Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 323 (2016) 125–142

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

# Canonical HybridLF: Extending Hybrid with Dependent Types

Roy L. Crole [1] Amy Furniss [2]

*Dept of Computer Science, University of Leicester, University Road, Leicester, LE1 7RH, U.K.*

**Abstract**

We introduce Canonical HybridLF (CHLF), a metalogic for proving properties of deductive systems, implemented in Isabelle HOL. CHLF is closely related to two other metalogics. The first is the Edinburgh Logical Framework (LF) by Harper, Honsell and Plotkin. The second is the Hybrid system developed by Ambler, Crole and Momigliano which provides a Higher-Order Abstract Syntax (HOAS) based on un-typed lambda calculus.
Historically there are two problems with HOAS: its incompatibility with inductive types and the presence of exotic terms. Hybrid provides a partial solution to these problems whereby HOAS functions that include bound variables in the metalogic are automatically converted to a machine-friendly de Bruijn representation hidden from the user.
The key innovation of CHLF is the replacement of the un-typed lambda calculus with a dependently-typed lambda calculus in the style of LF. CHLF allows signatures containing constants representing the judgements and syntax of an object logic, together with proofs of metatheorems about its judgements, to be entered using a HOAS interface. Proofs that metatheorems defined in the signature are valid are created using the M2 metalogic of Schurmann and Pfenning.
We make a number of advances over existing versions of Hybrid: we now have the utility of dependent types; the unitary bound variable capability of Hybrid is now potentially finitary; a type system performs the role of Hybrid well-formedness predicates; and the old method of indicating errors using special elements of core datatypes is replaced with a more streamlined one that uses the Isabelle option type.

*Keywords:* dependent types, HOAS, logical frameworks, metalogical reasoning, variable binding

## 1 Introduction

This paper is about reasoning about deductive systems such as logics, programming languages and so on. In general we refer to a typical deductive system as an *object logic*. One may reason about an object logic by translating it into a *metalogic* and then performing reasoning in the metalogic, provided that properties of the object logic are suitably reflected in the metalogic.

In particular this paper is concerned with Higher Order Abstract Syntax. This is a well-known metalogical technique that can be applied to object logics that have

[1] Email: rlc3@le.ac.uk
[2] Email: mjf29@le.ac.uk

http://dx.doi.org/10.1016/j.entcs.2016.06.009

variable binding: variable binders of the HOAS metalogic are used to implement the variable binders of an object logic.

The first author, Crole, along with Ambler and Momigliano, developed the Hybrid system [1]. This is an implementation of Higher Order Abstract Syntax within an Isabelle HOL package. The key novel feature is that a user may write down *higher order* abstract syntax using "user friendly" named bound variables and then the package *converts* such syntax to a "machine friendly" *first order* de Bruijn notation for its "internal reasoning". The use of HOAS is not without its problems. One issue is that HOAS is typically not compatible with induction. However Hybrid provides a partial solution and provides a user with a form of HOAS which embeds consistently within (Isabelle) HOL and hence with principles of induction [1].

The Hybrid system is underpinned by the untyped $\lambda$-calculus. While Hybrid has been successfully used by the authors and other researchers (see for example [12,9]) we thought it interesting to consider whether one could develop a typed version. This is not simply intellectual curiosity. When using Hybrid one typically has to implement well-formedness predicates which are deployed within inductive definitions. However within a typed system these predicates might be rendered redundant, instead using types to build well-formed expressions. This is both interesting and a potential important simplification and advantage for the user.

At the heart of Hybrid are *conversion* functions mapping untyped higher order syntax to first order syntax. *Our contribution is to show that the conversion technique extends not just to a simply typed setting, but in fact to a new system,* Canonical HybridLF*, that is dependently typed. We can indeed dispense with well-formedness predicates. Further, the new system provides for Twelf-style reasoning with the judgements-as-types methodology.*

First we developed an extension of Hybrid based on the simply typed $\lambda$-calculus. Having developed suitable conversion functions which were faithful to the conceptual ideas underpinning Hybrid, we began to consider dependent types. We decided to explore using Edinburgh LF [10] as a basis since it is a system for meta-reasoning and so too is Hybrid. However, whereas Hybrid (and the simply typed version) provide a $\lambda$-calculus HOAS interface for encoding an object system, after which a user reasons directly within (Isabelle) higher order logic, basing a version of Hybrid on LF could mean the possibility of reasoning using judgements-as-types *within a general purpose tactical theorem prover.*

In fact Canonical LF underpins Canonical HybridLF (see Section 3.1 for further details). Both LF and Canonical LF have pros and cons. Key factors behind the choice of Canonical LF rather than LF as the basis for our system are the simplicity of equality in Canonical LF and technical issues regarding termination of the unification algorithm that we employ for LF. This, along with type decidability, will be discussed in future work, but here we concentrate on Canonical HybridLF.

There is more about the theory of Hybrid in [5]. A version of Hybrid was created by Martin [12] and Amy Felty. Versions of Hybrid developed in Coq appear in [3] and [9] with work by Capretta, Felty and Habli. Our methodology of reduction of higher to first order syntax is exploited in [15].

In section 2 we overview the HYBRID system. We point the reader to the original papers, and provide some necessary background reading. In section 3 we briefly recall the details of the Canonical LF logical framework, and then describe the main contributions of this paper, as set out in the Introduction, in detail. In section 4 we give some examples including a *direct comparison* of HYBRID and CANONICAL HYBRIDLF encodings of quantified propositional logic, and discuss an implementation of a logic for proofs in our system.

## 2 The Hybrid Metalogic

In HYBRID terms of any object logic are entered by the user as HOAS metalogic terms with *named* bound variables. Such terms are then automatically converted to a *nameless de Bruijn* form in which instances of bound variables are given by a numerical *index* or *level*. *The idea is that the user can work with named terms while the machine works with equivalent nameless terms that are automatically generated. The system is a "hybrid" of named and nameless variable binding.* (HYBRID utilises *locally nameless* de Bruijn terms [2]: Bound variables are denoted by instances of `BND`, and have a natural number index; free variables are indicated by instances of `VAR`, and are also indexed by the natural numbers.)

The original version of HYBRID is implemented as a package for the Isabelle theorem prover. The implementation is based around a core inductive datatype `expr` that implements locally nameless de Bruijn terms:

**Definition 2.1** [Core HYBRID Datatype]

$$\texttt{'a expr} \ ::= \ \texttt{BND nat} \mid \texttt{VAR nat} \mid \texttt{CON 'a} \mid \texttt{ABS expr} \mid \underbrace{\texttt{APP expr expr}}_{\texttt{expr \$\$ expr}} \mid \texttt{ERR}$$

The `CON` constructor is used to denote an instance of an object logic constant. The `expr` type has a type parameter `'a`, and the elements of this type are used to specify the constants of an object logic. The object constructors $\forall, \exists$ would be rendered in HYBRID as `CON` $'a$ (where, for example, $'\texttt{a} = \mathsf{cForAll} \mid \mathsf{cExists}$ specifies object logic quantifiers). The `APP` constructor denotes application (often written as infix \$\$) and the `ABS` constructor denotes (de Bruijn) function abstraction. Finally the `ERR` constructor is a special element used to indicate if an error occurs during conversion from HOAS function to de Bruijn indices.

In HYBRID, a general HOAS term $\lambda v.e$ is written by the user as $\mathsf{LAM}\ v.e$; this is legitimate Isabelle syntax. Thus $\forall\ (\lambda v.\phi)$ would be written by the HYBRID user as

$$(\texttt{CON}\ \mathsf{cforAll})\ \$\$\ (\mathsf{LAM}\ v.\phi)$$

The point is that HYBRID provides a very natural user "interface" for HOAS with named bound variables.

To explain further, $\mathsf{LAM}\ v.e$ is an abbreviation for $\mathsf{lambda}(\lambda v.e)$ where $\mathsf{lambda}$ is an Isabelle function that *automatically converts $\lambda v.e$ to a de Bruijn term*, and

$\lambda$ denotes Isabelle function abstraction. lambda lies at the heart of the HYBRID system. Full details can be found in [1,5]. Here we cannot explain in detail the intuition behind the implementation of lambda, but give a brief overview and the Isabelle definitions. lambda($\lambda v.e$) is defined to be ABS (lbind 0 ($\lambda v.e$)).

The main things readers need to know [3] are

**(i)** The abstr (c.f. section 3.3) function tests whether $\lambda v.e$ is a (unary) HYBRID abstraction (which is a prerequisite for conversion).

**(ii)** The lbind (c.f. section 3.4) function performs the conversion of $\lambda v.e$ to de Bruijn form.

To explain the notion of abstraction we first recall the adjectives *dangling* and *level* (which are well-known properties of de Bruijn terms). A variable BND $j$ in a term $e$ is said to be *dangling* if $j$ or less ABS nodes occur on the path from $j$ to the root of $e$. For instance, in the example term $\mathbf{T}$ = ABS (BND 0 $$ BND 1) the bound variable instance BND 0 is not dangling because there are zero such ABS nodes, but BND 1 is indeed dangling. Terms in HYBRID have a *level*. An arbitrary term $e$ is at level $l \geq 1$ if enclosing $e$ in $l$ ABS nodes ensures that the resulting expression has no dangling variables. For instance, the term $\mathbf{T}$ would be at level 1 because it requires one extra enclosing ABS to ensure that no variables are dangling. Terms at level 0–with no dangling indices–are called *proper* terms. Now we define *abstraction*. A *unary* abstraction is, informally, an expression at level *1* in which any dangling index is replaced by a metavariable (say $v$) and then abstracted (enclosed by $\lambda v$). An example is $\mathbf{A} = \lambda v.$ ABS (BND 0 $$ $v$). *This is a key idea. A direct correspondence between de Bruijn indices, such as* BND 1 *in* $\mathbf{T}$*, and the binding mechanism of the metalogic, such as* $\lambda v. \ldots v$ *in* $\mathbf{A}$*, is set up.* In the original HYBRID [1] there is a predicate abstr that defines when its argument is a valid unary abstraction. It is implemented using an inductive relation abst. The definition is omitted here, but background details appear in [1]. The formal definition of abstr is then abstr $e \equiv$ abst 0 $e$.

Briefly, abstr works by recursion on $e$ over the expr constructors, removing the constructor at each call, and moving each $\lambda v$ towards the leaf nodes of $e$. The index $i$ increases when recursing over each ABS node, hence counting the nodes. At a leaf, $\lambda v.\ v$ is deemed an abstraction as is $\lambda v.$ VAR $n$. In the case of a leaf $\lambda v.$ BND $j$, $i$ is now equal to the number of ABS nodes on the path from the leaf to the root, and so BND $j$ is NOT dangling provided that $j < i$ in which case $\lambda v.$ BND $j$ is an abstraction. Overall abstr returns the conjunction of the leaf node results.

lbind is defined using an inductive relation lbnd. The definition is omitted here, but background details appear in [1]. lbind is defined as lbind $i\ e \equiv \epsilon\ s.$ lbnd $i\ e\ s$. lbind works by recursion on $e$ over the expr constructors, leaving each constructor in place, and moving each $\lambda v$ towards the leaf nodes of $e$. The index $i$ increases when recursing over each ABS node, hence counting them. At a leaf, $\lambda v.\ v$ is replaced by

[3] The section references refer to descriptions of the analogues of abstr and lbind for CANONICAL HYBRIDLF.

$$\frac{a{:}K \in \Sigma}{\Gamma \vdash_\Sigma \ a : K}\ \text{CON\_AT\_KIND} \qquad \frac{\Gamma, x{:}A \vdash_\Sigma \ M : A'}{\Gamma \vdash_\Sigma \ \lambda x.M : \Pi x{:}A.A'}\ \text{ABS\_CAN\_TY}$$

$$\frac{\Gamma \vdash_\Sigma \ P : \Pi x{:}A.K \qquad \Gamma \vdash_\Sigma \ M : A \qquad [M/x]^{\mathsf{k}}_{\alpha} K = K'}{\Gamma \vdash_\Sigma \ P\ M : K'}\ \text{APP\_AT\_KIND}$$

$$\frac{x{:}A \in \Gamma}{\Gamma \vdash_\Sigma \ x : A}\ \text{VAR\_AT\_TY} \qquad \frac{c{:}A \in \Sigma}{\Gamma \vdash_\Sigma \ c : A}\ \text{CON\_AT\_TY}$$

$$\frac{\Gamma \vdash_\Sigma \ R : \Pi x{:}A.A' \qquad \Gamma \vdash_\Sigma \ M : A \qquad [M/x]^{\mathsf{a}}_{\alpha} A' = A''}{\Gamma \vdash_\Sigma \ R\ M : A''}\ \text{APP\_AT\_TY}$$

Fig. 1. Canonical LF Judgements, with signature $\Sigma$ and context $\Gamma$

`BND` $i$, the correct de Bruijn index. All other leaf nodes remain unaltered.

# 3 Canonical HybridLF

## *3.1 Recalling Canonical LF*

LF has a notion of *canonical form*, which in [10] consists of kinds, types and terms that are $\beta$-normal, $\eta$-long, and correspond to terms of an object logic. It also has notions of *definitional equality*. All expressions in LF have a canonical form. Watkins et al [20] give a *canonical* presentation of LF in which only kinds, terms and types in canonical form can be formed due to restrictions on the grammar, reducing definitional equality to syntactic equality. This ensures that only LF terms that actually represent terms of the object logic can exist, and eliminates the need to reason about definitional equality. The grammar of canonical LF is as follows:

**Definition 3.1**

$$K ::=_{\mathsf{k}} \text{Type} \mid \Pi x{:}A.K \qquad A ::=_{\mathsf{a}} \ P \mid \Pi x{:}A.A \qquad P ::=_{\mathsf{p}} \ a \mid P\ M$$

$$M ::=_{\mathsf{m}} \ R \mid \lambda x.M \qquad R ::=_{\mathsf{r}} \ x \mid c \mid R\ M$$

Here we use $K$ to indicate an arbitrary kind, $A$ to indicate a canonical type, $P$ to denote an atomic type, $M$ to denote a canonical term and $R$ to indicate an atomic term. Signatures $\Sigma$ consist of either the empty signature $\langle\rangle$, or a list of term constants $c$ and type constants $a$ with their types or kinds. There is no need to define definitional equality relations as Harper et al [10] do in the original LF paper, since definitional equality in canonical LF is syntactic equality. The typing rules for canonical LF are given in figure 1. The main disadvantage of this "canonical"

approach is that it requires some care when performing substitution. However Watkins et al define a suitable notion of *hereditary* substitution and in fact we follow the definition given by Harper and Licata [11]. In our paper we discuss this no further, apart from commenting that we use $[M/x]^{\iota}_{\alpha}\xi$ to denote the hereditary substitution where $\alpha$ is the simple type of $M$ defined by type-erasure (see [11]) and $\iota$ is syntactic category from Definition 3.1.

### *3.2 The Core Datatype*

In essence, the original HYBRID is a system that provides a HOAS interface to an untyped $\lambda$-calculus. As explained in Section 2, the user's methodology is to encode an object logic using "$\lambda$-calculus HOAS terms". Such terms are then automatically converted to de Bruijn terms and reasoned about directly in Isabelle, using Isabelle's higher-order logic and the Isabelle `lemma` construct to create proofs. The users's methodology in CANONICAL HYBRIDLF is slightly different. CANONICAL HYBRIDLF is a system that, *using an adaptation of the* HYBRID *approach to variable binding* provides a *HOAS interface to* a dependently-typed $\lambda$-calculus in the style of *Canonical LF*. Theorems and meta-theorems are

- defined in a signature, where the HOAS interface enables the user to enter the LF signature as Isabelle functions with named bound variables (which are then converted to de Bruijn form), and
- proved correct using the proof rules of a logic $M_2$ [16] described in Section 4.2.

Since CANONICAL HYBRIDLF implements canonical LF it permits the use of the *judgements-as-types approach to proving theorems* [10]. A proof that any particular judgement holds is specified be giving an LF term that inhabits the type that represents the judgement. CANONICAL HYBRIDLF is *consistent with tactical theorem proving and principles of (co)induction* in the same way that HYBRID is. While not discussed explicitly in this paper we also have derived *explicit induction principles*.

Recall the *five* syntactic categories of Definition 3.1. CANONICAL HYBRIDLF is based around *five* mutually-inductively defined datatypes `kind`, `ctype`, `atype`, `cterm` and `aterm`. These mutually defined datatypes yield an overall core datatype inhabited by terms corresponding to those of Definition 3.1, much as `expr` is the core datatype at the heart of HYBRID. The new core datatype is built out of `VAR` and `BND` constructors so that, ultimately, within the machine implementation variable binding is once again boiled down to de Bruijn nameless terms. However a user can still work with a HOAS-style interface with named bound variables.

**Definition 3.2** [Core Canonical HybridLF Datatype]

```
datatype ('a, 'b) kind = TYPE | KPI "('a, 'b) ctype" "('a, 'b) kind"
    and ('a, 'b) ctype = ATYPE "('a, 'b) atype"
                       | PI "('a, 'b) ctype" "('a, 'b) ctype"
    and ('a, 'b) atype = FCON  'b
                       | FAPP "('a, 'b) atype" "('a, 'b) cterm"
    and ('a, 'b) cterm = ATERM "('a, 'b) aterm"
                       | ABS  "('a, 'b) ctype" "('a, 'b) cterm"
    and ('a, 'b) aterm = VAR  nat | BND nat | CON 'a
                       | APP "('a, 'b) aterm" "('a, 'b) cterm"
```

Notice that there are two type parameters 'a, 'b: one parameter is used to specify the object constants and one to specify the type constants of the object logic. (Recall that the analogous Core Datatype `expr` in the original Hybrid–Definition 2.1 on page 3–has just a single type parameter).

### *3.3 Abstractions in* Canonical HybridLF

Like Hybrid, Canonical HybridLF employs the concept of abstraction, and has function predicates that determine if certain terms are valid abstractions. However there are two main differences. The first is that we now have *type abstractions* as well as *term abstractions*. The second is that Canonical HybridLF extends the general notion of abstraction from unary abstractions to $k$-ary abstractions. Intuitively speaking these are Isabelle functions that bind exactly $k$ variables, are syntactic terms and have no dangling indices.

Canonical HybridLF has four families of functions that determine if a function represents a valid term abstraction or type abstraction. By 'families' of functions, we mean sets of functions each with the same name apart from a numerical suffix (e.g. 12) indicating the expected arity of the input (e.g. a 12-ary abstraction). The prefix of the function indicates which syntactic category the function acts upon. The production of these function variants numbered up to 12 is a pragmatic choice based on the trade-off between theory processing time and availability of functions for the user. Each additional function adds to the time necessary to process the Canonical HybridLF theory file in Isabelle. On the other hand, we required variants of ctype_bind (see Section 3.4) numbered up to 11 for our examples, so producing versions taking up to 12 variables seemed a reasonable step in practice.

These functions are analogous to abstr in the original Hybrid (see page 4). In Canonical HybridLF these four families consist of

a ctype_abstr to ctype_abstr12, p atype_abstr to atype_abstr12,

m cterm_abstr to cterm_abstr12, r aterm_abstr to aterm_abstr12.

For example cterm_abstr determines if an Isabelle function with one bound variable is a valid unary abstraction that returns a canonical term, while cterm_abstr12 determines if a function with twelve bound variables is a valid abstraction that returns a canonical term. The other three families of functions perform the same task for abstractions representing canonical types, atomic types and lastly atomic terms. Variants for up to twelve variables suffice in practice; there are no limits on the number of variables that can be handled so we could define abstr functions that allow the analysis of functions with a greater number of bound variables. We give one example definition, of cterm_abstr.

**Definition 3.3**

$$\begin{aligned}
&\mathsf{cterm_abstr}\ i\ (\lambda x.\ \mathtt{ATERM}\ x) = \mathtt{True}\\
&\mathsf{cterm_abstr}\ i\ (\lambda x.\ \mathtt{ATERM}\ (\mathtt{CON}\ a)) = \mathtt{True}\\
&\mathsf{cterm_abstr}\ i\ (\lambda x.\ \mathtt{ATERM}\ (\mathtt{BND}\ n)) = (n < i)\\
&\mathsf{cterm_abstr}\ i\ (\lambda x.\ \mathtt{ATERM}\ (\mathtt{VAR}\ n)) = \mathtt{True}\\
&\mathsf{cterm_abstr}\ i\ (\lambda x.\ \mathtt{ATERM}\ ((f\ x)\ \$\$_{\mathrm{O}}\ (g\ x))) = \mathsf{aterm_abstr}\ i\ f \wedge \mathsf{cterm_abstr}\ i\ g\\
&\mathsf{cterm_abstr}\ i\ (\lambda x.\ \mathtt{ABS}\ (t\ x)\ (f\ x)) = \mathsf{ctype_abstr}\ i\ t \wedge \mathsf{cterm_abstr}\ (i+1)\ f\\
&\neg\mathsf{cterm_ordinary}\ f \Longrightarrow \mathsf{cterm_abstr}\ i\ f = \mathtt{False}
\end{aligned}$$

Note that cterm_ordinary is used as a guard against exotic terms.

### *3.4 Conversion to de Bruijn in* Canonical HybridLF

Recall the functions lambda and lbind from section 2. These functions implement the automatic conversion of HOAS expressions with named Isabelle binders $v$ to de Bruijn form. A key contribution of the current paper is to show that the higher order pattern matching techniques used in implementing these functions in Hybrid transfer smoothly to an analogous implementation of Canonical LF, thus yielding the Canonical HybridLF system.

Conversion in Canonical HybridLF from HOAS expressions to de Bruijn terms is performed by families of functions

a ctype_bind to ctype_bind12, p atype_bind to atype_bind12,

m cterm_bind to cterm_bind12, r aterm_bind to aterm_bind12.

To create these functions we define families of functions ctype_bind' to ctype_bind'12, atype_bind' to atype_bind'12, cterm_bind' to cterm_bind'12, and finally aterm_bind' to aterm_bind'12. *The former unprimed functions are analogues of lambda, and the primed functions are analogues of lbind.* Here is the definition of cterm_bind':

**Definition 3.4**

$$\begin{aligned}
&\mathsf{cterm_bind'}\ i\ (\lambda x.\ \mathtt{ATERM}\ x) = \mathtt{Some}\ (\mathtt{ATERM}\ (\mathtt{BND}\ i))\\
&\mathsf{cterm_bind'}\ i\ (\lambda x.\ (\mathtt{ATERM}\ (\mathtt{BND}\ k))) = \mathtt{Some}\ (\mathtt{ATERM}\ (\mathtt{BND}\ k))\\
&\mathsf{cterm_bind'}\ i\ (\lambda x.\ (\mathtt{ATERM}\ (\mathtt{VAR}\ n))) = \mathtt{Some}\ (\mathtt{ATERM}\ (\mathtt{VAR}\ n))\\
&\mathsf{cterm_bind'}\ i\ (\lambda x.\ \mathtt{ATERM}\ (\mathtt{CON}\ a)) = \mathtt{Some}\ (\mathtt{ATERM}\ (\mathtt{CON}\ a))\\
&\mathsf{cterm_bind'}\ i\ (\lambda x.\ \mathtt{ATERM}\ (\mathtt{APP}\ (F\ x)\ (G\ x))) = (\mathtt{case}\\
&\quad (\mathsf{aterm_bind'}\ i\ F)\ \mathtt{of\ Some}\ atm\ \Rightarrow (\mathtt{case}\ (\mathsf{cterm_bind'}\ i\ G)\\
&\quad \mathtt{of\ Some}\ ctm \Rightarrow \mathtt{Some}\ (\mathtt{ATERM}\ (\mathtt{APP}\ atm\ ctm))\ |\ \mathtt{None} \Rightarrow \mathtt{None})\\
&\quad |\ \mathtt{None} \Rightarrow \mathtt{None})\\
&\mathsf{cterm_bind'}\ i\ (\lambda x.\ \mathtt{ABS}\ (ty\ x)\ (F\ x)) = (\mathtt{case}\\
&\quad (\mathsf{ctype_bind'}\ i\ ty)\ \mathtt{of\ Some}\ t \Rightarrow (\mathtt{case}\ (\mathsf{cterm_bind'}\ (i+1)\ F)\\
&\quad \mathtt{of\ Some}\ m \Rightarrow \mathtt{Some}\ (\mathtt{ABS}\ t\ m)\ |\ \mathtt{None} \Rightarrow \mathtt{None})\\
&\quad |\ \mathtt{None} \Rightarrow \mathtt{None})\\
&\neg\mathsf{cterm_ordinary}\ expr \Longrightarrow \mathsf{cterm_bind'}\ i\ expr = \mathtt{None}
\end{aligned}$$

The natural number argument $i$ of cterm_bind' tracks how many ABS nodes have been recursed over. Note that cterm_bind' returns a `cterm option`, returning None in the last equation when its argument is not a syntactic term, and Some $m$ when the result is a canonical term $m$. This is in contrast to the original HYBRID, which makes use of an ERR element of the core expr datatype to indicate that an error has occurred (see page 3): coding and error handling are now slicker.

The definition of cterm_bind (an analogue of lambda) is as follows:

**Definition 3.5**

$$\begin{aligned}
&\mathsf{cterm_bind}\ t\ e \equiv \mathtt{if}\ \mathsf{cterm_abstr}\ 0\ e\ \mathtt{then}\ (\mathtt{case}\ (\mathsf{cterm_bind'}\ 0\ e)\\
&\quad \mathtt{of\ Some}\ e' \Rightarrow \mathtt{Some}\ (\mathtt{ABS}\ t\ e')\ |\ \mathtt{None}\ \Rightarrow \mathtt{None})\ \mathtt{else\ None}
\end{aligned}$$

cterm_bind takes as parameters a canonical type $t$ (for the type of the bound variable) and a function $e$ to be converted to de Bruijn form. This conversion is performed by calling the cterm_bind' function on $e$ with the initial parameter of 0 for the number of binders recursed over. The aterm_bind function is defined similarly, with an aterm_bind' function performing the actual work of converting HOAS functions to de Bruijn form.

ctype_bind is defined like so:

**Definition 3.6**

$$\begin{aligned}
&\mathsf{ctype_bind}\ t\ e \equiv \mathtt{if}\ \mathsf{ctype_abstr}\ 0\ e\ \mathtt{then}\ (\mathtt{case}\ (\mathsf{ctype_bind'}\ 0\ e)\\
&\quad \mathtt{of\ Some}\ e' \Rightarrow \mathtt{Some}\ (\mathtt{PI}\ t\ e')\ |\ \mathtt{None}\ \Rightarrow \mathtt{None})\ \mathtt{else\ None}
\end{aligned}$$

Again, $t$ is a canonical type and $e$ is function to be converted to de Bruijn form. In variants of ctype_bind that convert a function with a number of variables greater

than one, there is more than one argument for the type of binders, more than one enclosing binder is added to the converted term, and the argument types of the types of binders are functions with increasing numbers of bound variables. For instance, ctype_bind3 is defined as follows:

**Definition 3.7**

$$\begin{aligned}\mathsf{ctype\_bind3}\ t1\ t2\ t3\ e \equiv\ &\texttt{if}\ \mathsf{ctype\_abstr3}\ 0\ e \wedge \mathsf{ctype\_abstr}\ 0\ t2\\ &\wedge \mathsf{ctype\_abstr2}\ 0\ t3\ \texttt{then}\ (\texttt{case}\ (\mathsf{ctype\_bind'}\ 0\ t2)\ \texttt{of Some}\ t2' \Rightarrow\\ &(\texttt{case}\ (\mathsf{ctype\_bind'2}\ 0\ t3)\ \texttt{of Some}\ t3' \Rightarrow (\texttt{case}\ (\mathsf{ctype\_bind'3}\ 0\ e)\\ &\texttt{of Some}\ e' \Rightarrow \texttt{Some}\ (\texttt{PI}\ t1\ (\texttt{PI}\ t2'\ (\texttt{PI}\ t3'\ e'))) \mid \texttt{None} \Rightarrow \texttt{None})\\ &\mid \texttt{None}\ \Rightarrow \texttt{None}) \mid \texttt{None}\ \Rightarrow \texttt{None})\ \texttt{else None}\end{aligned}$$

Note that three PI binders are added to the start of the converted term $e'$, that the type of $t1$ is `('a, 'b) ctype`, the type of $t2$ is ('a, 'b) aterm $\to$ ('a,'b) ctype and the type of $t3$ is ('a, 'b) aterm $\to$ ('a, 'b) aterm $\to$ ('a,'b) ctype. The types $t2$ and $t3$ of the latter two binders are given as functions because the variables bound in the preceding binders may appear within them.

### *3.5 Typing, kinding and substitution in* Canonical HybridLF

Recall the typing and kinding rules of canonical LF in Figure 1. In Canonical HybridLF these rules are implemented by a number of mutually-defined functions. Kinds are determined to be valid by the validkind function, while types are determined to be valid by the validtype function. Kinds are assigned to atomic types by the atom_kindof function (implementing rules con_at_kind and app_at_kind). Types are assigned to atomic and canonical terms by functions atom_typeof and canon_typeof respectively (implementing rules var_at_ty, con_at_ty, app_at_ty, and abs_can_ty).

Substitution is performed by functions corresponding to the syntactic categories of Definition 3.1. For space reasons we omit details; there is for example a function ctype_subst_fv, which substitutes a canonical term for free variables in a canonical type, implementing $[M/x]^{\mathsf{a}}_{\alpha}A$ in Figure 1.

To further explain these functions we first make some general comments. The first five arguments of the typing and substitution functions are the same; see figure 2. Since Isabelle requires all functions to terminate, we introduce a numerical recursion-depth argument to ensure that this is so. Note that all of the functions have a case for when this argument is zero which simply returns None to indicate failure. The cases for when this parameter is non-zero all pattern-match on Suc q for some q, distinguishing them from the zero case, and recursive calls within the body of the functions all give q as the first parameter, ensuring that this decreases with each recursive call. The second parameter is a context, while the third and fourth are the signature, split into object constants and type constants. The fifth parameter is the binding environment, an Isabelle list of the canonical types of the enclosing binders that allows us to type bound variables.

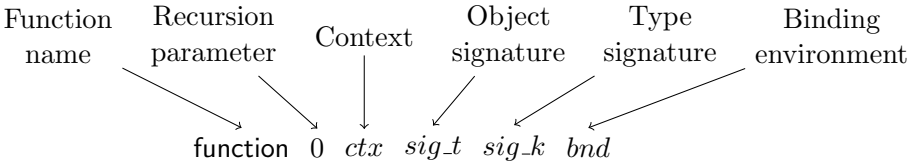


Fig. 2. First Five Parameters of Typing and Substitution Functions

In canon_typeof and atom_typeof, the sixth parameter is the canonical term or atomic term to determine the type of, while the sixth parameter in the substitution functions is the canonical term that we are substituting for free variables. The seventh parameter in the substitution functions is a natural number, the number of the variable to substitute for. The eighth is the term into which we are substituting.

We complete this section by giving some examples of the code for these functions. The atom_kindof function computes the kind of an atomic type:

**Definition 3.8**

atom_kindof 0 *ctx sig_t sig_k bnd a* = None
atom_kindof $(q+1)$ *ctx sig_t sig_k bnd* (FCON *a*) = (case sig_k_lookup
*sig_k a* of Some *k* ⇒ (if kind_level 0 *k* then Some *k* else None)
| None ⇒ None)
atom_kindof $(q+1)$ *ctx sig_t sig_k bnd* (FAPP *p m*) = (case atom_kindof
*q ctx sig_t sig_k bnd p* of Some (KPI *a k*) ⇒ (case canon_typeof *q ctx sig_t sig_k bnd m* of Some *a* ⇒ kind_subst_bv *q ctx sig_t sig_k bnd m* 0 0 *k*
| None ⇒ None) | None ⇒ None)

Notice that the single function atom_kindof provides an implementation of two rules that are found in figure 1. The second definitional function equation (that is, the one for FCON *a*) corresponds to the LF kinding rule CON_AT_KIND and the third definitional function equation (for FAPP *p m*) corresponds to the LF rule APP_AT_KIND. Remember that the first argument is used to ensure the termination of the atom_kindof function, with the base case of 0 indicating failure to determine a kind. In the case of a constant FCON *a*, a look-up is made to see if *a* is declared in the signature, and if so a check is made that the kind is of level 0, a "proper kind". In the case of an application FAPP *p m*, for example where atom_kindof succeeds, the kind of *p* and the type of *m* are extracted, then the hereditary substitution of *m* into the kind of *p* completes the computation of the kind.

The canon_typeof function computes the type of a canonical term:

**Definition 3.9**

$$
\begin{aligned}
&\mathsf{canon\_typeof}\ 0\ ctx\ sig\_t\ sig\_k\ bnd\ m = \mathtt{None}\\
&\mathsf{canon\_typeof}\ (q+1)\ ctx\ sig\_t\ sig\_k\ bnd\ (\mathtt{ATERM}\ r) =\\
&\qquad \mathsf{atom\_typeof}\ q\ ctx\ sig\_t\ sig\_k\ bnd\ r\\
&\mathsf{canon\_typeof}\ (q+1)\ ctx\ sig\_t\ sig\_k\ bnd\ (\mathtt{ABS}\ a'\ m) =\\
&\qquad (\mathtt{case}\ \mathsf{canon\_typeof}\ q\ ctx\ sig\_t\ sig\_k\ (a' \#\ bnd)\ m\ \mathtt{of\ Some}\ a \Rightarrow\\
&\qquad (\mathtt{if}\ \mathsf{ctype\_level}\ 0\ a'\ \mathtt{then\ Some}\ (\mathtt{PI}\ a'\ a)\ \mathtt{else\ None}) \mid \mathtt{None} \Rightarrow \mathtt{None})
\end{aligned}
$$

Like atom_kindof, the base case in canon_typeof returns None when the recursion-depth limiting first parameter is 0. The second equation simply calls the atom_typeof function when the last parameter is an atomic type wrapped in the ATERM constructor. The third equation (for ABS $a'$ $m$) corresponds to LF typing rule ABS_CAN_TY, and determines the type of the body $m$ of the abstraction (updating the binding environment with the type of the binder $a'$) and returning either None (if no type could be computed for $m$ or $a'$ is not a proper type) or Some $\Pi$-type with the computed type of $m$ as its body.

The atom_typeof function computes the type of an atomic term:

**Definition 3.10**

$$
\begin{aligned}
&\mathsf{atom\_typeof}\ 0\ ctx\ sig\_t\ sig\_k\ bnd\ r\ \ = \mathtt{None}\\
&\mathsf{atom\_typeof}\ (q+1)\ ctx\ sig\_t\ sig\_k\ bnd\ (\mathtt{VAR}\ v) = (\mathtt{case}\ \mathsf{ctx\_lookup}\ ctx\ v\\
&\qquad \mathtt{of\ Some}\ t \Rightarrow (\mathtt{if}\ \mathsf{ctype\_level}\ 0\ t\ \mathtt{then\ Some}\ t\ \mathtt{else\ None}) \mid \mathtt{None} \Rightarrow \mathtt{None})\\
&\mathsf{atom\_typeof}\ (q+1)\ ctx\ sig\_t\ sig\_k\ bnd\ (\mathtt{CON}\ c) = (\mathtt{case}\ \mathsf{sig\_t\_lookup}\ sig\_t\ c\\
&\qquad \mathtt{of\ Some}\ t \Rightarrow (\mathtt{if}\ \mathsf{ctype\_level}\ 0\ t\ \mathtt{then\ Some}\ t\ \mathtt{else\ None}) \mid \mathtt{None} \Rightarrow \mathtt{None})\\
&\mathsf{atom\_typeof}\ (q+1)\ ctx\ sig\_t\ sig\_k\ bnd\ (\mathtt{APP}\ r\ m) =\\
&\qquad (\mathtt{case}\ \mathsf{atom\_typeof}\ q\ ctx\ sig\_t\ sig\_k\ bnd\ r\ \mathtt{of\ Some}\ (\mathtt{PI}\ a'\ a) \Rightarrow\\
&\qquad (\mathtt{case}\ \mathsf{canon\_typeof}\ q\ ctx\ sig\_t\ sig\_k\ bnd\ m\ \mathtt{of\ Some}\ a' \Rightarrow\\
&\qquad \mathsf{ctype\_subst\_bv}\ q\ ctx\ sig\_t\ sig\_k\ bnd\ m\ 0\ 0\ a \mid \mathtt{None} \Rightarrow \mathtt{None}) \mid \mathtt{None} \Rightarrow \mathtt{None})
\end{aligned}
$$

The second equation (for VAR $v$) corresponds to typing rule VAR_AT_TY, the fourth equation (for CON $c$) corresponds to CON_AT_TY and the fifth equation (for APP $r$ $m$) corresponds to APP_AT_TY.

# 4 Case Studies and Proof System

## *4.1 Encoding Simply Typed Lambda Calculus and Quantified Propositional Logic*

We define a CANONICAL HYBRIDLF signature for the simply-typed lambda calculus, based on an example from the Twelf documentation [18]. Recalling Definition 3.2

we specify the type and object constants:

$$\texttt{datatype t\_cons} = \texttt{tp} \mid \texttt{tm} \mid \texttt{var\_of\_type} \mid \texttt{pres} \mid \texttt{val} \mid \texttt{step}$$
$$\texttt{o\_cons} = \texttt{unit} \mid \texttt{arrow} \mid \texttt{singleton} \mid \texttt{app} \mid \texttt{lam} \mid \texttt{val\_lam} \ \ldots$$

For example `step` is a standard call-by-value single-step operational reduction; `val` is a predicate for *values*, that is, final results of reductions; and `app` builds an application expression

The signature is split into two parts. `sig_kind` specifies the kinds of type constants. `sig_type_option` specifies the types of object constants and is defined by

$$(\texttt{o\_cons} \times \boxed{(\texttt{o\_cons},\ \texttt{t\_cons})\ \texttt{ctype}}\ \texttt{option})\ \texttt{list}$$

meaning that the `option` element of the type must be removed by the user, but fortunately this is easy to do. For example, the constant `val_lam` is used to assert that any lambda-expression is a value, and has type [4] `Some` $\xi$ where $\xi$ is

$$\texttt{PI}\ (\texttt{tm} \Rightarrow \texttt{tm})\ (\lambda E.\ \texttt{PI tp}\ (\lambda\, T.\ \texttt{val}\ \$\$_{\mathrm{F}}\ (\texttt{lam}\ \$\$_{\mathrm{O}}\ T\ \$\$_{\mathrm{O}}\ E))) \\ : (\texttt{o\_cons},\ \texttt{t\_cons})\ \texttt{ctype}$$

and $\lambda$ is Isabelle function abstraction. Here we are exploiting HOAS, with the (higher order) type of $E$ being $\texttt{tm} \Rightarrow \texttt{tm}$; there is no need for a well-formedness predicate (as in Hybrid) since the type system performs this role. $\xi$ is sugar for a call to a conversion function `ctype_bind2`

$$\texttt{ctype\_bind2}\ (\texttt{tm} \Rightarrow \texttt{tm})\ (\lambda E.\ \texttt{tp})\ (\lambda \texttt{E}.\ \lambda T.\ \ \texttt{val}\ \$\$_{\mathrm{F}}\ (\texttt{lam}\ \$\$_{\mathrm{O}}\ T\ \$\$_{\mathrm{O}}\ \texttt{E}))$$

and this evaluates in Canonical HybridLF to the (first order) expression

$$\texttt{PI}\ (\texttt{tm} \Rightarrow \texttt{tm})\ (\texttt{PI tp}\ (\texttt{val}\ \$\$_{\mathrm{F}}\ ((\texttt{lam}\ \$\$_{\mathrm{O}}\ (\texttt{BND}\ 0))\ \$\$_{\mathrm{O}}\ (\texttt{BND}\ 1))))$$

The *metatheorem* `pres` defined in this example is that of *type preservation during evaluation*. A proof that the type preservation metatheorem represented by the `pres` type holds would be derived in the $M_2$ metalogic which we briefly describe in the next section.

For the second case study, and a comparison to the original Hybrid system as described in [1], we return to the example of Quantified Propositional Logic (QPL) and produce definitions in both Hybrid and Canonical HybridLF. QPL has syntax

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

We can inductively define negation normal form, a function (implemented as a deterministic relation) to convert QPL expressions to negation normal form, and

[4] Note, for readability, we have omitted all constructors from instances of Definition 3.2.

prove a correctness theorem that asserts the conversion function outputs negation normal forms; see [1].

*QPL in* Hybrid

Here is a snapshot of code (recall the Hybrid Core Datatype `'a expr` on page 3)

```
datatype cons = cATOM | cAND |  cIMP | cFORALL | ... | cZERO | cSUCC

inductive nnf :: "cons expr -> cons expr -> bool"

...
nnf_forall:
"[[ abstr f; abstr g;
    ALL n. isNat n --> nnf (f (cATOM $$ n)) (g (cATOM $$ n)) ]]
 ===>  nnf (cFORALL $$ (LAM x. f x)) (cFORALL $$ (LAM x. g x))"
...
lemma is_nnf_form: "is_nnf a ===> isForm a"
...
lemma nnf_correct: "nnf a b ===> is_nnf b"
```

Notice that in defining `nnf`, the rule `nnf_forall` for ∀-quantification is encoded in HOAS; we declare the abstractions `f`, `g` (and `LAM` from the Hybrid package) and then write (for example) `cFORALL $$ (LAM x. f x)`. However the "untyped" Core Datatype leads to the additional burden of a predicate `isNat` that picks out expressions corresponding to natural numbers and a predicate `isForm` that picks out expressions corresponding to QPL formulae, which are used during proofs.

*QPL in* Canonical HybridLF

A snapshot of code appears below, where we write `\v` to denote the Isabelle function abstraction (recall the Canonical HybridLF Core Datatype `'a expr` on page 7)

```
datatype t_cons = nat | nnf | form | is_nnf | nnf_correct

datatype o_cons = cZERO | cSUCC | cATOM | .. | cFORALL | cEXISTS
                | is_nnf_* | nnf_* | nnf_correct_*

                where * is ATOM, FORALL, NOT_FORALL, EXISTS, .. etc

...
(nnf, (KPI (ATYPE (FCON form)) (KPI (ATYPE (FCON form)) TYPE))),
...
(nnf_FORALL, (ctype_bind4 ((nat))
  (\n. PI ((form)) ((form))) (\n.\x. (PI ((form)) ((form))))
  (\n.\x.\y. (nnf $$T ((x $$0 (cATOM $$0 n)))
                                        $$T (y $$0 (cATOM $$0 n))))
  (\n.\x.\y.\z. (nnf $$T ((cFORALL $$0 x)) $$T ((cFORALL $$0 y))))))
```

Notice that some of the type constants are used to name the judgements to be proved. `form` is the type of formulae; `nnf` maps a QPL formula to one in negation normal form. Corresponding to the Hybrid inductive definition of `nnf`, with rules such as `nnf_forall`, there are object constants such as `nnf_FORALL`.

*Comparing the Two Encodings*

The Canonical HybridLF definition of QPL avoids the need for predicates such as `isNat` that are required in Hybrid, which is a clear improvement. However, the Canonical HybridLF signature which specifies (that is, implements) QPL is considerably longer than the Isabelle HOL datatype, relation and lemma definitions that constitute the Hybrid implementation of QPL. The terms comprising the Canonical HybridLF signature are also more visually cluttered than the Hybrid definitions.

### *4.2 A Logic for* Canonical HybridLF

Canonical HybridLF employs the $M_2$ metalogic of Schürmann and Pfenning [16] to prove that theorems in the signature are valid. A principal reason for choosing $M_2$ is a general familiarity with work by Pfenning and co-authors, plus its relative simplicity, and the fact that it is powerful enough to prove a range of metatheorems. Indeed $M_2$ is designed for constructing proofs over LF specifications (we could also consider more powerful logics, but $M_2$ seemed a sensible first choice).

$M_2$ is a first-order sequent calculus with proof terms, where the proof terms of a complete derivation form a total function from universally-quantified variables to existentially-quantified variables. Formulae in $M_2$ have the form

$$\forall x_1{:}A_1 \ldots \forall x_k{:}A_k.\exists x_{k+1}{:}A_{k+1} \ldots \exists x_m{:}A_m.\top$$

where $A_1$ to $A_m$ are valid LF types, $x_1 \ldots x_k$ and $x_{k+1} \ldots x_m$ are valid contexts, quantifiers range over closed LF objects from the signature that the formula is defined for and the $\top$ symbol stands for truth. Such formulae correspond to *totality* assertions in the *schema-checking* approach employed by Twelf in which variables of the meta-theorem are designated as inputs and outputs, and the system checks that every input has an output. Wang and Nadathur [19] formalise the connection between Twelf schema-checking operations and $M_2$ derivations. $M_2$ is limited in that it can only reason about closed objects, not open objects (i.e. with free variables); Schürmann [17] introduces a further logic $M_2^+$ that allows reasoning about open terms in a non-empty context, which would be an important enhancement to Canonical HybridLF, but we leave the implementation of this to future work.

$M_2$ has two judgements: $\longrightarrow$ and $\longrightarrow_\Sigma$. The $\longrightarrow$ judgement determines derivability in $M_2$, while $\longrightarrow_\Sigma$ is an additional judgement that performs case analysis on the LF signature, attempting to unify a variable in the context with terms from the signature. If the terms unify to produce an MGU $\sigma$ then $\longrightarrow_\Sigma$ requires that a derivation of $\longrightarrow$ exists for the goal formula after $\sigma$ has been applied.

To define the proof terms of $M_2$, we must first define type synonyms for substi-

tutions, contexts, formulae and patterns. A substitution consists of pairs relating a variable number to be substituted for to the term that is to be substituted for it. Contexts consist of pairs of a free variable number and the type of the free variable. Formulae are given by a pair of contexts - the first containing universally quantified input variables, and the second containing existentially quantified output variables.

We then define the datatype representing $M_2$ proof terms (definition omitted). In a complete $M_2$ derivation these proof terms form a total function from universally-quantified variables in the formula to existentially-quantified variables.

We implement the proof rules of $M_2$ as a pair of Isabelle inductive relations: derivation (corresponding to $\longrightarrow$) and sig_derivation (corresponding to $\longrightarrow_\Sigma$). Space prevents us from a detailed exposition of this important aspect of our work.

## 5 Conclusions and Future and Related Work

Our primary question, asking if the techniques of Hybrid can be migrated to a dependently typed setting, has been answered affirmatively. We would like to thank Christian Urban for his useful comments and feedback on our work, and to the referees' comments that have lead to considerable improvements.

In Canonical HybridLF we make a number of advances over the Hybrid systems. All of these perform as they should, and they enable the use of an additional way of reasoning about deductive systems. The primary gain is the ability to exploit the judgements as types methodology and use the type system to guide the well-formedness of expressions. Another key advantage with Canonical HybridLF over earlier incarnations of Hybrid is the ability to convert HOAS functions with more than one bound variable (the finitary abstractions). The method by which we achieve this, by creating families of binding functions, may initially seem like a 'brute force' approach. However, it is important to remember that Isabelle does not support variadic functions. We therefore do not lose anything by defining a binding function for each arity of HOAS function. That said, it will be interesting in the future to find more radical ways in which this issue can be circumvented, or even implemented in a system other than Isabelle HOL (recall that there are Coq implementations of Hybrid [3].)

In Hybrid, HOAS terms are converted to de Bruijn terms and reasoned about directly in Isabelle, using Isabelle's higher-order logic and the Isabelle `lemma` construct to create proofs. In Canonical HybridLF the theorems and meta-theorems are defined in the signature, and proved correct using the proof rules of $M_2$. The main drawback to this approach is that the search for a proof of correctness is long and tedious, whereas in Twelf this proof search is carried out automatically. As it stands, Canonical HybridLF is perhaps not so suitable for "practical" theorem proving, as it lacks automation of unification and proofs of totality which can be very unwieldy to create by hand. We acknowledge this and wish to make it clear that further work needs to be done to properly exploit Canonical HybridLF.

Further work also needs to be done on the underlying theory of our systems. This could be at a metalevel, for example looking at adequacy along the lines of [5].

But we also need to better understand the properties of Canonical HybridLF, such as decidability of the type system which is allied to the rather pure nature of the underlying equality. Unification and decidability also require considerable further study in the case of our system based on LF itself.

To finish, some comments on related work: A number of approaches to circumventing the limitations of HOAS appear in the literature. One such approach is that of Despeyroux et al [6], in which a type `var` of variables is introduced and used as the source type of the binder functions, thus allowing injections of (`var` $\Rightarrow$ `exp`) into `exp`. The key disadvantage with this methodology is that substitution in the object logic is no longer implemented as $\beta$-reduction in the metalogic. Momigliano et al [13] refer to techniques such as this, in which object logic bound variables are encoded as metalogic bound variables and object logic contexts are encoded as metalogic contexts but substitution is not implemented as metalogic $\beta$-reduction, as *weak HOAS*. Chlipala [4] describes *parametric higher-order abstract syntax*, another (but related) approach to weak HOAS, in which a type parameter is introduced as the type of variables. Recently, some challenges for HOAS have been documented in [8] by Felty, Momigliano, and Pientka. Hybrid provided the possibility of blending HOAS and co-induction principles and it also remains further work to consider co-induction within Canonical HybridLF.

## References

[1] S. J. Ambler, R. L. Crole, and A. Momigliano. *Combining Higher Order Abstract Syntax with Tactical Theorem Proving and (Co)Induction*. In Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics, Hampton, VA, USA, volume 2410 of Lecture Notes in Computer Science, pp 13-30. Springer-Verlag, 2002. Editors Victor Carreño, César Muñoz, and Sofiène Tahar.

[2] N. de Bruijn, *Lambda Calculus Notation with Nameless Dummies: a Tool for Automatic Formula Manipulation, with Application to the Church Rosser Theorem*. In Indagationes Mathematicae, volume 34, pp 381-391, 1972.

[3] Venanzio Capretta and Amy Felty, Higher-Order Abstract Syntax in Type Theory. In Logic Colloquium '06, ASL Lecture Notes in Logic, 32, Cambridge University Press, October 2009 (pdf, Coq formalization, Example application).

[4] Adam Chlipala. *Parametric higher-order abstract syntax for mechanized semantics*. In ICFP '08 Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming, pages 143-156, 2008.

[5] Roy L. Crole, *Representational Adequacy for Hybrid*. In Mathematical Structures in Computer Science, volume 21, number 3, pp 585-646, 2011.

[6] Joëlle Despeyroux, Amy Felty and André Hirschowitz, *Higher-order abstract syntax in Coq*. In Typed Lambda Calculi and Applications, Lecture Notes in Computer Science Volume 902, pp 124-138, 1995.

[7] Amy Felty and Alberto Momigliano, *Hybrid: A Definitional Two Level Approach to Reasoning with Higher-Order Abstract Syntax*. In Journal of Automated Reasoning, 48(1):43-105, 2012, DOI 10.1007/s10817-010-9194-x (SpringerLink, pdf).

[8] Amy P. Felty, Alberto Momigliano, and Brigitte Pientka, *The Next 700 Challenge Problems for Reasoning with Higher-Order Abstract Syntax Representations: Part 1A Common Infrastructure for Benchmarks*, by 2015, (arXiv, pdf).

[9] Nada Habli and Amy Felty, *Translating Higher-Order Specifications to Coq Libraries Supporting Hybrid Proofs*. In Third International Workshop on Proof Exchange for Theorem Proving, EasyChair Proceedings in Computing, Volume 14, pages 67-76, 2013, (pdf).

[10] Robert Harper, Furio Honsell and Gordon Plotkin. *A Framework for Defining Logics*. In Journal of the ACM, Volume 40 Issue 1, pp. 143-184, 1993.

[11] Robert Harper and Daniel R. Licata. *Mechanizing metatheory in a logical framework*. In Journal of Functional Programming, Volume 17 Issue 4-5, pp 613-673, 2007.

[12] Alan Martin. *Reasoning Using Higher-Order Abstract Syntax in a Higher-Order Logic Proof Environment: Improvements to Hybrid and a Case Study*. PhD Thesis, University of Ottawa, 2010.

[13] A. Momigliano, S. J. Ambler, and R. L. Crole. *A Comparison of Formalizations of the Meta-Theory of a Language with Variable Bindings in Isabelle*. In Richard J. Boulton and Paul B. Jackson, editors, Supplemental Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics, pp 267-282, 2001.

[14] Frank Pfenning and Carsten Schu̇rmann. *System description: Twelf - a meta-logical framework for deductive systems*. In H. Ganzinger (ed.) CADE 1999. LNCS(LNAI), Volume 1632, pp 202-206, 1999.

[15] A. Popescu, E. Gunter, C. J. Osborn. Strong Normalisation for System F by HOAS on Top of FOAS. LICS 2010, pp 31-40.

[16] Carsten Schürmann and Frank Pfenning. *Automated Theorem Proving in a Simple Meta-Logic for LF*, In CADE-15 Proceedings of the 15th International Conference on Automated Deduction, pp 286-300, 1998.

[17] Carsten Schürmann. *Automating the Meta Theory of Deductive Systems*, PhD Thesis, Carnegie-Mellon University, 2000.

[18] *Proving Metatheorems With Twelf*. http://www.twelf.org/wiki/Proving_metatheorems_with_Twelf, 2007 [Accessed 24th April 2015].

[19] Yuting Wang and Gopalan Nadathur. *Towards Extracting Explicit Proofs From Totality Checking In Twelf*, In LFMTP '13 Proceedings of the Eighth ACM SIGPLAN International Workshop on Logical Frameworks and Meta-Languages, Pages 55-66, 2013.

[20] Kevin Watkins, Iliano Cervesato, Frank Pfenning and David Walker. *A Concurrent Logical Framework I: Judgments and Properties*. Carnegie-Mellon University Department of Computer Science Technical Report CMU-CS-02-101, 2003.