ELSEVIER

Theoretical Computer Science 192 (1998) 201–231

Theoretical Computer Science

A lambda-calculus for dynamic binding

Laurent Dami *

Centre Universitaire d'Informatique, 24, rue Général-Dufour, CH-1211 Genève 4, Switzerland

Abstract

Dynamic binding is a runtime lookup operation which extracts values corresponding to some "names" from some "environments" (finite, unordered associations of names and values). Many situations related with flexible software assembly involve dynamic binding: first-class modules, mobile code, object-oriented message passing. This paper proposes λN, a compact extension of the λ-calculus to model dynamic binding, where variables are labelled by names, and where arguments are passed to functions along named channels. The resulting formalism preserves familiar properties of the λ-calculus, has a Curry-style-type inference system, and has a formal notion of compatibility for reasoning about extensible environments. It can encode records and record extensions, as well as first-class contexts with context-filling operations, and therefore provides a basic framework for expressing a wide range of name-based coordination mechanisms. An experimental functional language based on λN illustrates the exploitation of dynamic binding in programming language design.

Keywords: Lambda-calculus; Records; Contexts; Dynamic binding

1. Introduction

Computer systems are required to be increasingly "open" – able to dynamically interact with other, possibly unknown or weakly specified systems, and able to coordinate together a global computation. In order to follow this evolution, computational models pay ever increasing attention to notions such as concurrency and distribution. However, open systems also often depend on another concept, more or less orthogonal to the previous ones, and which seems to have been less investigated in theoretical studies: *dynamic binding*. This appears in a family of programming constructs in which the runtime system includes some notions of "names" and "environments" (associations from names to values), and where the operation of looking up some name in some environment is performed dynamically. A number of popular languages use dynamic binding, under various forms: *quote* and *eval* in LISP, stacks of dictionaries in FORTH

* E-mail: laurent.dami@cui.unige.ch.

PII S0304-3975(97)00150-3

or Postscript, late binding of message names to methods in object-oriented languages, communication channels in concurrent systems. More recently, several proposals have been made to use first-class environments as a tool for flexible modularity [17, 21]; furthermore, in the new context of *coordination* models and languages, most proposals addressing distribution issues include some scheme for dynamic binding: the name server of [22], the flexible records of Ariadne [12], the tuples of named values in Sonia [3] are just a few examples. So in several contexts some form of dynamic binding has been acknowledged as a good mechanism for incremental assembly and coordination of software fragments.

When comparing these different implementations of a simple concept, it appears that small variations in the name lookup operation or in the constructs for building environments may generate quite different properties. Hence, formal models developed so far for some of these paradigms, in which the dynamic binding features are implicitly incorporated but merged with other computational aspects, are not adequate to perform comparisons and to study dynamic binding in an abstract, general setting. For example, several object calculi have been designed to study message-passing, but they can hardly be used to express the semantics of LISP. By contrast, a formal model in which dynamic binding is factored out from other computational aspects can throw some light on the relationships between various paradigms. We propose such a model, in the form of a λ-calculus in which arguments are passed to functions along named channels – so it is called λ-calculus with names, or λN for short. We also show how this model is a natural foundation for introducing dynamic binding in a typeful way into functional programming languages like ML [20] or Haskell [16].

Clearly, dynamic binding has an associated cost in terms of computing resources (memory to store the environments, time to perform lookup operations), but it also has the very appealing aspect of *extensibility*, i.e. the possibility to add more functionality to an existing piece of code, without affecting its previous behaviour. This comes from the fact that an environment defining a given set of names can be replaced by a bigger environment, defining more names: all name lookup operations involving the original set of names are still valid, but in addition some new lookup operations become possible. As a result, the modified code is "compatible" with the original code, which is very convenient for software evolution. These notions are central to the spirit of object-oriented programming, and are key factors for its success. Hence, semantic studies of languages with dynamic binding should attempt to capture this compatibility relationship, which is asymmetric, rather than usual equivalence relations between programs. A partial answer comes from the methodologies developed for describing subtyping in typed object calculi: one is based on "partial equivalence relationships" (PERs) [6], which indicate when two values are equivalent at a given type, and the other is based on coercion functions from subtypes to supertypes [5]. However, these do not directly express the fact, very intuitive to programmers, that for example record $\{x=1, y=2\}$ totally subsumes record $\{x=1\}$, i.e. can safely replace it *at all types*. In order to deal with this notion, we explicitly introduce a notion of runtime error in untyped λN, and then define an operational ordering based on the observation of error generation. By

this means, we can formally prove when an extension of a term is "compatible" with its original term, and we get some general laws for safe program manipulations.

The expressive power of λN is close to the "uniform system of parameterization" of [18], and to the recent calculus of contexts of [19]. However, these do not have a formal notion of compatibility, and do not address typing issues; furthermore, λN with its four syntactic constructs is more compact and therefore seems to be the minimal extension of the λ-calculus to support dynamic binding. The binding structures of [25] also deal with similar mechanisms, but with an emphasis on unification and term rewriting. Their complex operations involving hole filling and substitution have important applications in the field of theorem provers, but are heavy for simple programming purposes; furthermore, these are meta-operations, not directly expressed as computation within the language. By contrast, the label-selective calculus of [13], although seemingly similar in surface, has quite different properties: labels (names) are used to address inner λ-abstractions out of their definition order. This combines label-selection with currying, but does not support extensibility and compatibility properties discussed above. Finally, λN is also closely related to a λ-calculus with extensible records [24, 27], although not fully equivalent. Calculi of extensible records internally distinguish between functional and record values, while λN treats everything as a function, much like the pure classical λ-calculus.

This paper borrows some material from a previous presentation of the λN calculus [10], but with a different emphasis. In [10] we were mainly concerned with inference of principal types for $\lambda N$and their use for filtering communication in a shared dataspace. The motivation for using names and dynamic binding for coordination purposes was discussed in some detail in this paper. Here, by contrast, we concentrate on the basic theory of λN, on its relationships with other calculi, and on applications of the model to programming language design. Section 2 presents the untyped calculus, together with its main properties (confluence, context lemma, compatibility laws). Section 3 gives an adaptation of Curry's simple type inference system to functions with named parameters. Section 4 discusses the encoding of record operations in λN, and compares the calculus with record calculi. Section 5 relates this work to other calculi with environments, contexts or labels. Finally, Section 6 displays some applications of the calculus in the field of typed functional programming; several constructs for dynamic binding were integrated into a prototype interpreter, with direct translation into the underlying model. This interpreter was one of the deliverables of the European project ESPRIT BRA 9102 "Coordination"; financial support of Swiss OFES for our participation to this project is gratefully acknowledged.

2. The untyped λN calculus

2.1. Syntax and reduction rules

The calculus is constructed from a set $\mathcal{V}$ of variables and a set $\mathcal{N}$ of *names* (or labels); both sets may be infinite, and need not be disjoint. Letters x, y, z are

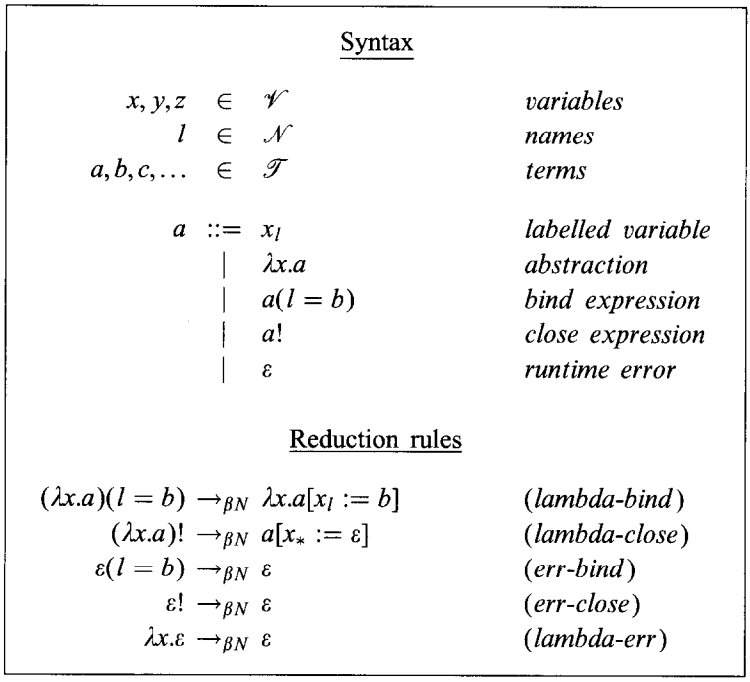

Fig. 1. Syntax and reduction rules.

metavariables for members of $\mathscr{V}$, and l is a metavariable for members of $\mathscr{N}$; concrete names in examples are written in serif font. Letters $a, b, c, \ldots$ are metavariables for arbitrary terms. The abstract syntax and reduction rules are displayed in Fig. 1. Variables carry several values at different names, so an expression of the form x_l corresponds to the value carried by variable x at name l. Lambda abstractions are exactly like in the standard lambda calculus, and the notions of free and bound variables are also the same (see [4]). We write $FV(a)$ for the set of free variables occurring in a, and $FN(a,x)$ for the set of names which index free occurrences of x in a; so if x_l occurs free in a then $x \in FV(a)$ and $l \in FN(a,x)$. A term is *closed* iff it has no free variables, and the set of closed terms is denoted by Λ_N^0. Usual application is split into two different parts: an expression of the form $a(l=b)$ (called *bind* expression) passes value b under name l to abstraction a; an expression of the form $a!$ (*close* expression) ends a sequence of bind expressions. Finally, ε is a constant representing runtime errors, i.e. the well-known "message not understood" error of object-oriented systems; errors are generated when trying to access a variable under a name for which that variable has no value (because there was no corresponding bind expression on the same name). Usual syntactic conventions apply, i.e. abstractions extend to the right as far as possible, and multiple abstractions of the form $\lambda x_1 \ldots \lambda x_n.a$ are abbreviated as $\lambda x_1 \ldots x_n.a$.

The capture-avoiding substitution of b for all free occurrences of x_l in a is written $a[x_l := b]$. Similarly, $a[x_* := b]$ denotes the substitution of b for all occurrences of

variable x in a, whatever their label index may be. Avoidance of variable capture is handled as in the standard lambda calculus, by considering equivalence classes of λN terms under α-substitution (renaming of bound variables) [4].

One-step reduction, written $\rightarrow_{\beta N}$, splits the usual β-reduction rule of standard λ-calculus into bind-reduction and close-reduction rules; in addition, three other rules ensure propagation of run-time errors. Notice how the *lambda-bind* rule performs a substitution without removing the outermost λ, while the *lambda-close* rule removes the λ and substitutes any remaining occurrence of the corresponding variable by ε. By contrast, β-reduction in the standard lambda calculus substitutes the variable and removes the λ in one single step. Following common conventions, the n-composition of $\rightarrow_{\beta N}$ is written $\overset{n}{\rightarrow}_{\beta N}$, the reflexive, transitive closure of $\rightarrow_{\beta N}$ is written $\overset{*}{\rightarrow}_{\beta N}$, and $\leftrightarrow_{\beta N}$ is its symmetric closure.

2.2. *Embedding the classical λ-calculus*

Assume an "invisible name" $\imath \in \mathcal{N}$, and let Λ denote the set of traditional λ-terms. These can be embedded into Λ_N by the translation function $\mathbf{LN}[-]$ below:

$$\mathbf{LN}[-] : \Lambda \rightarrow \Lambda_N$$

$$\mathbf{LN}[x] = x_\imath$$

$$\mathbf{LN}[\lambda x.a] = \lambda x.\mathbf{LN}[a]$$

$$\mathbf{LN}[ab] = \mathbf{LN}[a](\imath = \mathbf{LN}[b])!$$

The translation preserves usual β-equality:

Lemma 1. (i) $\forall a, b \in \Lambda.\ \mathbf{LN}[a[x := b]] \equiv \mathbf{LN}[a][x_\imath := \mathbf{LN}[b]]$.
(ii) $\forall a, b \in \Lambda.\ a \rightarrow_\beta b \Rightarrow \mathbf{LN}[a] \overset{2}{\rightarrow}_{\beta N} \mathbf{LN}[b]$.
(iii) $\forall a \in \Lambda, c \in \Lambda_N.\ \mathbf{LN}[a] \rightarrow_{\beta N} c \Rightarrow \exists b \in \Lambda.\ [a \rightarrow_\beta b] \wedge [c \rightarrow_{\beta N} \mathbf{LN}[b]]$.

Proof.

(i) Induction on a.

(ii) Let $(\lambda x.a_1)a_2$ be the redex involved in the reduction step $a \rightarrow_\beta b$, with contractum $a_1[x := a_2]$. This has a corresponding redex $(\lambda x.\mathbf{LN}[a_1])(\imath = \mathbf{LN}[a_2])!$ in $\mathbf{LN}[a]$. After a bind reduction and a close reduction we get $\mathbf{LN}[a_1][x_\imath := \mathbf{LN}[a_2]][x_* := \varepsilon]$. Since no other label than $\imath$ is used in the translation, the second substitution has no effect. Then by (i) the result is equivalent to $\mathbf{LN}[a_1[x := a_2]]$.

(iii) Every initial redex in $\mathbf{LN}[a]$ comes from some redex $(\lambda x.a_1)a_2$ in a, and therefore is necessarily of shape $(\lambda x.\mathbf{LN}[a_1])(\imath = \mathbf{LN}[a_2])!$ Hence the first reduction step must be a bind reduction, yielding a new redex $(\lambda x.\mathbf{LN}[a_1][x_\imath := \mathbf{LN}[a_2]])!$. After performing the close reduction, we get the exact image of the contractum $a_1[x := a_2]$. Hence, b is obtained by contraction of the redex $(\lambda x.a_1)a_2$ in a. □

So in the following we will freely use classical λ-calculus syntax – unlabelled variables x and application constructs (ab) – within λN expressions, assuming this translation to be implicit.

2.3. Example: Boolean values and extensibility

Some intuition about the calculus will be given through an encoding of boolean values. Remember that in the classical λ-calculus, Church encoded **true** as $\lambda xy.x$, **false** as $\lambda xy.y$, and **not** as $\lambda xyz.xzy$. The distinction between truth values is based on the position of variables (ordering of λ abstractions). The same approach could be used in λN, but another solution for distinguishing truth values is to exploit the orthogonal dimension provided by names:

$$\begin{aligned}
\mathbf{true} &\stackrel{\text{def}}{=} \lambda x.x_{\mathsf{true}} \\
\mathbf{false} &\stackrel{\text{def}}{=} \lambda x.x_{\mathsf{false}} \\
\mathbf{not} &\stackrel{\text{def}}{=} \lambda x.x(\mathsf{true} = \mathbf{false})(\mathsf{false} = \mathbf{true})! \\
&\equiv \lambda x.x_i(\mathsf{true} = \lambda x.x_{\mathsf{false}})(\mathsf{false} = \lambda x.x_{\mathsf{true}})!
\end{aligned}$$

In contrast with the Church encoding, the boolean values here use only one abstraction level (one single λ), but access the corresponding variable through different names. The advantage is extensibility: additional names can be used for additional values, without changing the basic protocol. For example, a three-valued logic, with an additional *unknown* value and a corresponding redefinition of the *not* operation, is obtained as follows:

$$\begin{aligned}
\mathbf{unknown} &\stackrel{\text{def}}{=} \lambda x.x_{\mathsf{unknown}} \\
\mathbf{notU} &\stackrel{\text{def}}{=} \lambda x.\mathbf{not}(x(\mathsf{unknown} = \mathbf{unknown})) \\
&\equiv \lambda x.\mathbf{not}(i = x_i(\mathsf{unknown} = \mathbf{unknown}))!
\end{aligned}$$

No recoding of **true** and **false** is needed, while in the standard Church encoding it would be necessary to recode them as functions with three abstraction levels instead of two. Furthermore, **notU** is defined *incrementally* as an extension of the previous **not** function.

To illustrate the reduction rules, here is a "standard reduction" (reducing leftmost outermost redex first) of the expression **not true**:

$$\begin{aligned}
\mathbf{not\ true} \quad = \quad & (\lambda x.x_i(\mathsf{true} = \lambda x.x_{\mathsf{false}})(\mathsf{false} = \lambda x.x_{\mathsf{true}})!) \\
& \qquad (i = (\lambda x.x_{\mathsf{true}}))! \\
\rightarrow_{\beta N} \ & (\lambda x'.(\lambda x.x_{\mathsf{true}})(\mathsf{true} = \lambda x.x_{\mathsf{false}})(\mathsf{false} = \lambda x.x_{\mathsf{true}})!)!
\end{aligned}$$

$$\rightarrow_{\beta N} (\lambda x.x_{\mathsf{true}})(\mathsf{true} = \lambda x.x_{\mathsf{false}})(\mathsf{false} = \lambda x.x_{\mathsf{true}})!$$
$$\rightarrow_{\beta N} (\lambda x'.\lambda x.x_{\mathsf{false}})(\mathsf{false} = \lambda x.x_{\mathsf{true}})!$$
$$\rightarrow_{\beta N} (\lambda x'.\lambda x.x_{\mathsf{false}})!$$
$$\rightarrow_{\beta N} \lambda x.x_{\mathsf{false}}$$

so the result is indeed **false**. Similarly, it can be verified easily that **notU unknown** yields **unknown**, or that **notU false** yields **true**. By contrast, consider what happens with an erroneous expression like **not unknown**:

$$\begin{aligned}
\textbf{not unknown} \quad = \quad & (\lambda x.x_\iota(\mathsf{true} = \lambda x.x_{\mathsf{false}})(\mathsf{false} = \lambda x.x_{\mathsf{true}})!) \\
& \qquad (\iota = (\lambda x.x_{\mathsf{unknown}}))! \\
\rightarrow_{\beta N} \ & (\lambda x'.(\lambda x.x_{\mathsf{unknown}})(\mathsf{true} = \lambda x.x_{\mathsf{false}})(\mathsf{false} = \lambda x.x_{\mathsf{true}})!)! \\
\rightarrow_{\beta N} \ & (\lambda x.x_{\mathsf{unknown}})(\mathsf{true} = \lambda x.x_{\mathsf{false}})(\mathsf{false} = \lambda x.x_{\mathsf{true}})! \\
\rightarrow_{\beta N} \ & (\lambda x.x_{\mathsf{unknown}})(\mathsf{false} = \lambda x.x_{\mathsf{true}})! \\
\rightarrow_{\beta N} \ & (\lambda x.x_{\mathsf{unknown}})! \\
\rightarrow_{\beta N} \ & \varepsilon
\end{aligned}$$

2.4. *Confluence*

We show confluence of the calculus through an adaptation of Takahashi's proof for the usual λ-calculus [26]; this proof itself is a simplification of the well-known Tait method, using parallel reductions. The idea is to define a relation over terms which simultaneously contracts several (possibly overlapping) redexes and show that this relation has the diamond property.

Definition 2. Parallel reduction, denoted $\Rightarrow_{\beta N}$, is defined inductively by the following rules:

$$(\textit{refl})\ x_l \Rightarrow_{\beta N} x_l \text{ and } \varepsilon \Rightarrow_{\beta N} \varepsilon;$$

if $a \Rightarrow_{\beta N} a'$ and $b \Rightarrow_{\beta N} b'$, then:

$$(\textit{err})\ \lambda x.\varepsilon \Rightarrow_{\beta N} \varepsilon \text{ and } \varepsilon(l = b) \Rightarrow_{\beta N} \varepsilon \text{ and } \varepsilon! \Rightarrow_{\beta N} \varepsilon$$
$$(\textit{congr})\ \lambda x.a \Rightarrow_{\beta N} \lambda x.a' \text{ and } a! \Rightarrow_{\beta N} a'!$$
$$\text{and } a(l = b) \Rightarrow_{\beta N} a'(l = b'),$$
$$(\textit{lam-bind})\ (\lambda x.a)(l = b) \Rightarrow_{\beta N} \lambda x.a'[x_l := b'],$$
$$(\textit{lam-close})\ (\lambda x.a)! \Rightarrow_{\beta N} a'[x_* := \varepsilon];$$

Parallel reduction obeys the following properties:

Lemma 3. (i) $a \to_{\beta N} b \Rightarrow a \Rightarrow_{\beta N} b$
(ii) $a \Rightarrow_{\beta N} b \Rightarrow a \overset{*}{\to}_{\beta N} b$
(iii) $[a \Rightarrow_{\beta N} a'] \wedge [b \Rightarrow_{\beta N} b'] \Rightarrow a[x_l := b] \Rightarrow_{\beta N} a'[x_l := b']$.

Proof. (1) induction on the context of the redex; (2) and (3) induction on a. □

Following Takahashi, instead of proving that $\Rightarrow_{\beta N}$ has the diamond property, we prove a stronger statement:

Lemma 4. $[a \Rightarrow_{\beta N} b] \Rightarrow [b \Rightarrow_{\beta N} a^*]$, *where* a^* *is a term determined by* a, *according to the following inductive definition*:

$$x_l^* = x_l$$

$$\varepsilon^* = \varepsilon$$

$$(\lambda x.a)^* = (a(l = b))^* = (a!)^* = \varepsilon \quad \text{if } a \equiv \varepsilon$$

if $a \not\equiv \varepsilon$, *then*

$$(\lambda x.a)^* = \lambda x.a^*$$

$$(a(l = b))^* = \begin{cases} \lambda x.(a'^*[x_l := b^*]) & \text{if } a \equiv \lambda x.a' \\ a^*(l = b^*) & \text{otherwise} \end{cases}$$

$$(a!)^* = \begin{cases} a'^*[x_* := \varepsilon] & \text{if } a \equiv \lambda x.a' \\ a^*! & \text{otherwise.} \end{cases}$$

Proof. Induction on a:
- case $a \equiv x_l$ or $a \equiv \varepsilon$: trivial
- case $a \equiv \lambda x.a'$: either $b \equiv \varepsilon$, which again is trivial, or b must be of the form $\lambda x.b'$, with $a' \Rightarrow_{\beta N} b'$. By induction hypothesis $b' \Rightarrow_{\beta N} a'^*$, which implies $b \Rightarrow_{\beta N} a^*$.
- case $a \equiv a_1(l = a_2)$: if $a_1 \equiv \varepsilon$ the result is trivial. If $a_1 \equiv \lambda x.a_1'$, then b is obtained either through rule *congr* or rule *lam-bind*, and therefore must be of the form $(\lambda x.b_1)(l = b_2)$ or $\lambda x.b_1[x_l := b_2]$, for some b_1, b_2 with $a_1' \Rightarrow_{\beta N} b_1, a_2 \Rightarrow_{\beta N} b_2$. By induction hypothesis, we have $b_1 \Rightarrow_{\beta N} a_1'^*, b_2 \Rightarrow_{\beta N} a_2^*$. Hence in either case $b \Rightarrow_{\beta N} \lambda x.a_1'^*[x_l := a_2^*] \equiv a^*$. Finally, if a_1 is neither an error nor a λ-abstraction, then b must be of the form $b_1(l = b_2)$, with $a_1 \Rightarrow_{\beta N} b_1, a_2 \Rightarrow_{\beta N} b_2$. By induction hypothesis $b_1 \Rightarrow_{\beta N} a_1^*$ and $b_2 \Rightarrow_{\beta N} a_2^*$, so $b \Rightarrow_{\beta N} a^*$.
- case $a \equiv a'!$: like preceding case. □

Theorem 5 (Confluence).

$$[a \overset{*}{\to}_{\beta N} b] \wedge [a \overset{*}{\to}_{\beta N} c] \Rightarrow \exists d.[b \overset{*}{\to}_{\beta N} d] \wedge [c \overset{*}{\to}_{\beta N} d]$$

Proof. By Lemma 3, $\overset{*}{\to}_{\beta N}$ is the reflexive, transitive closure of $\Rightarrow_{\beta N}$, so to prove confluence it suffices to show that $\Rightarrow_{\beta N}$ has the diamond property:

$$[a \Rightarrow_{\beta N} b_1] \wedge [a \Rightarrow_{\beta N} b_2] \Rightarrow \exists a'.[b_1 \Rightarrow_{\beta N} a'] \wedge [b_2 \Rightarrow_{\beta N} a']$$

But Lemma 4 shows that such an a' does exist, and is uniquely determined by a. □

2.5. Term ordering

Intuitively, **notU** can safely be used instead of **not** in any context: it behaves as **not** on **true** and **false** arguments, and behaves "better" on **unkown** argument (yielding **unknown** again, while **not** yields an error). To formalize this idea, we observe the error generation behaviour of terms, first in arbitrary contexts, and then in applicative contexts (applicative bisimulation). In both cases, a statement $a \sqsubseteq b$ can be read intuitively as "a is better than b", or "a generates less errors than b". As in the standard λ-calculus, λN is *operationally extensional*, which means that the two orderings coincide.

Following common definitions, a *context* is obtained by extending the syntax with $[-]$ (a *hole*). For any context $C[-]$, $C[a]$ is the term obtained by plain syntactic substitution of a for every occurrence of the hole in $C[-]$, i.e. with possible capture of free variables of a.

Definition 6. The *contextual order* $\sqsubseteq^{\text{ctxt}}$ on Λ_N is defined as

$$a \sqsubseteq^{\text{ctxt}} b \Leftrightarrow \forall C[-].[C[a] \overset{*}{\to}_{\beta N} \varepsilon \Rightarrow C[b] \overset{*}{\to}_{\beta N} \varepsilon]$$

Definition 7. An *applicative operation* o is either a bind operation ($l = b$), with b closed, or a close operation (!). An *applicative sequence* $\vec{o}$ is a finite list of applicative operations. A *closing substitution* σ for terms a and b is a finite list of atomic substitutions such that both $a\sigma$ and $b\sigma$ are closed. The *applicative order* $\sqsubseteq^{\text{appl}}$ on Λ_N is defined as:

(i) for closed a, b: $a \sqsubseteq^{\text{appl}} b \Leftrightarrow \forall \vec{o}.[a\vec{o} \overset{*}{\to}_{\beta N} \varepsilon \Rightarrow b\vec{o} \overset{*}{\to}_{\beta N} \varepsilon]$

(ii) for arbitrary a, b: $a \sqsubseteq^{\text{appl}} b \Leftrightarrow \forall \sigma. a\sigma \sqsubseteq^{\text{appl}} b\sigma$.

Lemma 8 (Context lemma, operational extensionality).

$$\forall a, b.\ a \sqsubseteq^{\text{ctxt}} b \Leftrightarrow a \sqsubseteq^{\text{appl}} b$$

Proof. The proof is inspired by Abramsky and Ong's proof of operational extensionality for the lazy λ-calculus [1]. Since an applicative context is a context, $\sqsubseteq^{\text{ctxt}}$ implies $\sqsubseteq^{\text{appl}}$, so we are left with the reverse implication. The proof proceeds by induction on the length of computation. Assuming (without loss of generality) that a

and b are closed, $a \sqsubseteq^{\text{appl}} b \Rightarrow a \sqsubseteq^{\text{ctxt}} b$ can be rewritten as

$$(\forall \vec{o}.a\vec{o} \xrightarrow{*}_{\beta N} \varepsilon \Rightarrow b\vec{o} \xrightarrow{*}_{\beta N} \varepsilon) \Rightarrow$$

$$(\forall n.\forall C[-].C[a] \xrightarrow{n}_{\beta N} \varepsilon \Rightarrow C[b] \xrightarrow{*}_{\beta N} \varepsilon).$$

The base case (when $n=0$ and $C[-] \equiv [-]$) is obvious. For the inductive step, consider without loss of generality the following cases of closed contexts:

(i) $C[-] \equiv (\lambda x.A[-])(l = B[-])\vec{O}[-]$
(ii) $C[-] \equiv (\lambda x.A[-])!\vec{O}[-]$
(iii) $C[-] \equiv (\lambda x.A[-])$
(iv) $C[-] \equiv \varepsilon\vec{O}[-]$
(v) $C[-] \equiv [-](l = B[-])\vec{O}[-]$
(vi) $C[-] \equiv [-]!\vec{O}[-]$
(vii) $C[-] \equiv [-]$

where $\vec{O}[-]$ is an applicative sequence of contexts. We proceed by case analysis:

(i) Suppose $C[a] \xrightarrow{n}_{\beta N} \varepsilon$. This is only possible if the binding $(l = B[-])$ is reduced by rule *lambda-bind* or *err-bind*, which implies either $\lambda x.A[a] \xrightarrow{k}_{\beta N} \varepsilon$ or $(\lambda x.A[a][x_l := B[a]])\vec{O}[a] \xrightarrow{k}_{\beta N} \varepsilon$, for some $k < n$. By induction hypothesis, either $\lambda x.A[b] \xrightarrow{*}_{\beta N} \varepsilon$, or $(\lambda x.A[b][x_l := B[b]])\vec{O}[b] \xrightarrow{*}_{\beta N} \varepsilon$, which implies $C[b] \xrightarrow{*}_{\beta N} \varepsilon$.

(ii) Like the previous case. To have $C[a] \xrightarrow{n}_{\beta N} \varepsilon$, we must have $\lambda x.A[a] \xrightarrow{k}_{\beta N} \varepsilon$ or $(A[a][x_* := \varepsilon])\vec{O}[a] \xrightarrow{k}_{\beta N} \varepsilon$, for some $k < n$, so again we can make an appeal to the induction hypothesis to show $C[b] \xrightarrow{*}_{\beta N} \varepsilon$.

(iii) Any proof of $C[a] \xrightarrow{n}_{\beta N} \varepsilon$ must end by rule *lambda-err*, so $A[a] \xrightarrow{n-1}_{\beta N} \varepsilon$; then $A[b] \xrightarrow{*}_{\beta N} \varepsilon$ by induction, and $C[b] \xrightarrow{*}_{\beta N} \varepsilon$.

(iv) Trivial.

(v) Since a is closed, it must be either of form of form $\varepsilon\vec{o}$, which is trivial, or of form $(\lambda x.a')\vec{o}$. Define $D[-] \equiv a(l = B[-])\vec{O}[-]$. $D[-]$ is a context of case (i) or (ii), and $D[a] \equiv C[a]$. By an appeal to the appropriate case, we show $D[b] \xrightarrow{*}_{\beta N} \varepsilon$. But since $D[b] \equiv a(l = B[b])\vec{O}[b]$, and since $a \sqsubseteq^{\text{appl}} b$, we can conclude $b(l = B[b])\vec{O}[b] \xrightarrow{*}_{\beta N} \varepsilon$, i.e. $C[b] \xrightarrow{*}_{\beta N} \varepsilon$.

(vi) Like the previous case.

(vii) Immediate, since the left part of the implication says that if $a \xrightarrow{*}_{\beta N} \varepsilon$ in the empty applicative context, then we must have $b \xrightarrow{*}_{\beta N} \varepsilon$. □

Since the two orderings coincide, we will simply write $\sqsubseteq$, and $\approx$ for its symmetric closure. $\sqsubseteq$ has the following properties:

Lemma 9. (i) $\sqsubseteq$ *is a precongruence, i.e.* $\forall C[-], a \sqsubseteq b \Rightarrow C[a] \sqsubseteq C[b]$.

(ii) $a \leftrightarrow_{\beta N} b \Rightarrow a \approx b$.

(iii) *For any term* a, $\Omega \sqsubseteq a \sqsubseteq \varepsilon$, *where* $\Omega \equiv \Delta\Delta$, *and* $\Delta \equiv \lambda x.xx$.

Proof. (i) Immediate from the definition of $\sqsubseteq^{\text{ctxt}}$. (ii) immediate from the definition of $\sqsubseteq^{\text{appl}}$. (iii) for any applicative sequence $\vec{o}, \varepsilon\ \vec{o} \xrightarrow{*}_{\beta N} \varepsilon$ and $\Omega\vec{o} \xrightarrow{*}_{\beta N} \Omega$, and $\neg(\Omega \xrightarrow{*}_{\beta N} \varepsilon)$. □

Also observe that this theory identifies all unsolvable terms different from ε, so in particular $\Omega \approx \lambda x_1 \ldots x_n.\Omega$ for any n. Moreover, with respect to the classical λ-calculus it has the particular property of possessing infinite descending chains if $\mathcal{N}$ is infinite, like for example

$$\lambda x.x! \sqsupseteq \lambda x.x(l_1 = a_1)! \sqsupseteq \lambda x.(l_1 = a_1)(l_2 = a_2)! \sqsupseteq \cdots$$

for any sequence of terms $a_1, a_2, \ldots$ different from ε.

The ordering is immediately useful for verifying some general program laws:

Theorem 10 (Program laws). $\forall a, b, c$,

(i) (1) $(l_1 \not\equiv l_2) \Rightarrow a(l_1 = b)(l_2 = c) \approx a(l_2 = c)(l_1 = b)$
(2) $a(l = b)(l = c) \approx a(l = b)$

(ii) (1) $a(l = b)! \sqsubseteq a!$
(2) $a(l = \varepsilon)! \approx a!$

(iii) (1) $x \notin FV(a) \Rightarrow a \sqsubseteq \lambda x.a(l_1 = x_{l_1}) \ldots (l_n = x_{l_n})!$
(2) $x \notin FV(a), FN(a, y) \subseteq \{l_1, \ldots l_n\} \Rightarrow \lambda y.a \approx \lambda x y.a(l_1 = x_{l_1}) \ldots (l_n = x_{l_n})!$

Proof. (i) Suppose a closed. If $a \approx \Omega$ or $a \approx \varepsilon$, the result is immediate; otherwise, we must have $a \xrightarrow{*}_{\beta N} (\lambda x.a')$. Then $a(l_1 = b)(l_2 = c) \xrightarrow{*}_{\beta N} \lambda x.a'[x_{l_1} := b][x_{l_2} := c]$. Since substitutions are capture-free, there is no occurrence of variable x in either b or c; hence, if $l_1 \not\equiv l_2$, then $\lambda x.a'[x_{l_1} := b][x_{l_2} := c] \equiv \lambda x.a'[x_{l_2} := c][x_{l_1} := b]$, and therefore $a(l_1 = b)(l_2 = c) \leftrightarrow_{\beta N} a(l_2 = c)(l_1 = b)$; similarly, if $l_1 \equiv l_2$, then $a(l_1 = b)(l_2 = c) \leftrightarrow_{\beta N} a(l_1 = b)$. The general case follows directly, by quantifying over closing substitutions for a.

(ii) (1) Again suppose a closed, and $a \xrightarrow{*}_{\beta N} (\lambda x.a')$. Then $a(l = b)! \xrightarrow{*}_{\beta N} a'[x_l := b][x_* := \varepsilon]$. Since $b \sqsubseteq \varepsilon$, and $\sqsubseteq$ is a precongruence, we have $a'[x_l := b][x_* := \varepsilon] \sqsubseteq a'[x_* := \varepsilon] \leftrightarrow_{\beta N} a!$. In the particular case (2) where $b \equiv \varepsilon$, $a'[x_l := \varepsilon][x_* := \varepsilon] \equiv a'[x_* := \varepsilon]$ so we have an equivalence.

(iii) (1) Let $\overline{(l_i = x_{l_i})}$ abbreviate $(l_1 = x_{l_1}) \ldots (l_n = x_{l_n})$. Using the definition of the applicative order, we have to prove that for any applicative sequence $\vec{o}$, $a\vec{o} \xrightarrow{*}_{\beta N} \varepsilon \Rightarrow (\lambda x.a\overline{(l_i = x_{l_i})}!)\vec{o} \xrightarrow{*}_{\beta N} \varepsilon$. The proof proceeds by induction on the length of $\vec{o}$. If $\vec{o}$ is empty, the result is immediate through reduction rules for errors. Otherwise, if $\vec{o} \equiv (!)\vec{o'}$, then

$$a!\vec{o'} \sqsubseteq a\overline{(l_i = \varepsilon)}!\vec{o'} \quad \text{(law (ii))}$$

$$\leftrightarrow_{\beta N} (\lambda x.a\overline{(l_i = x_{l_i})}!)!\vec{o'} \quad \text{if } x \notin FV(a)$$

Finally, if $\vec{o} \equiv (l = b)\vec{o'}$, then

$$\begin{aligned} a(l=b)\vec{o'} &\sqsubseteq (\lambda x.a(l=b)\overline{(l_i = x_{l_i})}!)\vec{o'} && \text{(induction hyp.)} \\ &\leftrightarrow_{\beta N} (\lambda x.a(l=x_l)\overline{(l_i = x_{l_i})}!)(l=b)\vec{o'} \\ &\sqsubseteq (\lambda x.a\overline{(l_i = x_{l_i})}!)(l=b)\vec{o'} && \text{(law (ii))} \end{aligned}$$

The proof for (2) proceeds similarly. □

By allowing any permutation of bindings on different names, the first part of this theorem justifies our intuition that each argument name acts as a channel which operates in parallel with the others; in other words, functions receive arguments simultaneously at different names. On the other hand, each name/channel can sequentially receive several values, but only the first is taken into account. The second part of the theorem says that any binding *immediately before a close operation* does no harm, i.e. supplying information at a given name can at best be useful, can at worse be ignored, but will never generate more run-time errors. Finally, the last part is the counterpart of η-equality in the traditional λ-calculus. Here we only have an inequality instead of full equality in (iii)(1), because the finite set of bindings $(l_1 = x_{l_1}) \ldots (l_n = x_{l_n})$ cannot cover all possible arguments to a. So in the general case we can safely perform η-reductions, but not η-expansions. However, part (iii)(2) says that if we can guarantee that the set $\{l_1 \ldots l_n\}$ covers all names used at the top-level abstraction, then both sides are equivalent.

An easy application of the theorem above is to show that **notU** $\sqsubseteq$ **not**:

$$\begin{aligned} \textbf{notU} &\equiv \lambda x.(\lambda y.y_i(\mathsf{true} = \textbf{false})(\mathsf{false} = \textbf{true})!) \\ &\qquad (i = x_i(\mathsf{unknown} = \textbf{unknown}))! \\ &\xrightarrow{*}_{\beta N} \lambda x.x_i(\mathsf{unknown} = \textbf{unknown})(\mathsf{true} = \textbf{false})(\mathsf{false} = \textbf{true})! \\ &\approx \lambda x.x_i(\mathsf{true} = \textbf{false})(\mathsf{false} = \textbf{true})(\mathsf{unknown} = \textbf{unknown})! \\ &\sqsubseteq \lambda x.x_i(\mathsf{true} = \textbf{false})(\mathsf{false} = \textbf{true})! \\ &\equiv \textbf{not} \end{aligned}$$

3. Simple type assignment

This section considers an adaptation of Curry's simple type assignment system to λN. The syntax of simple types is:

$$T ::= \top \mid X \mid P \to T$$

$$P ::= (l_1 : T_1, \ldots, l_n : T_n) \quad (\text{all } l_i \text{ distincts})$$

where $\top$ is a type constant (the type of anything, including errors), X is a type variable, and $P \to T$ is an arrow type mapping a *parameter type* to a type. Parameter types are

finite associations of names to types. Any name not explicitly mentioned in the set is implicitly associated with $\top$; therefore the empty parameter type, written $()$, maps every name to $\top$. $P(l)$ denotes the type associated to name l in parameter type P, and $P\backslash l$ denotes the parameter type P in which name l has been remapped to $\top$; more formally,

$$P(l) = \begin{cases} T_i & \text{if } (l_i : T_i) \in P, l = l_i \\ \top & \text{otherwise} \end{cases}$$

$$P\backslash l = \{(l' : T) | (l' : T) \in P, l' \neq l\}$$

Parameter types are treated modulo the following syntactic equivalence relationship:

$$P_1 \equiv P_2 \Leftrightarrow \forall l. P_1(l) \equiv P_2(l)$$

This says that declarations in parameter types can be arbitrarily permuted, and that declarations of the form $l : \top$ can be added or removed. We write $dom(P)$ for $\{l \mid P(l) \not\equiv \top\}$.

We will use the letters T, U, V for types, P, Q for parameter types, and X, Y, Z for type variables. As usual, arrow types associate to the right. Furthermore, we adopt a syntactic sugar convention for types which corresponds to the similar convention for terms in Section 2.2: type expressions of the form $T \to U$ are abbreviations for types of form $(\imath : T) \to U$; these are arrow types in which the left-hand side is a parameter type mapping all names to $\top$, except for the invisible name $\imath$. Thanks to this convention, the types of usual lambda terms (i.e. terms which do not contain names other than $\imath$) look exactly like in the usual lambda calculus.

Types are ordered through a subtyping relationship given in Fig. 2. Obviously, we write $T = U$ iff $T \leqslant U$ and $U \leqslant T$. Observe that the rule for arrow types is covariant on the right and contravariant on the left of the arrow, as usual in type systems with subtyping. The rule *top-arrow* is motivated by the reduction rules for errors: $\varepsilon \vec{o} \xrightarrow{*}_{\beta N} \varepsilon$ for any applicative sequence $\vec{o}$, so $\top$ is equal to any functional type ending with $\top$.

Lemma 11. *The subtyping relation is reflexive and transitive.*

Proof. Easy induction on the structure of types. □

A *basis* Γ is a finite association of variables to parameter types; the set of variables for which a parameter type is associated in Γ is denoted $dom(\Gamma)$. If $x : P \in \Gamma$, then Γ associates type $P(l)$ to labelled variable x_l; this is sometimes written $\Gamma(x_l)$. Furthermore, $\Gamma, x : P$ denotes the extension of basis Γ with association $x : P$ (assuming $x \notin dom(\Gamma)$). Typing judgements for the Curry simple type system are of the form $\Gamma \vdash a : T$, saying that "a has type T in basis Γ". Such judgements are derived from the rules of Fig. 3. This type system has an unusual aspect in comparison with many other systems, where each type constructor has one introduction and one elimination rule. Here the arrow type is introduced by rule *abs*, but is eliminated in several steps: a type $(l_1 : T_1, \ldots,$

$$\frac{}{T \leqslant \top}\ (top) \qquad \frac{}{\top \leqslant P \to \top}\ (top-arrow) \qquad \frac{}{X \leqslant X}\ (tvar)$$

$$\frac{P_2 \leqslant P_1 \qquad T_1 \leqslant T_2}{P_1 \to T_1 \leqslant P_2 \to T_2}\ (arrow)$$

$$\frac{\forall l \in dom(P_1) \cup dom(P_2).\ P_1(l) \leqslant P_2(l)}{P_1 \leqslant P_2}\ (ptype)$$

Fig. 2. Subtyping rules.

$$\frac{}{\Gamma \vdash a : \top}\ (top) \qquad \frac{\Gamma \vdash a : T \qquad T \leqslant U}{\Gamma \vdash a : U}\ (subs)$$

$$\frac{}{\Gamma, x : P \vdash x_l : P(l)}\ (var) \qquad \frac{\Gamma, x : P \vdash a : T}{\Gamma \vdash \lambda x.a : P \to T}\ (abs)$$

$$\frac{\Gamma \vdash a : P \to T \qquad \Gamma \vdash b : P(l)}{\Gamma \vdash a(l = b) : P \backslash l \to T}\ (bind)$$

$$\frac{\Gamma \vdash a : () \to T}{\Gamma \vdash a! : T}\ (close)$$

Fig. 3. Typing rules.

$l_n : T_n) \to T$ is progressively reduced to $() \to T$ through multiple invocations of the *bind* rule; only then can it be eliminated through the *close* rule. This is obviously related to the asymmetry between lambda abstractions, which introduce several named parameters at the same time, and the bind and close constructs, which supply parameters in several steps.

Example 12. The boolean values of Section 2.3 have the following types:

true : (true : X) $\to X$

false : (false : X) $\to X$

not : ((false : (true : X) $\to X$, true : (false : Y) $\to Y$) $\to Z$) $\to Z$

Like Curry's original system, this type assignment system assigns many types to any given term. For some simple cases like the examples above it is possible to find a *principal* type, i.e. a type such that all other possible types for the same term can

be generated by subsumption and substitution of type variables. However, this is not generally true: in particular, open sequences of bind operations may have many types. For example if $a:T_1$, then $\lambda x.x(l_1=a)$ has type $((l_1:T_1)\to X)\to()\to X$, but more generally it has all types of the shape

$$((l_1:T_1,\ldots,l_n:T_n)\to X)\to(l_2:T_2,\ldots,l_n:T_n)\to X$$

for $n\geqslant 1$. In order to capture all of these in a single type scheme, it is necessary to resort to a more powerful type system, like the one studied in [10], which uses a mechanism similar to the "row variables" of record calculi [23, 27].

The type system posesses the usually expected properties: types are preserved by computation, and terms with type different from $\top$ do not reduce to ε.

Theorem 13 (Subject reduction).

$$[[\Gamma\vdash a:T]\wedge[a\xrightarrow{*}_{\beta N}a']]\ \Rightarrow\ \Gamma\vdash a':T$$

Proof. Induction on the length of $a\xrightarrow{*}_{\beta N}a'$, following standard techniques. A complete development is given in [10]. □

Lemma 14. *If* $\Gamma\vdash\varepsilon:T$, *then* $T=\top$.

Proof. Proofs of $\varepsilon:T$ can only use the *top* axiom and subsumption. Since subtyping is transitive (Lemma 11), we must have $\top\leqslant T$, and therefore $T=\top$. □

Theorem 15 (Soundness). *A closing substitution* σ satisfies *a basis* Γ, *written* $\sigma\models\Gamma$, *iff*, $\forall x\in dom(\Gamma),\forall l\in dom(\Gamma(x)),\ \vdash x_l\sigma:\Gamma(x_l)$. *Then*

$$[[\Gamma\vdash a:T]\wedge[T\neq\top]]\ \Rightarrow\ [\forall\sigma\models\Gamma,\neg(a\sigma\xrightarrow{*}_{\beta N}\varepsilon)]$$

Proof. Direct consequence of Theorem 13 and Lemma 14. □

4. Modelling record operations

In this section we first show that records (i.e. named products) have a very natural encoding in λN, and then discuss the relationship of λN with record calculi. The dual notion of named coproducts will be treated in Section 6.3.

4.1. Products and records

First let us briefly recall how n-tuples (products) can be encoded in the usual lambda-calculus:

$$(a_1,\ldots,a_n)\equiv\lambda x.\,x\,a_1\ldots a_n\quad\left(x\notin\bigcup_{i=1}^{n}FV(a_i)\right).$$

This is a straightforward generalization of the well-known Church encoding for pairs. The x argument to the λ-abstraction is used to receive a selector function, which will extract the desired values from the tuple. Projection of the product, i.e. selection of the ith component of a n-tuple, is performed by passing to the tuple a function of the form $\lambda x_1 \ldots x_n.x_i$. Similarly, we have in λN:

Definition 16 (*Records*).

$$\{l_1 = a_1, \ldots, l_n = a_n\} \stackrel{\text{def}}{=} \lambda x.x(l_1 = a_1)\ldots(l_n = a_n)! \quad \left(x \notin \bigcup_{i=1}^{n} FV(a_i)\right)$$

$$a.l \stackrel{\text{def}}{=} a\,(\lambda x.x_l)$$

Here again, the λ-abstraction has an argument x to receive a "selector", to which all named fields of the record are bound, with a closing "!". By Theorem 10, any permutation of the bindings yields an observationally equivalent function, so this justifies the fact that the order of fields in a record is irrelevant. Furthermore, by the second part of the same theorem, we have

$$\lambda x.x(l_1 = a_1)\ldots(l_n = a_n)(l_{n+1} = a_{n+1})\ldots(l_{n+k} = a_{n+k})! \sqsubseteq \lambda x.x(l_1 = a_1)\ldots(l_n = a_n)!$$

which proves that a record with more fields can always be used in place of a record with fewer fields. This property of records is called *width subsumption*; depth subsumption, i.e. the possibility to replace the value a_i at some field by some other value a_i' iff $a_i' \sqsubseteq a_i$, is derived directly from the fact that $\sqsubseteq$ is a precongruence.

Selection of field l in a record r is performed by passing to the record a function of shape $\lambda x.x_l$, i.e. an identity function on name l. It can be verified easily that

$$\{l_1 = a_1, \ldots, l_n = a_n\}.l_i \equiv (\lambda x.x(l_1 = a_1)\ldots(l_n = a_n)!)(\lambda x.x_{l_i}) \stackrel{*}{\rightarrow}_{\beta N} a_i$$

Here the selector $\lambda x.x_{l_i}$ only accesses one field at a time, which is the common way to project labelled products in most record calculi. However, the "selector" function passed to a record could as well depend on several names, i.e. access several fields simultaneously. For example, the function

$$\lambda r.r(\lambda x.x_{\mathsf{fun}}\, x_{\mathsf{arg}})$$

expects a record argument r, extracts from that record the fields fun and arg, and applies the first to the second. This is like a "quoted expression" in LISP, which gets evaluated within the environment supplied by the record; further examples will be displayed in Section 6.

4.2. Record extensions

Records are encoded as a list of bindings immediately followed by a close operation. If the close operation is removed, a different kind of components is obtained, which we will call *record extensions*, denoted by curly braces with an initial "+":

Definition 17 (*Record extensions*).

$$\{+l_1 = a_1, \ldots, l_n = a_n\} \stackrel{\text{def}}{=} \lambda x.x(l_1 = a_1)\ldots(l_n = a_n)$$

$$a \ll b \stackrel{\text{def}}{=} \lambda x.a(bx) \qquad (x \notin FV(a) \cup FV(b))$$

Record extensions have an open list of bindings, which can be completed later by another component. As a result, a record extension can be composed with a record, in order to prepend its own fields to the fields of the record. This operation is written $\ll$, and is exactly the same as usual functional composition. We can check by simple computation that composition indeed behaves as a concatenation operator:

$$\begin{aligned}
&\{l_1 = a_1, \ldots, l_n = a_n\} \ll \{+l'_1 = a'_1, \ldots, l'_m = a'_m\} \\
&\quad \equiv \lambda x.(\lambda y.y(l_1 = a_1)\ldots(l_n = a_n)!)((\lambda z.z(l'_1 = a'_1)\ldots(l'_m = a'_m))x) \\
&\quad \stackrel{*}{\to}_{\beta N} \lambda x.(\lambda y.y(l_1 = a_1)\ldots(l_n = a_n)!)(x(l'_1 = a'_1)\ldots(l'_m = a'_m)) \\
&\quad \stackrel{*}{\to}_{\beta N} \lambda x.x(l'_1 = a'_1)\ldots(l'_m = a'_m)(l_1 = a_1)\ldots(l_n = a_n)!
\end{aligned}$$

By part (i)(1) of Theorem 10, if the two sets of names $\{l_1 \ldots l_n\}$ and $\{l'_1 \ldots l'_m\}$ are disjoint, then all bindings in the result can be permuted. However, if some names appear in both sets, then, by part (i)(2) of the same theorem, the first binding takes precedence. In other words, if there are i, j such that $l'_i \equiv l_j$, then only the first binding $(l'_i = a'_i)$ is considered, and the other binding on the same name $(l_j = a_j)$ can be removed. As a consequence, the result is a new record in which fields $l'_1 \ldots l'_m$ have been added or overridden. A similar computation shows that two record extensions can be concatenated through the same operator $\ll$, yielding a new record extension. So if R is a record and RE_1, RE_2 are two record extensions, then $(R \ll RE_1) \ll RE_2 \approx R \ll (RE_1 \ll RE_2)$, which is not surprising since functional composition is associative.

Since record extensions do not contain a close operation, part (ii) of Theorem 10 does not apply, so record extensions do *not* support width subsumption: in other words,

$$\{+l_1 = a_1, \ldots, l_i = a_i, l_{i+1} = a_{i+1}, \ldots, l_n = a_n\} \not\subseteq \{+l_1 = a_1, \ldots, l_i = a_i\}$$

4.3. Typing record operations

If $a_1 : T_1, \ldots, a_n : T_n$, then $\lambda x.x(l_1 = a_1)\ldots(l_n = a_n)!$ has all types of the shape $((l_1 : T_1, \ldots, l_n : T_n) \to T) \to T$. This can be captured by a fresh type variable, so we could add the following derived rule for typing records:

$$\frac{\Gamma \vdash a_1 : T_1 \quad \ldots \quad \Gamma \vdash a_n : T_n}{\Gamma \vdash \{l_1 = a_1, \ldots, l_n = a_n\} : ((l_1 : T_1, \ldots, l_n : T_n) \to X) \to X}$$

Similarly, field selection is typed through the following derived rule:

$$\frac{\Gamma \vdash a : ((l : X) \to X) \to T}{\Gamma \vdash a.l : T}$$

Record extensions have more complex types because the collection of bindings has no final close operation. A family of derived typing rules can be expressed as

$$\frac{\Gamma \vdash a_1 : T_1 \quad \ldots \quad \Gamma \vdash a_n : T_n}{\Gamma \vdash \{+l_1 = a_1, \ldots, l_n = a_n\} : ((l_1 : T_1, \ldots, l_n : T_n, l_{n+1} : X_1, \ldots, l_{n+k} : X_k) \to Y) \to (l_{n+1} : X_1, \ldots, l_{n+k} : X_k) \to Y}$$

but, as explained in Section 3, no single principal type can capture all instances for different values of k. The more complex system of [10] uses families of type variables and therefore generates principal type

$$((l_1 : T_1, \ldots, l_n : T_n, * : X_*) \to Y) \to (* : X_*) \to Y$$

where $* : X_*$ implicitly assigns type X_l to any label $l \notin \{l_1 \ldots l_n\}$.

4.4. Comparison with record calculi

Motivated by research on theoretical foundations for objects [7], several record calculi have been studied in the literature. In the most simple cases, only record formation and field selection are supported, which provides no support for extensibility and inheritance. Other calculi provide a record concatenation operator, but with the important restriction that the two records being concatenated should have no fields in common [11]. The calculi of [24, 27] are more flexible: they use a construct of the form

$$a \textbf{ with } l = b$$

to extend or override field l in a with value b. This can be modelled directly in our setting as $a \ll \{+l = b\}$. Cancelling field l, as in [8], is equivalent to replacing it by the error term, i.e. $a \ll \{+l = \varepsilon\}$.

Conversely, one would imagine that λN is encodable in a suitable record calculus, by accumulating named parameters in records and then passing these to functions. An apparent hurdle is the way arguments accumulate in λN: recall from Theorem 10 that $a(l = b)(l = c) \approx a(l = b)$, so the first bind operation takes precedence, while the calculus of extensible records has the law

$$((a \textbf{ with } l = b) \textbf{ with } l = c) \approx (a \textbf{ with } l = c)$$

giving precedence to the second extension. However, the following translation, proposed by Didier Rémy, and directly inspired from [24], uses a kind of "continuation-passing

style" on records which reverses the order of record updates:

$$\begin{aligned}
[[x_l]] &= x.l \\
[[\lambda x.a]] &= \lambda x.[[a]] \\
[[a(l=b)]] &= \lambda x.[[a]](x \textbf{ with } l=[[b]]) \\
[[a!]] &= [[a]]\{\} \\
[[\varepsilon]] &= \varepsilon
\end{aligned}$$

Although quite close to the parameter-passing mechanism of λN, this translation is nevertheless not fully faithful. The precise technical study of this translation and comparison with various record calculi is left for another forthcoming paper; but the point where the translation fails can be shown easily through an example: consider $(\lambda x.x_l)(l=\Omega)$, which reduces to $(\lambda x.\Omega)\approx\Omega$. The translation is $\lambda y.(\lambda x.\ x.l)(y \textbf{ with } l=\Omega)$ which reduces to $\lambda y.(y \textbf{ with } l=\Omega).l$. This term is not equal to Ω, because it might yield ε if the argument y is ε.

5. Contexts and holes

This section relates λN with the meta-operation of hole filling in the classical λ-calculus, and with other calculi involving names or contexts.

5.1. Open λ-terms and contexts

We have already shown how usual λ-terms can be embedded in the λN calculus. Here we will extend the translation so that both *open terms* (terms containing free variables) and *contexts* (terms containg "holes") are encoded as λN terms. A higher-order operation in λN expresses the context-filling operation with (intended!) capture of variables.

Definition 18. Let **h** and **e** be two reserved variables in $\mathscr{V}$, i.e. not used anywhere except for the following translation. Furthermore, assume that the set of variables from the classical λ-calculus is contained in both $\mathscr{V}$ and $\mathscr{N}$. Then the translation $\mathbf{LNCt}[-]_L$ from contexts in the traditional λ-calculus into λN is defined inductively as

$$\begin{aligned}
\mathbf{LNCt}[x]_L &= \begin{cases} x_i & \text{if } x \in L \\ \mathbf{e}.x & \text{if } x \notin L \end{cases} \\
\mathbf{LNCt}[\lambda x.a]_L &= \lambda x.\mathbf{LNCt}[a]_{x:L} \\
\mathbf{LNCt}[ab]_L &= \mathbf{LNCt}[a]_L(i=\mathbf{LNCt}[b]_L)! \\
\mathbf{LNCt}[-]_{[x_1,\ldots,x_n]} &= \mathbf{h}(\mathbf{e} \ll \{+x_1 = x_{1i}, \ldots, x_n = x_{ni}\})
\end{aligned}$$

The parameter L keeps track of the λ-abstractions crossed during the translation, so the translation starts at the top level with an empty list. On closed λ-terms, $\mathbf{LNCt}[-]_L$ coincides with the translation $\mathbf{LN}[-]$ given in Section 2.2. On open terms, free variables become lookup operations into a global environment carried by the special variable $\mathbf{e}$. Holes perform a local update on the environment, according to the number of λ-abstractions in their surrounding context; this is quite similar to the substitutions associated with holes in [25]. The translation yields quasi-closed λN terms, with $\mathbf{h}$ and $\mathbf{e}$ as only free variables.

Definition 19. The operation of filling a context $C[-]$ with a term a (which might be another context!) is encoded in λN as

$$\mathbf{fill}_L(C[-],a) = (\lambda\mathbf{h}.\mathbf{LNCt}[C[-]]_L)(\lambda\mathbf{e}.\mathbf{LNCt}[a]_L)$$

Theorem 20. *Context filling commutes with the λN encoding, i.e.*

$$\forall E \equiv \{l_1 = a_1, \ldots, l_n = a_n\}.\mathbf{fill}_L(C[-],a)[\mathbf{e} := E] \approx \mathbf{LNCt}[C[a]]_L[\mathbf{e} := E]$$

Proof. Induction on the shape of $C[-]$:

– $C[-] \equiv [-]$:

$$\begin{aligned}
\mathbf{fill}_L(C[-],a) &\equiv (\lambda\mathbf{h}.\mathbf{h}(\mathbf{e} \ll \{+\}))(\lambda\mathbf{e}.\mathbf{LNCt}[a]_L) \\
&\equiv (\lambda\mathbf{h}.\mathbf{h}(\lambda x.\mathbf{e}((\lambda y.y)x))(\lambda\mathbf{e}.\mathbf{LNCt}[a]_L) \\
&\xrightarrow{*}_{\beta N} (\lambda\mathbf{e}.\mathbf{LNCt}[a]_L)(\lambda x.\mathbf{e}x)
\end{aligned}$$

Now by the assumption on the shape of E, $\lambda x.Ex \approx E$ through Theorem 10 (iii)(2), so $((\lambda\mathbf{e}.\mathbf{LNCt}[a]_L)(\lambda x.\mathbf{e}x))[\mathbf{e} := E] \approx ((\lambda\mathbf{e}.\mathbf{LNCt}[a]_L)\mathbf{e})[\mathbf{e} := E] \xrightarrow{*}_{\beta N} \mathbf{LNCt}[a]_L[\mathbf{e}:=E]$

– $C[-] \equiv x$: $\mathbf{LNCt}[x]_L$ does not contain any occurrence of $\mathbf{h}$, so

$$\begin{aligned}
\mathbf{fill}_L(C[-],a) &\equiv (\lambda\mathbf{h}.\mathbf{LNCt}[x]_L)(\lambda\mathbf{e}.\mathbf{LNCt}[a]_L) \\
&\xrightarrow{*}_{\beta N} \mathbf{LNCt}[x]_L \\
&\equiv \mathbf{LNCt}[C[a]]_L
\end{aligned}$$

– $C[-] \equiv \lambda x.A[-]$:

$$\begin{aligned}
\mathbf{fill}_L(C[-],a) &\equiv (\lambda\mathbf{h}x.\mathbf{LNCt}[A[-]]_{x:L})(\lambda\mathbf{e}.\mathbf{LNCt}[a]_L) \\
&\leftrightarrow_{\beta N} \lambda x.(\lambda\mathbf{h}.\mathbf{LNCt}[A[-]]_{x:L})(\lambda\mathbf{e}.\mathbf{LNCt}[a]_{x:L}) \\
&\equiv \lambda x.\mathbf{fill}_{x:L}(A[-],a)
\end{aligned}$$

By induction hypothesis, this is equivalent to

$$\lambda x.\mathbf{LNCt}[A[a]]_{x:L} \equiv \mathbf{LNCt}[\lambda x.A[a]]_L \equiv \mathbf{LNCt}[C[a]]_L$$

– $C[-] \equiv A[-]B[-]$:

$$\begin{aligned}\mathbf{fill}_L(C[-],a) &\equiv (\lambda\mathbf{h}.\mathbf{LNCt}[A[-]]_L(\imath = \mathbf{LNCt}[B[-]]_L)!)\\ &\qquad (\lambda\mathbf{e}.\mathbf{LNCt}[a]_L)\\ &\leftrightarrow_{\beta N} ((\lambda\mathbf{h}.\mathbf{LNCt}[A[-]]_L)(\lambda\mathbf{e}.\mathbf{LNCt}[a]_L))\\ &\qquad (\imath = (\lambda\mathbf{h}.\mathbf{LNCt}[B[-]]_L)(\lambda\mathbf{e}.\mathbf{LNCt}[a]_L))!\\ &\equiv \mathbf{fill}_L(A[-],a)(\imath = \mathbf{fill}_L(B[-],a))!\end{aligned}$$

By induction hypothesis, this is equivalent to

$$\mathbf{LNCt}[A[a]]_L(\imath = \mathbf{LNCt}[B[a]]_L)! \equiv \mathbf{LNCt}[C[a]]_L \qquad \square$$

For the sake of simplicity, the **h** variable was implicitly labelled by $\imath$ in this encoding, following the convention of Section 2.2. As a result, the context-filling operation simultaneously fills all occurrences of the hole. More complex context-filling systems have been studied [19, 15] in which holes are decorated with labels; occurrences of holes are partitioned into classes with common labels, which can be filled independently. A similar behaviour could be obtained here by a simple modification of our encoding scheme, using labelled instances of the **h** variable, and using separate bind operations to fill separate classes of holes.

5.2. *Related calculi*

The idea of embedding contexts, environments, holes or names as first-class constructs in the λ-calculus has motivated several recent proposals, which are summarized in Fig. 4.

The label-selective calculus [13] uses variables and λ-abstractions as in the classical λ-calculus, but assigns a label to each abstraction level. As a result, application

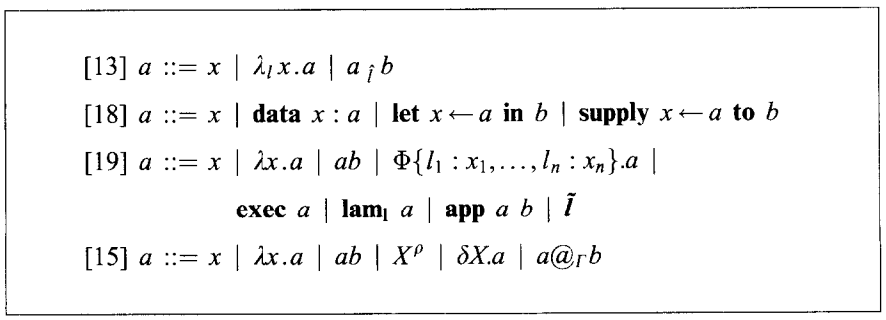

Fig. 4. Related calculi.

constructs are not forced to follow the order of the abstractions: they can directly address an inner abstraction. This simple extension of the λ-calculus supports out-of-order parameter passing, but *not* extensibility: a function still has to be fed by a number of arguments corresponding exactly to the number of its parameters. Hence, the properties of this calculus are clearly different from λN and from other calculi described below. Lee and Friedman managed to simulate the label-selective calculus within their system [19], and it is likely that a similar exercise could be done within λN; however the encoding involves fairly elaborate constructions and therefore does not establish any direct or instructive correspondence between the two models.

The unified system of parameterization (USP) of Lamping [18] is much closer to λN, except that name abstractions occur independently at each name, instead of being related to a common λ. The **data** construct abstracts over a given name, and the **supply** construct passes an argument along a named channel, much like our *bind* construct. However, since data parameters are "transparent", and commute with the other constructs of the language, it is as if all data abstractions were done at a unique global level; hence, for example,

$$\textbf{supply } x \leftarrow 1 \textbf{ to } (\textbf{supply } x \leftarrow 2 \textbf{ to } (\textbf{data } x : x + (\textbf{data } x : x)))$$

yields 4 and not 3 as one would perhaps expect. This is likely to create some difficulties related to substitutions and capture of data parameters when trying to implement the language. Moreover, the fact that there are no multiple abstraction levels and no operation corresponding to our *close* construct implies the absence of subsumption and extensibility. For example, Lamping's encoding of "bounds" $\{id1 \leftarrow exp1, \ldots, idN \leftarrow expN\}$ as

$$\textbf{data } body : \textbf{supply } id1 \leftarrow exp1, \ldots, idN \leftarrow expN \textbf{ to } body$$

is almost like our record extensions of Section 4.2, but, as in our case, these do not obey the width subsumption law.

The λ-calculus with contexts $\lambda\mathbf{C}$ of [19] distinguishes between "source code" (contexts with holes) and "compiled code" (λ-terms), and has internal operators for assembling source code and "compiling" it. The Φ construct abstracts over a set of labels, which are associated to some variables within the body of the abstraction. In order to pass an argument to one of these labelled parameters **l**, one first "captures" this label through an operator $\mathbf{lam}_{\mathbf{l}}$, yielding a usual λ-abstraction, and then uses the **app** construct to apply this abstraction to the given argument. Among the calculi considered here, $\lambda\mathbf{C}$ is the only one which, like λN, has multiple levels of name spaces through hierarchies of abstractions, and has a construct (namely **exec**) for "closing" a name space and passing to the next level. As a matter of fact, its expressive power seems comparable to λN, since a simulation of λN is displayed in [19], while we can go

through the reverse exercise:

$$\begin{aligned}
[[\Phi\{l_1 = x_1, \ldots, l_n = x_n\}.a]] &= \lambda\mathbf{e}.[[a[x_1 := \mathbf{e}.l_1] \ldots [x_n := \mathbf{e}.l_n]]] \\
[[\mathbf{lam}_l\ a]] &= \lambda\mathbf{e}.\lambda x.a(\mathbf{e} \ll \{+x = x_l\}) \\
[[\mathbf{app}\ a\ b]] &= \lambda\mathbf{e}.(a\mathbf{e})(b\mathbf{e}) \\
[[\tilde{\mathbf{I}}]] &= \varepsilon
\end{aligned}$$

However, since $\lambda\mathbf{C}$ so far has no associated theory, it is not obvious to check whether equational properties of the calculi are preserved through translations in both directions. Furthermore, we feel that λN is closer to classical λ-calculus syntax and conventions, and therefore requires less adaptation efforts to inherit known results from the λ-calculus.

Finally, the typed context calculus of [15], announced very recently, uses holes X, hole abstractions $\delta X.a$, and a hole-filling operation $a@_\Gamma b$. This is an explicitly typed calculus, in which types are useful to work out the mechanics of substitutions and hole filling, and to guarantee that no unfilled hole ever gets evaluated. However, the price to pay is some drastic restrictions on term formation: in particular, the hole-filling operation @ is indexed by a type environment, and therefore requires to know statically much information about the context and the term filling it. An additional constraint comes from the fact that holes are only allowed to occur linearly, i.e. exactly once, in a well-typed term. Moreover, the δ construct, which only abstracts one hole at a time, imposes an ordering of hole abstractions, which is an impediment to extensibility. So it seems doubtful that this calculus would provide an appropriate foundation for dynamic binding.

6. Functional programming with dynamic binding

This section displays several linguistic constructs for exploiting dynamic binding in the framework of functional programming languages, on the basis of the λN model. [1] Dynamic binding brings support for incremental assembly of software fragments. This section mostly uses informal examples, but the formal translation $\mathbf{T}[-]$ into λN is fully given in Appendix B.

6.1. Functions, named parameters, and scoping

The language has two kinds of functions: (i) usual λ-abstractions, written like in Haskell, which can be α-converted, like

$$\mathbf{F} = \backslash \mathrm{f\ x\ y} \rightarrow (\mathrm{f\ x}) + (\mathrm{f\ y});;$$

$$\mathbf{Y} = \backslash \mathrm{f} \rightarrow (\backslash \mathrm{x} \rightarrow \mathrm{f\ (x\ x)})(\backslash \mathrm{x} \rightarrow \mathrm{f(x\ x)});;$$

[1] An experimental language *HOP* implementing these constructs as well as the record operations discussed above is available at http://cuiwww.unige.ch/~dami/Hop.

and (ii) abstractions with *named parameters*, written with a set of names enclosed in parenthesis:

$$\mathbf{F2} = \backslash(\mathrm{f\ x\ y}) \rightarrow (\mathrm{f\ x}) + (\mathrm{f\ y});;$$

F2 is translated into $\lambda x.(x_{\mathrm{f}}(\imath = x_{\mathrm{x}})!) + (x_{\mathrm{f}}(\imath = x_{\mathrm{y}})!)$ and therefore has only one abstraction level, while **F** has the obvious translation $\lambda xyz.(x_i(\imath = y_i)!) + (x_i(\imath = z_i)!)$. Named parameters are useful to simultaneously extract several fields from a record. For example, the expression

$$\{\mathrm{x} = 1,\ \mathrm{y} = 2,\ \mathrm{z} = \mathrm{True},\ \mathrm{w} = \text{“foo”},\ \mathrm{f} = \backslash\mathrm{x} \rightarrow \mathrm{x} + 1\}\ \mathrm{F2};;$$

yields 6: values for the named parameters f, x and y are taken from the record, and then are substituted inside the body of the lambda abstraction.

Named parameters and usual parameters may be freely mixed in a single λ-expression. Lexical scoping is treated as in most programming languages: a local declaration takes precedence over a previous declaration using the same name. However, the previous declaration is not irremediably lost. Variables can be preceded by n occurrences of the scope escape operator "^", in order to specify that one should ignore the first n abstraction levels when looking for their corresponding declaration. So in the expression

$$\backslash(\mathrm{a\ b})\ \mathrm{x\ y}\ (\mathrm{x\ y\ z}) \rightarrow \mathrm{x} + \hat{}\mathrm{x} + \hat{}\hat{}\mathrm{x} + \hat{}\hat{}\hat{}\mathrm{x};;$$

the first summand corresponds to the innermost declaration of x, the next two summands correspond to the outermost declaration of x, and the last summand is an error (because after crossing three abstraction levels no declaration of name x can be found). The formal translation of the expression above is

$$\lambda x\, x'\, x''\, x'''.\, x_x''' + x_i' + x_i' + \varepsilon$$

Finally, scope control can also be achieved by explicitly labelling sets of named parameters through a "@" operator:

$$\backslash\mathrm{level1@(a\ b\ x)}\ (\mathrm{x\ y})\ \mathrm{level3@(x\ y\ z)} \rightarrow \mathrm{x} + \hat{}\mathrm{x} + \mathrm{level1@x};;$$

6.2. *Quote and eval*

Quoting is a mechanism for abstracting over all free names of an expression; because of the similarity with LISP, it is written with a quote character:

$$\mathbf{quoted\_expr} = \text{'}(\mathrm{y} + (\backslash\mathrm{x} \rightarrow \mathrm{x} * \mathrm{n})\ \mathrm{z});;$$

This is a closed expression, which abstracts over the free names y, n and z, but *not* over the bound name x. Quoted expressions are encoded as functions parameterized by a record, so the expression above corresponds to

$$\lambda x.x(\lambda y.\, y_y + (\lambda z.z * y_n)y_z)$$

Evaluating quoted expressions in some context is extremely simple: just by application of a record. For example,

quoted_expr { x = 7, y = 9, z = 11, n = 20 };;

yields 229.

The scope escape operator can be used within quoted expressions, so that we can finely tune between the names which should be statically scoped and those which should be "quoted", achieving something similar to the "backquote" operator of LISP. For example, in

\x y → '(x + ^y + z);;

the names x and z are abstracted upon, so the first summand x does *not* refer to the first parameter of the function. By contrast, the name y escapes the quoting operation, and therefore is statically bound to the second parameter of the function.

Record concatenation together with quoting offers interesting possibilities for modelling state operations: the function

\mem → mem ≪ {+x = '(x + y + z) mem };;

takes a "memory" (just a record) as argument, adds the contents of locations x, y and z, and puts the result back in location x. The distinction between what is called "rvalues" and "lvalues" in imperative languages is clearly reflected by the two different uses of name x.

Finally, the encoding of quoted terms allows us to define some operators for reflexive programming, in order to build new quoted terms from quoted terms:

QApp = \a b → \e → (a e) (b e);;

QLambda_x = \$a$ → \$e$ → \$x$ → a (e ≪ {+ x = x });;

QApp takes two quoted terms, and creates a new term which is the application of the first to the second. **QLambda_x** is equivalent to the context $\lambda x.[-]$: it takes a quoted term a, and creates a new term which is a lambda abstraction *capturing free occurrences* of x in a. Both operators are very close to the ones of [19], except that here they are just derived operators, instead of being basic constructs of the language.

6.3. *Variants*

Mathematically, records are labelled products. The dual notion, i.e. labelled coproducts, is called *variant*. The use of coproducts in programming is to support user-defined concrete datatypes. For example, a datatype for lists, which would be written in Haskell:

data List a = Nil | Cons a (List a)

implicitly creates two data constructors **Nil** and **Cons**; these act as two injection functions into the datatype. They can be encoded as follows in the classical λ-calculus:

$$\mathbf{Nil} = \lambda n\, c.\, n$$

$$\mathbf{Cons}\ h\ t = \lambda n\, c.\, c\ h\ t$$

Both constructors take two "deconstructor" functions as arguments, and invoke the appropriate one. This approach is based on the positions of the arguments, so a list can only be built through the two constructors displayed above. Moreover, in order to use a list one has to pass exactly two "deconstructor" functions to it, corresponding to the two possible cases – empty list or non-empty list. In consequence, the **List** datatype cannot be extended in an incremental way into another datatype with more constructors, like for example a **Circular** constructor which would encode circular lists.

By contrast the notion of "variant", i.e. labelled coproduct, makes each data constructor (injection function) independent of the others, and independent of the datatype in which it is used. Following the notation of [7], we write variants with square brackets. In the simple case, these can be just a set of labels, like enumerated types in C or Pascal, so for example we can encode boolean values as [true], [false], or colors as [red], [green], [blue], etc. However, variants can also construct more complex datatypes, like a different version of lists:

```
Nil = [nil];;
Cons = \h t → [cons h t];;
```

The expression within square brackets must start with a name (the "label" of the variant), followed by an applicative sequence. The encoding of variants is again very similar to the standard encoding of coproducts in the λ-calculus: each data constructor takes a collection of "deconstructors" as arguments, and then invokes the appropriate one. The only difference is that the deconstructors are distinguished by names, instead of positions. So, for example, we have

$$[\mathtt{nil}] = \lambda x.x_{\mathsf{nil}}$$

$$[\mathtt{cons}\ h\ t] = \lambda x.x_{\mathsf{cons}}\ h\ t$$

All constructors use only one abstraction level λx, so the encoding can be consistently extended with a new constructor, which would access variable x under a new name.

In order to use a variant, one first has to identify the label with which it was built. Then, depending on that label, one may access its internal data and pursue the computation. Usually this kind of deconstruction of coproducts in functional languages is performed by a **case** construct. Here we do not need an additional syntactic construct: case selection is simply achieved through records. Let us start with a simple example, directly inspired from Section 2.3:

```
Not x = {true = [false], false = [true]} x;;
And x y = {true = x, false = [false]} y;;
```

The **Not** function performs a case selection on its argument x, yielding [false] if x is [true] and [true] if x is [false]. Similarly, the **And** function proceeds by case selection over its y argument. These examples only involves simple variants, i.e. just labels. For more complex variants, we need to be able to access the "internal data". This is done by putting functions in record fields:

Head l = {nil = Error "empty list", cons = \h t → h} l;;

The language actually has some syntactic sugar for this use of records, so we can write, for example,

Head l = {nil = Error "empty list", cons h t = h} l;;

which is a restricted form of pattern matching.

Now consider the definitions of **Not** and **And** above. These are functions which take an argument, and do nothing else than applying something to it. Experienced functional programmers will be immediately tempted to perform a so-called η-conversion, rewriting them as

Not = {true = [false], false = [true]};;

And = \x → {true = x, false = [false]};;

which is legitimate according to Theorem 10. The advantage is that these functions can now be extended through the "≪" operator. Again consider the example of Section 2.3, defining a three-valued logic with an *unknown* value. To encode it, we just need incremental extensions of what we already have:

NotU = **Not** ≪ {+ unknown = [unknown]};;

AndU = \x → (**And** x) ≪ {+ unknown =
{true = [unknown], false = [false], unknown = [unknown]} x};;

The **NotU** and **AndU** functions, designed for the three-value logic, are *fully compatible* with the previous versions for usual logic, i.e. existing code based on the old logic needs no modification. This is exactly the kind of software extensibility offered by object-oriented programming. However, here it was done just with extensible **case** statements, instead of the classes/inheritance machinery. We do not claim that this form of software reuse can totally subsume object-oriented mechanisms, but it can complement it in some cases: for example it seems more natural to handle booleans or lists in this way, rather than defining an abstract class **List** with two concrete subclasses **Nil** and **Cons**.

7. Conclusion

λN is a very simple extension of the classical λ-calculus, which nevertheless has sufficient expressing power to cover various mechanisms involving dynamic binding.

Unlike other proposals with similar ideas [18, 19, 15], it has an inequational theory and a collection of laws to formally reason about program compatibilities. Furthermore, it completely relies on standard techniques for managing substitutions and α-equivalence, and therefore can be implemented easily using de Bruijn indices. An adaptation of standard Curry-style type assignment is straightforward, but requires more complex extensions, as in [10], to get principal types; the difficulties are basically the same as for object-oriented calculi, where the combination of polymorphism, subtyping and recursion is not easily captured by well-known Hindley-Milner inference techniques.

Starting from this work, several interesting research directions are open: one is to explore some extensions of the calculus, like adding a construct for name abstraction, or considering a "default bind" operator of the form $a(*=b)$, binding all remaining arguments of a; another is to follow the Curry-Howard isomorphism and apply the same ideas to logic, probably yielding a system with extensible and reusable proofs.

Finally, there is much room for improvements at the language design level. Basic support for dynamic binding offers a wealth of interesting possibilities for flexible software construction. Some of the examples shown here, like quoting, extensible records or variants, demonstrate the wide range of design directions which can be taken, and give hints on how to exploit dynamic binding in higher-level coordination constructs such as first-class modules or mobile code. Tuning up the language design so as to provide an attractive set of useful constructs in a single high-level environment will require more work and experimentation. However, the fact that we have a very basic underlying formalism, with well-understood equational/inequational properties, and with a type inference algorithm, proves to be an unvaluable tool for exploring the design space. For example, we did not realize until working out the λN semantics of **case** statements that this construct was actually not necessary, and could be subsumed by records. Similarly, the current design of record operations, with a clear separation between "records" and "record extensions", and with the possibility to use records in functional position, could never have been invented without seeing the λN translation.

Appendix A. *HOP* Syntax

Term syntax is displayed in Fig. 5. Precise precedence and associativity rules are not displayed here, but standard conventions apply (i.e. "*" has higher precedence than "+", functional applications are left-associative, etc.). Usual conventions are also used for building integers, strings and "names" (identifiers). For programming convenience, *HOP* does not use disjoint lexical sets for the sets $\mathcal{V}$ ("variables") and $\mathcal{N}$ ("names") of the λN-calculus; disambiguation is done according to context. The implicit rules for disambiguation can be overridden by scope escape or explicit scoping operators.

⟨ *term* ⟩	=	⟨ *integer* ⟩	
	|	⟨ *string* ⟩	
	|	⟨ *var* ⟩	
	|	'\ ' ⟨ *plist* ⟩+ '->' ⟨ *term* ⟩	*lambda-abstraction*
	|	⟨ *term* ⟩ ⟨ *term* ⟩	*application*
	|	⟨ *term* ⟩ '(' ⟨ *name* ⟩ '=' ⟨ *term* ⟩ ')'	*bind construct*
	|	⟨ *term* ⟩ '!'	*close construct*
	|	'{' ⟨ *field* ⟩+ '}'	*record*
	|	'{+' ⟨ *field* ⟩+ '}'	*record extension*
	|	⟨ *term* ⟩ '<<' ⟨ *term* ⟩	*record concat.*
	|	⟨ *term* ⟩ '.' ⟨ *name* ⟩	*field selection*
	|	''(' ⟨ *term* ⟩ ')'	*quoted term*
	|	'[' ⟨ *name* ⟩ ⟨ *term* ⟩* ']'	*variant*
	|	'-' ⟨ *term* ⟩	*prefix minus*
	|	⟨ *term* ⟩ ⟨ *binop* ⟩ ⟨ *term* ⟩	
	|	'(' ⟨ *term* ⟩ ')'	
⟨ *var* ⟩	=	'^'* ⟨ *name* ⟩	*optional scope escape*
	|	⟨ *name* ⟩ '@' ⟨ *name* ⟩	*explicit scoping*
⟨ *plist* ⟩	=	⟨ *name* ⟩	*"usual" param.*
	|	[⟨ *name* ⟩ '@'] '(' ⟨ *name* ⟩+ ')'	*named param.*
	|	[⟨ *name* ⟩ '@'] '(*)'	*implicit param.*
⟨ *field* ⟩	=	⟨ *name* ⟩ ⟨ *plist* ⟩* '=' ⟨ *term* ⟩	
⟨ *binop* ⟩	=	'+' | '-' | '*' | '/'	*integer arithmetic*
	|	'<' | '>' | '<=' | '>=' | '==' | '/='	*integer comparison*
	|	'++'	*string concatenation*

Fig. 5. Syntax.

Appendix B. Translation from *HOP* to λN

The translation function $\mathbf{T}[-]_-$ displayed in Fig. 6 is subscripted by a stack D of *declarations*, recording the names declared at each lambda abstraction. A declaration d may be of shape

- (x, l), saying that name l corresponds to variable x_l,
- $(x, \{l_1 \ldots l_n\})$, saying that name l_i corresponds to variable x_{l_i}, or
- $(x, *)$, saying that any name l corresponds to variable x_l.

Pushing the declaration d on top of stack D is written $d : D$. Translation of top-level expressions starts with an empty stack. In all cases where an abstraction $\lambda x. \ldots$ is generated, it is implicitly understood that x is a new variable.

$$
\begin{array}{|r|}
\hline
\mathbf{T}[\backslash l \text{ -> } a]_D = \lambda x.\ \mathbf{T}[\ a]_{(x,l):D} \\
\mathbf{T}[\backslash (l_1 \ldots l_n\ ) \text{ -> } a]_D = \lambda x.\ \mathbf{T}[\ a]_{(x,\{l_1 \ldots l_n\}):D} \\
\mathbf{T}[\backslash (*) \text{ -> } a]_D = \lambda x.\ \mathbf{T}[\ a]_{(x,*):D} \\
\mathbf{T}[\backslash l \ @(\ \ l_1 \ldots l_n\ ) \text{ -> } a]_D = \lambda l.\ \mathbf{T}[\ a]_{(l,\{l_1 \ldots l_n\}):D} \\
\mathbf{T}[\backslash l \ @(*) \text{ -> } a]_D = \lambda l.\ \mathbf{T}[\ a]_{(l,*):D} \\
\hline
\mathbf{T}[\hat{}v]_{[]} = \varepsilon \\
\mathbf{T}[\hat{}v]_{d:D} = \mathbf{T}[v]_D \\
\mathbf{T}[l]_{(x,l'):D} = x_i \text{ if } l \equiv l',\ \mathbf{T}[l]_D \text{ otherwise} \\
\mathbf{T}[l]_{(x,\{l_1 \ldots l_n\}):D} = x_l \text{ if } \exists l_i.\ l \equiv l_i,\ \mathbf{T}[l]_D \text{ otherwise} \\
\mathbf{T}[l]_{(x,*):D} = x_l \\
\mathbf{T}[l\ @\ l']_{[]} = \varepsilon \\
\mathbf{T}[l\ @\ l']_{(l'',\{l_1 \ldots l_n\}):D} = \begin{cases} l_{l'} & \text{if } l \equiv l'' \wedge \exists l_i.\ l' \equiv l_i \\ \varepsilon & \text{if } l \equiv l'' \wedge \neg(\exists l_i.\ l' \equiv l_i) \\ \mathbf{T}[l\ @\ l']_D & \text{otherwise} \end{cases} \\
\mathbf{T}[l\ @\ l']_{(l'',l'''):D} = \mathbf{T}[\ l\ @\ l']_D \\
\mathbf{T}[l\ @\ l']_{(l'',*):D} = l_{l'} \text{ if } l \equiv l'',\ \mathbf{T}[\ l\ @\ l']_D \text{ otherwise} \\
\hline
\mathbf{T}[\ \{\ f_1 \ldots f_n\ \}\ ]_D = \lambda x.x\ \mathbf{Tf}[\ f_1]_D \ldots\ \mathbf{Tf}[\ f_n]_D! \\
\mathbf{T}[\ \{+\ f_1 \ldots f_n\ \}\ ]_D = \lambda x.x\ \mathbf{Tf}[\ f_1]_D \ldots\ \mathbf{Tf}[\ f_n]_D \\
\mathbf{T}[\ a \ll b]_D = \lambda x.\mathrm{T}[a]_D(\mathrm{T}[b]_D\, x) \\
\mathbf{T}[\ a\ .\ l]_D = \mathbf{T}[\ a]_D(\lambda x.x_l) \\
\mathbf{T}[\ '(\ a\ )\ ]_D = \lambda x.\, x(\lambda y.\mathrm{T}[a]_{(y,*):D}) \\
\hline
\mathbf{Tf}\,[l p_1 \ldots p_n = a]_D = (l = \mathbf{T}[\backslash p_1 \ldots p_n \text{ -> } a]_D) \\
\hline
\mathbf{T}[\ ab]_D = \mathbf{T}[\ a]_D(\imath = \mathrm{T}[b]_D)! \\
\mathbf{T}[\ [l\, a_1 \ldots a_n\ ]\ ]_D = \lambda x.x_l \mathrm{T}[a_1]_D \ldots \mathrm{T}[a_n]_D \\
\hline
\end{array}
$$

Fig. 6. Translation.

References

[1] S. Abramsky, C.-H. L. Ong, Full abstraction in the lazy lambda calculus, Inform. Comput. 105 (1993) 159–267.

[2] J.-M. Andreoli, C. Hankin, D. Le Métayer (Eds.), Coordination Programming: Mechanisms, Models, Semantics, Imperial College Press, London, 1996.

[3] M. Banville, Sonia: an adaptation of linda for coordination of activities in organizations, in: [9], pp. 57–74.

[4] H. Barendregt, The Lambda-Calculus, its Syntax and Semantics, Studies in Logic and the Foundations of Mathematics, North-Holland, Amsterdam, 1984.

[5] V. Breazu-Tannen, T. Coquand, C.A. Gunter, A. Scedrov, Inheritance as implicit coercion, Inform. Comput. 93 (1991) 172–221; also in: [14], pp. 197–245.

[6] K. Bruce, G. Longo, A modest model of records, inheritance, and bounded quantification, Inform. Comput. 87 (1990) 196–240; also in: [14], pp. 151–195.

[7] L. Cardelli, A semantics of multiple inheritance, Inform. Comput. 76 (1988) 138–164.

[8] L. Cardelli, J. Mitchell, Operations on records, Mathematical Structures in Computer Science, Cambridge Univ. Press, Cambridge, 1991, pp. 3–48; also in: [14], pp. 295–350.

[9] P. Ciancarini, C. Hankin (Eds.), Proc. Coordination Languages and Models '96, Lecture Notes in Computer Science, vol. 1061, Springer, Berlin, 1996.

[10] L. Dami, Type inference and subtyping for higher-order generative communication, in: [2], pp. 98–138.

[11] R. Harper, B. Pierce, A record calculus based on symmetric concatenation, in: Proc. 18th ACM Symp. on Principles of Programming Languages, ACM Press, New York, 1991, pp. 131–142.
[12] G. Florijn, T. Besamusca, D. Greefhorst, Ariadne and HOPLa: flexible coordination of collaborative processes, in: [9], pp. 197–214.
[13] J. Garrigue, H. Aït-Kaci, The typed polymorphic label-selective lambda calculus, in: Proc. 21st ACM Symp. on Principles of Programming Languages, ACM Press, New York, 1994, pp. 35–47.
[14] C.A. Gunter, J.C. Mitchell (Eds.), Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design, Foundation of Computing Series, MIT Press, Cambridge, MA, 1994.
[15] M. Hashimoto, A. Ohori, A typed context calculus, preprint RIMS-1098, Research Institute for Mathematical Sciences, Kyoto University, 1996, available at http://www.kurims.kyoto-u.ac.jp/ohori/.
[16] P. Hudak et al., The Haskell Report and Haskell Tutorial, ACM SIGPLAN Notices 27 (5) (1992).
[17] S. Jagannathan, Dynamic modules in higher-order languages, IEEE Internat. Conf. on Computer Languages, Toulouse, France, 1994, pp. 74–87.
[18] J. Lamping, A unified system of parameterization for programming languages, in: Proc. ACM Conf. on LISP and Func. Prog., ACM Press, New York, 1988, pp. 316–326.
[19] S.-D. Lee, D.P. Friedman, Enriching the lambda calculus with contexts: towards a theory of incremental program construction, Proc. ACM Internat. Conf. on Functional Programming, ACM SIGPLAN Notices 31 (6) (1996) 239–250.
[20] R. Milner, M. Tofte, R. Harper, The Definition of Standard ML, MIT Press, New York, 1991.
[21] C. Queinnec, D. de Roure, Sharing code through first-class environments, Proc. ACM Internat. Conf. on Func. Prog.; ACM SIGPLAN Notices 31 (6) (1996) 251–261.
[22] M. Radestock, S. Eisenbach, Semantics of a higher-order coordination language, in: [9], pp. 339–356.
[23] D. Rémy, Typechecking records and variants in a natural extension of ML, in: Proc. 16th ACM Symp. on Principles of Programming Languages, ACM Press, New York, 1989, pp. 242–249; also in: [14], pp. 67–96.
[24] D. Rémy, Typing record concatenation for free, in: Proc. 19th ACM Symp. on Principles of Programming Languages, ACM Press, New York, 1992, pp. 166–176; also in: [14], pp. 351–372.
[25] C. Talcott, A theory of binding structures and applications to rewriting, Theoret. Comput. Sci. 112 (1993) 99–143.
[26] M. Takahashi, Parallel reductions in λ-calculus, Inform. Comput. 118 (1) (1995) 120–127.
[27] M. Wand, Type inference for record concatenation and multiple inheritance, Inform. Comput. 93 (1) (1991) 1–15.