The Representational Adequacy of HYBRID

R. L. Crole

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

14th March 2011.

E: R.Crole@mcs.le.ac.uk
T: +44 116 252 3404
F: +44 116 252 3915

ABSTRACT

The HYBRID system (Ambler et al., 2002b), implemented within Isabelle/HOL, allows object logics to be represented using higher order abstract syntax (HOAS), and reasoned about using tactical theorem proving in general and principles of (co)induction in particular. The form of HOAS provided by HYBRID is essentially a lambda calculus with constants.

Of fundamental interest is the form of the lambda abstractions provided by HYBRID. The user has the convenience of writing lambda abstractions using names for the binding variables. However each abstraction is actually a *definition* of a de Bruijn expression, and HYBRID can unwind the user's abstractions (written with names) to machine friendly de Bruijn expressions (without names). In this sense the formal system contains a *hybrid* of named and nameless bound variable notation.

In this paper, we present a formal theory in a logical framework which can be viewed as a model of core HYBRID, and state and prove that the model is representationally adequate for HOAS. In particular, it is the canonical translation function from λ-expressions to HYBRID that witnesses adequacy. We also prove two results that characterise how HYBRID represents certain classes of λ-expressions.

We provide the first detailed proof to be published that proper locally nameless de Bruijn expressions and α-equivalence classes of λ-expressions are in bijective correspondence. This result is presented as a form of de Bruijn representational adequacy, and is a key component of the proof of HYBRID adequacy.

The HYBRID system contains a number of different syntactic classes of expression, and associated abstraction mechanisms. Hence this paper also aims to provide a self-contained theoretical introduction to both the syntax and key ideas of the system; background in automated theorem proving is not essential, although this paper will be of considerable interest to those who wish to work with HYBRID in Isabelle/HOL.

1. Introduction

1.1. *Overview*

Many people are concerned with the development of computing systems which can be used to reason about and prove properties of programming languages. In previous work (Ambler et al., 2002b), we developed the HYBRID logical system, implemented as a theory in Isabelle/HOL, for exactly this purpose. *In this paper we develop an underpinning theory for* HYBRID. The key features of HYBRID are:

— it provides a form of *logical system* within which the syntax of an object level logic can be adequately represented by *higher order abstract syntax* (HOAS);

— it is consistent with *tactical theorem proving* in general, and principles of *induction and coinduction* in particular; and

— it is *definitional* which guarantees consistency *within a classical type theory.*

We will give an overview of the first feature, which should provide sufficient details for the understanding of this paper. The other features have been discussed in (Ambler et al., 2002b; Ambler et al., 2004). We then proceed to prove our main theorem, *an adequacy result for the* HYBRID *system.* Informally, adequacy shows that HYBRID yields a well-defined form of HOAS, into which object level logics can be translated and reasoned about. More formally, we define an idealised mathematical model of HYBRID, and then by taking HOAS to be a λ-calculus with constants, we will prove that there is a representationally adequate mapping (see (Harper et al., 1993; Pfenning, 2003; Harper and Licata, 2007)) from HOAS into our mathematical model of HYBRID.

In order to achieve this, we prove that proper locally nameless de Bruijn expressions and α-equivalence classes of λ-expressions are in bijective correspondence. Although this is "known" to everyone in the community in an informal sense—one knows how to convert a de Bruijn expression to a λ-expression and vice-versa—proving the existence of a bijection is not very easy. There are some formal proofs of similar bijections in the literature, and we discuss such results in detail in Section 7. However, the published results either do not concern locally nameless de Bruijn expressions, or do not consider α-equivalence classes of λ-expression syntax trees with substitution defined by primitive recursion. This is the first time a detailed proof has been written down for this particular pair of systems. While it is true that the conceptual ideas underlying the proof and appearing in this paper are quite similar to other related proofs in the literature, we include a detailed proof since doing so allows us to *set up our own notation and machinery which is used to give crucial intensional details within our proofs of* HYBRID *adequacy. The proof method plays a key rôle in this paper.*

1.2. *A Roadmap for the Paper*

We proceed as follows. In the remainder of the introduction we define abstractly our notion of representational adequacy; review notation for de Bruijn expressions and λ-expressions; and give a table summarising the various forms of syntax used in this paper.

In Section 2 we explain informally how HYBRID represents and manipulates binders. A particular function *lbnd* plays a central rôle, so we motivate the definition of this function and give examples. In Section 3 we present a mathematical model of (a core of) HYBRID and prove the existence of *lbnd*. In Section 4 we prove in detail that de Bruijn expressions provide a representationally adequate model of the λ-expressions. The key result is Theorem 4.1. In Section 5 we state and prove HYBRID adequacy making use of the results and notation from Section 4. The key result is Theorem 5.2. In Section 6 we state and prove some simple representation results. In Section 7 we present an overview of related work with the common theme of variable binding. In Section 8 we conclude.

1.3. *Representational Adequacy*

We explain the precise form of representational adequacy that we use in this paper. Accounts of adequacy may be found in (Harper et al., 1993; Harper and Licata, 2007).

Suppose that E_1 and E_2 are sets of expressions from equational (type) theories which make use of a notion of substitution. Let V be a set of variables. A notion of substitution (Fiore, 2006) is (typically) a function

$$s : \mathsf{E}_i \times \mathsf{E}_i \times \mathsf{V} \to \mathsf{E}_i$$

and λ-calculus with capture avoiding substitution is an obvious example. We shall say $\theta : \mathsf{E}_1 \to \mathsf{E}_2$ is a **compositional homomorphism** if it preserves substitution, that is,

$$\theta(s(E, E', v)) = s(\theta(E), \theta(E'), \theta(v))$$

We say that E_1 is **compositionally isomorphic** to E_2, $\mathsf{E}_1 \cong \mathsf{E}_2$, if there are mutually inverse homomorphisms

$$\theta : \mathsf{E}_1 \rightleftarrows \mathsf{E}_2 : \phi$$

Note that this is easily seen to be equivalent to requiring θ to be a *bijective compositional homomorphism* (which we refer to as properties **B** and **CH**).

Then we say that E_2 provides an **adequate representation** of E_1 provided that we can find a subset $S \subseteq \mathsf{E}_2$ together with an isomorphism of theories $\theta : \mathsf{E}_1 \cong S : \phi$. We shall normally show this by proving that θ is a bijection (**B**) (with inverse ϕ) and that θ is a compositional homomorphism (**CH**). See for example Theorem 4.1 on page 15.

1.4. *de Bruijn Expressions and λ-Expressions*

We assume familiarity with de Bruijn expressions and λ-expressions but summarise our own notation. In particular we wish to make the reader aware of the kind of de Bruijn expressions with which we work in this paper. If required, more details can be found in the Appendix in Section B.

The set of (object level) **locally nameless de Bruijn expressions** (de Bruijn, 1972; Gordon, 1994) is denoted by $\mathcal{DB}$ and generated by

$$D ::= \mathsf{con}(\nu) \mid \mathsf{var}(i) \mid \mathsf{bnd}(j) \mid \mathsf{abs}(D) \mid D_1 \ \$ \ D_2$$

where i and j range over the natural numbers $\mathbb{N}$, and ν over a set of names. We use the usual notion of *level* (a definition is in the Appendix), written down as a predicate $\mathit{level}\, n : \mathcal{DB} \to \mathbb{B}$ for each $n \in \mathbb{N}$. We write $\mathcal{DB}(l)$ for the set of de Bruijn expressions at level l, and so $\mathcal{PDB} \stackrel{\text{def}}{=} \mathcal{DB}(0)$ is the set of **proper** de Bruijn expressions.

To state adequacy we will need a suitable notion of substitution on de Bruijn expressions, and this is formulated in the next lemma.

Proposition 1.1. *(de Bruijn Substitution) For any $m \geq m' \geq 0$ and $k \geq 0$ there is a function $\mathcal{DB}(m) \times \mathcal{DB}(m') \times \mathbb{N} \to \mathcal{DB}(m)$ given by $(D, D', k) \mapsto D[D'/\mathsf{var}(k)]$, which, informally, maps (D, D', k) to the expression D in which all occurrences of $\mathsf{var}(k)$ are replaced by D'.*

Proof. The substitution function can be defined by simple structural recursion in the expected way: there is no notion of variable renaming because bound and free variables are syntactically distinguished—one key advantage of locally nameless de Bruijn expressions. One does need to prove that the function has the stated source and target but we omit the easy proof. □

We also set up a notation for the traditional λ-calculus. The (object level) expressions are inductively defined by the grammar

$$E ::= \nu \mid v_k \mid \lambda\, v_k.\, E \mid E\ E$$

and we write $\mathcal{LE}$ for the set of all expressions. Given expressions E and E', and a variable v_k, then we write $E[E'/v_k]$ for a *unique* expression which, informally, is E with free occurrences of v_k replaced by E', with re-naming to avoid capture. If expressions E and E' are α-equivalent, we write $E \sim_\alpha E'$. We write $[E]_\alpha$ for the α-equivalence class of E and $\mathcal{LE}/\!\sim_\alpha$ for the set of α-equivalence classes of expressions. For this paper we will need a notion of substitution on $\mathcal{LE}/\!\sim_\alpha$, analogous to Proposition 1.1.

Proposition 1.2. *Let Var be the set of variables. There is a well-defined function*

$$\mathcal{LE}/\!\sim_\alpha \times \mathcal{LE}/\!\sim_\alpha \times \mathit{Var} \to \mathcal{LE}/\!\sim_\alpha \qquad ([E]_\alpha, [E']_\alpha, v_i) \mapsto [E[E'/v_i]]_\alpha$$

Syntax	Informal Description	Defined Informally	Defined Formally
$\mathsf{abs}(D)$	object level de Bruijn abstraction	4	40
$D_1\ \$\ D_2$	object level de Bruijn application	4	40
$\lambda\, v_i.\, E$	object level lambda abstraction	4	41
$E_1\ E_2$	object level lambda application	4	41
$\mathsf{ABS}\ C$	HYBRID de Bruijn abstraction	7, 7	11
$\mathsf{LAM}\, v_i.\, C$	HYBRID lambda abstraction	7, 7	19
$C_1\ \$\$\ C_2$	HYBRID de Bruijn application	7, 7	11
$\Lambda\, v_i.\, C$	HYBRID meta-abstraction	5	11
$C_1\ C_2$	HYBRID meta-application	5	11

Fig. 1. Abstraction and Application Notation

1.5. *Object Level and* HYBRID *Level Syntax*

In this paper there are a variety of binding operations arising from variants of functional abstraction, together with associated applications. This is potentially confusing so we give a look-up table in Figure 1. In the remainder of the paper we will explain the rôle of HYBRID syntax which is summarised in Figure 1 (along with the syntax for de Bruijn expressions and λ-expressions). The informal rôles of the expressions are as follows:

— The object level syntax should be thought of as an idealised mathematical system, and independent of HYBRID. We will use it when we establish a formal bijection between de Bruijn expressions and λ-expressions.
— The HYBRID level syntax corresponds to that in the implemented system. There is a single de Bruijn application \$\$. There are two (hybrid) forms of abstraction operators, ABS and $\mathsf{LAM}\, v_i.$, and these will be explained in detail in due course. The meta-application and meta-abstraction correspond to the application and abstraction of the Isabelle/HOL theorem prover at the meta level.
— We will connect the two systems by showing that HYBRID is adequate for λ-expressions.

Please also note that, strictly speaking, we should distinguish between object level and meta level variables, writing for example V_i and v_i respectively. However no technical problems either arise, or are hidden, by identifying the syntax for variables.

We shall often wish to define and talk about higher order functions in this paper. Suppose that f has a type $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3 \Rightarrow \tau_4$. Formally there are families of functions

$(f\ e_1 \mid e_1 \in \tau_1)$ and $(f\ e_1\ e_2 \mid e_1 \in \tau_1 \wedge e_2 \in \tau_2)$ and so on. We shall sometimes use informal phrases such as

— "the function $f\ e_1$" or
— "the family of functions $f\ e_1\ e_2$" or
— "the function $f\ e_1\ e_2$"

when talking about such families, and will vary the exact choice of words according to the discussion at hand. For example, it is often cleaner to write "we shall define the function $f\ e_1$" than to be pedantic and write "we shall define the (family of) functions $f\ e_1$ where e_1 ranges over all elements of τ_1"—the former should be understood as standing for the latter.

2. The Heart of the HYBRID System

2.1. *A New Variable Binding Mechanism and Induction Principle*

The goal of this section is to give a high level overview of the development of the key ideas in HYBRID, which were originally developed in collaboration with Simon Ambler and Alberto Momigliano. Our goal was to define a datatype for λ-calculus with constants over which we can deploy (co)induction principles. The datatype is regarded as a form of HOAS into which object logics can be translated and reasoned about; one can view HYBRID as a form of logical framework, in which both HOAS and (co)induction are consistent and available to the user. The user can work with binding constructs and in particular named variables—object level variable binding is realised by Isabelle/HOL's internal meta-variable binding—and we will explain how exactly this works very shortly. This makes the system user friendly. However, crucially, HYBRID can convert expressions of the λ-calculus datatype into de Bruijn expressions for reasoning within the machine.

The starting point was the work (Gordon, 1994) of Andrew Gordon, which we briefly review. Gordon defines a notation in which expressions have *named free variables* given by *strings*. He can write $E =$ dLAMBDA v e (where v is a string) which corresponds to a λ-abstraction in which dLAMBDA acts as the usual binder. But in fact dLAMBDA is a function and is defined so that E may be proven equal to a de Bruijn expression which has an outer abstraction, and an immediate subexpression which is e in de Bruijn form (in which any of the free occurrences of v in e—which were bound by the outer binder dLAMBDA in E—have been converted to bound de Bruijn indices). For example,

dLAMBDA v (dAPP (dVAR v) (dVAR u)) =
dABS (dAPP (dBND 0) (dVAR u))

Gordon demonstrates the utility of this idea within a theorem prover. The user may work with expressions which have explicit named binding variables (such as the v above), but such expressions are automatically converted to de Bruijn form by the machine, with

which it is easier for the machine to work. The idea provides a good mechanism through which the user may work with named bound variables, but it does not exploit the built in meta-level α-equivalence which a theorem prover typically possesses. *What would be ideal is a system in which the string binders of Gordon are replaced by meta-level binders. This is exactly what* HYBRID *achieves, and is one of the key novelties of our approach.*

HYBRID provides a binding mechanism similar to dLAMBDA. Gordon's E would be written $\mathsf{LAM}\, v.\, C$ in HYBRID. This is in fact a *definition* for a de Bruijn expression; and $\mathsf{LAM}\, v.\, C$ can indeed be proved equal to a de Bruijn expression involving only ABS and \$\$. A *crucial difference* in our approach is that *bound variables* are actually *bound meta-variables in Isabelle/HOL*. Thus the v in $\mathsf{LAM}\, v.\, C$ is a meta-variable (and not a string as in Gordon's approach) allowing us to exploit the meta-level α-equivalence.

We give some illustrative examples. Let $E_O = \lambda\, v_8.\, \lambda\, v_2.\, v_8\ v_3$ and think of this as an object level expression. Gordon would represent this by

$$E_G = \mathsf{dLAMBDA}\ \mathtt{v8}\ (\mathsf{dLAMBDA}\ \mathtt{v2}\ (\mathsf{dAPP}\ (\mathsf{dVAR}\ \mathtt{v8})\ (\mathsf{dVAR}\ \mathtt{v3})))$$

which equals dABS (dABS (dAPP (dBND 1) (dVAR v3))). In HYBRID we choose to denote object level free variables by expressions of the form $\mathsf{VAR}\ i$. This has essentially no impact on the key technical details relating to binders. In HYBRID the E_O above is rendered as $E_H \stackrel{\text{def}}{=} \mathsf{LAM}\, v_8.\, (\mathsf{LAM}\, v_2.\, (v_8\ \$\$\ \mathsf{VAR}\ 3))$ where each subexpression $\mathsf{LAM}\, v_i.\, \xi$ is an Isabelle/HOL binder. And in HYBRID E_H is provably equal to an expression

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

with the overall effect being analogous to Gordon's approach, but the underlying definitions and details being very different.

To summarise, HYBRID is a theory defined in Isabelle/HOL. The theory contains a specification of a datatype *exp* of de Bruijn expressions, where i and j are de Bruijn indices, and ν are names for constants:

$$C ::= \mathsf{CON}\ \nu \mid \mathsf{VAR}\ i \mid \mathsf{BND}\ j \mid C\ \$\$\ C \mid \mathsf{ABS}\ C$$

Moreover, there is a specification for expressions of the form $\mathsf{LAM}\, v_i.\, C$ which are built out of Isabelle/HOL binders. The reader may wish to look back at the syntax summary in Figure 1 on page 5 and there is a summary of the key principles of HYBRID in Figure 2. The aim of this paper is to show that HYBRID really does provide a representation of the λ-calculus:

One can regard the clauses of Figure 2 as informally specifying a function

$$\Theta :\ \text{(object level) } \lambda\text{-expressions} \longrightarrow \text{HYBRID},$$

and our main theorem (Theorem 5.2) is a proof that the function Θ (which we shall formally define) is representationally adequate. The function will be defined by recursion where (for example) $\Theta\, (\lambda\, v_j.\, E) \stackrel{\text{def}}{=} \mathsf{LAM}\, v_j.\, (\Theta\, E)$ and so on.

— object level free variables v_i are expressed as HYBRID expressions of the form $\mathsf{VAR}\ i$;
— object level bound variables v_j are expressed as HYBRID (bound) meta-variables v_j;
— object level abstractions $\lambda\, v_j.\, E$ are expressed as HYBRID expressions $\mathsf{LAM}\, v_j.\, C$; and
— object level applications $E_1\ E_2$ are expressed as HYBRID expressions $C_1\ \$\$\ C_2$.

– Eg: $\lambda\, v_8.\, \lambda\, v_2.\, v_8\ v_3$ is expressed as $\mathsf{LAM}\, v_8.\, (\mathsf{LAM}\, v_2.\, (v_8\ \$\$\ \mathsf{VAR}\ 3))$.

— HYBRID expressions are provably equal to de Bruijn expressions.†

– Eg: $\mathsf{LAM}\, v_8.\, (\mathsf{LAM}\, v_2.\, (v_8\ \$\$\ \mathsf{VAR}\ 3))$ is provably equal to the de Bruijn expression $\mathsf{ABS}\ (\mathsf{ABS}\ (\mathsf{BND}\ 1\ \$\$\ \mathsf{VAR}\ 3))$.

Fig. 2. Key Principles of HYBRID Syntax

2.2. *Reducing Named Binders to Nameless Binders*

In formally stating the adequacy theorem we utilise an auxiliary function *lbnd* (whose existence is proven in the future Proposition 3.1). This function is of central importance, and so we will give some illustrative examples and informal explanation of how *lbnd* works before stating and proving the proposition. Consider the expression $E_O \stackrel{\text{def}}{=} \lambda\, v_8.\, \lambda\, v_2.\, v_8\ v_2$. A key feature of HYBRID is that it provides the user with a syntax involving explicitly named binders. This expression is encoded in HYBRID as

$$E_H \stackrel{\text{def}}{=} \mathsf{LAM}\, v_8.\, (\mathsf{LAM}\, v_2.\, (v_8\ \$\$\ v_2))$$

Notice that it is very easy to write down HYBRID representations of object level expressions: one replaces object level abstractions λ with "meta-level abstractions" LAM, and object level application (juxtaposition) with de Bruijn application \$\$.

Let us think about how we might be able to "prove that HYBRID expressions are equal to de Bruijn expressions". Of course the only difficulty arises from LAM binders.

Suppose, as a first thought, that in fact $\mathsf{LAM}\, v_i.\, \xi$ denotes $\mathsf{ABS}\ (\Lambda\, v_i.\, \xi)$.‡ Then E_H would be

$$\mathsf{ABS}\ (\Lambda\, v_8.\, (\mathsf{ABS}\ (\Lambda\, v_2.\, (v_8\ \$\$\ v_2))))$$

This expression almost has the structure of an appropriate de Bruijn expression, except that the meta-variables v_8 and v_2 should be $\mathsf{BND}\ 1$ and $\mathsf{BND}\ 0$ respectively, and the

‡ This is not type correct since in fact $\mathsf{ABS} :: \mathrm{exp} \Rightarrow \mathrm{exp}$ (see Figure 3)! However, it is a first informal thought in the progression towards HOAS—a LAM abstraction is understood as a constructor ABS applied to a meta-abstraction. These type-incorrect expressions appear only in the current informal explanations.

meta-abstractions should be removed. We need a scheme to count up the number of ABS nodes on the path from each (bound) variable to the root in order to compute bound de Bruijn indices such as the 1 and 0. During the counting process, the Λ-meta binders and binding variables should be removed. Thus we will define a function, say $lbnd_n$, with a parameter n and defined by recursion which

— recursively descends through the ABS nodes and increases parameter n by one each time—hence enabling computation of the bound de Bruijn indices;
— recursively descends over \$\$ nodes;
— and in each case recursively moves the meta-binders Λ towards the bound meta-variables—both will be removed at the leaf nodes.

The key idea is to enable this by pattern matching against the Λ meta-binders (which we can do in Isabelle/HOL). Notice that ABS *nodes are, by construction, in one to one correspondence with the Λ nodes.*

For example, suppose that $C[v_i, v_j]$ is a HYBRID expression which contains the named variables v_i and v_j. Then for $i \neq j$, we would expect

$$\begin{array}{lcl} lbnd_0(\Lambda\, v_i.\, \mathsf{ABS}\ (C[v_i, v_j])) & = & \mathsf{ABS}\ (lbnd_1(\Lambda\, v_i.\, C[v_i, v_j])) \\ & \vdots & \\ & = & \mathsf{ABS}\ (C[lbnd_{n_1}(\Lambda\, v_i.\, v_i), lbnd_{n_2}(\Lambda\, v_i.\, v_j)]) \qquad (*) \end{array}$$

We are left with the definition of $lbnd_{n_1}(\Lambda\, v_i.\, v_j)$ and $lbnd_{n_2}(\Lambda\, v_i.\, v_i)$. Now the binding variable v_i originated with the ABS node tied to the binding $\Lambda\, v_i$. Hence we should define $lbnd_n(\Lambda\, v_i.\, v_i) \stackrel{\text{def}}{=} \mathsf{BND}\ n$. The (bound) v_j should be left alone (to be matched at some other time with a binder $\Lambda\, v_j$) so $lbnd_n(\Lambda\, v_i.\, v_j) \stackrel{\text{def}}{=} v_j$. In either case, Λ-binders and binding variables are finally removed. Hence the expression $(*)$ above should equal $\mathsf{ABS}\ (C[\mathsf{BND}\ n, v_j])$. Here is a concrete example, featuring only an \$\$ node:

$$\underline{lbnd_0(\Lambda\, v_2.\, v_8\ \$\$\ v_2)} = lbnd_0(\Lambda\, v_2.\, v_8)\ \$\$\ lbnd_0(\Lambda\, v_2.\, v_2) = \underline{v_8\ \$\$\ \mathsf{BND}\ 0}$$

We are in fact led to define $\mathsf{LAM}\, v_i.\, \xi$ as $\mathsf{ABS}\ (lbnd_0(\Lambda\, v_i.\, \xi))$ and hence

$$\begin{array}{l} \mathsf{LAM}\, v_8.\, (\mathsf{LAM}\, v_2.\, (v_8\ \$\$\ v_2)) \\ \quad = \mathsf{ABS}\ (lbnd_0(\Lambda\, v_8.\, \mathsf{ABS}\ (\underline{lbnd_0(\Lambda\, v_2.\, v_8\ \$\$\ v_2)}))) \\ \quad = \mathsf{ABS}\ (lbnd_0(\Lambda\, v_8.\, \mathsf{ABS}\ (\underline{v_8\ \$\$\ \mathsf{BND}\ 0}))) \\ \quad = \mathsf{ABS}\ (\mathsf{ABS}\ (lbnd_1(\Lambda\, v_8.\, v_8)\ \$\$\ lbnd_1(\Lambda\, v_8.\, (\mathsf{BND}\ 0)))) \\ \quad = \mathsf{ABS}\ (\mathsf{ABS}\ (\mathsf{BND}\ 1\ \$\$\ \mathsf{BND}\ 0)) \end{array}$$

In summary, each instance of the *lbnd* function descends recursively through its argument via higher-order matching of the meta-abstractions. At each ABS or \$\$ node, a meta-abstraction is "moved" towards the leaf nodes; and when descending over ABS nodes, a counter is increased. All leaf nodes are left unchanged, unless they are variables. In the case of variables, if the name of the meta-abstraction (eg $\Lambda\, v_2.$) which has been "moved" to the leaf (eg v_8) has a different name (eg $\Lambda\, v_2.\, v_8$), the leaf node remains unchanged (eg v_8). If it has the same name (eg $\Lambda\, v_8.\, v_8$) the node becomes $\mathsf{BND}\ n$ where n is the

counter—thus the original bound name becomes the correct de Bruijn bound index (eg v_8 becomes BND 1).

2.3. *Formalizing lbnd*

It is the formalisation and implementation of the function *lbnd* that is at the heart of the HYBRID system. In Section 3 we show how to develop a *model* of the full HYBRID system, and we will give a definition and existence proof of *lbnd*.

3. A Model of Core HYBRID

3.1. *Modelling* HYBRID *in a Logical Framework*

The specific goal of this section is to describe a mathematical model of HYBRID and then show that the model provides an adequate representation of the λ-calculus. Please recall the informal description of

$$\Theta : \text{ (object level) } \lambda\text{-expressions} \longrightarrow \text{HYBRID}$$

at the end of Section 2.1.

We must formally define Θ and prove it to be representationally adequate. We will take the λ-calculus to be the set $\mathcal{LE}/\sim_\alpha$. We now define exactly what HYBRID is for the purpose of this paper. Recall once again that there is an Isabelle/HOL theory called HYBRID. As such, we could consider working formally within higher order logic in this paper. This would lead to a considerable amount of additional technical detail: we would have to express the core content of this paper within higher order logic, and we feel that this would obscure the key ideas with yet a further layer of syntax. The important observation is that the functions and reasoning used in this paper are indeed implemented in Isabelle/HOL in a consistent manner, and so the proof of adequacy could be written down working within higher order logic if desired. However, since the intention of this paper is to focus on core features of our system, we work here with a *model of the core subset* of the system implemented in Isabelle/HOL. We shall still refer to this (model of the) core subset as HYBRID.

In order to realise this *model* we shall use the machinery of logical frameworks (see for example (Harper et al., 1993; Pfenning, 2003) and perhaps (Anderson and Pfenning, 2004; Harper and Pfenning, 2005)). More precisely, **we define our model of the** HYBRID **core subset as a theory in a logical framework**. *Note that we will work with a logical framework in which the types are just simple types generated from some ground types, and not the more general type systems of, for example, (Harper et al., 1993; Pfenning, 2003)).* Key to this is the following convention:

$$\begin{array}{rcl} \nu & :: & con \\ i & :: & var \\ j & :: & bnd \\ \mathsf{CON} & :: & con \Rightarrow exp \\ \mathsf{VAR} & :: & var \Rightarrow exp \\ \mathsf{BND} & :: & bnd \Rightarrow exp \\ \$\$ & :: & exp \Rightarrow exp \Rightarrow exp \\ \mathsf{ABS} & :: & exp \Rightarrow exp \end{array}$$

Fig. 3. Constructor Constants

$$\frac{\Gamma(v_k) = \sigma_1 \Rightarrow \sigma_2 \Rightarrow \ldots \sigma_n \Rightarrow \gamma \qquad \Gamma \vdash_{can} C_i :: \sigma_i \quad (0 \leq i \leq n)}{\Gamma \vdash_{can} v_k\, \vec{C} :: \gamma}\ \text{VAR}$$

$$\frac{\kappa :: \sigma_1 \Rightarrow \sigma_2 \Rightarrow \ldots \sigma_n \Rightarrow \gamma \qquad \Gamma \vdash_{can} C_i :: \sigma_i \quad (0 \leq i \leq n)}{\Gamma \vdash_{can} \kappa\, \vec{C} :: \gamma}\ \text{CST}$$

$$\frac{\Gamma, v_k :: \sigma \vdash_{can} C :: \sigma'}{\Gamma \vdash_{can} \Lambda\, v_k . C :: \sigma \Rightarrow \sigma'}\ \text{ABS}$$

Fig. 4. Inductive Definition of Canonical Forms

— The meta-variables of the logical framework play the rôle of Isabelle/HOL meta-variables of implemented HYBRID; and
— logical framework abstraction and application play the rôle of Isabelle/HOL meta-abstraction and meta-application respectively.

Let us now define the theory. The theory has ground types *con*, *var*, *bnd*, and *exp*, ranged over by γ. The (higher) types are given by $\sigma ::= \gamma \mid \sigma \Rightarrow \sigma$. We declare constructor constants in Figure 3 where i and j range over the natural numbers, and ν over a set of names for (object level) constants. We shall use κ to range over the constructor constants of the theory. The judgements are generated using the standard type assignment system of such a logical framework. More precisely, suppose that Γ is a **context**, that is, a finite partial function from the set of framework meta-variables to types. Then the type assignment system has judgements of the form $\Gamma \vdash e :: \sigma$. We omit the (usual) inductive definition. We define $\mathcal{LF}_\sigma(\Gamma) \stackrel{\text{def}}{=} \{e \mid \Gamma \vdash e :: \sigma\}$ and $\mathcal{LF} \stackrel{\text{def}}{=} \bigcup_{\sigma,\Gamma} \mathcal{LF}_\sigma(\Gamma)$. We give the inductive definition of canonical forms C in Figure 4. They are introduced using the judgements $\Gamma \vdash_{can} C :: \sigma$. We write $\mathcal{CLF}_\sigma(\Gamma) \stackrel{\text{def}}{=} \{C \mid \Gamma \vdash_{can} C :: \sigma\}$. Given a list of meta-variables $L = v_{k_1}, \ldots, v_{k_m}$, then we write Γ^L_{exp} for the context (partial function) $v_{k_1} :: exp, \ldots, v_{k_m} :: exp$. Note that it is standard to prove that $\mathcal{CLF}_\sigma(\Gamma) \subseteq \mathcal{LF}_\sigma(\Gamma)$.

Remark The ellipsis notation in Figure 4 is of course a short-hand for an infinite

$$\frac{}{\Gamma \vdash_{atm} v_k :: \sigma}\ \text{VAR'}\ [\Gamma(v_k) = \sigma] \qquad \frac{}{\Gamma \vdash_{atm} \kappa :: \sigma}\ \text{CST'}$$

$$\frac{\Gamma \vdash_{atm} A :: \gamma}{\Gamma \vdash_{can'} A :: \gamma}\ \text{INC}$$

$$\frac{\Gamma \vdash_{atm} A :: \sigma \Rightarrow \sigma' \qquad \Gamma \vdash_{can'} C :: \sigma}{\Gamma \vdash_{atm} A\, C :: \sigma}\ \text{APP}$$

$$\frac{\Gamma, v_k :: \sigma \vdash_{can'} C :: \sigma'}{\Gamma \vdash_{can'} \Lambda\, v_k.\, C :: \sigma \Rightarrow \sigma'}\ \text{ABS'}$$

Fig. 5. Inductive Definition of Canonical Forms via Atomic Forms

collection of formal inductive rules, one for each value of n ranging over $\mathbb{N}$. These rules are quite convenient for presenting the proofs in this paper, but one can also generate the set of canonical forms using a *finite* set of rules (Harper and Licata, 2007) as in Figure 5. One can prove (by rule induction) that

$$\mathcal{CLF}_{\sigma}(\Gamma) = \{C \mid \Gamma \vdash_{can'} C :: \sigma\}$$

We now give the formal definition of *lbnd*.

Proposition 3.1. *[Defining lbnd] For all $n \geq 0$ and lists L there is a unique function with the following source and target*

$$lbnd\ n : \mathcal{CLF}_{exp \Rightarrow exp}(\Gamma^L_{exp}) \to \mathcal{CLF}_{exp}(\Gamma^L_{exp})$$

which satisfies the following recursion equations

— $lbnd\ n\ (\Lambda\, v_k.\, \mathsf{CON}\ \nu) = \mathsf{CON}\ \nu$
— $lbnd\ n\ (\Lambda\, v_k.\, v_k) = \mathsf{BND}\ n$
— $lbnd\ n\ (\Lambda\, v_k.\, v_{k'}) = v_{k'}$ *where* $k \neq k'$
— $lbnd\ n\ (\Lambda\, v_k.\, \mathsf{VAR}\ i) = \mathsf{VAR}\ i$
— $lbnd\ n\ (\Lambda\, v_k.\, \mathsf{BND}\ j) = \mathsf{BND}\ j$
— $lbnd\ n\ (\Lambda\, v_k.\, C_1\ \$\$\ C_2) = (lbnd\ n\ (\Lambda\, v_k.\, C_1))\ \$\$\ (lbnd\ n\ (\Lambda\, v_k.\, C_2))$
— $lbnd\ n\ (\Lambda\, v_k.\, \mathsf{ABS}\ C) = \mathsf{ABS}\ (lbnd\ (n+1)\ (\Lambda\, v_k.\, C))$

Proof. Informally, the existence and uniqueness of each *lbnd* n is straightforward; the equations above can be regarded as a recursive definition arising from an inductively defined set, namely the canonical forms (in context) of Figure 4. However, since it is not completely immediate what are the exact inductive rules that specify the source of the function we give a detailed proof of existence (see Section A of the Appendix).

Let I be the inductively defined set of all canonical forms in context specified in Figure 4. Let $S \stackrel{\text{def}}{=} \mathcal{CLF}_{exp \Rightarrow exp}(\Gamma^L_{exp})$. Let $W \stackrel{\text{def}}{=} \mathcal{CLF}_{exp}(\Gamma^L_{exp})$. Let $\mathbb{N} \Rightarrow W$ be the set

of functions from $\mathbb{N}$ to W. We will give an existence proof of a function $F\colon S \to (\mathbb{N} \Rightarrow W)$ and define for any $n \in \mathbb{N}$

$$\mathit{lbnd}\ n\ C \stackrel{\text{def}}{=} F(C)(n)$$

The recursion equations will follow trivially from the action of F.

Let $I_0 \stackrel{\text{def}}{=} \varnothing$ and let I_{h+1} be those elements of I with deduction trees of height less than or equal to $h+1$ where $h \in \mathbb{N}$. It is standard that $I = \bigcup_{h\in\mathbb{N}} I_h$ and that the I_h form an increasing sequence of sets ordered by inclusion.

Let $S_0 \stackrel{\text{def}}{=} \varnothing$, $S_1 \stackrel{\text{def}}{=} \varnothing$ and for $h \geq 1$ define

$$S_{h+1} \stackrel{\text{def}}{=} \{C \mid \Gamma^L_{\mathit{exp}} \vdash_{\mathit{can}} C :: \mathit{exp} \Rightarrow \mathit{exp} \in I_{h+1} - I_h\}$$

It is easy to see that $S = \bigcup_{h\in\mathbb{N}} S_h$, and moreover this is a union of pairwise disjoint sets. We will now construct the function $F : S \to (\mathbb{N} \Rightarrow W)$ by defining a sequence of functions $F_h : S_h \to (\mathbb{N} \Rightarrow W)$ and taking $F \stackrel{\text{def}}{=} \bigcup_{h\in\mathbb{N}} F_h$, a union of functions with pairwise disjoint sources (and hence trivially a function).

First we consider the form of $C \in S_{h+1}$ for $h \geq 1$ (recall Figure 4). This element cannot be generated with final rule VAR or CST as no ground type γ is the higher type $\mathit{exp} \Rightarrow \mathit{exp}$. Hence the final rule is ABS and we must have $\Gamma^L_{\mathit{exp}}, v_k :: \mathit{exp} \vdash_{\mathit{can}} C' :: \mathit{exp}$ where $C \equiv \Lambda\, v_k.\, C'$. Note also that $C' \in I_h - I_{h-1}$.

$\boxed{F_0 : S_0 \to (\mathbb{N} \Rightarrow W)}$ Take F_0 to be the empty function.

$\boxed{F_1 : S_1 \to (\mathbb{N} \Rightarrow W)}$ Take F_1 to be the empty function.

$\boxed{F_2 : S_2 \to (\mathbb{N} \Rightarrow W)}$ If $\Lambda\, v_k.\, C' \in S_2$ then $C' \in I_1 - I_0 = I_1$ and C' also has type exp. $C' \in I_1$ implies it can only be derived using VAR or CST with empty hypothesis set; and the type forces the rule to be VAR. Thus $C' \equiv v_{k'}$, making $C \equiv \Lambda\, v_k.\, v_{k'}$. Hence F_2 is fully specified for any $n \in \mathbb{N}$ by

— $F_2(\Lambda\, v_k.\, v_k)(n) \stackrel{\text{def}}{=} \mathsf{BND}\ n$
— $F_2(\Lambda\, v_k.\, v_{k'})(n) \stackrel{\text{def}}{=} v_{k'}$ where $k \neq k'$

$\boxed{F_3 : S_3 \to (\mathbb{N} \Rightarrow W)}$ If $\Lambda\, v_k.\, C' \in S_3$ then $C' \in I_2 - I_1$ and C' also has type exp. By examining the rules we see that C' must be either $\mathsf{CON}\ \nu$, $\mathsf{VAR}\ i$, $\mathsf{BND}\ j$, or $v_{k'}$ \$\$ $v_{k''}$, or $\mathsf{ABS}\ v_{k'''}$. Hence F_2 is fully specified for any $n \in \mathbb{N}$ by

— $F_3(\Lambda\, v_k.\, \mathsf{CON}\ \nu)(n) \stackrel{\text{def}}{=} \mathsf{CON}\ \nu$
— $F_3(\Lambda\, v_k.\, \mathsf{VAR}\ i)(n) \stackrel{\text{def}}{=} \mathsf{VAR}\ i$
— $F_3(\Lambda\, v_k.\, \mathsf{BND}\ j)(n) \stackrel{\text{def}}{=} \mathsf{BND}\ j$
— $F_3(\Lambda\, v_k.\, v_{k'}\ \$\$\ v_{k''})(n) \stackrel{\text{def}}{=} (F_2(\Lambda\, v_k.\, v_{k'})(n))\ \$\$\ (F_2(\Lambda\, v_k.\, v_{k''})(n))$
— $F_3(\Lambda\, v_k.\, \mathsf{ABS}\ v_{k'''})(n) \stackrel{\text{def}}{=} \mathsf{ABS}\ (F_2(\Lambda\, v_k.\, v_{k'''})(n+1))$

$\boxed{F_h : S_h \to (\mathbb{N} \Rightarrow W) \text{ where } h \geq 4}$ We prove by induction that for all $h \geq 4$ there is such a function F_h which is fully specified by the clauses

— $F_h(\Lambda\, v_k.\, C_1\ \$\$\ C_2)(n) \stackrel{\text{def}}{=} ((\bigcup_0^{h-1} F_r)(\Lambda\, v_k.\, C_1)(n))\ \$\$\ ((\bigcup_0^{h-1} F_r)(\Lambda\, v_k.\, C_2)(n))$
— $F_h(\Lambda\, v_k.\, \mathsf{ABS}\ C_3)(n) \stackrel{\text{def}}{=} \mathsf{ABS}\ ((\bigcup_0^{h-1} F_r)(\Lambda\, v_k.\, C_3)(n+1))$

together with the functions F_h defined above for $h = 0, 1, 2, 3$. If $h = 0$ the proposition is vacuous. Suppose that it holds for all numbers r less than or equal to an arbitrary h. We prove it holds for $h+1$. So now assume that $h+1 \geq 4$. Suppose that $\Lambda\, v_k.\, C' \in S_{h+1}$ where $C' \in I_h - I_{h-1}$ and of course $\Gamma^L_{exp}, v_k :: exp \vdash_{can} C' :: exp$. Now $h \geq 3$ and by examining the rules used to generate C' we see that either $C' \equiv C_1\ \$\$\ C_2$ with each $C_r \in I_{h-1}$ and of type *exp*, or $C' \equiv \mathsf{ABS}\, C_3$ with $C_3 \in I_{h-1}$ and of type *exp*. Hence $\Lambda\, v_k.\, C_s \in \bigcup_0^h S_r$ for $s = 1, 2, 3$. By induction, each of the functions F_r exist for $0 \leq r \leq h$. Hence F_{h+1} is indeed completely specified by the given clauses.

Defining $F \stackrel{\text{def}}{=} \bigcup_{h\in\mathbb{N}} F_h$ which exists given that the F_h all exist, it is virtually immediate that the given definition of *lbnd* satisfies the stated recursion conditions. Uniqueness is an easy exercise.

To finish the proof we need to verify that *lbnd* preserves α-equivalence (defined over all expressions). Now suppose that $e \sim_\alpha e'$ and moreover $e, e' \in \mathcal{CLF}_{exp\Rightarrow exp}(\Gamma^L_{exp})$. By inspecting the rules defining α-equivalence and the rules of Figure 4 we see that $e \equiv \Lambda\, v_k.\, C$ and the only applicable rule of α-equivalence is the re-naming axiom. So we show that whenever $\Gamma^L_{exp} \vdash_{can} \Lambda\, v_k.\, C :: exp \Rightarrow exp$, we have

$$lbnd\ n\ (\Lambda\, v_k.\, C) = lbnd\ n\ (\Lambda\, v_{k'}.\, C[v_{k'}/v_k])$$

provided that $v_{k'}$ does not occur free in C.

Suppose that the final rule used in the derivation is ABS. Of course

$$\Gamma^L_{exp}, v_k :: exp \vdash_{can} C :: exp \qquad (*)$$

We must now consider all the possible ways this judgement could have been derived. The statement $(*)$ cannot have been derived using ABS, as *exp* is not a higher type. If $(*)$ was derived using VAR then it is easy to see that $C \equiv v_{k''}$ for some k''. If $k \neq k''$ we have

$$lbnd\ n\ (\Lambda\, v_k.\, v_{k''}) \stackrel{\text{def}}{=} v_{k''} = lbnd\ n\ (\Lambda\, v_{k'}.\, v_{k''}) = lbnd\ n\ (\Lambda\, v_{k'}.\, v_{k''}[v_{k'}/v_k])$$

where the assumption that $v_{k'}$ is not free implies that $k' \neq k''$. In the case $k = k''$ both expressions equal $\mathsf{BND}\ n$. We omit all routine details in the case $(*)$ was derived using CST.

□

4. The Adequacy of de Bruijn Expressions for λ-Expressions

In this section we demonstrate in a very precise way that de Bruijn expressions provide an *adequate* representation of the expressions of the λ-calculus. In establishing HYBRID adequacy in Section 5, we will make extensive use of the notation and functions which we use in our proof of de Bruijn adequacy, as well as the result itself.

Recall that in Section 1.4 we set up our syntax for λ-expressions and de Bruijn expressions. In Section 4.1 we state de Bruijn adequacy formally. In Section 4.2 we give an outline of a proof, listing the key propositions—the formal statements of the propositions,

together with their proofs, appear in the Appendix Section C. In Section 4.3 we define the families of functions used to establish the required bijection—the proofs of existence also appear in the Appendix Section C. In Section 4.4 we prove the adequacy result, Theorem 4.1.

4.1. *The Representational Adequacy Theorem*

We wish to prove the following theorem which will be used in Section 5 to prove an adequacy result for HYBRID.

Theorem 4.1. *[de Bruijn Representational Adequacy] There is a function*

$$\theta : \mathcal{LE}/\!\sim_\alpha \to \mathcal{PDB} \subseteq \mathcal{DB}$$

which is ***representationally adequate****, that is to say θ is a compositional isomorphism $\mathcal{LE}/\!\sim_\alpha \cong \mathcal{PDB}$. Equivalently*

B *θ is bijective; and*
CH *θ is a compositional homomorphism*

$$\theta([E]_\alpha[[E']_\alpha/v_k]) = \theta([E]_\alpha)[\theta([E']_\alpha)/\mathsf{var}(k)]$$

Note that we appeal to Proposition 1.1 and Proposition 1.2 in stating ***CH****.*

4.2. *A Proof Outline*

We shall give an informal outline of the proof. First, though, we need some more notation.

A **list** L is one whose elements (if any) are object level variables v_k. We write ϵ for the **empty list**, and write v_k, L and L, L' for **cons** and **concatenation**. Thus a typical non-empty list is $v_{10}, v_{70}, v_{70}, v_6, v_2, v_0, v_{10}$. If a list L is non-empty, the head has **position** 0, and the last element **position** $|L| - 1$ where $|L|$ is the **length** of the list. Thus v_{70} occurs at positions 1 and 2 in the recent example. If L is non-empty and v_k occurs in it, we write $v_k \in L$. Suppose also that the *first* occurrence is at position p. Then we write $pos\, v_k\, L$ for p. If $v_k \notin L$ then $pos\, v_k\, L$ is undefined. We write $elt\, p\, L$ for the variable v_k at position p if such exists; otherwise $elt\, p\, L$ is undefined. We say that L is **ordered** if L is a list, has no repeated elements, and the indices occur in decreasing order. Thus a typical non-empty ordered list is $v_{100}, v_7, v_6, v_2, v_0$. If S is a set of variables, we will use informal notation such as $S \cap L$ to mean the intersection of S and the set of variables in L.

The compositional isomorphism θ will be built out of a certain pair of L-indexed families of functions

$$[\![-]\!]_L : \mathcal{LE} \rightleftarrows \mathcal{DB}(|L|) : (\!|-|\!)_L$$

We will also make use of these functions in Section 5. Here we give informal descriptions of them before the formal definitions in the next section.

The functions $[\![-]\!]_L$, one for each L, are defined by recursion over the structure of λ-expressions (readers may care to glance ahead at Proposition 4.2). The list L should be thought of as naming certain binding variables in a λ-expression. Roughly speaking $[\![-]\!]_L$ will descend through the constructors of an expression, replacing λ-application nodes by de Bruijn application nodes, and replacing λ-abstraction nodes by de Bruijn abstraction nodes. When descending through a λ-abstraction node with binder v_i, the list L is updated to v_i, L. At variable nodes (leaves), if the leaf is in the list of binding variables it becomes a de Bruijn bound variable index, otherwise it becomes a de Bruijn free variable index. The bound variable index is determined using the position of the variable in L. Note that $[\![-]\!]_L$ delivers a de Bruijn expression at level $|L|$.

In order to show that θ is an isomorphism, we show that each function $(\!|-|\!)_L$ is, roughly speaking, an inverse to $[\![-]\!]_L$. The existence of the family of functions $(\!|-|\!)_L$ is proved in Proposition 4.3. Once again, each $(\!|-|\!)_L$ is defined by recursion, here over de Bruijn expressions. When recursing over a de Bruijn abstraction node, a binding variable v_I is created, and added to the list L of binding variables. The value of index I is chosen to be larger than any index in L and any index in the de Bruijn expression that $(\!|-|\!)_L$ is recursing over.

In Proposition C.3 we show that for any L, if $E \sim_\alpha E'$ then $[\![E]\!]_L = [\![E']\!]_L$, establishing that $[\![-]\!]_L$ preserves α-equivalent expressions. We will be able to use this fact to define θ, whose source is $\mathcal{LE}/\!\sim_\alpha$, from the family $[\![-]\!]_L$ whose source is $\mathcal{LE}$.

In Proposition C.4 we show that

$$[\![(\!|D|\!)_L]\!]_L = D$$

thus demonstrating that $[\![-]\!]_L$ is an inverse for $(\!|-|\!)_L$.

In Proposition C.5 we show that

$$(\!|[\![E]\!]_L|\!)_{L'} \sim_\alpha E[L'/L]$$

and hence that up to α-equivalence $(\!|-|\!)_L$ is an inverse for $[\![-]\!]_L$. Note that the strength of the proposition, which involves $L \neq L'$, is required during inductive stages of its proof. Only once proved may we consider the case when $L = L'$.

In Proposition C.6 we show that the functions $[\![-]\!]_L$ are indeed compositional homomorphisms.

We can then prove Theorem 4.1. The idea, roughly speaking, is that θ is the function $[\![-]\!]_\epsilon$ and that it has an inverse ϕ given by $(\!|-|\!)_\epsilon$; when θ and ϕ are first applied to expressions, the list ϵ of binding variables is empty. The propositions have established that θ is well defined on α-equivalence classes, and that θ and ϕ yield an isomorphism

since they are inverses for each other. Finally θ is a compositional homomorphism since $[\![-]\!]_\epsilon$ is.

4.3. *Setting up the Bijection*

We state the existence of the families of functions required to establish the bijection. The proofs are in Appendix Section C. We keep the definition of the functions in the main text since other proofs that appear in the paper make direct use of the definitions.

Proposition 4.2. *[Defining $[\![-]\!]_L$] For any L, there is a function*

$$[\![-]\!]_L : \mathcal{LE} \to \mathcal{DB}(|L|)$$

satisfying the recursion equations below; in particular, $[\![-]\!]_\epsilon : \mathcal{LE} \to \mathcal{PDB}$.

— $[\![\nu]\!]_L = \mathsf{con}(\nu)$
— *On variables we have*

$$[\![v_i]\!]_L \quad = \quad \begin{cases} \mathsf{bnd}(pos\, v_i\, L) \text{ if } v_i \in L \\ \mathsf{var}(i) \text{ if } v_i \notin L \end{cases}$$

where $pos\, v_i\, L$ is the position of v_i in L.
— $[\![E_1\ E_2]\!]_L = [\![E_1]\!]_L\ \$\ [\![E_2]\!]_L$
— $[\![\lambda\, v_i.\, E]\!]_L = \mathsf{abs}([\![E]\!]_{v_i, L})$

Proposition 4.3. *[Defining $(\!|-|\!)_L$] For any ordered L, there is a function*

$$(\!|-|\!)_L : \mathcal{DB}(|L|) \to \mathcal{LE}$$

satisfying the recursion equations below; in particular, $(\!|-|\!)_\epsilon : \mathcal{PDB} \to \mathcal{LE}$.

— $(\!|\mathsf{con}(\nu)|\!)_L = \nu$
— $(\!|\mathsf{var}(i)|\!)_L = v_i$
— $(\!|\mathsf{bnd}(j)|\!)_L = elt\, j\, L$ *where $elt\, j\, L$ is the jth element of L*
— $(\!|D_1\ \$\ D_2|\!)_L = (\!|D_1|\!)_L\ (\!|D_2|\!)_L$
— $(\!|\mathsf{abs}(D)|\!)_L = \lambda\, v_{M+1}.\, (\!|D|\!)_{v_{M+1}, L}$ *where $M = \mathsf{Max}(D; L)$ with*

$$\mathsf{Max}(D; L) \stackrel{\text{def}}{=} \mathsf{Max}\ \{i \mid \mathsf{var}(i) \in D\} \bigcup \underbrace{\{j \mid \mathsf{head}(L) = v_j\}}_{\varnothing \text{ if } L \text{ empty}}$$

We take $\mathsf{Max}\ \varnothing \stackrel{\text{def}}{=} 0$. *Informally* $\mathsf{Max}(D; L)$ *denotes the maximum of the free indices which occur in D and the indices of L.*

4.4. *Proving de Bruijn Adequacy*

We can now prove Theorem 4.1:

Proof. Please consider revisiting Section 4.2 and using the Appendix if required.

B Recall that from Proposition 4.2 [*Defining* $[\![-]\!]_L$] and Proposition 4.3 [*Defining* $(\!|-|\!)_L$] we have the existence of a pair of families of functions

$$[\![-]\!]_L : \mathcal{LE} \rightleftarrows \mathcal{DB}(|L|) : (\!|-|\!)_L$$

Consider the following diagram, with q the surjective quotient map, and ϵ the empty list where of course $\mathcal{PDB} \stackrel{\text{def}}{=} \mathcal{DB}(|\epsilon|) = \mathcal{DB}(0)$

$$\mathcal{LE}/\!\sim_\alpha \xleftarrow{\quad q \quad} \mathcal{LE} \underset{(\!|-|\!)_\epsilon}{\overset{[\![-]\!]_\epsilon}{\rightleftarrows}} \mathcal{PDB}$$

We may define

$$\theta : \mathcal{LE}/\!\sim_\alpha \rightleftarrows \mathcal{PDB} : \phi$$

by setting $\theta([E]_\alpha) \stackrel{\text{def}}{=} [\![E]\!]_\epsilon$ for any $E \in \mathcal{LE}$ and $\phi \stackrel{\text{def}}{=} q \circ (\!|-|\!)_\epsilon$. Note that Proposition C.3 [$[\![-]\!]_L$ *preserves* α*-equivalence*] shows us that $[\![-]\!]_\epsilon$ is equal on α-equivalent expressions, and so the definition of θ is a good one. We then have

$$(\theta \circ \phi)(D) = [\![(\!|D|\!)_\epsilon]\!]_\epsilon = D$$

using Proposition C.4 [*The Identity* $[\![-]\!]_L \circ (\!|-|\!)_L$], and

$$(\phi \circ \theta)[E]_\alpha = [(\!|[\![E]\!]_\epsilon|\!)_\epsilon]_\alpha = [E]_\alpha$$

using Proposition C.5 [*The Identity* $(\!|-|\!)_L \circ [\![-]\!]_L$].

CHThe fact that θ is a compositional homomorphism is immediate from Proposition C.6 [$[\![-]\!]_L$ *Compositional Homomorphism*] and the definition of θ.

□

5. The Adequacy of HYBRID for λ-Expressions

5.1. *The Representational Adequacy Theorem*

Before we can state the adequacy theorem, we will need a notion of substitution for HYBRID. The existence of the substitution function is given in the following lemma.

Lemma 5.1. *There is a function*

$$\mathcal{CLF}_{exp}(\Gamma^L_{exp}) \times \mathcal{CLF}_{exp}(\Gamma^L_{exp}) \times \mathbb{N} \to \mathcal{CLF}_{exp}(\Gamma^L_{exp})$$

denoted by $(C, C', k) \mapsto C[C'/\mathsf{VAR}\ k]$*, which, informally, maps* (C, C', k) *to the expression* C *in which all occurrences of* VAR k *are replaced by* C'*.*

Proof. The definition of the function, and the proof, are omitted. Note that the set $\mathcal{CLF}_{exp}(\Gamma^L_{exp})$ does not involve expressions which bind variables, and so the definition of the function is entirely straightforward since there is no need for re-naming. □

Theorem 5.2. *[HYBRID Representational Adequacy] There is a well-defined function*

$$\Theta_\epsilon : \mathcal{LE}/\!\sim_\alpha \to \Theta_\epsilon\,(\mathcal{LE}/\!\sim_\alpha) \subseteq \mathcal{CLF}_{exp}(\epsilon)$$

arising from the family of unique well-defined functions

$$\Theta_L : \mathcal{LE}/\!\sim_\alpha \to \mathcal{CLF}_{exp}(\Gamma^L_{exp})$$

satisfying the recursion equations

— $\Theta_L\,([\nu]_\alpha) \stackrel{\text{def}}{=} \mathsf{CON}\ \nu$

$$\Theta_L\,([v_i]_\alpha) \stackrel{\text{def}}{=} \begin{cases} v_i \ \textit{if}\ v_i \in L \\ \mathsf{VAR}\ i \ \textit{if}\ v_i \notin L \end{cases}$$

— $\Theta_L\,([E_1\ E_2]_\alpha) \stackrel{\text{def}}{=} (\Theta_L\,[E_1]_\alpha)\ \$\$\ (\Theta_L\,[E_2]_\alpha)$

— $\Theta_L\,([\lambda\, v_i.\, E]_\alpha) \stackrel{\text{def}}{=} \mathsf{LAM}\, v_i.\, \Theta_{v_i,L}\,([E]_\alpha)$ *where we write* $\mathsf{LAM}\, v_i.\, \xi$ *as an abbreviation for* $\mathsf{ABS}\ (\mathit{lbnd}\ 0\ (\Lambda\, v_i.\, \xi))$.

Θ_ϵ *is* ***representationally adequate****, that is to say* Θ_ϵ *is a compositional isomorphism* $\mathcal{LE}/\!\sim_\alpha \cong \Theta_\epsilon\,(\mathcal{LE}/\!\sim_\alpha)$*. Equivalently:*

B *it is bijective (onto its image); and*
CH *it is a compositional homomorphism which means that*

$$\Theta_\epsilon\,([E]_\alpha[[E']_\alpha/v_k]) = \Theta_\epsilon\,([E]_\alpha)[\Theta_\epsilon\,([E']_\alpha)/\mathsf{VAR}\ k]$$

with the right hand expression existing from Lemma 5.1.

Fig. 6. HYBRID Representational Adequacy Theorem

The key theorem of this paper is stated in Figure 6.

We suggest the reader experiments with computing the adequacy function Θ: Take $E_O \stackrel{\text{def}}{=} \lambda\, v_8.\, \lambda\, v_2.\, v_8\ v_3$ and then compute $\Theta_\epsilon\,[E_O]_\alpha$ which is the formal definition of the HYBRID encoding of E_O. The answer should be the expression E_H (below) where the abbreviations above have been fully expanded. The function Θ_ϵ works recursively. λ-binder nodes are replaced by instances of LAM (recall that $\mathsf{LAM}\, v_i.\, \xi$ is an abbreviation for $\mathsf{ABS}\ (\mathit{lbnd}\ 0\ (\Lambda\, v_i.\, \xi))$). Application nodes are replaced by \$\$. Θ_L recursively collects the names of the λ-binders in L, and when it reaches leaf node variables it checks to see if the leaf is in scope of a binder or not. If it is (eg v_8), the leaf remains unchanged; if not, (eg v_3) the leaf v_i becomes $\mathsf{VAR}\ i$. Thus

$$E_H = \mathsf{ABS}\ (\mathit{lbnd}\ 0\ \Lambda\, v_8.\, (\mathsf{ABS}\ (\mathit{lbnd}\ 0\ \Lambda\, v_2.\, (v_8\ \$\$\ \mathsf{VAR}\ 3))))$$

and the reader can also check that this equals

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

5.2. *Factoring* HYBRID *Adequacy*

Key to proving Theorem 5.2 is the existence of a function *inst*. This provides an explicit connection between the function Θ_ϵ, and the function θ of Theorem 4.1. In fact, as we shall see, informally $\Theta_\epsilon = inst \circ \theta$. We already have an adequacy result for θ; and we will be able to obtain the same for *inst*. We proceed like this because Θ maps λ-expressions to HYBRID de Bruijn expressions in a very intensional and indirect way (utilising calls to the function *lbnd* at each node of the λ-expression) and it is difficult to prove adequacy of this function in a direct way. It is far easier to prove properties, such as injectivity, of *inst*. Let us introduce *inst* using examples.

Recall (where an overline connects related abstraction nodes, binding and bound variables) that

$$\Theta_\epsilon\,[\lambda\, v_8.\, \lambda\, v_2.\, v_8\ v_2]_\alpha = \overline{\mathsf{ABS}}\ (\overline{lbind\ 0\ \Lambda\ v_8.}\ \overline{\overline{\mathsf{ABS}}}\ (\overline{\overline{lbind\ 0\ \Lambda\ v_2.}}\ \overline{v_8}\ \$\$\ \overline{\overline{v_2}})) \quad (*)$$

and that

$$\Theta_{\overline{v_8}}\,[\lambda\, v_2.\, v_8\ v_2]_\alpha = \overline{\overline{\mathsf{ABS}}}\ (\overline{\overline{lbind\ 0\ \Lambda\ v_2.}}\ \overline{v_8}\ \$\$\ \overline{\overline{v_2}}) \qquad (**)$$

Although the function *lbnd* has a complex intensional definition, the "key" information specified in $(**)$ can be simplified to the data

$$\begin{array}{rcl} ([v_8], \hat{}\mathsf{ABS}\ (v_8\ \$\$\ v_2), 0) & = & ([v_8], \mathsf{ABS}\,\hat{}(v_8\ \$\$\ v_2), 1) \\ & = & ([v_8], \mathsf{ABS}\ (\hat{}v_8\ \$\$\ \hat{}v_2), 1) \end{array}$$

where an indicator $\hat{}$ descends recursively through the expression, and a counter records that there is one ABS node between the variables and the root node. Note that the list of variables $[v_8]$ records the "once bound $(*)$ but now free $(**)$" variables, that is, the "dangling" variables. This is sufficient information to

— replace v_2 with its de Bruijn index—we counted just one abstraction, and v_2 is *not* in the list of "dangling" variables; and

— leave v_8 alone—entries in the list of "dangling" variables remain unchanged.

Hence the expression $([v_8], \mathsf{ABS}\ (\hat{}v_8\ \$\$\ \hat{}v_2), 1)$ "equals" $\mathsf{ABS}\ (v_8\ \$\$\ \mathsf{BND}\ 0)$.

This leads us to formulate *inst*. This function takes as inputs a counter, a list of n variables, and any de Bruijn expression. It descends recursively through the expression, counting abstraction nodes. When any leaf node bound index j is reached it can use the counter to determine if the the index is dangling or not. If it is, the index is replaced by

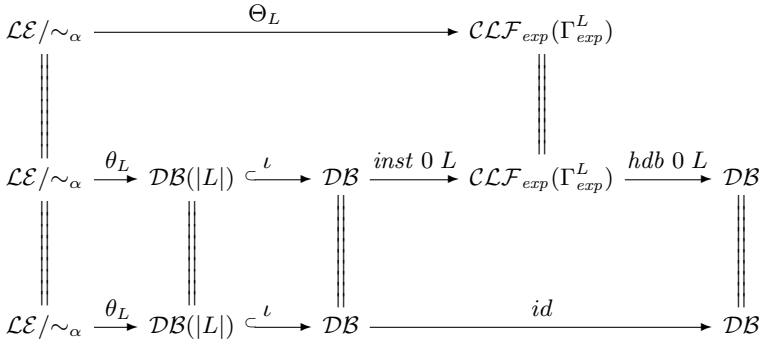

Fig. 7. Functions Used in the Adequacy Proof

the $(j-n)$th variable; if it is not, it is left alone. Thus we would have

$$\begin{array}{lll} inst\ 0\ v_8\ (\underbrace{\mathsf{ABS}\ (\mathsf{BND}\ 1\ \$\$\ \mathsf{BND}\ 0)}_{[\![\lambda\, v_2.\, v_8\ v_2]\!]_{v_8}}) & = & \\ & \vdots & \\ & = & \mathsf{ABS}\ (v_8\ \$\$\ \mathsf{BND}\ 0) \\ & = & \Theta_{v_8}\ [\lambda\, v_2.\, v_8\ v_2]_\alpha \end{array}$$

Putting all of this together suggests that we can prove $\Theta_L = (inst\ 0\ L) \circ [\![-]\!]_L$, and in fact subject to tidying up the technical details, this is what we shall do. We can then prove representational adequacy of each function in the function composition.

5.3. *A Proof Outline*

We have now assembled enough infrastructure for our proof of HYBRID adequacy. We first give an outline of the top-level structure of the adequacy proof. The proof of the theorem uses three key propositions, described below. These three propositions together with Theorem 4.1 allow us to prove HYBRID adequacy, that is, Theorem 5.2. Consider the diagram in Figure 7.

In Proposition 5.6 we show that the function Θ_L exists (and hence that Θ_ϵ exists). We do this by proving that the graph of Θ_L is equal to the graph of the composition $(inst\ 0\ L) \circ \iota \circ \theta_L$ of three well defined functions. Recall that $[\![-]\!]_L$ is defined on page 17. The function θ_L is defined by $[\![-]\!]_L \circ q^{-1}$ where $q\colon \mathcal{LE} \to \mathcal{LE}/\sim_\alpha$ is the quotient function. In Lemma 5.3 we prove the existence of the function *inst*. Two other technical lemmas 5.4 and 5.5 are also required to establish that $\Theta_L = (inst\ 0\ L) \circ \iota \circ \theta_L$.

In Proposition 5.8 we show that the function *hdb*, which is shown to exist in Lemma 5.7,

is a left inverse for the function *inst*, in the formal sense that $(hdb\ 0\ L) \circ (inst\ 0\ L) = id_{\mathcal{DB}}$. Hence *inst* is injective.

In Proposition 5.10 we see that *inst* is a compositional homomorphism. The technical Lemma 5.9 is used in the proof of Proposition 5.10.

The HYBRID adequacy proof follows from this: θ_ϵ is representationally adequate by Theorem 4.1 [*de Bruijn Representational Adequacy*] since it is equal to θ from the theorem. And since $\Theta_\epsilon = (inst\ 0\ \epsilon) \circ \iota \circ \theta_\epsilon$, all we need to know is that $inst\ 0\ \epsilon$ is injective and compositional, which follows from Proposition 5.8 and Proposition 5.10.

5.4. *Infrastructure for the Proof Outline*

We give a formal definition of the function *inst*. Recall that its input includes a list of variables and a de Bruijn expression. The output is the de Bruijn expression in which dangling bound indices are replaced by variables from the list. As such, each output is a canonical form which can be generated in a context whose domain of definition is the (set of) variables from the list, each with type *exp*.

Lemma 5.3. *[Defining inst] For each $n \geq 0$ and list L, there is a unique function with the following source and target*

$$inst\ n\ L\ : \mathcal{DB} \to \mathcal{CLF}_{exp}(\Gamma^L_{exp})$$

which satisfies the recursion equations

— $inst\ n\ L\ \mathsf{con}(\nu) = \mathsf{CON}\ \nu$
— $inst\ n\ L\ \mathsf{var}(i) = \mathsf{VAR}\ i$
—

$$inst\ n\ L\ \mathsf{bnd}(j) \quad = \quad \begin{cases} elt\ (j-n)\ L & \text{if } n \leq j \text{ and } j-n < |L| \\ \mathsf{BND}\ j & \text{otherwise, that is } n > j \text{ or } j-n \geq |L| \end{cases}$$

(recall, page 15, that $elt\ p\ L$ computes pth element in L)
— $inst\ n\ L\ (D_1\ \$\ D_2) = (inst\ n\ L\ D_1)\ \$\$\ (inst\ n\ L\ D_2)$
— $inst\ n\ L\ \mathsf{abs}(D) = \mathsf{ABS}\ (inst\ (n+1)\ L\ D)$

Proof. $\mathcal{DB}$ is an inductively defined set, so by standard results (see Appendix Section A) the unique function $inst\ n\ L$ exists by appealing to structural recursion over $\mathcal{DB}$, provided that the expressions on the right hand sides of the equations are elements of $\mathcal{CLF}_{exp}(\Gamma^L_{exp})$ on the (inductive) assumption that recursive calls of *inst* are elements of $\mathcal{CLF}_{exp}(\Gamma^L_{exp})$.

Note that $elt\ (j-n)\ L$ is defined since $0 \leq j-n < |L|$ and it is clear from VAR (see Figure 4) that $\Gamma^L_{exp} \vdash_{can} elt\ (j-n)\ L :: exp$. All the remaining left hand sides are trivially in $\mathcal{CLF}_{exp}(\Gamma^L_{exp})$.

□

The next significant step is to prove that $\Theta_L = (\mathit{inst}\ 0\ L) \circ \theta_L$. In order to do this we will need two lemmas which are rather technical in nature. Before stating them formally and giving proofs, we will give some informal motivation and illustrations.

The reader should show that

$$\mathit{inst}\ 1\ \epsilon\ (\mathsf{abs}((\mathsf{bnd}(1)\ \$\ \mathsf{bnd}(0))))$$

and

$$\mathit{lbnd}\ 0\ (\Lambda\, v_8.\, (\mathit{inst}\ 0\ v_8\ (\mathsf{abs}((\mathsf{bnd}(1)\ \$\ \mathsf{bnd}(0))))))$$

are equal. In fact Lemma 5.4 (to appear) is central to our main proof. It shows how the functions *lbnd* and *inst* interact by stating that such equations hold in general. We motivate it by giving further informal examples and analysis.

Suppose that $D[\mathsf{bnd}(j)]$ is a de Bruijn expression in which there are n abs nodes between the bound index j and the root. Consider $\mathit{inst}\ 1\ L\ D[\mathsf{bnd}(j)]$. The output of this function is essentially $D[\mathsf{bnd}(j)]$ but with

— any $\mathsf{bnd}(j)$ replaced by $\mathit{elem}(j-(n+1))L$ when

$$0 \leq j-(n+1) \leq |L| \qquad \equiv \qquad n+1 \leq j \leq |L|+(n+1)$$

Now consider $\mathit{inst}\ 0\ (v_k, L)\ D[\mathsf{bnd}(j)]$. The output of this function is essentially $D[\mathsf{bnd}(j)]$ but with

— any $\mathsf{bnd}(j)$ replaced by $\mathit{elem}(j-n)L$ when

$$0 \leq j-n \leq |v_k, L| \qquad \equiv \qquad n \leq j \leq (1+|L|)+n$$

Thus the second output is the same as the first, but in addition any $\mathsf{bnd}(n)$ gets replaced by v_k. Hence one can see that

$$(\mathit{inst}\ 1\ L\ D[\mathsf{bnd}(j)])[v_k/\mathsf{bnd}(n)] = \mathit{inst}\ 0\ (v_k, L)\ D[\mathsf{bnd}(j)] \qquad (*)$$

and one may conjecture that

$$\begin{aligned}\mathit{lbnd}\ 0\ (\Lambda\, v_k.\, \mathit{inst}\ 0\ (v_k, L)\ D[\mathsf{bnd}(j)]) \\ &= (\mathit{inst}\ 0\ (v_k, L)\ D[\mathsf{bnd}(j)])[\mathsf{bnd}(n)/v_k] \\ &= (\mathit{inst}\ 1\ L\ D[\mathsf{bnd}(j)])[v_k/\mathsf{bnd}(n)][\mathsf{bnd}(n)/v_k] \\ &= \mathit{inst}\ 1\ L\ D[\mathsf{bnd}(j)]\end{aligned}$$

with the first equation following from the definition of *lbnd* and the second from $(*)$. This is formalised in the next lemma.

Lemma 5.4. *[Relating lbnd & inst (for Prop 5.6)] Given any $D \in \mathcal{DB}$, list L, $n \geq 0$, and meta-variable v_k fresh for L, we have*

$$\mathit{lbnd}\ n\ (\Lambda\, v_k.\, \mathit{inst}\ n\ (v_k, L)\ D) = \mathit{inst}\ (n+1)\ L\ D$$

Proof. We prove this by induction on D. The argument for constants $\mathsf{con}(\nu)$ is similar to that for the inductive case of free indices, and applications are easy.

$\boxed{\mathsf{var}(i)}$

$$\begin{aligned} &\mathit{lbnd}\ n\ (\Lambda\, v_k.\ \mathit{inst}\ n\ (v_k, L)\ \mathsf{var}(i)) \\ &\quad = \ \mathit{lbnd}\ n\ (\Lambda\, v_k.\, \mathsf{VAR}\ i) \\ &\quad = \ \mathsf{VAR}\ i \\ &\quad = \ \mathit{inst}\ (n+1)\ L\ \mathsf{var}(i) \end{aligned}$$

The equations follow from routine calculations.

$\boxed{\mathsf{bnd}(j)}$ Suppose that $n \leq j$ and $j - n < |v_k, L|$. We consider separately the cases of $j - n = 0$ and $j - n \geq 1$. If $j - n = 0$ then

$$\begin{aligned} &\mathit{lbnd}\ n\ (\Lambda\, v_k.\ \mathit{inst}\ n\ (v_k, L)\ \mathsf{bnd}(j)) \\ &\quad = \ \mathit{lbnd}\ n\ (\Lambda\, v_k.\ \mathit{elt}\ (j-n)\ (v_k, L)) \\ &\quad = \ \mathit{lbnd}\ n\ (\Lambda\, v_k.\, v_k) \\ &\quad = \ \mathsf{BND}\ n = \mathsf{BND}\ j \\ &\quad = \ \mathit{inst}\ (n+1)\ L\ \mathsf{bnd}(j) \end{aligned}$$

where the final equality follows since $n + 1 > n = j$. If $j - n \geq 1$ then

$$\begin{aligned} &\mathit{lbnd}\ n\ (\Lambda\, v_k.\ \mathit{inst}\ n\ (v_k, L)\ \mathsf{bnd}(j)) \\ &\quad = \ \mathit{lbnd}\ n\ (\Lambda\, v_k.\ (\mathit{elt}\ (j-n)\ (v_k, L))) \\ &\quad = \ \mathit{lbnd}\ n\ \Lambda\, (v_k.\, v_{k'}) \qquad\qquad \textit{where } k \neq k' \textit{ by freshness} \\ &\quad = \ v_{k'} \\ &\quad = \ \mathit{elt}\ (j - (n+1))\ L \\ &\quad = \ \mathit{inst}\ (n+1)\ L\ \mathsf{bnd}(j) \end{aligned}$$

where the final equality follows since $n + 1 \leq j$ and $j - (n + 1) < |L|$.

Now suppose that either $n > j$ or $j - n \geq |v_k, L|$.

$$\begin{aligned} &\mathit{lbnd}\ n\ (\Lambda\, v_k.\ \mathit{inst}\ n\ (v_k, L)\ \mathsf{bnd}(j)) \\ &\quad = \ \mathit{lbnd}\ n\ (\Lambda\, v_k.\, \mathsf{BND}\ j) \\ &\quad = \ \mathsf{BND}\ j \\ &\quad = \ \mathit{inst}\ (n+1)\ L\ \mathsf{bnd}(j) \end{aligned}$$

where the final equality follows since either $n + 1 > j$ or $j - (n + 1) \geq |L|$.

$\boxed{\mathsf{abs}(D)}$

$$\begin{aligned} &\mathit{lbnd}\ n\ (\Lambda\, v_k.\ \mathit{inst}\ n\ (v_k, L)\ \mathsf{abs}(D)) \\ &\quad = \ \mathit{lbnd}\ n\ (\Lambda\, v_k.\, \mathsf{ABS}\ (\mathit{inst}\ (n+1)\ (v_k, L)\ D)) \\ &\quad = \ \mathsf{ABS}\ (\mathit{lbnd}\ (n+1)\ (\Lambda\, v_k.\ \mathit{inst}\ (n+1)\ (v_k, L)\ D)) \\ &\quad = \ \mathsf{ABS}\ (\mathit{inst}\ (n+2)\ L\ D) \\ &\quad = \ \mathit{inst}\ (n+1)\ L\ \mathsf{abs}(D) \end{aligned}$$

The first, second and fourth equalities are true by definition. The third is by induction. □

The next lemma allows us to do some variable re-naming when applying the function *inst*. It will be used during inductive steps of proofs which deal with abstractions and hence also variable re-naming. Recall that $[\![-]\!]_{\_}$ was defined on page 17.

Lemma 5.5. *(for Prop 5.6) For all $E \in \mathcal{LE}$, and all $n \geq 0$, lists L, L' such that $n = |L'|$, and variables v_i, v_k we have*

$$(inst\ n\ (v_i, L)\ [\![E]\!]_{L', v_i, L})[v_k/v_i] = inst\ n\ (v_k, L)\ [\![E]\!]_{L', v_i, L}$$

Proof. The proof is by induction over $E \in \mathcal{LE}$. The details for constants and applications are easy and omitted.

$\boxed{v_\alpha}$
(Case $v_\alpha \notin L', v_i, L$): $[\![v_\alpha]\!]_{L', v_i, L} = \mathsf{var}(\alpha)$ and so

$$\begin{array}{rcl}(inst\ n\ (v_i, L)\ [\![v_\alpha]\!]_{L', v_i, L})[v_k/v_i] & = & (inst\ n\ (v_i, L)\ \mathsf{var}(\alpha))[v_k/v_i] \\ & = & (\mathsf{VAR}\ \alpha)[v_k/v_i] \\ & = & (\mathsf{VAR}\ \alpha) \\ & = & inst\ n\ (v_k, L)\ \mathsf{var}(\alpha)\end{array}$$

(Case $v_\alpha \in L'$): $[\![v_\alpha]\!]_{L', v_i, L} = \mathsf{bnd}(j)$ where $j = pos\ v_\alpha\ (L', v_i, L) < |L'| = n$

$$\begin{array}{rcl}(inst\ n\ (v_i, L)\ [\![v_\alpha]\!]_{L', v_i, L})[v_k/v_i] & = & (inst\ n\ (v_i, L)\ \mathsf{bnd}(j))[v_k/v_i] \\ & = & \mathsf{BND}\ j \\ & = & inst\ n\ (v_k, L)\ \mathsf{bnd}(j)\end{array}$$

(Case $v_\alpha \notin L'$ and $\alpha = i$): $[\![v_\alpha]\!]_{L', v_i, L} = \mathsf{bnd}(pos\ v_\alpha\ (L', v_i, L)) = \mathsf{bnd}(n)$

$$\begin{array}{rcl}(inst\ n\ (v_i, L)\ [\![v_\alpha]\!]_{L', v_i, L})[v_k/v_i] & = & (inst\ n\ (v_i, L)\ \mathsf{bnd}(n))[v_k/v_i] \\ & = & v_i[v_k/v_i] \\ & = & v_k \\ & = & inst\ n\ (v_k, L)\ \mathsf{bnd}(n)\end{array}$$

(Case $v_\alpha \notin L', v_i$ and $v_\alpha \in L$): $[\![v_\alpha]\!]_{L', v_i, L} = \mathsf{bnd}(j)$ where

$$j = pos\ v_\alpha\ (L', v_i, L) > n$$

and so $0 < j - n = pos\ v_\alpha\ (v_i, L) < |v_i, L|$

$$\begin{array}{rcl}(inst\ n\ (v_i, L)\ [\![v_\alpha]\!]_{L', v_i, L})[v_k/v_i] & = & (inst\ n\ (v_i, L)\ \mathsf{bnd}(j))[v_k/v_i] \\ & = & (elt\ (j - n)\ (v_i, L))[v_k/v_i] \\ & = & v_\alpha[v_k/v_i] \\ & = & v_\alpha \\ & = & elt\ (j - n)\ (v_k, L) \\ & = & inst\ n\ (v_k, L)\ \mathsf{bnd}(j)\end{array}$$

$\boxed{\lambda\, v_\alpha.\, E}$ We have

$$\begin{array}{rcl}(\mathit{inst}\ n\ (v_i, L)\ [\![\lambda\, v_\alpha.\, E]\!]_{L',v_i,L})[v_k/v_i] & & \\ = & (\mathit{inst}\ n\ (v_i, L)\ \mathsf{abs}([\![E]\!]_{v_\alpha,L',v_i,L}))[v_k/v_i] & \\ = & \mathsf{ABS}\ (\mathit{inst}\ (n+1)\ (v_i, L)\ [\![E]\!]_{v_\alpha,L',v_i,L}) & \\ = & \mathsf{ABS}\ ((\mathit{inst}\ (n+1)\ (v_k, L)\ [\![E]\!]_{v_\alpha,L',v_i,L})[v_k/v_i]) & \\ = & (\mathit{inst}\ n\ (v_i, L)\ \mathsf{abs}([\![E]\!]_{v_\alpha,L',v_i,L}))[v_k/v_i] & \\ = & \mathit{inst}\ n\ (v_k, L)\ [\![\lambda\, v_\alpha.\, E]\!]_{L',v_i,L} & \end{array}$$

with the third equality following by induction. □

Recall Section 5.2 in which we explained that the function Θ_L would be factored through the function θ_L. We can now formalise this and provide a proof. Note that the following proposition provides an explicit connection between the HYBRID encoding function Θ_ϵ and the de Bruijn adequacy function $\theta = \theta_\epsilon$.

Proposition 5.6. *[Factoring Θ_L] For any $[E]_\alpha \in \mathcal{LE}/\!\sim_\alpha$ and list L we have*

$$\Theta_L\, [E]_\alpha = \mathit{inst}\ 0\ L\ (\theta_L([E]_\alpha))$$

where the action of Θ_L is specified in the statement of Theorem 5.2. Recall (page 21) that for any E we set $\theta_L([E]_\alpha) \stackrel{\text{def}}{=} ([\![-]\!]_L \circ q^{-1})([E]_\alpha) = [\![E]\!]_L$ well defined by Proposition C.3. Hence the action yields a well defined function Θ_L as it is the composition action of three other well defined functions:

$$\Theta_L\ = (\mathit{inst}\ 0\ L\ ) \circ \iota \circ \theta_L : \mathcal{LE}/\!\sim_\alpha \to \mathcal{DB}(|L|) \to \mathcal{DB} \to \mathcal{CLF}_{exp}(\Gamma^L_{exp})$$

Proof. Note that the function $\mathit{inst}\ 0\ L$ is defined on $\mathcal{DB}$ and hence on any subset! We prove by induction on $E \in \mathcal{LE}$,

$$(\forall E)(\forall L)(\mathit{inst}\ 0\ L\ (\theta_L([E]_\alpha)) = \Theta_L\, ([E]_\alpha))$$

Constants and applications are easy; details are omitted.
$\boxed{v_i}$ When $v_i \in L$ we have

$$\begin{array}{rl}\mathit{inst}\ 0\ L\ (\theta_L([v_i]_\alpha)) = \mathit{inst}\ 0\ L\ (\mathsf{bnd}(\mathit{pos}\, v_i\, L)) = & \\ \mathit{elt}\ (\mathit{pos}\, v_i\, L)\, L = v_i \stackrel{\text{def}}{=} \Theta_L\, ([v_i]_\alpha) & \end{array}$$

where the second equality follows since $0 \leq \mathit{pos}\, v_i\, L < |L|$ and the other simple case is left to the reader.

$\boxed{\lambda\, v_i.\, E}$ We choose v_k to be fresh for L, E and v_i. Then

$$
\begin{array}{rcl}
\mathit{inst}\ 0\ L\ (\theta_L([\lambda\, v_i.\, E]_\alpha)) & & \\
\quad (\text{Def } \theta_L \text{ and C.3}) & = & \mathit{inst}\ 0\ L\ (\theta_L([\lambda\, v_k.\, E[v_k/v_i]]_\alpha)) \\
& \stackrel{\text{def}}{=} & \mathit{inst}\ 0\ L\ (\mathsf{abs}([\![E[v_k/v_i]]\!]_{v_k,L})) \\
& \stackrel{\text{def}}{=} & \mathsf{ABS}\ (\mathit{inst}\ 1\ L\ [\![E[v_k/v_i]]\!]_{v_k,L}) \\
\quad (5.4 \text{ with } v_k \notin L) & = & \mathsf{ABS}\ (\mathit{lbnd}\ 0\ (\Lambda\, v_k.\, \mathit{inst}\ 0\ (v_k, L)\ [\![E[v_k/v_i]]\!]_{v_k,L})) \\
\quad (\text{C.11 with } v_k \notin \mathit{fv}(E)) & = & \mathsf{ABS}\ (\mathit{lbnd}\ 0\ (\Lambda\, v_k.\, \mathit{inst}\ 0\ (v_k, L)\ [\![E]\!]_{v_i,L})) \\
\quad (5.5) & = & \mathsf{ABS}\ (\mathit{lbnd}\ 0\ (\Lambda\, v_k.\, (\mathit{inst}\ 0\ (v_i, L)\ [\![E]\!]_{v_i,L})[v_k/v_i])) \\
\quad (\alpha) & = & \mathsf{ABS}\ (\mathit{lbnd}\ 0\ (\Lambda\, v_i.\, \mathit{inst}\ 0\ (v_i, L)\ [\![E]\!]_{v_i,L})) \\
& = & \mathsf{ABS}\ (\mathit{lbnd}\ 0\ (\Lambda\, v_i.\, \mathit{inst}\ 0\ (v_i, L)\ \theta_{v_i,L}([E]_\alpha))) \\
& = & \mathsf{ABS}\ (\mathit{lbnd}\ 0\ (\Lambda\, v_i.\, \Theta_{v_i,L}\ [E]_\alpha)) \\
& \stackrel{\text{def}}{=} & \Theta_L\ ([\lambda\, v_i.\, E]_\alpha)
\end{array}
$$

We appeal to Lemma 5.4, Lemma C.11 and Lemma 5.5 for the steps labelled above respectively. Step (α) follows from Proposition 3.1 since canonical forms are identified up to α-equivalence: Note that we must ensure $v_k \notin \mathit{inst}\ 0\ (v_i, L)\ [\![E]\!]_{v_i,L}$. In fact it is easy to see that for any D, $\hat{L}$ and m, the variables occurring in $\mathit{inst}\ m\ \hat{L}\ D$ must come from $\hat{L}$. Thus the condition holds as v_k is fresh for v_i, L. The penultimate equality is by induction. □

The other key step in proving the main theorem is in showing that *inst* is an injective function. We achieve this by defining a left inverse, whose existence is proved in the next lemma. Before reading the proof, it may be useful for the reader to verify the following example calculation.

$$
\begin{array}{l}
\mathit{hdb}\ 0\ v_8\ (\mathit{inst}\ 0\ v_8\ (\mathsf{ABS}\ (\mathsf{BND}\ 1\ \$\$\ \mathsf{BND}\ 0))) \\
\quad = \mathit{hdb}\ 0\ v_8\ (\mathsf{ABS}\ (v_8\ \$\$\ \mathsf{BND}\ 0)) \\
\quad = \mathsf{ABS}\ ((\mathit{hdb}\ 1\ v_8\ v_8)\ \$\$\ (\mathit{hdb}\ 1\ v_8\ (\mathsf{BND}\ 0))) \\
\quad = \mathsf{ABS}\ (\mathsf{BND}\ 1\ \$\$\ \mathsf{BND}\ 0)
\end{array}
$$

Lemma 5.7. *[Defining hdb (for Prop 5.8)] For all $n \geq 0$ and list L, there is a unique function with the following source and target*

$$\mathit{hdb}\ n\ L : \mathcal{CLF}_{exp}(\Gamma^L_{exp}) \to \mathcal{DB}$$

satisfying the recursion equations below

— $\mathit{hdb}\ n\ L\ v_k = \mathsf{bnd}((\mathit{pos}\ v_k\ L) + n)$ *(recall, page 15, that pos e L computes position of e in L)*
— $\mathit{hdb}\ n\ L\ (\mathsf{CON}\ \nu) = \mathsf{con}(\nu)$
— $\mathit{hdb}\ n\ L\ (\mathsf{VAR}\ i) = \mathsf{var}(i)$
— $\mathit{hdb}\ n\ L\ (\mathsf{BND}\ j) = \mathsf{bnd}(j)$
— $\mathit{hdb}\ n\ L\ (C_1\ \$\$\ C_2) = (\mathit{hdb}\ n\ L\ (C_1))\ \$\ (\mathit{hdb}\ n\ L\ (C_2))$
— $\mathit{hdb}\ n\ L\ (\mathsf{ABS}\ C) = \mathsf{abs}(\mathit{hdb}\ (n+1)\ L\ C)$

Proof.

Let I be the inductively defined set of all canonical forms in context specified in Figure 4. Let L be arbitrary and $S \stackrel{\text{def}}{=} \mathcal{CLF}_{exp}(\Gamma^L_{exp})$. We will give an existence proof of a function $F : S \to (\mathbb{N} \Rightarrow (\{L\} \Rightarrow W))$ and define for any $n \in \mathbb{N}$

$$hdb\ n\ L\ C \stackrel{\text{def}}{=} F(C)(n)(L)$$

Let $I_0 \stackrel{\text{def}}{=} \varnothing$ and let I_{h+1} be those elements of I with deduction trees of height less than or equal to $h+1$ where $h \in \mathbb{N}$. It is standard that $I = \bigcup_{h\in\mathbb{N}} I_h$ and that the I_h form an increasing sequence of sets ordered by inclusion.

Let $S_0 \stackrel{\text{def}}{=} \varnothing$, and for $h \geq 0$ define

$$S_{h+1} \stackrel{\text{def}}{=} \{C \mid \Gamma^L_{exp} \vdash_{can} C :: exp \in I_{h+1} - I_h\}$$

It is easy to see that $S = \bigcup_{h\in\mathbb{N}} S_h$, and moreover this is a union of pairwise disjoint sets. We will now construct the function $F : S \to \mathbb{N} \Rightarrow (\{L\} \Rightarrow W)$ by defining a sequence of functions $F_h : S_h \to (\mathbb{N} \Rightarrow (\{L\} \Rightarrow W))$ and taking $F \stackrel{\text{def}}{=} \bigcup_{h\in\mathbb{N}} F_h$, a union of functions with pairwise disjoint sources.

$\boxed{F_0 : S_0 \to (\mathbb{N} \Rightarrow (\{L\} \Rightarrow W))}$ Take F_0 to be the empty function.

$\boxed{F_1 : S_1 \to (\mathbb{N} \Rightarrow (\{L\} \Rightarrow W))}$ If $C \in S_1$ then by examining the generating rules we see that C must be v_k. Hence F_1 is fully specified for each $n \in \mathbb{N}$ by

— $F_1(v_k)(n)(L) \stackrel{\text{def}}{=} \mathsf{bnd}((pos\ v_k\ L) + n)$

$\boxed{F_2 : S_2 \to (\mathbb{N} \Rightarrow (\{L\} \Rightarrow W))}$ If $C \in S_2$ then by examining the rules we see that C must be either $\mathsf{CON}\ \nu$, $\mathsf{VAR}\ i$, $\mathsf{BND}\ j$, or $v_k\ \$\$\ v_{k'}$, or $\mathsf{ABS}\ v_{k''}$. Hence F_2 is fully specified for each $n \in \mathbb{N}$ by

— $F_2(\mathsf{CON}\ \nu)(n)(L) \stackrel{\text{def}}{=} \mathsf{CON}\ \nu$
— $F_2(\mathsf{VAR}\ i)(n)(L) \stackrel{\text{def}}{=} \mathsf{VAR}\ i$
— $F_2(\mathsf{BND}\ j)(n)(L) \stackrel{\text{def}}{=} \mathsf{BND}\ j$
— $F_2(v_k\ \$\$\ v_{k'})(n)(L) \stackrel{\text{def}}{=} (F_1(v_k)(n)(L))\ \$\$\ (F_1(v_{k'})(n)(L))$
— $F_2(\mathsf{ABS}\ v_{k''})(n)(L) \stackrel{\text{def}}{=} \mathsf{ABS}\ (F_1(v_{k''})(n+1)(L))$

$\boxed{F_h : S_h \to (\mathbb{N} \Rightarrow (\{L\} \Rightarrow W)) \text{ where } h \geq 3}$ We prove by induction that for all $h \geq 3$ there is such a function F_h which is fully specified by the clauses

— $F_h(C_1\ \$\$\ C_2)(n)(L) \stackrel{\text{def}}{=} ((\bigcup_0^{h-1} F_r)(C_1)(n)(L))\ \$\$\ ((\bigcup_0^{h-1} F_r)(C_2)(n)(L))$
— $F_h(\mathsf{ABS}\ C_3)(n)(L) \stackrel{\text{def}}{=} \mathsf{ABS}\ ((\bigcup_0^{h-1} F_r)(C_3)(n)(L))$

together with the functions F_h defined above for $h = 0, 1, 2$. If $h = 0$ the proposition is vacuous. Suppose that it holds for all numbers r less than or equal to an arbitrary h. We prove it holds for $h+1$. So now assume that $h+1 \geq 3$. Suppose that $C \in S_{h+1}$. Now $h \geq 2$ and by examining the rules used to generate C we see that either $C \equiv C_1\ \$\$\ C_2$ with each $C_s \in I_{h-1}$ and of type exp, or $C \equiv \mathsf{ABS}\ C_3$ with $C_3 \in I_{h-1}$ and of type exp. Hence $C_s \in \bigcup_0^h S_r$ for $s = 1, 2, 3$. By induction, each of the functions F_r exist for $0 \leq r \leq h$. Hence F_{h+1} is indeed completely specified by the given clauses.

Defining $F \stackrel{\text{def}}{=} \bigcup_{h\in\mathbb{N}} F_h$ which exists given that the F_h all exist, it is virtually immediate that the given definition of *inst* satisfies the stated recursion conditions. □

The next proposition states that the function *inst* has a left inverse and is thus injective. The result is a key step in proving HYBRID adequacy.

Proposition 5.8. *[Left Inverse for inst] Given any $D \in \mathcal{DB}$, $n \geq 0$ and ordered list L, we have*

$$hdb\ n\ L\ (inst\ n\ L\ D) = D$$

In particular, each function

$$inst\ n\ L\ : \mathcal{DB} \to \mathcal{CLF}_{exp}(\Gamma^{L}_{exp})$$

is injective, with left sided inverse hdb n L.

Proof. The existence of the functions follows from Lemmas 5.3 and 5.7. The equalities are proved by induction over de Bruijn expressions. The details for constants, variables and applications are easy.

$\boxed{\mathsf{bnd}(j)}$ The first possibility is that

$$hdb\ n\ L\ (inst\ n\ L\ \mathsf{bnd}(j)) = hdb\ n\ L\ (elt\ (j-n)\ L) = \mathsf{bnd}((pos\ (elt\ (j-n)\ L)\ L) + n) = \mathsf{bnd}(j)$$

Note that this depends crucially on the fact that L is ordered. The second possibility is that

$$hdb\ n\ L\ (inst\ n\ L\ \mathsf{bnd}(j)) = hdb\ n\ L\ (\mathsf{BND}\ j) = \mathsf{bnd}(j)$$

$\boxed{\mathsf{abs}(D)}$

$$\begin{array}{rcl} hdb\ n\ L\ (inst\ n\ L\ \mathsf{abs}(D)) & = & hdb\ n\ L\ (\mathsf{ABS}\ (inst\ (n+1)\ L\ D)) \\ & = & \mathsf{abs}(hdb\ (n+1)\ L\ (inst\ (n+1)\ L\ D)) \\ & = & \mathsf{abs}(D) \end{array}$$

where the final equality follows by induction. □

Before finally proving the adequacy theorem we give a short technical lemma which allows us to show in Proposition 5.10 that the function *inst* is a compositional homomorphism.

Lemma 5.9. *(for Prop 5.10) For all $n \geq 0$, ordered L, and $D \in \mathcal{DB}(|L|)$, if $n \geq m$ where m is the minimum level of D, then*

$$inst\ (n+1)\ L\ D = inst\ n\ L\ D$$

Proof. This is a routine induction over D. In the case that D is $\mathsf{bnd}(j)$, note that *inst* returns $\mathsf{BND}\ j$ as the minimum level m of $\mathsf{bnd}(j)$ is $j+1$, and thus both n and $n+1$ are strictly greater than j. □

Proposition 5.10. *[inst Compositional Homomorphism] For any $D, D' \in \mathcal{DB}$, $n, k \geq 0$, and ordered L, if $n \geq m$ where m is the minimum level of D', then*

$$\mathit{inst}\ n\ L\ (D[D'/\mathsf{var}(k)]) = (\mathit{inst}\ n\ L\ D)[\mathit{inst}\ n\ L\ D'/\mathsf{VAR}\ k]$$

Proof. The proof is by induction on D. All of the cases are easy, except for abstractions $\mathsf{abs}(D)$ which require Lemma 5.9. We have

$$\begin{array}{rcl} \mathit{inst}\ n\ L\ (\mathsf{abs}(D)[D'/\mathsf{var}(k)]) & & \\ \quad = & \mathsf{ABS}\ (\mathit{inst}\ (n+1)\ L\ (D[D'/\mathsf{var}(k)])) & \\ \quad = & \mathsf{ABS}\ ((\mathit{inst}\ (n+1)\ L\ D)[\mathit{inst}\ (n+1)\ L\ D'/\mathsf{VAR}\ k]) & \\ \quad = & (\mathit{inst}\ n\ L\ \mathsf{abs}(D))[\mathit{inst}\ (n+1)\ L\ D'/\mathsf{VAR}\ k] & \end{array}$$

The second equality is by induction, as $n+1 \geq n \geq m$ where m is the minimum level of D'. The third follows from simple applications of the definitions, and we are done by appeal to Lemma 5.9 applied to D'. □

5.5. *Proving* Hybrid *Adequacy*

We can now prove Theorem 5.2.

Proof.

B We only need to show that Θ_ϵ is injective. From Proposition 5.6 [*Factoring* Θ_L] we know that $\Theta_\epsilon = (\mathit{inst}\ 0\ \epsilon) \circ \iota \circ \theta_\epsilon$. But $\theta_\epsilon = \theta$ and θ is injective by Theorem 4.1 [*de Bruijn Representational Adequacy*]. Inclusion ι is trivially injective. Furthermore *inst* $0\ \epsilon$ is injective by Proposition 5.8 [*Left Inverse for inst*] (noting that ϵ is ordered by definition).

CH The Hybrid substitution function exists by appeal to Lemma 5.1. We calculate

$$\begin{array}{rcll} \Theta_\epsilon\,([E]_\alpha[[E']_\alpha/v_k]) & \stackrel{\text{def}}{=} & \Theta_\epsilon\,[E[E'/v_k]]_\alpha & (1) \\ & = & \mathit{inst}\ 0\ \epsilon\ [\![E[E'/v_k]]\!]_\epsilon & (2) \\ & = & \mathit{inst}\ 0\ \epsilon\ ([\![E]\!]_\epsilon[[\![E']\!]_\epsilon/\mathsf{var}(k)]) & (3) \\ & = & (\mathit{inst}\ 0\ \epsilon\ [\![E]\!]_\epsilon)[\mathit{inst}\ 0\ \epsilon\ [\![E']\!]_\epsilon/\mathsf{VAR}\ k] & (4) \\ & = & (\Theta_\epsilon\,[E]_\alpha)[\Theta_\epsilon\,[E']_\alpha/\mathsf{VAR}\ k] & (5) \end{array}$$

Equation 2 follows from Proposition 5.6 [*Factoring* Θ_L]. Equation 3 follows from Theorem 4.1 [*de Bruijn Representational Adequacy*]. Equation 4 follows from Proposition 5.10 [*inst Compositional Homomorphism*]—note that the value of n in the lemma is 0 and so $n \geq m$ where $m = 0$ is the minimum level of $[\![E']\!]_\epsilon \in \mathcal{DB}(0)$. Equation 5 follows from Proposition 5.6 [*Factoring* Θ_L]. □

6. Representation Results

We can use the results of the previous sections to prove some facts about the HYBRID representation of λ-expressions. The reader was reminded of the notion of a *proper* de Bruijn term on page 4 (and see also Appendix Section B.1). We provide an analogous definition for canonical expressions in HYBRID and prove that they correspond exactly to λ-expressions. We also define the notion of a HYBRID *abstraction* and show that such expressions correspond to λ-abstraction expressions. These predicates are important, since they are required for the formulation of induction principles. We illustrate the notion of abstraction by example. Suppose that $\mathsf{ABS}\ C$ is proper; for example let $C = \mathsf{ABS}\ (\mathsf{BND}\ 0\ \$\$\ \mathsf{BND}\ 1)$. Then C is of level 1, and in particular there may be some bound indices which now dangle; for example $\mathsf{BND}\ 1$ in $\mathsf{ABS}\ (\mathsf{BND}\ 0\ \$\$\ \mathsf{BND}\ 1)$. An abstraction is produced by replacing each occurrence of a dangling index with a metavariable and then abstracting the meta variable. Our example yields the abstraction $\Lambda\, v.\, \mathsf{ABS}\ (\mathsf{BND}\ 0\ \$\$\ v)$.

We mention briefly an induction principle for HYBRID. When HYBRID is put into practice, an object logic will be translated by regarding the datatype of de Bruijn *plus* $\mathsf{LAM}\, v_i.\, \xi$ expressions as a form of HOAS. This requires that the constructors CON, VAR, \$\$, and LAM should be injective, with disjoint images (van Dalen, 1989). The approach is to identify predicated subsets of *exp* and $exp \Rightarrow exp$. The subset of *exp* consists of expressions which reduce to proper de Bruijn expressions (see page 4). The subset of $exp \Rightarrow exp$ consists of functions C such that $\mathsf{LAM}\, v_i.\, C\ v_i$ is proper. The subsets consist of **proper** and **abstraction** expressions respectively. With this one may prove in HYBRID:

$$\frac{\begin{array}{l}\forall i.\ \Phi(\mathsf{VAR}\ i)\\ \forall C\ , C'.\ \mathsf{proper}\ C \wedge \Phi(C) \wedge \mathsf{proper}\ C' \wedge \Phi(C') \Longrightarrow \Phi(C\ \$\$\ C')\\ \forall C.\ \mathsf{abst}\ C \wedge (\forall C'.\ \mathsf{proper}\ C' \Longrightarrow \Phi(C') \Longrightarrow \Phi(C\ C')) \Longrightarrow \Phi(\mathsf{LAM}\, v_i.\, C\ v_i)\end{array}}{\Phi(C)}$$

6.1. *Describing* HYBRID *Proper Expressions Using* $\mathcal{LE}$

Proposition 6.1. *For all $n \geq 0$ and lists L there is a unique function with the following source and target*

$$level\, n : \mathcal{CLF}_{exp}(\Gamma^{L}_{exp}) \to \mathbb{B}$$

which satisfies the following recursion equations

— $level\ n\,(\mathsf{CON}\ \nu) = T$
— $level\ n\, v_k = F$
— $level\ n\,(\mathsf{VAR}\ i) = T$
— $level\ n\,(\mathsf{BND}\ j) = n > j$
— $level\ n\,(C_1\ \$\$\ C_2) = (level\ n\, C_1) \wedge (level\ n\, C_2)$
— $level\ n\,(\mathsf{ABS}\ C) = level\ (n+1)\, C$

Proof. We omit the proof which is similar in spirit to that given for Proposition 3.1. □

We say that an element $C \in \mathcal{CLF}_{exp}(\Gamma^{L}_{exp})$ is **proper** if *level* $0\,C$ is equal to T.

Theorem 6.2. *Suppose that* $C \in \mathcal{CLF}_{exp}(\epsilon)$ *and that* C *is proper. Then there exists* $[E]_\alpha \in \mathcal{LE}/{\sim_\alpha}$ *such that*

$$\Theta_\epsilon\,[E]_\alpha = C.$$

Proof. We omit the proof which is similar to that given for Theorem 6.5 below. □

6.2. *Describing* Hybrid *Abstraction Expressions Using* $\mathcal{LE}$

Proposition 6.3. *For all* $n \geq 0$ *and lists* L *there is a unique function with the following source and target*

$$abst\ n : \mathcal{CLF}_{exp \Rightarrow exp}(\Gamma^{L}_{exp}) \to \mathbb{B}$$

which satisfies the following recursion equations

— $abst\ n\ (\Lambda\, v_k.\, \mathsf{CON}\ \nu) = T$
— $abst\ n\ (\Lambda\, v_k.\, v_k) = T$
— $abst\ n\ (\Lambda\, v_k.\, v_{k'}) = F$
— $abst\ n\ (\Lambda\, v_k.\, \mathsf{VAR}\ i) = T$
— $abst\ n\ (\Lambda\, v_k.\, \mathsf{BND}\ j) = n < j$
— $abst\ n\ (\Lambda\, v_k.\, C_1\ \$\$\ C_2) = (abst\ n\ (\Lambda\, v_k.\, C_1)) \wedge (abst\ n\ (\Lambda\, v_k.\, C_2))$
— $abst\ n\ (\Lambda\, v_k.\, \mathsf{ABS}\ C) = abst\ (n+1)\ (\Lambda\, v_k.\, C)$

Proof. We omit the proof which is very similar to that of Proposition 3.1. □

We say that an element $C \in \mathcal{CLF}_{exp \Rightarrow exp}(\Gamma^{L}_{exp})$ is an **abstraction** if $abst\ 0\ C$ is equal to T.

Lemma 6.4. *Given any canonical expression* $C \in \mathcal{CLF}_{exp}(v_i :: exp)$ *and* $n \geq 0$ *for which*

$$abst\ n\ (\Lambda\, v_i.\, C) = T$$

we have

$$inst\ n\ v_i\ (hdb\ n\ v_i\ C) = C$$

Proof. We prove the lemma by induction over the derivations of canonical forms.
VAR. By the assumption C is a variable, and only $C = v_i$ ensures that we have $abst\ n\ (\Lambda\, v_i.\, C) = T$. Hence

$$\begin{array}{rcll} \mathit{inst}\ n\ v_i\ (\mathit{hdb}\ n\ v_i\ v_i) & = & \mathit{inst}\ n\ v_i\ (\mathsf{bnd}(n)) & (*) \\ & = & \mathit{elt}\ (n-n)\ v_i & \\ & = & \mathit{elt}\ 0\ v_i & \\ & = & v & \end{array}$$

with $(*)$ holding since $n \leq n$ and $n - n = 0 < |v_i| = 1$.

CST. The cases where κ is CON or VAR are trivial. The case $$ is also immediate by induction. When κ is BND we have $\mathit{abst}\ n\ (\Lambda\, v_i.\, \mathsf{BND}\ j) = n < j$ and then

$$\begin{array}{rcl} \mathit{inst}\ n\ v_i\ (\mathit{hdb}\ n\ v_i\ (\mathsf{BND}\ j)) & = & \mathit{inst}\ n\ v_i\ (\mathsf{bnd}(j)) \\ & = & \mathsf{BND}\ j \end{array}$$

Finally, when κ is ABS we have $\mathit{abst}\ n\ (\Lambda\, v_i.\, \mathsf{ABS}\ C) = \mathit{abst}\ (n+1)\ C$ and then

$$\begin{array}{rcl} \mathit{inst}\ n\ v_i\ (\mathit{hdb}\ n\ v_i\ (\mathsf{ABS}\ C)) & = & \mathit{inst}\ n\ v_i\ (\mathsf{abs}(\mathit{hdb}\ (n+1)\ v_i\ C)) \\ & = & \mathsf{ABS}\ (\mathit{inst}\ (n+1)\ v_i\ (\mathit{hdb}\ (n+1)\ v_i\ C)) \\ & = & \mathsf{ABS}\ C \end{array}$$

with the final equation valid by induction. □

Theorem 6.5. *Suppose that $C \in \mathcal{CLF}_{exp \Rightarrow exp}(\epsilon)$ and that C is an abstraction. Then there exists $[\lambda\, v_i.\, E]_\alpha \in \mathcal{LE}/\!\sim_\alpha$ such that*

$$\Theta_\epsilon\, [\lambda\, v_i.\, E]_\alpha = \mathsf{LAM}\, v_i.\, C\ v_i$$

Proof. We define $E \stackrel{\text{def}}{=} (\!|\mathit{hdb}\ 0\ v_i\ (C\ v_i)|\!)_{v_i}$. From Proposition C.4 it follows that $[\![E]\!]_{v_i} = \mathit{hdb}\ 0\ v_i\ (C\ v_i)$. Then we have

$$\begin{array}{rcll} \Theta_\epsilon\, [\lambda\, v_i.\, E]_\alpha & = & ((\mathit{inst}\ 0\ \epsilon) \circ \iota \circ \theta_\epsilon)[\lambda\, v_i.\, E]_\alpha & (6) \\ & = & \mathit{inst}\ 0\ \epsilon\ [\![\lambda\, v_i.\, E]\!]_\epsilon & (7) \\ & = & \mathit{inst}\ 0\ \epsilon\ \mathsf{abs}((\![E]\!]_\epsilon)) & (8) \\ & = & \mathsf{ABS}\ (\mathit{inst}\ 1\ \epsilon\ [\![E]\!]_\epsilon) & (9) \\ & = & \mathsf{ABS}\ (\mathit{lbnd}\ 0\ (\Lambda\, v_i.\, \mathit{inst}\ 0\ v_i\ [\![E]\!]_v)) & (10) \\ & = & \mathsf{ABS}\ (\mathit{lbnd}\ 0\ \Lambda\, v_i.\, \mathit{inst}\ 0\ v\ (\mathit{hdb}\ 0\ v\ (C\ v_i))) & (11) \\ & = & \mathsf{ABS}\ (\mathit{lbnd}\ 0\ \Lambda\, v_i.\, C\ v_i) & (12) \\ & \stackrel{\text{def}}{=} & \mathsf{LAM}\, v_i.\, C\ v_i & (13) \end{array}$$

Equation 6 follows from Proposition 5.6. Equation 7 follows from the definition of θ_L on page 21 and equation 8 follows by calculating with θ_L. Equation 9 follows from Lemma 5.3. Equation 10 follows from Lemma 5.4. We appeal to Lemma 6.4 to obtain 12. □

7. Related Work

7.1. Hybrid *Systems*

In this section we review some of the work that has been done on developing systems for representing, and reasoning about, syntax with variable binding.

Our original work demonstrated the utility of the Hybrid approach. Simple case studies appeared in (Ambler et al., 2002b), and the (somewhat notorious) Howe's method was tackled in (Ambler et al., 2002a). Comparisons of the Hybrid approach with other systems can be found in (Momigliano et al., 2001; Felty and Pientka, 2010).

In such case studies one needs to know that the translation of an object logic into the logic used for reasoning (that is, the meta logic, which in our case studies is Hybrid itself) is adequate. One reason for doing this is that it ensures (or at least provides evidence) that results proven about the *representations* of the object logic in the meta logic actually *do* hold for the object logic itself. From this point of view one might regard the result of this paper as a kind of "generic" adequacy. It is future work to investigate if this can be made technically precise and is indeed of practical use; see (Aydemir et al., 2008) for background discussions about adequacy.

For work by others that is directly related to ours, note that Venanzio Capretta and Amy Felty have recently implemented a Hybrid system in Coq (Capretta and Felty, 2007). Although the approach is slightly different, this work was inspired by the techniques presented in (Ambler et al., 2002b). See also Section 7.4.

Hybrid systems have also been formulated and implemented with respect to two-level logic approach to specification and reasoning (Momigliano and Ambler, 2003; Felty and Momigliano, 2010). Typically there is a level defined by a specification logic for encoding structural semantics. Another level is provided for the development of proofs of properties about specifications; such proofs may involve (co)induction but the additional level avoids the usual problems met when trying to combine inductive proofs and hypothetical judgements.

In each of these papers there are slight technical variations on the approaches towards hybrid syntax. For example in Coq, the *lbnd* function is defined using the description axiom from the classical libraries, whereas Isabelle/HOL Hybrid uses the description operator. However, at heart one finds that such syntax is formulated using the key notions that are (mathematically) presented in this paper. It remains future work to present a summary of the methodologies developed by other authors, and to prove the conjecture that the adequacy proofs developed here can be easily translated to these other scenarios, but it seems likely that this is the case.

7.2. *Nameless Binders*

This notion was introduced by de Bruijn in 1972 (de Bruijn, 1972). One finds syntax such as $\lambda.(\lambda.1)0$ in which binding structure is specified by binding indices. Free variables may be named, eg as v, or realised as indices—the *locally nameless* and *pure* approaches described elsewhere in this paper. For a textbook account of nameless systems see (Hindley and Seldin, 1988). A comparison of nameless and named binding systems can be found in (Berghofer and Urban, 2006).

Shankar investigated bijections between pure de Bruijn expressions and λ-expressions in (Shankar, 1988), studying a variety of meta-theoretic properties. It seems that this paper is probably the closest other work in the literature from the point of detailing a bijection between de Bruijn and λ-expressions and some of the technical details are rather similar to ours. Shankar does not need to identify proper expressions, an advantage of pure de Bruijn, but does have to work with extremely complicated definitions of substitution.

Gordon (Gordon, 1994) proves that there is a bijection between locally nameless de Bruijn expressions (as used in this paper) and a formulation of λ-expressions (called `META`) that is very close to that found in HYBRID. If one examines Gordon's work in detail, then one could consider viewing `META` expressions as ordinary λ-expressions, but only in as much as one might do so with the named binding syntax in HYBRID. Gordon does not formalise α-equivalence classes of syntax trees as we do in our paper, and he does not work with standard primitive recursive substitution.

Norrish and Vestergaard (Norrish and Vestergaard, 2007) also undertake such a proof, but with yet another variation of de Bruijn expression, although closely related to pure de Bruijn—they comment that "The result most similar to that in this [(Norrish and Vestergaard, 2007)] paper is Shankars ... [(Shankar, 1988)]". Norrish and Vestergaard provide a very thorough survey indeed of the work undertaken to formalize such bijections, and this paper is a great place to learn the state-of-the-art.

Interestingly, recent work by Aydemir, Charguéraud, Pierce, Pollack and Weirich (Aydemir et al., 2008) has led to a novel logical framework that combines such a locally nameless representation of terms with cofinite quantification of free variable names in inductive definitions of relations on terms. This deals with the other side of the same coin: the problem of renaming of free names in proof derivations. It seems they are able to obtain structural induction principles using cofinite quantification that are strong enough for meta-theoretic reasoning.

In our paper only limited forms of (simple) substitution of de Bruijn expressions are required. One may wish to deal with substitutions that realise, for example, β-reductions. For functions which encode such substitutions, see for example the excellent book by Paulson (Paulson, 1997).

Machines are quite good at manipulating binding indices, but humans are not. It is much easier for most users to have an explicit link between a binding λ and bound variable, and fundamentally this can be encoded as a pair (v, E) where the named variable v would inhabit the binding λ node in a typical abstract syntax tree. This introduces a fundamental complication of α-equivalence. Traditional mechanizations worked with raw abstract syntax trees but must ensure that α-equivalence is an invariant (Ford and Mason, 2001; Melham, 1994; Vestergaard and Brotherson, 2001). These tools provide considerable support for induction and recursion, but dealing with α-equivalence is a considerable burden. As we discussed earlier, Gordon (Gordon, 1994) *defines* syntax with name binding in terms of an underlying type of de Bruijn λ-expressions, which yields an automated system for α-conversion. The work presented in our paper provides a similar working environment, but deals with named binders in a more sophisticated and convenient way, with the convenience of making direct use of the binding system of an implemented meta-logic (here, Isabelle/HOL). For other issues concerning renaming arising in proofs see (McKinna and Pollack, 1999); and for a short comparison of definitions of α-equivalence using named binders see (Crole, 2010).

All of these approaches deal with α-conversion which is overlaid on expressions of inductive datatypes whose elements are fundamentally raw syntax trees. A conceptually different approach is to work in a world where α-convertability is a native property, and then to construct datatypes. Such a novel approach was pioneered by Gabbay and Pitts (Gabbay and Pitts, 1999). They introduced a non-classical set-theory with an internal notion of *permutation* of atoms. Permutation is then used to provide a form of name swapping; elements can be identified up to swapped names, and this provides an in-built form of α-equivalence. Such a set-theory yields a natural notion of structural induction and recursion over α-equivalence classes of expressions, but it is incompatible with the axiom of choice. For the most recent developments see (Pitts, 2006). These basic ideas were developed into a first order axiomatic presentation formalising a primitive notion of swapping and *freshness* of names from which binding can be derived (Pitts, 2001; Pitts, 2003). An axiomatic approach closer to the spirit of this paper can be found in (Gordon and Melham, 1996). For the use of Isabelle in implementing nominal techniques see (Urban and Tasson, 2005) and (Urban, 2008). The latter paper describes a formalisation of the λ-calculus using nominal techniques. Central to the formalisation is an inductive set that is bijective with α-equivalence classes of λ-expressions. Further work has studied unification within the nominal framework (Urban et al., 2004). One aim of this work is to develop a framework for meta-programming applications, especially for developing operational semantics (see also (Miller, 2006)). As such an ML-like programming language, *FreshML*, has been coded (Shinwell et al., 2003; Shinwell and Pitts, 2005). More recently a metalanguage targeted at operational semantics has been developed (Lakin and Pitts, 2007). A number of people are now working on nominal logic (Pitts, 2003), and for example (concepts from) this system has been considerably developed in (Clouston and Pitts, 2007; Clouston, 2010). Cheney has developed a simple type theory for nominal

logic (Cheney, 2009) and has proven results such as type soundness and normalisation. Formulations of LF style frameworks in a nominal setting appear in (Berhofer et al., 2008) with updates in (Berhofer et al., 2010).

Capture-avoiding substitution is central to our work, and appears for example in specifications of logics and type theories. Murdoch Gabbay and Aad Mathijssen axiomatise capture-avoiding substitution using Nominal Algebra in (Gabbay and Mathijssen, 2008). More recently, in (Gabbay and Mathijssen, 2010), $\alpha\beta$-equivalence has been axiomatised in Nominal Algebra and proven sound and complete. This provides evidence that Nominal Algebra in particular is a good syntax in which to express axioms for names and binding.

7.4. *Functional Abstraction Binders*

In this setting, binding is encapsulated either

1 through functions from *names* to expressions; or
2 through functions from *expressions* to expressions.

For a general survey of such approaches see (Momigliano et al., 2001). We have already mentioned that Venanzio Capretta and Amy Felty implemented a HYBRID system in Coq (Capretta and Felty, 2007).

Further material about HOAS in type theory can be found in (Capretta and Felty, 2009); this paper contains generalisations both of our own work on HYBRID and also that in (Capretta and Felty, 2007). The paper, roughly speaking, provides a language of universal algebra with bindings (and higher order signatures) that has an underlying de Bruijn syntax. It has interesting connections to many other current approaches to this general area of research.

Approach (1) first appeared in (Despeyroux et al., 1995). It was developed to deal with the issues arising from *exotic* expressions which are created when realising binders: if binders are realised as functions on inductive datatypes, then there will exist expressions whose type matches the datatype but are not equal to the expressions which are supposed to arise from the type. (Of course, in HYBRID, non-exotic terms are isolated through the predicates of properness and abstraction.) This kind of approach to binding is logically axiomatised in Honsell et al.'s paper on the *Theory of Contexts,* (Honsell et al., 2001a). They define a higher-order logic inconsistent with unique choice, but extended with axioms that capture properties of freshness. Higher-order induction and recursion on expressions are assumed. An application of this approach to the π-calculus appears in (Honsell et al., 2001b). For another case-study see (Miculan, 2001), where the axioms seem less successful—coinduction is available, but substitution must be coded explicitly. A possible disadvantage of this approach is the complexity of the axiom system; indeed,

establishing consistency is a non-trivial task. For deeper connections between the nominal logic approach of Pitts and the Theory of Contexts see (Honsell et al., 2005).

For approach (2) see (Pfenning and Elliott, 1988; Harper et al., 1993). In such a setting there are two ways to integrate HOAS and induction: one where they coexist in the same language and the other where inductive reasoning is conducted at an additional meta-level. In the first one, a key problem is how to formulate (primitive) recursive definitions on functions of higher type while preserving adequacy of representations. This has been realised for the simply-typed case in (Despeyroux et al., 1997) and more recently for the dependently-typed case in (Despeyroux and Leleu, 2000). The idea is to separate at the type-theoretic level, via an S4 modal operator, the *primitive* recursive space (which encompasses functions defined via case analysis and iteration) from the *parametric* function space (whose members are those convertible to expressions built only via the constructors). In the *Twelf* project (Pfenning and Schürmann, 1999; Felty, 2002a) inductive reasoning is conducted at an additional meta-level in a fully automated way. In the meta-logic it is possible to express and inductively prove meta-logical properties of an object logic. The encoding is adequate, so the proof of the existence of the appropriate meta-level object(s) guarantees the proof of the corresponding object-level property.

Felty and Momigliano have undertaken considerable work on the two level reasoning approach (Momigliano et al., 2009; Felty and Momigliano, 2009; Felty, 2002b; Felty and Momigliano, 2010) that is also seen in systems such as Abella and also Twelf. Properties such as type soundness for a simple pure functional language have been proven via an intuitionistic specification logic. More advanced work has considered a similar result for a continuation style presentation of the operational semantics, this time using an ordered linear logic for the specification layer. This is particularly pleasing, showing the possibility of incorporating new specification logics, while also dealing with a fairly complex example.

Related to this, Miller & McDowell (McDowell and Miller, 1997) introduced a meta-meta logic, $FO\lambda^{\Delta IN}$, that is based on intuitionistic logic augmented with definitional reflection (Hallnas, 1991) and induction on natural numbers. Other inductive principles are *derived* via the use of appropriate measures. At the meta-meta level, they reasoned about object-level judgements formulated in second-order logic. They prove the consistency of the method by showing that $FO\lambda^{\Delta IN}$ enjoys cut-elimination (McDowell, 1997). Note that the $FO\lambda^{\Delta IN}$ system of (McDowell and Miller, 2002) is interactive. The latest developments of these kinds of ideas appear in the Abella system documented in (Gacek, 2008). Abella is an interactive system for reasoning about object languages (Gacek, 2008; Gacek et al., 2008) and has a two level structure. Specifications are made in the logic of second-order hereditary Harrop formulas and the logic is executable. The reasoning logic of Abella is able to encode the semantics of the specification logic as a definition and thereby reason over specifications.

7.5. *Models of Binders*

Although not directly related to the work in this paper, one should remark that a considerable amount of work has been done on the use of presheaf categories to model variable binding. The basic ideas appear in (Fiore et al., 1999; Hofmann, 1999), and Gabbay and Pitts also develop presheaf models in (Gabbay and Pitts, 2002). Roughly speaking, the idea is that for each $n \in \mathbb{N}$ there is a set of expressions with n free variables, yielding a contravariant functor from $(\mathbb{N}, \subseteq)$ to $\mathcal{S}et$ in which injections $m \subseteq n$ are mapped to binding functions which map expressions with n free variables to expressions with m free variables. One can find an interesting example in (Ambler et al., 2004) which is related to the HYBRID system.

8. Conclusions

We have shown that the core of the HYBRID system is adequate for the λ-calculus. In particular binding is realised through a form of functional abstraction and thus we have an adequate formulation of HOAS. We have also stated some simple representation results that establish direct links between HYBRID predicates and λ-expressions. Further work could involve the investigation of the notion of n-ary abstraction and associated higher order induction principles.

HYBRID presents users with a variety of forms of binding constructs and the internal operation of certain key functions can be confusing to beginners. We hope that this paper achieves its intended purpose of outlining the rôles of these key functions, and providing informative details about their operation through the various formal results presented here. One can hopefully concentrate on key mathematical details without being burdened by the full implementation of HYBRID. Of course it is only fair to say that any user will need to understand how to apply these functions in practice, and this will involve skills and knowledge over and above what is presented in this paper.

I would very much like to thank Alberto Momigliano for his support during the early stages of this work, and the anonymous referees whose detailed reports have led to considerable improvements.

Appendix A. Induction and Recursion

Induction and recursion play a central role in this paper, so we outline briefly what we shall assume (Aczel, 1977; Crole, 1998). Suppose we have a universal set U. A set of **(finitary) rules** $\mathcal{R} \subseteq \mathcal{P}(U) \times U$ is a collection of pairs (A, c) where A is a finite subset of U. If A is empty we call c a **base** element. A set I is **inductively defined** by a set of rules $\mathcal{R}$ if

$$I = \mu(X \overset{G}{\mapsto} \{e \in U \mid \exists (A, e) \in \mathcal{R} \wedge A \subseteq X\})$$

where μ denotes the least fixpoint of the endofunction G on $\mathcal{P}(U)$. From this one can derive the usual **principle of induction** for proving $\forall i \in I.\Phi(i)$. One can also prove that the elements i of I are exactly those elements of U for which there is a finite rooted tree with root i and such that any node c with set of children A forms a rule in $\mathcal{R}$. Moreover, if I_h is the collection of roots of such trees with height at most h, then one can prove that $I = \bigcup_{h\in\mathbb{N}} I_h$.

One can show further that functions $f : I \to W$ can be defined through recursion equations (van Dalen, 1989). Suppose that for each base element c we specify $f(c) = w \in W$ and for each rule $(\{a_1, \ldots, a_n\}, c)$ we specify $f(c) = E[f(a_1), \ldots, f(a_n)] \in W$ where E is some element of W depending on the $f(a_i)$. Then under certain conditions (van Dalen, 1989) one can prove that an f satisfying the equations $f(c) = \xi$ exists and is unique. Typically the existence proof goes by specifying functions $f_h : I_h \to W$ such that $f_{h+1}(c) = E[(\bigcup_{r=0}^{h} f_r)(a_1), \ldots, (\bigcup_{r=0}^{h} f_r)(a_n)]$ and setting $f \stackrel{\text{def}}{=} \bigcup_{h\in\mathbb{N}} f_h$, and in this paper we will prove the existence of functions using minor adaptations of this approach.

Appendix B. de Bruijn Expressions and λ-Expressions

B.1. *Syntax*

We inductively define a set of (object level) de Bruijn expressions. The set of expressions is denoted by $\mathcal{DB}$, with expressions generated by

$$D ::= \mathsf{con}(\nu) \mid \mathsf{var}(i) \mid \mathsf{bnd}(j) \mid \mathsf{abs}(D) \mid D_1 \ \$ \ D_2$$

where i and j range over the natural numbers $\mathbb{N}$, and ν over a set of names. One should think of a de Bruijn expression as a finite rooted syntax tree. The leaf nodes are labelled either by constants $\mathsf{con}(\nu)$; by $\mathsf{var}(i)$ which corresponds to a free variable; or by $\mathsf{bnd}(j)$ which corresponds to a bound variable. We employ informal notation for occurrences of subtrees. For example we may write $\mathsf{var}(i) \in D$ or possibly even $i \in D$. We call the j in expressions $\mathsf{bnd}(j)$ **bound indices**. We call the i in expressions $\mathsf{var}(i)$ **free indices**. Given a de Bruijn expression D, a bound index j which occurs in D is said to be **dangling** if the number of abs nodes occurring on the path between the index j and the root of D is j or less. Otherwise it is not dangling. D is said to be at **level** l, where $l \geq 0$, if enclosing[§] D inside l nodes, each labelled with abs, ensures that the resulting expression has no dangling indices. We can define a predicate $level\, n : \mathcal{DB} \to \mathbb{B}$ for each $n \in \mathbb{N}$ where $\mathbb{B} \stackrel{\text{def}}{=} \{T, F\}$ such that the Boolean $level\, l\, D$ is true just in case D is of level l, by setting

$$\begin{aligned} level\ l\ (\mathsf{con}(\nu)) &= T \\ level\ l\ (\mathsf{var}(i)) &= T \end{aligned}$$

[§] D enclosed by two such nodes is $\mathsf{abs}(\mathsf{abs}(D))$.

$$\begin{aligned} \mathit{level}\ l\,(\mathsf{bnd}(j)) &= l > j \\ \mathit{level}\ l\,(D_1\ \$\ D_2) &= (\mathit{level}\ l\,D_1) \wedge (\mathit{level}\ l\,D_2) \\ \mathit{level}\ l\,(\mathsf{abs}(D)) &= \mathit{level}\ (l+1)\,D \end{aligned}$$

It is (informally) clear that for any D, a (unique) minimum level m exists, and that D is at level l for any $l \geq m$. We write $\mathcal{DB}(l)$ for the set of de Bruijn expressions at level l.

This particular form of de Bruijn expression was originally chosen for the HYBRID system since it offers a good mixture of desirable features. Although we choose to specify free variables using natural numbers rather than strings, results and proofs in HYBRID may be written down in a way that is similar to what one would see in conventional syntax with names for free and bound variables (though such issues are not of direct concern in this paper).

There are other presentations of de Bruijn terms. Pure de Bruijn has been studied extensively in (Huet, 1994; Nipkow, 2001; Shankar, 1988). In this notation, indices solely of the form $\mathsf{var}(i)$ are used to stand for both free and bound variables. A problem with this approach is that any particular pure de Bruijn expression could represent a family of λ-expressions. The *pure de Bruijn* expression $\mathsf{abs}(\mathsf{abs}(\mathsf{V3}\ \$\ \mathsf{V0}))$ could be a representation of any λ-expression like this $\lambda\, v_i.\, \lambda\, v_k.\, v_j\ v_k$ where $i \neq j$. The point is that the value j for the *free* variable is obviously not determined uniquely from informal working. Thus any machine formalization must deal not only with tracking bound indices, but also there has to be a fixed enumeration of free variables that are used to specify the actual values of indices like j. One is led to complex operations on indices that appear in both statements and proofs. In particular, substitution is a painful beast. HYBRID's locally nameless de Bruijn expressions go some way to alleviating these problems.

However, in order to consider a formal correspondence with λ-expressions we do have to introduce a notion of *proper* de Bruijn expressions (one does not need to do this when using pure de Bruijn). Recall that $\mathcal{DB}(l)$ is the set of de Bruijn expressions at level l. It follows that

$$\mathcal{PDB} = \mathcal{DB}(0) \subseteq \mathcal{DB}(1) \subseteq \ldots \subseteq \mathcal{DB}(l) \subseteq \ldots$$

and it is easy to see that $\mathcal{DB} = \bigcup_{l<\omega} \mathcal{DB}(l)$ by considering minimum levels. Let $\mathcal{PDB} \stackrel{\text{def}}{=} \mathcal{DB}(0)$ be the set of **proper** de Bruijn expressions. A proper expression is one that has no dangling indices (this follows from the formal definition). Such a proper expression corresponds to a λ-expression. We will formulate the correspondence in detail in this paper, since it forms a key part of our adequacy proof.

We also set up a notation for the traditional λ-calculus. The expressions will consist of constants, variables, applications and abstractions. More precisely, we have a countable set of variables, with a typical variable denoted by v_k where $k \geq 0$, that is $k \in \mathbb{N}$. The expressions are inductively defined by the grammar

$$E ::= \nu \mid v_k \mid \lambda\, v_k.\, E \mid E\ E$$

We adopt the usual notions of **free** and **bound** variables, and α-equivalence. For completeness we outline our notation. If v_k occurs in E then we write $v_k \in E$; we omit the usual definition of **occurs in**. We write $fv(E)$ for the set of variables occurring freely in E. In abstractions of the form $\lambda\, v_k.\, E$, we refer to the occurrence of v_k immediately after the binder λ as a **binding** occurrence, and any free occurrences of v_k in E are **bound** in $\lambda\, v_k.\, E$. We sometimes call E the **scope** of the abstraction, and in general any variable $v_{k'}$ occurring in any E' is **bound** if it occurs in a sub-expression either as a binding occurrence, or within the scope of a binding occurrence of $v_{k'}$. Given expressions E and E', and a variable v_k, then we write $E[E'/v_k]$ for a *unique* expression which, informally, is E with free occurrences of v_k replaced by E', with re-naming to avoid capture. Our definition appears on page 43, and it ensures that the action

$$(E, E', v_k) \mapsto E[E'/v_k]$$

really is a function. We say that v_w is **fresh** for E if the variable has *no occurrences* in the expression. Having defined substitution, we can then define α-equivalence. We write $\mathcal{LE}$ for the set of all expressions. If expressions E and E' are α-equivalent, we write $E \sim_\alpha E'$. In this paper, α-equivalence is an inductively defined subset of $\mathcal{LE} \times \mathcal{LE}$ generated by formal axioms and rules. There is a single axiom of the form $\lambda\, v_k.\, E \sim_\alpha \lambda\, v_{k'}.\, E[v_{k'}/v_k]$ where $k \neq k'$ and $v_{k'}$ is any variable for which $v_{k'} \notin fv(E)$; structural congruence rules for application and abstraction; plus the usual rules for equivalence relations. We write $[E]_\alpha$ for the α-equivalence class of E and $\mathcal{LE}/\!\sim_\alpha$ for the set of α-equivalence classes of expressions. For this paper we will need a notion of substitution on $\mathcal{LE}/\!\sim_\alpha$, analogous to Proposition 1.1.

Proposition B.1. *Let Var be the set of variables. There is a well-defined function*

$$\mathcal{LE}/\!\sim_\alpha \times\, \mathcal{LE}/\!\sim_\alpha \times\, Var \to \mathcal{LE}/\!\sim_\alpha \qquad ([E]_\alpha, [E']_\alpha, v_i) \mapsto [E[E'/v_i]]_\alpha$$

Proof. This is an immediate consequence of Lemma B.5. Note that the lemma makes use of simultaneous substitutions. It is necessary to define such a notion in order to complete the proofs presented in the appendix; proofs by induction over the structure of λ-expressions involve α-renaming, and simultaneous substitution is required for the provision of suitably strong inductive hypotheses. □

B.2. *Substitution for λ-expressions*

We noted in the proof of Proposition B.1 that a notion of simultaneous substitution is required in order to carry out proofs of its subsidiary lemmas. Further, because the function $[\![-]\!]_L$ involves *arbitrary* lists L, we require a definition which mirrors this, so that it interacts well with the full machinery of the paper, and other lemmas in this appendix.

Let L and $L^{\mathcal{LE}}$ be lists of equal length, where L is a list of variables v_k as usual, and

Let *Var* be the set of variables. There is a well-defined function

$$\mathcal{LE} \times \mathcal{LE} \times \mathit{Var} \to \mathcal{LE} \qquad (E, E', v_i) \mapsto E[E'/v_i]$$

$$\nu[L^{\mathcal{LE}}/L] \stackrel{\text{def}}{=} \nu$$

$$v_k[L^{\mathcal{LE}}/L] \stackrel{\text{def}}{=} \begin{cases} \mathit{elt}\,(\mathit{pos}\,v_k\,L)\,L^{\mathcal{LE}} \text{ if } v_k \in L \\ v_k \text{ if } v_k \notin L \end{cases}$$

$$(E_1\ E_2)[L^{\mathcal{LE}}/L] \stackrel{\text{def}}{=} (E_1[L^{\mathcal{LE}}/L])\ (E_2[L^{\mathcal{LE}}/L])$$

$$(\lambda\, v_k.\, E)[L^{\mathcal{LE}}/L] \stackrel{\text{def}}{=} \begin{cases} \lambda\, v_k.\, E[\overline{L^{\mathcal{LE}}}/\overline{L}] \\ \qquad \text{if } (\forall X \in \overline{L}) \begin{pmatrix} X{\downarrow} \vee X \notin \mathit{fv}(E) \\ \vee \\ (v_k \notin \mathit{fv}(\mathit{elt}\,(\mathit{pos}\,X\,\overline{L})\,\overline{L^{\mathcal{LE}}})) \end{pmatrix} \\ \lambda\, v_w.\, E[v_w/v_k][\overline{L^{\mathcal{LE}}}/\overline{L}] \\ \qquad \text{if } (\exists X \in \overline{L}) \begin{pmatrix} X{\uparrow} \wedge X \in \mathit{fv}(E) \\ \wedge \\ v_k \in \mathit{fv}(\mathit{elt}\,(\mathit{pos}\,X\,\overline{L})\,\overline{L^{\mathcal{LE}}}) \end{pmatrix} \end{cases}$$

where

— w is chosen to be the maximum of the indices occurring E, $L^{\mathcal{LE}}$ and L, plus 1; note that as $v_k \in \mathit{fv}(\mathit{elt}\,(\mathit{pos}\,X\,\overline{L})\,\overline{L^{\mathcal{LE}}})$ holds in the clause involving w, then $w > k$; and

— given lists $L^{\mathcal{LE}}$ and L of equal length, $\overline{L^{\mathcal{LE}}}$ and $\overline{L}$ are the same lists in which any occurrences of v_k in L together with their mates in $L^{\mathcal{LE}}$ are removed.

Fig. 8. Simultaneous Substitution

$L^{\mathcal{LE}}$ is a list of $\mathcal{LE}$ expressions. Suppose that $E' \in L^{\mathcal{LE}}$ and $v_k \in L$ both occur at some position p. Then we shall call the expression and variable **mates**, and say that one is the **mate** of the other. If $v_k \in L$, then we refer to the first occurrence as **active**, written $v_k{\uparrow}$. Any other occurrences are referred to as **inactive**, written $v_k{\downarrow}$. We write $E[L^{\mathcal{LE}}/L]$ for, informally, the simultaneous capture avoiding substitution of each expression $E' \in L^{\mathcal{LE}}$ for free occurrences in E of its mate in L. The definition of the simpler $E[E'/v_k]$ is then immediate. Note that some free variables in E may have multiple occurrences in L; if so, the expression in $L^{\mathcal{LE}}$ which is the mate of the active occurrence is the one that is substituted—see the formal definition below. For example $(v_1\ v_2)[E_6, E_5, E_8/v_1, v_2, v_1] = E_6\ E_5$.

We want to define such substitutions to be functions on syntax. This will give us a clean and direct definition of α-equivalence. We have to take great care with the definition of capture avoiding substitution on abstractions where a re-naming takes place, in particular with the choice of the re-naming variable. The definition is in Figure 8. The reader may wonder if our proofs can be simplified by defining substitution on abstrac-

tions so that renaming *always* takes place; this would appear to eliminate the cases which one finds in Figure 8. However, although this reduces case analyses, one finds that the extra burden of always renaming is quite significant, and can really add clutter to the many substitutions under abstractions that one might otherwise perform without such renaming.

Lemma B.2. *Suppose that $E \sim_\alpha E'$ for any expressions E and E'. Then for any lists L and $L^{\mathcal{LE}}$,*

$$E[L^{\mathcal{LE}}/L] \sim_\alpha E'[L^{\mathcal{LE}}/L]$$

Proof. The proof is by induction over $\sim_\alpha$. A full proof is, however, surprisingly tricky. In fact many authors gloss over the details, and even suggest that the induction is routine. The proof can only be regarded as routine once a number of other small results have been proven, each one formalizing a fact about properties of simultaneous substitution. Moreover, the "proofs" of each of the results referred to requires the "other" results in its own proof. The upshot is that they must all be proven by induction at the same time, with the proof of each result calling the inductive hypotheses of the others. We collect together these required results in Lemma B.3 and Lemma B.4. □

Lemma B.3. *In this lemma, we will regard lists of variable L as lists of (simple) expressions $L^{\mathcal{LE}}$. Suppose that*

$$\begin{aligned}\Phi(E) \stackrel{\text{def}}{=}\ & (\forall L_1, L_1', L_2, L_2') \\ & (L_1 \cap L_2 = \varnothing \wedge L_1 \cap L_2' = \varnothing \Longrightarrow \\ & \qquad E[L_1'/L_1][L_2'/L_2] \sim_\alpha E[L_2'/L_2][L_1'[L_2'/L_2]/L_1])\end{aligned}$$

and

$$\begin{aligned}\Phi(E) \stackrel{\text{def}}{=}\ & (\forall L, L', M, M')(\forall v_k) \\ & ((\forall X \in L(X{\downarrow} \vee X \notin fvE \vee \mathsf{mate}(X) \neq v_k)) \Longrightarrow \\ & \qquad E[L'[M'/M]/L] \sim_\alpha E[L'[\overline{M'}/\overline{M}]/L])\end{aligned}$$

and

$$\Theta(E) \stackrel{\text{def}}{=} (\forall L, L')(\forall v_k)(v_k \notin fvE \Longrightarrow E[L'/L] \sim_\alpha E[\overline{L'}/\overline{L}])$$

Then for all $E \in \mathcal{LE}$ we have $\Phi(E) \wedge \Psi(E) \wedge \Theta(E)$

Proof. The proof is a very tedious strong induction over the size of expressions and is omitted. In verifying, for example, an inductive step for $\Phi(E)$, one typically not only requires inductive hypotheses $\Phi(E')$ but also $\Psi(E')$ and $\Theta(E')$. The three conjuncts cannot be proven independently. □

Lemma B.4. *For any $E \in \mathcal{LE}$, any L_1 and $L_1^{\mathcal{LE}}$ of equal length, and L_2 and $L_2^{\mathcal{LE}}$ of equal length, such that no free variable in $L_1^{\mathcal{LE}}$ occurs in L_2, then*

$$E[L_1^{\mathcal{LE}}/L_1][L_2^{\mathcal{LE}}/L_2] \sim_\alpha E[L_1^{\mathcal{LE}}, L_2^{\mathcal{LE}}/L_1, L_2]$$

Proof. The proof is by induction over the size of E. We omit the proof, but remark that Lemma B.3 is crucial. □

Lemma B.5. *The simultaneous substitution function for $\mathcal{LE}$ (Figure 8) can be extended to $\mathcal{LE}/\sim_\alpha$. More precisely there is a well defined function specified by*

$$([E]_\alpha, [L^{\mathcal{LE}}]_\alpha, L) \mapsto [E[L^{\mathcal{LE}}/L]]_\alpha$$

where $[L^{\mathcal{LE}}]_\alpha$ means a list of α-equivalence classes of expressions.

Proof. The idea is to combine Lemma B.2 with the fact (provable by induction) that if $L_1^{\mathcal{LE}}$ and $L_2^{\mathcal{LE}}$ are two lists of equal length and consisting of pairwize α-equivalent expressions, then the function $-[L_1^{\mathcal{LE}}/+]$ equals $-[L_2^{\mathcal{LE}}/+]$. □

Appendix C. Proofs of Propositions for de Bruijn Adequacy

C.1. *The Propositions*

This section contains the proofs of the propositions outlined in Section 4.2. The proofs themselves refer to subsidiary lemmas that are stated and proved in Section C.2.

Proposition C.1. *[Defining $[\![-]\!]_L$] For any L, there is a function*

$$[\![-]\!]_L : \mathcal{LE} \to \mathcal{DB}(|L|)$$

satisfying the recursion equations below; in particular, $[\![-]\!]_\epsilon : \mathcal{LE} \to \mathcal{PDB}$.

— $[\![\nu]\!]_L = \mathsf{con}(\nu)$
— *On variables we have*

$$[\![v_i]\!]_L \quad = \quad \begin{cases} \mathsf{bnd}(pos\, v_i\, L) & \text{if } v_i \in L \\ \mathsf{var}(i) & \text{if } v_i \notin L \end{cases}$$

where $pos\, v_i\, L$ is the position of v_i in L.
— $[\![E_1\ E_2]\!]_L = [\![E_1]\!]_L\ \$\ [\![E_2]\!]_L$
— $[\![\lambda\, v_i.\, E]\!]_L = \mathsf{abs}([\![E]\!]_{v_i,L})$

Proof of Proposition 4.2=C.1 One first proves by induction on E,

$$(\forall E \in \mathcal{LE})(\forall L)([\![E]\!]_L \in \mathcal{DB})$$

(which is virtually immediate) so that uses of *level* type check, and then one can prove

$$(\forall E \in \mathcal{LE})(\forall L)(level\ |L|\ [\![E]\!]_L).$$

We give details of the proof. Inductive cases are indicated using a $\boxed{\ }$, and within each case L is an arbitrary list. Details for constants and applications are easy.

$\boxed{v_i}$ If $v_i \notin L$, which includes the case when L is empty,

$$\mathit{level}\ |L|\ [\![v_i]\!]_L = \mathit{level}\ |L|\ \mathsf{var}(i) = T$$

If $v_i \in L$,

$$\mathit{level}\ |L|\ [\![v_i]\!]_L = \mathit{level}\ |L|\ (\mathit{pos}\, v_i\, L) = (\mathit{pos}\, v_i\, L) < |L| = T$$

$\boxed{\lambda\, v_i.\, E}$

$$\mathit{level}\ |L|\ [\![\lambda\, v_i.\, E]\!]_L = \mathit{level}\ |L|\ \mathsf{abs}([\![E]\!]_{v_i,L}) = \mathit{level}\ (|L|+1)\ [\![E]\!]_{v_i,L} = T$$

with the final equality holding by induction. □

Proposition C.2. *[Defining $(\!|-|\!)_L$] For any ordered L, there is a function*

$$(\!|-|\!)_L : \mathcal{DB}(|L|) \to \mathcal{LE}$$

satisfying the recursion equations below; in particular, $(\!|-|\!)_\epsilon : \mathcal{PDB} \to \mathcal{LE}$.

— $(\!|\mathsf{con}(\nu)|\!)_L = \nu$
— $(\!|\mathsf{var}(i)|\!)_L = v_i$
— $(\!|\mathsf{bnd}(j)|\!)_L = \mathit{elt}\, j\, L$ *where* $\mathit{elt}\, j\, L$ *is the* j*th element of* L
— $(\!|D_1\ \$\ D_2|\!)_L = (\!|D_1|\!)_L\ (\!|D_2|\!)_L$
— $(\!|\mathsf{abs}(D)|\!)_L = \lambda\, v_{M+1}.\, (\!|D|\!)_{v_{M+1},L}$ *where* $M = \mathsf{Max}(D; L)$ *with*

$$\mathsf{Max}(D; L) \stackrel{\text{def}}{=} \mathsf{Max}\ \{i \mid \mathsf{var}(i) \in D\} \bigcup \underbrace{\{j \mid \mathsf{head}(L) = v_j\}}_{\varnothing \text{ if } L \text{ empty}}$$

We take $\mathsf{Max}\ \varnothing \stackrel{\text{def}}{=} 0$. *Informally* $\mathsf{Max}(D; L)$ *denotes the maximum of the free indices which occur in* D *and the indices of* L.

Proof of Proposition 4.3=C.2 One proves by induction on D

$$(\forall D \in \mathcal{DB})(\forall \text{ ordered } L)(\mathit{level}\ |L|\ D \Longrightarrow (\!|D|\!)_L \in \mathcal{LE})$$

The easy details for constants, free indices and applications are omitted.

$\boxed{\mathsf{bnd}(j)}$ If $\mathit{level}\ |L|\ \mathsf{bnd}(j)$ then $0 \le j < |L|$ so that $L \neq \epsilon$. Hence $(\!|\mathsf{bnd}(j)|\!)_L = \mathit{elt}\, j\, L$ is *defined* and hence exists in $\mathcal{LE}$.

$\boxed{\mathsf{abs}(D)}$ Note that $\mathit{level}\ |L|\ \mathsf{abs}(D) = \mathit{level}\ (|L|+1)\ D$. Hence by induction, for any ordered list L', $(\!|D|\!)_{L'} \in \mathcal{LE}$ if $|L'| = |L|+1$. If $M = \mathsf{Max}(D; L)$, then v_{M+1}, L is ordered. Hence $(\!|D|\!)_{v_{M+1},L}$ is in $\mathcal{LE}$, and thus so is $\lambda\, v_{M+1}.\, (\!|D|\!)_{v_{M+1},L}$.

Note that the choice of M in $\lambda\, v_{M+1}.\, (\!|D|\!)_{v_{M+1},L}$ ensures that v_{M+1}, L is ordered, so the recursive definition makes sense, and moreover the binding variable is chosen so that when free indices $\mathsf{var}(i)$ in D are mapped recursively to λ-calculus variables v_i which are in the scope of the binding variable v_{M+1}, they will not be (accidently) captured, as $M + 1 > i$. Here is an example which should be checked as an exercise.

$$(\!|\mathsf{abs}(\mathsf{abs}(\mathsf{bnd}(0))\ \$\ \mathsf{abs}(\mathsf{bnd}(3))\ \$\ \mathsf{var}(8))|\!)_{v_7,v_6} = \lambda\, v_9.\, (\lambda\, v_{10}.\, v_{10})\ (\lambda\, v_{10}.\, v_6)\ v_8$$

□

Proposition C.3. *[$[\![-]\!]_L$ preserves α-equivalence] For any L, if $E \sim_\alpha E'$ then $[\![E]\!]_L = [\![E']\!]_L$. In particular $[\![E]\!]_\epsilon = [\![E']\!]_\epsilon$.*

Proof of Proposition C.3 By induction on the axioms and rules defining alpha equivalence, one proves

$$(\forall (E, E') \in \sim_\alpha)(\forall L)([\![E]\!]_L = [\![E']\!]_L)$$

The only difficult part concerns the axiom $\lambda\, v_k.\, E \sim_\alpha \lambda\, v_{k'}.\, E[v_{k'}/v_k]$ in which $v_{k'}$ is chosen so that it is not free in E. We have

$$[\![\lambda\, v_k.\, E]\!]_L \stackrel{\text{def}}{=} \mathsf{abs}([\![E]\!]_{v_k, L}) = \mathsf{abs}([\![E[v_{k'}/v_k]]\!]_{v_{k'}, L}) \stackrel{\text{def}}{=} [\![\lambda\, v_k.\, E[v_{k'}/v_k]]\!]_L$$

The equality follows by appealing to Lemma C.11, with $L' = \epsilon$ so that (trivially) v_k and $v_{k'}$ are not in L'. □

Proposition C.4. *[The Identity $[\![-]\!]_L \circ (\!|-|\!)_L$] Let $D \in \mathcal{DB}$, and L be any ordered list such that for all $v_k \in L$ if any, $k \geq \mathsf{Max}(D; \epsilon) + 1$. Then*

$$\mathit{level}\ |L|\ D \Longrightarrow [\![(\!|D|\!)_L]\!]_L = D$$

Proof of Proposition C.4 A straightforward induction over $D \in \mathcal{DB}$. Constants are trivial. Note that the other two base cases make crucial use of the assumptions in the proposition.

$\boxed{\mathsf{var}(i)}$ We have $[\![(\!|\mathsf{var}(i)|\!)_L]\!]_L = [\![v_i]\!]_L = \mathsf{var}(i)$ for all L, for if $v_i \in L$ then $i \geq \mathsf{Max}(\mathsf{var}(i); \epsilon) + 1 = i + 1$, a contradiction.

$\boxed{\mathsf{bnd}(j)}$ We have

$$[\![(\!|\mathsf{bnd}(j)|\!)_L]\!]_L = \mathsf{bnd}(\mathit{pos}\ (\mathit{elt}\ j\ L)\ L) = \mathsf{bnd}(j)$$

because (crucially) L is ordered. The details for the two inductive cases, abstraction and application, are easy and omitted. □

Proposition C.5. *[The Identity $(\!|-|\!)_L \circ [\![-]\!]_L$] Let $E \in \mathcal{LE}$, and let L and L' be lists, with L' ordered, such that $|L| = |L'|$. Then*

$$(\!|[\![E]\!]_L|\!)_{L'} \sim_\alpha E[L'/L]$$

Proof of Proposition C.5

We apply induction over $\mathcal{LE}$. As ever, constants and applications are trivial.

$\boxed{v_i}$

If $v_i \notin L$, then $(\!|[\![v_i]\!]_L|\!)_{L'} \stackrel{\text{def}}{=} (\!|\mathsf{var}(i)|\!)_{L'} \stackrel{\text{def}}{=} v_i = v_i[L'/L]$.

If $v_i \in L$, then

$$(\!|[\![v_i]\!]_L|\!)_{L'} \stackrel{\text{def}}{=} (\!|\mathit{pos}\, v_i\, L|\!)_{L'} \stackrel{\text{def}}{=} \mathit{elt}\ (\mathit{pos}\, v_i\, L)\ L' \stackrel{\text{def}}{=} v_i[L'/L]$$

$\boxed{\lambda\, v_i.\, E}$

$$
\begin{aligned}
(\!|[\![\lambda\, v_i.\, E]\!]_L|\!)_{L'} &\stackrel{\text{def}}{=} (\!|\mathsf{abs}([\![E]\!]_{v_i,L})|\!)_{L'} && (14)\\
&\sim_\alpha \lambda\, v_w.\, (\!|[\![E]\!]_{v_i,L}|\!)_{v_w,L'} && (15)\\
&\sim_\alpha \lambda\, v_w.\, E[v_w, L'/v_i, L] && (16)\\
&\sim_\alpha \lambda\, v_w.\, E[v_w/v_i][L'/L] && (17)\\
&= (\lambda\, v_w.\, E[v_w/v_i])[L'/L] && (18)\\
&\sim_\alpha (\lambda\, v_i.\, E)[L'/L] && (19)
\end{aligned}
$$

Equivalence (15) holds by appeal to Corollary C.9 where w is also chosen large enough to be fresh for E, L, L' and v_i. Equivalence (16) holds by induction, noting that v_w, L' is indeed ordered. Equivalence (17) holds by appeal to Lemma B.4. Equation (18) follows from the definition of substitution; note that the choice of w ensures that there is no deletion of mate pairs. The final step (19) follows using the axiom of α-equivalence, and Proposition B.2.

□

We can now prove that the translation functions $[\![-]\!]_L$ are compositional homomorphisms.

Proposition C.6. *$[\![-]\!]_L$ Compositional Homomorphism] For any expressions $E, E' \in \mathcal{LE}$, list L, and variable v_k, if $v_k \notin L$ and $fv(E') \cap L = \varnothing$, then*

$$[\![E[E'/v_k]]\!]_L = [\![E]\!]_L[[\![E']\!]_L/\mathsf{var}(k)] \qquad (*)$$

Proof of Proposition C.6 The substitution functions exist by appeal to Proposition 1.1 and Proposition B.1. The proof proceeds by induction over the size of the expression E. Write $size(E)$ for the size of E where constants and variables have size 1, the size of an application is the sum of the sizes of the two subterms, and the size of an abstraction is the size of the body plus 1. Write $\Phi(E)$ for $(*)$ in which E', L, and k are universally quantified and satisfy the given constraints. Write $\Psi(n)$ for $(\forall E)(size(E) = n \Longrightarrow \Phi(E))$ and we prove $\forall n.\Psi(n)$ by induction on n. We write *LHS* and *RHS* for the appropriate instance of $(*)$ in the inductive steps below.

$\boxed{\Psi(1)}$ If E is a constant the result is immediate. Else choose E to be v_i, of size 1, and prove $\Phi(v_i)$.

(Case $i = k$):

$$LHS = [\![E']\!]_L = \mathsf{var}(i)[[\![E']\!]_L/\mathsf{var}(k)] = [\![v_i]\!]_L[[\![E']\!]_L/\mathsf{var}(k)] = RHS$$

with the third equality following because $v_i = v_k \notin L$.

(Case $i \neq k$):

$$LHS = [\![v_i]\!]_L = [\![v_i]\!]_L[[\![E']\!]_L/\mathsf{var}(k)] = RHS$$

where if $v_i \in L$ the second equality is immediate; and if not, the equality follows because $i \neq k$.

$\boxed{(\forall n)[(\forall m < n)(\Psi(m)) \Longrightarrow \Psi(n)]}$ where $n \geq 2$. Consider the case when the expression is $\lambda\, v_i.\, E$ and $size(\lambda\, v_i.\, E) = n$. We prove $\Phi(\lambda\, v_i.\, E)$.

(Case $i = k$):

$$LHS = [\![\lambda\, v_i.\, E]\!]_L = \mathsf{abs}([\![E]\!]_{v_i,L}) = \\ \mathsf{abs}([\![E]\!]_{v_i,L}[[\![E']\!]_L/\mathsf{var}(k)]) = \mathsf{abs}([\![E]\!]_{v_i,L})[[\![E']\!]_L/\mathsf{var}(k)] = RHS$$

where the third equality follows from Lemma C.12 since $v_k = v_i \in v_i, L$ and so $\mathsf{var}(k) = \mathsf{var}(i) \notin [\![E]\!]_{v_i,L}$.

(Case $i \neq k$): We examine sub-cases according to whether the substitution involves a name clash or not.

(Subcase $v_k \notin fv(\lambda\, v_i.\, E)$ or $v_i \notin fv(E')$): If $v_k \notin fv(\lambda\, v_i.\, E)$ we have

$$LHS = [\![\lambda\, v_i.\, E]\!]_L = \mathsf{abs}([\![E]\!]_{v_i,L}) = \\ \mathsf{abs}([\![E]\!]_{v_i,L}[[\![E']\!]_L/\mathsf{var}(k)]) = \mathsf{abs}([\![E]\!]_{v_i,L})[[\![E']\!]_L/\mathsf{var}(k)] = RHS$$

where the third equality follows from Lemma C.12 because $v_k \notin fv(\lambda\, v_i.\, E)$ and $i \neq k$ imply $v_k \notin fv(E)$.

If $v_i \notin fv(E')$

$$\begin{aligned} LHS &= [\![\lambda\, v_i.\, E[E'/v_k]]\!]_L && (20)\\ &= \mathsf{abs}([\![E[E'/v_k]]\!]_{v_i,L}) && (21)\\ &= \mathsf{abs}([\![E]\!]_{v_i,L}[[\![E']\!]_{v_i,L}/\mathsf{var}(k)]) && (22)\\ &= \mathsf{abs}([\![E]\!]_{v_i,L}[[\![E']\!]_L/\mathsf{var}(k)]) && (23)\\ &= RHS && (24) \end{aligned}$$

where equality (22) follows by induction as $size(E) = n - 1$ and so $\Phi(E)$ holds, and $v_k \notin v_i, L$ and $fv(E') \cap (v_i, L) = \varnothing$; and equality (23) follows from an instance of Lemma C.13 in which $L' = \epsilon$ and again $fv(E') \cap (v_i, L) = \varnothing$.

(Subcase $v_k \in fv(\lambda\, v_i.\, E)$ and $v_i \in fv(E')$):

$$\begin{aligned} LHS &= [\![\lambda\, v_w.\, E[v_w/v_i][E'/v_k]]\!]_L && (25)\\ &= \mathsf{abs}([\![E[v_w/v_i][E'/v_k]]\!]_{v_w,L}) && (26)\\ &= \mathsf{abs}([\![E[v_w/v_i]]\!]_{v_w,L}[[\![E']\!]_{v_w,L}/\mathsf{var}(k)]) && (27)\\ &= \mathsf{abs}([\![E]\!]_{v_i,L}[[\![E']\!]_L/\mathsf{var}(k)]) && (28)\\ &= RHS && (29) \end{aligned}$$

where equality (27) follows by induction since

$$size(E[v_w/v_i]) = size(E) = n - 1$$

and further w is the maximum of the indices in E, E' and v_k, plus 1; and equality (28) follows by appeal to Lemma C.11 and Lemma C.13. The details for applications are easy and omitted. □

C.2. *Lemmas for the Propositions*

We present a series of lemmas that are required for the proofs of the propositions in Section C.1. They are presented without detailed motivation and explanation, since the main results can be understood conceptually without a deep understanding of the lemmas. Lemmas C.9, C.11, C.12, and C.13 are used in the proofs of the propositions found in Section C.1. Lemma C.9 depends on Lemmas C.7 and C.8. Lemma C.8 depends on Lemma C.7, and Lemma C.11 on Lemma C.10.

In the remainder of this section, in any $E[L'/L]$ the list L' will in fact be a list of variables.

Lemma C.7. *Let $D \in \mathcal{DB}(|L|)$ be any expression, with L any ordered list. Suppose also that $k \geq \mathsf{Max}(D; L) + 1$. Then v_k is not free in $(\!|D|\!)_L$.*

Proof. We prove this by induction over $\mathcal{DB}$. Constants and applications are trivial.
$\boxed{\mathsf{var}(i)}$ $(\!|\mathsf{var}(i)|\!)_L = v_i$. If $k \geq \mathsf{Max}(\mathsf{var}(i); L) + 1$ then $k > i$ and we are done.
$\boxed{\mathsf{bnd}(j)}$ $(\!|\mathsf{bnd}(j)|\!)_L = \mathit{elt}\, j\, L$. Similar to previous case, k is strictly greater than any index in L.
$\boxed{\mathsf{abs}(D)}$ Pick $k \geq \mathsf{Max}(\mathsf{abs}(D); L) + 1 = \mathsf{Max}(D; L) + 1$. Note that

$$(\!|\mathsf{abs}(D)|\!)_L \stackrel{\text{def}}{=} \lambda\, v_{M+1}.\, (\!|D|\!)_{v_{M+1},L}$$

where $M \stackrel{\text{def}}{=} \mathsf{Max}(D; L)$. Thus $k \geq M + 1 = \mathsf{Max}(D; v_{M+1}, L)$. If $k = M + 1$ then v_k is not free in $(\!|\mathsf{abs}(D)|\!)_L$, as any free occurrence will be captured. If $k > M + 1$ then $k \geq \mathsf{Max}(D; v_{M+1}, L) + 1$ and so by induction v_k is not free in $(\!|D|\!)_{v_{M+1},L}$ and so we are done.

□

Lemma C.8. *Let L', L and $\hat{L}, L$ be ordered lists, with $|L'| = |\hat{L}| \geq 1$. Let $D \in \mathcal{DB}(|\hat{L}, L|)$. Then*

$$(\!|D|\!)_{\hat{L},L}[L'/\hat{L}] \sim_\alpha (\!|D|\!)_{L',L}$$

whenever

$$\mathsf{Min}\{k \mid \exists v_k \in L'\} \geq \mathsf{Max}(D; \hat{L}, L) + \#\mathsf{Abs}(D) + 1 \qquad (*)$$

where $\#\mathsf{Abs}(D)$ is the number of "abstraction" nodes in D; and

$$\mathsf{Min}\{k \mid \exists v_k \in \hat{L}\} \geq \mathsf{Max}(D; \epsilon) + 1 \qquad (**)$$

Proof.
The proof is by induction over $\mathcal{DB}$. Constants and applications are trivial.
$\boxed{\mathsf{bnd}(j)}$ We have

$$(\!|\mathsf{bnd}(j)|\!)_{\hat{L},L}[L'/\hat{L}] \stackrel{\text{def}}{=} (\mathit{elt}\, j\, (\hat{L}, L))[L'/\hat{L}] = \mathit{elt}\, j\, (L', L) \stackrel{\text{def}}{=} (\!|\mathsf{bnd}(j)|\!)_{L',L}$$

where each step follows from the definitions, and the second equality holds because L', L and $\hat{L}, L$ are ordered and $|\hat{L}| = |L'|$.

$\boxed{\mathsf{var}(i)}$

$$(\!|\mathsf{var}(i)|\!)_{\hat{L},L}[L'/\hat{L}] \stackrel{\text{def}}{=} v_i[L'/\hat{L}] = v_i \stackrel{\text{def}}{=} (\!|\mathsf{var}(i)|\!)_{L',L}$$

where the second equality holds because any index in $\hat{L}$ is greater than or equal to $\mathsf{Max}(\mathsf{var}(i);\epsilon) + 1 = i + 1 > i$ by assumption $(**)$.

$\boxed{\mathsf{abs}(D)}$ Suppose that $M \stackrel{\text{def}}{=} \mathsf{Max}(\mathsf{abs}(D); \hat{L}, L)$ and

$$\begin{array}{rcll} \mathsf{Min}\{k \mid \exists v_k \in L'\} & \geq & \mathsf{Max}(\mathsf{abs}(D); \hat{L}, L) + \#\mathsf{Abs}(\mathsf{abs}(D)) + 1 & \\ & = & M + (\#\mathsf{Abs}(D) + 1) + 1 & (i) \\ & > & M + 1 & (ii) \end{array}$$

and $\mathsf{Min}\{k \mid \exists v_k \in \hat{L}\} \geq \mathsf{Max}(\mathsf{abs}(D);\epsilon) + 1 = \mathsf{Max}(D;\epsilon) + 1 \quad (iii)$

Then we have

$$\begin{array}{rcll} (\!|\mathsf{abs}(D)|\!)_{\hat{L},L}[L'/\hat{L}] & \stackrel{\text{def}}{=} & (\lambda\, v_{M+1}.\, (\!|D|\!)_{v_{M+1},\hat{L},L})[L'/\hat{L}] & (30) \\ & = & \lambda\, v_{M+1}.\, (\!|D|\!)_{v_{M+1},\hat{L},L}[L'/\hat{L}] & (31) \\ & \sim_\alpha & \lambda\, v_{M'+1}.\, (\!|D|\!)_{v_{M+1},\hat{L},L}[L'/\hat{L}][v_{M'+1}/v_{M+1}] & (32) \\ & \sim_\alpha & \lambda\, v_{M'+1}.\, (\!|D|\!)_{v_{M+1},\hat{L},L}[L', v_{M'+1}/\hat{L}, v_{M+1}] & (33) \\ & = & \lambda\, v_{M'+1}.\, (\!|D|\!)_{v_{M+1},\hat{L},L}[v_{M'+1}, L'/v_{M+1}, \hat{L}] & (34) \\ & \sim_\alpha & \lambda\, v_{M'+1}.\, (\!|D|\!)_{v_{M'+1},L',L} & (35) \\ & \stackrel{\text{def}}{=} & (\!|\mathsf{abs}(D)|\!)_{L',L} & (36) \end{array}$$

From (ii) $v_{M+1} \notin L'$ and so the substitution in equation (30) does not involve re-naming. Further, recall that by definition, $M \stackrel{\text{def}}{=} \mathsf{Max}(\mathsf{abs}(D); \hat{L}, L)$. Hence $v_{M+1} \notin \hat{L}$ and so, recalling the definition of substitution, equation (31) holds with $\overline{L'} = L'$ and $\overline{\hat{L}} = \hat{L}$.

We set $M' \stackrel{\text{def}}{=} \mathsf{Max}(D; L', L)$ and so $M' > M + 1$ by (ii). By appeal to Lemma C.7, $v_{M'+1}$ is not free in $(\!|D|\!)_{v_{M+1},\hat{L},L}$ provided that

$$M' + 1 \geq \mathsf{Max}(D; v_{M+1}, \hat{L}, L) + 1.$$

But this holds, as $M' > M + 1$, and by (iii) and list order, we have

$$\mathsf{Max}(D; v_{M+1}, \hat{L}, L) = M + 1$$

Further, $M' + 1$ is strictly greater than the indices in L' by definition, and thus $v_{M'+1}$ is not free in $(\!|D|\!)_{v_{M+1},\hat{L},L}[L'/\hat{L}]$. By the axiom for α-equivalence, equation (32) holds.

Again, as $v_{M+1} \notin L'$ (proved above), by Lemma B.4 we have equation (33).

The equality (34) holds as we have $v_{M+1} \notin \hat{L}$ (proved above).

The equality (35) holds by induction together with the congruence of abstraction, as

the conditions of the lemma both hold as follows: Note that $(*)$ is true as

$$\begin{array}{rcl}\mathsf{Min}\{k \mid \exists v_k \in v_{M'+1}, L'\} & = & \mathsf{Min}\{k \mid \exists v_k \in L'\} \\ & \geq & (M+1) + (\#\mathsf{Abs}(D)+1) \\ & = & \mathsf{Max}(D; v_{M+1}, \hat{L}, L) + \#\mathsf{Abs}(D) + 1\end{array}$$

where the inequality holds by (i) and final equality follows from the arguments above. Further, $(**)$ is true because (iii) implies

$$\mathsf{Min}\{k \mid \exists v_k \in v_{M+1}, \hat{L}\} = \mathsf{Min}\{k \mid \exists v_k \in \hat{L}\} \geq \mathsf{Max}(D; \epsilon) + 1$$

□

Lemma C.9. *For any ordered L, and $D \in \mathcal{DB}$, there is a sufficiently large $w \geq 0$ for which*

$$(\!|\mathsf{abs}(D)|\!)_L \sim_\alpha \lambda\, v_w.\, (\!|D|\!)_{v_w, L}$$

Proof. Recall that $(\!|\mathsf{abs}(D)|\!)_L \stackrel{\text{def}}{=} \lambda\, v_{M+1}.\, (\!|D|\!)_{v_{M+1}, L}$ with $M \stackrel{\text{def}}{=} \mathsf{Max}(D; L)$. In particular v_{M+1}, L is ordered. It follows from Lemma C.8 that

$$(\!|D|\!)_{v_{M+1}, L}[v_w/v_{M+1}] \sim_\alpha (\!|D|\!)_{v_w, L} \qquad \dagger$$

provided that $(*)$ and $(**)$ hold. $(*)$ holds provided we choose

$$w \geq \mathsf{Max}(D; v_{M+1}, L) + \#\mathsf{Abs}(D) + 1$$

$(**)$ holds because

$$\mathsf{Min}\{k \mid v_k \in v_{M+1}\} = M + 1 = \mathsf{Max}(D; L) + 1 \geq \mathsf{Max}(D; \epsilon) + 1$$

Further, v_w, L is ordered. We have

$$\lambda\, v_w.\, (\!|D|\!)_{v_w, L} \sim_\alpha \lambda\, v_w.\, (\!|D|\!)_{v_{M+1}, L}[v_w/v_{M+1}]$$

by applying a congruence rule to †; and

$$\lambda\, v_w.\, (\!|D|\!)_{v_{M+1}, L}[v_w/v_{M+1}] \sim_\alpha \lambda\, v_{M+1}.\, (\!|D|\!)_{v_{M+1}, L} = (\!|\mathsf{abs}(D)|\!)_L$$

by appeal to Lemma C.7, noting $w \geq \mathsf{Max}(D; v_{M+1}, L) + 1$ implies $v_w \notin fv(\!|D|\!)_{v_{M+1}, L}$, along with an instance of the axiom for α-equivalence. □

Lemma C.10. *For any $E \in \mathcal{LE}$ and lists L and L', and any v_k, if the following conditions*

$$v_k \notin fv(E) \vee v_k \in L' \qquad (*)$$

and

$$v_{k'} \notin fv(E) \vee v_{k'} \in L' \qquad (**)$$

hold, then

$$[\![E]\!]_{L', v_k, L} = [\![E]\!]_{L', v_{k'}, L}$$

Proof. We use induction over $\mathcal{LE}$. Constants and applications are trivial.

$\boxed{v_i}$ We have to check that

$$[\![v_i]\!]_{L',v_k,L} = [\![v_i]\!]_{L',v_{k'},L}$$

This requires a case analysis. If $v_i \in L'$ we are done. Now suppose $v_i \notin L'$. Note that if $v_i = v_k$ then by condition $(*)$ we must have $v_i \notin v_i$, a contradiction. Thus $v_i \neq v_k$. A symmetric argument for $(**)$ shows that $v_i \neq v_{k'}$. Thus either $v_i \in L$ and we are done, or in fact v_i is not in *any* of the lists and both sides of the required equality are equal to $\mathsf{var}(i)$.

$\boxed{\lambda\, v_i.\, E}$ We have to check that

$$\mathsf{abs}([\![E]\!]_{v_i,L',v_k,L}) = \mathsf{abs}([\![E]\!]_{v_i,L',v_{k'},L})$$

under the assumptions

$$v_k \notin \mathit{fv}(\lambda\, v_i.\, E) \vee v_k \in L'$$

and

$$v_{k'} \notin \mathit{fv}(\lambda\, v_i.\, E) \vee v_{k'} \in L'$$

The equality will follow by induction provided that both

$$v_k \notin \mathit{fv}(E) \vee v_k \in v_i, L' \qquad (*)$$

and

$$v_{k'} \notin \mathit{fv}(E) \vee v_{k'} \in v_i, L' \qquad (**)$$

hold. In $(*)$ suppose that $v_k \notin v_i, L'$. We must then have $v_k \notin \mathit{fv}(\lambda\, v_i.\, E)$ and $v_k \neq v_i$, so $v_k \notin \mathit{fv}(E)$. Thus $(*)$ holds. The argument for $(**)$ is analogous.

□

Lemma C.11. *Let $E \in \mathcal{LE}$, let L' and L be any lists, and let v_k, $v_{k'}$ be variables for which $v_{k'} \notin \mathit{fv}(E)$, $v_k \notin L'$ and $v_{k'} \notin L'$. Then we have*

$$[\![E[v_{k'}/v_k]]\!]_{L',v_{k'},L} = [\![E]\!]_{L',v_k,L} \qquad (*)$$

Proof. We prove this result by induction on the size of the expression E, where constants and variables have size 1, the size of an application is the sum of the sizes of the two subterms, and the size of an abstraction is the size of the body plus 1. Write $\mathit{size}(E)$ for the size of E and $\Phi(E)$ for $(*)$ in which L, L', k and k' are universally quantified and satisfy the given constraints. Write $\Psi(n)$ for

$$(\forall E)(\mathit{size}(E) = n \Longrightarrow \Phi(E))$$

and we prove $\forall n.\Phi(n)$ by strong induction on n. Note (carefully!) that we have to prove $\Psi(1)$ explicitly—the base case for the induction. We write *LHS* and *RHS* for the left and right hand sides of the equality in the lemma.

$\boxed{\Psi(1)}$ Consider arbitrary expressions of size 1. If a constant the result is immediate. Otherwise we have a variable, say v_i. We have to prove $\Phi(v_i)$. Note that $v_{k'}$ must not be free in v_i, so $i \neq k'$.

(Case $i = k$): If $i = k$ then

$$LHS = [\![v_{k'}]\!]_{L',v_{k'},L} = pos\, v_{k'}\, (L', v_{k'}, L) = \\ pos\, v_i\, (L', v_k, L) = [\![v_i]\!]_{L',v_k,L} = RHS$$

where the positions are equal due to the constraints of the lemma concerning L'.

(Case $i \neq k$): We have to prove that $[\![v_i]\!]_{L',v_{k'},L} = [\![v_i]\!]_{L',v_k,L}$. If $v_i \in L'$ we are done. Now suppose $v_i \notin L'$. Note further that $k \neq i \neq k'$. Hence either $v_i \in L$ and we are done, or else $v_i \notin L$ and then $\mathsf{var}(i) \stackrel{\text{def}}{=} [\![v_i]\!]_{L',v_{k'},L} = [\![v_i]\!]_{L',v_k,L} \stackrel{\text{def}}{=} \mathsf{var}(i)$.

$\boxed{(\forall n)((\forall m < n)(\Psi(m) \Longrightarrow \Psi(n)))}$ Choose arbitrary $n \geq 2$, and suppose that $\Psi(m)$ holds for all m smaller than n. We must prove $\Psi(n)$. So consider an arbitrary expression N of size n. We have to prove that $\Phi(N)$ holds, assuming $\Phi(M)$ for all expressions M of size smaller than N.

In the case when $N \in \mathcal{LE}$ is of the form $E_1\ E_2$ the result follows by a routine inductive argument which we omit.

In the case when $N \in \mathcal{LE}$ is of the form $\lambda\, v_i.\, E$, note that we can assume that $\Phi(M)$ for any M with $size(M) < size(E) + 1$. We have to prove that

$$LHS \stackrel{\text{def}}{=} [\![(\lambda\, v_i.\, E)[v_{k'}/v_k]]\!]_{L',v_{k'},L} = [\![\lambda\, v_i.\, E]\!]_{L',v_k,L} \stackrel{\text{def}}{=} RHS$$

when $v_{k'} \notin fv(\lambda\, v_i.\, E) \stackrel{\text{def}}{=} fv(E) \setminus \{v_i\}$, and $v_k \notin L'$ and $v_{k'} \notin L'$.

(Case $i = k$): Here the variable v_k is not free in $\lambda\, v_i.\, E$ and so $LHS = \mathsf{abs}([\![E]\!]_{v_i,L',v_{k'},L})$ and further $RHS = \mathsf{abs}([\![E]\!]_{v_i,L',v_k,L})$. Equality follows from Lemma C.10—we check the assumptions of the lemma are satisfied. Note that if $i = k'$ then we are (trivially) done. So suppose $i \neq k'$. $(*)$ holds as $i = k$ implies $v_k \in v_i, L$. $(**)$ holds as $v_{k'} \notin fv(E) \setminus \{v_i\}$ and $i \neq k'$ imply $v_{k'} \notin fv(E)$.

(Case $i \neq k$): Here the variable v_k may be free in $\lambda\, v_i.\, E$.

(Subcase $v_k \notin fv(E)$ or $v_i \neq v_{k'}$): This case is when the substitution does not involve a renaming. Note that $LHS = \mathsf{abs}([\![E[v_{k'}/v_k]]\!]_{v_i,L',v_{k'},L})$ and $RHS = \mathsf{abs}([\![E]\!]_{v_i,L',v_k,L})$.

In the case that $v_i \neq v_{k'}$ then $v_{k'} \notin fv(E)$ follows using the same reasoning as in case $i = k$ above. Further, as $i \neq k$ and $i \neq k'$, $LHS = RHS$ follows inductively from $\Phi(E)$, as $size(E) < size(E) + 1$.

In the case that $v_k \notin fv(E)$ (and $i = k'$ say), then $LHS = \mathsf{abs}([\![E]\!]_{v_i,L',v_{k'},L})$. Equality follows from Lemma C.10—we check the assumptions of the lemma are satisfied. $(*)$ holds as $v_k \notin fv(E)$. $(**)$ holds as $i = k'$ implies $v_{k'} \in v_i, L$.

(Subcase $v_k \in fv(E)$ and $v_i = v_{k'}$): Here we have

$$\begin{aligned} LHS &= [\![\lambda\, v_w.\, E[v_w/v_i][v_{k'}/v_k]]\!]_{L',v_{k'},L} && (37)\\ &\stackrel{\text{def}}{=} \mathsf{abs}([\![E[v_w/v_i][v_{k'}/v_k]]\!]_{v_w,L',v_{k'},L}) && (38)\\ &= \mathsf{abs}([\![E[v_w/v_i]]\!]_{v_w,L',v_k,L}) && (39)\\ &= \mathsf{abs}([\![E]\!]_{v_i,L',v_k,L}) && (40)\\ &\stackrel{\text{def}}{=} RHS && (41) \end{aligned}$$

In equation (37), w is the maximum of the indices in E, v_k and $v_{k'}$, plus 1. Equation (39) follows from inductive hypothesis $\Phi(E[v_w/v_i])$, since

$$size(E[v_w/v_i]) = size(E) < size(E) + 1,$$

both v_k and $v_{k'}$ are not in v_w, L', and also $v_{k'} \notin fv(E[v_w/v_i])$ using the original assumption about $v_{k'}$ and the definition of w. Equation (40) follows by induction in a similar fashion. This completes the proof.

□

By examining the definition of the de Bruijn expression $[\![E]\!]_L$ it is easy to see that the expression can only have a "free variable" subterm $\mathsf{var}(k)$ if $v_k \notin L$ (for example $[\![v_1\ v_2]\!]_{v_1,v_3} = \mathsf{bnd}(0)\ \$\ \mathsf{var}(2)$). Since L enumerates binding variables as the function $[\![-]\!]_L$ descends through expressions E, then $v_k \notin L$ implies that $v_k \in fv(E)$ (consider the contrapositive example $[\![\lambda\, v_2.\, v_1\ v_2]\!]_\epsilon = \mathsf{abs}([\![v_2]\!]_{v_1\ v_2})$). Hence we have

Lemma C.12. *For any $E \in \mathcal{LE}$, list L and variable v_k, if either $v_k \notin fv(E)$ or $v_k \in L$, then $\mathsf{var}(k) \notin [\![E]\!]_L$.*

Proof. The proof is by a simple induction over $E \in \mathcal{LE}$. □

The next lemma gives conditions enabling variables to be dropped from a list. It plays a role when computing de Bruijn expressions from λ-expressions involving substitutions with re-naming, allowing the fresh variable to be eliminated at suitable points in calculations.

Lemma C.13. *Let $E \in \mathcal{LE}$, let L and L' be any lists, and v_i a variable. Further, suppose that for any $v_k \in fv(E)$, either $v_k \in L'$ or $v_k \notin v_i, L$. Then $[\![E]\!]_{L',v_i,L} = [\![E]\!]_{L',L}$.*

Proof. The proof is by a simple induction over $E \in \mathcal{LE}$. □

References

Aczel, P. (1977). An Introduction to Inductive Definitions. *Handbook of Mathematical Logic*. Latest impression 1993.

Ambler, S. J., Crole, R. L., and Momigliano, A. (2002a). A Hybrid Encoding of Howe's Method for Establishing Congruence of Bisimilarity. (Extended Abstract). *Electronic Notes in Theoretical Computer Science*, 70(2). Proceedings of the Third International Workshop on Logical Frameworks and Meta-Languages (LFM'02). A FLoC'02 affiliated workshop, Copenhagen, Denmark, July 26, 2002.

Ambler, S. J., Crole, R. L., and Momigliano, A. (2002b). 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*, pages 13–30. Springer-Verlag.

Ambler, S. J., Crole, R. L., and Momigliano, A. (2004). A Combinator and Presheaf Topos Model for Primitive Recursion over Higher Order Abstract Syntax. In *Collegium Logicum (Proceedings of the Kurt Godel Society)*, pages 83–90. Springer-Verlag. Computer Science Logic/8th Kurt Godel Colloquium Poster Collection, Vienna, August, 2003.

Anderson, P. and Pfenning, F. (2004). Verifying Uniqueness in a Logical Framework. In *Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics*, volume 3223 of *Lecture Notes in Computer Science*. Springer-Verlag.

Aydemir, B. E., Charguéraud, A., Pierce, B. C., Pollack, R., and Weirich, S. (2008). Engineering Formal Metatheory. *SIGPLAN Not.*, 43(1):3–15.

Berghofer, S. and Urban, C. (2006). A Head-to-Head Comparison of de Bruijn Indices and Names. In *Proceedings of the International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2006)*, pages 53–67, Seattle, USA.

Berhofer, S., Cheney, J., and Urban, C. (2008). Mechanizing the Metatheory of LF. In *Proc. of the 23rd IEEE Symposium on Logic in Computer Science (LICS 2008)*, pages 45–56. IEEE Computer Society.

Berhofer, S., Cheney, J., and Urban, C. (2010). Mechanizing the Metatheory of LF. Technical report, Edinburgh and Munich. ACM Transations on Computational Logic, to appear.

Capretta, V. and Felty, A. (2007). Combining de Bruijn Indices and Higher-Order Abstract Syntax in Coq. In Altenkirch, T. and McBride, C., editors, *Proceedings of TYPES 2006*, volume 4502 of *LNCS*, pages 63–77. Springer.

Capretta, V. and Felty, A. (2009). Higher Order Abstract Syntax in Type Theory. In Cooper, S. B., Geuvers, H., Pillay, A., and Väänänen, J., editors, *Logic Colloquium 2006*, volume 32 of *Lecture Notes in Logic*, pages 65–90. Cambridge University Press.

Cheney, J. (2009). A Simple Nominal Type Theory. *Electronic Notes in Computer Science*, 228:37–52.

Clouston, R. (2010). *Equational Logic for Names and Binders*. PhD thesis, University of Cambridge Computer Laboratory.

Clouston, R. A. and Pitts, A. M. (2007). Nominal equational logic. In Cardelli, L., Fiore, M., and Winskel, G., editors, *Computation, Meaning and Logic, Articles dedicated to Gordon Plotkin*, volume 1496 of *Electronic Notes in Theoretical Computer Science*, pages 223–257. Elsevier.

Crole, R. L. (1998). Lectures on [Co]Induction and [Co]Algebras. Technical Report 1998/12, Department of Mathematics and Computer Science, University of Leicester.

Crole, R. L. (2010). α-Equivalence Equalities. Submitted for journal publication.

de Bruijn, N. (1972). Lambda Calculus Notation with Nameless Dummies: a Tool for Automatic Formula Manipulation, with Application to the Church Rosser Theorem. *Indagationes Mathematicae*, 34(5):381–392.

Despeyroux, J., Felty, A., and Hirschowitz, A. (1995). Higher-order abstract syntax in Coq. In Dezani-Ciancaglini, M. and Plotkin, G., editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 124–138, Edinburgh, Scotland. Springer-Verlag LNCS 902.

Despeyroux, J. and Leleu, P. (2000). Metatheoretic results for a modal λ-calculus. *Journal of Functional and Logic Programming*, 1.

Despeyroux, J., Pfenning, F., and Schürmann, C. (1997). Primitive recursion for higher-order abstract syntax. In Hindley, R., editor, *Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, pages 147–163, Nancy, France. Springer-Verlag LNCS. An extended version is available as Technical Report CMU-CS-96-172, Carnegie Mellon University.

Felty, A. (2002a). Interactive Theorem Proving in Twelf. *The Association of Logic Programming Newsletter*, 15(3).

Felty, A. (2002b). Two-level Meta-reasoning in Coq. In Carreño, V. A., editor, *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics, Hampton, VA, 1-3 August 2002*, volume 2342 of *LNCS*. Springer Verlag.

Felty, A. and Pientka, B. (2010). Reasoning with Higher-Order Abstract Syntax and Contexts: A Comparison. In Kaufmann, M. and Paulson, L., editors, *International Conference on Interactive Theorem Proving*, volume 6172, pages 228–243. Lecture Notes in Computer Science.

Felty, A. P. and Momigliano, A. (2009). Reasoning with Hypothetical Judgments and Open Terms in Hybrid. In *Principles and Practice of Declarative Programming*, pages 83–92.

Felty, A. P. and Momigliano, A. (2010). A Definitional Two-Level Approach to Reasoning with Higher-Order Abstract Syntax. *Journal of Automated Reasoning*.

Fiore, M., Plotkin, G. D., and Turi, D. (1999). Abstract Syntax and Variable Binding. In *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 193–202. IEEE Computer Society Press.

Fiore, M. P. (2006). On the structure of substitution. Invited address for the 22nd Mathematical Foundations of Programming Semantics Conf. (MFPS XXII), DISI, University of Genova (Italy).

Ford, J. and Mason, I. A. (2001). Operational Techniques in PVS – A Preliminary Evaluation. In *Proceedings of the Australasian Theory Symposium, CATS '01*.

Gabbay, M. and Mathijssen, A. (2008). Capture-Avoiding Substitution as a Nominal Algebra (journal version). *Formal Aspects of Computing*, 20(4-5):451–479.

Gabbay, M. and Mathijssen, A. (2010). A Nominal Axiomatisation of the Lambda-Calculus. *Journal of Logic and Computation*, 20(2):501–531.

Gabbay, M. and Pitts, A. (1999). A New Approach to Abstract Syntax Involving Binders. In Longo, G., editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 214–224, Trento, Italy. IEEE Computer Society Press.

Gabbay, M. J. and Pitts, A. M. (2002). A New Approach to Abstract Syntax with Variable Binding. *Formal Aspects of Computing*, 13:341–363.

Gacek, A. (2008). The Abella Interactive Theorem Prover (System Description). In Armando, A., Baumgartner, P., and Dowek, G., editors, *Proceedings of IJCAR*, volume 5195 of *LNCS*, pages 154–161. Springer.

Gacek, A., Miller, D., and Nadathur, G. (2008). Combining Generic Judgments with Recursive Definitions. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*, pages 33–44. IEEE Computer Society.

Gordon, A. (1994). A Mechanisation of Name-Carrying Syntax up to Alpha-Conversion. In J.J. Joyce and C.-J.H. Seger, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 414–427, Vancouver, Canada. University of British Columbia, Springer-Verlag, published 1994.

Gordon, A. D. and Melham, T. (1996). Five Axioms of Alpha-Conversion. In von Wright, J., Grundy, J., and Harrison, J., editors, *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *Lecture Notes in Computer Science*, pages 173–190, Turku, Finland. Springer-Verlag.

Hallnas, L. (1991). Partial Inductive Definitions. *Theoretical Computer Science*, 87(1):115–147.

Harper, R., Honsell, F., and Plotkin, G. (1987). A Framework for Defining Logics. In *Logic in Computer Science*, pages 194–204, Ithaca, New York.

Harper, R., Honsell, F., and Plotkin, G. (1993). A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184. (See also (Harper et al., 1987).).

Harper, R. and Licata, D. R. (2007). Mechanizing Metatheory in a Logical Framework. *Journal of Functional Programming*, 17(4-5):613–673.

Harper, R. and Pfenning, F. (2005). On Equivalence and Canonical Forms in the LF Type Theory. *Transactions on Computational Logic*, 6:61–101.

Hindley, J. and Seldin, J. (1988). *Introduction to Combinators and the Lambda Calculus*, volume 1 of *London Mathematical Society Student Texts*. Cambridge University Press.

Hofmann, M. (1999). Semantical Analysis of Higher-Order Abstract Syntax. In *Proc. of 14th Ann. IEEE Symp. on Logic in Computer Science, LICS'99, Trento, Italy, 2–5 July 1999*, pages 204–213. IEEE Computer Society Press, Los Alamitos, CA.

Honsell, F., Miculan, M., and Scagnetto, I. (2001a). An axiomatic approach to metareasoning on systems in higher-order abstract syntax. In *Proc. ICALP'01*, number 2076 in LNCS, pages 963–978. Springer-Verlag.

Honsell, F., Miculan, M., and Scagnetto, I. (2001b). Π-calculus in (Co)Inductive Type Theories. *Theoretical Computer Science*, 2(253):239–285.

Honsell, F., Miculan, M., and Scagnetto, I. (2005). Translating Specifications from Nominal Logic to CIC with the Theory of Contexts. In Pollack, R., editor, *MERLIN'05*. ACM Digital Library.

Huet, G. P. (1994). Residual Theory in Lambda-Calculus: a Formal Development. *Journal of Functional Programming*, 4(3):371394.

Lakin, M. R. and Pitts, A. M. (2007). A Metalanguage for Structural Operational Semantics. In *Eighth Symposium on Trends in Functional Programming (TFP 2007), New York, USA*, pages I1–I16. Draft proceedings.

McDowell, R. (1997). *Reasoning in a Logic with Definitions and Induction*. PhD thesis, University of Pennsylvania.

McDowell, R. and Miller, D. (1997). A Logic for Reasoning with Higher-Order Abstract Syntax: An Extended Abstract. In Winskel, G., editor, *Proceedings of the Twelfth Annual Symposium on Logic in Computer Science*, pages 434–445, Warsaw, Poland.

McDowell, R. and Miller, D. (2002). Reasoning with Higher-Order Abstract Syntax in a Logical Framework. *ACM Transactions on Computational Logic*, 3(1):80–136.

McKinna, J. and Pollack, R. (1999). Some Lambda Calculus and Type Theory Formalized. *Journal of Automated Reasoning*, 23(3-4):373–409.

Melham, T. F. (1994). A Mechanized Theory of the π-Calculus in HOL. *Nordic Journal of Computing*, 1(1):50–76.

Miculan, M. (2001). Developing (Meta)Theory of Lambda-Calculus in the Theory of Contexts. In Ambler, S., Crole, R., and Momigliano, A., editors, *MERLIN 2001: Proceedings of the Workshop on MEchanized Reasoning about Languages with variable bINding*, volume 58 of *Electronic Notes in Theoretical Computer Scienc*, pages 1–22.

Miller, D. (2006). Representing and reasoning with operational semantics. In *Third International Joint Conference on Automated Reasoning*. Springer-Verlag.

Momigliano, A. and Ambler, S. J. (2003). Multi-Level Meta-Reasoning with Higher Order Abstract Syntax. In Gordon, A. D., editor, *Foundations of Software Science and Computation Structures*, volume 2620 of *Lecture Notes in Computer Science*, pages 375–391. Springer-Verlag.

Momigliano, A., Ambler, S. J., and Crole, R. L. (2001). A Comparison of Formalizations of the Meta-Theory of a Language with Variable Bindings in Isabelle. In Boulton, R. J. and

Jackson, P. B., editors, *Supplemental Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics*, pages 267–282. Report EDI-INF-RR-0046.

Momigliano, A., Martin, A. J., and Felty, A. P. (2009). Two-level Hybrid: A System for Reasoning using Higher Order Abstract Syntax. In *Proceedings of the International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2008). Volume 228 of Electronic Notes in Theoretical Computer Science*, volume 196, pages 85–93. Elsevier.

Nipkow, T. (2001). More Church-Rosser Proofs (in Isabelle/HOL). *Journal of Automated Reasoning*, 26:51–66.

Norrish, M. and Vestergaard, R. (2007). Proof Pearl: de Bruijn Terms Really Do Work. In Schneider, K. and Brandt, J., editors, *Theorem Proving in Higher Order Logics, 20th International Conference*, volume 4732 of *Lecture Notes in Computer Science*, pages 207–222. Springer.

Paulson, L. (1997). *ML for the Working Programmer*. Cambridge University Press. 2nd Edition.

Pfenning, F. (2003). Computation and deduction. To appear. Cambridge University Press.

Pfenning, F. and Elliott, C. (1988). Higher-Order Abstract Syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208.

Pfenning, F. and Schürmann, C. (1999). System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In Ganzinger, H., editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy. Springer-Verlag LNAI 1632.

Pitts, A. M. (2001). Nominal Logic: A First Order Theory of Names and Binding. In Kobayashi, N. and Pierce, B. C., editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001, Sendai, Japan, October 29-31, 2001. Proceedings*, volume 2215 of *Lecture Notes in Computer Science*, pages 219–242. Springer-Verlag, Berlin.

Pitts, A. M. (2003). Nominal Logic, A First Order Theory of Names and Binding. *Information and Computation*, 186:165–193.

Pitts, A. M. (2006). Alpha-Structural Recursion and Induction. *Journal of the ACM*, 53:459–506.

Shankar, N. (1988). A Mechanical Proof of the Church-Rosser Theorem. *ACM*, 35(3):475522.

Shinwell, M. R. and Pitts, A. M. (2005). Fresh Objective Caml User Manual. Technical Report UCAM-CL-TR-621, University of Cambridge Computer Laboratory.

Shinwell, M. R., Pitts, A. M., and Gabbay, M. J. (2003). FreshML: Programming with Binders Made Simple. In *Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden*, pages 263–274. ACM Press.

Urban, C. (2008). Nominal Techniques in Isabelle/HOL. *J. Autom. Reason.*, 40(4):327–356.

Urban, C., Pitts, A. M., and Gabbay, M. J. (2004). Nominal unification. *Theoretical Computer Science*, 323:473–497.

Urban, C. and Tasson, C. (2005). Nominal Reasoning Techniques in Isabelle/HOL. In *Proceedings of the 20th Conference on Automated Deduction (CADE 2005)*, volume 3632, pages 38–53, Tallinn, Estonia.

van Dalen, D. (1989). *Logic and Structure*. Universitext. Springer-Verlag, 3rd edition. Corrected Third Printing.

Vestergaard, R. and Brotherson, J. (2001). A Formalized First-Order Conflence Proof for the λ-Calculus using One Sorted Variable Names. In Middelrop, A., editor, *Proceedings of RTA'12*, volume 2051 of *LNCS*, pages 306–321. Springer-Verlag.