Math. Struct. in Comp. Science: page 1 of 62.
doi:10.1017/S0960129511000041

The representational adequacy of HYBRID

R. L. CROLE

Department of Computer Science, University of Leicester, University Road, Leicester, LE1 7RH, United Kingdom
Email: R.Crole@mcs.le.ac.uk

Received 11 August 2010; revised 14 December 2010

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-expression.

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. Although this paper will be of considerable interest to those who wish to work with HYBRID in Isabelle/HOL, a background in automated theorem proving is not essential.

1. Introduction

1.1. *Overview*

Many people are involved in the development of computing systems that 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, which was 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 begin with an overview of the first feature, which should provide sufficient details for an understanding of this paper. The other features have been discussed in Ambler *et al.* (2002b; 2004). We will then proceed to prove our main theorem, which is *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) and 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 the informal sense that everyone knows how to convert a de Bruijn expression to a λ-expression and *vice-versa*, proving the existence of a bijection is not 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 role in this paper.*

1.2. *A roadmap for the paper*

In the remainder of the introduction: we give an abstract definition of our notion of representational adequacy; review our 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 role, 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. Finally, in Section 8, we present our conclusions.

1.3. *Representational adequacy*

In this section, 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) and Harper and Licata (2007).

Suppose 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 if 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.

1.4. *de Bruijn expressions and λ-expressions*

We assume familiarity with de Bruijn expressions and λ-expressions, but in this section we summarise the notation we use. In particular, we wish to make clear what kind of de Bruijn expressions we will be working with in this paper. If required, more details can be found in Appendix B.

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

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

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

In order 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 $m \geqslant m' \geqslant 0$ and $k \geqslant 0$ there is a function $\mathscr{DB}(m) \times \mathscr{DB}(m') \times \mathbb{N} \to \mathscr{DB}(m)$ 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, which is one key advantage of locally nameless de Bruijn expressions. Of course, we do need to prove that the function has the stated source and target, but we will omit the easy proof here. □

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

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

and we write $\mathscr{LE}$ for the set of all expressions. Given expressions E and E', and a variable v_k, we write $E[E'/v_k]$ for a *unique* expression, which, informally, is E with free occurrences of v_k replaced by E', with renaming 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 $\mathscr{LE}/\sim_\alpha$ for the set of α-equivalence classes of expressions. For this paper we will need a notion of substitution on $\mathscr{LE}/\sim_\alpha$ analogous to Proposition 1.1.

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

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

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 provide a look-up table in Figure 1. In the remainder of the paper we will explain the role of Hybrid syntax, which is summarised in Figure 1 (along with the syntax for de Bruijn expressions and λ-expressions). The informal roles 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 the syntax 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 these two systems in this paper by showing that Hybrid is adequate for λ-expressions.

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 arise, or are hidden, by identifying the syntax for variables.

Syntax	Informal Description	Defined Informally	Defined Formally
$\mathsf{abs}(D)$	object level de Bruijn abstraction	§1.4	§B.1
D_1 \$ D_2	object level de Bruijn application	§1.4	§B.1
$\lambda v_i.\, E$	object level lambda abstraction	§1.4	§B.1
$E_1\ E_2$	object level lambda application	§1.4	§B.1
$\mathsf{ABS}\ C$	HYBRID de Bruijn abstraction	§2.1	§3
$\mathsf{LAM}\, v_i.\, C$	HYBRID lambda abstraction	§2.1	§5.2
C_1 \$\$ C_2	HYBRID de Bruijn application	§2.1	§3
$\Lambda v_i.\, C$	HYBRID meta-abstraction	§1.5	§4
$C_1\ C_2$	HYBRID meta-application	§1.5	§4

Fig. 1. Abstraction and application notation

We shall often wish to define and talk about higher order functions in this paper. Suppose 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 exactly how 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 of Andrew Gordon (Gordon 1994), which we will now 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 proved equal to a de Bruijn expression with an outer abstraction and an immediate subexpression, which is e in de Bruijn form, and 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,

$$\text{dLAMBDA v (dAPP (dVAR v) (dVAR u)) = dABS (dAPP (dBND 0) (dVAR u)).}$$

Gordon demonstrated the utility of this idea within a theorem prover. The user may work with expressions that have explicit named binding variables (such as the v above), but such expressions are automatically converted to de Bruijn form by the machine, which the machine can work with more easily. 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 that a theorem prover typically possesses. *What would be ideal is a system in which Gordon's string binders 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 LAM $v.\, C$ in HYBRID. This is, in fact, a *definition* for a de Bruijn expression; and 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 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 will now give some 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

$$E_G = \text{dLAMBDA v8 (dLAMBDA v2 (dAPP (dVAR v8) (dVAR v3)))},$$

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

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

with an overall effect 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

- 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 $\mathsf{LAM}\, v_j.\, C$.
- Object level applications $E_1\ E_2$ are expressed as HYBRID expressions $C_1\ \$\$\ C_2$.
  - For example, $\lambda v_8.\, \lambda v_2.\, v_8\ v_3$ 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†.
  - For example, $\mathsf{LAM}\, v_8.\, (\mathsf{LAM}\, v_2.\, (v_8\ \$\$\ \mathsf{VAR}\ 3))$ 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

indices, and v are names for constants:

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

Moreover, there is a specification for expressions of the form $\mathsf{LAM}\, v_i.\, C$, which are built out of Isabelle/HOL binders. Figure 2 provides a summary of the key principles of HYBRID – it may be useful to look back at the syntax summary in Figure 1.

The aim of the current paper is to show that HYBRID really does provide a representation of the λ-calculus, in the sense that:

> 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.

2.2. *Reducing named binders to nameless binders*

In formally stating the adequacy theorem, we utilise an auxiliary function *lbnd* (whose existence is proved in Proposition 3.2). This function is of central importance, so we will give some examples and an 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 \$\$.

† Strictly speaking, we should write *proper* expressions.

Consider 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 $\mathsf{LAM}\, v_i.\, \xi$ in fact 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 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 that:

— descends recursively through the ABS nodes and increases parameter n by one each time, thereby enabling computation of the bound de Bruijn indices;
— descends recursively over \$\$ nodes; and
— in each case, recursively moves the meta-binders Λ towards the bound meta-variables, and 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 $C[v_i, v_j]$ is a HYBRID expression containing the named variables v_i and v_j. Then for $i \neq j$, we would expect

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

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 $\mathsf{ABS}\ (C[\mathsf{BND}\ n, v_j])$. The following is a concrete example, featuring only an \$\$ node:

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

[†] This is not type correct since, in fact, ABS :: exp ⇒ 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.

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

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

In summary, each instance of the *lbnd* function descends recursively through its argument through 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 (for example, $\Lambda\, v_2.$) that has been 'moved' to the leaf (for example, v_8) is different (for example, $\Lambda\, v_2.\, v_8$), the leaf node remains unchanged (for example, v_8). However, if it has the same name (for example, $\Lambda\, v_8.\, v_8$), the node becomes BND n where n is the counter, and thus the original bound name becomes the correct de Bruijn bound index (for example, v_8 becomes BND 1).

2.3. *Formalising lbnd*

It is the formalisation and implementation of the function *lbnd* that lies at the heart of the HYBRID system. In the next section we will show how to develop a *model* of the full HYBRID system, and 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. Recall the informal description of

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

given at the end of Section 2.1.

We will now formally define Θ and prove it to be representationally adequate. We will take the λ-calculus to be the set $\mathscr{L}\mathscr{E}/\sim_\alpha$. We now define exactly what we will mean in the rest of this paper when we write HYBRID. 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: in particular, we would need 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, 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

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

Fig. 3. Constructor constants

with a *model of the core subset* of the system implemented in Isabelle/HOL, but 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) and Pfenning (2003), and, perhaps, Anderson and Pfenning (2004) and 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) and Pfenning (2003)). Key to this is the following convention:

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

We will 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 v 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 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

$$\mathscr{LF}_\sigma(\Gamma) \stackrel{\text{def}}{=} \{e \mid \Gamma \vdash e :: \sigma\}$$
$$\mathscr{LF} \stackrel{\text{def}}{=} \bigcup_{\sigma,\Gamma} \mathscr{LF}_\sigma(\Gamma).$$

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

$$\mathscr{CLF}_\sigma(\Gamma) \stackrel{\text{def}}{=} \{C \mid \Gamma \vdash_{can} C :: \sigma\}.$$

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

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

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

Fig. 4. Inductive definition of canonical forms

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

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

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

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

Fig. 5. Inductive definition of canonical forms through atomic forms

Given a list of meta-variables $L = v_{k_1}, \ldots, v_{k_m}$, 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 $\mathscr{CLF}_\sigma(\Gamma) \subseteq \mathscr{LF}_\sigma(\Gamma)$.

Remark 3.1. The ellipsis notation in Figure 4 is of course a short-hand for an infinite 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

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

We will now give the formal definition of *lbnd*.

Proposition 3.2 (defining *lbnd*). For all $n \geqslant 0$ and lists L, there is a unique function with the following source and target

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

satisfying the following recursion equations

$$\begin{aligned}
\mathit{lbnd}\ n\ (\Lambda v_k.\,\mathsf{CON}\ v) &= \mathsf{CON}\ v\\
\mathit{lbnd}\ n\ (\Lambda v_k.\,v_k) &= \mathsf{BND}\ n\\
\mathit{lbnd}\ n\ (\Lambda v_k.\,v_{k'}) &= v_{k'}\ \text{where}\ k \neq k'\\
\mathit{lbnd}\ n\ (\Lambda v_k.\,\mathsf{VAR}\ i) &= \mathsf{VAR}\ i\\
\mathit{lbnd}\ n\ (\Lambda v_k.\,\mathsf{BND}\ j) &= \mathsf{BND}\ j\\
\mathit{lbnd}\ n\ (\Lambda v_k.\,C_1\ \$\$\ C_2) &= (\mathit{lbnd}\ n\ (\Lambda v_k.\,C_1))\ \$\$\ (\mathit{lbnd}\ n\ (\Lambda v_k.\,C_2))\\
\mathit{lbnd}\ n\ (\Lambda v_k.\,\mathsf{ABS}\ C) &= \mathsf{ABS}\ (\mathit{lbnd}\ (n+1)\ (\Lambda v_k.\,C)).
\end{aligned}$$

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 the exact inductive rules that specify the source of the function are not completely immediate, we will give a detailed proof of existence (see Appendix A for background on what this means).

Let I be the inductively defined set of all canonical forms in context specified in Figure 4. Let

$$\begin{aligned}
S &\stackrel{\text{def}}{=} \mathscr{CLF}_{exp\Rightarrow exp}(\Gamma^L_{exp})\\
W &\stackrel{\text{def}}{=} \mathscr{CLF}_{exp}(\Gamma^L_{exp}).
\end{aligned}$$

Let $\mathbb{N} \Rightarrow W$ be the set of functions from $\mathbb{N}$ to W. We will give an existence proof of a function $F : S \to (\mathbb{N} \Rightarrow W)$ and define

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

for any $n \in \mathbb{N}$. The recursion equations will follow trivially from the action of F.

Let $I_0 \stackrel{\text{def}}{=} \varnothing$ and 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

$$\begin{aligned}
S_0 &\stackrel{\text{def}}{=} \varnothing\\
S_1 &\stackrel{\text{def}}{=} \varnothing\\
S_{h+1} &\stackrel{\text{def}}{=} \{C \mid \Gamma^L_{exp} \vdash_{can} C :: exp \Rightarrow exp \in I_{h+1} - I_h\} \qquad (\text{for } h \geqslant 1).
\end{aligned}$$

It is easy to see that $S = \bigcup_{h\in\mathbb{N}} S_h$, and, moreover, that 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 $F \stackrel{\text{def}}{=} \bigcup_{h\in\mathbb{N}} F_h$, which is 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 $h \geqslant 1$ (recall Figure 4). This element cannot be generated with final rule VAR or CST as no ground type γ is the higher type

$exp \Rightarrow exp$. Hence, the final rule is ABS, and we must have

$$\Gamma^{L}_{exp}, v_k :: exp \vdash_{can} C' :: exp$$

where $C \equiv \Lambda v_k . C'$. Note also that $C' \in I_h - I_{h-1}$.

The sequence of functions F_h is defined by:

— $F_0 : S_0 \to (\mathbb{N} \Rightarrow W)$:
We take F_0 to be the empty function.

— $F_1 : S_1 \to (\mathbb{N} \Rightarrow W)$:
We take F_1 to be the empty function.

— $F_2 : S_2 \to (\mathbb{N} \Rightarrow W)$:
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

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

— $F_3 : S_3 \to (\mathbb{N} \Rightarrow W)$:
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 one of $\mathsf{CON}\ v$, $\mathsf{VAR}\ i$, $\mathsf{BND}\ j$, $v_{k'}\ \$\$\ v_{k''}$ or $\mathsf{ABS}\ v_{k'''}$. Hence F_2 is fully specified for any $n \in \mathbb{N}$ by

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

— $F_h : S_h \to (\mathbb{N} \Rightarrow W)$ where $h \geqslant 4$:
We prove by induction that for all $h \geqslant 4$ there is such a function F_h that 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.
We will now assume the proposition holds for all numbers r less than or equal to an arbitrary h, and prove it holds for $h + 1$. So we now assume $h + 1 \geqslant 4$ and 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 $h \geqslant 3$, 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 $0 \leqslant r \leqslant h$. 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, $e, e' \in \mathscr{CLF}_{exp\Rightarrow exp}(\Gamma^L_{exp})$. 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 renaming axiom. So we need to 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 $v_{k'}$ does not occur free in C.

Suppose 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, it is easy to see that $C \equiv v_{k''}$ for some k''. If $k \neq k''$, we have

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

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 the routine details for the cases where $(*)$ was derived using CST. □

4. The adequacy of de Bruijn expressions for λ-expressions

In this section we demonstrate very precisely 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 used 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 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 C. In Section 4.4, we prove the adequacy result in the form of 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

$$\theta \,:\, \mathscr{LE}/{\sim_\alpha} \to \mathscr{PDB} \subseteq \mathscr{DB}$$

that is **representationally adequate**, that is to say, θ is a compositional isomorphism $\mathscr{LE}/{\sim_\alpha} \cong \mathscr{PDB}$. 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 Propositions 1.1 and 1.2 in stating **CH**.

4.2. *A proof outline*

In this section we will give an informal outline of the proof, but first 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 v_k, L and L, L' for **cons** and **concatenation**, respectively. 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 has **position** $|L| - 1$ where $|L|$ is the **length** of the list. Thus v_{70} occurs at positions 1 and 2 in the example just given. 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 there is one, otherwise $elt\, p\, L$ is undefined. We say that L is **ordered** if L is a list and 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:

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

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 (see 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 $\mathscr{LE}/\sim_\alpha$, from the family $[\![-]\!]_L$ whose source is $\mathscr{LE}$.

In Proposition C.4 we show that

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

thereby 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 the inductive stages of its proof. Only once we have proved this 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*

Here we will just state the existence of the families of functions required to establish the bijection – the proofs are given in Appendix C. The reason for giving the definition of the functions in the main text is that other proofs in the paper make direct use of the definitions.

Proposition 4.2 (defining $[\![-]\!]_L$). For any L, there is a function

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

satisfying the following recursion equations (in particular, $[\![-]\!]_\epsilon : \mathscr{LE} \to \mathscr{PDB}$):

$$[\![v]\!]_L = \mathsf{con}(v)$$

$$[\![v_i]\!]_L = \begin{cases} \mathsf{bnd}(pos\ v_i\ L) \text{ if } v_i \in L \\ \mathsf{var}(i) \text{ if } v_i \notin L \end{cases} \qquad \begin{array}{l}(\text{where } pos\ v_i\ L \text{ is the position} \\ \text{of the variable } v_i \text{ in } L)\end{array}$$

$$[\![E_1\ E_2]\!]_L = [\![E_1]\!]_L\ \$\ [\![E_2]\!]_L$$

$$[\![\lambda v_i.\, E]\!]_L = \mathsf{abs}([\![E]\!]_{v_i,L}).$$

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

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

satisfying the following recursion equations (in particular, $(\!|-|\!)_\epsilon : \mathscr{PDB} \to \mathscr{LE}$):

$$(\!|\mathsf{con}(v)|\!)_L = v$$

$$(\!|\mathsf{var}(i)|\!)_L = v_i$$

$$(\!|\mathsf{bnd}(j)|\!)_L = elt\ j\ L \qquad (\text{where } elt\ j\ L \text{ is the } j\text{th element of } L)$$

$$(\!|D_1\ \$\ D_2|\!)_L = (\!|D_1|\!)_L\ (\!|D_2|\!)_L$$

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

where in the last equation $M = \mathsf{Max}(D; L)$ 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 occurring in D and the indices of L.

4.4. *Proving de Bruijn adequacy*

We can now prove Theorem 4.1.

Proof of Theorem 4.1. (It may help to revisit Section 4.2 and refer to the Appendix when required.)

B Recall from Propositions 4.2 and 4.3, defining $[\![-]\!]_L$ and $(\!|-|\!)_L$, respectively, that we have the existence of a pair of families of functions:

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

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

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

We may define

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

by setting $\theta([E]_\alpha) \stackrel{\text{def}}{=} [\![E]\!]_\epsilon$ for any $E \in \mathscr{LE}$ and $\phi \stackrel{\text{def}}{=} q \circ (\!|-|\!)_\epsilon$. Note that by Proposition C.3, $[\![-]\!]_\epsilon$ is equal on α-equivalent expressions, 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 $(\!|-|\!)_L \circ [\![-]\!]_L$).

CH The 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 by the following lemma.

Lemma 5.1. There is a function

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

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 $\mathsf{VAR}\ k$ are replaced by C'.

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

Theorem 5.2 is the key theorem of this paper.

Theorem 5.2 (HYBRID representational adequacy). There is a well-defined function

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

arising from the family of unique well-defined functions

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

satisfying the recursion equations

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

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

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

$$\Theta_L\,([\lambda\, v_i.\, E]_\alpha) \stackrel{\text{def}}{=} \mathsf{LAM}\, v_i.\, \Theta_{v_i, L}\,([E]_\alpha)$$

where in the last equation we write $\mathsf{LAM}\, v_i.\, \xi$ as an abbreviation for $\mathsf{ABS}\ (\mathit{lbnd}\ 0\ (\Lambda\, v_i.\, \xi))$. Then Θ_ϵ is **representationally adequate**, that is to say, Θ_ϵ is a compositional isomorphism $\mathscr{L}\mathscr{E}/{\sim_\alpha} \cong \Theta_\epsilon(\mathscr{L}\mathscr{E}/{\sim_\alpha})$. Equivalently:

B it is bijective (onto its image); and

CH it is a compositional homomorphism, which means that

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

where the right-hand expression exists by Lemma 5.1.

At this stage it will be instructive to experiment with computing the adequacy function Θ. For example, 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 (for example, v_8), the leaf remains unchanged; if not, (for example, 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 one can also check that this equals

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

5.2. *Factoring* HYBRID *adequacy*

A 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 speaking, $\Theta_\epsilon = \mathit{inst} \circ \theta$. 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 directly. It is far easier to prove properties, such as injectivity, of *inst*. We will first introduce *inst* using examples.

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

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

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

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

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

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 have 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 as it goes. When any leaf node bound index j is reached, it can use the counter to determine if the index is dangling or not. If it is, the index is replaced by the $(j-n)$th variable, and if it is not, it is left alone. Thus we would have

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

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, which are 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 6.

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 $(inst\ 0\ L) \circ \imath \circ \theta_L$ of three well-defined functions. Recall that $[\![-]\!]_L$ is defined in Proposition 4.2. The function θ_L is defined by $[\![-]\!]_L \circ q^{-1}$ where $q : \mathscr{LE} \to \mathscr{LE}/\!\sim_\alpha$ is the quotient function. In Lemma 5.3 we prove the existence of the function *inst*. Two other technical lemmas, Lemmas 5.4 and 5.5, are also required to establish that $\Theta_L = (inst\ 0\ L) \circ \imath \circ \theta_L$.

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

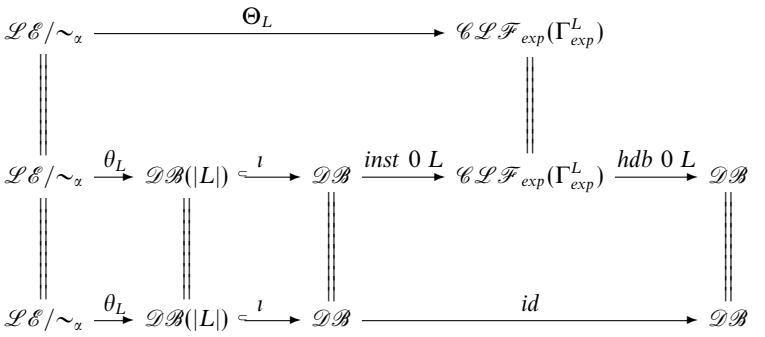

Fig. 6. Functions used in the adequacy proof

$$(hdb\ 0\ L) \circ (inst\ 0\ L) = id_{\mathscr{DB}}.$$

Hence *inst* is injective.

In Proposition 5.10 we show that *inst* is a compositional homomorphism, using the technical Lemma 5.9.

The HYBRID adequacy proof follows from this. θ_ϵ is representationally adequate by Theorem 4.1 for 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 then is that $inst\ 0\ \epsilon$ is injective and compositional, which follows from Propositions 5.8 and 5.10.

5.4. *Infrastructure for the proof outline*

We begin with 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 that 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 $n \geqslant 0$ and list L, there is a unique function with the following source and target

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

satisfying the recursion equations

$$inst\ n\ L\ \mathsf{con}(v) = \mathsf{CON}\ v$$

$$inst\ n\ L\ \mathsf{var}(i) = \mathsf{VAR}\ i$$

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

(recall from Section 4.2, that $elt\ p\ L$ computes the 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. $\mathscr{DB}$ is an inductively defined set, so by standard results (see Appendix A) the unique function *inst* $n\ L$ exists by appealing to structural recursion over $\mathscr{DB}$, provided the expressions on the right-hand sides of the equations are elements of $\mathscr{CLF}_{exp}(\Gamma^L_{exp})$ on the (inductive) assumption that recursive calls of *inst* are elements of $\mathscr{CLF}_{exp}(\Gamma^L_{exp})$.

Note that $elt\ (j-n)\ L$ is defined since $0 \leqslant j-n < |L|$ 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 $\mathscr{CLF}_{exp}(\Gamma^L_{exp})$. □

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

A useful exercise at this point is to show that

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

and

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

are equal. In fact the next lemma, Lemma 5.4, 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 $D[\mathsf{bnd}(j)]$ is a de Bruijn expression in which there are n abs nodes between the bound index j and the root. Consider $inst\ 1\ L\ D[\mathsf{bnd}(j)]$. The output of this function is essentially $D[\mathsf{bnd}(j)]$ but with:

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

$$0 \leqslant j-(n+1) \leqslant |L| \ \equiv\ n+1 \leqslant j \leqslant |L|+(n+1).$$

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

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

$$0 \leqslant j-n \leqslant |v_k, L| \ \equiv\ n \leqslant j \leqslant (1+|L|)+n.$$

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

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

and one may conjecture that

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

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

Lemma 5.4 (relating *lbnd* and *inst* for Proposition 5.6). Given any $D \in \mathscr{DB}$, list L, $n \geqslant 0$, and meta-variable v_k fresh for L, we have

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

Proof. We prove the lemma by induction on D.

— The argument for constants $\mathsf{con}(v)$ is similar to that for the inductive case of free indices, and applications are easy, and thus omitted.

— $\mathsf{var}(i)$:
Routine calculations give

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

— $\mathsf{bnd}(j)$:
We first assume that $n \leqslant j$ and $j - n < |v_k, L|$, and consider the cases of $j - n = 0$ and $j - n \geqslant 1$ separately:

– If $j - n = 0$, we have

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

where the final equality follows since $n + 1 > n = j$.

– If $j - n \geqslant 1$, we have

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

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

Now suppose that either $n > j$ or $j - n \geqslant |v_k, L|$. So we have

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

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

— $\mathsf{abs}(D)$:
We have

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

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

The following lemma allows us to do some variable renaming when applying the function *inst*. It will be used during the inductive steps of proofs that deal with abstractions, and hence also variable renaming. Recall that $[\![-]\!]_{\_}$ was defined in Proposition 4.2.

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

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

Proof. The proof is by induction over $E \in \mathscr{LE}$:

— The details for constants and applications are easy and thus omitted.
— v_α:
We consider the following cases:

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

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

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

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

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

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

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

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

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

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

— $\lambda v_\alpha.\, E$:
We have

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

with the third equality following by induction. □

We can now formalise our explanation in Section 5.2 in which we said that the function Θ_L would be factored through the function θ_L, 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 $[E]_\alpha \in \mathscr{LE}/\sim_\alpha$ and list L, we have

$$\Theta_L\ [E]_\alpha = inst\ 0\ L\ (\theta_L([E]_\alpha))$$

where the action of Θ_L is specified in the statement of Theorem 5.2.

Recall from Section 5.3 that for any E we set

$$\theta_L([E]_\alpha) \stackrel{\text{def}}{=} ([\![-]\!]_L \circ q^{-1})([E]_\alpha) = [\![E]\!]_L,$$

which is 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:

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

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

$$(\forall E)(\forall L)(inst\ 0\ L\ (\theta_L([E]_\alpha)) = \Theta_L([E]_\alpha)) :$$

— Constants and applications are easy, so the details are omitted.
— v_i:
When $v_i \in L$, we have

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

where the second equality follows because $0 \leqslant pos\ v_i\ L < |L|$.
The other simple case is left as an exercise.
— $\lambda v_i.E$:
We choose v_k to be fresh for L, E and v_i. Then

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

Step (α) follows from Proposition 3.2 since canonical forms are identified up to α-equivalence: note that we must ensure $v_k \notin inst\ 0\ (v_i, L)\ [\![E]\!]_{v_i,L}$. In fact, it is easy to

see that for any D, $\hat{L}$ and m, the variables occurring in *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 to show 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 help to work through the following example calculation:

$$\begin{aligned} hdb\ 0\ v_8\ (inst\ 0\ v_8\ (\mathsf{ABS}\ (\mathsf{BND}\ 1\ \$\$\ \mathsf{BND}\ 0))) \\ &= hdb\ 0\ v_8\ (\mathsf{ABS}\ (v_8\ \$\$\ \mathsf{BND}\ 0)) \\ &= \mathsf{ABS}\ ((hdb\ 1\ v_8\ v_8)\ \$\$\ (hdb\ 1\ v_8\ (\mathsf{BND}\ 0))) \\ &= \mathsf{ABS}\ (\mathsf{BND}\ 1\ \$\$\ \mathsf{BND}\ 0). \end{aligned}$$

Lemma 5.7 (defining *hdb* for Proposition 5.8)). For all $n \geqslant 0$ and list L, there is a unique function with the following source and target

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

satisfying the following recursion equations:

$$\begin{aligned} hdb\ n\ L\ v_k &= \mathsf{bnd}((pos\ v_k\ L) + n) \\ hdb\ n\ L\ (\mathsf{CON}\ v) &= \mathsf{con}(v) \\ hdb\ n\ L\ (\mathsf{VAR}\ i) &= \mathsf{var}(i) \\ hdb\ n\ L\ (\mathsf{BND}\ j) &= \mathsf{bnd}(j) \\ hdb\ n\ L\ (C_1\ \$\$\ C_2) &= (hdb\ n\ L\ (C_1))\ \$\ (hdb\ n\ L\ (C_2)) \\ hdb\ n\ L\ (\mathsf{ABS}\ C) &= \mathsf{abs}(hdb\ (n+1)\ L\ C). \end{aligned}$$

For the first equation, recall from Section 4.2 that *pos* $e\ L$ computes position of e in L.

Proof. Let I be the inductively defined set of all canonical forms in context specified in Figure 4. Let L be arbitrary and $S \stackrel{\text{def}}{=} \mathscr{CLF}_{exp}(\Gamma^L_{exp})$. 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 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.

Define

$$\begin{aligned} S_0 &\stackrel{\text{def}}{=} \varnothing \\ S_{h+1} &\stackrel{\text{def}}{=} \{C \mid \Gamma^L_{exp} \vdash_{can} C :: exp \in I_{h+1} - I_h\} \qquad (\text{for } h \geqslant 0). \end{aligned}$$

It is easy to see that $S = \bigcup_{h\in\mathbb{N}} S_h$, and, moreover, that 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 $F \stackrel{\text{def}}{=} \bigcup_{h\in\mathbb{N}} F_h$, which is a union of functions with pairwise disjoint sources:

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

— $F_1 : S_1 \to (\mathbb{N} \Rightarrow (\{L\} \Rightarrow W))$:
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

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

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

$$\begin{aligned}
F_2(\mathsf{CON}\ v)(n)(L) &\stackrel{\text{def}}{=} \mathsf{CON}\ v\\
F_2(\mathsf{VAR}\ i)(n)(L) &\stackrel{\text{def}}{=} \mathsf{VAR}\ i\\
F_2(\mathsf{BND}\ j)(n)(L) &\stackrel{\text{def}}{=} \mathsf{BND}\ j\\
F_2(v_k\ \$\$\ v_{k'})(n)(L) &\stackrel{\text{def}}{=} (F_1(v_k)(n)(L))\ \$\$\ (F_1(v_{k'})(n)(L))\\
F_2(\mathsf{ABS}\ v_{k''})(n)(L) &\stackrel{\text{def}}{=} \mathsf{ABS}\ (F_1(v_{k''})(n+1)(L)).
\end{aligned}$$

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

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

together with the functions F_h defined above for $h = 0, 1, 2$.
If $h = 0$, the proposition is vacuous.
So we now assume the proposition holds for all numbers r less than or equal to an arbitrary h, and prove it holds for $h + 1$. So we assume that $h + 1 \geqslant 3$. Suppose $C \in S_{h+1}$. Now $h \geqslant 2$ and by examining the rules used to generate C we see that either $C \equiv C_1\ \$\$\ C_2$ with each $C_s \in I_{h-1}$ of type *exp*, or $C \equiv \mathsf{ABS}\ C_3$ with $C_3 \in I_{h-1}$ 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 $0 \leqslant r \leqslant h$, so 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 $D \in \mathscr{DB}$, $n \geqslant 0$ and ordered list L, we have

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

In particular, each function

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

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.

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

$$\begin{aligned} hdb\ n\ L\ (inst\ n\ L\ \mathsf{bnd}(j)) &= hdb\ n\ L\ (elt\ (j-n)\ L) \\ &= \mathsf{bnd}((pos\ (elt\ (j-n)\ L)\ L) + n) \\ &= \mathsf{bnd}(j). \end{aligned}$$

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

$$\begin{aligned} hdb\ n\ L\ (inst\ n\ L\ \mathsf{bnd}(j)) &= hdb\ n\ L\ (\mathsf{BND}\ j) \\ &= \mathsf{bnd}(j). \end{aligned}$$

— $\mathsf{abs}(D)$:
We have

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

where the final equality follows by induction. □

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

Lemma 5.9 (for Proposition 5.10). For all $n \geqslant 0$, ordered L, and $D \in \mathscr{DB}(|L|)$, if $n \geqslant m$ where m is the minimum level of D, we have

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

Proof. This is a routine induction over D. For the case where 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 $D, D' \in \mathscr{DB}$, $n, k \geqslant 0$ and ordered L, if $n \geqslant m$ where m is the minimum level of D', we have

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

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

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

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

5.5. *Proving* Hybrid *adequacy*

We can now prove Theorem 5.2.

Proof of Theorem 5.2.

B We only need to show that Θ_ϵ is injective. From Proposition 5.6 (giving factoring of Θ_L), we know that

$$\Theta_\epsilon = (\mathit{inst}\ 0\ \epsilon) \circ \imath \circ \theta_\epsilon.$$

But $\theta_\epsilon = \theta$ and θ is injective by de Bruijn representational adequacy, Theorem 4.1. Inclusion $\imath$ is trivially injective. Furthermore, $\mathit{inst}\ 0\ \epsilon$ is injective from the left inverse, which is given by Proposition 5.8 (noting that ϵ is ordered by definition).

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

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

where: Equation 2 follows from Proposition 5.6 (giving factoring of Θ_L); Equation 3 follows from de Bruijn representational adequacy, Theorem 4.1; Equation 4 follows from Proposition 5.10 showing that *inst* is a compositional homomorphism – note that the value of n in the lemma is 0, so $n \geqslant m$ where $m = 0$ is the minimum level of $[\![E']\!]_\epsilon \in \mathscr{DB}(0)$; and Equation 5 follows from Proposition 5.6 again. □

6. Representation results

We can now use the results of the previous sections to prove some facts about the Hybrid representation of λ-expressions. Recall the notion of a *proper* de Bruijn term from Section 1.4 – see also Appendix B.1. We will now 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 can illustrate the notion of abstraction through an example. Suppose ABS C is proper: for example, let C = ABS (BND 0 $$ BND 1). Then C is of level 1, and, in particular, there may now be some dangling bound indices: for example, BND 1 in ABS (BND 0 $$ 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 $\Lambda v.$ ABS (BND 0 $$ v).

We will now briefly mention 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* 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 that reduce to proper de Bruijn expressions (see Section 1.4). The subset of $exp \Rightarrow exp$ consists of functions C such that LAM $v_i.\, C\ v_i$ is proper. The subsets consist of **proper** and **abstraction** expressions, respectively. With this, one may prove the following 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* $\mathscr{LE}$

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

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

satisfying the following recursion equations

$$\begin{aligned} level\ n(\mathsf{CON}\ v) &= T\\ level\ n\, v_k &= F\\ level\ n(\mathsf{VAR}\ i) &= T\\ level\ n(\mathsf{BND}\ j) &= n > j\\ level\ n(C_1\ \$\$\ C_2) &= (level\ n\, C_1) \wedge (level\ n\, C_2)\\ level\ n(\mathsf{ABS}\ C) &= level\ (n+1)\, C. \end{aligned}$$

Proof. We will omit the proof, which is similar in spirit to the proof given for Proposition 3.2. □

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

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

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

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

6.2. *Describing* HYBRID *abstraction expressions using* $\mathscr{LE}$

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

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

satisfying the following recursion equations:

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

Proof. We omit the proof, which is very similar to the proof of Proposition 3.2. □

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

Lemma 6.4. Given any canonical expression $C \in \mathscr{CLF}_{exp}(v_i :: exp)$ and $n \geqslant 0$ 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

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

with $(*)$ holding since $n \leqslant n$ 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 $abst\ n\ (\Lambda v_i.\,\mathsf{BND}\ j) = n < j$, and then

$$\begin{aligned} inst\ n\ v_i\ (hdb\ n\ v_i\ (\mathsf{BND}\ j)) &= inst\ n\ v_i\ (\mathsf{bnd}(j)) \\ &= \mathsf{BND}\ j. \end{aligned}$$

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

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

with the final equation valid by induction. □

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

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

Proof. We define

$$E \stackrel{\text{def}}{=} (\!|hdb\ 0\ v_i\ (C\ v_i)|\!)_{v_i}.$$

From Proposition C.4 it follows that

$$[\![E]\!]_{v_i} = hdb\ 0\ v_i\ (C\ v_i).$$

Then we have

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

where: Equation 6 follows from Proposition 5.6; Equation 7 follows from the definition of θ_L in Section 5.3; Equation 8 follows by calculating with θ_L; Equation 9 follows from Lemma 5.3; Equation 10 follows from Lemma 5.4; and Equation 12 is obtained by appeal to Lemma 6.4. □

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) and 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 proved 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 for 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, their 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 approaches 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. And then 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 in 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 we have presented (mathematically) in this paper. It remains for 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*

The notion of nameless binders was introduced in 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, for example, as v, or realised as indices – the *locally nameless* and *pure* approaches described elsewhere in this paper. See Hindley and Seldin (1988) for a textbook account

of nameless systems; a comparison of nameless and named binding systems can be found in Berghofer and Urban (2007).

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 work to ours 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 (1994) proved 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, 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 (2007) also undertakes such a proof, but with yet another variation of de Bruijn expressions, 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 formalise such bijections, and their 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 the 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 that encode such substitutions, see, for example, the excellent book Paulson (1997).

7.3. *Named binders*

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 a 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 the fundamental complication of α-equivalence. Traditional mechanisations worked with raw abstract syntax trees but had to 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 *defines* syntax with name binding in terms of an underlying type of de Bruijn λ-expression, which yields an automated system for α-conversion (Gordon 1994). The work presented in the current 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 that 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, which 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 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 have been considerably developed in Clouston and Pitts (2007) and Clouston (2010). Cheney has developed a simple type theory for nominal logic (Cheney 2009) and has proved results such as type soundness and normalisation. Formulations of LF style frameworks in a nominal setting appear in Berghofer *et al.* (2008) with updates in Berghofer *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 proved 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 through either:

(1) functions from *names* to expressions; or
(2) 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 have implemented a HYBRID system in Coq (Capretta and Felty 2007).

Further material on HOAS in type theory can be found in Capretta and Felty (2009), which contains generalisations both of our own work on HYBRID and of that in Capretta and Felty (2007). Roughly speaking, Capretta and Felty (2009) provides a language of universal algebra with bindings (and higher order signatures) that has an underlying de Bruijn syntax. It has interesting connections with 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 created when realising binders: if binders are realised as functions on inductive datatypes, there will exist expressions whose type matches the datatype but are not equal to the expressions that 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.* (2001a) on the *Theory of Contexts*, which defines 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, since, although coinduction is available, 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) and 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 of these, 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, using an S4 modal operator, the *primitive* recursive space (which encompasses functions defined through case analysis and iteration) from the *parametric* function space (whose members are those convertible to expressions built only using 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. Using the meta-logic, one can 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 Twelf. Properties such as type soundness for a simple pure functional language have been proved using an intuitionistic specification logic. More advanced work has considered a similar result for a continuation style presentation of the operational semantics, but this time using an ordered linear logic for the specification layer. This is particularly pleasing as it shows the possibility of incorporating new specification logics, while also dealing with a fairly complex example.

Related to this, McDowell and Miller 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 (McDowell and Miller 1997). Other inductive principles are *derived* through the use of appropriate measures. At the meta-meta level, they reasoned about object-level judgements formulated in second-order logic. They proved 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, thereby enabling reasoning over specifications.

7.5. *Models of binders*

Although not directly related to the work in this paper, we should note 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) and 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 $\mathscr{S}et$ in which injections $m \subseteq n$ are mapped to binding functions that map expressions with n free variables to expressions with m free variables. Ambler *et al.* (2004) gives an interesting example 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, so 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 roles of these key functions, and providing informative details about their operation through the various formal results presented here. Hopefully, one can concentrate on the 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 we have presented in this paper.

Appendix A. Induction and recursion

Induction and recursion play a central role in this paper, so in this appendix we give a brief outline of our assumptions in this paper in this regard (Aczel 1977; Crole 1998). Suppose we have a universal set U. A set of **(finitary) rules** $\mathscr{R} \subseteq \mathscr{P}(U) \times U$ 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 $\mathscr{R}$ if

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

where μ denotes the least fixpoint of the endofunction G on $\mathscr{P}(U)$. 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 $\mathscr{R}$. Moreover, if I_h is the collection of roots of such trees with height at most h, 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 is carried out 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

$$f \overset{\text{def}}{=} \bigcup_{h \in \mathbb{N}} f_h.$$

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

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

where i and j range over the natural numbers $\mathbb{N}$, and v ranges over a set of names. One should think of a de Bruijn expression as a finite rooted syntax tree. The leaf nodes are labelled: by constants $\mathsf{con}(v)$; by $\mathsf{var}(i)$, which corresponds to a free variable; or by $\mathsf{bnd}(j)$, which corresponds to a bound variable. We will be informal in our 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 $\mathsf{bnd}(j)$ a **bound index**. We call the i in expressions $\mathsf{var}(i)$ a **free index**. Given a de Bruijn expression D, a bound index j that 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 $l \geqslant 0$, if enclosing[†] D inside l nodes, each labelled with abs, ensures that the resulting expression has no dangling indices. We can define a predicate $level\ n : \mathscr{DB} \to \mathbb{B}$ for each $n \in \mathbb{N}$ where $\mathbb{B} \stackrel{\text{def}}{=} \{T, F\}$ such that the Boolean $level\ l\ D$ is true just when D is of level l by setting

$$level\ l\,(\mathsf{con}(v)) = T$$

$$level\ l\,(\mathsf{var}(i)) = T$$

$$level\ l\,(\mathsf{bnd}(j)) = l > j$$

$$level\ l\,(D_1 \ \$ \ D_2) = (level\ l\ D_1) \wedge (level\ l\ D_2)$$

$$level\ l\,(\mathsf{abs}(D)) = level\ (l+1)\ D.$$

It is (informally) clear that for any D, a (unique) minimum level m exists, and that D is at level l for any $l \geqslant m$. We write $\mathscr{DB}(l)$ 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).

Note that there are other presentations of de Bruijn terms. Pure de Bruijn has been studied extensively in Huet (1994), Nipkow (2001) and 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 $\mathsf{abs}(\mathsf{abs}(\mathsf{V3} \ \$ \ \mathsf{V0}))$

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

could be a representation of any λ-expression of the form $\lambda v_i.\, \lambda v_k.\, v_j\ v_k$ where $i \neq j$. The point is that the value j for the *free* variable is obviously not determined uniquely from informal working. Thus not only must any machine formalisation deal with tracking bound indices, but there must also be a fixed enumeration of free variables that are used to specify the actual values of indices like j. One is thus 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 have to introduce a notion of *proper* de Bruijn expressions (this is not required when using pure de Bruijn). Recall that $\mathscr{DB}(l)$ is the set of de Bruijn expressions at level l. It follows that

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

and it is easy to see that $\mathscr{DB} = \bigcup_{l<\omega} \mathscr{DB}(l)$ by considering minimum levels. Let $\mathscr{PDB} \stackrel{\text{def}}{=} \mathscr{DB}(0)$ 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 $k \geqslant 0$, that is $k \in \mathbb{N}$. The expressions are inductively defined by the grammar

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

We adopt the usual notions of **free** and **bound** variables, and α-equivalence. For completeness, we will now outline our notation. If v_k occurs in E, 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, we write $E[E'/v_k]$ for a *unique* expression, which, informally, is E with free occurrences of v_k replaced by E', with renaming to avoid capture. Our definition appears in Figure 7, and 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 now define α-equivalence. We write $\mathscr{LE}$ 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 $\mathscr{LE} \times \mathscr{LE}$ 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 $v_{k'} \notin fv(E)$. There are also 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 $\mathscr{LE}/\sim_\alpha$ for the set of α-equivalence classes of expressions. For the purposes of this paper, we will also need a notion of substitution on $\mathscr{LE}/\sim_\alpha$, analogous to Proposition 1.1.

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

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

Proof. This is an immediate consequence of Lemma B.5. Note that the lemma makes use of simultaneous substitutions. We need 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. Furthermore, because the function $[\![-]\!]_L$ involves *arbitrary* lists L, we require a definition that mirrors this so that it interacts well with the full machinery of the paper, and other lemmas in this appendix.

Let L and $L^{\mathscr{LE}}$ be lists of equal length, where L is a list of variables v_k as usual, and $L^{\mathscr{LE}}$ is a list of $\mathscr{LE}$ expressions. Suppose $E' \in L^{\mathscr{LE}}$ and $v_k \in L$ both occur at some position p. Then we shall say the expression and variable are **mates**, and say that one is the **mate** of the other. If $v_k \in L$, 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 $E[L^{\mathscr{LE}}/L]$ for, informally, the simultaneous capture avoiding substitution of each expression $E' \in L^{\mathscr{LE}}$ 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 this happens, the expression in $L^{\mathscr{LE}}$ that is the mate of the active occurrence is the one that is substituted – see the formal definition in Figure 7. 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. However, we have to take great care with the definition of capture avoiding substitution on abstractions where a renaming takes place: in particular, with the choice of the renaming variable. The definition is given in Figure 7. One may wonder if our proofs could be simplified by defining substitution on abstractions so that renaming *always* takes place; this would appear to eliminate the cases in Figure 7. However, although this reduces case analyses, the extra burden of always renaming is

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

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

$$v[L^{\mathscr{LE}}/L] \stackrel{\text{def}}{=} v$$

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

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

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

where

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

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

Fig. 7. Simultaneous substitution

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 $E \sim_\alpha E'$ for any expressions E and E'. Then for any lists L and $L^{\mathscr{LE}}$,

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

Proof. The proof is by induction over $\sim_\alpha$. However, the full proof is surprisingly tricky, and 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 proved, each formalising a fact about the properties of simultaneous substitution. Moreover, the 'proofs' of each of the results referred to requires the 'other' results within its own proof. The upshot is that these results must all be proved by induction at the same time, with the proof of each result calling the inductive hypotheses of the others. We collect these results together in Lemmas B.3 and B.4. □

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

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

$$\Psi(E) \stackrel{\text{def}}{=} (\forall L, L', M, M')(\forall v_k)((\forall X \in L(X{\downarrow} \vee X \notin fv\, E \vee \mathsf{mate}(X) \neq v_k)) \Longrightarrow E[L'[M'/M]/L] \sim_\alpha E[L'[\overline{M'}/\overline{M}]/L])$$

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

Then for all $E \in \mathscr{LE}$, 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 proved independently. □

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

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

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

Lemma B.5. The simultaneous substitution function for $\mathscr{LE}$ (Figure 7) can be extended to $\mathscr{LE}/\sim_\alpha$. More precisely, there is a well-defined function specified by

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

where $[L^{\mathscr{LE}}]_\alpha$ 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 $L_1^{\mathscr{LE}}$ and $L_2^{\mathscr{LE}}$ are two lists of equal length and consisting of pairwise α-equivalent expressions, then the function $-[L_1^{\mathscr{LE}}/+]$ equals $-[L_2^{\mathscr{LE}}/+]$. □

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 (≡ Proposition 4.2 – defining $[\![-]\!]_L$). For any L, there is a function

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

satisfying the following recursion equations (in particular, $[\![-]\!]_\epsilon : \mathscr{LE} \to \mathscr{PDB}$):

$$\begin{aligned}
[\![v]\!]_L &= \mathsf{con}(v) \\
[\![v_i]\!]_L &= \begin{cases} \mathsf{bnd}(pos\ v_i\ L) \text{ if } v_i \in L \\ \mathsf{var}(i) \text{ if } v_i \notin L \end{cases} \qquad \begin{array}{l}(\text{where } pos\ v_i\ L \text{ is the position} \\ \text{of the variable } v_i \text{ in } L)\end{array} \\
[\![E_1\ E_2]\!]_L &= [\![E_1]\!]_L\ \$\ [\![E_2]\!]_L \\
[\![\lambda\, v_i.\, E]\!]_L &= \mathsf{abs}([\![E]\!]_{v_i,L}).
\end{aligned}$$

Proof. We first prove by induction on E that

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

(which is virtually immediate). This ensures that uses of *level* type check, so we can then prove

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

We give details of the proof (within each case L is an arbitrary list):

— The details for constants and applications are easy.

— v_i:

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

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

If $v_i \in L$,

$$\begin{aligned}
level\ |L|\ [\![v_i]\!]_L &= level\ |L|\ (pos\ v_i\ L) \\
&= (pos\ v_i\ L) \\
&< |L| \\
&= T.
\end{aligned}$$

— $\lambda\, v_i.\, E$:

We have

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

with the final equality holding by induction. □

Proposition C.2 (≡ Proposition 4.3 – defining $(\!|-|\!)_L$). For any ordered L, there is a function

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

satisfying the following recursion equations (in particular, $(\!|-|\!)_\epsilon \,:\, \mathscr{PDB} \to \mathscr{LE}$):

$$\begin{aligned}
(\!|\mathsf{con}(v)|\!)_L &= v \\
(\!|\mathsf{var}(i)|\!)_L &= v_i \\
(\!|\mathsf{bnd}(j)|\!)_L &= \mathit{elt}\ j\, L \qquad \text{(where } \mathit{elt}\ j\, L \text{ is the } j\text{th element of } L\text{)} \\
(\!|D_1\ \$\ D_2|\!)_L &= (\!|D_1|\!)_L\ (\!|D_2|\!)_L \\
(\!|\mathsf{abs}(D)|\!)_L &= \lambda v_{M+1}.\, (\!|D|\!)_{v_{M+1},L}
\end{aligned}$$

where in the last equation $M = \mathsf{Max}(D;L)$ 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 occurring in D and the indices of L.

Proof. We prove by induction on D that

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

— The details for constants, free indices and applications are easy and omitted.
— $\mathsf{bnd}(j)$:
If $\mathit{level}\ |L|\, \mathsf{bnd}(j)$, then $0 \leqslant j < |L|$ so $L \neq \epsilon$. So $(\!|\mathsf{bnd}(j)|\!)_L = \mathit{elt}\ j\, L$ is *defined* and hence exists in $\mathscr{LE}$.
— $\mathsf{abs}(D)$:
Note that

$$\mathit{level}\ |L|\, \mathsf{abs}(D) = \mathit{level}\ (|L|+1)\, D.$$

Hence, by induction, for any ordered list L', we have $(\!|D|\!)_{L'} \in \mathscr{LE}$ 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 $\mathscr{LE}$, 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 in the scope of the binding variable v_{M+1}, they will not be (accidently) captured, as $M + 1 > i$. The following example can be checked as an exercise:

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

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. We prove

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

by induction on the axioms and rules defining alpha equivalence. 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

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

The equality follows from 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 $D \in \mathscr{DB}$ and L be any ordered list such that for all $v_k \in L$, if any, $k \geqslant \mathsf{Max}(D;\epsilon) + 1$. Then

$$level\ |L|\ D \Longrightarrow [\![(\!|D|\!)_L]\!]_L = D.$$

Proof. The proof is a straightforward induction over $D \in \mathscr{DB}$ (constants are trivial, but note that the other two base cases make crucial use of the assumptions in the proposition):

— Constants are trivial.

— $\mathsf{var}(i)$:
We have $[\![(\!|\mathsf{var}(i)|\!)_L]\!]_L = [\![v_i]\!]_L = \mathsf{var}(i)$ for all L, since if $v_i \in L$ then

$$i \geqslant \mathsf{Max}(\mathsf{var}(i);\epsilon) + 1 = i + 1,$$

which is a contradiction.

— $\mathsf{bnd}(j)$:
We have

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

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 $E \in \mathscr{LE}$, and let L and L' be lists, with L' ordered, such that $|L| = |L'|$. Then

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

Proof. We apply induction over $\mathscr{LE}$:

— As ever, constants and applications are trivial.

— v_i:
If $v_i \notin L$, we have

$$\begin{aligned}(\!|[\![v_i]\!]_L|\!)_{L'} &\stackrel{\text{def}}{=} (\!|\mathsf{var}(i)|\!)_{L'} \\ &\stackrel{\text{def}}{=} v_i \\ &= v_i[L'/L].\end{aligned}$$

If $v_i \in L$, then

$$\begin{aligned}(\!|[\![v_i]\!]_L|\!)_{L'} &\stackrel{\text{def}}{=} (\!|pos\ v_i\ L|\!)_{L'} \\ &\stackrel{\text{def}}{=} elt\ (pos\ v_i\ L)\ L' \\ &\stackrel{\text{def}}{=} v_i[L'/L].\end{aligned}$$

— $\lambda v_i.\, E$:
We have

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

where: 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; and the final step (19) follows from 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). Given any expressions $E, E' \in \mathscr{LE}$, list L and variable v_k, if $v_k \notin L$ and $fv(E') \cap L = \emptyset$, then

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

Proof. The substitution functions exist from Propositions 1.1 and B.1. The proof proceeds by induction over the size of the expression E. We 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. We write $\Phi(E)$ for $(*)$ when E', L and k are universally quantified and satisfy the given

constraints. We write $\Psi(n)$ for

$$(\forall E)(size(E) = n \Longrightarrow \Phi(E))$$

and prove $\forall n.\Psi(n)$ by induction on n. We write LHS and RHS for the appropriate instance of $(*)$ in the inductive steps below.

— $\Psi(1)$:
If E is a constant, the result is immediate. Otherwise, we choose E to be v_i, of size 1, and prove $\Phi(v_i)$.

– Case $i = k$:

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

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

– Case $i \neq k$:

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

where if $v_i \in L$, the second equality is immediate, otherwise, the equality follows because $i \neq k$.

— $(\forall n)[(\forall m < n)(\Psi(m)) \Longrightarrow \Psi(n)]$ where $n \geqslant 2$:
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 $i = k$:

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

where the third equality follows from Lemma C.12 since $v_k = v_i \in v_i, L$, 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

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

    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 $v_i \notin fv(E')$, we have

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

    where: Equation (22) follows by induction as $size(E) = n - 1$, so $\Phi(E)$ holds, and $v_k \notin v_i, L$ and $fv(E') \cap (v_i, L) = \emptyset$; and Equation (23) follows from an instance of Lemma C.13 in which $L' = \epsilon$ and again $fv(E') \cap (v_i, L) = \emptyset$.
  - 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: Equation (27) follows by induction since

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

    and w is the maximum of the indices in E, E' and v_k, plus 1; and Equation (28) follows from Lemmas C.11 and C.13.

— The details for applications are easy and omitted. □

C.2. *Lemmas for the propositions*

In this section 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 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 depends on Lemma C.10).

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

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

Proof. We prove this by induction over $\mathscr{DB}$:

— Constants and applications are trivial.

— $\mathsf{var}(i)$:
We have $(\!|\mathsf{var}(i)|\!)_L = v_i$. So, if $k \geqslant \mathsf{Max}(\mathsf{var}(i); L) + 1$, we have $k > i$ and we are done.

— $\mathsf{bnd}(j)$:
We have $(\!|\mathsf{bnd}(j)|\!)_L = elt\ j\ L$. This is similar to previous case: k is strictly greater than any index in L.

— $\mathsf{abs}(D)$:
We pick

$$k \geqslant \mathsf{Max}(\mathsf{abs}(D); L) + 1 = \mathsf{Max}(D; L) + 1.$$

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

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

If $k = M + 1$, we have v_k is not free in $(\!|\mathsf{abs}(D)|\!)_L$, as any free occurrence will be captured.

If $k > M + 1$, we have $k \geqslant \mathsf{Max}(D; v_{M+1}, L) + 1$, so, by induction, v_k is not free in $(\!|D|\!)_{v_{M+1},L}$, and we are done. □

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

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

whenever

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

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

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

Proof.

The proof is by induction over $\mathscr{DB}$:

— Constants and applications are trivial.

— $\mathsf{bnd}(j)$:
We have

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

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

— $\mathsf{var}(i)$:
We have

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

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 (**).

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

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

and

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

Then we have

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

From (ii), $v_{M+1} \notin L'$, so the substitution in Equation (30) does not involve renaming. Furthermore, recall that, by definition,

$$M \stackrel{\text{def}}{=} \mathsf{Max}(\mathsf{abs}(D);\hat{L},L).$$

Hence $v_{M+1} \notin \hat{L}$ and thus, recalling the definition of substitution, Equation (31) holds with $\overline{L'} = L'$ and $\overline{\hat{L}} = \hat{L}$.

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

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

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

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

Furthermore, $M' + 1$ 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).

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

Equation (35) holds by induction together with the congruence of abstraction, as the conditions of the lemma both hold as follows. Note that (*) is true because

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

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

$$\begin{aligned}\mathsf{Min}\{k \mid \exists v_k \in v_{M+1}, \hat{L}\} &= \mathsf{Min}\{k \mid \exists v_k \in \hat{L}\} \\ &\geqslant \mathsf{Max}(D; \epsilon) + 1.\end{aligned}$$

□

Lemma C.9. For any ordered L and $D \in \mathscr{DB}$, there is a sufficiently large $w \geqslant 0$ 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 (*) and (**) hold:

(*) holds provided we choose

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

(**) holds because

$$\begin{aligned}\mathsf{Min}\{k \mid v_k \in v_{M+1}\} &= M + 1 \\ &= \mathsf{Max}(D; L) + 1 \\ &\geqslant \mathsf{Max}(D; \epsilon) + 1.\end{aligned}$$

Furthermore, 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 Lemma C.7, noting that

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

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 $E \in \mathscr{LE}$ and lists L and L', and any v_k, if the conditions

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

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

hold, then

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

Proof. We use induction over $\mathscr{LE}$:

— Constants and applications are trivial.

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

— $\lambda v_i.\, E$:

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

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

The equality will follow by induction provided both

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

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

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

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

Proof. The proof is 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. We write $size(E)$ for the size of E and $\Phi(E)$ for (*) in which L, L', k and k' are universally quantified and satisfy the given constraints. We write $\Psi(n)$ for

$$(\forall E)(size(E) = n \Longrightarrow \Phi(E))$$

and 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.

— $\Psi(1)$:
Consider arbitrary expressions of size 1. If it is 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

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

where the positions are equal due to the constraints of the lemma on 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, so we suppose $v_i \notin L'$. Note further that $k \neq i \neq k'$. Hence either $v_i \in L$, and we are done, or $v_i \notin L$ and

$$\begin{aligned}\mathsf{var}(i) &\stackrel{\text{def}}{=} [\![v_i]\!]_{L',v_{k'},L} \\ &= [\![v_i]\!]_{L',v_k,L} \\ &\stackrel{\text{def}}{=} \mathsf{var}(i).\end{aligned}$$

— $(\forall n)((\forall m < n)(\Psi(m) \Longrightarrow \Psi(n)))$:
We choose arbitrary $n \geqslant 2$ and assume 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 $N \in \mathscr{LE}$ is of the form $E_1\ E_2$, the result follows by a routine induction argument, which we omit.
Note that for the case when $N \in \mathscr{LE}$ is of the form $\lambda v_i.\, E$, we can assume that $\Phi(M)$ for any M with $size(M) < size(E) + 1$. We have to prove that

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

when

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

and $v_k \notin L'$ and $v_{k'} \notin L'$.
We consider:

– Case $i = k$:
Here the variable v_k is not free in $\lambda v_i.\, E$, so

$$\begin{aligned}LHS &= \mathsf{abs}([\![E]\!]_{v_i,L',v_{k'},L}) \\ RHS &= \mathsf{abs}([\![E]\!]_{v_i,L',v_k,L}).\end{aligned}$$

Equality follows from Lemma C.10 provided the assumptions of the lemma are satisfied. Note that if $i = k'$, we are (trivially) done, so suppose $i \neq k'$. Now $(*)$ holds because $i = k$ implies $v_k \in v_i, L$, and $(**)$ holds because $v_{k'} \notin fv(E) \setminus \{v_i\}$ 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 occurs when the substitution does not involve a renaming. Note that

$$\begin{aligned}LHS &= \mathsf{abs}([\![E[v_{k'}/v_k]]\!]_{v_i,L',v_{k'},L}) \\ RHS &= \mathsf{abs}([\![E]\!]_{v_i,L',v_k,L}).\end{aligned}$$

If $v_i \neq v_{k'}$, then $v_{k'} \notin fv(E)$ follows using the same reasoning as in case $i = k$ above. Also, since $i \neq k$ and $i \neq k'$, we get that $LHS = RHS$ follows inductively from $\Phi(E)$ because $size(E) < size(E) + 1$.
If $v_k \notin fv(E)$ (and $i = k'$ say), then $LHS = \mathsf{abs}(\llbracket E \rrbracket_{v_i, L', v_{k'}, L})$. Equality follows from Lemma C.10 provided the assumptions of the lemma are satisfied: (*) holds because $v_k \notin fv(E)$; and (**) holds because $i = k'$ implies $v_{k'} \in v_i, L$.

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

$$LHS = \llbracket \lambda\, v_w.\, E[v_w/v_i][v_{k'}/v_k] \rrbracket_{L', v_{k'}, L} \tag{37}$$

$$\stackrel{\text{def}}{=} \mathsf{abs}(\llbracket E[v_w/v_i][v_{k'}/v_k] \rrbracket_{v_w, L', v_{k'}, L}) \tag{38}$$

$$= \mathsf{abs}(\llbracket E[v_w/v_i] \rrbracket_{v_w, L', v_k, L}) \tag{39}$$

$$= \mathsf{abs}(\llbracket E \rrbracket_{v_i, L', v_k, L}) \tag{40}$$

$$\stackrel{\text{def}}{=} RHS. \tag{41}$$

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

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

  neither v_k nor $v_{k'}$ is in v_w, L'; and $v_{k'} \notin fv(E[v_w/v_i])$ 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 $\llbracket E \rrbracket_L$, 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, $\llbracket v_1\ v_2 \rrbracket_{v_1, v_3} = \mathsf{bnd}(0)\ \$\ \mathsf{var}(2)$). Since L enumerates binding variables as the function $\llbracket - \rrbracket_L$ descends through expressions E, we have $v_k \notin L$ implies that $v_k \in fv(E)$ (consider the contrapositive example $\llbracket \lambda\, v_2.\, v_1\ v_2 \rrbracket_\epsilon = \mathsf{abs}(\llbracket v_2 \rrbracket_{v_1\ v_2})$). Hence we have the following lemma.

Lemma C.12. For any $E \in \mathscr{LE}$, list L and variable v_k, if either $v_k \notin fv(E)$ or $v_k \in L$, we have $\mathsf{var}(k) \notin \llbracket E \rrbracket_L$.

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

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 renaming, allowing the fresh variable to be eliminated at suitable points in calculations.

Lemma C.13. Let $E \in \mathscr{LE}$, L and L' be any lists and v_i be a variable. Also, suppose that for any $v_k \in fv(E)$, either $v_k \in L'$ or $v_k \notin v_i, L$. Then $\llbracket E \rrbracket_{L', v_i, L} = \llbracket E \rrbracket_{L', L}$.

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

Acknowledgements

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

References

Aczel, P. (1977) An Introduction to Inductive Definitions. In: *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). In: Proceedings of the Third International Workshop on Logical Frameworks and Meta-Languages (LFM'02). *Electronic Notes in Theoretical Computer Science* **70** (2).

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. *Springer-Verlag Lecture Notes in Computer Science* **2410** 13–30.

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: Baaz, M., Makowsky, J. and Voronkov, A. (eds.) Computer Science Logic/8th Kurt Godel Colloquium Poster Collection, Vienna, August 2003. *Collegium Logicum (Proceedings of the Kurt Godel Society)*, KGS Publications 83–90.

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. *Springer-Verlag Lecture Notes in Computer Science* **3223**.

Aydemir, B. E., Charguéraud, A., Pierce, B. C., Pollack, R. and Weirich, S. (2008) Engineering Formal Metatheory. *SIGPLAN Notices* **43** (1) 3–15.

Berghofer, S. and Urban, C. (2007) A Head-to-Head Comparison of de Bruijn Indices and Names. In: Proceedings of the First International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2006). *Electronic Notes in Theoretical Computer Science* **174** (5) 53–67.

Berghofer, 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)*, IEEE Computer Society 45–56.

Berghofer, S., Cheney, J. and Urban, C. (2010) Mechanizing the Metatheory of LF. Technical report, Edinburgh and Munich.

Capretta, V. and Felty, A. (2007) Combining de Bruijn Indices and Higher-Order Abstract Syntax in Coq. In: Altenkirch, T. and McBride, C. (eds.) Proceedings of TYPES 2006. *Springer-Verlag Lecture Notes in Computer Science* **4502** 63–77.

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

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*, Ph.D. 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. (eds.) Computation, Meaning and Logic, Articles dedicated to Gordon Plotkin. *Electronic Notes in Theoretical Computer Science* **1496** 223–257.

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.

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. (eds.) Proceedings of the International Conference on Typed Lambda Calculi and Applications. *Springer-Verlag Lecture Notes in Computer Science* **902** 124–138.

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. (ed.) Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97). *Springer-Verlag Lecture Notes in Computer Science* **1210** 147–163. (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. (ed.) Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics. *Springer-Verlag Lecture Notes in Computer Science* **2342**.

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

Felty, A. P. and Momigliano, A. (2009) Reasoning with Hypothetical Judgments and Open Terms in Hybrid. In: *PPDP '09: Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*, ACM 83–92.

Felty, A. P. and Momigliano, A. (2010) Hybrid – A Definitional Two-Level Approach to Reasoning with Higher-Order Abstract Syntax. *Journal of Automated Reasoning* 1–63.

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)*, IEEE Computer Society Press 193–202.

Fiore, M. P. (2006) On the structure of substitution. Invited address for the 22nd Mathematical Foundations of Programming Semantics Conference (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. (ed.) *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, IEEE Computer Society Press 214–224.

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. (eds.) Proceedings of IJCAR. *Springer-Verlag Lecture Notes in Computer Science* **5195** 154–161.

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*, IEEE Computer Society 33–44.

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

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

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: *Proceedings, Symposium on Logic in Computer Science, Ithaca, New York*, IEEE Computer Society Press 194–204.

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

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*, London Mathematical Society Student Texts **1**, Cambridge University Press.

Hofmann, M. (1999) Semantical Analysis of Higher-Order Abstract Syntax. In: *Proceedings of 14th Annual IEEE Symposium on Logic in Computer Science, LICS'99, Trento, Italy*, IEEE Computer Society Press 204–213.

Honsell, F., Miculan, M. and Scagnetto, I. (2001a) An axiomatic approach to metareasoning on systems in higher-order abstract syntax. In: Proceedings ICALP'01. *Springer-Verlag Lecture Notes in Computer Science* **2076** 963–978.

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. (ed.) *MERLIN'05, Mechanized Reasoning about Languages with Variable Binding 2005 Workshop*, 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: Morazán, M. (ed.) *Eighth Symposium on Trends in Functional Programming (TFP 2007)*, Intellect 19–35.

McDowell, R. (1997) *Reasoning in a Logic with Definitions and Induction*, Ph.D. 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. (ed.) *Proceedings of the Twelfth Annual Symposium on Logic in Computer Science*, IEEE Computer Society Press 434–445.

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. (eds.) MERLIN 2001: Proceedings of the Workshop on MEchanized Reasoning about Languages with variable bINding. *Electronic Notes in Theoretical Computer Science* **58** 1–22.

Miller, D. (2006) Representing and reasoning with operational semantics. In: Furbach, U. and Shankar, N. (eds.) Automated Reasoning: Third International Joint Conference, IJCAR 2006. *Springer-Verlag Lecture Notes in Computer Science* **4130** 4–20.

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

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. (eds.) Supplemental Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics, Report EDI-INF-RR-0046 267–282.

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). *Electronic Notes in Theoretical Computer Science* **196** 85–93.

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. (eds.) Theorem Proving in Higher Order Logics, 20th International Conference. *Springer-Verlag Lecture Notes in Computer Science* **4732** 207–222.

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

Pfenning, F. (2003) Computation and deduction. (Available from: `http://www.cs.cmu.edu/~twelf/notes/cd.ps`.)

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

Pfenning, F. and Schürmann, C. (1999) System Description: Twelf – A Meta-Logical Framework for Deductive Systems. In: Ganzinger, H. (ed.) Proceedings of the 16th International Conference on Automated Deduction (CADE-16). *Springer-Verlag Lecture Notes in Artificial Intelligence* **1632** 202–206.

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

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*, ACM Press 263–274.

Urban, C. (2008) Nominal Techniques in Isabelle/HOL. *Journal of Automated Reasoning* **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 Techniques in Isabelle/HOL. In: Nieuwenhuis, R. (ed.) Proceedings of the 20th Conference on Automated Deduction (CADE 2005), Tallinn, Estonia. *Springer-Verlag Lecture Notes in Computer Science* **3632** 38–53.

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

Vestergaard, R. and Brotherson, J. (2001) A Formalized First-Order Conflence Proof for the λ-Calculus using One Sorted Variable Names. In: Middelrop, A. (ed.) Proceedings of RTA'12. *Springer-Verlag Lecture Notes in Computer Science* **2051** 306–321.