ @\ M' \mid \mathbf{z} \mid \mathbf{s}\ M \mid (\mathbf{case}\ M\ \mathbf{of}\ \mathbf{z} \Rightarrow M_1 \mid \mathbf{s}\ x \Rightarrow M_2) \mid \mathbf{rec}\ x\,.\ M$$

and the language has type assignments $\Gamma \vdash M : A$, big step evaluation $M \Downarrow V$ where V is a value, and small step transitions $M \leadsto M'$. The operational semantics is call-by-name; the rules are given in Figure 1. We formalize type assignment, evaluation (natural/big-step) and transition (single step) operational semantics, and prove some basic results, such as the determinacy of operational semantics; subject reduction (run-time type preservation); the progress lemma (well-typed closed non-values do not get stuck at run-time); and the equivalence of big and small step evaluation. These properties are standard, and quite elementary. However, they can be regarded as a benchmark for such core languages as ours (see for example [26, 19, 17]).

We refer to our three methods as *de Bruijn*, *weak HOAS* and *full HOAS*. The de Bruijn method for describing binding operators is very well known [4]. It is exemplified in the Isabelle library by Nipkow's Church-Rosser's proofs [23] (see also for example [31, 2]). The method tackles some of the standard problems (such as α-renaming) associated with name-handling over concrete syntax. However, de Bruijn systems incur

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \mathbf{lam}\ x\ .\ M : A \to B}\ \mathit{tp\_lam} \qquad \frac{\Gamma, x{:}A \vdash M : A}{\Gamma \vdash \mathbf{rec}\ x\ .\ M : A}\ \mathit{tp\_fix}$$

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A}\ \mathit{tp\_var} \qquad \frac{\Gamma \vdash M : B \to A \qquad \Gamma \vdash N : B}{\Gamma \vdash M\ @\ N : A}\ \mathit{tp\_app}$$

$$\frac{}{\Gamma \vdash \mathbf{z} : \mathbf{Nat}}\ \mathit{tp\_}\mathbf{z} \qquad \frac{\Gamma \vdash M : \mathbf{Nat}}{\Gamma \vdash \mathbf{s}\ M : \mathbf{Nat}}\ \mathit{tp\_}\mathbf{s}$$

$$\frac{\Gamma \vdash M : \mathbf{Nat} \qquad \Gamma \vdash M_1 : A \qquad \Gamma, x{:}\mathbf{Nat} \vdash M_2 : A}{\Gamma \vdash \mathbf{case}\ M\ \mathbf{of}\ \mathbf{z} \Rightarrow M_1 \mid \mathbf{s}\ x \Rightarrow M_2 : A}\ \mathit{tp\_case}$$

$$\frac{}{\mathbf{lam}\ x\ .\ M \Downarrow \mathbf{lam}\ x\ .\ M}\ \mathit{eval\_lam} \qquad \frac{M \Downarrow \mathbf{lam}\ x\ .\ E \qquad [N/x]E \Downarrow V}{M\ @\ N \Downarrow V}\ \mathit{eval\_app}$$

$$\frac{}{\mathbf{z} \Downarrow \mathbf{z}}\ \mathit{eval\_}\mathbf{z} \qquad \frac{M \Downarrow V}{\mathbf{s}\ M \Downarrow \mathbf{s}\ V}\ \mathit{eval\_}\mathbf{s} \qquad \frac{[\mathbf{rec}\ x\ .\ M/x]M \Downarrow V}{\mathbf{rec}\ x\ .\ M \Downarrow V}\ \mathit{eval\_fix}$$

$$\frac{M \Downarrow \mathbf{s}\ V' \qquad [V'/x]M_2 \Downarrow V}{\mathbf{case}\ M\ \mathbf{of}\ \mathbf{z} \Rightarrow M_1 \mid \mathbf{s}\ x \Rightarrow M_2 \Downarrow V}\ \mathit{eval\_case\_s} \qquad \frac{M \Downarrow \mathbf{z} \qquad M_1 \Downarrow V}{\mathbf{case}\ M\ \mathbf{of}\ \mathbf{z} \Rightarrow M_1 \mid \mathbf{s}\ x \Rightarrow M_2 \Downarrow V}\ \mathit{eval\_case\_z}$$

$$\frac{}{(\mathbf{lam}\ x\ .\ M)\ @\ N \rightsquigarrow [N/x]M}\ \mathit{step\_beta} \qquad \frac{}{\mathbf{rec}\ x\ .\ M \rightsquigarrow [\mathbf{rec}\ x\ .\ M/x]M}\ \mathit{step\_fix}$$

$$\frac{M \rightsquigarrow M'}{M\ @\ N \rightsquigarrow M'\ @\ N}\ \mathit{step\_app\_src} \qquad \frac{M \rightsquigarrow V}{\mathbf{s}\ M \rightsquigarrow \mathbf{s}\ V}\ \mathit{step\_}\mathbf{s}$$

$$\frac{}{\mathbf{case}\ \mathbf{z}\ \mathbf{of}\ \mathbf{z} \Rightarrow M_1 \mid \mathbf{s}\ x \Rightarrow M_2 \rightsquigarrow M_1}\ \mathit{step\_case\_z}$$

$$\frac{}{\mathbf{case}\ \mathbf{s}\ E\ \mathbf{of}\ \mathbf{z} \Rightarrow M_1 \mid \mathbf{s}\ x \Rightarrow M_2 \rightsquigarrow [E/x]M_2}\ \mathit{step\_case\_s}$$

$$\frac{M \rightsquigarrow M'}{\mathbf{case}\ M\ \mathbf{of}\ \mathbf{z} \Rightarrow M_1 \mid \mathbf{s}\ x \Rightarrow M_2 \rightsquigarrow \mathbf{case}\ M'\ \mathbf{of}\ \mathbf{z} \Rightarrow M_1 \mid \mathbf{s}\ x \Rightarrow M_2}\ \mathit{step\_case\_src}$$

Fig. 1. Static and dynamic semantics

costs of additional programming infrastructure and the notation itself is unreadable. Higher Order Abstract Syntax (HOAS) attempts to deal with some of these problems. It is perhaps less well known, so we give some details—however, we assume readers are broadly familiar with the ideas. Note that HOAS is not a precisely defined system; a number of "systems" fit the title which seems to appear first in [27].

The ideas underlying HOAS go back to Church [3], with his observation that every variable binding operator can be seen in terms of lambda abstraction and also appear in Martin-Löf's theory of arities [24]. The basic idea is that the meta-language into which an object language (such as **PL**) is translated will typically have a single variable binding construct (usually given by function abstraction) and a single definition of infrastructure such as capture-avoiding substitution, along with notions of variable-renaming and freshness of names. This is used to represent the various binding operations found in the object language. In particular, if the object language is altered, one does not need to substantially re-code the infrastructure for binding operations. We introduce some non-standard terminology. A meta-language for HOAS is *full* if

- Object-level bound variables are encoded as meta-level bound variables.
- Object level contexts are encoded as meta-level contexts.
- Object-level substitution is encoded as meta-level β-conversion.

It is called *weak* if the third condition does not hold. In such a case substitution is typically defined as an inductive relation at the object level.

It is a reasonable assumption that the closer a formal derivation mirrors the correspondent informal mathematical proof, the better. To be concrete, let us consider the proof of *subject reduction* for big step evaluation, which is spelled out in every detail for example in [26]. It requires three preliminary lemmata concerning the typing judgment, namely:

1. Typing rules are invertible.
2. They enjoys the structural properties of weakening and exchange.
3. Substitution preserves typing.

After that, it is a straightforward structural induction on the evaluation judgment, using inversion on typing and the substitution lemma for application, case and recursor. Ideally, a formal derivation would reflect this; the script would only state the induction parameter(s), the appropriate introduction and elimination rules (which encode the inversion principle) and the required lemmata; then it should perform a successful fully automated search. This is indeed possible, since *Twelf* would yield such a proof[1]. For example, the proof of subject reduction for weak HOAS in Section 3 requires two additional axioms and would have a slighly higher cost.

We have carried out our study with Isabelle and Isabelle/HOL because they are systems which seem to offer most automation, but many of our observations may apply to other tactics-based systems such as Coq [8]. A direct comparison with a system such as Twelf [28] is complicated by the fact the latter is not biased towards interactive theorem-proving. The paper is organized as follows. In section 2 we report on de Bruijn formalizations in Isabelle/HOL. In Section 3 we move to weak HOAS, by following Despeyroux et al's approach in the Calculus of Inductive Constructions. This too is in Isabelle/HOL. In Section 4 we restrict to the primitive Isabelle language to give a full HOAS encoding in the spirit of LF [14]. In Section 5 we review work related to ours. In Section 6 we summarize the results of our comparisons, and mention a novel approach to a hybrid de Bruijn/HOAS syntax. The Appendix contains some selected code.

2 The de Bruijn Method

The use of de Bruijn notation is a standard technology in the implementation of theorem provers and other systems which represent and manipulate syntax with variable binding [4]. The same technology can be used within a theorem prover to represent and reason about an object language with variable binding. This approach has been used successfully by a number of authors (see for example [23]). In the pure form of

[1] To be pedantic the cost of such a proof would be even lower since in this case the use of full higher-order syntax makes the above-mentioned lemmata unnecessary.

de Bruijn notation, all variables within an expression, whether free or bound, are represented by natural number indices. The work reported in [2] used a pure form of de Bruijn notation to represent the functional programming language fragment **PL** in Isabelle/HOL and reason about its operational semantics. This included a theory of program equivalence based on Abramsky's applicative bisimulation and a mechanization of Howe's proof that this relation is a congruence. Examining the Isabelle theories produced for **PL** reveals both the strengths and the weaknesses of the de Bruijn approach. A clear strength is that the technology is available off-the-shelf: expressions are represented by the elements of an inductive datatype `exp`, operations such as substitution are given by primitive recursive functions, and judgements are given by (co)inductively defined sets. These features are all well supported by packages within Isabelle/HOL. On the other hand, it is a weakness of this approach that the de Bruijn notation differs considerably from that used in informal mathematical exposition. It is difficult, even with experience, to read such expressions. A greater problem is that the index assigned to a variable is given by its position relative to the corresponding binder (in the case of bound variables) or relative to a corresponding occurrence of that variable in an environment (in the case of free variables). The consequence is that any operation which alters the relative position of variables, for example, swapping the position of two variables in the typing environment or substituting one expression into another, necessitates the relabelling of indices. The de Bruijn encoding of **PL**, in common with other theories of this sort, is dominated by a large number of lemmas dealing with the relabelling of indices. These lemmas are more a book-keeping aspect of the representation rather than anything to do with the object language being formalized.

An alternative to the pure form of de Bruijn notation is a mixed form in which bound variables are represented by indices but free variables have explicit names. It is a variant of this form which is used in the implementation of Isabelle itself. For our language we define a datatype of expressions by

$$\begin{array}{rcl}\texttt{datatype exp} & = & \texttt{Var var} \mid \texttt{Bnd bnd} \mid \texttt{Abs exp} \mid \texttt{App exp exp} \mid \texttt{Fix exp} \\ & & \texttt{Zero} \mid \texttt{Succ exp} \mid \texttt{Case exp exp exp}\end{array}$$

The type `var` of free variables and the type `bnd` of bound variable indices are both declared to be the type of natural numbers. Taking `var` as the naturals is *simply* a way to ensure that we have a countably infinite supply of distinct free variables—there are no index computations.

Proofs involving the mixed form of de Bruijn notation are less dependent on lemmas about relabelling of indices. However, the work is transfered elsewhere. As free and bound variables are explicitly distinguished, we require functions to transform a free variable to a bound variable, and to instantiate a bound variable as a free variable or possibly some other expression. These functions are defined by primitive recursion on the structure of expressions, and have types given by

$$\mathit{bind} :: \texttt{var}{\Rightarrow}\texttt{bnd}{\Rightarrow}\texttt{exp}{\Rightarrow}\texttt{exp}$$

$$\mathit{inst} :: \texttt{bnd}{\Rightarrow}\texttt{exp}{\Rightarrow}\texttt{exp}{\Rightarrow}\texttt{exp}$$

The standard presentation of the typing judgement uses an explicit type environment coded as a list. This is not a requirement or a feature specific to the de Bruijn approach, but a choice. One could equally well imagine an implicit treatment of the typing environment in the style of Section 4 with the same advantages but also with the same problems of mathematical foundation. This presentation has the advantage that it is easy to code using the tools already available in Isabelle-HOL. Typing judgements $env \vdash e ::: a$ are given as an inductive definition. In particular, the introduction rule for the type of an abstraction takes the form

$$\frac{\texttt{newIn}\ n\ env \qquad \texttt{newFor}\ n\ s \qquad Cons(n,a)\ env \vdash (\texttt{inst}\ 0\ (\texttt{Var}\ n)\ s) ::: b}{env \vdash (Abs\ s) ::: a \to b}\ \mathit{dbtype\_abs}$$

The predicates `newIn` and `newFor` are used to check that the nth free variable does not occur in the typing environment or the expression s, that is, n is the name of a fresh free variable. The task of reasoning about the typing judgement is quickly dominated by quite routine lemmas about the freshness of names.

In the proof of the subject reduction lemma for 'big-step' evaluation we first have to relate typing, and substitution of an expression for a free variable:

$$\frac{Cons(n,a)\ env \vdash s ::: b \qquad env \vdash t ::: a}{env \vdash s[t//n] ::: b}\ \mathit{SubLemma}$$

The proof is by induction on the derivation of the typing of s and uses a second result, which states that weakening is admissible for the typing judgement:

$$\frac{env \vdash s ::: a \qquad \texttt{newIn}\ i\ env \qquad \texttt{newFor}\ i\ s}{Cons(i,b)\ env \vdash s ::: a}\,TWeak$$

A straightforward proof of this by induction on the derivation of the typing of s runs into problems in choosing suitably fresh names for the variables introduced in the abstraction case. A proof technique which overcomes these difficulties is given by McKinna and Pollack [20]. The trick is to define a second version of the typing judgement in which the introduction rule for an abstraction has a universal quantifier

$$\frac{\forall n\,.\,\texttt{newIn}\ n\ env \to \texttt{newFor}\ n\ s \to Cons(n,a)\ env \vdash \texttt{inst}\ 0\ (\texttt{Var}\ n)\ s ::: b}{env \vdash (Abs\ s) ::: a \to b}\,dbtype\_abs'$$

and the reader should compare this rule to the rule *typeof_abs* in Section 3 and rule *ofLam* in Section 4 which addresses a similar issue in HOAS. Thus the premise states that s instantiated with `Var` n has type b *for all* choices of fresh variable n, rather than some specific choice. One develops a theory of relabelling for the free variables within an expression and uses this to show that the two versions of the typing judgement coincide. The key idea is to show that these typing judgements are preserved by a bijective relabelling of free variables and so, in particular, they are preserved when the names of two free variables are swapped. The cost in terms of programming infrastructure is quite high but the mathematics is relatively straightforward.

It is worth stressing that the approaches described in this section are entirely compatible with classical higher order logic and are purely definitional. It is one of the main tenets of the philosophy amongst users of HOL and Isabelle-HOL that new concepts should, wherever possible, be defined via some representation involving the concepts that are already given. Properties of the new concept can then be derived from the properties of its underlying representation, rather than postulated axiomatically. This ensures the consistency of each extension relative to a small core of the logic. The work needed to achieve this is all done within the theorem prover. An axiomatic approach dispenses with the need to prove derived properties, but must rely on an external mathematical justification (i.e. models) for its consistency (see Section 4).

3 The Weak HOAS Method

In this section we explore Despeyroux et al.'s approach [5] to HOAS, which leads us to a formalization of a weak form of HOAS in Isabelle/HOL. Variable binders are represented as functions in the meta-logic. This "requires" an inductive datatype definition involving function types, such as `datatype exp = Abs (exp⇒exp) | App exp exp ...` which is not allowed because the source `exp` occurs negatively (the associated operator would not be monotone). Thus we introduce a separate type for variables, with an explicit coercion for variables into expressions, and say that the datatype has been *positivized*. This yields the following HOL data-type:

```
datatype exp  =  Var var | Abs (var⇒exp) | App exp exp | Fix (var⇒exp) |
                 Zero | Succ exp | Case exp exp (var⇒exp)
```

It is standard to define a translation function $\ulcorner - \urcorner$ from the object language to expressions in the meta-logic, with object level binding captured by function abstraction; for example $\ulcorner \mathbf{lam}\ x\,.\,x \urcorner = \texttt{Abs}(\lambda x{:}\texttt{var}.\,(\texttt{Var}\ x))$. Operations involving variable binding (i.e. α-renaming etc) are delegated to the meta-level. Further, no lemmas concerning de Bruijn index manipulation are required.

In an inductive framework, the choice of representation of the set of bound variables is crucial. Despeyroux [5] took this to be any set isomorphic to the natural numbers, to realize a possibly infinite number of bound variables. However, taking `var` to be an inductive set will create so-called *exotic* terms. These are terms of type `exp` which do not represent any object-level functional program and hence the translation function can not yield an adequate representation. In fact not only are exotic terms given by case-analysis possible, but there are terms representing "free" variables, such as `Var(Succ Zero)`. However, as pointed out in [10], if the set of variables is *not* inductive, no *closed* exotic term is produced. In fact, in the Calculus of Construction it is possible to leave the set of variables completely unspecified ("parametric" in Coq terminology).

For the formalization of the semantics of our programming language, we do not need to have an infinite supply of names for bound variables. On the contrary we do not need to name them at all. Our only requirement is that the set of bound variables contains at least two distinct elements. In Isabelle/HOL, types need to be non-empty and in fact we identify `var` with a set of at least two elements (we take exactly two) and we add the axiom

$$\frac{}{\forall y{:}\mathtt{var}.\,\lambda x\,.\,(\mathtt{Var}\ x) \neq \lambda x\,.\,(\mathtt{Var}\ y)}\ Var\_neq$$

which is clearly validated by our representation. This ensures that both kinds of closed exotic terms present in Despeyroux et al.'s approach are excluded. However, the primitive language of HOL contains the description operator which creates closed exotic terms such as

$$\mathtt{Abs}(\lambda x\,.\,(\mathtt{if}\ x = (\epsilon w.True)\ \mathtt{then}\ (\mathtt{Var}\ x)\ \mathtt{else}\ (\mathtt{App}\ (\mathtt{Var}\ x)\ (\mathtt{Var}\ x)))$$

Note however that no exotic terms can be *constructively* created: in an intuitionistic version of HOL, all exotic terms would be ruled out.

Isabelle/HOL is a classical set-theory, so we do introduce in Figure 2 an object-level predicate (traditionally called '$\mathtt{valid}_0$') which identifies the class of meta-level terms representing programs (we write upper and lower case characters, i.e. E and e, for second and first order terms respectively). The identification is modulo extensionality—built-in in Isabelle/HOL, although not in Coq. Moreover, validity can be useful in so far as it provides an induction principle over second-order terms. We can prove that substitution is a total relation (which otherwise would be, in general, unprovable). In [29] validity is used to prove the theory of contexts.

$$\frac{}{valid_1\ (\lambda x\,.\,(\mathtt{Var}\ v))}\ v_1\_var \qquad \frac{}{valid_1\ (\lambda x\,.\,(\mathtt{Var}\ x))}\ v_1\_ref$$

$$\frac{valid_1\ E \qquad valid_1\ E'}{valid_1\ (\lambda x\,.\,(\mathtt{App}\ (E\ x)\ (E'\ x)))}\ v_1\_app$$

$$\frac{(\forall u.\,valid_1\ (\lambda x\,.\,(E\ u\ x))) \qquad (\forall u\,.\,valid_1\ (\lambda x\,.\,(E\ x\ u)))}{valid_1\ (\lambda x\,.\,(\mathtt{Abs}\ (E\ x)))}\ v_1\_abs$$

$$\frac{}{valid_0\ (\mathtt{Var}\ v)}\ v_0\_var \qquad \frac{valid_0\ e \qquad valid_0\ f}{valid_0\ (\mathtt{App}\ e\ f)}\ v_0\_app \qquad \frac{valid_1\ E}{valid_0\ (\mathtt{Abs}\ E)}\ v_0\_abs$$

Fig. 2. Validity predicates

Weak HOAS does not obviously extend beyond second-order binding. Consider a third-order operator such as `callcc`, which in full HOAS would have the type
((exp⇒exp)⇒exp)⇒exp. It is not clear how this might be positivized. On the other hand, we can formalize several other binding operators such as logical quantifiers or operators in process-calculi. For instance, in the polymorphic λ-calculus, type abstraction can be given the type (tp⇒exp)⇒exp, while the polymorphic quantifier should be positivized into (var⇒tp)⇒tp.

Substitutions In Weak HOAS, meta-level functions have source type `var` ⇒`exp`; thus substitution of expression for expression cannot be rendered by meta-level β-conversion. Ideally we seek a primitive recursive definition which would be directly applicable to a substitution-based evaluation semantics. It is well known that this is not possible in the traditional setting of inductive definitions [7, 11]. Substitution must be implemented, in the spirit of [22], as an inductive relation with rules such as

$$\frac{}{\mathit{subst}\ (\lambda x\,.\,\mathtt{Var}\ x)\ p\ p}\ \mathit{s\_var1} \qquad \frac{}{\mathit{subst}\ (\lambda x\,.\,\mathtt{Var}\ u)\ p\ (\mathtt{Var}\ \ u)}\ \mathit{s\_var2}$$

$$\frac{\mathit{subst}\ S_1\ t\ U_1 \qquad \mathit{subst}\ S_2\ t\ U_2}{\mathit{subst}\ (\lambda x\,.\,\mathtt{App}\ (S_1\ x)\ (S_2\ x))\ t\ (\mathtt{App}\ U_1\ U_2)}\ \mathit{s\_app}$$

$$\frac{(\forall v\,.\,\mathit{subst}\ (\lambda x\,.\,(M\ x\ v))\ p\ (N\ v))}{\mathit{subst}\ (\lambda x\,.\,(\mathtt{Abs}\ (M\ x)))\ p\ (\mathtt{Abs}\ N)}\ \mathit{s\_abs}$$

In fact establishing the functionality of the relation in Isabelle/HOL turns out to be quite tricky. Getting an appropriate elimination rule is far from immediate, partly because the simplifier (which refines the general elimination rule stemming from the fixed point definition via some free equality rewriting) does not simplify at higher types. Since simply adding extensionality to the reasoner is not practical, we have to establish freeness of constructors manually. While injectivity of constructors is fairly immediate, distinctness needs to rely on the `Var_neq` axiom. Once established and temporarily added to the simplifier, the correct elimination rules are obtained.

The proof of functionality of substitution is a simple structural induction. Instead, the proof that substitution is a total relation requires the *validity* condition for functions, and also the ϵ-operator. We comment on this point further in Section 6.

Typing A key feature of the (Weak) HOAS approach is *that an object-level context $\Gamma \equiv x_1{:}A_1 \ldots x_n{:}A_n$ is represented as a meta-level context.* This provides benefits which we illustrate by examining typing judgements. We can represent typing without explicitly mentioning the typing context as follows. First consider these rules

$$::: \ ::\ (\mathtt{exp}\ *\ \mathtt{tp})\ \mathit{set}$$

$$\mathit{tpv} :: \mathtt{exp} \Rightarrow \mathtt{tp} \Rightarrow \mathtt{bool}$$

$$\frac{\mathit{tpv}\ x\ t}{(\mathtt{Var}\ x)\ :::\ \ t}\ \mathit{typeof\_var} \qquad \frac{e\ :::\ \ (t' \to t) \qquad e'\ :::\ \ t'}{(\mathtt{App}\ e\ e')\ :::\ \ t}\ \mathit{typeof\_app}$$

$$\frac{(\forall x\,.\,\mathit{tpv}\ x\ t \to (E\ x)\ :::\ \ t')}{(\mathtt{Abs}\ E)\ :::\ \ (t \to t')}\ \mathit{typeof\_abs}$$

We extend the representation function by defining $\ulcorner\Gamma\urcorner$ to be the *set* of Isabelle assumptions $\mathit{tpv}\ x_1\ \ulcorner A_1\urcorner \ldots \mathit{tpv}\ x_n\ \ulcorner A_n\urcorner$. Clearly $\ulcorner\Gamma, x{:}A\urcorner = \ulcorner\Gamma\urcorner, \ulcorner x{:}A\urcorner$. Again, note that we have a positivized encoding, achieved by introducing an auxiliary predicate `tpv`. As `tpv` is *not* an inductive predicate, we need to axiomatize its behavior so that it faithfully encodes object-level contexts. In particular, the latter are functional. This is enforced by the axiom *tpv_unique*. Moreover, for meta-reasoning, we need to ensure (axiom *tpv_ex*) that there exists at least one variable for every type:

$$\frac{\mathit{tpv}\ x\ t \qquad \mathit{tpv}\ x\ t'}{t = t'}\ \mathit{tpv\_unique} \qquad \frac{}{\forall t\,.\,\exists\, x.\,\mathit{tpv}\ x\ t}\ \mathit{tpv\_ex}$$

Since the object-level context Γ is represented as a meta-level context, weakening, contraction and exchange come for free. It is essential that the notion of provability is constructive. This will force a proof of $\phi \to \psi$ to a be a proof of ϕ from the assumption ψ, rather than, say the tautology $\neg\phi \vee \psi$. For example, there is a classical proof of $\exists y\,.\,\mathtt{Abs}(\lambda x\,.\,(\mathtt{Var}\ y))\ :::\ s \to s$, although the expression does not correspond to any object-level term.

We now sketch a proof of the adequacy of the typing encoding, w.r.t. *positive intuitionistic* provability:

Theorem 1. $\Gamma \vdash e : A$ *iff* $\ulcorner\Gamma\urcorner \vdash_{I+} \ulcorner e : A\urcorner$.

Proof. ($\rightarrow$) By induction on the judgement $\Gamma \vdash e : A$; we show some crucial cases:

Case: $e \equiv x$

$\Gamma, x{:}A \vdash_{I+} x : A$	By assumption
$\ulcorner\Gamma\urcorner, tpv\ x\ \ulcorner A\urcorner \vdash_{I+} tpv\ x\ \ulcorner A\urcorner$	(Isabelle) Axiom
$\ulcorner\Gamma\urcorner, tpv\ x\ \ulcorner A\urcorner \vdash_{I+} (Var\ x) ::: \ulcorner A\urcorner$	By rule *typeof_var*
$\ulcorner\Gamma\urcorner, \ulcorner x : A\urcorner \vdash_{I+} \ulcorner x : A\urcorner$	By def. of the encoding
$\ulcorner\Gamma, x : A\urcorner \vdash_{I+} \ulcorner x : A\urcorner$	By def. of the encoding

Case: $e \equiv \mathbf{lam}\ x\,.\,e$

$\Gamma, x{:}A \vdash e : B$	By sub-derivation
$\ulcorner\Gamma, x{:}A\urcorner \vdash_{I+} \ulcorner e : B\urcorner$	By IH
$\ulcorner\Gamma\urcorner, \ulcorner x{:}A\urcorner \vdash_{I+} \ulcorner e : B\urcorner$	By above remark
$\ulcorner\Gamma\urcorner, tpv\ x\ \ulcorner A\urcorner \vdash_{I+} \ulcorner e\urcorner ::: \ulcorner B\urcorner$	By def. of the encoding
$\ulcorner\Gamma\urcorner \vdash_{I+} tpv\ x\ \ulcorner A\urcorner \rightarrow \ulcorner e\urcorner ::: \ulcorner B\urcorner$	By impI
$\ulcorner\Gamma\urcorner \vdash_{I+} \forall x{:}var.\, tpv\ x\ \ulcorner A\urcorner \rightarrow \ulcorner e\urcorner ::: \ulcorner B\urcorner$	By allI since x does not appear in $\ulcorner\Gamma\urcorner$
$\ulcorner\Gamma\urcorner \vdash_{I+} Abs(\lambda x{:}var.\ulcorner e\urcorner) ::: (ar\ \ulcorner A\urcorner\ \ulcorner B\urcorner)$	By rule `typeof_abs`
$\ulcorner\Gamma\urcorner \vdash_{I+} \ulcorner\mathbf{lam}\ x\,.\,e : A \rightarrow B\urcorner$	By def. of encoding

Case: $e \equiv e_1\ e_2$. By a direct application of the IH and def. of the encoding

($\leftarrow$) By induction on the derivation of $\pi :: \ulcorner\Gamma\urcorner \vdash_{I+} \ulcorner e\urcorner ::: \ulcorner A\urcorner$, similar to the above

Although inversion (as usual) is provided by the elimination rule(s) of the inductive definition, we do need to prove that substitution preserves typing. Our formalization is directly inherited from [5]:

$$\frac{subst\ E\ p\ n \qquad \exists x\,.\,(\mathtt{Var}\ x) :::\ t' \wedge p\ :::\ t' \rightarrow E\ x :::\ t}{n\ :::\ t}\ SubLemma$$

The mechanized proof mirrors the informal one (induction on substitution and inversion on typing), but with appeal to the `tpv_unique` axioms in the base case.

An issue which did not manifest itself in Despeyroux et al [5] is that sometimes the induction principle generated by ::: is not appropriate, because it will always contain a base case for (free) variables. (This cannot happen in full HOAS—see Section 4.) This turns out to be a problem, for example, when formalizing the progress lemma. Indeed, the lemma holds only for closed expressions which must have empty typing contexts. But contexts in the encoding of ::: are implicit and therefore cannot be explicitly manipulated. Thus we need to introduce another version of typing with an additional argument, to be read as a boolean flag which signals whether the context is empty or not. These typing rules (given below) will update the flag from its initial default value false, to true.[2]

$$typf :: (\mathtt{exp} * \mathtt{tp} * \mathtt{bool})\ set$$

$$\frac{tpv\ x\ t \qquad f = True}{typf\ (\mathtt{Var}\ x)\ t\ f}\ typf\_var \qquad \frac{typf\ e\ (t' \rightarrow t)\ f \qquad typf\ e'\ t'\ f}{typf\ (\mathtt{App}\ e\ e')\ t\ f}\ typf\_app$$

$$\frac{(\forall x.\, tpv\ x\ t \rightarrow typf\ (E\ x)\ t'\ (\lambda y\,.\,True)\ f)}{typf\ (\mathtt{Abs}\ E)\ (t \rightarrow t')\ f}\ typf\_abs$$

We have proven the two typing judgements equivalent with a straightforward derivation.

[2] A more elegant treatment would use a linear context for the same aim.

Evaluation The encoding of operational semantics as inductive relations does not bring in any new ideas. However, note the role of substitution, for example in:

$$>> :: (\texttt{exp} \ * \texttt{exp}) \ set$$
$$\stackrel{1}{\longrightarrow} :: (\texttt{exp} \ * \texttt{exp}) \ set$$

$$\frac{e_1 \ >> \ (\texttt{Abs}\ E) \qquad subst\ E\ e_2\ e_3 \qquad e_3 \ >> \ v_1}{(\texttt{App}\ e_1\ e_2) \ >> \ v_1}\ eval\_app$$

$$\frac{subst\ E\ e2\ e3}{\texttt{App}\ (\texttt{Abs}\ E)\ e_2\ )\stackrel{1}{\longrightarrow} e_3}\ step\_app$$

It is clear that properties of substitution are central in every proof involving evaluation, and moreover no other infrastructure is required in the Weak HOAS setting:

– Determinacy of operational semantics: for evaluation there is a fully automatic proof which uses as expected functionality of substitution plus distinctness of numerals. Its cost is therefore the cost of the former. Similarly for SOS, which also uses the facts that there are no steps possible from values and specialized elimination rules for the latter.
– Subject reduction for evaluation. The proof (by induction on the evaluation judgement) uses as expected introduction and elimination rules for typing and the substitution lemma, but it also requires the `tpv_ex` axiom every time a dynamic typing assumption is needed. The same applies to subject reduction for small step reduction.
– The Progress lemma requires the flagged type judgement, so that we can get rid of the variable case. Further, since we need totality of substitution to fire the β rules, we need to add the *validity* condition.

$$\frac{typf\ e\ t\ f \qquad (f = False) \qquad valid_0\ e}{value\ e \vee \exists e'.\ e\stackrel{1}{\longrightarrow} e}\ Progress$$

4 Full HOAS

The final approach we consider is a way to address full HOAS in a system such as Isabelle. In the LF style, expressions are declared by simply introducing an appropriate signature and *not* a data-type: `exp` and `tp` have kind `term`, and lambda abstraction, for example, receives the second-order type (`exp`⇒`exp`)⇒`exp`. We thus give the following declaration.

$$\begin{array}{rl}
types & \texttt{exp} \\
types & \texttt{tp} \\
arities & \texttt{exp} :: term \\
arities & \texttt{tp} :: term \\
\texttt{App} & :: \texttt{exp}\Rightarrow\texttt{exp}\Rightarrow\texttt{exp} \\
\texttt{Abs} & :: (\texttt{exp}\Rightarrow\texttt{exp})\Rightarrow\texttt{exp} \\
\texttt{Fix} & :: (\texttt{exp}\Rightarrow\texttt{exp})\Rightarrow\texttt{exp} \\
\texttt{Zero} & :: \texttt{exp} \\
\texttt{Succ} & :: \texttt{exp}\Rightarrow\texttt{exp} \\
\texttt{Case} & :: \texttt{exp}\Rightarrow\texttt{exp}\Rightarrow(\texttt{exp}\Rightarrow\texttt{exp})\Rightarrow\texttt{exp}
\end{array}$$

This has the crucial feature that we can easily prove that every α-equivalent object-term is uniquely represented by a *canonical* (that is η-long β-normal) term built out of `App`, `Abs`, `Zero`, `Succ`, `Case`, `Fix`. In particular, there is no construct for bound variables and no exotic term may arise. It is of course vital that its expressive power is essentially the one of the simply-typed λ-calculus and its derivability strictly intuitionistic. We thus step back to Isabelle's *IFOL* (intuitionistic first-order logic).

Relations such as typing or evaluation are not inductively defined. The benefits of "left-recursiveness" show themselves for example in the typing relation (given below), where note how the variable case is left entirely implicit. To stress that we are not in HOL anymore, we use IFOL first-order syntax:

$$of :: \texttt{exp}{\Rightarrow}\texttt{tp}{\Rightarrow}\texttt{bool}$$
$$eval :: \texttt{exp}{\Rightarrow}\texttt{exp}{\Rightarrow}\texttt{bool}$$

$$\frac{of(e_1, t' \to t) \qquad of(e_2, t')}{of(\texttt{App}(e_1, e_2), t)}\, ofApp \qquad \frac{(\forall x\,.\,of(x, t_1) \to of(E(x), t_2))}{of(\texttt{Abs}(\lambda x\,.\,E(x)), t_1 \to t_2)}\, ofLam$$

$$\frac{of(e_1, \mathbf{Nat}) \qquad of(e_2, t) \qquad (\forall x\,.\,of(x, Num) \to of(E3(x), t))}{of(\texttt{Case}(e_1, e_2, E3), t)}\, ofCase$$

It is the eigenvariable condition in the meta-level which replaces the `tpv_unique` axiom in Weak HOAS to guarantee functionality of contexts. The use of meta-level substitution is apparent in the encoding of evaluation:

$$\frac{eval(e_1, \texttt{Abs}(\lambda x\,.\,E(x))) \qquad eval(E(e_2), v)}{eval(\texttt{App}(e_1, e_2), v)}\, evalApp \qquad \frac{eval(E(\texttt{Fix}(E)), v)}{eval(\texttt{Fix}(E), v)}\, evalFix$$

Note how in the case for application E_1 returns a function $\texttt{Abs}(\lambda x\,.\,E(x))$ and the object-level reduction is encoded by application in the logical framework, that is $E(e_2)$.

This is very elegant and well-known as far as object-level reasoning, such as typing or evaluation, but it is clearly not enough for meta-reasoning, which needs at least some form of case analysis and induction. One way to provide the former is to view a relation such as typing as a *Partial Inductive Definition* [13], a generalization of inductive definitions [1] to definiens containing parametric and hypothetical judgements. If $b \Leftarrow G$ is an introduction rule for the predicate a and $\mathcal{D}(a)$ is the collection of the former, then the rule of *Definitional Reflection*

$$\frac{\{\sigma\Gamma, \sigma G \vdash_{\mathcal{D}} A \mid \sigma = mgu(a, b), b \Leftarrow G \in \mathcal{D}(a)\}}{\Gamma, a \vdash_{\mathcal{D}} A}\, \mathcal{D} - L$$

is a proof-theoretic device that allows theories to be considered as "closed" (in the sense of logic programming closed worlds) and thus provides a convenient way to perform case-analysis on judgements. Indeed, it can be seen as the 'mother' of every elimination rule, since, once we endow the meta-logic (i.e. IFOL) with such a rule, the elimination rules (as well as freeness equality of the data-type) of an inductive definition can be *derived*, thus freeing the user from stating and asserting them in the theory.[3] For example, let us derive the *distinction* axiom for numerals, that is $\forall y.\ s(y) \neq 0$:

$$\dfrac{\dfrac{}{0 = s(y) \vdash_{X=X} \bot}\, \mathcal{D} - L}{\vdash_{X=X} 0 = s(y) \to \bot}\, \to R$$

since the definiens of $0 = s(y)$ is empty, by convention identified with $\bot$. Moreover, *injectivity* of successor is proven as follows:

$$\dfrac{\dfrac{\top \vdash_{X=X} x = x}{s(w) = s(y) \vdash_{X=X} w = y}\, \mathcal{D} - L}{\vdash_{X=X} s(w) = s(y) \to w = y}\, \to R$$

[3] For the sake of this study, we have not implemented the rule of definitional reflection yet, so we actually assert the elimination rules as axioms.

where the top sequent is trivially solvable. Further, given, for instance, the definition *val* of *values*, we can derive the approppariate elimination rule:

$$\frac{V = \mathbf{lam}\ x\,.\,E \vdash_{val} C \qquad V = \mathbf{z} \vdash_{val} C \qquad V = (\mathbf{s}\ M), value(M) \vdash_{val} C}{value(V) \vdash_{val} C}\ val\_E$$

As well-known, induction in this setting can cause problems. For example, paradoxes such as non-termination of the logical framework [12] arise when the higher-order encoding of a relation is seen as an inductive type, with corresponding strong elimination rule. It is the rule that allows a definition of the offending terms. Other less dramatic problems, such as the loss of the adequacy of the representation via creation of exotic terms, can come from a naive combination of higher-order induction principles in impredicative systems with the axiom of *unique choice* [16].

The main challenge is to-reintroduce some form of induction. Some recent research [28, 19] has shown that one way to make sense of this is to separate a meta-logic where we formalize our object logic from a meta-meta-logic where we inductively reason about the meta-logic. Miller & McDowell [19] introduce a meta-meta logic, $FO\lambda^{\Delta I\!N}$, that is based on intuitionistic logic augmented with definitional reflection and induction on natural numbers. Other inductive principles are *derived* via the use of appropriate measures. At the meta-meta level, they reason about object-level judgements formulated in second-order logic. They prove the consistency of the method by showing that $FO\lambda^{\Delta I\!N}$ enjoys cut-elimination [18].

Our approach is somewhat more naive than Miller's in that we blur the distinction between meta-meta and meta-logic and we do not formalize an explicit logic of judgements, which we represent directly in an Isabelle. This entails considering judgements directly as *definitions*, which is not allowed in Miller's approach due to the restriction on the cut-elimination proof. Another difference is that we assert structural induction as an *axiom* on any class of judgements. For example, we assert the structural induction principle for typing (restricting here to the $\mathtt{Abs}, \mathtt{App}$ fragment):

$$\frac{\begin{array}{l} eval(b, a) \\ eval(\mathtt{Abs}\ \lambda x\,.\,E(x), \mathtt{Abs}\ \lambda x\,.\,E(x)) \\ eval(e_1, \mathtt{Abs}\ \lambda x\,.\,E(x)) \wedge P(e_1, \mathtt{Abs}\ \lambda x\,.\,E(x)) \wedge P(E(e_2), v_1) \to P(\mathtt{App}(e_1, e_2), v_1) \end{array}}{P(b, a)}\ eval\_induct$$

$$\frac{\begin{array}{l} of(b, a) \\ of(e, (t' \to t)) \wedge P(e, (t' \to t) \wedge of(e', t') \wedge P(e', t') \to P(\mathtt{App}(e, e'), t)) \\ (\forall v\,.\,of(v, t) \to of(E(v), t') \wedge P(E(v), t') \to P(\mathtt{Abs}\ \lambda x\,.\,E(x), t \to t')) \end{array}}{P(b, a)}\ of\_induct$$

(Note that the size of v is not limited, so that this does not present a straightforward induction on the size of terms, and must be justified some other way.) A consequence of this is that the cut-elimination result does not apply directly and thus consistency is not automatically established. Another way to establish consistency of our axioms is via a semantics [17], as we discuss in the next Section 5.

In a robust implementation, there must be some support to automatically infer such principles from the defined judgements. Indeed, it is possible to modify the ML code which produces induction rules for HOL inductive definitions; in fact, the above axioms are simple manual modifications of the principles generated in Weak HOAS. Once this is in place, the proof-scripts tend to shrink dramatically: the structure of the top-level theorems does not alter much, but the required infra-structure does indeed diminish. In fact, they are comparable to *Twelf's* proofs. We find that

- Subject reduction can be proven with *no* preliminaries.
- Determinism of operational semantics: no need for functionality of substitution.
- Progress Lemma: the standard typing judgement is sufficient, and no need for totality of substitution.

5 Related Work

In the literature, there are several large-scale machine-assisted proofs of properties of languages with variable binding using first-order encodings, see for example [31, 15]. Here we review papers that try to overcome the problems of first-order encodings by using some form of HOAS.

We can distinguish two main (and not unrelated) approaches to the integration of HOAS and induction: one "functional" and the other "logical". In the "functional" approach, the emphasis is on trying to allow (primitive) recursive definitions on functions of higher type (cf our substitution function introduced in Subsection 3). The aim is to preserve adequacy of representations, while still allowing "functional programming with higher-order terms". This was first suggested for a fragment of ML by Miller in [21] and was realized first for the simply-typed case in [7] and more recently for the dependently-typed case in [6]. Here 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 distinction and iteration) from the *parametric* function space (whose members are those convertible to terms built only via the constructors). An interpretation of the modal operator in terms of *functor* categories has been given in [16], providing an alternative proof of adequacy. Such categorical semantics can also be used to give a sensible interpretation to higher-order induction principles. Recently, in [11] Gabbay and Pitts have introduced a novel approach to this issue, based on a set-theory with an internal notion of *permutation* of variables. This yields a primitive notion of *freshness* and of *name*-abstraction, so that it can adequately encode object-level syntax modulo α-conversion. This work takes an alternative route to HOAS, with the intention of manipulating *names* of bound variables explicitly, something that HOAS can manage only up to a certain point (but see [17] reviewed later on). Such a set-theory yields a natural notion of structural induction and recursion *over α-equivalence classes of expressions.* On the other hand, the approach shies away from the other two focal points in logical frameworks, namely delegating to the latter object-level substitutions and environment maintainance. An ML-like programming language, *FreshML* is under construction geared towards meta-programming applications. The fertility of such an approach is still to be tested.

On the "logical" side, the *Twelf* project [28] is built on the idea of devising an explicit (meta-)meta-logic, 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 meta-logic is possible to express and inductively prove meta-logical 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. *Twelf* is to date the only implementation of a theorem-prover which successfully marries HOAS and induction, as the fully automated proofs of non-trivial results such as Church-Rosser or cut-elimination testify. In particular, all the results mentioned are proven in a fully automated way with *no* infrastructure. Thus, it represents a benchmark for any other system which aims to use HOAS, although it must be remarked that *Twelf* is explicitly built as a non-interactive system (i.e. not programmable by tactics). $FO\lambda^{\Delta I\!N}$ approach [19] is interactive, and briefly over-viewed in Section 4. While all the derivations mentioned in [19] have been proof-checked via the *Pi* editor [9], an automated tool, code-named *Iris*, is under development.

An even more recent development is Honsell et al. 's Ψ framework [17], which explicitly embraces an *axiomatic* approach to meta-reasoning with HOAS. It consists of classical higher-order logic extended with a set of axioms, called *the theory of contexts,* parametric to a HOAS signature. Those axioms include the reification of some properties of bound variables such as `unsat` $\forall M.\, \exists x.\, x \notin M$, which asserts the existence of fresh names. More crucially, higher-order induction and recursion schemata are also assumed, such as the principle of induction on terms of type `var⇒exp`. The consistency of such axioms w.r.t. functor categories is left to a forthcoming paper. Two main applications have been investigated so far. One is the development of the formal theory of *strong late bisimilarity* in the π-calculus [10]. The other is an implementation for the simply-typed λ-calculus which studies the same properties we have considered. In particular, the proof of totality of substitution relies essentially on the *axiom* of induction on second-order expressions, while the axiom `unsat` is required in the proof of determinacy of substitution (playing the role of our `Var_neq`) Further, extensionality axioms are also needed in the binding cases. After that the script follows ours, confirming the crucial interaction of substitution and bound variables.

In [29] the authors present a Weak HOAS formalization of the π-calculus in Isabelle/HOL. Binding operators are represented as functions from names to processes into processes. Because both channels and messages are *names*, there is no need for object-level substitution. Variables are isomorphic to the natural numbers, hence exotic terms are excluded via *well-formedness* inductive predicates analogous to our validity predicates, the former also providing the only structural induction principle. The paper shows how Honsell's theory of contexts, namely the principles of monotonicity, extensionality and expansion can be *proven*, rather than axiomatized, provided there is an inductive characterisation of the set of variables. Nevertheless, the proof of the last property is quite intricate and requires an embedding into a first-order encoding. Due to the limited scope of the paper, the authors do not employ hypothetical judgements, so none of the above mentioned problems with classical logic arise. At the same time, it makes this problematic to extend to a richer fragment of the calculus.

6 Conclusions

It is now generally accepted that HOAS is a promising tool in meta-reasoning over syntax with binders. Representing bound variables in expressions as meta-level bound variables makes the statement of theorems far closer to established mathematical practice than in de Bruijn notation, and representing object-level environments as meta-level assumptions can lead to elegant proofs. Unfortunately, there is currently no off-the-shelf system which supports HOAS with all of the features that one would like for the formal verification of languages with variable bindings. Such features include structural induction and primitive recursion; co-induction and co-recursion; interactive proofs combined with powerful automatic provers and re-writers; and various kinds of meta-logics. We conclude by summarizing the benefits (B) and costs (C) of each of the three methodologies.

The de Bruijn Method

B The encodings are purely definitional.
B The logic can be classical.
B Utilises and is well supported by (co)inductive datatype packages and (co)recursion.

C Plain de Bruijn notation is unreadable.
C Proofs do not resemble informal mathematical practice.
C Any operation requiring variable transposition requires tedious index relabelling via significant book-keeping infrastructure.
C Mixed de Bruijn is more readable, with a reduction in relabelling infrastructure, but requires coercion functions between free and bound variables which leads to new infrastructure to establish freshness of names.
C Weakening for the typing judgement needs to be proved explicitly, for example in the case of mixed de Bruijn by the method of Pollack and McKinna. The same would apply to any other judgement involving environments.
C Infrastructure must be redone on a case-by-case basis for each object level language.

Following the definitional spirit, we have developed a novel hybrid approach to syntax which bridges the gap between the concrete de Bruijn notation and HOAS notation. This introduces a binding operation `Abs :: (exp => exp) => exp` so that λ-abstractions can be written in the form $\texttt{Abs}(\lambda x\,.\,E\ x)$. However, the meaning of such expressions is reduced to an underlying de Bruijn representation. The use of HOAS is thus a form of syntactic sugar. An expression can be translated freely back and forth between its sugared version and its underlying representation. We have derived some basic properties of expressions written in this HOAS form, for example, the following induction principle over the structure of expressions is derived.

$$\frac{\begin{array}{l} \textit{proper}\ u \\ \forall n.\ P\ (\texttt{Var}\ n) \\ \forall s.\ \forall t.\ (P\ s \wedge P\ t) \rightarrow P\ (\texttt{App}\ s\ t) \\ \forall E.\ (\textit{abstr}\ E \wedge \forall n.\ P\ (E\ (\texttt{Var}\ n))) \rightarrow P\ (\texttt{Abs}\ (\lambda x\,.\,E\ x)) \end{array}}{P\ u}$$

The predicates *proper* and *abstr* are used to rule out exotic terms and can be seen as analogues of Despeyroux's predicates $valid_0$ and $valid_1$.

The state of this work is not sufficiently advanced to be able to prove all of the results discussed in this paper. This is the focus of ongoing research. However, some results have been proved (e.g. determinacy of evaluation and single step reduction). We intend to give a full account of this idea in a forthcoming paper.

The Weak HOAS Method

B Utilises the inductive datatype package.
B Bound variables handled at the meta-level.
B Provides a limited form of implicit context.

C The logic must be intuitionistic.
C One may have to deal with exotic terms and associated validity predicates.
C The only infrastructure required concerns object-level substitution—this must be defined as an inductive relation.
C For some applications, substitution needs to be proven to be a function (Progress lemma), yet totality appears to require the choice operator, which is problematic (see below), or the postulation of a induction principle on second-order expressions.

Hoffman [16] in fact has shown how the principle of *unique choice* (AC!) is troublesome w.r.t. HOAS. While this is avoided in a system such as Coq where AC! is not provable and hence second-order induction on expressions can be assumed, the same cannot be said for HOL. The description axiom entails AC!; more generally, classical typed set-theories do not match very well with the much weaker intuitionistic derivability that HOAS requires.

This applies also to the issue of implicit management of contexts; to begin with the forced positivization of hypothetical judgements in an inductive framework makes them sometimes ineffective, compared to full HOAS, since the auxiliary new predicates which are introduced in place need to be axiomatized in order to be useful in meta-reasoning. For example, the very concise proof of subject reduction in Full HOAS, which relies on the representation of $\Gamma, x{:}A \vdash e : B$ with $\forall x{:}exp.\, of\ (x, A) \to of\ (E(x), B)$ can only be awkwardly simulated by the Weak HOAS encoding $\forall x{:}var.\, tpv\ x\ A \to (E\ x)\ :::\ B \wedge \texttt{tpv\_ex}$. In this sense, we agree with [19]: "These approaches also lessen the power of the meta-level cut rule as a reasoning tool [...]" Moreover, adequacy of the representation of such a judgement holds only constructively. On the other hand, as pointed out by John Harrison (personal communication), it is certainly possible to realize an intuitionistic version of HOL, which would erase those problems. Alas, such a system is not immediately attainable, since the inductive package is heavily based on classical set-theory.

The Full HOAS Method

B Substitution is captured by meta-level application and beta conversion.
B Embedded implication can be used to full effect.

C The logic must be intuitionistic.
C Induction and case analysis should be moved to an additional meta-level to ensure consistency. A straightforward two-level approach such the one in Section 4 is meta-theoretically uncertain.
C It is difficult to allow (primitive) recursion on higher-order syntax and especially to combine it with induction over open terms [30].
C No support for co-induction at this time.

The simulation of Full HOAS in Isabelle's IFOL seems successful. Indeed, the proofs of subject reduction, determinism of operational semantics, and the progress lemma resemble informal mathematical practice very closely. However, much of the hard work has been moved elsewhere, namely to the justification of the inductive principles (of course this is a one-time-only effort).

In summary, the practitioner who wishes to verify significant properties of a language with variable binding using machine assistance is faced with some uncomfortable choices: either rely on trusted but labour-intense first-order technology such as variants of de Bruijn, or adopt an HOAS approach. In this case there is no single framework which is endowed with all the reasoning tools we may require.

The Isabelle code described in this paper can be found at

`http://www.mcs.le.ac.uk/~amomigliano/isabelle/compar.html`

Acknowledgements. We wish to thank Amy Felty and Marino Miculan for having made available to us the Coq proof scripts associated with [5] and [17], respectively.

References

1. P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, chapter C.7, pages 739–782. North-Holland, Amsterdam, 1977.
2. S. J. Ambler and R. L. Crole. Mechanised Operational Semantics via (Co)Induction. In *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, volume 1690 of *Lecture Notes in Computer Science*, pages 221–238. Springer-Verlag, 1999. .
3. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
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. G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.
9. L.-H. Eriksson. Pi: An interactive derivation editor for the calculus of partial inductive definitions. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 821–825, Nancy, France, June 1994. Springer Verlag LNAI 814.
10. I. S. Furio Honsell, Marino Miculan. π-calculus in (co)inductive type theories. *Theoretical Computer Science*, 2(253):239–285, 2001.
11. 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, July 1999. IEEE Computer Society Press.
12. E. Gimenez. A tutorial on recursive types in Coq. Technical Report RT-0221, Inria, Institut National de Recherche en Informatique et en Automatique, 1998.
13. L. Hallnas. Partial inductive definitions. *Theoretical Computer Science*, 87(1):115–147, July 1991.
14. 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.
15. D. Hirschkoff. A full formalization of pi-calculus theory in the Calculus of Constructions. In E. Gunter and A. Felty, editors, *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)*, pages 153–169, Murray Hill, New Jersey, Aug. 1997.
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. S. I. Honsell F., Miculan M. An axiomatic approach to metareasoning on systems in higher-order abstract syntax. 2001. Submitted.
18. R. McDowell. *Reasoning in a Logic with Definitions and Induction*. PhD thesis, University of Pennsylvania, 1997.
19. R. McDowell and D. Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Transaction in Computational Logic*, 2001. To appear.

20. J. McKinna and R. Pollack. Some lambda calculus and type theory formalized. *JAR*, 1998.
21. D. Miller. An extension to ML to handle bound variables in data structures: Preliminary report. In *Informal Proceedings of the Logical Frameworks BRA Workshop*, June 1990.
22. D. Miller. Unification of simply typed lambda-terms as logic programming. In K. Furukawa, editor, *Eighth International Logic Programming Conference*, pages 255–269, Paris, France, June 1991. MIT Press.
23. T. Nipkow. More Church-Rosser proofs (in Isabelle/HOL). *JAR*, 2000. To appear.
24. B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction.* Oxford University Press, 1990.
25. L. C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer-Verlag LNCS 828, 1994.
26. F. Pfenning. Computation and deduction. Unpublished lecture notes, 277 pp. Revised May 1994, April 1996, May 1992.
27. 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.
28. F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical 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.
29. H. D. Rockl C. and B. S. Higher-order abstract syntax with induction in Isabelle/HOL: Formalizing the π-calculus and mechanizing the theory of contexts. In *FOSSACS'01*. Forthcoming, 2001.
30. C. Schürmann. *Automating the Meta-Theory of Deductive Systems.* PhD thesis, Carnegie-Mellon University, 2000. CMU-CS-00-146.
31. M. VanInwegen. *The Machine-Assisted Proof of Programming Language Properties.* PhD thesis, University of Pennsylvania, Aug. 1996.

A Weak HOAS

```
typedef var = "{True,False}"

datatype tp =  Num | Arrow tp tp ("(_) /-> (_)" [200,201] 200)

datatype exp =  Var var |
                App exp exp  ("(_) /$ (_)" [200,201] 200) |
                Abs " var => exp"  (binder "Lam " 190)|
                Z |
                S exp  | Case exp  exp "var => exp"  |
                Fix  " var => exp"   (binder "Rec " 190)

inductive substitution
intrs
subst_var1 "subst (% v. Var v) p p"
subst_var2 "subst (% v. Var u) p (Var u)"
subst_app  "[| subst s1 t u1; subst s2 t u2 |]
            ==> subst (% v. App (s1 v) (s2 v)) t (App u1 u2)"
subst_abs  "(! v . subst (% x . (M x v)) p (N v) )
            ==> subst (% x. (Abs (M x))) p (Abs N)"
subst_fix   "(! v . subst (% x . (M x v)) p (N v) )
            ==> subst (% x. (Fix (M x))) p (Fix N)"
subst_z     "subst (% x . Z) p Z"
subst_s     "subst s1 t u1 ==> subst (% v. S (s1 v) ) t (S u1)"
subst_case  "[|subst e1 p ne1;subst e2 p ne2 ;(! v .subst (% x . (M x v)) p (N v))|]
             ==> subst(% x. (Case (e1 x) (e2 x) (M x))) p (Case ne1 ne2 N)"

inductive valid1
intrs
valid1_var "valid1 (%x . (Var v))"
```

```
valid1_ref "valid1 (%x .(Var x))"
valid1_app "[|valid1 e ; valid1 e'|] ==> valid1 (%x .(App (e x) (e' x)))"
valid1_abs "[| (!u. valid1 (%v . (e u v)));(!u. valid1 (%v . (e v u)))|]
            ==> valid1 (%x. (Abs (e x)))"
valid1_fix "[| (!u. valid1 (%v . (e u v)));(!u. valid1 (%v . (e v u)))|]
            ==> valid1 (%x. (Fix (e x)))"
valid1_z   "valid1 (%x . Z)"
valid1_s   "valid1 e ==> valid1 (%x .(S (e  x)))"
valid1_case"[|(!u. valid1 (%v .(e u v)));(!u. valid1 (%v .(e v u))); valid1 e1;valid1 e2|]
            ==> valid1 (%x. (Case (e1 x) (e2 x) (e x)))"

inductive valid0
intrs
valid0_var  "valid0 (Var v)"
valid0_app  "[|valid0 a;valid0 b|] ==> valid0 (App a b)"
valid0_abs  "valid1 E ==> valid0 (Abs E)"
valid0_fix  "valid1 E ==> valid0 (Fix E)"
valid0_z    "valid0 Z"
valid0_s    "valid0 a ==> valid0 (S a)"
valid0_case "[|valid0 a; valid0 b; valid1 E|] ==> valid0 (Case a b E)"

inductive typeof
intrs
typeof_var  "tpv x t ==> (Var x) ::: t"
typeof_app  "[| e ::: ( t' -> t); e' ::: t'|] ==> (App e e') ::: t"
typeof_abs  "(!x:: var. tpv x t --> (E x) ::: t') ==> (Abs E) ::: ( t -> t')"
typeof_fix  "(!x:: var. tpv x t --> (E x) ::: t ) ==> (Fix E) ::: t"
typeof_z    "Z ::: Num"
typeof_s    "e ::: Num ==> (S e) ::: Num"
typeof_case "[|e1 ::: Num; e2 ::: t; (!x. tpv x Num --> (E3 x) ::: t)|]
              ==> (Case e1 e2 E3) ::: t"

inductive typf
intrs
typf_var  "[|tpv x t; f = True|] ==> typf (Var x) t f"
typf_app  "[|typf e (t' -> t) f;typf e' t' f|] ==> typf (App e e') t f"
typf_abs  "(!x .tpv x t --> typf (E x) t' (% y . True) f) ==> typf (Abs E) (t -> t') f"
typf_fix  "(!v .tpvar v t --> typf (E v) t (truef f)) ==> typf (Fix E) t f"
typf_z    "typf Z Num b"
typf_s    "typf e Num b ==> typf (S e) Num b"
typf_case "[|typf e1 Num b;typf e2 t b; (!x .tpvar x Num --> typf (E3 x) t (truef b))|]
           ==> typf (Case e1 e2 E3) t b"

inductive eval
intrs
eval_abs    "Abs E >> Abs E"
eval_fix    "[|subst E (Fix E) F;F >> V|] ==> Fix E >> V"
eval_app    "[|e1 >> (Abs E); subst E e2 e3;e3 >> v1|] ==> App e1 e2 >> v1"
eval_z      "Z >> Z"
eval_s      "e >> v ==> S e >> S v"
eval_case_z "[|e1 >> Z;e2 >> v|] ==> Case e1 e2 e3 >> v"
eval_case_s "[|e1 >> S e;subst e3 e n n >> v|] ==>  Case e1 e2 e3 >> v"
```

```
inductive step
intrs
step_src  "e1 >>> e1' ==> App e1 e2 >>> App e1' e2"
step_app  "subst E e2 e3 ==> App (Abs E) e2 >>> e3"
step_z    "Case Z E1 E2 >>> E1"
step_s    "[|value V;subst E2 V E3|] ==> Case (S V) E1 E2 >>> E3"
step_fix  "subst E (Fix E) N ==> Fix E >>> N"
step_S    "E >>> E' ==> S E >>> S E'"
step_case "E1 >>> E1' ==> Case E1 E2 E3 >>> Case E1' E2 E3"
```

Sample Proofs

```
val iset =(claset() delrules [select_equality,disjCI,impCE,iffCE])
                    addSEs [impE,iffE];

Goal "(e >> v) ==> !t.  (e ::: t) --> (v ::: t)";
be eval.induct 1;

by (REPEAT((cut_facts_tac [tpvar_ex] 1) THEN (fast_tac (iset
  addss(simpset() addsimps [sametype_def])
  addEs [typeof_App_E,typeof_Abs_E,typeof_Fix_E,typeof_z_E,typeof_s_E,typeof_case_E]
  addIs typeof.intrs addDs [subst_lemma_mml]  ) 1)));
qed_spec_mp "subject_reduction_mml";

Goal "[|typf e t l |]   ==> (l=False) --> valid0 e --> value e | (? e'  . e >>> e')";
be typf.induct 1;
(* Case *)
by(blast_tac (claset() addDs[subst_ex]
                       addEs valid0_E@[value_E,typf_Abs_E]
                       addIs  [step.step_z,step.step_s,step.step_case]) 7);
by(ALLGOALS(blast_tac (claset()
                      addDs[subst_ex]
                      addEs valid0_E@[value_E,typf_Abs_E,typf_z_E,typf_s_E]
                      addIs step.intrs@value.intrs)));
qed "progressCBNf";
```

B Full HOAS

```
NExp = IFOL + FOL+
types  exp
types  tp
arities exp :: term
arities tp :: term
consts
        App :: exp => exp => exp
        Lam :: (exp => exp ) => exp (binder "lam " 10)
        Fix :: (exp => exp ) => exp (binder "rec " 10)
        Z :: exp
        S :: exp => exp
        Case :: exp =>  exp => (exp => exp) => exp
```

```
        Num :: tp
         "->" :: [tp, tp] => tp                (infixl 30)

        eval :: [exp, exp ]=> o
        of :: [exp, tp ]=> o
        ">>>" :: [exp, exp ]=> o               (infixl 50)

rules

ofApp  "[|of(e1,t'->t);of(e2,t')|] ==> of(App(e1,e2),t)"
ofZ    "of(Z,Num)"
ofS    "of(e,Num) ==> of(S(e),Num)"
ofLam  "[|(ALL x . of(x,t1) --> of (E(x),t2))|] ==> of((lam x . E(x)),t1 -> t2)"
ofFix  "[|(ALL x . of(x,t) --> of(E(x),t))|] ==> of((rec  x . E(x)),t)"
ofCase "[|of(e1,Num);of(e2,t);(ALL x. of(x,Num) --> of(E3(x),t))|]
        ==> of(Case(e1,e2,E3),t)"

(* Elimination rules: admissible *)
ofZ_E    "[|of(Z,t);t=Num ==> P |] ==> P"
ofS_E    "[|of(S(e),t); [|of(e,Num);t=Num|] ==> P |] ==> P"
ofApp_E  "[|of(App(e1,e3),t);[|of(e1,(t' -> t));of(e3,t')|]
         ==> P |] ==> P"
ofLam_E  "[|of(Lam(E),T);[|ALL v. of(v,t1) --> of(e(v),t2);T= (t1 -> t2)|]
          ==> P |]   ==> P"
ofCase_E "[|of(Case(e1,e2,e),t);[|of(e1,Num);of(e2,t);ALL x. of(x,Num) --> of(e(x),t)|]
          ==> P |] ==> P"
ofFix_E  "[|of(rec x . e(x),t); ALL v. of (v, t) --> of (e(v),t)  ==> P |] ==> P"

of_induct "[|of(xb,xa);
          !!E t.
          ALL v. of( v, t) --> of(E(v),t) & P(E(v),t) ==> P(rec x . E(x),t);
          !!E3 e1 e2 t.
          [|of(e1,Num);P(e1,Num);of(e2,t);P(e2,t);
            ALL x. of(x,Num) --> of(E3(x),t) & P(E3(x),t)|]
          ==> P(Case(e1,e2,E3),t);
          P(Z,Num);
          !!e.
          [|of(e,Num);P(e,Num)|] ==> P(S(e),Num);
          !!e e' t t'.
          [|of(e,(t' -> t));P(e,(t'-> t));of(e',t');P(e',t')|]
          ==> P(App(e,e'),t);
          !!E t t'.
          ALL v. of(v,t) --> of(E(v),t') & P(E(v),t') ==> P(lam x . E(x),t -> t')|]
          ==> P(xb,xa)"

stepBeta     "App(lam x  . E(x),F) >>> E(F)"
stepCong     "e1 >>> e1' ==> App(e1,e2) >>> App(e1',e2)"
stepS        "E >>> E' ==> S(E) >>>  S(E')"
stepCaseZ    "Case (Z,E1,E2) >>>  E1"
stepCaseS    "[|value(V)|] ==> Case(S(V),E1,E2) >>>  E2(V)"
stepCaseCong "E1 >>> E1' ==> Case(E1,E2,E3) >>>  Case(E1',E2,E3)"
stepFix      "Fix(E) >>> E(Fix(E))"
```

```
(* Elimination rules: omitted *)

step_induct "[|xb >>> xa;
            !!E . P(Fix(E),E(Fix(E)));
            !!E1 E2. P(Case(Z,E1,E2),E1);
            !!E1 E2 E3 V.
            [| value(V)|] ==> P(Case(S(V),E1,E2),E2(V));
            !!E1 E1' E2 E3.
            [|E1 >>> E1';P(E1,E1')|]
            ==> P(Case(E1,E2,E3),Case(E1',E2,E3));
            !!e1 e1' e2.
            [|e1>>> e1'; P(e1,e1')|] ==> P(App(e1,e2),App(e1',e2));
            !!E e2.
            P(App(lam x . E(x),e2),E(e2));
            !!E E'.
            [|E >>> E';P(E,E')|] ==> P(S(E),S(E'))|]
            ==> P(xb,xa)"

evalLam   "eval((lam x. E(x)),lam x . E(x))"
evalApp   "[|eval(E1,(lam  x . E(x)));eval((E(E2)),V)|] ==> eval(App(E1,E2),V)"
evalZ     "eval(Z,Z)"
evalS     "eval(e,v) ==> eval(S(e),S(v))"
evalCaseZ "[|eval(e1,Z);eval(e2,v)|] ==> eval(Case(e1,e2,e3),v)"
evalCaseS "[|eval(e1,(S(e)));eval(e3(e),v)|] ==> eval(Case(e1,e2,e3),v)"
evalFix   "eval(E(Fix(E)),V)==> eval(Fix(E),V)"

(* Elimination rules: omitted *)

eval_induct
        "[|eval(b,a);
        !!E.
       P((lam x .  E(x)),((lam x .  E(x))));
        !!E  V.
        [|eval(E(Fix(E)),V);P(E(Fix(E)),V)|] ==> P(Fix(E),V);
        !!E e1 e2 e3 v1.
        [|eval (e1, (lam x .  E(x)));P(e1,(lam x . E(x)));P((E(e2)),v1)|]
        ==> P(App(e1,e2),v1);P(Z,Z);
        !!e v.
        [|eval(e,v);P(e,v)|] ==> P(S(e),S(v));
        !!e1 e2 e3 v.
        [| val(e1,Z);P(e1,Z);eval(e2,v);P(e2,v)|]
         ==> P(Case(e1,e2,e3),v);
        !!e e1 e2 e3  v.
        [|eval(e1,S(e));P(e1,(S(e)));eval(e3(e),v);P(e3(e),v)|]
        ==> P(Case(e1,e2,e3),v)|]
        ==>  P(b,a)"
```

Sample Proofs

```
Goal "eval (e, v) ==> ALL t.  of( e, t) --> of( v, t)";
be eval_induct 1;
```

```
(* Fix *)
strip 2;
be ofFix_E 2;
by (forw_inst_tac [("x","Fix(E)")] spec 2);
by(fast_tac (iset addIs [ofFix]) 2);
by(ALLGOALS(fast_tac (iset addEs of_E addIs of_I ) ));
qed "hosubred";

(* progress lemma-- seems to need blast, but hand proof is int.  *)

Goal "of( e, t)    ==>  value( e) | (EX  e'  . e >>> e')";
be of_induct 1;
by(cut_facts_tac [tp_distinct] 2); (* Case*)
by(cut_facts_tac [tp_distinct] 5); (* App *)
by(ALLGOALS(blast_tac (claset() addEs of_E@[val_E]  addIs step_I@value_I ) ));
qed "hoprogress";
```