The λ-context calculus

Murdoch J. Gabbay Stéphane Lengrand

Abstract

We present a simple but expressive lambda-calculus whose syntax is populated by variables which behave like meta-variables. It can express both capture-avoiding and capturing substitution (instantiation). To do this requires several innovations, including a key insight in the confluence proof and a set of reduction rules which manages the complexity of a calculus of contexts over the 'vanilla' lambda-calculus in a very simple and modular way. This calculus remains extremely close in look and feel to a standard lambda-calculus with explicit substitutions, and good properties of the lambda-calculus are preserved.

Keywords: Lambda-calculus, contexts, meta-variables, capture-avoiding substitution, capturing substitution, instantiation, confluence, nominal techniques, calculus of explicit substitutions.

1 Introduction

This is a paper about a λ-calculus for contexts. A **context** is a term with a 'hole'. The canonical example is probably $C[\text{-}] = \lambda x.\text{-}$ in the λ-calculus. This is not λ-calculus syntax because it has a hole -, but if we fill that hole with a term t then we obtain something, we usually write it $C[t]$, which *is* a λ-calculus term.

For example if $C[\text{-}] = \lambda x.\text{-}$ then $C[x] = \lambda x.x$ and $C[y] = \lambda x.y$. This cannot be modelled by a combination of λ-abstraction and application, because β-reduction avoids capture. Formally: there is no λ-term f such that $ft = C[t]$. The term $\lambda z.\lambda x.z$ is the obvious candidate, but $(\lambda z.\lambda x.z)x =_\alpha \lambda x'.x$. (Here $=_\alpha$ is α-equality.)

Contexts arise often in proofs of meta-properties in functional programming. They have been substantially investigated in papers by Pitts on contextual equivalence between terms in λ-calculi (with global state) [18,20]. This work was about proving programs equivalent in all contexts — **contextual equivalence**. The idea is that two programs, represented by possibly-open λ-terms, are equivalent when one can be exchanged for another in code (without changing whichever notion of observation we prefer to use).

This suggests that we should call holes *context variables* X (say they have 'level 2') distinct from 'normal' variables x (say they have 'level 1') and allow λ-abstraction over them to obtain a λ-calculus of *contexts*, so that we can study program contexts with the full panoply of vocabulary, and hopefully with many of the theorems, of the λ-calculus. For example $\lambda x.\text{-}$ may be represented by $\lambda X.\lambda x.X$. Substitution for X does not avoid capture with respect to 'ordinary' λ-abstraction, so $(\lambda X.\lambda x.X)x$

reduces to $\lambda x.x$.

The Lambda Context Calculus internalises context variables (as variables of 'level 2', which we write X, Y, Z). X, Y, and Z are now variables which can occur any number of times anywhere in a term — *and* they can be λ-abstracted. The Lambda Context Calculus therefore goes further and internalises another level of contexts (variables of 'level 3', which we write $\mathcal{W}, \mathcal{W}'$) — and so on. There are several possibilities where such a calculus might be applied.

Consider formalising mathematics in a logical framework based on Higher-Order Logic (**HOL**) [28]. Typically we have a goal and some assumptions and we want a derivation of one from the other. This derivation may be represented by a λ-term (the *Curry-Howard correspondence*). But the derivation is arrived at by stages in which it is *in*complete.

To the right are two derivations of $A{\Rightarrow}B{\Rightarrow}C, A{\Rightarrow}B \vdash A{\Rightarrow}C$. The bottom one is complete, the top one is incomplete.[1] An issue arises because the right-most $[A]^i$ in the bottom derivation is discharged, which means that we have to be able to instantiate ? in a sub-derivation for an assumption which will be discharged. Discharge corresponds in the Curry-Howard correspondence precisely to λ-abstraction, and this instantiation corresponds to capturing substitution. Similar issues arise with existential variables [10, Section 2, Example 3].

$$\dfrac{\dfrac{\dfrac{A{\Rightarrow}B{\Rightarrow}C \quad [A]^i}{B{\Rightarrow}C} \qquad \dfrac{?}{B}}{C}\;{}^{i}}{A{\Rightarrow}C}$$

$$\dfrac{\dfrac{\dfrac{A{\Rightarrow}B{\Rightarrow}C \quad [A]^i}{B{\Rightarrow}C} \qquad \dfrac{A{\Rightarrow}B \quad [A]^i}{B}}{C}\;{}^{i}}{A{\Rightarrow}C}$$

The central issue for any calculus of contexts is the interaction of context variables with α-equivalence. Let x, y, z be 'ordinary' variables and let X be a context variable. If $\lambda x.X =_\alpha \lambda y.X$ then $(\lambda X.\lambda x.X)x =_\alpha (\lambda X.\lambda y.X)x \rightsquigarrow \lambda y.x$, giving non-confluent reductions. Dropping α-equivalence entirely is too drastic; we need $\lambda y.\lambda x.y$ to be α-convertible with $\lambda z.\lambda x.z$ to reduce a term like $(\lambda y.\lambda x.y)x$.

Solutions include clever control of substitution and evaluation order [23], types to prevent 'bad' α-conversions [21,11,22], explicit labels on meta-variables [10,13], and more [4, Section 2]. More on this in the Conclusions.

We took our technical ideas for handling α-equivalence, not from the literature on context calculi cited above, but from *nominal unification* [27]. This was designed to manage α-equivalence in the presence of holes, in unification — 'unification of contexts of syntax', in other words. Crudely put, we obtained the λ-context calculus (**LCC**) by allowing λ-abstraction over the holes and adding β-reduction.

This work has similar goals to previous work by the first author [6] which presented a calculus called NEWcc. The LCC possesses a significatnly more elementary set of reduction rules; notably, we dispense entirely with the freshness contexts and freshness logic of the NEWcc. Indeed, the LCC has only one single non-obvious side-condition, it is on $(\sigma\mathbf{p})$ in Figure 5.

The result is a system with a powerful hierarchy of context variables and which still manages to be clean and, we hope, easy to use and to study.

In Section 2 we present the syntax and reductions of the LCC. The look-and-feel is of a λ-calculus with explicit substitutions, except that each variable has a 'level'

[1] This example 'borrowed' from [10].

which determines how 'strongly' binders by that variable resist capture. We give example reductions and discuss the technical issues which motivated our design. In Section 3 we discuss the λ-free part of the language, prove strong normalisation, and give an algorithm for calculating normal forms. In the usual λ-calculus this normal form is calculated in big-step style and written $s[a{\mapsto}t]$; as is standard for a calculus of explicit substitutions, here this part of evaluation is dissected in detail. In Section 4 we treat confluence, first of the λ-free part of the language, then of the full reduction system. The proof may look elementary but it is not, and we give enough technical detail to show how all the side-conditions interact to ensure confluence. It is not sufficient to give a λ-calculus without *binding*, but the hierarchy of levels means that λ itself is no longer necessarily a binder. We address that issue with a new И in Section 5. We conclude with brief discussions of programming in Section 6, and then discuss related and future work.

2 Syntax and reductions

2.1 Syntax

We suppose a countably infinite set of disjoint infinite **sets of variables** $\mathbb{A}_1$, $\mathbb{A}_2$, ... i, j, k range over levels; we usually maintain a convention that $k \leq i < j$, where we break it we clearly say so. We always use a **permutative convention** that $a_i, b_j, c_k, \ldots$ range *permutatively* over variables of level i; so a_i, b_j, and c_k are always distinct variables. There is no particular connection between a_1 and a_2; we have just given them similar names.

Definition 2.1 *LCC syntax is given by* $s, t ::= a_i \mid tt \mid \lambda a_i.t \mid t[a_i{\mapsto}t]$.

Application associates to the left, e.g. $tt't''$ is $(tt')t''$. We say that a_i **has level** i. We call b_j **stronger** than a_i, and a_i **weaker** than b_j, when $j > i$. If $i = j$ we say that b_j and a_i have the same strength. We call $s[a_i{\mapsto}t]$ an **explicit substitution** (of level i). We call $\lambda a_i.t$ an **abstraction** (of level i).

By convention $x, y, z, X, Y, Z, \mathcal{W}$ are distinct variables; x, y, z have level 1, X, Y, Z have level 2, and $\mathcal{W}$ has level 3. Note that levels are $1, 2, 3, \ldots$ but our proofs would work as well for levels being integers, reals, or any totally ordered set.

The stronger a variable, the more 'meta' its behaviour. The intuition of $\lambda x.X$ is of the context $\lambda x.$- where - is a hole; this is because, as we shall see, substitution for the relatively strong X does not avoid capture by the relatively weak λx. Strong variables can be abstracted as usual; the intuition of $\lambda X.X$ is of the 'normal' identity function; the intuition of $\lambda X.\lambda x.X$ is of the mapping 't maps to $\lambda x.t$'.

Our syntax has no constant symbols though we shall be lax and use them where convenient, for example $1, 2, 3, \ldots$. This can be accommodated by extending syntax, or by declaring them to be variables of a new level $0 < 1$ which we do not abstract over or substitute for.

Definition 2.2 *Define the* **level** $\mathsf{level}(s)$ *and the* **free variables** $\mathsf{fv}(s)$ *by the rules in Figure 1.*

Here $\mathsf{max}(i, j)$ is the greater of i and j, and $\mathsf{max}(i, j, k)$ is the greatest of i, j, and k. Later we shall write '$\mathsf{level}(s_1, \ldots, s_n) \leq i$' as shorthand for '$\mathsf{level}(s_1) \leq i$ and ...

$$\begin{array}{rlrl}
\mathsf{level}(a_i) & = i & \mathsf{fv}(a_i) & = \{a_i\} \\
\mathsf{level}(ss') & = \mathsf{max}(\mathsf{level}(s), \mathsf{level}(s')) & \mathsf{fv}(\lambda a_i.s) & = \mathsf{fv}(s) \backslash \{a_i\} \\
\mathsf{level}(\lambda a_i.s) & = \mathsf{max}(i, \mathsf{level}(s)) & \mathsf{fv}(s[a_i \mapsto t]) & = (\mathsf{fv}(s) \backslash \{a_i\}) \cup \mathsf{fv}(t) \\
\mathsf{level}(s[a_i \mapsto t]) & = \mathsf{max}(i, \mathsf{level}(s), \mathsf{level}(t)) & \mathsf{fv}(st) & = \mathsf{fv}(s) \cup \mathsf{fv}(t)
\end{array}$$

Fig. 1. Levels $\mathsf{level}(s)$ and free variables $\mathsf{fv}(s)$

$$\frac{}{a_i R a_i} \qquad \frac{sRs' \ tRt'}{st\ R\ s't'} \qquad \frac{sRs' \quad tRt'}{s[a_i \mapsto s']\ R\ t[a_i \mapsto t']} \qquad \frac{sRs'}{\lambda a_i.s\ R\ \lambda a_i.s'} \qquad \frac{sRs'}{s'Rs} \qquad \frac{sRs' \ s'Rs''}{sRs''}$$

Fig. 2. Rules for a congruence

$$\begin{array}{rll}
(a_i\ b_i)a_i = b_i & & \\
(a_i\ b_i)b_i = a_i & & \\
(a_i\ b_i)c = c & & (c \text{ any atom other than } a_i \text{ or } b_i) \\
(a_i\ b_i)(ss') = ((a_i\ b_i)s)((a_i\ b_i)s') & & \\
(a_i\ b_i)(\lambda c.s) = \lambda(a_i\ b_i)c.(a_i\ b_i)s & & (c \text{ any atom}) \\
(a_i\ b_i)(s[c \mapsto t]) = ((a_i\ b_i)s)[(a_i\ b_i)c \mapsto (a_i\ b_i)t] & & (c \text{ any atom})
\end{array}$$

Fig. 3. Rules for swapping

$$\begin{array}{ll}
\lambda a_i.s =_\alpha \lambda b_i.(b_i\ a_i)s & \text{if } \ b_i \# \mathsf{fv}(s) \\
s[a_i \mapsto t] =_\alpha ((b_i\ a_i)s)[b_i \mapsto t] & \text{if } \ b_i \# \mathsf{fv}(s)
\end{array}$$

Fig. 4. Rules for α-equivalence

and $\mathsf{level}(s_n) \leq i$', similarly for '$\mathsf{level}(s_1, \ldots, s_n) < i$'.

Lemma 2.3 *If* $\mathsf{level}(s) = 1$ *then* $\mathsf{fv}(s)$ *coincides with the usual notion of 'free variables of' for the* λ*-calculus, if we read* $s[a_1 \mapsto t]$ *as* $(\lambda a_1.s)t$*.*

We shall see that the operational behaviour of such terms is the same as well.

A **congruence** is a binary relation $s\ R\ s'$ satisfying the conditions of Figure 2. Define an **(atoms) swapping** $(a_i\ b_i)s$ by the rules in Figure 3. Swapping is characteristic of the underlying 'nominal' method we use in this paper [9,27]. We let swapping $(a_i\ b_i)$ act pointwise on sets of variables S: $(a_i\ b_i)S = \{(a_i\ b_i)c \mid c \in S\}$. Here c ranges over all elements of S, including a_i and b_i (if they are in S).

Lemma 2.4 $\mathsf{fv}((a_i\ b_i)s) = (a_i\ b_i)\mathsf{fv}(s)$ *and* $\mathsf{level}((a_i\ b_i)s) = \mathsf{level}(s)$*.*

If S is a set of variables write $a_i \# S$ when $a_i \notin S$ and also there exists no variable $b_j \in S$ such that $j > i$.

Definition 2.5 *Call the two rules in Figure 4* α**-conversion** *of* a_i*. Let* α**-equivalence** $=_\alpha$ *be the least congruence relation containing* α*-conversion.*

Note that: a_i may be α-converted in $\lambda a_i.s$ if $\mathsf{level}(s) \leq i$, so $\lambda x.x =_\alpha \lambda y.y$. a_i may be α-converted in $s[a_i {\mapsto} t]$ if $\mathsf{level}(s) \leq i$, so $x[x {\mapsto} X] =_\alpha y[y {\mapsto} X]$. We cannot α-convert a_i in s if $b_j \in \mathsf{fv}(s)$ for $j > i$. For example $\lambda x.X \neq_\alpha \lambda y.X$. This is consistent with a reading of strong variables as unknown terms with respect to weaker variables. We cannot α-convert variables to variables of other levels.

Lemma 2.6 *If s mentions only variables of level 1, then α-equivalence collapses to the usual α-equivalence on untyped λ-terms (plus an explicit substitution).*

Theorem 2.7 *If $s =_\alpha s'$ then $\mathsf{fv}(s) = \mathsf{fv}(s')$ and $\mathsf{level}(s) = \mathsf{level}(s')$.*

Proofs of all results above are by easy inductions.

In the rest of this paper we find it convenient to work on terms up to α-equivalence ($=_\alpha$-equivalence classes of terms). When later we write '$s = t$', the intended reading is that the *α-equivalence classes* of s and t are equal.

2.2 Reductions

Definition 2.8 *Define* **the reduction relation** *by the rules in Figure 5.*

Recall our permutative convention; for example in $(\sigma\lambda')$ a_i and c_i are distinct. Subsection 2.3 shows examples of these rules at work, and Subsection 2.4 discusses their design. We shall use the following notation:

- We write $\rightsquigarrow^*$ for the transitive reflexive closure of $\rightsquigarrow$.
- We write $s \not\rightsquigarrow$ when there exists no t such that $s \rightsquigarrow t$. If $s \not\rightsquigarrow$ we call s a **normal form**, as is standard.
- We write $s \overset{(\textbf{ruleset})}{\rightsquigarrow} t$ when we can deduce $s \rightsquigarrow t$ using only rules in $(\textbf{ruleset})$ and the rules $(\textbf{Rapp})$ to $(\mathbf{R}\sigma')$, where $(\textbf{ruleset}) \subseteq \{(\beta), (\sigma\mathbf{a}), (\sigma\mathsf{fv}), (\sigma\mathbf{p}), (\sigma\sigma), (\sigma\lambda), (\sigma\lambda')\}$. (Later in Section 5 we extend reduction with rules for a binder И.)
- Call $\rightsquigarrow$ **terminating** when there is no infinite sequence $t_1 \rightsquigarrow \cdots \rightsquigarrow t_i \rightsquigarrow \cdots$ Similarly for $\overset{(\textbf{ruleset})}{\rightsquigarrow}$. Call $\rightsquigarrow$ **confluent** when if $s \rightsquigarrow^* t$ and $s \rightsquigarrow^* t'$ then there exists some u such that $t \rightsquigarrow^* u$ and $t' \rightsquigarrow^* u$. Similarly for $\overset{(\textbf{ruleset})}{\rightsquigarrow}$.

This is all standard [25,1].

We note two easy but important technical properties: reductions does not increase the level of a term or its set of free variables.

Lemma 2.9 *If $s \rightsquigarrow s'$ then $\mathsf{level}(s') \leq \mathsf{level}(s)$.*

Lemma 2.10 *If $s \rightsquigarrow s'$ then $\mathsf{fv}(s') \subseteq \mathsf{fv}(s)$, and if $s \rightsquigarrow^* s'$ then $\mathsf{fv}(s') \subseteq \mathsf{fv}(s)$.*

2.3 Example reductions

The LCC is a λ-calculus with explicit substitutions [15]. The general form of the σ-rules is familiar from the literature though the conditions, especially those involving levels, are not; we discuss them in Subsection 2.4 below. First, we consider some example reductions. Recall our convention that we write x, y, z for variables of level 1, and X, Y, Z for variables of level 2.

- (β) is standard for a calculus with explicit substitutions.

$$\begin{array}{lll}
(\beta) & (\lambda a_i.s)t \rightsquigarrow s[a_i{\mapsto}t] & \\
(\sigma\mathbf{a}) & a_i[a_i{\mapsto}t] \rightsquigarrow t & \\
(\sigma\mathsf{fv}) & s[a_i{\mapsto}t] \rightsquigarrow s & a_i\#\mathsf{fv}(s) \\
(\sigma\mathbf{p}) & (ss')[a_i{\mapsto}t] \rightsquigarrow (s[a_i{\mapsto}t])(s'[a_i{\mapsto}t]) & \mathsf{level}(s,s',t) \le i \\
(\sigma\sigma) & s[a_i{\mapsto}t][b_j{\mapsto}u] \rightsquigarrow s[b_j{\mapsto}u][a_i{\mapsto}t[b_j{\mapsto}u]] & i<j \\
(\sigma\lambda) & (\lambda a_i.s)[b_j{\mapsto}u] \rightsquigarrow \lambda a_i.(s[b_j{\mapsto}u]) & i<j \\
(\sigma\lambda') & (\lambda a_i.s)[c_i{\mapsto}u] \rightsquigarrow \lambda a_i.(s[c_i{\mapsto}u]) & a_i\#\mathsf{fv}(u)
\end{array}$$

$$\frac{s \rightsquigarrow s'}{st \rightsquigarrow s't}\,(\mathbf{Rapp}) \qquad \frac{t \rightsquigarrow t'}{st \rightsquigarrow st'}\,(\mathbf{Rapp}') \qquad \frac{s \rightsquigarrow s'}{\lambda a_i.s \rightsquigarrow \lambda a_i.s'}\,(\mathbf{R}\lambda)$$

$$\frac{s \rightsquigarrow s'}{s[a_i{\mapsto}t] \rightsquigarrow s'[a_i{\mapsto}t]}\,(\mathbf{R}\sigma) \qquad \frac{t \rightsquigarrow t'}{s[a_i{\mapsto}t] \rightsquigarrow s[a_i{\mapsto}t']}\,(\mathbf{R}\sigma')$$

Fig. 5. Reduction rules of the LCC

- The behaviour of a substitution on a variable depends on strengths:

  $$x[X{\mapsto}t] \overset{(\sigma\mathsf{fv})}{\rightsquigarrow} x \qquad x[x'{\mapsto}t] \overset{(\sigma\mathsf{fv})}{\rightsquigarrow} x \qquad x[x{\mapsto}t] \overset{(\sigma\mathbf{a})}{\rightsquigarrow} t \qquad X[x{\mapsto}t] \not\rightsquigarrow$$

  The term $X[x{\mapsto}t]$ will not reduce until a suitable strong substitution $[X{\mapsto}t]$ arrives from the surrounding context, if any.

- Substitutions for relatively strong variables may distribute using $(\sigma\sigma)$ or $(\sigma\lambda)$ under substitutions or λ-abstractions for relatively weaker variables:

  $$X[x{\mapsto}t][X{\mapsto}x] \overset{(\sigma\sigma)}{\rightsquigarrow} X[X{\mapsto}x][x{\mapsto}t[X{\mapsto}x]] \overset{(\sigma\mathbf{a})}{\rightsquigarrow} x[x{\mapsto}t[X{\mapsto}x]] \overset{(\sigma\mathbf{a})}{\rightsquigarrow} t[X{\mapsto}x]$$
  $$(\lambda x.X)[X{\mapsto}x] \rightsquigarrow \lambda x.(X[X{\mapsto}x]) \rightsquigarrow \lambda x.x$$

  This makes strong variables behave like 'holes'. Instantiation of holes is compatible with β-reduction; here is a typical example:

  $$\begin{array}{ll}
  ((\lambda x.X)t)[X{\mapsto}x] \overset{(\sigma\mathbf{p})}{\rightsquigarrow} (\lambda x.X)[X{\mapsto}x](t[X{\mapsto}x]) & ((\lambda x.X)t)[X{\mapsto}x] \overset{(\beta)}{\rightsquigarrow} X[x{\mapsto}t][X{\mapsto}x] \\
  \qquad\overset{(\sigma\lambda)}{\rightsquigarrow} (\lambda x.(X[X{\mapsto}x]))(t[X{\mapsto}x]) & \qquad\overset{(\sigma\sigma)}{\rightsquigarrow} X[X{\mapsto}x][x{\mapsto}t[X{\mapsto}x]] \\
  \qquad\overset{(\sigma\mathbf{a})}{\rightsquigarrow} (\lambda x.x)(t[X{\mapsto}x]) & \qquad\overset{(\sigma\mathbf{a})}{\rightsquigarrow} x[x{\mapsto}t[X{\mapsto}x]] \overset{(\sigma\mathbf{a})}{\rightsquigarrow} t[X{\mapsto}x] \\
  \qquad\overset{(\beta)}{\rightsquigarrow} x[x{\mapsto}t[X{\mapsto}x]] \overset{(\sigma\mathbf{a})}{\rightsquigarrow} t[X{\mapsto}x] &
  \end{array}$$

- There is no restriction in $s[a_i{\mapsto}t]$ that $\mathsf{level}(t) < i$; for example the terms $X[x{\mapsto}Y]$ and $X[x{\mapsto}\mathcal{W}]$ are legal.

- $[a_i{\mapsto}t]$ is not a term, but the term $\lambda b_j.b_j[a_i{\mapsto}t]$ where $j > i$ and $j > \mathsf{level}(t)$ will achieve the effect of 'the substitution $[a_i{\mapsto}t]$ as a term':

  $$(\lambda b_j.b_j[a_i{\mapsto}t])s \overset{(\beta)}{\rightsquigarrow} b_j[a_i{\mapsto}t][b_j{\mapsto}s] \overset{(\sigma\sigma)}{\rightsquigarrow} b_j[b_j{\mapsto}s][a_i{\mapsto}t[b_j{\mapsto}s]] \overset{(\sigma\mathsf{fv})}{\rightsquigarrow} b_j[b_j{\mapsto}s][a_i{\mapsto}t] \overset{(\sigma\mathbf{a})}{\rightsquigarrow} s[a_i{\mapsto}t].$$

2.4 Comments on the side-conditions

- $(\sigma\mathsf{fv})$ is a form of garbage-collection. We do not want to garbage-collect $[x{\mapsto}2]$ in $X[x{\mapsto}2]$ because $(\sigma\sigma)$ could turn X into something with x free — for example x

itself; this is why the side-condition is not $a_i \notin \mathsf{fv}(s)$ but $a_i \# \mathsf{fv}(s)$.

It is unusual for a garbage collection rule to appear in a calculus of explicit substitutions; we might hope to 'push substitutions into a term until they reach variables' and so make do with a rule of the form $c_k[a_i \mapsto t] \rightsquigarrow c_k$ (for $k \leq i$). In the LCC this will not do because side-conditions (such as that of $(\sigma\mathbf{p})$) can stop a substitution going deep into a term. Without $(\sigma\mathsf{fv})$ we lose confluence (see the second case of Theorem 4.10). A version of $(\sigma\mathsf{fv})$ appears in the literature as 'garbage collection' [3].

- Recall that the level of a term is the level of the strongest variable it contains, free *or* bound. The side-condition $\mathsf{level}(s, s', t) \leq i$ in $(\sigma\mathbf{p})$ seems to be fundamental for confluence to work; we have not been able to sensibly weaken it, even if we also change other rules to fix what goes wrong when we do. Here is what happens if we drop the side-condition entirely:

$$\begin{array}{rcl} X[x \mapsto y][y \mapsto x] \overset{(\beta)}{\leftsquigarrow} ((\lambda x.X)y)[y \mapsto x] & \overset{(\sigma\mathbf{pFALSE})}{\rightsquigarrow} & ((\lambda x.X)[y \mapsto x])(y[y \mapsto x]) \\ & \overset{(\sigma\mathbf{a})}{\rightsquigarrow} & ((\lambda x.X)[y \mapsto x])x \end{array}$$

- The side-conditions on $(\sigma\sigma)$, $(\sigma\lambda)$, and $(\sigma\lambda')$ implement that a strong substitution can capture. There is no $(\sigma\sigma')$ since that would destroy termination of the part of the LCC without λ — and we have managed to get confluence without it.
- There is no rule permitting a weak substitution to propagate under a stronger abstraction, *even if* we avoid capture:

$$(\sigma\lambda'\mathbf{FALSE}) \qquad (\lambda a_i.s)[c_k \mapsto u] \rightsquigarrow \lambda a_i.(s[c_k \mapsto u]) \qquad a_i \# \mathsf{fv}(u),\ k \leq i$$

Such a rule causes the following problem for confluence:

$$\begin{array}{rcl} (\lambda Y.(xZ))[x \mapsto 3][Z \mapsto \mathcal{W}] & \overset{(\sigma\lambda'\mathbf{FALSE})}{\rightsquigarrow} & (\lambda Y.(xZ)[x \mapsto 3])[Z \mapsto \mathcal{W}] \\ (\lambda Y.(xZ))[x \mapsto 3][Z \mapsto \mathcal{W}] & \overset{(\sigma\sigma)}{\rightsquigarrow} & (\lambda Y.(xZ))[Z \mapsto \mathcal{W}][x \mapsto 3[Z \mapsto \mathcal{W}]] \\ & \overset{(\sigma\mathsf{fv})}{\rightsquigarrow} & (\lambda Y.(xZ))[Z \mapsto \mathcal{W}][x \mapsto 3] \end{array}$$

As is the case for the side-condition of $(\sigma\mathbf{p})$, any stronger form of $(\sigma\lambda')$ than what we admit in the LCC seems to provoke a cascade of changes which make the calculus more complex.

Investigation of these side-conditions is linked to strengthening the theory of freshness and α-equivalence, and possibly to developing a good semantic theory to guide us. This is future work and some details are mentioned in the Conclusions.

3 The substitution action

Define $(\mathbf{sigma}) = \{(\sigma\mathbf{a}), (\sigma\mathsf{fv}), (\sigma\mathbf{p}), (\sigma\sigma), (\sigma\lambda), (\sigma\lambda')\}$ (so $(\mathbf{sigma})$ is 'everything except for (β)'). It would be good if this is is terminating [3,15]. Do we sacrifice this property because of the hierarchy of variables? No. To prove it we translate

$$\begin{array}{rlr}
s[a_i{:=}t] = s & a_i\#\mathsf{fv}(s),\ \text{and } \textit{otherwise} \\
a_i[a_i{:=}t] = t & \\
(ss')[a_i{:=}t] = (s[a_i{:=}t])(s'[a_i{:=}t]) & \mathsf{level}(s,s',t) \leq i \\
s[c_k{\mapsto}u][a_i{:=}t] = s[a_i{:=}t][c_k{:=}u[a_i{:=}t]] & k < i \\
(\lambda c_k.s)[a_i{:=}t] = \lambda c_k.(s[a_i{:=}t]) & k < i \\
(\lambda c_i.s)[a_i{:=}t] = \lambda c_i.(s[a_i{:=}t]) & c_i\#\mathsf{fv}(t) \\
s[a_i{:=}t] = s[a_i{\mapsto}t] &
\end{array}
\qquad
\begin{array}{r}
a_i^* = a_i \\
(\lambda a_i.s)^* = \lambda a_i.(s^*) \\
(s[a_i{\mapsto}t])^* = s^*[a_i{:=}t^*] \\
(st)^* = (s^*)(t^*)
\end{array}$$

Fig. 6. Substitution $s[a_i{:=}t]$ and (**sigma**)-normal form s^*

LCC syntax to first-order terms (terms without binding [1,25]) in the signature

$$\Sigma = \{\star, \mathsf{Abs}, \mathsf{App}\} \cup \{\mathsf{Sub}^i \mid i\}$$

as follows:

$$\overline{x} = \star \quad \overline{\lambda a_i.s} = \mathsf{Abs}(\overline{s}) \quad \overline{s\ t} = \mathsf{App}(\overline{s}, \overline{t}) \quad \overline{s[a_i{\mapsto}t]} = \mathsf{Sub}^i(\overline{s}, \overline{t})$$

Here $\star$ has arity 0, Abs has arity 1, App has arity 2, and Sub^i has arity 2 for all i (i ranges over levels). Give symbols precedence (lowest precedence on the right)

$$\ldots, \mathsf{Sub}^j, \ldots, \mathsf{Sub}^i, \ldots, \mathsf{App}, \mathsf{Abs}, \star \qquad\qquad (j > i).$$

Define the **lexicographic path ordering** [14,1] by:

$$\frac{}{t_i \ll f(t_1,\ldots,t_n)} \qquad\qquad \frac{s \ll t_i}{s \ll f(t_1,\ldots,t_n)}$$

$$\frac{(t'_1,\ldots,t'_n) \ll (t_1,\ldots,t_n)}{f(t'_1,\ldots,t'_n) \ll f(t_1,\ldots,t_n)} \qquad \frac{u_i \ll f(t_1,\ldots,t_n) \text{ for } 1 \leq i \leq m}{g(u_1,\ldots,u_m) \ll f(t_1,\ldots,t_n)}$$

Here g and f are first-order symbols, g has strictly lower precedence than f, and $t_1,\ldots,t_n,t'_1,\ldots,t'_n,u_1,\ldots,u_m,s$ are first-order terms. It is a fact [14,1] that $\ll$ is a well-founded order on first-order terms satisfying the *subterm property*, i.e. if s is a subterm of t then $s \ll t$.

Theorem 3.1 *If* $t \overset{\text{(sigma)}}{\rightsquigarrow} u$ *then* $\overline{t} \gg \overline{u}$. *Thus* (**sigma**)*-reduction terminates.*

The proof is by checking that a (**sigma**)-reduction strictly reduces the lexicographic path order of the associated first-order term; this is not hard.

Let x have level 1. $(\lambda x.xx)(\lambda x.xx)$ has an infinite series of reductions in the LCC. It follows that — even with a hierarchy of variables — (β) strictly adds power to the LCC.

Call s (**sigma**)**-normal** when $s \overset{\text{(sigma)}}{\not\rightsquigarrow}$. What does a (**sigma**)-normal form look like? Define a **substitution action** $s[a_i{:=}t]$ and using it define s^*, by the rules in Figure 6. Rules are listed in order of precedence so that a later rule is only used if no earlier rule is applicable. We apply the rule $(\lambda c_i.s)[a_i{:=}t]$ renaming where possible to ensure $c_i\#\mathsf{fv}(t)$.

Lemma 3.2 $s[a_i{\mapsto}t] \overset{\text{(sigma)}}{\rightsquigarrow^*} s[a_i{:=}t]$.

Proof. Each clause in the definition of $s[a_i{:=}t]$ is simulated by a (**sigma**)-rule. □

Theorem 3.3 $s \overset{(\mathbf{sigma})}{\rightsquigarrow^*} s^*$ *and* s^* *is a* (**sigma**)*-normal form.*

Proof. The first part is by an easy induction on the definition of s^*; the case of $(s[a_i{\mapsto}t])^*$ uses Lemma 3.2. The second part is by a routine induction on s. □

4 Confluence

Let (**beta**) be the set $\{(\beta), (\sigma\lambda), (\sigma\lambda'), (\sigma\mathsf{fv})\}$. (**sigma**) $\cap$ (**beta**) is non-empty; we discuss why at the end of Subsection 4.3.

Theorem 4.1 $\rightsquigarrow$ *is confluent.*

The proof of Theorem 4.1 occupies this section. Two standard proof-methods are: *(1)* Use a *parallel reduction relation* $\Rightarrow$, and *(2)* for all s define a $s^{\downarrow}$ such that $s \rightsquigarrow^* s^{\downarrow}$ and if $s \rightsquigarrow s'$ then $s' \rightsquigarrow^* s^{\downarrow}$. Both methods are standard [25]. Which to use for the LCC? It seems that λ 'wants' method 1 — but σ 'wants' method 2. Confluence is (relatively) easy to prove if we split the reduction relation into (**sigma**) and (**beta**) and apply different methods to each — and then join them together.

4.1 Confluence of (**sigma**)

Note there is no capture-avoidance condition in Lemma 4.2, because $i < j$. The full proofs also contain another version where $i = j$ and $a_i \# \mathsf{fv}(u)$.

Lemma 4.2 *If* $i < j$ *then* $s[a_i{:=}t][b_j{:=}u] = s[b_j{:=}u][a_i{:=}t[b_j{:=}u]]$.

Proof. By induction on i, then on s. We illustrate the induction with two cases.

- Suppose $i < j < k$. Note that usually we take $k \leq i$; this is an exception. Then:

  $$\begin{aligned} c_k[a_i{:=}t][b_j{:=}u] &= c_k[a_i{\mapsto}t][b_j{:=}u] \\ &= c_k[b_j{:=}u][a_i{\mapsto}t[b_j{:=}u]] \\ &= c_k[b_j{\mapsto}u][a_i{\mapsto}t[b_j{:=}u]] \end{aligned} \qquad \begin{aligned} c_k[b_j{:=}u][a_i{:=}t[b_j{:=}u]] &= c_k[b_j{\mapsto}u][a_i{:=}t[b_j{:=}u]] \\ &= c_k[b_j{\mapsto}u][a_i{\mapsto}t[b_j{:=}u]] \end{aligned}$$

- Suppose that $\mathsf{level}(s, s', t) < j$. By Lemma 3.2 we have $(ss')[a_i{\mapsto}t] \rightsquigarrow^* (ss')[a_i{:=}t]$. By Lemma 2.9 we have $\mathsf{level}((ss')[a_i{:=}t]) \leq \mathsf{level}((ss')[a_i{\mapsto}t]) = \mathsf{level}(s, s', t) < j$. Then by our assumptions on levels,

  $$(ss')[a_i{:=}t][b_j{:=}u] = (ss')[a_i{:=}t] = (ss')[b_j{:=}u][a_i{:=}t[b_j{:=}u]].$$

□

Lemma 4.3 (i) $(a_i[a_i{\mapsto}t])^* = t^*$.

(ii) $(c_k[a_i{\mapsto}t])^* = c_k$ *where* $k \leq i$.

(iii) $((ss')[a_i{\mapsto}t])^* = ((s[a_i{\mapsto}t])(s'[a_i{\mapsto}t]))^*$ *where* $\mathsf{level}(s, s', t) \leq i$.

(iv) $(s[a_i{\mapsto}t][b_j{\mapsto}u])^* = (s[b_j{\mapsto}u][a_i{\mapsto}t[b_j{\mapsto}u]])^*$ *if* $i < j$.

(v) $((\lambda a_i.s)[b_j{\mapsto}u])^* = (\lambda a_i.(s[b_j{\mapsto}u]))^*$ *if* $i < j$.

(vi) $((\lambda a_i.s)[c_i{\mapsto}u])^* = (\lambda a_i.(s[c_i{\mapsto}u]))^*$ *if (renaming where possible)* $a_i \# \mathsf{fv}(u)$.

$$\frac{}{a_i \Longrightarrow a_i}\,(\mathbf{Pa}) \qquad \frac{s \Longrightarrow s' \qquad t \Longrightarrow t'}{s[a_i{\mapsto}t] \Longrightarrow s'[a_i{\mapsto}t']}\,(\mathbf{P}\sigma) \qquad \frac{s \Longrightarrow s' \quad t \Longrightarrow t'}{st \Longrightarrow s't'}\,(\mathbf{Papp}) \qquad \frac{s \Longrightarrow s'}{\lambda a_i.s \Longrightarrow \lambda a_i.s'}\,(\mathbf{P}\lambda)$$

$$\frac{s \Longrightarrow s' \quad t \Longrightarrow t' \quad s'[a_i{\mapsto}t'] \overset{R_\epsilon}{\rightsquigarrow} u}{s[a_i{\mapsto}t] \Longrightarrow u}\,(\mathbf{P}\sigma\epsilon) \qquad \frac{s \Longrightarrow s' \quad t \Longrightarrow t' \quad s't' \overset{R_\epsilon}{\rightsquigarrow} u}{st \Longrightarrow u}\,(\mathbf{Papp}\epsilon) \qquad (R \in (\mathbf{beta}))$$

Fig. 7. Parallel reduction relation for the LCC

Proof. Most cases are easy; we consider only the fourth one. Recall that we assume $i < j$. Using Lemma 4.2

$$(s[a_i{\mapsto}t][b_j{\mapsto}u])^* = s^*[a_i{:=}t^*][b_j{:=}u^*] = s^*[b_j{:=}u^*][a_i{:=}t^*[b_j{:=}u^*]] \\ = (s[b_j{\mapsto}u][a_i{\mapsto}t[b_j{\mapsto}u]])^*.$$

□

Lemma 4.4 *If* $s \overset{(\mathbf{sigma})}{\rightsquigarrow} s'$ *then* $s' \overset{(\mathbf{sigma})}{\rightsquigarrow^*} s^*$.

Proof. By induction on the derivation of $s \overset{(\mathbf{sigma})}{\rightsquigarrow} s'$, using Lemma 4.3. □

Theorem 4.5 $\overset{(\mathbf{sigma})}{\rightsquigarrow}$ *is confluent.*

Proof. By an easy inductive argument using Lemma 4.4. □

4.2 (**beta**)*-reduction*

Define the **parallel reduction relation** $\Longrightarrow$ by the rules in Figure 7.

In rules $(\mathbf{P}\sigma\epsilon)$ and $(\mathbf{Papp}\epsilon)$, $s't' \overset{R_\epsilon}{\rightsquigarrow} u$ and $s'[a_i{\mapsto}t'] \overset{R_\epsilon}{\rightsquigarrow} u$ indicate a rewrite with $R \in (\mathbf{beta})$ derivable without using $(\mathbf{Rapp})$, $(\mathbf{Rapp}')$, $(\mathbf{R}\lambda)$, $(\mathbf{R}\sigma)$, or $(\mathbf{R}\sigma')$.

Lemma 4.6 $s \Longrightarrow^* s'$ *if and only if* $s \overset{(\mathbf{beta})}{\rightsquigarrow^*} s'$.

Corollary 4.7 *If* $s \Longrightarrow s'$ *then* $\mathsf{fv}(s') \subseteq \mathsf{fv}(s)$ *and* $\mathsf{level}(s') \leq \mathsf{level}(s)$.

Proof. From Lemma 4.6 and Lemma 2.10. □

Lemma 4.8 $\Longrightarrow$ *satisfies the diamond property. That is, if* $s' \Longleftarrow s \Longrightarrow s''$ *then there is some* s''' *such that* $s' \Longrightarrow s''' \Longleftarrow s''$.

Proof. We work by induction on the depth of the derivation of $s \Longrightarrow s'$ proving $\forall s''.\ s \Longrightarrow s'' \Rightarrow \exists s'''.\ (s' \Longrightarrow s''' \wedge s'' \Longrightarrow s''')$. We consider possible pairs of rules which could derive $s \Longrightarrow s_1$ and $s \Longrightarrow s_2$. All cases are very easy, we only sketch that of $(\mathbf{P}\sigma)$ and $(\mathbf{P}\sigma\epsilon)$ for $(\sigma\lambda')$, which is the least trivial.

Suppose $s \Longrightarrow s'$ and $u \Longrightarrow u'$ and also $s \Longrightarrow s''$ and $u \Longrightarrow u''$. Suppose also that (renaming where necessary) $a_i\#u''$ so that by $(\mathbf{P}\sigma)$ and $(\mathbf{P}\sigma\epsilon)$ for $(\sigma\lambda')$

$$(\lambda a_i.s')[c_i{\mapsto}u'] \Longleftarrow (\lambda a_i.s)[c_i{\mapsto}u] \Longrightarrow \lambda a_i.(s''[c_i{\mapsto}u'']).$$

By inductive hypothesis there are s''' and u''' such that $s' \Longrightarrow s''' \Longleftarrow s''$ and $u' \Longrightarrow u''' \Longleftarrow u''$. By Corollary 4.7 $a_i\#u'''$. Using $(\mathbf{P}\sigma\epsilon)$ for $(\sigma\lambda')$ and $(\mathbf{P}\sigma)$

$$(\lambda a_i.s')[c_i{\mapsto}u'] \Longrightarrow \lambda a_i.(s'''[c_i{\mapsto}u''']) \Longleftarrow \lambda a_i.(s''[c_i{\mapsto}u'']).$$

□

Theorem 4.9 $\overset{(\mathbf{beta})}{\rightsquigarrow}$ *is confluent.*

Proof. By Lemmas 4.6 and 4.8 and a standard argument [2]. □

4.3 Combining (sigma) *and* (beta)

Theorem 4.10 *If* $s \Longrightarrow s'$ *and* $s \overset{(\mathbf{sigma})}{\rightsquigarrow} s''$ *then there is some* s''' *such that* $s' \overset{(\mathbf{sigma})}{\rightsquigarrow^*} s'''$ *and* $s'' \Longrightarrow s'''$*.*

Proof. We work by induction on the derivation of $s \Longrightarrow s'$. For brevity we merely indicate the non-trivial parts. We always assume that $s \Longrightarrow s'$, $t \Longrightarrow t'$, and $u \Longrightarrow u'$, where appropriate.

- (β) has a divergence with $(\sigma\mathbf{p})$ in the case that $i < j$ and $\mathsf{level}(s, t, u) \leq j$. This can be closed using a $\Longrightarrow$-rewrite which uses $(\sigma\lambda)$:

$$(\lambda a_i.s)[b_j{\mapsto}u](t[b_j{\mapsto}u]) \overset{(\sigma\mathbf{p})}{\leftsquigarrow} ((\lambda a_i.s)t)[b_j{\mapsto}u] \Longrightarrow s'[a_i{\mapsto}t'][b_j{\mapsto}u']$$

$$(\lambda a_i.s)[b_j{\mapsto}u](t[b_j{\mapsto}u]) \Longrightarrow s'[b_j{\mapsto}u'][a_i{\mapsto}t'[b_j{\mapsto}u']] \overset{(\sigma\sigma)}{\leftsquigarrow} s'[a_i{\mapsto}t'][b_j{\mapsto}u']$$

- $(\sigma\sigma)$ has a divergence with $(\sigma\lambda')$. Suppose $i{<}j$ and (renaming if necessary) $c_i\#\mathsf{fv}(t)$:

$$(\lambda c_i.s)[b_j{\mapsto}u][a_i{\mapsto}t[b_j{\mapsto}u]] \overset{(\sigma\sigma)}{\leftsquigarrow} (\lambda c_i.s)[a_i{\mapsto}t][b_j{\mapsto}u] \Longrightarrow (\lambda c_i.(s'[a_i{\mapsto}t']))[b_j{\mapsto}u']$$

  We know $b_j\#\mathsf{fv}(t)$ because $c_i\#\mathsf{fv}(t)$ and $i < j$. We deduce $b_j\#\mathsf{fv}(t')$ using Corollary 4.7. This justifies the $\Longrightarrow$-rewrite below, which uses $(\sigma\mathsf{fv})$:

$$\begin{aligned} \lambda c_i.(s'[a_i{\mapsto}t'])[b_j{\mapsto}u'] &\overset{(\sigma\lambda')}{\rightsquigarrow} \lambda c_i.(s'[a_i{\mapsto}t'][b_j{\mapsto}u']) \\ &\overset{(\sigma\sigma)}{\rightsquigarrow} \lambda c_i.(s'[b_j{\mapsto}u'][a_i{\mapsto}t'[b_j{\mapsto}u']]) \\ &\overset{(\sigma\mathsf{fv})}{\rightsquigarrow} \lambda c_i.(s'[b_j{\mapsto}u'][a_i{\mapsto}t']) \Longleftarrow (\lambda c_i.s)[b_j{\mapsto}u][a_i{\mapsto}t[b_j{\mapsto}u]] \end{aligned}$$

□

$(\sigma\lambda)$ is in $(\mathbf{sigma}) \cap (\mathbf{beta})$ to make the case of $(\sigma\mathbf{p})$ with (β) work. $(\sigma\lambda')$ is in $(\mathbf{sigma}) \cap (\mathbf{beta})$ to make a similar divergence of $(\sigma\mathbf{p})$ with (β) work. $(\sigma\mathsf{fv})$ is in $(\mathbf{sigma}) \cap (\mathbf{beta})$ to make the case of $(\sigma\sigma)$ with $(\sigma\lambda')$ work.

Theorem 4.1 now follows by an easy diagrammatic argument using Theorem 4.10, Theorem 4.5, and Lemma 4.8.

5 A NEW part for the LCC

x is not α-convertible in $\lambda x.X$. Suppose we really do want to bind x; we can do so with И. We extend syntax: $s, t ::= \ldots \mid \text{И}a_i.t$. We extend the notions of level, fv, congruence, and swapping with cases for И which are identical to those for λ (except that we write И instead). For example $\mathsf{fv}(\text{И}a_i.s) = \mathsf{fv}(s) \setminus \{a_i\}$.

The difference is in the α-equivalence: $\text{И}a_i.s =_\alpha \text{И}b_i.(b_i\ a_i)s$ if $b_i \not\in \mathsf{fv}(s)$.

Note the $b_i \notin \mathsf{fv}(s)$ instead of $b_i \# \mathsf{fv}(s)$ as in the clause for λ. This lets variables bound by И α-convert regardless of whether stronger variables are present. For example $\lambda x.X \neq_\alpha \lambda y.X$ but $\text{И}x.\lambda x.X =_\alpha \text{И}y.\lambda y.X$. We add reduction rules:

$$\begin{array}{lll}
(\text{И}\mathbf{p}) & (\text{И}a_i.s)t \rightsquigarrow \text{И}a_i.(st) & a_i \notin \mathsf{fv}(t) \\
(\text{И}\sigma) & (\text{И}c_k.s)[a_i {\mapsto} t] \rightsquigarrow \text{И}c_k.(s[a_i {\mapsto} t]) & k \leq i,\ c_k \notin \mathsf{fv}(t) \\
(\text{И} \notin) & \text{И}a_i.s \rightsquigarrow s & a_i \notin \mathsf{fv}(s)
\end{array}
\qquad
\frac{s \rightsquigarrow s'}{\text{И}a_i.s \rightsquigarrow \text{И}a_i.s'}\,(\mathbf{R}\text{И})$$

x is not bound in $\lambda y.s$ if s mentions a strong variable, for example in $\lambda y.(Xy)$ substitution for X can capture y. We may want y to be *really* local and avoid capture by substitutions for X. We can increase the level of y; $\lambda Y.(XY)$ will do in this case. This has a hidden cost because side-conditions (especially on $(\sigma\mathbf{p})$) look at strengths of variables, so having strong variables can block reductions in the context. И avoids this, for example $\text{И}y.\lambda y.(Xy)$ has the behaviour we need:

$$(\text{И}y.\lambda y.(Xy))[X {\mapsto} y] \overset{(\text{И}\sigma)}{\rightsquigarrow} \text{И}y'.((\lambda y'.(Xy'))[X {\mapsto} y]) \overset{(\text{И}\lambda)}{\rightsquigarrow} \text{И}y'.\lambda y'.(Xy')[X {\mapsto} y]) \rightsquigarrow^* \text{И}y'.\lambda y'.yy'$$

И is reminiscent of π-calculus restriction [16]. $(\text{И}\mathbf{p})$ and $(\text{И}\sigma)$ are reminiscent of scope-extrusion. $(\text{И} \notin)$ is reminiscent of 'garbage-collection'.

We do not admit a rule '$s(\text{И}a.t) \rightsquigarrow \text{И}a.(st)$ if $a \notin \mathsf{fv}(s)$':

$$\text{И}y.\text{И}y'.(yy') \;{}^*\!\!\leftsquigarrow (\text{И}y.y)\text{И}y'.y' \;{}^*\!\!\leftsquigarrow (\lambda x.xx)\text{И}y.y \rightsquigarrow \text{И}y.(\lambda x.xx)y \rightsquigarrow^* \text{И}y.yy$$

For similar reasons we do not admit a rule '$s[b {\mapsto} \text{И}a.t] \rightsquigarrow \text{И}a.(s[b {\mapsto} t])$ if $a \notin \mathsf{fv}(s)$'.

Why the side-conditions on $(\text{И}\sigma)$? $c_k \notin \mathsf{fv}(t)$ comes from the intuition of И as defining a scope. We need $k \leq i$ for confluence:

$$\text{И}X.(X[x {\mapsto} 2]) \leftsquigarrow (\text{И}X.X)[x {\mapsto} 2] \overset{(\sigma\mathsf{fv})}{\rightsquigarrow} \text{И}X.X$$

The proof of termination of **(sigma)** extends smoothly if we add the rules for И to **(sigma)** (to make a set **(sigmanew)**). The proof of confluence for the system as a whole also extends smoothly. We see some examples of the use of И in a moment.

6 Programming in the calculus

Call t **single-leveled** of level i when all variables in it (free or bound) have level i. Then it is easy to prove that notions of free variable and substitution coincide with the 'traditional' definitions and we have:

Theorem 6.1 *For any i the single-leveled terms of level i, with their reductions, form an isomorphic calculus to λx with garbage collection [3].*

As a corollary, the trivial mapping from the untyped λ-calculus to single-leveled terms of level 1 (say), preserves normal forms and strong normalisation.

We can exploit the hierarchy to do some nice things. Here is one example: $R = X[x {\mapsto} 2][y {\mapsto} 3]$ can be viewed as a record with 'handle' X and with 2 stored at x and 3 at y. Then $\lambda\mathcal{W}.(\mathcal{W}[X {\mapsto} x])$ applied to R looks up the data stored at x, and $\lambda\mathcal{W}.(\mathcal{W}[X {\mapsto} X[x {\mapsto} 3]])$ updates it. In fact these terms do a little more than this, because their effect is the same when applied to a term in which a record with ‘handle’ X is buried deep in the term, perhaps as part of a β-redex or substitution. $\lambda\mathcal{W}.(\mathcal{W}[X\mapsto(\mathcal{W}[X\mapsto x])+1])$ increments the value stored at x.

Here is an example reduction:

$$\begin{array}{lcl}
(\lambda\mathcal{W}.\mathcal{W}[X\mapsto X[x\mapsto 3]])\ R & \overset{(\beta)}{\leadsto} & \mathcal{W}[X\mapsto X[x\mapsto 3]][\mathcal{W}\mapsto R] \\
 & \overset{(\sigma\sigma),(\sigma\mathbf{a}),(\sigma\mathsf{fv})}{\leadsto^*} & R[X\mapsto X[x\mapsto 3]] = X[x\mapsto 2][y\mapsto 3][X\mapsto X[x\mapsto 3]] \\
 & \overset{(\sigma\sigma)}{\leadsto^*} & X[X\mapsto X[x\mapsto 3]][x\mapsto 2[X\mapsto X[x\mapsto 3]]][y\mapsto 3[X\mapsto X[x\mapsto 3]]] \\
 & \overset{(\sigma\mathbf{a}),(\sigma\mathbf{b})}{\leadsto^*} & X[x\mapsto 3][x\mapsto 2][y\mapsto 3].
\end{array}$$

There is some garbage here, but a later look-up on x returns 3, not 2:

$$(\lambda\mathcal{W}.\mathcal{W}[X\mapsto x])(X[x\mapsto 3][x\mapsto 2][y\mapsto 3]) \leadsto^* x[x\mapsto 3][x\mapsto 2][y\mapsto 3] \leadsto^* 3$$

We can use И to assign fresh storage. The following program, if applied to a value and R, extends R with a fresh location and returns the new record together with a lookup function for the new location:

$$\lambda Z.\text{И}x.\lambda Y.(Y[x\mapsto Z], \lambda\mathcal{W}.\mathcal{W}[X\mapsto x]).$$

Here we use a pairing constructor $(\text{-},\text{-})$ just for convenience.

Note that we access data in R by applying a substitution for X; in this sense the ‘handle’ X in R is externally visible. We can hide it by λ-abstracting X to obtain $\lambda X.(X[x\mapsto 2][y\mapsto 3])$. Then lookup at x becomes $\lambda\mathcal{W}.(\mathcal{W}x)$ and update becomes $\lambda\mathcal{W}.\lambda X.(\mathcal{W}[X\mapsto X[x\mapsto 1]])$.

We can parameterise over the data stored in the record: $\lambda X'.(X'[x\mapsto X][y\mapsto Y])$. Furthermore a term of the form $\lambda X.(X[x\mapsto X][y\mapsto X])$ can capture a form of self-reference within the record. Finally, $\lambda X.(X[x\mapsto\mathcal{W}][y\mapsto\mathcal{W}'])$ makes no committment about the data stored.

7 Related work, conclusions, and future work

The LCC of this paper is simpler than the NEWcc [6]. Compare the side-condition of $(\sigma\mathbf{a})$ (there is none) with that of $(\sigma\mathbf{a})$ from [6]. The notion of freshness is simpler and intuitive; we no longer require a logic of freshness, or the ‘freshness context with sufficient freshnesses’, see most of page 4 in [6]. A key innovation in attaining this simplicity is our use of conditions involving $\mathsf{level}(s)$ the level of s, which includes information about the levels of free *and* bound variables.

But there is a price: this calculus has fewer reductions. Notably $(\sigma\lambda')$ will not reduce $(\lambda a_i.s)[c_k\mapsto u]$ where $k < i$; a rule $(\sigma\lambda')$ in [6] does. That stronger version seems to be a major source of complexity.

Still, the LCC is part of something larger yet to be constructed. Other papers on nominal techniques contain elements of the developments we have in mind when we imagine such a system. So for example:

In this paper we cannot α-convert x in $\lambda x.X$. Nominal terms can: swappings are in the syntax (here swappings are in the meta-level) and also freshness contexts

[27]. A problem is that we do not yet understand the theory of swappings for strong variables; the underlying Fraenkel-Mostowski sets model [9] only has (in the terminology of this paper) one level of variable. A semantic model of the hierarchy of variables would be useful and this is current work.

In this paper we cannot deduce $x\#\mathsf{fv}(\lambda x.X)$ even though for every instance this does hold (for example $x\#\lambda x.x$ and $x\#\lambda x.y$). Hierarchical nominal rewriting [7] has a more powerful notion of freshness which can prove the equivalent of $x\#\mathsf{fv}(\lambda x.X)$. Note that hierarchical nominal rewriting does not have the conditions on *levels* which we use to good effect in this paper.

We cannot reduce $(\lambda x.y)[y{\mapsto}Y]$ because there is no z such that $z\#Y$. We can allow programs to dynamically generate fresh variables in the style of FreshML [19] or the style of a sequent calculus for Nominal Logic by Cheney [5].

We cannot reduce $X[x{\mapsto}2][y{\mapsto}3]$ to $X[y{\mapsto}3][x{\mapsto}2]$. Other work [8] gives an equational system which can do this, and more.

There is no denotational semantics for the LCC. This is current work.

More related work (not using nominal techniques). The calculi of contexts λm and $\lambda\mathcal{M}$ [23] also have a hierarchy of variables. They use carefully-crafted scoping conventions to manage problems with α-conversion. Other work [21,11,22] uses a type system; connections with this work are unclear. λc of Bognar's thesis contains [4, Section 2] an extensive literature survey on the topic of context calculi.

A separation of abstraction λ and binding $\mathsf{И}$ appears in one other work we know of [24], where they are called q and ν. In this vein there is [12], which manages scope explicitly in a completely different way, just for the fun. Finally, the reduction rules of $\mathsf{И}$ look remarkably similar to π-calculus restriction [16], and it is probably quite accurate to think of $\mathsf{И}$ as a 'restriction in the λ-calculus'.

Ours is a calculus with explicit substitutions. See [15] for a survey. Our treatment of substitution is simple-minded but still quite subtle because of interactions with the rest of the language. We note that the translation of possibly open terms of the untyped λ-calculus into the LCC preserves strong normalisation. One reduction rule, $(\sigma\mathsf{fv})$, is a little unusual amongst such calculi, though it appears as 'garbage collection' of λx [3].

The look and feel of the LCC is squarely that of a λ-calculus with explicit substitutions. All the real cleverness has been isolated in the side-condition of $(\sigma\mathbf{p})$; other side-conditions are obvious given an intuition that strong variables can cause capturing substitution (in the NEWcc [6] complexity spilled over into other rules and into a logic for freshness). $\mathsf{И}$ is only necessary when variables of different strengths occur, and the hierarchy of variables only plays a rôle to trigger side-conditions.

Further work. Desirable and nontrivial meta-properties of the λ-calculus survive in the LCC including confluence, and preservation of strong normalisation for a natural encoding of the untyped λ-calculus into the LCC. It is possible, in principle at least, to envisage an extension of ML or Haskell [17,26] with meta-variables based on the LCC's notion of strong and weak variables.

We can go in the direction of logic, treating equality instead of reduction and imitating higher-order logic, which is based on the simply-typed λ-terms enriched with constants such as $\forall : (o \to o) \to o$ and $\Rightarrow : o \to o \to o$ where o is a type of truth-values [29], along with suitable equalities and/or derivation rules. There should be no problem with imposing a simple type system on LCC and writing down a 'context higher-order logic'. This takes the LCC in the direction of calculi of contexts for incomplete proofs [13,10]. The non-trivial work (in no particular order) is to investigate cut-elimination, develop a suitable theory of models/denotations, and possibly to apply it to model incomplete proofs.

An implementation is current work.

The LCC is simple, clear, and it has good properties. It seems to hit a technical sweet spot: every extension of it which we have considered, provokes significant non-local changes. Often in computer science the trick is to find a useful balance between simplicity and expressivity. Perhaps the LCC does that.

References

[1] Franz Baader and Tobias Nipkow, *Term rewriting and all that*, Cambridge University Press, 1998.

[2] H. P. Barendregt, *The lambda calculus: its syntax and semantics (revised ed.)*, North-Holland, 1984.

[3] Roel Bloo and Kristoffer Høgsbro Rose, *Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection*, CSN-95: Computer Science in the Netherlands, 1995.

[4] Mirna Bognar, *Contexts in lambda calculus*, Ph.D. thesis, Vrije Universiteit Amsterdam, 2002.

[5] James Cheney, *A simpler proof theory for nominal logic*, FOSSACS, Springer, 2005, pp. 379–394.

[6] Murdoch J. Gabbay, *A new calculus of contexts*, PPDP '05: Proc. of the 7th ACM SIGPLAN int'l conf. on Principles and Practice of Declarative Programming, ACM Press, 2005, pp. 94–105.

[7] ______, *Hierarchical nominal rewriting*, LFMTP'06: Logical Frameworks and Meta-Languages: Theory and Practice, 2006, pp. 32–47.

[8] Murdoch J. Gabbay and Aad Mathijssen, *Capture-avoiding substitution as a nominal algebra*, ICTAC'2006: 3rd Int'l Colloquium on Theoretical Aspects of Computing, 2006, pp. 198–212.

[9] Murdoch J. Gabbay and A. M. Pitts, *A new approach to abstract syntax with variable binding*, Formal Aspects of Computing **13** (2001), no. 3–5, 341–363.

[10] Herman Geuvers and Gueorgui I. Jojgov, *Open proofs and open terms: A basis for interactive logic*, CSL, Springer, 2002, pp. 537–552.

[11] Masatomo Hashimoto and Atsushi Ohori, *A typed context calculus*, Theor. Comput. Sci. **266** (2001), no. 1-2, 249–272.

[12] Dimitri Hendriks and Vincent van Oostrom, *Adbmal*, CADE, 2003, pp. 136–150.

[13] Gueorgui I. Jojgov, *Holes with binding power.*, TYPES, LNCS, vol. 2646, Springer, 2002, pp. 162–181.

[14] Samuel Kamin and Jean-Jacques Lévy, *Attempts for generalizing the recursive path orderings*, Handwritten paper, University of Illinois, 1980.

[15] Pierre Lescanne, *From lambda-sigma to lambda-upsilon a journey through calculi of explicit substitutions*, POPL '94: Proc. 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, 1994, pp. 60–69.

[16] Robin Milner, Joachim Parrow, and David Walker, *A calculus of mobile processes, II*, Information and Computation **100** (1992), no. 1, 41–77.

[17] Lawrence C. Paulson, *ML for the working programmer (2nd ed.)*, Cambridge University Press, 1996.

[18] A. M. Pitts, *Operationally-based theories of program equivalence*, Semantics and Logics of Computation (P. Dybjer and A. M. Pitts, eds.), Publications of the Newton Institute, Cambridge University Press, 1997, pp. 241–298.

[19] A. M. Pitts and Murdoch J. Gabbay, *A metalanguage for programming with bound names modulo renaming*, Mathematics of Program Construction. 5th Int'l Conf. , MPC2000, Ponte de Lima, Portugal, July 2000. Proceedings (R. Backhouse and J. N. Oliveira, eds.), LNCS, vol. 1837, Springer-Verlag, 2000, pp. 230–255.

[20] A. M. Pitts and I. D. B. Stark, *Operational reasoning for functions with local state*, Higher Order Operational Techniques in Semantics (A. D. Gordon and A. M. Pitts, eds.), Publications of the Newton Institute, Cambridge University Press, 1998, pp. 227–273.

[21] Masahiko Sato, Takafumi Sakurai, and Rod Burstall, *Explicit environments*, Fundamenta Informaticae **45:1-2** (2001), 79–115.

[22] Masahiko Sato, Takafumi Sakurai, and Yukiyoshi Kameyama, *A simply typed context calculus with first-class environments*, Journal of Functional and Logic Programming **2002** (2002), no. 4, 359 – 374.

[23] Masahiko Sato, Takafumi Sakurai, Yukiyoshi Kameyama, and Atsushi Igarashi, *Calculi of meta-variables*, Computer Science Logic and 8th Kurt Gödel Colloquium (CSL'03 & KGC), Vienna, Austria. Proccedings (M. Baaz, ed.), LNCS, vol. 2803, 2003, pp. 484–497.

[24] Francois Maurel Sylvain Baro, *The qnu and qnuk calculi : name capture and control*, Tech. report, Université Paris VII, 2003, Extended Abstract, Prépublication PPS//03/11//n16.

[25] Terese, *Term rewriting systems*, Cambridge Tracts in Theoretical Computer Science, no. 55, Cambridge University Press, 2003.

[26] Simon Thompson, *Haskell: The Craft of Functional Programming*, Addison Wesley, 1996.

[27] C. Urban, A. M. Pitts, and Murdoch J. Gabbay, *Nominal unification*, Theoretical Computer Science **323** (2004), no. 1–3, 473–497.

[28] Johan van Benthem, *Modal foundations for predicate logic*, Logic Journal of the IGPL **5** (1997), no. 2, 259–286.

[29] ______, *Higher-order logic*, Handbook of Philosophical Logic, 2nd Edition (D.M. Gabbay and F. Guenthner, eds.), vol. 1, Kluwer, 2001, pp. 189–244.

Acknowledgements. Supported by grant RYC-2006-002131 at the Polytechnic University of Madrid.