An Improved Implementation and Abstract Interface for Hybrid

Alan J. Martin
Department of Mathematics and Statistics,
University of Ottawa, Canada
amart045@site.uottawa.ca

Amy P. Felty
School of Electrical Engineering and Computer Science and
Department of Mathematics and Statistics,
University of Ottawa, Canada
afelty@site.uottawa.ca

Hybrid is a formal theory implemented in Isabelle/HOL that provides an interface for representing and reasoning about object languages using higher-order abstract syntax (HOAS). This interface is built around an HOAS variable-binding operator that is constructed definitionally from a de Bruijn index representation. In this paper we make a variety of improvements to Hybrid, culminating in an abstract interface that on one hand makes Hybrid a more mathematically satisfactory theory, and on the other hand has important practical benefits. We start with a modification of Hybrid’s type of terms that better hides its implementation in terms of de Bruijn indices, by excluding at the type level terms with dangling indices. We present an improved set of definitions, and a series of new lemmas that provide a complete characterization of Hybrid’s primitives in terms of properties stated at the HOAS level. Benefits of this new package include a new proof of adequacy and improvements to reasoning about object logics. Such proofs are carried out at the higher level with no involvement of the lower level de Bruijn syntax.

1 Introduction

Hybrid is a system developed to specify and reason about logics, programming languages, and other formal systems expressed in higher-order abstract syntax (HOAS). It is implemented as a formal theory in Isabelle/HOL [15]. By providing HOAS in a modern proof assistant, Hybrid automatically gains the latter’s capabilities for meta-theoretical reasoning. This approach is intended to provide advantages in flexibility and proof automation, in contrast to systems that directly implement logical frameworks, which must build their own meta-reasoning layers from the ground up. Building a system such as Hybrid within a general purpose theorem prover poses a variety of challenges. Our goal in this work is to improve the implementation and interface of Hybrid’s basic theory, bringing it to a point where its potential advantages can be more fully realized.

Using HOAS, binding constructs in the represented language (the object logic or OL) are encoded using the binding constructs provided by an underlying λ-calculus or function space of the meta-logic, thus representing the arguments of these constructs as functions of the meta-level. Isabelle/HOL implements an extension of higher-order logic, where the function types are “too large” for HOAS in two senses. First, they contain elements with irreducible occurrences of logical constants, which do not represent syntax. Second, the function space \( \tau \rightarrow \tau \) has larger cardinality than \( \tau \), so a variable-binding operator represented as a functional \( \Phi \) of type \( (\tau \rightarrow \tau) \rightarrow \tau \) cannot be injective. This makes it unsuitable for syntax, for we cannot uniquely recover the argument \( F \) from a term of the form \( \Phi(F) \). Our work builds directly on the original Hybrid system [1], whose solution to both problems is to use only a subset of the function type, identified by a predicate called abstv. It builds a type \( \text{expr} \) of terms with an HOAS variable-binding operator definitionally in terms of a de Bruijn index representation.
In earlier work joint with Alberto Momigliano, we gave a system presentation of Hybrid [14], which built on the original Hybrid and serves as a starting point for the work presented here. In this paper, we fill in many details that could not be described in a short system description, as well as make significant further improvements, allowing us to complete a characterization of Hybrid’s type $expr$ in terms of properties stated at the HOAS level. In the new Hybrid, the type $expr$, its constructors, and these properties form an abstract interface that allows users to reason at the higher level with no involvement of the lower level implementation details. This interface was motivated by and is illustrated by a new proof of representational adequacy for Hybrid [12, Sect. 3.4] that does not make any reference to de Bruijn syntax.

We start in Sect. 2 by giving an abstract view of Hybrid that motivates and explains the interface. Sections 3–7 fill in many of the details of its implementation. The type $dB$ implementing the de Bruijn index representation is defined in Sect. 3, along with a predicate level to keep track of dangling indices. The original Hybrid [1] used a datatype corresponding to our $dB$ directly as $expr$. Section 4 defines the new version of $expr$, which excludes at the type level terms with dangling indices. This simplifies the representation of object languages by eliminating the need to carry a predicate for this purpose (called proper in [1]) along with Hybrid terms in meta-theoretic reasoning. Section 5 defines Hybrid’s variable binding operator $\text{LAM}$ and the $\text{abstr}$ predicate. These definitions support a stronger injectivity property, presented in Sect. 6 with only one $\text{abstr}$ premise rather than two. This property was also proved in [14]; the results here generalize and simplify these definitions as well as simplify other related Hybrid internals. (In particular, we eliminate the need for the auxiliary function $dB_{\text{fn}}$ defined in [14] using the function package first introduced in Isabelle/HOL 2007, and we eliminate some other auxiliary functions by using a more systematic treatment of $\text{level}$.)

In Sect. 7, we formally prove that a version of $\text{abstr}$ for two-argument functions (as described in [13]) is equivalent to a conjunction of one-argument $\text{abstr}$ conditions on “slices” of the function (fixing one argument). We use this result to prove a case-distinction lemma for functions satisfying $\text{abstr}$, and a lemma that enables compositional proof of $\text{abstr}$ conditions at the HOAS level, without conversion to de Bruijn indices as required in [1]. These two lemmas represent important new results that complete the abstract interface for Hybrid.

In Sect. 8, we discuss related work as well as ongoing work with Hybrid.

The Isabelle/HOL 2011 theory file for the present version of Hybrid is available online at:

http://hybrid.dsi.unimi.it/download/Hybrid.thy

and a more thorough presentation can be found in the first author’s Ph.D. thesis [11, 12]. In addition to the results described here, this theory file also replaces tactic-style proofs of the original version of Hybrid with Isar proofs. This style of proof is both more readable and more robust against changes to the underlying proof assistant. It also includes rewrite rules for Isabelle’s simplifier to convert automatically between HOAS at type $expr$ and de Bruijn indices at type $dB$. With the improvements allowing users to work exclusively at the HOAS level, this is no longer needed, and only included for illustrative purposes.

2 An Abstract View of Hybrid

We use a pretty-printed version of Isabelle/HOL concrete syntax in this and the following sections. A double colon :: separates a term from its type, and the arrow $\Rightarrow$ is used in function types. We stick to the usual logical symbols for connectives and quantifiers ($\neg$, $\land$, $\lor$, $\rightarrow$, $\forall$, $\exists$). Free variables (upper-case) are implicitly universally quantified (from the outside). The sign $\equiv$ (Isabelle meta-equality) is used for equality by definition, and $\Longrightarrow$ for Isabelle meta-level implication. In the notation $\left[ P_1; \ldots ; P_n \right] \Longrightarrow$
The problem with the above definition is that Hybrid only approximates one such pseudo-datatype, not the datatype package. Hybrid may be viewed as an attempt to approximate a datatype definition that is not well-formed because of its higher-order features:

\[
\texttt{datatype} \ expr = \texttt{CON} \ con \mid \texttt{VAR} \ var \mid \texttt{APP} \ expr \ expr \quad \text{(notation \( \langle s \ $$ t \rangle \))}
\mid \texttt{LAM} \ (expr \Rightarrow \ expr) \quad \text{(notation \( \lambda \ (LAM \ x. \ B) \))}
\]

where \( \texttt{CON} \) represents constants, from an OL-specific type \( \texttt{con} \) (typically a trivial datatype); \( \texttt{VAR} \) may be used to represent free variables, from a countably infinite type \( \texttt{var} \) (actually a synonym for \texttt{nat}); \( \texttt{APP} \) represents pairing, which is sufficient to encode list- or tree-structured syntax; and \( \texttt{LAM} \) represents variable binding in HOAS style, using the bound variable of an Isabelle/HOL \( \lambda \)-abstraction to represent a bound variable of the object language.\(^1\)

It should be noted that Hybrid only approximates one such pseudo-datatype, not the datatype package with its ability to define multiple types for first-order abstract syntax. That is, Hybrid is untyped, so predicates rather than types must be used to distinguish different kinds of OL terms encoded into expr.

The problem with the above definition is \( \texttt{LAM} \), whose argument type includes a negative occurrence of \texttt{expr} (underlined above). This is essential for HOAS, but it is not permitted in a datatype definition [16, Sect. 2.6], and it will require modifications to some of the properties expected for a constructor of a datatype; we will return to this issue later.

Hybrid does provide a type \texttt{expr} with operators \texttt{CON}, \texttt{VAR}, \texttt{APP}, and \texttt{LAM} of the appropriate types. This type and the latter three operators can be used directly as a representation of the untyped \( \lambda \)-calculus. When encoding OLs in general, however, it is usual to represent each OL construct as a list built using $$ and headed by a \texttt{CON} term identifying the particular construct. To illustrate this idea, we take the untyped \( \lambda \)-calculus as our OL with its usual named-variable syntax, using capital letters for variables \( (V_i, i \in \mathbb{N}) \) and \( \lambda \)-abstraction \( (A) \) to avoid confusion with Isabelle’s \( \lambda \) operator. In this form, an object language term \( (A \, V_1, A \, V_2, (V_1 \, V_2) \, V_3) \), for example, can be represented as

\[
c_{\text{lam}} \, $$ \, (LAM \, x. \, c_{\text{lam}} \, $$ \, (LAM \, y. \, C_{\text{app}} \, $$ \, (C_{\text{app}} \, $$ \, x \, $$ \, y) \, $$ \, (VAR \, 3)),
\]

where \( c_{\text{lam}} = \texttt{CON} \, c_1 \) and \( c_{\text{app}} = \texttt{CON} \, c_2 \) for distinct constants \( c_1, c_2 :: \texttt{con} \). We may use Isabelle’s ability to define abbreviations and infix notations to recover a reasonable concrete syntax:

\[
fn \, x. \, (x \, $$ \, y) \, $$ \, (VAR \, 3).
\]

Note that although de Bruijn indices do not appear in such terms, numbers can appear as arguments to Hybrid’s \texttt{VAR} operator, which is included to allow a representation of free variables that is distinct from bound variables.

We now turn to the properties required of \texttt{expr} and its operators to function as HOAS. We motivate the requirements by considering adequacy, an important meta-theoretic property. This can take several forms, but the proof presented in [12] uses bijectivity of a set-theoretic semantics on a \( \lambda \)-calculus-like subset of the Isabelle/HOL terms of type \texttt{expr}, called the syntactic terms:

\[
s ::= x \mid \texttt{CON} \ a \mid \texttt{VAR} \ n \mid s_1 \ $$ \, s_2 \mid \texttt{LAM} \ x. \ s
\]

---

\(^1\)While \( \texttt{APP} \) and \( \texttt{LAM} \) were inspired by the untyped \( \lambda \)-calculus, in Hybrid they are used only as syntax, without built-in notions of \( \beta \)-conversion, normal forms, etc.
where $s$ (with possible subscripts) stands for a syntactic term, $x$ for a variable of type $\text{expr}$, $a$ for a constant of type $\text{con}$, and $n$ for a natural-number constant. Note that $s$ is an informal mathematically defined set; it is not a formal Isabelle/HOL definition.

However, open terms present a complication. Suppose we have a theory where the semantics is bijective on closed syntactic terms, which it maps to a set $S$. Then it will map open terms with $n$ free variables to functions from the Cartesian power $S^n$ to $S$. But there are many such functions that do not correspond to syntactic terms; for example, the function $S \rightarrow S$ corresponding to the Isabelle/HOL term

$$\lambda x. \text{if } (\exists a. x = \text{CON } a) \text{ then } (x \cdot x) \text{ else } x$$

of type $(\text{expr} \Rightarrow \text{expr})$. Indeed, there are a countable infinity of syntactic terms, while the set of functions from $S^n$ to $S$ is uncountable for $n \geq 1$.

Thus, Hybrid must define a subset of the function space to be used as its representation for open syntactic terms. This is done using a predicate $\text{abstr} :: ((\text{expr} \Rightarrow \text{expr}) \Rightarrow \text{bool})$. The functions satisfying $\text{abstr}$ will be those of the form $(\lambda x.s)$ where $s$ is a syntactic term with (at most) one free variable $x$; we call these the syntactic functions. \(^2\) (Syntactic terms with more than one free variable can be handled one variable at a time.)

In the first-order case, three properties hold of a type defined using Isabelle/HOL’s datatype: distinctness of the datatype constructors, injectivity of each constructor, and an induction principle. In the case of Hybrid, distinctness of all the operators and injectivity of the first-order operators (i.e., all except LAM) are straightforward to achieve, e.g.:

$$\forall (c :: \text{con}) \ (S :: \text{expr} \Rightarrow \text{expr}) \ (\text{CON } c \neq \text{LAM } S)$$

$$\forall (s t s' t' :: \text{expr}) \ (s \cdot t = s' \cdot t' ) \rightarrow (s = s') \land (t = t') .$$

(These properties are used as rewrite rules for Isabelle’s simplifier, to reduce equalities of Hybrid terms with known operators on both sides; typically this results in equalities where one side is just an Isabelle/HOL variable, which can then be eliminated by substitution. \(^3\))

Injectivity of LAM must be restricted to functions satisfying $\text{abstr}$; indeed, it can be proven in Isabelle/HOL that no injective function from $(\text{expr} \Rightarrow \text{expr})$ to $\text{expr}$ exists, by formalizing Cantor’s diagonal argument. As mentioned earlier, our improved version requires an $\text{abstr}$ condition for only one side of the equality:

$$[ \text{abstr } S \lor \text{abstr } T ; \text{LAM } S = \text{LAM } T ] \Rightarrow S = T.$$  

Requiring only a single condition reduces the need for explicit $\text{abstr}$ conditions in object-language encodings, because they can be transported across equalities of LAM terms. It is achieved by adding to the type $\text{expr}$ an additional constant $\text{ERR}$, and defining LAM to take the value $\text{ERR}$ on functions not satisfying $\text{abstr}$. (The constant $\text{ERR}$ will sometimes appear as an additional case alongside the operators of Hybrid, in lemmas that impose an $\text{abstr}$ condition for the LAM case. We also include it among the syntactic terms.)

Since $\text{abstr}$ appears as a premise of injectivity—and it would in any case be needed to state properties of open syntactic terms—we must also include properties sufficient to characterize it. While Hybrid

---

\(^2\)Previous work called such functions *abstractions* \([1]\) – thus the predicate name $\text{abstr}$; and called functions not satisfying $\text{abstr}$ *exotic terms* \([1, 5]\).

\(^3\)Indeed, most use of Hybrid’s lemmas in object-language work is automated using Isabelle’s simplifier and classical reasoner, and as a result, direct references to Hybrid’s lemmas may be rare.
proves a number of lemmas regarding \texttt{abstr} for convenience and proof automation, the desired characterization can be given in a single statement:

\[
\text{abstr } Y \equiv \begin{cases} 
(Y = (\lambda x. x)) \\
(\exists a. Y = (\lambda x. \text{CON } a)) \\
(\exists n. Y = (\lambda x. \text{VAR } n)) \\
(\exists S T. Y = (\lambda x. S \, x $$ T \, x) \land \text{abstr } S \land \text{abstr } T) \\
(\exists W. Y = (\lambda x. \text{LAM } y. W \, x \, y) \land \text{abstr } W) \\
(Y = (\lambda x. \text{ERR}))
\end{cases}
\]

Once again the \texttt{LAM} case complicates matters: the underlined occurrence of \((\text{abstr } W)\) applies \texttt{abstr} to a function \(W :: ([\text{expr}, \text{expr}] \Rightarrow \text{expr})\). This should be possible by using type classes to give a polymorphic definition for \texttt{abstr}, but that is future work. The present version of Hybrid instead replaces \((\text{abstr } W)\) with \((\forall y. \text{abstr } (\lambda x. W \, x \, y)) \land (\forall x. \text{abstr } (\lambda y. W \, x \, y)).\)

As for induction, it can take several forms. First, a kind of size induction on \texttt{expr} is available, similar to size induction for types defined by Isabelle/HOL’s datatype package. This induction has limited applicability in the higher-order setting, although it was used in the proof of adequacy [12]. We also retain an induction principle from the original version of Hybrid [1] where the first-order induction cases are standard, while the \texttt{LAM} case is:

\[
\forall S :: (\text{expr} \Rightarrow \text{expr}). \space \text{abstr } S \land (\forall n. P (S (\text{VAR } n))) \rightarrow P (\lambda x. S \, x).
\]

A common form of induction used in many case studies involves some form of structural induction on the encoding of the inference rules of an OL. For this kind of reasoning, a two-level approach is adopted, similar in spirit to other systems such as \texttt{Twelf} [18] and \texttt{Abella} [10]. An intermediate layer between the meta-logic (Isabelle/HOL) and the OL, called a \textit{specification logic}, is defined inductively in Isabelle/HOL. This middle layer allows succinct and direct encodings of object logic inference rules, which are also defined as inductive definitions. Successful applications of this kind of induction can be found in [8, 12], for example.

Finally, Hybrid aims to build \texttt{expr} and its operators definitonally in Isabelle/HOL. While the description above is an informal but reasonably complete specification of Hybrid, it is not directly usable as a definition because it is circular: the arguments of \texttt{LAM} and \texttt{abstr} may themselves contain \texttt{LAM}, and injectivity of \texttt{LAM} depends on \texttt{abstr}. It could be formalized as an axiomatic theory, leaving consistency as a meta-theoretical problem; but instead, Hybrid is built definitonally in terms of a first-order representation of variable binding based on de Bruijn indices. The definitions and lemmas involved in achieving this are the subject of the next sections.

\section{De Bruijn syntax}

The Hybrid theory defines the type \texttt{expr} in terms of an Isabelle/HOL datatype \texttt{dB}, which represents abstract syntax using a nameless first-order representation of bound variables called \textit{de Bruijn indices} [2].

This approach differs from the original version of Hybrid [1], which used a datatype corresponding to our \texttt{dB} directly as \texttt{expr}; the significance of this difference will be explained in Sections 4 and 6. However, the datatype itself is very similar, and this section follows [1] closely.

\begin{definition}
\begin{itemize}
    \item \texttt{types}
    \begin{itemize}
        \item \texttt{var} = \texttt{nat}
        \item \texttt{bnd} = \texttt{nat}
    \end{itemize}
\end{itemize}
\end{definition}
**4 The type “expr” of proper de Bruijn terms**

Defining a type specifically to represent syntax has been used in a variety of approaches to reasoning about the \(\lambda\)-calculus and other object logics (e.g. [17, 22]). Here, we use Isabelle/HOL’s **typedef** mechanism to define \(\text{expr}\) as a bijective image of the set of proper terms of type \(dB\).\(^4\) That eliminates the proper conditions in object-language work using Hybrid, at the expense of having to convert terms between \(\text{expr}\) and \(dB\) in defining LAM and abstr. This is a good trade-off, because those definitions are internal to Hybrid and need only be made once. It also turns out to be essential for strengthening the quasi-injectivity property of LAM, as described in Sect. 6.

**Definition 2**

\[
\text{typedef (open)} \quad a \text{ expr} = \{ x : a dB. \text{level} 0 x \} \quad \text{morphisms} dB \text{ expr}
\]

This **typedef** statement first demands a proof that the specified set is nonempty (which is trivial here). Then it introduces the type \(\text{expr}\), the functions \(dB : (\text{expr} \Rightarrow dB)\) and \(\text{expr} : (dB \Rightarrow \text{expr})\), and axioms stating that they are inverse bijections between the type \(\text{expr}\) and the set \(\{ x : dB. \text{level} 0 x \}\). (Although axioms are used, the overall mechanism is a form of definitional extension and preserves consistency of the theory.)

\(^4\)The version of \(\text{expr}\) presented here is a modification of the one used in [14].
We may now define all of the first-order operators of Hybrid (i.e., all except \( \text{LAM} \), with its functional-type argument) in the obvious way.

**Definition 3**

\[
\begin{align*}
\text{CON} &:: a \Rightarrow a \text{ expr} & &\text{CON} \ a &\equiv \text{expr} (\text{CON'} \ a) \\
\text{VAR} &:: \text{ var} \Rightarrow a \text{ expr} & &\text{VAR} \ n &\equiv \text{expr} (\text{VAR'} \ n) \\
\text{APP} &:: [a \text{ expr}, a \text{ expr}] \Rightarrow a \text{ expr} & &s \ $$ \ t &\equiv \text{expr} (\text{dB} \ s \ $$ \ t) \\
& & &\quad \quad \quad \quad \text{(notation } (s \ $$ \ t)) \\
\text{ERR} &:: a \text{ expr} & &\text{ERR} &\equiv \text{expr} \ \text{ERR'}
\end{align*}
\]

\( \text{ERR} \) is defined as if it were a separate operator, and it will sometimes be treated as such, but it will also be generated by \( \text{LAM} \) applied to a non-syntactic function.

The functions \( \text{dB} \) and \( \text{expr} \) translate these operators to the corresponding constructors of \( \text{dB} \) (Definition 1) and vice versa. This is formalized by a set of lemmas that follow straightforwardly from the definitions, of which we present just those for \( \text{APP} \ (\$$) as an example.

**Lemma 4**

\[
\text{dB} (s \ $$ \ t) = \text{dB} s \ $$ \ \text{dB} t \\
\text{expr} (s \ $$ \ t) = \text{expr} s \ $$ \ \text{expr} t
\]

Distinctness and injectivity for these operators follow from the corresponding properties of \( \text{dB} \). In Sect. 6, we will extend these results to \( \text{LAM} \) as well.

The \((\text{level} \ 0)\) premises in the lemma above are needed because the \texttt{typedef}-generated function \( \text{expr} \) is undefined on terms with dangling indices. These premises could be eliminated by defining a more tightly-specified version of \( \text{expr} \), satisfying the same \texttt{typedef}-generated axioms while preserving the structure of its argument except for any dangling indices. This was done in the previous version of Hybrid [14] (with the help of an auxiliary function called \texttt{trim}). However, with a more systematic treatment of \text{level} and some additional lemmas for it, this was found to be unnecessary.

All versions of Hybrid follow a general pattern of making definitions and proving lemmas first for arbitrary levels, and then deriving the desired results for proper terms as corollaries. In the present version, arbitrary levels are handled by recursion and induction over de Bruijn syntax, using the type \( \text{dB} \) and the predicate \text{level}, while the results for proper terms are stated at type \text{expr}.

### 5 Definition of “abstr” and “LAM”

We now turn to the task of defining abstr and \( \text{LAM} \). The main ideas are from [1], but the details of the definitions and proofs are original. There are some improvements over the original version of Hybrid, which will be described in this section and Sect. 6.

Since we will be defining abstr and \( \text{LAM} \) in terms of de Bruijn syntax, the definition of syntactic functions from Sect. 2 is not directly usable here: we need an analogous definition using de Bruijn syntax in place of \( \text{LAM} \).

For recursion, we must work with \( \text{dB} \)-valued functions (arbitrary levels) rather than \text{expr}-valued functions. However, the argument type need not also be \( \text{dB} \), and in fact it will be more convenient to work with functions of type \((\text{expr} \Rightarrow \text{dB})\). This simplifies the treatment of \text{level} by avoiding negative occurrences of the type \( \text{dB} \).

Thus we define the **syntactic dB-terms**, as a subset of Isabelle/HOL terms of type \( \text{dB} \), using variables of type \text{expr} converted via \text{dB}:

\[
s ::= \text{dB} x \mid \text{CON'}a \mid \text{VAR'}n \mid s_1 \ $$ s_2 \mid \text{ERR'} \mid \text{BND'}i \mid \text{ABS'}s
\]
where \( s \) (with possible subscripts) stands for a syntactic \( dB \)-term, \( x \) for a variable of type \( expr \), \( a \) for a constant of type \( con \), and \( n \) and \( i \) for natural-number constants. We define the syntactic \( dB \)-functions as the functions of type \(( expr \Rightarrow dB )\) of the form \(( \lambda x . s )\), where \( s \) is a syntactic \( dB \)-term with (at most) one free variable \( x \). Such functions mix de Bruijn indices \( (dB') \) with HOAS (using the Isabelle/HOL bound variable \( x \) to represent an object-language variable).

We define a predicate \( Abstr \) to recognize the syntactic \( dB \)-functions, which formally defines the so-far only informally identified set. We also define an auxiliary predicate \( ordinary \) needed in the definition of \( Abstr \):

**Definition 5**

\[
\text{ordinary :: } (b \Rightarrow a dB) \Rightarrow bool
\]

\[
\text{ordinary } X \equiv \begin{cases} 
\exists a. X = (\lambda x. \text{CON}' a) \lor \exists n. X = (\lambda x. \text{VAR}' n) \lor \\
\exists S T. X = (\lambda x. S x S$'$ T x) \lor (X = (\lambda x. \text{ERR}')) \lor \\
\exists j. X = (\lambda x. \text{BND}' j)) \lor (\exists S. X = (\lambda x. \text{ABS}' (S x))) 
\end{cases}
\]

**Definition 6**

\[
\text{function Abstr :: } (a expr \Rightarrow a dB) \Rightarrow bool
\]

\[
\text{Abstr } (\lambda x. s) = \text{True where } s \text{ is (CON'} a), \text{(VAR'} n), \text{ERR'} , \text{or (BND'} i)
\]

\[
\text{Abstr } (\lambda x. S x S$'$ T x) = (\text{Abstr } S \land \text{Abstr } T)
\]

\[
\text{Abstr } (\lambda x. \text{ABS}' (S x)) = \text{Abstr } S
\]

\[
\neg \text{ordinary } S \implies \text{Abstr } S = (S = dB)
\]

Syntactically, the defining equations for \( Abstr \) have the form of recursion on the \textit{body} of a \( \lambda \)-abstraction. Mathematically, they define \((\text{Abstr } S)\) by recursion on the \textit{common structure} of all the values of the function \( S \), i.e., on the common structure (if any) of \((S x)\) for all \( x :: expr \). The predicate \( ordinary \) recognizes those functions that match one of the first three equations, so that the condition \(\neg \text{ordinary } S\) on the last equation may be read as “otherwise”; that equation corresponds to the variable case for syntactic \( dB \)-terms as defined above.

This definition is formalized with the help of Isabelle/HOL’s \texttt{function} command. It demands proofs of pattern completeness, compatibility, and termination (not shown), and then in addition to defining \( Abstr \) and proving its defining equations, it automatically generates structural induction and case-distinction rules for the type \((expr \Rightarrow dB)\) corresponding to the pattern of recursion used in the definition; these are called \( Abstr \).\texttt{induct} and \( Abstr \).\texttt{cases} respectively, and will be referred to later.

We may now define the predicate \( abstr \) in terms of \( Abstr \) by using post-composition with \( dB \) to convert its function argument from the type \((expr \Rightarrow expr)\) to \((expr \Rightarrow dB)\).

**Definition 7**

\[
\text{abstr :: } (a expr \Rightarrow a expr) \Rightarrow bool
\]

\[
\text{abstr } S \equiv \text{Abstr } (dB \circ S)
\]

Note that unlike the situation in [1], the definition of \( Abstr \) does not need to impose a constraint on the argument of \( dB' \), because in the case of \((abstr S)\) dangling indices are excluded by the type of the function \( S :: (expr \Rightarrow expr)\).

**Lemma 8**

\[
\text{Abstr_const :: } Abstr \ (\lambda x. s)
\]

The lemma \( Abstr \_const \) shows that any constant function of type \((expr \Rightarrow dB)\) satisfies \( Abstr \). It is used to prove a similar property for \( abstr \), and will later be used directly as well. It is proved by induction on \( s \) using Definition 6 (\( Abstr \)).
**Lemma 9**

\[
\begin{align*}
\text{abstr\_id: } & \text{ abstr } (\lambda \ x. \ x) \\
\text{abstr\_const: } & \text{ abstr } (\lambda \ x. \ s) \\
\text{abstr\_APP: } & \text{ abstr } (\lambda \ x. \ S \ x \ T \ x) = (\text{abstr } S \land \text{abstr } T)
\end{align*}
\]

The lemma abstr\_const is a corollary of Abstr\_const, while the other two lemmas are proved directly, using Definitions 7 (abstr) and 6 (Abstr).

These lemmas allow abstr conditions for syntactic functions to be proved compositionally without unfolding the definition, except when the body of the function contains a LAM subterm that involves the function argument (so that it is not just a constant). In that case, previous versions of Hybrid required unfolding the definitions of abstr and LAM to convert HOAS to de Bruijn syntax. The present work improves on that situation by providing a compositional rule also for the LAM case (Lemma 20 in Sect. 7).

The lemma abstr\_const will be important for Hybrid terms with nested LAM operators, to show that the argument of an inner LAM satisfies abstr when its body contains a bound variable from an outer LAM; such a bound variable is a placeholder for an arbitrary term of type expr, which is exactly the role of s in abstr\_const.

We now define the function LAM, using the same form of recursion that was used in the definition of abstr.

**Definition 10**

\[
\begin{align*}
\text{LAM } :: & \ (a \ expr \Rightarrow a \ expr) \Rightarrow a \ expr \\
\text{LAM } S & \equiv \ expr \ (\text{Lambda } (dB \circ S)) \\
\text{Lambda } :: & \ (a \ expr \Rightarrow a \ dB) \Rightarrow a \ dB \\
\text{Lambda } S & \equiv \ \text{if } (\text{Abstr } S) \ \text{then } (\text{ABS'} (\text{Lbind } 0 S)) \ \text{else } \text{ERR'}
\end{align*}
\]

The function LAM, like abstr, first composes dB with the given function. It then applies the auxiliary function Lambda and converts the resulting term from type dB to type expr.

The function Lambda first checks if its argument satisfies Abstr, and produces ERR' if not. (This is equivalent to checking if the argument of LAM satisfies abstr.) The original version of Hybrid [1] did not do this check (and did not have the constant ERR'), making it impossible to determine from (LAM S) whether S was a syntactic function or not. We include these features to support the stronger injectivity property for LAM proved in Sect. 6.

If its argument does satisfy Abstr, then Lambda applies another auxiliary function Lbind, defined by recursion, to convert HOAS to de Bruijn syntax; i.e., to convert the variable represented by the function argument into a dangling de Bruijn index. It then applies a new ABS' node to bind the variable and obtain a proper de Bruijn term.

**Definition 11**

\[
\begin{align*}
\text{function } \text{Lbind } :: & \ [bnd, \ (a \ expr \Rightarrow a \ dB)] \Rightarrow a \ dB \\
\text{Lbind } i (\lambda \ x. \ s) & = s \ \text{where } s \ \text{is } (\text{CON'} a), \ (\text{VAR'} n), \ \text{ERR'}, \ \text{or } (\text{BND'} j) \\
\text{Lbind } i (\lambda \ x. \ S \ x \ T \ x) & = \text{Lbind } i S \ S' \ \text{Lbind } i T \\
\text{Lbind } i (\lambda \ x. \ \text{ABS'} (S \ x)) & = \text{ABS'} (\text{Lbind } (i + 1) S) \\
\neg \ \text{ordinary } S & \Rightarrow \ \text{Lbind } i S = \text{BND'} i
\end{align*}
\]

The auxiliary function Lbind extracts the common structure of the values of its function argument, replacing indecomposable uses of the bound variable (i.e., functions that do not match any of the first three equations) with (BND' i). This is a dangling de Bruijn index, and i is incremented each time the
recursion passes an ABS′ node so that all such instances of BND′ will refer to the ABS′ node added by Lambda. The Abstr condition checked in the definition of Lambda ensures that the last equation will be applied only when S = (λ x. dB x).

**Lemma 12**

\[
\text{Lbind\_const: Lbind}\ i\ (\lambda \ x. s) = s
\]

The lemma Lbind\_const shows that applying (Lbind i) to a constant function of type \((expr \Rightarrow dB)\) gives the constant value of that function. It is proved by induction on s. This lemma will be important for Hybrid terms with nested LAM operators, to allow the argument of an outer LAM to satisfy abstr when its bound variable occurs in the scope of an inner LAM.

**Lemma 13**

\[
\text{dB\_LAM: dB}\ (\text{LAM}\ S) = \text{if}\ (\text{abstr}\ S)\ \text{then}\ (\text{ABS}'\ (\text{Lbind}\ 0\ (\text{dB} \circ S)))\ \text{else}\ \text{ERR}'
\]

\[
\text{abstr\_dB\_LAM: abstr}\ S \Rightarrow dB\ (\text{LAM}\ S) = \text{ABS}'\ (\text{Lbind}\ 0\ (\text{dB} \circ S))
\]

The lemma dB\_LAM combines unfolding of Definition 10 (LAM and Lambda) with cancellation of the functions dB and expr, using the fact that both ERR′ and \((\text{ABS}'\ (\text{Lbind}\ 0\ (\text{dB} \circ S)))\) are proper. (Dangling indices are excluded from \(S :: (expr \Rightarrow expr)\) by its type, and the one introduced by Lbind is bound by the enclosing ABS′.) The lemma abstr\_dB\_LAM is a weaker version intended as a conditional rewrite rule for Isabelle’s simplifier, to do the unfolding only if the abstr condition simplifies to True.

With the definitions above, Hybrid terms using LAM (i.e., closed syntactic terms) are provably equal to the corresponding de Bruijn syntax representations, converted to the type \(expr\) using the function expr. (This is much the same situation as in [1], except for the type conversion which was not necessary there.) Thus, starting from two distinct representations for free variables, we have established two ambiguous representations for bound variables, in the sense that any given element of expr may be viewed as having either form. In the following sections, we will state results using the HOAS representation (LAM) but use the de Bruijn syntax representation (ABS'/BND') in proofs by induction, aiming to characterize the former representation so that it stands on its own.

All versions of Hybrid have used essentially the same form of recursion to define abstr and LAM, and the corresponding form of induction to prove their properties. However, the means of formalizing it have varied greatly. The original version [1] used inductively-defined predicates and induction on those predicates; the following version [14] used primitive recursion and induction on an auxiliary datatype dB\_fn; while the present version avoids many of the complications of the previous approaches with the help of the function command.

A predicate called ordinary has also been present in all versions of Hybrid, though it originally included the variable case as well. Removing this case allowed ordinary to be generalized to dB-valued functions on any type; this will allow us to reuse it for binary functions in Sect. 7. (It is also reused for n-ary functions in [12, Sect. 3.3].)

6 Injectivity of “LAM”

As stated in Sect. 2, Hybrid proves injectivity of LAM restricted to functions of type \((expr \Rightarrow expr)\) satisfying abstr. Improving on [1], this property is strengthened by requiring only one abstr premise, using the fact that LAM maps functions not satisfying abstr to a recognizable placeholder term ERR.

We begin with an injectivity result for arbitrary de Bruijn levels. To state this result concisely, we first define an abbreviation Level for pointwise application of level to a function:
Definition 14
abbreviation Level :: [bnd, (b ⇒ a dB)] ⇒ bool

Level i S ≡ ∀ x. level i (S x)

Lemma 15
Abstr_Lbind_inject:
[Abstr S; Abstr T; Level i S; Level i T] ⇒ (Lbind i S = Lbind i T) = (S = T)

This lemma is proved by a straightforward induction on S :: (expr ⇒ dB) using Abstr.induct (from Definition 6).

Theorem 16 (Injectivity of LAM)
[LAM S = LAM T; abstr S ∨ abstr T] ⇒ S = T

Proof. If one of S and T satisfies abstr and the other does not, then by Lemma 13 (dB_LAM), one of the terms (dB (LAM S)) and (dB (LAM T)) is of the form (ABS' t) for some t :: dB, while the other is ERR'. But these terms cannot be equal, which contradicts the premise LAM S = LAM T. Thus the original assumption must be false, and we must have both (abstr S) and (abstr T).

We apply dB to both sides of the equality LAM S = LAM T and simplify using abstr dB_LAM (Lemma 13) to obtain

ABS' (Lbind 0 (dB ◦ S)) = ABS' (Lbind 0 (dB ◦ T)).

ABS' is a datatype constructor and thus injective, so we may cancel it:

Lbind 0 (dB ◦ S) = Lbind 0 (dB ◦ T).

We have (Abstr (dB ◦ S)) and (Abstr (dB ◦ T)) by unfolding Definition 7 (abstr), and we also have (Level 0 (dB ◦ S)) and (Level 0 (dB ◦ T)) since terms converted from type expr are proper by Definition 2. Thus we may apply the preceding lemma (Abstr_Lbind_inject) to deduce dB ◦ S = dB ◦ T. Since dB is injective, it can be canceled to obtain S = T, as was to be proven.

Note that (Lbind 0) is only injective on functions from expr to dB whose values are proper terms, i.e., those that factor through dB, because any pre-existing dangling indices at level 1 would be indistinguishable from those resulting from conversion of the HOAS variable. For example,

Lbind 0 (λ x. dB x) = BND' 0 = Lbind 0 (λ x. BND' 0).

Thus, without the typedef limiting expr to proper terms, we would not be able to avoid conditions on both S and T; at best, we could replace one abstr condition with something like (∀ x. proper x −→ proper (T x)).

The advantage of an injectivity property that can work with a condition on only one of S and T is that it simplifies the elimination rules for inductively-defined predicates on Hybrid terms, such as the formalization of evaluation for Mini-ML with references in [12, Sect. 5.3]. As a result, abstr conditions are more often available where they are needed, without having to add them as premises.

Distinctness of LAM from the first-order operators of Definition 3 follows straightforwardly from Definition 10, except that (LAM F) is distinct from ERR only under the premise (abstr F).
7 Characterizing “abstr”

In Sect. 5, an incomplete set of simplification rules for abstr was provided as Lemma 9. The missing case is \((\lambda x. \text{LAM} y. W x y))\).

Both previous versions of Hybrid [1, 14] relied on conversion from HOAS to de Bruijn syntax to handle this case. That is sufficient for proving that particular syntactic functions satisfy abstr, but it is less useful for partially-specified functions as found in inductive proofs.

We could obtain a compositional introduction rule for this case by defining a predicate \(\text{biAbstr}::\) \([(\text{expr}, \text{expr}) \Rightarrow \text{expr}) \Rightarrow \text{bool} \) generalizing abstr, and proving

\[
\text{biAbstr} W \equiv abstr (\lambda x. \text{LAM} y. W x y).
\]

This was done by Momigliano et al. [13]; their formal theory BiAbstr is available online [6]. However, the \(\text{LAM}\) case arises again for \(\text{biAbstr}\), and for any higher-arity generalization. There are several ways to address this:

- Use Isabelle/HOL’s axiomatic type classes to define a polymorphic predicate generalizing abstr to curried functions of arbitrary arity. This looks like a promising approach, but it remains as future work.
- Find a single type that can represent functions of arbitrary arity, and generalize Hybrid’s constructs to that type. (Some experimental work has been done in that direction [12, Sect. 3.3].) Such a type is also useful as a representation of open terms for induction.
- Prove a result that reduces \(\text{biAbstr}\) to abstr. This seems to be the most direct solution, and it is the approach we take in the present work.

In this section, we will represent functions of two arguments using pairs, rather than in the usual curried form, so that we may reuse Definition 5 (ordinary) and some technical lemmas (left unstated as they are mathematically trivial), all of which refer to the polymorphic type \((b \Rightarrow dB)\).

**Definition 17**

\[
\text{abstr}_2 :: (a \text{ expr} \times a \text{ expr} \Rightarrow a \text{ expr}) \Rightarrow \text{bool}
\]

\[
\text{abstr}_2 S \equiv \text{Abstr}_2 (dB \circ S)
\]

The predicate \(\text{abstr}_2\) generalizes abstr to functions on the Cartesian product type \((\text{expr} \times \text{expr})\); it corresponds to \(\text{biAbstr}\) [13]. It is defined in the same way as abstr, composing \(dB\) with its argument and then applying a recursively-defined auxiliary predicate \(\text{Abstr}_2\).

**Definition 18**

\[
\text{function Abstr}_2 :: (a \text{ expr} \times a \text{ expr} \Rightarrow a dB) \Rightarrow \text{bool}
\]

\[
\text{Abstr}_2 (\lambda p. s) = \text{True where s is (CON’ a), (VAR’ n), ERR’, or (BND’ i)}
\]

\[
\text{Abstr}_2 (\lambda p. S p \Downarrow T p) = (\text{Abstr}_2 S \land \text{Abstr}_2 T)
\]

\[
\text{Abstr}_2 (\lambda p. \text{ABS’} (S p)) = \text{Abstr}_2 S
\]

\[
\neg \text{ordinary S} \equiv\text{Abstr}_2 S = (S = dB \circ \text{fst} \lor S = dB \circ \text{snd})
\]

The predicate \(\text{Abstr}_2\) is similar to \(\text{Abstr}\), except that it has two variable cases: \((dB \circ \text{fst})\) and \((dB \circ \text{snd})\), or equivalently, \((\lambda (x, y). dB x)\) and \((\lambda (x, y). dB y)\).

**Lemma 19**

\[
\text{abstr}_2 S = ((\forall y. \text{abstr} (\lambda x. S (x, y))) \land (\forall x. \text{abstr} (\lambda y. S (x, y))))
\]
This lemma shows that if a two-argument function satisfies abstr in each argument for any fixed value of the other argument, then it satisfies abstr_2. (And the converse, which is easier.) We omit the formal proof, but note that it is fairly long and requires several lemmas.

Having thus reduced abstr_2 to componentwise abstr, we may now derive the desired simplification rule for the case (abstr (\lambda x. LAM y. W x y)).

Lemma 20
\textit{abstr\_LAM}: \forall x. \ \text{abstr} (\lambda y. W x y) \implies \text{abstr} (\lambda x. LAM y. W x y) = (\forall y. \ \text{abstr} (\lambda x. W x y))

This lemma provides a compositional rule for proving abstr conditions on functions of the form (\lambda x. LAM y. x $$ y), via the reverse direction of the biconditional. Both directions are also used in the proof of adequacy. It was proved with the help of (a variant of) Lemma 19.

We consider a small example, the term (LAM x. LAM y. x $$ y), illustrating abstr\_LAM by proving that the argument of the outer LAM satisfies abstr, without the use of de Bruijn syntax:

\begin{align*}
\forall x. \ \text{abstr} (\lambda y. x) & \quad \text{(by abstr\_const)} \\
\text{abstr} (\lambda y. y) & \quad \text{(by abstr\_id)} \\
\forall x. \ \text{abstr} (\lambda y. (x $$ y)) & \quad \text{(by abstr\_APP)} \\
\text{abstr} (\lambda x. x) & \quad \text{(by abstr\_id)} \\
\forall y. \ \text{abstr} (\lambda x. y) & \quad \text{(by abstr\_const)} \\
\forall y. \ \text{abstr} (\lambda x. (x $$ y)) & \quad \text{(by abstr\_APP)} \\
\text{abstr} (\lambda x. LAM y. (x $$ y)) & \quad \text{(by abstr\_LAM)}
\end{align*}

Not only does the lemma abstr\_LAM allow abstr statements to be proved without the use of de Bruijn syntax, but it also completes the task of characterizing expr on its own terms – that is, without reference to the underlying de Bruijn syntax. This is demonstrated in [12] by the fact that representational adequacy follows from Hybrid’s lemmas concerning the type expr, and it is a significant improvement over both previous versions of Hybrid [1, 14].

We also obtain the characterization of abstr stated in Sect. 2 as a corollary of abstr\_LAM:

Lemma 21
\textit{abstr} Y = \left( (Y = (\lambda x. x)) \lor \\
(\exists a. Y = (\lambda x. CON a)) \lor (\exists n. Y = (\lambda x. VAR n)) \lor \\
(\exists S. \ \text{abstr} S \land \text{abstr} T \land Y = (\lambda x. S x $$ T x)) \lor \\
(\exists W. (\forall x. \ \text{abstr} (\lambda y. W x y)) \land (\forall y. \ \text{abstr} (\lambda x. W x y)) \land \\
Y = (\lambda x. LAM y. W x y)) \lor (Y = (\lambda x. ERR)) \right)

8 Conclusion

Hybrid is the first approach to formalizing variable-binding constructs that is both based on full HOAS and is built definitionally in a general-purpose proof assistant (Isabelle/HOL). More recently, Popescu et. al. have developed an approach motivated by a new proof of strong normalization for System F that takes advantage of HOAS techniques [21]. It is also definitionial, implements full HOAS, and is implemented in Isabelle/HOL, though the details of the formalizations as well as the case studies carried out in each system are quite different. A more in-depth comparison is the subject of future work.

There are many other related approaches, and we mention only a few here. See [8, 12] for a fuller discussion. Systems that implement logics designed specifically for reasoning using HOAS include
Twelf [18] (one of the most mature systems in this category), Abella [10], and Beluga [19]. These systems have the advantage of being purpose-built for reasoning about formal systems, but this can also be a disadvantage in that they cannot exploit the extensive libraries of formalized mathematics available for proof assistants such as Isabelle/HOL. For a comparison of Hybrid to Twelf and Beluga, see [7]. The nominal datatype package [22] implements a different approach which seeks to formalize equivalence of classes of terms up to renaming of bound variables, and also the Barendregt variable convention, using concepts from nominal logic [9, 20].

There are several versions of Hybrid based on the Coq proof assistant. One such version [8] closely follows the structure of the Isabelle/HOL version; another implements a constructive variant of Hybrid for Coq [3] that aims to leverage the use of dependent types to simplify and provide new ways to specify OLs. There have also been a number of applications and case studies for Hybrid, the largest being the comparison of five formalizations of subject reduction for Mini-ML with references [12], which uses the improved Hybrid described in this paper. Future work includes porting other applications to use the new Hybrid. This will be straightforward since they are simpler and will be further simplified by the new interface. Future work also includes carrying out new case studies to further illustrate the benefits of the new Hybrid.

Although we have significantly improved Hybrid, there is always room for further improvement. For example, the induction principle discussed at the end of Sect. 2 (the one whose LAM case is displayed) falls back to named (or numbered) variables for inductive proofs, which means giving up some of the advantages of HOAS. We are working on a more general approach to induction that preserves the HOAS feature of substitution by function application. In fact, we have proved an induction principle for a type that represents n-ary functions on the type expr [12], which we hope will serve as the basis for general induction principles for HOAS in Hybrid. Its integration into Hybrid remains as future work. As another example, we mentioned that Hybrid is untyped, requiring predicates to be introduced to distinguish different kinds of OL terms encoded into expr. On one hand, these well-formedness predicates can provide a convenient form of induction within the context of the two-level approach; on the other hand this is a potential area for improvement. Some work in this direction has been done in the Coq version of Hybrid [4].

References


