The lambda-context calculus (extended version)[1]

Murdoch J. Gabbay, Stéphane Lengrand

Abstract

We present the Lambda Context Calculus. This simple lambda-calculus features variables arranged in a hierarchy of strengths such that substitution of a strong variable does not avoid capture with respect to abstraction by a weaker variable. This allows the calculus to express both capture-avoiding *and* capturing substitution (instantiation). The reduction rules extend the 'vanilla' lambda-calculus in a simple and modular way and preserve the look and feel of a standard lambda-calculus with explicit substitutions.

Good properties of the lambda-calculus are preserved. The LamCC is confluent, and a natural injection into the LamCC of the untyped lambda-calculus exists and preserves strong normalisation.

We discuss the calculus and its design with full proofs. In the presence of the hierarchy of variables, functional binding splits into a functional abstraction λ (lambda) and a name-binder $\mathcal{N}$ (new). We investigate how the components of this calculus interact with each other and with the reduction rules, with examples. In two more extended case studies we demonstrate how global state can be expressed, and how contexts and contextual equivalence can be naturally internalised using function application.

Keywords: Lambda-calculus, calculi of contexts, functional programming, binders, nominal techniques, explicit substitutions, capturing substitution.

[1]This paper is an extended version of: Murdoch J. Gabbay and Stéphane Lengrand, *The lambda-context calculus*, Electronic Notes in Theoretical Computer Science 196, Elsevier, 2008, pages 19-35.

Contents

1 Introduction **3**
- 1.1 An outline of the LamCC as a λ-calculus with capturing substitution . . . 3
- 1.2 Motivation for considering capturing substitution: nominal terms and nominal algebra . . . 4
- 1.3 Motivation for considering capturing substitution: incomplete derivations . . . 4
- 1.4 Outline of the technical issues . . . 5

2 Syntax, freshness, reductions **6**
- 2.1 Syntax . . . 6
- 2.2 Levels and free variables . . . 6
- 2.3 α-equivalence . . . 7
- 2.4 Reductions . . . 9
- 2.5 Example reductions . . . 11
- 2.6 Comments on the side-conditions . . . 13

3 Properties of the explicit substitution $s[a_i \mapsto t]$ **15**
- 3.1 Termination of (**sigma**) . . . 15
- 3.2 A meta-level substitution action: $s[a_i := t]$ . . . 16
- 3.3 Calculating (**sigma**)-normal forms: s^* . . . 19

4 Confluence **20**
- 4.1 Confluence of (**sigma**) . . . 22
- 4.2 (**beta**)-reduction . . . 22
- 4.3 Combining **(sigma)** and **(beta)** . . . 25

5 The untyped lambda-calculus **27**

6 Contexts and contextual equivalence **29**

7 A NEW part for the LamCC **30**
- 7.1 Some NEW rules . . . 30
- 7.2 Some false NEW rules . . . 32
- 7.3 Global state using NEW . . . 32

8 Design variations **34**
- 8.1 Different schemes of levels . . . 34
- 8.2 LamCC with only one binder . . . 35
- 8.3 LamCC with nominal terms style alpha-equivalence . . . 36

9 Related work, conclusions, and future work **36**

1. Introduction

1.1. An outline of the LamCC as a λ-calculus with capturing substitution

In this paper we present the Lambda Context Calculus (**LamCC**). To a first approximation the LamCC is a λ-calculus with explicit substitutions. However, variables are arranged into a hierarchy of levels; we call variables higher up in the hierarchy *stronger*, and we call variables lower down in the hierarchy *weaker*. An explicit substitution for a strong variable acts without avoiding capture with respect to weaker variables. That is, strong variables undergo capturing substitution, which we will also call *instantiation*.

Here are two example LamCC reductions exemplifying this behaviour (full definitions follow in the body of the paper; these examples are taken from Subsection 2.5):

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

X is a variable of level 2 and x is a variable of level 1. t denotes any term.

The substitutions of different levels are compatible with β-reduction; here is a typical example:

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

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

The infinite hierarchy of levels guarantees that for every variable there is a stronger one. Thus for any abstraction by a variable, we can always find a stronger variable and ensure capturing substitution. We shall see how we can choose our levels carefully and mix capture-avoiding and capturing substitution. We shall also see that if a term mentions variables of only one level, then that term behaves like an 'ordinary λ-term'; thus, the usual programming power of the λ-calculus remains undisturbed within each level.

We intend the LamCC as a simple λ-calculus within which to study the combination of capture-avoiding substitution and capturing substitution in the context of functional programming.

The LamCC turns out to be expressive in unexpected ways — see the example reductions in Subsection 2.5, the analysis of contexts in Section 6, and the case study of global state in Subsection 7.3. The LamCC also has some unexpected new behaviour. In particular a split emerges between functional abstraction and name binding which manifests itself as λ (a well-known construct) and И. И is related to the Gabbay-Pitts 'new' quantifier [21], but is not identical to it; see Section 7. We take care to give examples and in particular to justify the inclusion of И in our calculus.

Capture-avoiding substitution is well-known and well-studied; it is the 'native' notion of substitution of the untyped λ-calculus. Some readers may ask why we should study capturing

substitution (instantiation). We devote the next two subsections to a discussion of two examples where instantiations naturally arise, both of which have become the topic of full investigation since this paper was first drafted [16, 19]. In the body of this paper the reader can also find many example programs showing things that can be done, given a capturing substitution.

1.2. Motivation for considering capturing substitution: nominal terms and nominal algebra

The authors became interested in capturing substitution because this is the substitution needed to directly model meta-variables at the 'informal meta-level'. The reduction of $(\lambda x.X)[X{\mapsto}x]$ to $\lambda x.x$ (given above) models what happens when we say 'let - be x is $\lambda x.$-'; meta-variables are *instantiated* with respect to object-level variables.

We came to this via nominal techniques [21]. Nominal techniques are, amongst other things, an approach to reasoning on abstract syntax with binding. This is in one sense a very simple situation; we are reasoning on abstract syntax trees up to binding, thus, the only equality we care about is α-equivalence. Yet a two level hierarchy of variables naturally manifests itself. For example consider the nominal term $[a]X$ which we read as 'abstract a in X' [45]. X is equipped with a *capturing* substitution, such that if X is instantiated to a we obtain $[a]a$.

The first author with Mathijssen has developed a logical theory based on two levels of variable and capturing substitution, *nominal algebra* [14, 16, 31] (the same ideas also appear in Pitts and Clouston's *nominal equational logic* [8]). There, capturing substitution of strong variables happens at the meta-level; it is a universal algebra system, so variables are *implicitly* universally quantified.

This behaviour is explicitly modelled in the LamCC, for example in the reduction of $(\lambda x.X)[X{\mapsto}x]$ above. That is, with the LamCC we internalise and study the computational action of instantiating a meta-variable. Of course, once we have internalised the informal meta-level once we still need another informal meta-level to discuss the extended language. If we repeat this process we arrive an a kind of fixed point where we have an infinite hierarchy of variables $1, 2, 3, 4, \ldots$ Thus, intuitively, if we internalise meta-variables, then allow λ-abstraction to internalise computing on those variables, then internalise meta-variables again, and then repeat this indefinitely — then we arrive at the LamCC.

1.3. Motivation for considering capturing substitution: incomplete derivations

We continue the motivational discussion of the previous subsection.

Consider formalising mathematics in a logical framework based on Higher-Order Logic (**HOL**) [46]. 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.

$$\dfrac{\dfrac{A{\Rightarrow}B{\Rightarrow}C \quad [A]^i}{B{\Rightarrow}C} \qquad \dfrac{?}{B}}{\dfrac{C}{A{\Rightarrow}C}\,{}^i} \tag{1}$$

$$\dfrac{\dfrac{A{\Rightarrow}B{\Rightarrow}C \quad [A]^i}{B{\Rightarrow}C} \qquad \dfrac{A{\Rightarrow}B \quad [A]^i}{B}}{\dfrac{C}{A{\Rightarrow}C}\,{}^i} \tag{2}$$

Above are two derivations of $A{\Rightarrow}B{\Rightarrow}C, A{\Rightarrow}B \vdash A{\Rightarrow}C$.[2] (1) is incomplete, (2) has been completed by instantiating $?$ to $\dfrac{A \Rightarrow B \quad A}{B}$.

[2] This example 'borrowed' from [22].

In the incomplete derivation (1), the right-most $[A]^i$ is *discharged* by the bottom-most inference step; thus, in general we require the ability to instantiate ? for a derivation some of whose assumptions *are discharged*. Discharge corresponds in the Curry-Howard correspondence with λ-abstraction. The complete derivation in (2) can be represented by the λ-term

$$\lambda x.y^{A\Rightarrow B\Rightarrow C}\ x^A\ (z^{A\Rightarrow B}\ x^A).$$

(We add types to make as explicit as possible the connection with the derivation.) The incomplete derivation can be represented by

$$\lambda x.y^{A\Rightarrow B\Rightarrow C}\ x^A\ X^B.$$

Completing the derivation corresponds with the *capturing* substitution

$$(\lambda x.y\ x\ X)[X\mapsto z\ x] \rightsquigarrow^* \lambda x.y\ x\ (z\ x).$$

Similar issues arise with existential variables [22, Section 2, Example 3].

The LamCC is an untyped λ-calculus; we will not address Curry-Howard for incomplete derivations in this paper. However, the basic technology developed in this paper has been used to investigate this elsewhere [18].[3] In fact [18] (a conference paper) is just a start because we only consider how to represent incomplete proofs, and we do not give a reduction system — but it exists, it is based on a fragment of the LamCC reduction system, and it is contained in a journal version.

We would like the reader to understand the LamCC as distilling capturing and capture-avoiding substitution into a relatively simple untyped-λ-calculus-like package (indeed, it contains infinitely many copies of the untyped λ-calculus, one for each level). Other notable features are its expressivity, the separation of functional abstraction and name-binding, and an infinite hierarchy of variables. This combination seems novel and the results seem elegant.

In the next section, we ask what issues arise when we internalise strong variables and design a multi-level λ-calculus.

1.4. Outline of the technical issues

The central issue, if we internalise strong variables, is with α-equivalence. Let x, y, z be have level 1 and let X have level 2. If $\lambda x.X$ is α-equivalent with $\lambda y.X$ (because 'x and y do not occur in X') then $(\lambda X.\lambda x.X)x$ reduces to $\lambda y.x$. This gives non-confluent reductions. This observation goes back at least to [29].

Dropping α-equivalence entirely is too drastic; we need $\lambda y.\lambda x.y$ to be α-convertible with $\lambda z.\lambda x.z$ so that we can reduce a term such as $(\lambda y.\lambda x.y)x$.

The LamCC is not the first attempt to address these issues in the context of a λ-calculus. Solutions include clever control of substitution and evaluation order [39], types to prevent 'bad' α-conversions [37, 24, 38], explicit labels on meta-variables [22, 26], and more [6, Section 2]. The complexity of the LamCC, the results we are able to prove of it, and the programs we find we can write in it, all compare favourably with previous work. The LamCC may be the closest to the untyped λ-calculus that anybody has so far managed to come with a reduction system with more than one level of variable. We discuss this further in the Conclusions.

The first author previously developed the NEWcc [11], with similar goals. The LamCC has simpler and more intuitive reduction rules. Indeed there is now only *one* non-obvious side-condition, which is on $(\sigma\mathbf{p})$ (see the discussion of the side-conditions in Subsection 2.6). Technical aspects of the previous work have been simplified, clarified, or eliminated altogether. Notably we dispense entirely with the freshness contexts and freshness logic of the NEWcc.

The result is a clean and simple system which is powerfully expressive yet which preserves the look and feel of an ordinary λ-calculus with explicit substitutions, along with many of the properties which make the λ-calculus so nice to work with. These include confluence (Section 4)

[3] Note to the referees: [18] was written and published after this paper.

and a natural translation of the untyped λ-calculus which preserves strong normalisation (Section 5). Because of the hierarchy of variables, contexts and contextual equivalence can be internalised (Section 6), and the LamCC is interestingly expressive in other ways; aside from expressing contexts (Section 6) we give example reductions in Subsection 2.5 and as an extended case study we express global state in Subsection 7.3.

2. Syntax, freshness, reductions

2.1. Syntax

Definition 2.1. *We suppose a countably infinite set of disjoint infinite* **sets of variables**[4] $\mathbb{A}_1$, $\mathbb{A}_2$, ..., *where we write* $a_i, b_i, c_i, n_i, \ldots \in \mathbb{A}_i$ *for* $i \geq 1$.

We shall use a **permutative naming convention**; we shall always assume that variables that we give different names, *are* different. Thus a_i and b_i range *permutatively* over elements of $\mathbb{A}_i$. In particular, when we write a_i and c_k we do *not* assume (unless stated otherwise) that $i \neq k$, but we *do* assume that even if $i = k$, a_i and c_k are different variables, because we have given them different names.

There is no particular connection between variables of different levels with the same name. For example a_1 and a_2 are different variables to which we happen to have given similar names.

Definition 2.2. *The syntax of the lambda context calculus (LamCC) is given by:*

$$s, t ::= a_i \mid tt \mid \lambda a_i.t \mid t[a_i \mapsto t].$$

Note that in Subsection 2.3 we develop a notion of α-equivalence; from that point on we will equate terms up to α-equivalence.

We use the following conventions:

- As is standard, application associates to the left. For example $tt't''$ is $(tt')t''$.
- We say that the variable a_i **has level** i.
- We call b_j **stronger** than a_i when $j > i$, that is, when b_j has higher level than a_i. If $j < i$ we call b_j **weaker** than a_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 (for the atom a, acting on s).
- We call $\lambda a_i.t$ an **abstraction** of level i (over the term t).
- Later on we shall use the convention that x, y, z are variables of level 1, X, Y, Z are variables of level 2, and $\mathcal{W}$ has level 3.

This syntax has no constant symbols. We might like to have constants like 1, 2, 3, ...for arithmetic, or $\top$ and $\bot$ for truth-values. These behave much like variables 'of level 0' which we do not abstract over and for which we do not substitute. We may add them where convenient for illustrative examples. Adding them formally to the syntax causes no particular difficulties aside from adding extra cases to a few proofs.

2.2. Levels and free variables

The hierarchy of variables induces a hierarchy of terms, characterised by their levels:

Suppose that i, j, and k are numbers. Write $\mathsf{max}(i, j)$ for the greater of i and j, and $\mathsf{max}(i, j, k)$ for the greatest of i, j, and k.

[4] It might be better to call these variable *symbols*, but we shall not bother.

$$
\begin{aligned}
\mathsf{level}(a_i) &= i \\
\mathsf{level}(ss') &= \mathsf{max}(\mathsf{level}(s), \mathsf{level}(s')) \\
\mathsf{level}(\lambda a_i.s) &= \mathsf{max}(i, \mathsf{level}(s)) \\
\mathsf{level}(s[a_i \mapsto t]) &= \mathsf{max}(i, \mathsf{level}(s), \mathsf{level}(t))
\end{aligned}
$$

Figure 1: Level $\mathsf{level}(s)$

$$
\begin{aligned}
fv(a_i) &= \{a_i\} \\
fv(\lambda a_i.s) &= fv(s) \setminus \{a_i\} \\
fv(s[a_i \mapsto t]) &= (fv(s) \setminus \{a_i\}) \cup fv(t) \\
fv(st) &= fv(s) \cup fv(t)
\end{aligned}
$$

Figure 2: Free variables $fv(s)$

Definition 2.3. *Define the* **level** $\mathsf{level}(s)$ *by the rules in Figure 1, and the* **free variables** $fv(s)$ *by the rules in Figure 2.*

If we ignore levels then $fv(s)$ is intuitively just the usual notion of 'free variables of' in the untyped λ-calculus, if we read $s[a_1 \mapsto t]$ as $(\lambda a_1.s)t$, and $fv(s)$ is directly the usual notion of 'free variables of' of the calculus with explicit substitutions λx [5]). When in Section 5 we encode the untyped λ-calculus in LamCC using only variables of level 1, this intuition finds formal expression in Lemma 5.1.

For the reader's convenience we mention now that we write '$\mathsf{level}(s_1, \ldots, s_n) \leq i$' as shorthand for '$\mathsf{level}(s_1) \leq i$ and $\ldots$ and $\mathsf{level}(s_n) \leq i$', similarly for '$\mathsf{level}(s_1, \ldots, s_n) < i$'; this will be useful later.

2.3. α-equivalence

The hierarchy of variables introduces subtleties to the management of variable renaming and introduces a non-trivial notion of α-equivalence. (In return, we will then be able to formalise the capturing substitution mentioned in the Introduction.) First, we must define notions of *congruence* and of *variable (symbol) swapping*.

Definition 2.4. *A* **congruence** *on LamCC terms is a binary relation $s \ R \ s'$ satisfying the conditions of Figure 3.*

It is easy to show that a congruence is reflexive, so perhaps a better name would be 'congruent equivalence relation' — but 'congruence' is shorter so we use that.

Definition 2.5. *Define an* **(atoms) swapping** $(a_i \ b_i)s$ *action inductively by the rules in Figure 4.*

$$
\frac{}{a_i \ R \ a_i} \qquad \frac{s \ R \ s' \quad t \ R \ t'}{st \ R \ s't'} \qquad \frac{s \ R \ s' \quad t \ R \ t'}{s[a_i \mapsto s'] \ R \ t[a_i \mapsto t']}
$$

$$
\frac{s \ R \ s'}{\lambda a_i.s \ R \ \lambda a_i.s'} \qquad \frac{s \ R \ s'}{s' \ R \ s} \qquad \frac{s \ R \ s' \quad s' \ R \ s''}{s \ R \ s''}
$$

Figure 3: Rules for a congruence

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

Figure 4: Rules for swapping

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

Figure 5: Rules for α-equivalence

Note that the definition of the swapping action is uniform, swapping a_i and b_i in s without regard for binders (and without capture-avoidance conditions or capture-avoiding renaming substitutinos). This is characteristic of the underlying 'nominal' method of this paper [21, 45].

We will use swapping $(a_i\ b_i)$ on sets of variables S acting pointwise by

$$(a_i\ b_i)S = \{(a_i\ b_i)c \mid c \in S\}.$$

Here c ranges over all elements of S, including a_i and b_i (if they are in S).

Lemma 2.6. *$fv((a_i\ b_i)s) = (a_i\ b_i)fv(s)$ and $\mathsf{level}((a_i\ b_i)s) = \mathsf{level}(s)$.*

Proof. By easy inductions on the definition of $(a_i\ b_i)s$. □

Definition 2.7. *If S is a set of variables write $a_i\#S$ for*

- *$a_i \notin S$, and*
- *there exists no variable $b_j \in S$ such that $j > i$.*

Definition 2.8. *Let* **α-equivalence** *be the least congruence relation $s =_\alpha s'$ satisfying the conditions of Figure 5.*

Suppose that x and y have level 1 and X has level 2. We can easily verify that:

- a_i may be α-converted in $\lambda a_i.s$ if $\mathsf{level}(s) \leq i$. In particular $\lambda x.x =_\alpha \lambda y.y$.
- a_i may be α-converted in $s[a_i{\mapsto}t]$ if $\mathsf{level}(s) \leq i$. In particular $x[x{\mapsto}X] =_\alpha y[y{\mapsto}X]$.
- It is not possible to α-convert a_i in s if $b_j \in fv(s)$ 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 can never α-convert variables to variables of *other* levels.

Remark 2.9. It is easy to see that in the fragment of LamCC syntax consisting of terms mentioning variables of level 1 only, the condition $b_i\#fv(s)$ collapses to $b_i \notin fv(s)$, and α-equivalence collapses to the usual α-equivalence on untyped λ-terms plus an explicit substitution as in the calculus λx [5].

Remark 2.10. The definitions above are descended from the notion of α-equivalence for nominal terms from [45]. In the terminology used here, [45] considered a syntax with a hierarchy with just levels 1 and 2 and no abstraction over variables of level 2. However LamCC is weaker in the sense that nominal terms include swappings in the syntax of terms. We use swappings as an operation *on* LamCC syntax (Definition 2.5) but we have not included them in the syntax of the LamCC itself.

Extending LamCC syntax with swappings is future work. To do that it would help to have a better understanding of LamCC models (this paper is purely syntactic) and of freshness $\#$. Two other papers explore the notion of freshness in the presence of a hierarchy [12], and the notion of full nominal-terms style α-equivalence based on swappings in the syntax [20]. Much future work is possible here.

Theorem 2.11. *If* $s =_\alpha s'$ *then* $fv(s) = fv(s')$ *and* $\mathsf{level}(s) = \mathsf{level}(s')$.

Proof. The proof is by an easy induction on the derivation rule defining a congruence. We give only the base cases:

- If $b_i \# fv(s)$ then $fv(\lambda a_i.s) =_\alpha fv(\lambda b_i.(b_i\ a_i)s)$.

  Suppose $\mathsf{level}(s) \leq i$. By Lemma 2.6 $\mathsf{level}((b_i\ a_i)s) \leq i$ as well. Then using Lemma 2.6 we have

  $$fv(\lambda a_i.s) = fv(s) \setminus \{a_i\} \qquad fv(\lambda b_i.(b_i\ a_i)s) = ((b_i\ a_i)fv(s)) \setminus \{b_i\}.$$

  It follows by easy set calculations that

  $$fv(\lambda a_i.s) = fv(\lambda b_i.(b_i\ a_i)s).$$

  We also observe that swapping a_i and b_i in s has no effect on the levels of the variables occurring in s, and it follows easily that

  $$\mathsf{level}(\lambda a_i.s) = \mathsf{level}(\lambda b_i.(b_i\ a_i)s).$$

- If $b_i \# fv(s)$ then $fv(s[a_i \mapsto t]) = fv(((b_i\ a_i))s[b_i \mapsto t])$.

  Note that

  $$fv(s[a_i \mapsto t]) = fv(\lambda a_i.s) \cup fv(t) \qquad \text{and}$$
  $$fv(((b_i\ a_i))s[b_i \mapsto t]) = fv(\lambda b_i.(b_i\ a_i)s) \cup fv(t).$$

  We use the previous part.

□

From now on we shall consider terms up to α-equivalence, as for the standard λ-calculus (without levels). Theorem 2.11 allows us to discuss the levels and sets of free variables of α-equivalence classes.

2.4. Reductions

Definition 2.12. *Define the* **reduction relation** *on terms (modulo α-equivalence) inductively by the rules in Figure 6.*

In Figure 6, consistent with our conventions, variables with different names are assumed distinct. For example in $(\sigma\lambda')$ we assume that a_i and c_i are distinct.

We shall see in Section 5 that our rewrite system refines β-reduction. Indeed, when only variables of level 1 are present, LamCC becomes isomorphic to the explicit substitution calculus λx with garbage collection [5].

Our side-conditions allow us to generalise the rules to the full hierarchy of variables. These side-conditions are mostly natural; we give brief intuitions.

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

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

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

Figure 6: Reduction rules of the LamCC

- (σfv) features a garbage side-condition that 'a_i does not occur in s, even if stronger variables are instantiated.'
- $(\sigma\sigma)$ and $(\sigma\lambda)$ feature natural side-conditions for the new *capturing* propagation of substitution.
- $(\sigma\lambda')$ has the usual capture-avoidance side-condition of (single-level) λ-calculi.
- The side-condition for $(\sigma\mathbf{p})$ is non-trivial and plays a key role in ensuring confluence. It disambiguates the semantics of substitutions competing to instantiate the same variable.

Subsection 2.5 illustrates with examples how these rules and their side-conditions interact. Subsection 2.6 discusses the side-conditions in greater detail, including the side-condition on $(\sigma\mathbf{p})$, with examples of what goes wrong if we relax them.

We take a moment to collect some standard notation for reductions, which will be useful in the rest of this paper:

- 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 $s \overset{(\mathbf{ruleset})}{\rightsquigarrow} t$ when we can deduce $s \rightsquigarrow t$ but using only rules in $(\mathbf{ruleset})$ where

  $$(\mathbf{ruleset}) \subseteq \{(\beta), (\sigma\mathbf{a}), (\sigma fv), (\sigma\mathbf{p}), (\sigma\sigma), (\sigma\lambda), (\sigma\lambda')\}.$$

  (Later in Section 7 we extend reduction with rules for a binder И.)
- We call $\rightsquigarrow$ **terminating** when there is no infinite sequences

  $$t_1 \rightsquigarrow \cdots \rightsquigarrow t_i \rightsquigarrow \cdots.$$

  Similarly for $\overset{(\mathbf{ruleset})}{\rightsquigarrow}$.
- We 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 $\overset{(\mathbf{ruleset})}{\rightsquigarrow}$.

This is all standard [43, 2].

2.5. Example reductions

The LamCC is a λ-calculus with explicit substitutions [30]. 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 2.6 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-reduction.** This is a standard β-reduction rule for a calculus with explicit substitutions:

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

- **Substitutions on variables.** The behaviour of a substitution on a variable depends on the relative strengths of the variable being substituted on, and the variable being substituted for:

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

  We can summarise the behaviour of substitutions on variables as follows:

  - A strong substitution acting on a weak variable 'evaporates'.
  - A substitution of a variable acting on itself acts 'normally'.
  - A substitution of a variable acting on another variable of the same strength 'evaporates'.
  - A weak substitution on a strong variable 'stays put'.

- **Traversing weak variables.**

  We reprise the examples discussed in the Introduction (before we had formally introduced LamCC syntax and operational semantics).

  An explicit substitution for a relatively strong variable may distribute using $(\sigma\sigma)$ under an explicit substitution for a relatively weaker variable, without avoiding capture:

$$\begin{aligned} X[x{\mapsto}t][X{\mapsto}x] &\overset{(\sigma\sigma)}{\rightsquigarrow} X[X{\mapsto}x][x{\mapsto}t[X{\mapsto}x]] \\ &\overset{(\sigma\mathbf{a})}{\rightsquigarrow} x[x{\mapsto}t[X{\mapsto}x]] \\ &\overset{(\sigma\mathbf{a})}{\rightsquigarrow} t[X{\mapsto}x]. \end{aligned}$$

  Similarly, an explicit substitutions for a relatively strong variable can traverse a λ-abstraction by a relatively weaker variable, using $(\sigma\lambda)$, without avoiding capture:

$$\begin{aligned} (\lambda x.X)[X{\mapsto}x] &\overset{(\sigma\lambda)}{\rightsquigarrow} \lambda x.(X[X{\mapsto}x]) \\ &\overset{(\sigma\mathbf{a})}{\rightsquigarrow} \lambda x.x. \end{aligned}$$

  This makes strong variables behave like 'holes'. Instantiation of holes is compatible with β-

reduction; here is a typical example:

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

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

- **Substitutions on no weaker substitutions.** These terms do *not* reduce:

$$X[x{\mapsto}z][y{\mapsto}z] \not\rightsquigarrow \qquad X[x{\mapsto}y][y{\mapsto}z] \not\rightsquigarrow .$$

  Here $(\sigma\mathbf{a})$ and (σfv) are not applicable because X has level 2 and x has level 1, and $(\sigma\sigma)$ is not applicable because both x and y have level 1. So weak substitutions are 'suspended' — until a stronger substitution turns X into something with internal structure which they can act on.

  We imagine stronger versions of the LamCC (i.e. with more reductions) in the Conclusions.

- **Substitutions for stronger terms** There is *no* restriction in $s[a_i{\mapsto}t]$ that $\mathsf{level}(t) < i$ or $\mathsf{level}(t) \leq i$; for example the terms $X[x{\mapsto}Y]$ and $X[x{\mapsto}\mathcal{W}]$ are legal (recall that $\mathcal{W}$ has level 3).

  Terms like $X[x{\mapsto}Y]$ are useful. Examples 'in nature' appear in the $\exists$-introduction rule in logic

$$\frac{\Gamma \vdash \phi[a{\mapsto}t]}{\Gamma \vdash \exists a.\phi}$$

  where it is understood that ϕ and t are meta-variables — and in the β-reduction rule

$$(\lambda a.s)t \rightsquigarrow s[a{\mapsto}t]$$

  where it is understood that s and t are meta-variables.

  A term such as $X[a{\mapsto}\mathcal{W}]$ is useful if there is a surrounding binder which we would like to conveniently link to. For example in the term $\lambda X.(X[x{\mapsto}\mathcal{W}])$ the variable $\mathcal{W}$ can be bound to X by a substitution arriving from some enclosing context:

$$(\lambda X.(X[x{\mapsto}\mathcal{W}]))[\mathcal{W}{\mapsto}X] \rightsquigarrow^* \lambda X.(X[x{\mapsto}X]).$$

- **Substitution-as-term** $[x{\mapsto}y]$ is not a term; it cannot be the argument to a function. However $\lambda X.X[x{\mapsto}y]$ *is* a term, and it acts like a substitution in the following sense:

$$\begin{aligned}(\lambda X.X[x{\mapsto}y])t &\overset{(\beta)}{\rightsquigarrow} X[x{\mapsto}y][X{\mapsto}t] \overset{(\sigma\sigma)}{\rightsquigarrow} X[X{\mapsto}t][x{\mapsto}y[X{\mapsto}t]]\\ &\overset{(\sigma fv)}{\rightsquigarrow} X[X{\mapsto}t][x{\mapsto}y]\\ &\overset{(\sigma\mathbf{a})}{\rightsquigarrow} t[x{\mapsto}y].\end{aligned}$$

As a general scheme, $\lambda b_j.b_j[a_i{\mapsto}s]$ encodes the substitution $[a_i{\mapsto}s]$ if $\mathsf{level}(s) \leq j$ and $i < j$.

We know of no other work in the literature which represents substitutions by λ-terms by abstracting on variables of higher levels. Representations of substitutions as first-class citizens in λ-calculi in the literature use a specific syntactic category, distinct from (though interacting with) that of λ-terms (see for example [30]).

- **Records using substitution** We obtain a model of records using the hierarchy of levels. Suppose x and y are variables of level 1, $\mathbf{W}$ is a variable of level 2, and suppose 2 and 3 are constants (just for convenience). Write $t.a_i$ ('lookup a_i') for $t[\mathbf{W}{\mapsto}a_i]$ and $t.a_i := u$ ('bind a_i to u') for $t[\mathbf{W}{\mapsto}\mathbf{W}[a_i{\mapsto}u]]$. Then

$$((\mathbf{W}.x := 2).y := 3).x \rightsquigarrow^* 2 \quad \text{and} \quad ((\mathbf{W}.x := 2).x := 3).x \rightsquigarrow^* 3.$$

For the reader's convenience we expand this a little; full details are easy to verify:

$$\begin{aligned}\mathbf{W}[\mathbf{W}{\mapsto}\mathbf{W}[x{\mapsto}2]][\mathbf{W}{\mapsto}\mathbf{W}[y{\mapsto}3]][\mathbf{W}{\mapsto}x] &\rightsquigarrow^* \mathbf{W}[y{\mapsto}3][x{\mapsto}2][\mathbf{W}{\mapsto}x]\\ &\rightsquigarrow^* 2\end{aligned}$$

$$\begin{aligned}\mathbf{W}[\mathbf{W}{\mapsto}\mathbf{W}[x{\mapsto}2]][\mathbf{W}{\mapsto}\mathbf{W}[x{\mapsto}3]][\mathbf{W}{\mapsto}x] &\rightsquigarrow^* \mathbf{W}[x{\mapsto}3][x{\mapsto}2][\mathbf{W}{\mapsto}x]\\ &\rightsquigarrow^* 3\end{aligned}$$

This is the behaviour we expect if we view $\mathbf{W}$ with some suspended level 1 substitutions as a record binding the variables to the expressions substituted for. This model can be refined in a number of ways, for example using $\lambda\mathbf{W}.(\mathbf{W}[\mathit{substitutions}])$ and application, instead of $\mathbf{W}[\mathit{substitutions}]$ and substitution for $\mathbf{W}$.

More on this in Subsection 7.3.

2.6. Comments on the side-conditions

The side-conditions of the LamCC reduction rules are where much of the technical 'magic' happens.

- *The side-condition $a_i\#fv(s)$ in (σfv).*

  (σfv) is a form of garbage-collection. For example with (σfv) we can garbage-collect $[x{\mapsto}2]$ in $(\lambda X.X)[x{\mapsto}2]$:

$$(\lambda X.X)[x{\mapsto}2] \overset{(\sigma fv)}{\rightsquigarrow} \lambda X.X.$$

  In a calculus of explicit substitutions we can usually 'push substitutions into a term until they reach variables'. Therefore, we can make do with a rule of the form $b[a{\mapsto}t] \rightsquigarrow b$. A version of (σfv) appears in the literature as Bloo's 'garbage collection' [5]. There the rule is dispensable — removing it does not affect the transitive symmetric closure of the reduction relation — but the designers had other reasons to include it [5, Remark 2.15]. In the LamCC we cannot always push substitutions into a term (in particular we cannot push $[x{\mapsto}2]$ under λX above) because of the conditions on $(\sigma\mathbf{p})$ and $(\sigma\lambda')$. Thus, if we want $(\lambda X.X)[x{\mapsto}2]$ to be related with $\lambda X.X$ by the transitive reflexive closure of the reduction relation, then (σfv) seems indispensable.

  Note that we do not 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 $a_i \notin fv(s)$. For example $x \notin fv(X) = \{X\}$ and so if we change our rewrite system by removing the reduction rule (σfv) and replacing it with a *false*

version of the rule with a side-condition that uses $\notin$ instead of $\#$, then we have reductions

$$\begin{array}{lcl} X[x{\mapsto}2][X{\mapsto}x] & \overset{(\sigma fv\mathbf{FALSE})}{\rightsquigarrow} & X[X{\mapsto}x] \\ & \overset{\sigma a}{\rightsquigarrow} & x \end{array}$$

$$\begin{array}{lcl} X[x{\mapsto}2][X{\mapsto}x] & \overset{(\sigma\sigma)}{\rightsquigarrow} & X[X{\mapsto}x][x{\mapsto}2[X{\mapsto}x]] \\ & \overset{(\sigma\mathbf{a}),(\sigma fv\mathbf{FALSE})}{\rightsquigarrow^*} & 2. \end{array}$$

This is blocked in LamCC, because $x\#X$ does not hold.

- *The side-condition* $\mathsf{level}(s,s',t) \leq i$ *in* $(\sigma\mathbf{p})$.

  Recall that the level of a term is the level of the strongest variable it contains, free *or bound*. Recall also that the side-condition $\mathsf{level}(s,s',t) \leq i$ in $(\sigma\mathbf{p})$ means that $\mathsf{level}(s) \leq i$ and similarly for s' and t.

  Capturing substitution allows two substitutions $[x{\mapsto}2]$ and $[x{\mapsto}3]$ to compete to instantiate of x. Here is what happens if we drop the side-condition $\mathsf{level}(s,s',t) \leq i$:

$$\begin{array}{lcl} ((\lambda X.X[x{\mapsto}3])\ x)[x{\mapsto}2] & \overset{(\sigma\mathbf{pFALSE})}{\rightsquigarrow} & (\lambda X.X[x{\mapsto}3])[x{\mapsto}2]\ (x[x{\mapsto}2]) \\ & \overset{(\sigma\mathbf{a})}{\rightsquigarrow} & (\lambda X.X[x{\mapsto}3])[x{\mapsto}2]\ 2 \end{array}$$

$$\begin{array}{lcl} ((\lambda X.X[x{\mapsto}3])\ x)[x{\mapsto}2] & \overset{(\beta)}{\rightsquigarrow} & X[x{\mapsto}3][X{\mapsto}x][x{\mapsto}2] \\ & \overset{(\sigma\sigma)}{\rightsquigarrow} & X[X{\mapsto}x][x{\mapsto}3[X{\mapsto}x]][x{\mapsto}2] \\ & \rightsquigarrow^* & x[x{\mapsto}3][x{\mapsto}2] \\ & \rightsquigarrow^* & 3 \end{array}$$

  Our intuition is that, to disambiguate this situation, the inner redex should take priority over the distribution of $[x{\mapsto}3]$ because it involves X which has level 2 and is stronger. This intuition is implemented by the side-condition of $(\sigma\mathbf{p})$, which acts crucially in our proofs to guarantee confluence. Investigating whether this side-condition can be sensibly weakened, while still ensuring confluence, remains future work.

- *The side-conditions on* $(\sigma\sigma)$, $(\sigma\lambda)$, *and* $(\sigma\lambda')$.

  These rules express that a relatively strong substitution can capture, and that substitution for variables of the same level avoids capture. There is no rule

$$(\sigma\sigma'\mathbf{FALSE}) \quad \mathbf{s}[\mathbf{a_i}{\mapsto}\mathbf{t}][\mathbf{c_k}{\mapsto}\mathbf{u}] \rightsquigarrow \mathbf{s}[\mathbf{c_i}{\mapsto}\mathbf{u}][\mathbf{a_i}{\mapsto}\mathbf{t}[\mathbf{c_i}{\mapsto}\mathbf{u}]] \quad \mathbf{a_i}\#fv(\mathbf{u}),\ \mathbf{k} \leq \mathbf{i}$$

  since that would destroy termination of the part of the LamCC without λ — and we have managed to get confluence without it.

- *More on* $(\sigma\lambda)$ *and* $(\sigma\lambda')$.

  There is no rule permitting a weak substitution to propagate under a *stronger* abstraction, *even if* we avoid capture:

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

  Such a rule would cause the following problem for confluence:

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

Neither of these terms reduces further.

As is the case for the side-condition of $(\sigma\mathbf{p})$, any stronger form of $(\sigma\lambda')$ 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. Properties of the explicit substitution $s[a_i \mapsto t]$

Definition 3.1. *Let* $(\mathbf{sigma})$ *be equal to the set of rewrite rules* other *than* (β)*, namely,*

$$(\mathbf{sigma}) = \{(\sigma\mathbf{a}), (\sigma fv), (\sigma\mathbf{p}), (\sigma\sigma), (\sigma\lambda), (\sigma\lambda')\}.$$

(See Definition 4.5 for $(\mathbf{beta})$, the sister-set to $(\mathbf{sigma})$.)

This is the part of the LamCC that handles substitution — the 'λ-free' part of the calculus.

We expect the λ-free part of other calculi of explicit substitutions to be terminating [5, 30] and this is useful behaviour — but now we have a hierarchy of variables. Do we lose this good behaviour? *No*, and in this section we prove it.

3.1. *Termination of* $(\mathbf{sigma})$

We show that $(\mathbf{sigma})$-reduction decreases terms with respect to a well-founded ordering based on mapping LamCC terms to first-order terms (no variables, no binders), and using a *lexicographic path ordering* [27, 2] on them.

We use first-order terms in the following infinite signature:

$$\Sigma = \{\star/0, \mathsf{Abs}/1, \mathsf{App}/2\} \cup \{\mathsf{Sub}^i/2 \mid i \text{ is an integer}\}.$$

Here f/n indicates that f has arity n. Symbols have the following precedence:

$$\mathsf{Sub}^j \succ \cdots \succ \mathsf{Sub}^i \succ \cdots \succ \mathsf{App} \succ \mathsf{Abs} \succ \star \qquad \text{if } j > i$$

We define the **lexicographic path ordering** by:

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

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

Here g/m and f/n are first-order symbols 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 [27, 2] 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 $s \ll t$.

We define a translation from LamCC to first-order terms as follows:

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

Theorem 3.2. *If* $t \overset{(\mathbf{sigma})}{\rightsquigarrow} u$ *then* $\overline{t} \gg \overline{u}$.

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

Figure 7: Substitution action

Proof. We check for each reduction rule that the corresponding first-order term on the left, is higher in the lexicographic path ordering than the one on the right. This is routine:

$$
\begin{array}{lll}
(\sigma\mathbf{a}) & \mathsf{Sub}^i(\star,t) & \gg t \\
(\sigma fv) & \mathsf{Sub}^i(s,t) & \gg s \\
(\sigma\mathbf{p}) & \mathsf{Sub}^i(\mathsf{App}(s,s'),t) & \gg \mathsf{App}(\mathsf{Sub}^i(s,t),\mathsf{Sub}^i(s',t)) \\
(\sigma\sigma) & \mathsf{Sub}^j(\mathsf{Sub}^i(s,t),u) & \gg \mathsf{Sub}^i(\mathsf{Sub}^j(s,u),\mathsf{Sub}^j(t,u)) \quad i<j \\
(\sigma\lambda) & \mathsf{Sub}^j(\mathsf{Abs}(s),u) & \gg \mathsf{Abs}(\mathsf{Sub}^j(s,u)) \\
(\sigma\lambda') & \mathsf{Sub}^k(\mathsf{Abs}(s),u) & \gg \mathsf{Abs}(\mathsf{Sub}^k(s,u))
\end{array}
$$

□

Corollary 3.3. (**sigma**)*-reduction terminates.*

We can now make a useful observation. Let x have level 1. It is easy to show that $(\lambda x.xx)(\lambda x.xx)$ has an infinite series of reductions if we allow rules in (**sigma**) *and* (β). It follows that — even with a hierarchy of variables — (β) strictly adds to the power of the reduction system.

Definition 3.4. *Call* s (**sigma**)**-normal** *when* $s \overset{(\mathbf{sigma})}{\not\rightarrow}$.

By Corollary 3.3, any chain of (**sigma**)-reductions must terminate, and by definition it terminates at a (**sigma**)-normal form.

Note that in Subsection 3.3 we give an algorithm for calculating a (**sigma**)-normal form.

3.2. A meta-level substitution action: $s[a_i{:=}t]$

In the 'normal' λ-calculus, β-reduction is an atomic step. This is because substitution is a meta-level operation.

In the LamCC substitution is managed by the reduction relation, but this was a design choice which could have been made differently. We could have used a substitution action at the meta-level, which we would write as $s[a_i{:=}t]$. This would avoid proving confluence of (**sigma**), at the expense of (not proving confluence of (**sigma**) and) a less intuitive and more powerfully reducing β-reduction rule using $s[a_i{:=}t]$.

In any case, to prove confluence it is useful to define $s[a_i{:=}t]$:

Definition 3.5. *Define a* **substitution action** $s[a_i{:=}t]$ *by the rules in Figure 7. In Figure 7 the precedence of which equality to use is from top to bottom. When we try to apply the equality* $(\lambda c_i.s)[a_i{:=}t] = \lambda c_i.(s[a_i{:=}t])$ *we rename* c_i *where possible to satisfy the side condition* $c_i\#fv(t)$.

Lemma 3.6 is an important basic correctness property:

Lemma 3.6. $s[a_i{\mapsto}t] \overset{(\mathbf{sigma})}{\rightsquigarrow^*} s[a_i{:=}t]$.

Proof. By induction on i and then s. We inspect the definition of $:=$ and see that each clause can be imitated by a rule for $[a_i{\mapsto}t]$. □

Theorems 3.10 and 3.12 are technical properties of meta-level substitution; they reflect well-known properties of 'ordinary' substitution and they will be useful later.

We need some technical lemmas:

Lemma 3.7. *If* $\mathsf{level}(t) < j$ *then* $b_j \# fv(t)$.

Proof. By a routine induction on the definition of $\mathsf{level}(t)$ (Figure 1) using the definition of $fv(t)$ (Figure 2). □

We need two technical lemmas for Theorem 3.10:

Lemma 3.8. *If* $\mathsf{level}(t) < j$ *then* $t[b_j{:=}u] = t$.

Proof. By Lemma 3.7 if $\mathsf{level}(t) < j$ then $b_j \# fv(t)$. The result follows from the definition of $[b_j{:=}u]$. □

Lemma 3.9. *If* $s \rightsquigarrow s'$ *then* $\mathsf{level}(s') \leq \mathsf{level}(s)$.

Proof. By a series of easy calculations on the rules in Figure 6 and an inductive argument. □

Theorem 3.10. *If* $i < j$ *then*

$$s[a_i{:=}t][b_j{:=}u] = s[b_j{:=}u][a_i{:=}t[b_j{:=}u]].$$

(Note the lack of capture-avoidance condition; this is because $i < j$.)

Proof. By induction on i and then on the structure of s.

- Suppose $s[a_i{:=}t] = s[a_i{\mapsto}t]$. We then have

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

  This covers the cases $s = c_k$ with $k > i$, $s = s_1 s_2$ with $\mathsf{level}(s_1, s_2, t) > i$, $s = s_1[c_k{\mapsto}s_2]$ with $k \geq i$, $s = \lambda c_k.s_1$ with $k > i$, or $k = i$ without $c_k \# fv(t)$.

- Suppose $k \leq i < j$. Note that by our permutative convention, c_k is distinct from a_i. Then:

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

- The cases of $a_i[a_i{:=}t][b_j{:=}u]$ and $b_j[a_i{:=}t][b_j{:=}u]$ are easy.

- Suppose that $\mathsf{level}(s, s', t) \leq i < j$. By Lemma 3.6 we have $(ss')[a_i{\mapsto}t] \rightsquigarrow^* (ss')[a_i{:=}t]$. By Lemma 3.9 we have

$$\mathsf{level}((ss')[a_i{:=}t]) \leq \mathsf{level}((ss')[a_i{\mapsto}t]) = \mathsf{level}(s, s', t, a_i) < j.$$

  Finally by Lemma 3.8 we have

$$\begin{aligned} (ss')[a_i{:=}t][b_j{:=}u] &= (ss')[a_i{:=}t] \\ (ss')[b_j{:=}u][a_i{:=}t[b_j{:=}u]] &= (ss')[a_i{:=}t] \end{aligned}$$

- Suppose $k < i$. Then

$$\begin{aligned} s[c_k{\mapsto}s'][a_i{:=}t][b_j{:=}u] &= s[a_i{:=}t][c_k{:=}s'[a_i{:=}t]][b_j{:=}u] \\ &\overset{\text{ind. hyp.}}{=} s[a_i{:=}t][b_j{:=}u][c_k{:=}s'[a_i{:=}t][b_j{:=}u]] \\ &\overset{\text{ind. hyp.}}{=} s[b_j{:=}u][a_i{:=}t[b_j{:=}u]][c_k{:=}s'[b_j{:=}u][a_i{:=}t[b_j{:=}u]]] \\ s[c_k{\mapsto}s'][b_j{:=}u][a_i{:=}t[b_j{:=}u]] &= s[b_j{:=}u][c_k{:=}s'[b_j{:=}u]][a_i{:=}t[b_j{:=}u]] \\ &\overset{\text{ind. hyp.}}{=} s[b_j{:=}u][a_i{:=}t[b_j{:=}u]][c_k{:=}s'[b_j{:=}u][a_i{:=}t[b_j{:=}u]]] \end{aligned}$$

- Suppose $k < i$. Then

$$\begin{aligned} (\lambda c_k.s)[a_i{:=}t][b_j{:=}u] &= \lambda c_k.(s[a_i{:=}t][b_j{:=}u]) \\ &\overset{\text{ind. hyp.}}{=} \lambda c_k.(s[b_j{:=}u][a_i{:=}t[b_j{:=}u]]) \\ (\lambda c_k.s)[b_j{:=}u][a_i{:=}t[b_j{:=}u]] &= \lambda c_k.(s[b_j{:=}u][a_i{:=}t[b_j{:=}u]]) \end{aligned}$$

- Suppose (renaming c_i where possible) that $c_i \# fv(t)$. Then

$$\begin{aligned} (\lambda c_i.s)[a_i{:=}t][b_j{:=}u] &= (\lambda c_i.s[a_i{:=}t][b_j{:=}u]) \\ &\overset{\text{ind. hyp.}}{=} (\lambda c_i.s[b_j{:=}u][a_i{:=}t[b_j{:=}u]]) \\ (\lambda c_i.s)[b_j{:=}u][a_i{:=}t[b_j{:=}u]] &= (\lambda c_i.s[b_j{:=}u][a_i{:=}t[b_j{:=}u]]) \end{aligned}$$

□

We need a technical lemma for Theorem 3.12:

Lemma 3.11. *If s is* (**sigma**)*-normal and* $\mathsf{level}(s) \leq i$ *then there is no substitution of level i in s.*

Proof. By a routine induction on s. □

Theorem 3.12. *Suppose that* $\mathsf{level}(s,t,u) \leq i$ *and* $a_i \# fv(u)$ *(which in view of the condition on levels, means just $a_i \notin fv(u)$). Suppose also that s, t, and u are* (**sigma**)*-normal. Then*

$$s[a_i{:=}t][b_i{:=}u] = s[b_i{:=}u][a_i{:=}t[b_i{:=}u]].$$

Proof. By induction on the structure of s.

- Suppose $k \leq i$. Recall that by our permutative convention c_k is distinct from a_i and b_i. Then:

$$\begin{aligned} c_k[a_i{:=}t][b_i{:=}u] &= c_k[b_i{:=}u] \\ &= c_k \\ c_k[b_i{:=}u][a_i{:=}t[b_i{:=}u]] &= c_k[a_i{:=}t[b_i{:=}u]] \\ &= c_k. \end{aligned}$$

- The cases of $a_i[a_i{:=}t][b_i{:=}u]$ and $b_i[a_i{:=}t][b_i{:=}u]$ are easy.
- We do not need to consider the case $c_k[a_i{:=}t][b_i{:=}u]$ for $k > i$, because then $\mathsf{level}(c_k) > i$.
-

$$\begin{aligned} (ss')[a_i{:=}t][b_i{:=}u] &= ((s[a_i{:=}t])(s'[a_i{:=}t]))[b_i{:=}u] \\ &= (s[a_i{:=}t][b_i{:=}u])(s'[a_i{:=}t][b_i{:=}u]) \\ &\overset{\text{ind. hyp.}}{=} (s[b_i{:=}u][a_i{:=}t[b_i{:=}u]])(s'[b_i{:=}u][a_i{:=}t[b_i{:=}u]]) \\ (ss')[b_i{:=}u][a_i{:=}t[b_i{:=}u]] &= ((s[b_i{:=}u])(s'[b_i{:=}u]))[a_i{:=}t[b_i{:=}u]] \\ &= (s[b_i{:=}u][a_i{:=}t[b_i{:=}u]])(s'[b_i{:=}u][a_i{:=}t[b_i{:=}u]]) \end{aligned}$$

- For the case of $s[c_k{\mapsto}s']$, assumed to be be (**sigma**)-normal, we know from the assumption $\mathsf{level}(s[c_k{\mapsto}s']) \leq i$ and Lemma 3.11 that $k < i$. Hence,

$$\begin{aligned} s[c_k{\mapsto}s'][a_i{:=}t][b_i{:=}u] &= s[a_i{:=}t][c_k{:=}s'[a_i{:=}t]][b_i{:=}u] \\ &\overset{\text{Thm.3.10}}{=} s[a_i{:=}t][b_i{:=}u][c_k{:=}s'[a_i{:=}t][b_i{:=}u]] \\ &\overset{\text{ind. hyp.}}{=} s[b_i{:=}u][a_i{:=}t[b_i{:=}u]][c_k{:=}s'[b_i{:=}u][a_i{:=}t[b_i{:=}u]]] \\ s[c_k{\mapsto}s'][b_i{:=}u][a_i{:=}t[b_i{:=}u]] &= s[b_i{:=}u][c_k{:=}s'[b_i{:=}u]][a_i{:=}t[b_i{:=}u]] \\ &\overset{\text{Thm.3.10}}{=} s[b_i{:=}u][a_i{:=}t[b_i{:=}u]][c_k{:=}s'[b_i{:=}u][a_i{:=}t[b_i{:=}u]]] \end{aligned}$$

- Suppose $k \geq i$. Since $\mathsf{level}(\lambda c_k.s, t, u) \leq i$ we can rename c_k so that $c_k \# fv(t) \cup fv(u)$. Then

$$\begin{aligned} (\lambda c_k.s)[a_i{:=}t][b_i{:=}u] &= (\lambda c_k.(s[a_i{:=}t]))[b_i{:=}u] \\ &= (\lambda c_k.(s[a_i{:=}t][b_i{:=}u])) \\ &\overset{\text{ind. hyp.}}{=} (\lambda c_k.(s[b_i{:=}u][a_i{:=}t[b_i{:=}u]])) \\ (\lambda c_k.s)[b_i{:=}u][a_i{:=}t[b_i{:=}u]] &= (\lambda c_k.(s[b_i{:=}u]))[a_i{:=}t[b_i{:=}u]] \\ &= (\lambda c_k.(s[b_i{:=}u][a_i{:=}t[b_i{:=}u]])) \end{aligned}$$

  $k \geq i$ so $\mathsf{level}(s) \leq \mathsf{level}(\lambda c_k.s)$, and this is why we can use the inductive hypothesis.

- The case of $(\lambda c_k.s)[a_i{:=}t][b_i{:=}u]$ where $k < i$ is similar, but easier.

□

3.3. Calculating (**sigma**)*-normal forms:* s^*

We now have everything we need to calculate (**sigma**)-normal forms:

Definition 3.13. *Define the catchily-named 'the star' function* s^* *inductively by:*

$$\begin{aligned} a_i^* &= a_i \\ (\lambda a_i.s)^* &= \lambda a_i.(s^*) \\ (s[a_i{\mapsto}t])^* &= s^*[a_i{:=}t^*] \\ (st)^* &= (s^*)(t^*) \end{aligned}$$

Remark 3.14. A precedent for Definition 3.13 has been observed by an anonymous referee: A similar function, also called -*, is used by Takahashi in a compact but extensive analysis of properties of reduction of the untyped λ-calculus [42]. Takahashi works with the (plain, normal) untyped λ-calculus; this does not have an explicit substitution, but unreduced β-reducts are used to emulate explicit substitution.

We were *led* to consider s^*, for reasons which we describe in Remark 4.2 and a surrounding discussion; although we do not prove or attempt to prove that our proof-method is the only way of proving confluence of the LamCC, it is the one that we could find.

We suspect that the underlying reason why Takahashi's use of -* leads to (in her words) a particularly direct and short proof, must have to do with an underlying mechanism also exploited by the proof-method we use here, even though for the case of the untyped λ-calculus other methods also work. Perhaps understanding this more deeply is future research.

Lemma 3.15. *If* s *and* t *are* (**sigma**)*-normal then* $s[a_i{:=}t]$ *is* (**sigma**)*-normal.*

Proof. By induction on i and then s. □

Theorem 3.16. s^* *is a* (**sigma**)*-normal form of* s*.*

Proof. There are two results to prove. The first is $s \overset{(\mathbf{sigma})}{\rightsquigarrow^*} s^*$, which is proved by an easy induction on the definition of s^* (the case of $(^*\sigma)$ uses Lemma 3.6). The second is that s^* is a $(\mathbf{sigma})$-normal form, which is proved by a routine induction on s, using Lemma 3.15. □

We conclude with a useful lemma:

Lemma 3.17. 1. $(a_i[a_i{\mapsto}t])^* = t^*$.
2. $(c_k[a_i{\mapsto}t])^* = c_k$ *where* $k \leq i$.
3. $((ss')[a_i{\mapsto}t])^* = ((s[a_i{\mapsto}t])(s'[a_i{\mapsto}t]))^*$ *where* $\mathsf{level}(s, s', t) \leq i$.
4. $(s[a_i{\mapsto}t][b_j{\mapsto}u])^* = (s[b_j{\mapsto}u][a_i{\mapsto}t[b_j{\mapsto}u]])^*$ *if* $i < j$.
5. $((\lambda a_i.s)[b_j{\mapsto}u])^* = (\lambda a_i.(s[b_j{\mapsto}u]))^*$ *if* $i < j$.
6. $((\lambda a_i.s)[c_i{\mapsto}u])^* = (\lambda a_i.(s[c_i{\mapsto}u]))^*$ *if (renaming* a_i *where possible)* $a_i \# fv(u)$.

Proof.

1. $(a_i[a_i{\mapsto}t])^* = a_i[a_i{:=}t^*] = t^*$.
2. Recall that we assume $k \leq i$.
$$(c_k[a_i{\mapsto}t])^* = c_k[a_i{:=}t^*] = c_k = c_k^*.$$
3. Recall that we assume that $\mathsf{level}(s, s', t) \leq i$.
$$((ss')[a_i{\mapsto}t])^* = (s^*[a_i{:=}t^*])(s'^*[a_i{:=}t^*])$$
$$((s[a_i{\mapsto}t])(s'[a_i{\mapsto}t]))^* = (s^*[a_i{:=}t^*])(s'^*[a_i{:=}t^*]).$$
4. Recall that we assume that $i < j$. Using Theorem 3.10
$$\begin{aligned}(s[a_i{\mapsto}t][b_j{\mapsto}u])^* &= s^*[a_i{:=}t^*][b_j{:=}u^*] \\ &= s^*[b_j{:=}u^*][a_i{:=}t^*[b_j{:=}u^*]]\end{aligned}$$
$$(s[b_j{\mapsto}u][a_i{\mapsto}t[b_j{\mapsto}u]])^* = s^*[b_j{:=}u^*][a_i{:=}t^*[b_j{:=}u^*]]$$
5. Recall that we assume that $i < j$.
$$\begin{aligned}((\lambda a_i.s)[b_j{\mapsto}u])^* &= (\lambda a_i.s^*)[b_j{:=}u^*] \\ &= \lambda a_i.(s^*[b_j{:=}u^*])\end{aligned}$$
$$(\lambda a_i.(s[b_j{\mapsto}u]))^* = \lambda a_i.(s^*[b_j{:=}u^*])$$
6.
$$\begin{aligned}((\lambda a_i.s)[c_k{\mapsto}u])^* &= (\lambda a_i.s^*)[c_k{:=}u^*] \\ &= \lambda a_i.(s^*[c_k{:=}u^*])\end{aligned}$$
$$(\lambda a_i.(s[c_k{\mapsto}u]))^* = \lambda a_i.(s^*[c_k{:=}u^*])$$

□

4. Confluence

A few words of overview. The LamCC is a set of terms (Definition 2.2) and a reduction relation $\rightsquigarrow$ (Definition 2.12). The main result of this section is confluence:

Theorem 4.1. $\rightsquigarrow$ *is confluent. That is, if* $s \rightsquigarrow^* t_1$ *and* $s \rightsquigarrow^* t_2$ *then there is some* u *such that* $t_1 \rightsquigarrow^* u$ *and* $t_2 \rightsquigarrow^* u$.

The proof of Theorem 4.1 occupies this section. First, we comment on the overall design of the proof, which is slightly unusual and, in some respects original.

Remark 4.2. Roughly speaking there are two standard ways to prove confluence:

- *Method 1.* Define a so-called *parallel reduction relation* $\Rightarrow$. such that $\Rightarrow \subseteq \rightsquigarrow^*$ and $\Rightarrow^* = \rightsquigarrow^*$. It then suffices to show that if $s \Rightarrow t_1$ and $s \Rightarrow t_2$ then there is some u such that $t_1 \Rightarrow u$ and $t_2 \Rightarrow u$.
- *Method 2.* Define for each term s a *canonical form* $s^{\downarrow}$ such that $s \rightsquigarrow^* s^{\downarrow}$ and then prove that for all possible reductions $s \rightsquigarrow s'$ it is the case that $s' \rightsquigarrow^* s^{\downarrow}$.

Both of these methods are standard [43]. But which to use for the LamCC? It seems that reductions using (β) admit method 1 above, whereas $(\mathbf{sigma})$-rules do not (see Remark 4.3).

Therefore, to prove confluence of the LamCC, we split the reduction relation into two sets: $(\mathbf{sigma})$ (see Section 3, Definition 3.1 above) and $(\mathbf{beta})$ (see Definition 4.5 below). We prove independent confluence results for $(\mathbf{sigma})$ and $(\mathbf{beta})$, using method 1 for $(\mathbf{beta})$ and using method 2 for $(\mathbf{sigma})$ (the normal form is s^* from Definition 3.13). Then, we show that $(\mathbf{sigma})$ and $(\mathbf{beta})$ commute. This suffices to prove confluence of the entire rewrite system [43, Exercise 1.3.4].

Remark 4.3. We briefly describe why method 1 from Remark 4.2 is not appropriate for proving confluence of $(\mathbf{sigma})$-reduction. Consider the following divergence, where $j > i$:

$$(s's)[a_i \mapsto t][b_j \mapsto u] \overset{(\sigma\mathbf{p})}{\rightsquigarrow} ((s'[a_i \mapsto t])(s[a_i \mapsto t]))[b_j \mapsto u]$$
$$(s's)[a_i \mapsto t][b_j \mapsto u] \overset{(\sigma\sigma)}{\rightsquigarrow} (s's)[b_j \mapsto u][a_i \mapsto t[b_j \mapsto u]]$$

We close this as follows:

$$\begin{array}{l} ((s'[a_i \mapsto t])(s[a_i \mapsto t]))[b_j \mapsto u] \overset{(\sigma\mathbf{p})}{\rightsquigarrow} (s'[a_i \mapsto t][b_j \mapsto u])(s[a_i \mapsto t][b_j \mapsto u]) \\ \qquad\qquad \overset{(\sigma\sigma)}{\rightsquigarrow^*} (s'[b_j \mapsto u][a_i \mapsto t[b_j \mapsto u]])(s[b_j \mapsto u][a_i \mapsto t[b_j \mapsto u]]) \\ (s's)[b_j \mapsto u][a_i \mapsto t[b_j \mapsto u]] \overset{(\sigma\mathbf{p})}{\rightsquigarrow} ((s'[b_j \mapsto u])(s[b_j \mapsto u]))[a_i \mapsto t[b_j \mapsto u]] \\ \qquad\qquad \overset{(\sigma\mathbf{p})}{\rightsquigarrow} (s'[b_j \mapsto u][a_i \mapsto t[b_j \mapsto u]])(s[b_j \mapsto u][a_i \mapsto t[b_j \mapsto u]]) \end{array}$$

In a parallel moves proof-method, the last two instances of $(\sigma\mathbf{p})$ are problematic because they both occur at the same level in the term (at the top).

Remark 4.4. One way to prove confluence is to split the reduction system into smaller modules, to prove confluence of each module separately, and then to put them together proving that confluence is preserved [43, Exercise 1.3.4]. Our confluence proof is modular; this is not original. However, our method of splitting reductions into (β)- and $(\mathbf{sigma})$-rules so as to apply methods 1 and 2 separately, and the rather subtle distribution of the rules into $(\mathbf{sigma})$ and $(\mathbf{beta})$ is original to this research as far as we know.

Recall from Definition 3.1 that $(\mathbf{sigma})$ is the set of rules defined by

$$(\mathbf{sigma}) = \{(\sigma\mathbf{a}), (\sigma fv), (\sigma\mathbf{p}), (\sigma\sigma), (\sigma\lambda), (\sigma\lambda')\}.$$

It is convenient to define:

Definition 4.5. *Let* $(\mathbf{beta})$ *be the set*

$$(\mathbf{beta}) = \{(\beta), (\sigma\lambda), (\sigma\lambda'), (\sigma fv)\}.$$

Note that $(\mathbf{sigma}) \cup (\mathbf{beta})$ is equal to the set of all reduction rules of the LamCC.

Note also that $(\mathbf{sigma}) \cap (\mathbf{beta})$ is non-empty. We mention the technical reasons for this in Remark 4.16 just after Lemma 4.15 — it seems to be vital for the proofs to work. Understanding the deeper mathematical reasons for this, if any, is future work.

4.1. Confluence of (**sigma**)

Given the constructions in Subsections 3.2 and 3.3, confluence of (**sigma**) is relatively easy:

Lemma 4.6. $s^*[a_i{\mapsto}t^*] \overset{(\mathbf{sigma})}{\rightsquigarrow^*} (s[a_i{\mapsto}t])^*$.

Proof. By definition $(s[a_i{\mapsto}t])^* = s^*[a_i{:=}t^*]$. We use Lemma 3.6. □

Lemma 4.7. *If* $s \overset{(\mathbf{sigma})}{\rightsquigarrow} s'$ *then* $s' \overset{(\mathbf{sigma})}{\rightsquigarrow^*} s^*$.

Proof. We work by induction on the derivation of $s \overset{(\mathbf{sigma})}{\rightsquigarrow} s'$, see Figure 6.

- (**Rapp**) Suppose $s \overset{(\mathbf{sigma})}{\rightsquigarrow} s'$ so that $st \overset{(\mathbf{sigma})}{\rightsquigarrow} s't$.
  By inductive hypothesis $s' \overset{(\mathbf{sigma})}{\rightsquigarrow^*} s^*$ and by Theorem 3.16 $t \overset{(\mathbf{sigma})}{\rightsquigarrow^*} t^*$, so also $s't \overset{(\mathbf{sigma})}{\rightsquigarrow^*} s^*t^* = (st)^*$. The case of (**Rapp**′) is similar.
- (**R**λ) Suppose $s \overset{(\mathbf{sigma})}{\rightsquigarrow} s'$ so that $\lambda a_i.s \overset{(\mathbf{sigma})}{\rightsquigarrow} \lambda a_i.s'$. By inductive hypothesis $s' \overset{(\mathbf{sigma})}{\rightsquigarrow^*} s^*$, and so $\lambda a_i.s' \overset{(\mathbf{sigma})}{\rightsquigarrow^*} \lambda a_i.(s^*) = (\lambda a_i.s)^*$.
- (**R**σ) Suppose $s \overset{(\mathbf{sigma})}{\rightsquigarrow} s'$ so that $s[a_i{\mapsto}t] \overset{(\mathbf{sigma})}{\rightsquigarrow} s'[a_i{\mapsto}t]$.
  By inductive hypothesis $s' \overset{(\mathbf{sigma})}{\rightsquigarrow^*} s^*$ and by Theorem 3.16 $t \overset{(\mathbf{sigma})}{\rightsquigarrow^*} t^*$, so also $s'[a_i{\mapsto}t] \overset{(\mathbf{sigma})}{\rightsquigarrow^*} s^*[a_i{\mapsto}t^*]$.
  By Lemma 4.6 $s^*[a_i{\mapsto}t^*] \overset{(\mathbf{sigma})}{\rightsquigarrow^*} (s[a_i{\mapsto}t])^*$.
  The case of (**R**σ′) is similar.
- Now suppose that $s \overset{(\mathbf{sigma})}{\rightsquigarrow} s'$ is derived using one of the rules (σ**a**), (σ**c**), (σ**p**), ($\sigma\sigma$), ($\sigma\lambda$), or ($\sigma\lambda$′). In these cases we use Theorem 3.16 and the relevant part of Lemma 3.17.

□

Theorem 4.8. $\overset{(\mathbf{sigma})}{\rightsquigarrow}$ *is confluent.*

Proof. By an easy inductive argument using Lemma 4.7. □

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

Definition 4.9. *Inductively define the* **parallel reduction relation** $\Longrightarrow$ *(for* (**beta**)*) by the rules in Figure 8.*

In rules (**P**$\sigma\epsilon$) and (**Papp**ϵ), $s't' \overset{R_\epsilon}{\rightsquigarrow} u$ and $s'[a_i{\mapsto}t'] \overset{R_\epsilon}{\rightsquigarrow} u$ indicate a *top-level rewrite* with any $R \in$ (**beta**) — that is, $s't' \overset{R}{\rightsquigarrow} u$ and $s'[a_i{\mapsto}t'] \overset{R}{\rightsquigarrow} u$ respectively are derivable *without* using (**Rapp**), (**Rapp**′), (**R**λ), (**R**σ), or (**R**σ′).

Lemma 4.10.
1. $s \Longrightarrow s$.
2. *If* $s \Longrightarrow s'$ *then* $s \overset{(\mathbf{beta})}{\rightsquigarrow^*} s'$.
3. *If* $s \overset{(\mathbf{beta})}{\rightsquigarrow} s'$ *then* $s \Longrightarrow s'$.

As a corollary, $s \Longrightarrow^* s'$ *if and only if* $s \overset{(\mathbf{beta})}{\rightsquigarrow^*} s'$.

Proof. All parts are by routine inductions:

1. By induction on the syntax of s.

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

$$\frac{s \Longrightarrow s' \quad t \Longrightarrow t'}{st \Longrightarrow s't'}\ (\mathbf{Papp}) \qquad \frac{s \Longrightarrow s'}{\lambda a_i.s \Longrightarrow \lambda a_i.s'}\ (\mathbf{P}\lambda)$$

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

$$\frac{s \Longrightarrow s' \quad t \Longrightarrow t' \quad s't' \overset{R_\epsilon}{\rightsquigarrow} u}{st \Longrightarrow u}\ (\mathbf{Papp}\epsilon) \qquad (R \in (\mathbf{beta}))$$

Figure 8: Parallel reduction relation for the LamCC

2. By induction on the derivation of $s \Longrightarrow s'$.
3. By induction on the derivation of $s \overset{(\mathbf{beta})}{\rightsquigarrow} s'$.

□

Lemma 4.11. *If $s \rightsquigarrow s'$ then $fv(s') \subseteq fv(s)$. As a corollary, if $s \rightsquigarrow^* s'$ then $fv(s') \subseteq fv(s)$.*

Proof. We work by induction on the derivation of $s \rightsquigarrow s'$. The base cases are:

- $fv(s[a_i \mapsto t]) = (fv(s) \setminus \{a_i\} \cup fv(t)) = fv((\lambda a_i.s)t)$.
- $fv(a_i[a_i \mapsto t]) = fv(t)$ and $fv(t)$ is a subset of itself.
- Suppose that $a_i \# fv(s)$. By definition

  $$fv(s[a_i \mapsto t]) = (fv(s) \setminus \{a_i\}) \cup fv(t).$$

  By Definition 2.7 since $a_i \# fv(s)$ also $a_i \notin fv(s)$ so $fv(s) \setminus \{a_i\} = fv(s)$ and $fv(s)$ is a subset of $fv(s[a_i \mapsto t])$.
- Suppose that $\mathsf{level}(s, s', t) \leq i$. Then

  $$\begin{aligned} fv((ss')[a_i \mapsto t]) &= ((fv(s) \cup fv(s')) \setminus \{a_i\}) \cup fv(t) \\ fv(s[a_i \mapsto t]s'[a_i \mapsto t]) &= ((fv(s) \setminus \{a_i\}) \cup fv(t)) \cup \\ &\quad ((fv(s') \setminus \{a_i\}) \cup fv(t)). \end{aligned}$$

  The subset inclusion follows by easy calculations on sets.

Other cases are no harder. The inductive argument is straightforward, relying on Lemma 3.9. The corollary is immediate. □

Corollary 4.12. *If $s \Longrightarrow s'$ then $fv(s') \subseteq fv(s)$ and $\mathsf{level}(s') \subseteq \mathsf{level}(s)$.*

Proof. From Lemma 4.10 and Lemma 4.11. □

Lemma 4.13. *$\Longrightarrow$ 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

$$\forall s''.s \Longrightarrow s'' \Rightarrow \exists s'''.(s' \Longrightarrow s''' \wedge s'' \Longrightarrow s''').$$

For simplicity we just consider possible pairs of rules which could derive $s \Longrightarrow s_1$ and $s \Longrightarrow s_2$.

- $(\mathbf{Pa})$ and $(\mathbf{Pa})$. There is nothing to prove.
- $(\mathbf{P}\sigma)$ and $(\mathbf{P}\sigma)$.
  $s \Longrightarrow s'$ and $t \Longrightarrow t'$ and also $s \Longrightarrow s''$ and $t \Longrightarrow t''$ so that by $(\mathbf{P}\sigma)$ and $(\mathbf{P}\sigma)$
  $$s'[a_i \mapsto t'] \Longleftarrow s[a_i \mapsto t] \Longrightarrow s''[a_i \mapsto t''].$$
  By inductive hypothesis there are s''' and t''' such that
  $$s' \Longrightarrow s''' \Longleftarrow s'' \quad \text{and} \quad t' \Longrightarrow t''' \Longleftarrow t''.$$
  It follows that
  $$s'[a_i \mapsto t'] \Longrightarrow s'''[a_i \mapsto t'''] \Longleftarrow s''[a_i \mapsto t''].$$
- $(\mathbf{P}\sigma)$ and $(\mathbf{P}\sigma\epsilon)$ for $(\sigma\lambda)$.
  Suppose $s \Longrightarrow s'$ and $t \Longrightarrow t'$ and also $s \Longrightarrow s''$ and $t \Longrightarrow t''$. Suppose also that $i < j$ so that by $(\mathbf{P}\sigma)$ and $(\mathbf{P}\sigma\epsilon)$ for $(\sigma\lambda)$
  $$(\lambda a_i.s')[b_j \mapsto t'] \Longleftarrow (\lambda a_i.s)[b_j \mapsto t] \Longrightarrow \lambda a_i.(s''[b_j \mapsto t'']).$$
  By inductive hypothesis there are s''' and t''' such that
  $$s' \Longrightarrow s''' \Longleftarrow s'' \quad \text{and} \quad t' \Longrightarrow t''' \Longleftarrow t''.$$
  Using $(\mathbf{P}\sigma\epsilon)$ for $(\sigma\lambda)$ and $(\mathbf{P}\sigma)$
  $$(\lambda a_i.s')[b_j \mapsto t'] \Longrightarrow \lambda a_i.(s'''[b_j \mapsto t''']) \Longleftarrow \lambda a_i.(s''[b_j \mapsto t'']).$$
- The case of $(\mathbf{P}\sigma\epsilon)$ for $(\sigma\lambda)$ and $(\mathbf{P}\sigma)$ is similar.
- $(\mathbf{P}\sigma)$ and $(\mathbf{P}\sigma\epsilon)$ for $(\sigma\lambda')$.
  Suppose $s \Longrightarrow s'$ and $u \Longrightarrow u'$ and also $s \Longrightarrow s''$ and $u \Longrightarrow u''$. Suppose also that (renaming a_i where necessary) $a_i \# fv(u'')$ 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
  $$s' \Longrightarrow s''' \Longleftarrow s'' \quad \text{and} \quad u' \Longrightarrow u''' \Longleftarrow u''.$$
  By Corollary 4.12 $a_i \# u'''$. Using $(\mathbf{P}\sigma\epsilon)$ for $(\sigma\lambda')$ and $(\mathbf{P}\sigma)$
  $$(\lambda a_i.s')[c_i \mapsto u'] \Longrightarrow \lambda a_i.(s'''[c_i \mapsto u''']) \Longleftarrow \lambda a_i.(s''[c_i \mapsto u'']).$$
- $(\mathbf{P}\lambda)$ with $(\mathbf{P}\lambda)$.
  Suppose $s' \Longleftarrow s \Longrightarrow s''$ so that $\lambda a_i.s' \Longleftarrow \lambda a_i.s \Longrightarrow \lambda a_i.s''$. By inductive hypothesis there is some s''' such that $s' \Longrightarrow s''' \Longleftarrow s''$. By $(\mathbf{P}\lambda)$ also
  $$\lambda a_i.s' \Longrightarrow \lambda a_i.s''' \Longleftarrow \lambda a_i.s''.$$

Other cases are similar and no harder. □

Theorem 4.14. $\overset{(\mathbf{beta})}{\rightsquigarrow}$ *is confluent.*

Proof. By Lemma 4.10 and Lemma 4.10 and a standard argument [3]. □

4.3. Combining **(sigma)** *and* **(beta)**

Lemma 4.15. *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 $\mathsf{level}(s,t,u) \leq j$:

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

  This can be closed by:

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

- (β) has a divergence with $(\sigma\mathbf{p})$ in the case that $i = j$ and $\mathsf{level}(s,t,u) \leq i$:

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

  We suppose, renaming a_i if necessary, that $a_i \# u$.

  By Corollary 4.12 $\mathsf{level}(s',t',u') \leq i$. By Lemma 3.6 and Theorem 3.16,

$$s'[a_i{\mapsto}t'][b_i{\mapsto}u'] \overset{(\mathbf{sigma})}{\rightsquigarrow^*} (s')^*[a_i{:=}t'][b_i{:=}u']$$

  and by Theorem 3.12 this is equal to $(s')^*[b_i{:=}u'][a_i{:=}t'[b_i{:=}u']]$.

  On the other hand (also using Lemma 3.6 and Theorem 3.16):

$$(\lambda a_i.s)[b_i{\mapsto}u](t[b_i{\mapsto}u]) \Longrightarrow s'[b_i{\mapsto}u'][a_i{\mapsto}t'[b_i{\mapsto}u']] \overset{(\mathbf{sigma})}{\rightsquigarrow^*} (s')^*[b_i{:=}u'][a_i{:=}t'[b_i{:=}u']]$$

  which closes the divergence above.

- $(\sigma\sigma)$ has a divergence with $(\sigma\lambda)$. Suppose that $k < i < j$:

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

  This can be closed by:

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

- $(\sigma\sigma)$ has a divergence with $(\sigma\lambda')$. Suppose that $i < j$ and (renaming c_i where possible) $c_i \# fv(t)$:

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

  We know that $b_j \# fv(t)$ because $c_i \# fv(t)$ and $i < j$. We then deduce $b_j \# fv(t')$ using Corollary 4.12. We use this to justify the $\Longrightarrow$-rewrite which uses (σfv) in a moment.

  This can be closed by:

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

□

Remark 4.16. We promised to explain why $(\mathbf{sigma}) \cap (\mathbf{beta}) \neq \emptyset$. We can now do so by reference to the details of the proof of Lemma 4.15.

- $(\sigma\lambda) \in (\mathbf{beta})$ and $(\sigma\lambda') \in (\mathbf{beta})$ because otherwise the two cases of $(\sigma\mathbf{p})$ and (β) above would not work.
- $(\sigma fv) \in (\mathbf{beta})$ because otherwise the case of $(\sigma\sigma)$ with $(\sigma\lambda')$ would not work.

We can easily generalise this lemma to several σ-steps:

Lemma 4.17. *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 length of the path $s \overset{(\mathbf{sigma})}{\rightsquigarrow^*} s''$. The case of the empty path is trivial. Otherwise we have $s \overset{(\mathbf{sigma})}{\rightsquigarrow^*} t \overset{(\mathbf{sigma})}{\rightsquigarrow} s''$ and the induction hypothesis provides t' such that $s' \overset{(\mathbf{sigma})}{\rightsquigarrow^*} t'$ and $t \Longrightarrow t'$. Lemma 4.15 then provides s''' such that $t' \rightsquigarrow^* s'''$ and $s'' \Longrightarrow s'''$. □

We now generalise this lemma even further:

Lemma 4.18. *If $s \Longrightarrow s'$ (respectively $s \overset{(\mathbf{sigma})}{\rightsquigarrow^*} s'$) and $s \rightsquigarrow^* s''$, then there is some s''' such that $s' \rightsquigarrow^* s'''$ and $s'' \Longrightarrow s'''$ (respectively $s \overset{(\mathbf{sigma})}{\rightsquigarrow^*} s'$).*

Proof. Again, we work by induction on the length of the path $s \rightsquigarrow^* s''$. The case of the empty path is trivial. Otherwise we have $s \rightsquigarrow^* t \overset{(\mathbf{sigma})}{\rightsquigarrow} s''$ or $s \rightsquigarrow^* t \overset{(\beta)}{\rightsquigarrow} s''$. In both cases, the induction hypothesis provides t' such that $s' \rightsquigarrow^* t'$ and $t \Longrightarrow t'$ (respectively $s \overset{(\mathbf{sigma})}{\rightsquigarrow^*} s'$). In the former case, Lemma 4.15 (respectively Theorem 4.8) provides s''' such that $t' \rightsquigarrow^* s'''$ and $s'' \Longrightarrow s'''$ (respectively $s'' \overset{(\mathbf{sigma})}{\rightsquigarrow^*} s'''$). In the latter case, Lemma 4.13 (respectively Lemma 4.17) provides s''' such that $t' \rightsquigarrow^* s'''$ and $s'' \Longrightarrow s'''$ (respectively $s'' \overset{(\mathbf{sigma})}{\rightsquigarrow^*} s'''$). □

We can now prove Theorem 4.1:

Proof. Suppose that $s \rightsquigarrow^* t$ and $s \rightsquigarrow^* t'$. We prove that there exists s' such that $t \rightsquigarrow^* s'$ and $t' \rightsquigarrow^* s'$, by induction on the length of the reduction path $s \rightsquigarrow^* t$. In the case of the empty path,

$t = s \rightsquigarrow^* t'$. Otherwise, we have either $s \rightsquigarrow^* s'' \overset{\textbf{(sigma)}}{\rightsquigarrow} t$ or $s \rightsquigarrow^* s'' \overset{\textbf{(beta)}}{\rightsquigarrow} t$. In both cases, the induction hypothesis provides t'' such that $s'' \rightsquigarrow^* t''$ and $t' \rightsquigarrow^* t''$, and then Lemma 4.18 provides s''' such that $t \rightsquigarrow^* s'''$, and $t'' \Longrightarrow s'''$ or $t'' \overset{\textbf{(sigma)}}{\rightsquigarrow^*} s'''$, and in both cases we have $t'' \rightsquigarrow^* s'''$ as required. □

5. The untyped lambda-calculus

We show how to translate the untyped λ-calculus into the LamCC.

For convenience, we identify the variables of level 1 in the LamCC as the variables used in λ-calculus; when we translate the λ-calculus into the LamCC, we will use this identification.

Terms of the untyped λ-calculus are given by

$$e ::= x \mid ee \mid \lambda x.e.$$

λ binds x in $\lambda x.e$. This is standard [3].

We define a **free variables of** $fv(t)$ function in the usual way:

$$fv(x) = \{x\} \qquad fv(ee') = fv(e) \cup fv(e') \qquad fv(\lambda x.e) = fv(e) \setminus \{x\}.$$

Call e **open** when there exists some x such that $x \in fv(e)$.

Define a **capture-avoiding substitution action** inductively by:

$$x[x{:=}e] = x \qquad y[x{:=}e] = y \qquad (e_1e_2)[x{:=}e] = (e_1[x{:=}e])(e_2[x{:=}e])$$
$$(\lambda x'.e')[x{:=}e] = \lambda x'.(e'[x{:=}e]) \qquad (x' \notin fv(e))$$

Here we may assume x' does not occur in e because we have equated syntax up to binding by λ.

Define a **reduction relation** inductively by:

$$\frac{}{(\lambda x.e)e' \to e[x{:=}e']} \qquad \frac{e_1 \to e_1' \quad e_2 \to e_2'}{e_1e_2 \to e_1'e_2'} \qquad \frac{e \to e'}{\lambda x.e \to \lambda x.e'}$$

We call e a **normal form** or **value** when there is no e' such that $e \to e'$. Note that normal forms may be open.

A translation into the LamCC is given by:

$$[\![x]\!] = x \qquad [\![ee']\!] = [\![e]\!][\![e']\!] \qquad [\![\lambda x.e]\!] = \lambda x.[\![e]\!]$$

The following results are very easy to prove:

Lemma 5.1. $fv(e) = fv([\![e]\!])$.

Proof. We consider the clauses of the definition of fv above, and of the definition of fv from Figure 2, and we see that they coincide in the special case that only variables of level 1 appear. □

Lemma 5.2. $[\![e[x{:=}e']]\!] = [\![e]\!][x{:=}[\![e']\!]]$.

Proof. We work by induction on the structure of e.

- $[\![x[x{:=}e']]\!] = [\![e']\!] = x[x{:=}[\![e']\!]]$.
- $$\begin{aligned}[\![(e_1e_2)[x{:=}e']]\!] &= [\![e_1[x{:=}e'](e_2[x{:=}e'])]\!] \\ &= [\![e_1[x{:=}e']]\!][\![e_2[x{:=}e']]\!]) \\ &\overset{ind.hyp.}{=} [\![e_1]\!][x{:=}[\![e']\!]]([\![e_2]\!][x{:=}[\![e']\!]]) \\ &= ([\![e_1]\!][\![e_2]\!])[x{:=}[\![e']\!]] \\ &= [\![e_1e_2]\!][x{:=}[\![e']\!]].\end{aligned}$$

- $[\![(\lambda y.e)[x{:=}e']]\!] = [\![\lambda y.(e[x{:=}e'])]\!]$
  $= \lambda y.[\![e[x{:=}e']]\!]$
  $= \lambda y.([\![e]\!][x{:=}[\![e']\!]])$
  $= (\lambda y.[\![e]\!])[x{:=}[\![e']\!]]$
  $\overset{\text{Lemma 5.1}}{=} [\![\lambda y.e]\!][x{:=}[\![e']\!]].$

  Here we assume that $y \notin fv(e')$.

□

Theorem 5.3. *If $e \to e'$ then $[\![e]\!] \rightsquigarrow^* [\![e']\!]$.*

Proof. We work by induction on the derivation of $e \to e'$.

- The case (β). Then $(\lambda x.e)e' \to e[x{:=}e']$, where (renaming x if necessary) we choose $x \notin fv(e')$.

$$\begin{aligned}[\![(\lambda x.e)e']\!] &= (\lambda x.[\![e]\!])[\![e']\!] \\ &\rightsquigarrow [\![e]\!][x \mapsto [\![e']\!]] \\ &\overset{\text{Lemma 3.6}}{\rightsquigarrow^*} [\![e]\!][x{:=}[\![e']\!]]. \\ &\overset{\text{Lemma 5.2}}{=} [\![e[x{:=}e']]\!].\end{aligned}$$

The other cases are easy. □

Write $e \not\to$ when there is no e' such that $e \to e'$. If $e \not\to$ call e a **normal form**.

Lemma 5.4 (Preservation of normal forms). *If e is a normal form then $[\![e]\!]$ is a normal form.*
As a corollary, if e is any untyped λ-term, then if e has a normal form then so does $[\![e]\!]$.

Proof. The corollary follows by Theorem 5.3.

It is a fact [3] that the normal forms of the untyped λ-calculus are inductively characterised (as a subset of the set of terms of the untyped λ-calculus) by:

$$V ::= x \mid xV \ldots V \mid \lambda x.V.$$

The proof is by induction on V.

- $[\![x]\!] = x$ and we check the reduction rules of the LamCC and observe that x is a normal form.
- $[\![xV_1 \ldots V_n]\!] = x[\![V_1]\!] \ldots [\![V_n]\!]$. We check the reduction rules of the LamCC and observe that if $[\![V_1]\!], \ldots, [\![V_n]\!]$ are normal forms, then so is $x[\![V_1]\!] \ldots [\![V_n]\!]$.
- $[\![\lambda x.V]\!] = \lambda x.[\![V]\!]$. By assumption $[\![V]\!]$ is a normal form. We check the reduction rules of the LamCC and observe that if $[\![V]\!]$ is a normal form then so is $\lambda x.[\![V]\!]$.

□

The encoding of and the results in this section can be factored through the explicit substitution calculus $\lambda\mathrm{x}$ with garbage collection [5], to which LamCC is isomorphic when only variables of level 1 are considered. From this we obtain the following theorem for free:

Theorem 5.5 (Preservation of strong normalisation). *If e is λ-term that is strongly normalising for β-reduction, then $[\![e]\!]$ is strongly normalising in LamCC.*

Proof. Since $\mathsf{level}([\![e]\!]) = 1$, the reductions to which $[\![e]\!]$ is subject are precisely those of $\lambda\mathrm{x}$ with garbage collection, for which the theorem holds [5]. □

6. Contexts and contextual equivalence

In this section we consider contextual equivalence of the LamCC. The hierarchy of variables in LamCC allows us to internalise capturing substitution, and one consequence of this is that we can express contexts as terms, and internalise context-closed equivalences (i.e congruences) as described in Theorem 6.2.

Recall that an **equivalence relation** is a transitive symmetric reflexive relation, and recall from Definition 2.4 that a **congruence** is an equivalence relation between terms that is closed under the rules of Figure 3.

Definition 6.1. *Write $s \leftrightarrow t$ for the least equivalence relation containing $\rightsquigarrow$ (the transitive reflexive symmetric closure of $\rightsquigarrow$).*

Theorem 6.2. *Suppose $\mathcal{X}$ is an equivalence relation containing $\leftrightarrow$.*
Then $\mathcal{X}$ is a congruence if and only if $\forall s,t.s\ \mathcal{X}\ t \Rightarrow \forall C.C\ s\ \mathcal{X}\ C\ t$.

Proof. The *only if* part is straightforward; the desired property is one of the rules defining the notion of congruence. For the *if* part we have to show that $\mathcal{X}$ is closed under the rules of congruence:

- Suppose $s\ \mathcal{X}\ s'$, then by assumption $(\lambda d_{i+1}.\lambda a_i.d_{i+1})\ s\ \mathcal{X}\ (\lambda d_{i+1}.\lambda a_i.d_{i+1})\ s'$. Since $\leftrightarrow \subseteq \mathcal{X}$, we have $\lambda a_i.s\ \mathcal{X}\ \lambda a_i.s'$.
- Suppose $s\ \mathcal{X}\ t$ and $s'\ \mathcal{X}\ t'$ and let $i = \mathsf{level}(s,s',t,t')$;
  then by assumption $(\lambda d_{i+1}.d_{i+1}\ d'_{i+1})\ s\ \mathcal{X}\ (\lambda d_{i+1}.d_{i+1}\ d'_{i+1})\ t$
  and thus $(\lambda d'_{i+1}.(\lambda d_{i+1}.d_{i+1}\ d'_{i+1})\ s)\ s'\ \mathcal{X}\ (\lambda d'_{i+1}.(\lambda d_{i+1}.d_{i+1}\ d'_{i+1})\ t)\ t'$.
  Since $\leftrightarrow \subseteq \mathcal{X}$, we have $s\ s'\ \mathcal{X}\ t\ t'$.
- The case for $s[a_i{\mapsto}s']\ \mathcal{X}\ t[a_i{\mapsto}t']$ is similar, combining the arguments of the two cases above.
- The cases for reflexivity, transitivity, and symmetry are given by the fact that $\mathcal{X}$ is an equivalence relation.

Hence $\mathcal{X}$ is a congruence. □

Now we apply Theorem 6.2 to a particular congruence:

Definition 6.3. *Let $\top = \lambda x.x$ (we could also take $\top$ to be a new constant such that $\top \not\rightsquigarrow$). Write $s{\searrow}$ when $s \rightsquigarrow^* \top$.*

Define **contextual equivalence** $=_{ctx}$ *as the greatest congruence such that*

$$\forall s,t.(s\ \mathcal{X}\ t \wedge s{\searrow}) \Rightarrow t{\searrow}$$

We can characterise $=_{ctx}$ as follows:

Theorem 6.4. *$=_{ctx}$ is the greatest equivalence relation $\mathcal{X}$ such that*

$$\forall s,t.s\ \mathcal{X}\ t \Rightarrow \forall C.C\ s\ \mathcal{X}\ C\ t \qquad \textit{and} \qquad \forall s,t.(s\ \mathcal{X}\ t \wedge s{\searrow}) \Rightarrow t{\searrow}.$$

Proof. *We show that $\mathcal{X} \subseteq =_{ctx}$.* It suffices to check that $\mathcal{X}$ is a congruence. This follows by Theorem 6.2 if we show that $\leftrightarrow \subseteq \mathcal{X}$. To do this, we need only show that

$$\forall s,t.s \leftrightarrow t \Rightarrow \forall C.C\ s \leftrightarrow C\ t \qquad \text{and} \qquad \forall s,t.(s \leftrightarrow t \wedge s{\searrow}) \Rightarrow t{\searrow}.$$

This follows from confluence (Theorem 4.1).

We show that $=_{ctx} \subseteq \mathcal{X}$. It suffices to show that $=_{ctx}$ is an equivalence relation and

$$\forall s,t.s =_{ctx} t \Rightarrow \forall C.C\ s =_{ctx} C\ t \qquad \text{and} \qquad \forall s,t.(s =_{ctx} t \wedge s{\searrow}) \Rightarrow t{\searrow}.$$

This is easy from the definitions. □

The applicative characterisation of contextual equivalence between closed terms only as follows

$$\forall s,t.s\ \mathcal{P}\ t \Rightarrow \forall u.su\ \mathcal{P}\ tu$$

requires the development of too many technicalities for this paper introducing the LamCC.

$$
\begin{array}{lll}
(\mathsf{И}\mathbf{p}) & (\mathsf{И}a_i.s)t \rightsquigarrow \mathsf{И}a_i.(st) & a_i \notin fv(t) \\
(\mathsf{И}\sigma) & (\mathsf{И}c_k.s)[a_i{\mapsto}t] \rightsquigarrow \mathsf{И}c_k.(s[a_i{\mapsto}t]) & k \leq i,\ c_k \notin fv(t) \\
(\mathsf{И}\notin) & \mathsf{И}a_i.s \rightsquigarrow s & a_i \notin fv(s)
\end{array}
$$

$$
\frac{s \rightsquigarrow s'}{\mathsf{И}a_i.s \rightsquigarrow \mathsf{И}a_i.s'}\ (\mathbf{R}\mathsf{И})
$$

Figure 9: New clauses

7. A NEW part for the LamCC

7.1. Some NEW rules

LamCC λ-abstraction is weak in the sense that for example x is not α-convertible in $\lambda x.X$, if x has level 1 and X has level 2.

In the presence of only one level of variable this difference between binding and abstraction is invisible. The LamCC of Definition 2.2 has abstraction, but it does not have binding. To recover it we introduce a dedicated binder $\mathsf{И}$ into the syntax, with appropriate reduction rules as follows:

Definition 7.1. *Extend the syntax of the LamCC (Definition 2.2):*

$$s, t ::= \ldots \mid \mathsf{И}a_i.t.$$

Extend the definition of level *and fv (Definition 2.3) with the clauses*

$$\mathsf{level}(\mathsf{И}a_i.s) = \mathsf{max}(i, \mathsf{level}(s)) \qquad fv(\mathsf{И}a_i.s) = fv(s) \setminus \{a_i\}.$$

Extend the definition of congruence (Definition 2.4) with a clause

$$\frac{s\ R\ s'}{\mathsf{И}a_i.s\ R\ \mathsf{И}a_i.s'}.$$

Extend the definition of swapping (Definition 2.5) with a clause

$$(a_i\ b_i)\mathsf{И}c.s = \mathsf{И}(a_i\ b_i)c.(a_i\ b_i)s$$

where c is any atom.

Extend the definition of α-equivalence (Definition 2.8) with a clause

$$\mathsf{И}a_i.s =_\alpha \mathsf{И}b_i.(b_i\ a_i)s \qquad \text{if } b_i \notin fv(s).$$

Note the difference that in α-equivalence for λ-abstraction we check $b_i \# fv(s)$ (Definition 2.7), and in α-equivalence for $\mathsf{И}$-binding we check $b_i \notin fv(s)$.

Variables bound by $\mathsf{И}$ rename regardless of whether stronger variables are present. Take x and y to be variables of level 1, and X and Y to be variables of level 2. Here are some example α-(non-)equivalences:

$$\lambda x.(Xx) \neq_\alpha \lambda y.(Xy) \qquad \text{but} \qquad \mathsf{И}x.\lambda x.(Xx) =_\alpha \mathsf{И}y.\lambda y.(Xy)$$
$$\text{and} \quad \mathsf{И}x.(Xx) =_\alpha \mathsf{И}y.(Xy).$$

Definition 7.2. *Extend the reduction rules (Definition 2.12) with the clauses in Figure 9.*

Here is an example reduction which exploits И:

$$\begin{aligned}(\lambda X.И x.\lambda x.X)x &\overset{(\beta)}{\rightsquigarrow} (И x.\lambda x.X)[X{\mapsto}x]\\ &\overset{(И\sigma)}{\rightsquigarrow} И x'.((\lambda x'.X)[X{\mapsto}x])\\ &\overset{(И\lambda)}{\rightsquigarrow} И x'.\lambda x'.(X[X{\mapsto}x])\\ &\overset{(И\mathbf{a})}{\rightsquigarrow} И x'.\lambda x'.x\end{aligned}$$

Compare with a pair of related rewrites in the syntax without И:

$$(\lambda X.\lambda x.X)x \rightsquigarrow^* \lambda x.x \qquad (\lambda X.\lambda y.X)x \rightsquigarrow^* \lambda y.x$$

So И binds, and this is separated from the functional abstraction, which is managed using λ.

LamCC syntax permits И also not directly above λ, for example in $И x.x$. This behaves like a constant symbol, and indeed

$$fv(И x.x) = \emptyset$$

— note however that level information is preserved; $\mathsf{level}(И a_i.a_i) = i$. The И which binds x ensures that x cannot be substituted for:

$$(И x.x)[x{\mapsto}t] \overset{(И\sigma)}{\rightsquigarrow} И x'.(x'[x{\mapsto}t]) \overset{(\sigma fv)}{\rightsquigarrow} И x'.x' =_\alpha И x.x.$$

(Recall that we take terms up to α-equivalence when discussing reductions.) See Subsection 7.3 for an example which makes non-trivial use of И independently of λ.

Remark 7.3. Choosing the symbol 'И' and calling it 'new', is an obvious quote of the Gabbay-Pitts И quantifier [21]. The И binder here is *not* the И quantifier. Firstly, it is not a logical connective (we are in a λ-calculus!). Secondly, it does not just generate a fresh *name* (as does the И-quantifier) — the И of this paper generates a fresh *variable*. To see the distinction observe that in $И x.(x[x{\mapsto}2])$ the variable x is substituted for 2 — we can also write $И x.(\lambda x.x)2$; its LamCC reduction passes through $И x.(x[x{\mapsto}2])$.

Therefore any denotational semantics for the entity 'x' that is generated by '$И x$' in the LamCC must be an entity which can be λ-abstracted and substituted for. This behaviour is not displayed by the atoms generated by the И-quantifier introduced in [21].

This extension of И beyond logic and beyond binding names, is part of a broader programme by the first author to enrich 'nominal techniques' from a basket of techniques for manipulating syntax-with-binding (of which the И quantifier is a part), in the direction of a general strategy for coping with other name-like entities in computer science. The multiple levels of variable in the LamCC and its operational semantics, including the И binder, are part of this thinking.

Remark 7.4. The behaviour of И is comparable with some of the behaviour of $\forall$ in logic. If we read st and $s[a_i{\mapsto}t]$ as '$t \Rightarrow s$', and $s \rightsquigarrow t$ as 't entails s' then rules $(И\mathbf{p})$ and $(И\sigma)$ look structurally like

$$\forall x.(\phi \Rightarrow \psi) \qquad \text{entails} \qquad \phi \Rightarrow \forall x.\psi \quad \text{if } x \notin fv(\phi),$$

and $(И\notin)$ looks structurally like

$$\phi \qquad \text{entails} \qquad \forall x.\phi \quad \text{if } x \notin fv(\phi).$$

This similarity comes from the fact that those properties of $\forall$ necessary to prove the entailments above in first-order logic (everything to do with right-introduction) are precisely those properties of $\forall$ that stem from the way its sequent right introduction rule introduces a fresh variable on the right [4]. Refining this observation a more formal mathematical result, if any, is for future work.

7.2. Some false NEW rules

We do not admit a rule

$$(\mathbf{И pFALSE}) \qquad \mathbf{s}(И\mathbf{a}.\mathbf{t}) \rightsquigarrow И\mathbf{a}.(\mathbf{st}) \qquad \mathbf{a} \notin fv(\mathbf{s}).$$

With $(\mathbf{И pFALSE})$ we can reduce as follows:

$$\begin{array}{lcl} (\lambda x.xx)Иy.y & \overset{(\mathbf{И pFALSE})}{\rightsquigarrow} & Иy.(\lambda x.xx)y \\ & \overset{(\beta),(\sigma\mathbf{p}),(\sigma\mathbf{a}),(\sigma\mathbf{a})}{\rightsquigarrow^*} & Иy.yy \\ (\lambda x.xx)Иy.y & \overset{(\beta),(\sigma\mathbf{p}),(\sigma\mathbf{a}),(\sigma\mathbf{a})}{\rightsquigarrow^*} & (Иy.y)Иy'.y' \\ & \overset{(И\mathbf{p}),(\mathbf{И pFALSE})}{\rightsquigarrow^*} & Иy.Иy'.(yy'). \end{array}$$

It is a fact that these terms are normal forms and they are *not* equal.

We can have an intuition of И as 'generating a fresh variable'. Then $(\mathbf{И pFALSE})$ (with the other rules of the LamCC) lets us make a non-confluent choice of whether to generate a name, then copy, or copy and then generate.

For similar reasons we do not admit $(И\sigma\mathbf{FALSE})$:

$$(И\sigma\mathbf{FALSE}) \qquad \mathbf{s}[\mathbf{b}\mapsto И\mathbf{a}.\mathbf{t}] \rightsquigarrow И\mathbf{a}.(\mathbf{s}[\mathbf{b}\mapsto\mathbf{t}]) \qquad \mathbf{a} \notin fv(\mathbf{s}).$$

Why the side-conditions on $(И\sigma)$? Clearly the condition $c_k \notin fv(t)$ comes from the intuition of И as defining a scope. We insist on $k \leq i$ to guarantee confluence:

$$\begin{array}{lcl} (ИX.x)[x\mapsto 2] & \overset{(\sigma fv)}{\rightsquigarrow} & ИX.x \\ & \overset{(И\notin)}{\rightsquigarrow} & x \\ (ИX.x)[x\mapsto 2] & \overset{(И\sigma\mathbf{FALSE})}{\rightsquigarrow} & ИX.(x[x\mapsto 2]) \\ & \overset{(\sigma\mathbf{a})}{\rightsquigarrow} & 2. \end{array}$$

Proofs extend smoothly to the calculus extended with rules for И, including confluence and termination of $(\mathbf{sigma})$ extended with the rules for И.

7.3. Global state using NEW

Consider an untyped λ-calculus with general references.

Definition 7.5. *Terms are generated by the grammar*

$$e ::= x \mid !l \mid l := e \mid Иl.e \mid \texttt{skip} \mid ee \mid \lambda x.e.$$

Here l is drawn from a set of **location variables** and x from a set of **program variables**. We read through informal intended meanings of $!l$, $l := e$, and $Иl.e$:

- $!l$ means 'the value which the global state associates to l'.
- $l := e$ means 'set l to e in the global state'.
- $Иl.e$ means 'allocate a fresh local location (with some unspecified or default value associated) and evaluate e in the extended state'.

The rest is standard as for the untyped λ-calculus.

This calculus is based on others in the literature, for example $\mathcal{L}$ [1, Fig. 1 and Fig. 2] and ReFS [35, Fig. 1 and Fig. 4]. The focus of study for $\mathcal{L}$ is a fully abstract game semantics; that of ReFS

is how to prove contextual equivalences between programs. Both $\mathcal{L}$ and ReFS are typed. For brevity we have considered an untyped calculus.

[1] contains a construction of the Y combinator using general references in a typed system, thus obtaining the power of recursion. Therefore the absence of types does not make the calculus above obviously more powerful than $\mathcal{L}$, though it is more powerful than ReFS, which has recursion but integer-only references.

We encode this in the LamCC as follows: Fix a level 2 variable $\mathbf{W}$ 'the state' and a level 1 variable $\mathbf{r}$ 'the result'. Equate location variables l and λ-variables x in the syntax of Definition 7.5 with disjoint classes of level 1 variables, which do not include $\mathbf{r}$. Consistent with the final example from Subsection 2.5 (records) we write

$$t.l \quad \text{for} \quad t[\mathbf{W}\mapsto l] \quad \text{and} \quad t.l := s \quad \text{for} \quad t[\mathbf{W}\mapsto \mathbf{W}[l\mapsto s]].$$

Then we define a translation to the LamCC as follows:

$$\begin{array}{rcll}
[\![x]\!] & = & \lambda\mathbf{W}.\mathbf{W}.\mathbf{r} := x & \text{'Bind } \mathbf{r} \text{ to } x\text{'} \\
[\![!l]\!] & = & \lambda\mathbf{W}.\mathbf{W}.\mathbf{r} := \mathbf{W}.l & \text{'Bind } \mathbf{r} \text{ to dereference } l\text{'} \\
[\![l := e]\!] & = & \lambda\mathbf{W}.(\mathbf{W}.l := e).\mathbf{r} := \top & \text{'Bind } l \text{ to } e\text{, bind } \mathbf{r} \text{ to } \top\text{'} \\
[\![\texttt{skip}]\!] & = & \lambda\mathbf{W}.\mathbf{W}.\mathbf{r} := \top & \text{'Bind } \mathbf{r} \text{ to } \top \text{ (no-op)'} \\
[\![\lambda x.e]\!] & = & \lambda\mathbf{W}.И x.\lambda x.([\![e]\!]\mathbf{W}x) & \text{'Input } x\text{, run } e \text{ applied to } x\text{'} \\
[\![ee']\!] & = & \lambda\mathbf{W}.[\![e]\!]([\![e']\!]([\![e']\!]\mathbf{W})(\mathbf{W}.\mathbf{r})) & \text{'Run } e'\text{, run } e \text{ in updated state} \\
 & & & \quad \text{applied to result of } e'\text{'} \\
[\![И l.e]\!] & = & \lambda\mathbf{W}.И l.([\![e]\!](\mathbf{W}.l := \top)) & \text{'Create new reference } l\text{,} \\
 & & & \quad \text{run } e \text{ in updated state'}
\end{array}$$

$[\![e]\!]$ is a 'state transformer', that is, a function which transforms states into other states. By convention the results of computations are placed in $\mathbf{r}$. It is not hard to verify that the definitions, along with the reductions of the LamCC, simulate the reductions we expect of a language with references and in particular those from [1].

The example of $И l.e$ illustrates a $И$ quantifier used independently of λ-abstraction, to generate a fresh variable without functionally abstracting it.

Here is a program which 'encapsulates' $[\![e]\!]$; it creates a global state, runs e in that state, throws away the state and returns the final result:

$$И\mathbf{W}.([\![e]\!]\mathbf{W}).\mathbf{r}$$

Note that $\lambda\mathbf{W}.([\![e]\!]\mathbf{W}.\mathbf{r})$ does not give the desired behaviour and is incorrect (the abstraction we need is not a functional abstraction). The ideas behind this program find expression in the monadic style of programming with global state described for example in [28] (see for example `runST` in [28, Subsection 2.4]) which uses ideas going back to [33].

We conclude this subsection with two detailed remarks about $И$, λ, and binding:

Remark 7.6. $И$ adds expressivity. When in $\lambda a_i.t$ levels of variables in t are low enough with respect to the level of the abstracted variable λa_i, the λ-abstractor integrates binding behaviour; see Definition 2.8. Thus the binding behaviour of λ coincides with the 'usual' binding behaviour in the λ-calculus as mentioned in Section 5, and with the binding behaviour of $И$ as described in Definition 7.1.

In other words, if the variables mentioned in t are not too strong, then $Иa.\lambda a.t$ has the same behaviour as $\lambda a.t$.

In the case of $\lambda a.X$ and $Иa.\lambda a.X$ where X is stronger than a, this ceases to be the case and the notions of abstraction and name-binding part company to become discernably distinct elements of the language's structure.

In order to emulate references we need to be in the latter case, i.e. we need to use strong (level 2, in our examples) variables — even though the source language only has one level of variable (level 1, in our examples).

It is future work to consider non-operational semantics for the LamCC, which might throw further light on this distinction. A denotational semantics would be very useful here and is currently lacking.

Remark 7.7. In presence of И Theorem 6.2 no longer holds. The context $Иx.[_]$ cannot be represented in LamCC syntax and reductions by application of the form $C\ [_]$. This is because, by design, И forbids the capture of the variable that it binds regardless of the strength of other variables.

We can recover Theorem 6.2 without defeating the purpose of introducing И, and without even going to too much effort. Perhaps the simplest way is to consider a transfinite hierarchy of variables. For example, we can take levels to be $1, 2, 3, \ldots, \omega$ where ω is the first infinite ordinal. We let И avoid capture only by variables that are not transfinitely stronger. According to this paradigm, substitution for b_j in $Иa_i.b_j$ avoids capture of a_i if i and j are finite (even if $j > i$), but it does not avoid capture if i is finite and $j = \omega$. This is an easy extension of the LamCC. Indeed, there are other ways to order our levels of variable, and the basic ideas on which LamCC reductions are based seem quite robust and amenable to extension. We discuss this further in Subsection 8.1.

8. Design variations

As is very often the case, design decisions have been made which could plausibly have been made differently. We sketch some of them now.

8.1. Different schemes of levels

We briefly explore to what extent the LamCC depends on it having levels $1, 2, 3, \ldots$ (instead of some other ordered set of levels). This continues an issue touched on in Remark 7.7.

Recall from Definition 2.1 that $\mathbb{A}_i$ is the set of variables of level i.

Definition 8.1. *Let f be an injective order-preserving map from numbers $1, 2, 3, \ldots$ to numbers (so $f(i) = f(j)$ implies $i = j$, and $i \leq j$ implies $f(i) \leq f(j)$).*

For each pair of distinct numbers i and j make a fixed but arbitrary choice of bijection ι_{ij} between $\mathbb{A}_i$ and $\mathbb{A}_j$. By convention, if $a_i \in \mathbb{A}_i$ then we shall write $\iota_{ij}(a_i)$ as a_j.

Define t^f inductively by:

$$(a_i)^f = a_{f(i)} \quad (t't)^f = (t')^f t^f \quad (\lambda a_i.t)^f = \lambda a_{f(i)}.t^f$$

$$(u[a_i \mapsto t])^f = u^f[a_{f(i)} \mapsto t^f] \quad (Иa_i.t)^f = Иa_{f(i)}.t^f$$

Technically we should show that t^f is well-defined on α-equivalence classes, but it is.

Lemma 8.2. *Suppose f is an injective order-preserving map from numbers to numbers. Then:*

- *$level(t^f) = f(level(t))$.*
- *$a_i \# fv(t)$ if and only if $a_{f(i)} \# fv(t^f)$.*

Proof. By routine inductions using the definitions of $level(t)$ and $fv(t)$ in Figures 1 and 2. We consider only the two very slightly non-trivial cases:

- The case of b_j where $j \leq i$. $level((b_j)^f) = level(b_{f(j)})$. The first part follows.

  $fv((b_j)^f) = \{b_{f(j)}\}$. Also $f(j) \leq f(i)$. By injectivity, we know that $a_{f(i)}$ and $b_{f(j)}$ are distinct variables, because a_i and b_j are distinct variables. It is then a fact that $a_i \# fv(b_j)$ and $a_{f(i)} \# fv(b_{f(i)})$.

- The case of b_j where $i < j$. $level((b_j)^f) = level(b_{f(j)})$. The first part follows. $fv((b_j)^f) = \{b_{f(j)}\}$. Also $f(i) < f(j)$. It is a fact that $a_i \# fv(b_j)$ is false, and $a_{f(i)} \# fv(b_{f(i)})$ is false.

□

Theorem 8.3. *$s \rightsquigarrow t$ if and only if $s^f \rightsquigarrow t^f$.*

Proof. By a routine induction on derivations, using Lemma 8.2 to verify that no side-conditions change validity under the translation. □

Theorem 8.3 gives a formal sense in which the precise scheme of levels does not matter, so long as it is a total order and we have 'enough levels'. In this paper we have taken $1, 2, 3, \ldots$, but other total orders would do as well. Considering orders that are not total is future work.

8.2. LamCC with only one binder

The LamCC has three binders; И which always binds, λ which binds only if there are no stronger variables in scope, and the explicit substitution which also binds only if there are no stronger variables in scope. For example, if x and y have level 1 and X has level 2, then in the LamCC we have that

- $И x.x =_\alpha И y.y$ and $И x.X =_\alpha И y.X$, and
- $\lambda x.x =_\alpha \lambda y.y$ and $\lambda x.X \neq_\alpha \lambda y.X$.
- $x[x \mapsto y] =_\alpha y[y \mapsto y]$ and $X[x \mapsto y] \neq_\alpha X[y \mapsto y]$.

We can consider a variant LamCC′, identical to the LamCC except that it has a less sophisticated functional abstraction $\lambda' x.t$ which never binds, replacing the λ which sometimes does, and similarly an explicit substitution $u[a_i \mapsto t]'$ which never binds, so that

- $И x.x =_\alpha И y.y$ and $И x.X =_\alpha И y.X$, and
- $\lambda' x.x \neq_\alpha \lambda' y.y$ and $\lambda' x.X \neq_\alpha \lambda' y.X$.
- $x[x \mapsto y]' \neq_\alpha y[y \mapsto y]'$ and $X[x \mapsto y]' \neq_\alpha X[y \mapsto y]'$.

That is, there is only one binder in LamCC′, and that is И.

We can then translate LamCC into LamCC′ such that:

- $\lambda a_i.t$ translates to $И a_i.\lambda' a_i.t$, if $fv(t)$ contains no variables of level greater than i, and
- $\lambda a_i.t$ translates to $\lambda' a_i.t$, if $fv(t)$ contains a variable of level greater than i.
- $u[a_i \mapsto t]$ translates to $И a_i.u[a_i \mapsto t]'$, if $fv(u)$ contains no variables of level greater than i and (renaming if necessary) $a_i \notin fv(t)$, and
- $u[a_i \mapsto t]$ translates to $u[a_i \mapsto t]'$, if $fv(u)$ contains a variable of level greater than i.

The definitions and basic properties of LamCC reductions transfer smoothly to LamCC′, as does the proof of confluence.

A notable difference is that in the LamCC′, 'ordinary' terms (only one level of variable, no И) can fail to reduce simply because an α-conversion cannot take place. For example $(\lambda' y.(\lambda' x.y))x$ will not reduce to $\lambda' z.x$ (in the LamCC it will, because λ binds in $(\lambda y.(\lambda x.y))x$). Accordingly, the translation of the untyped λ-calculus described in Section 5 must be changed, so that

$$[\![\lambda x.e]\!] = И x.\lambda' x.[\![e]\!].$$

It is a fact that Theorem 5.5 is still true, but we must prove it from scratch; we cannot so directly exploit the work in [5].

For these reasons we have allowed λ to bind in the LamCC; it makes for an easier presentation and better properties. However, there is a good argument that LamCC′ is the more primitive underlying system, because the LamCC can be translated into it, and LamCC′, unlike LamCC, more clearly separates functional abstraction and name-binding. The first author made this design choice in the NEWcc [11], which is previous related work; see the Conclusions.

For the purposes of this paper, we use the LamCC. In further work, LamCC′ may be the more convenient system, because there would be no need to case-split in proofs on whether functional abstraction λ and explicit substitution bind, or not — and binding becomes just another term-former, like functional abstraction or application, which is explicitly marked in the syntax by a $\mathsf{И}$, and there is no need to examine inside a term to check whether variables are strong or weak in order to decide whether a variable is bound or not.

8.3. LamCC with nominal terms style alpha-equivalence

We briefly mention one more design variant of the LamCC. In this paper we cannot α-convert x in $\lambda x.X$. Nominal terms can: swappings are in the syntax (here swappings are purely a meta-level) and also freshness contexts [45].

In separate work [20] based partly on experience from this paper, the first author with Mulligan considers a two-level λ-calculus with a full nominal terms theory of α-equivalence. We call the calculus of [20] 'two level λ-calculus', because it has two levels of variable, versus the infinite hierarchy of the LamCC, but the two level λ-calculus has a far stronger theory of α-equivalence; in the two level λ-calculus, it is possible to α-convert x in $\lambda x.X$ to $\lambda y.(y\ x) \cdot X$ (this is in nominal terms style, following [45, 10]).

In [20] swappings only exist for level 1 variables. We can envisage a calculus with the nominal terms theory of α-equivalence from [20], and the infinite hierarchy of variables of the LamCC. What blocks us from doing so, as much as anything else, is that we do not yet understand the theory of swappings for stronger variables (what we might write as '$(X\ Y)$'), or how swappings of different levels interact. We also do not yet fully understand how freshness contexts [45] interact with evaluations.

For the moment what we can say is that — so it seems to us — the LamCC strikes a sweet spot between complexity versus simplicity, and expressiveness versus good mathematical properties. However, the LamCC exists in a space of related λ-calculi, which remain to be properly explored.

9. Related work, conclusions, and future work

Related work (using nominal techniques)

In a previous conference paper we presented the NEW calculus of contexts [11]. The LamCC of this journal paper updates and improves that that work. The LamCC is simpler than the NEWcc. Compare the side-condition of $(\sigma\mathbf{a})$ (there is none) with that of $(\sigma\mathbf{a})$ from [11]. 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 [11]. 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, and the condition on rule $(\sigma\mathbf{p})$.

In the LamCC we permit α-conversion of an abstracted variable if no stronger variables are in the scope of the abstraction. In the NEWcc we did not do this. For example, if x and y have level 1 then $\lambda x.x$ and $\lambda y.y$ are equated in the LamCC, but not in the NEWcc. (If X has level 2, then $\lambda x.X$ and $\lambda y.Y$ are not equated in either system.) When we designed the NEWcc we wanted to make a point, by letting functional abstraction be managed exclusively by λ, and letting variable symbol name binding be managed exclusively by $\mathsf{И}$. As discussed in Subsection 8.2, we could have done the same in the LamCC and this would have caused no essential difficulties — indeed, things would become simpler, because there would be exactly one binder to consider, $\mathsf{И}$. The advantage of setting things up as in this paper is just for usability, relative to what the reader is likely to expect; if there is only one level of variable in a term then λ-abstraction is 'normal' and binds.

The LamCC does have fewer reductions than the NewCC in the sense that $(\sigma\lambda')$ will not reduce $(\lambda a_i.s)[c_k \mapsto u]$ where $k < i$ whereas a rule $(\sigma\lambda)$ in [11] does.[5] The stronger version of the rule turns out to be a major source of extra complexity, and we seem not to miss the extra reductions; empirically, we observe that the reductions that we need in order to express our examples and case studies, including those inherited from [11], are unaffected.

Still, it seems clear that we are approximating something larger. Other papers on nominal techniques have useful elements which we can import, now that we have a solid basis to work from. For example:

- As discussed in Subsection 8.3, we cannot α-convert x in $\lambda x.X$. Nominal terms can: swappings are in the syntax (here swappings are purely a meta-level) and also freshness contexts [45]. A problem is that we do not yet understand the theory of *swappings* for strong variables; the underlying Fraenkel-Mostowski sets model [21] 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 $x\#fv(\lambda x.X)$ even though for every *instance* this does hold (for example $x\#\lambda x.x$ and $x\#\lambda x.y$). Hierarchical nominal rewriting [12] has a more powerful notion of freshness which can prove the equivalent of $x\#fv(\lambda x.X)$. Note that hierarchical nominal rewriting does 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 [36] or the style of a sequent calculus for Nominal Logic by Cheney [7].

- Finally, we cannot reduce $X[x \mapsto 2][y \mapsto 3]$ to $X[y \mapsto 3][x \mapsto 2]$. Other work [15, 17] gives an *equational* system which can do this, and more.

Desirable meta-properties of the λ-calculus survive in the LamCC: the LamCC is confluent and, although the LamCC is a calculus with an explicit substitution, the λ-calculus naturally translates into it in a way which supports strong normalisation.

More related work (not using nominal techniques)

The calculi of contexts λm and $\lambda\mathcal{M}$ [39] also have a hierarchy of variables. They use carefully-crafted scoping conventions to manage problems with α-conversion. Other work [37, 24, 38] uses a type system; connections with this work are unclear. λc of Bognar's thesis contains [6, Section 2] an extensive literature survey on the topic of context calculi.

A separation of abstraction λ and binding И appears in one other (unpublished) work we know of [41], where they are called q and ν. In this vein there is [25], which manages scope explicitly in a completely different way, just for the fun. Finally, the reduction rules of И look remarkably similar to π-calculus restriction [32], and it is probably quite accurate to think of И as a 'restriction in the λ-calculus'.

Hamana takes a semantic approach to meta-variables [23]. We have not developed the semantic theory of the LamCC. We should do this in future work, and when we do we would expect to arrive at something similar to Hamana's construction. However we do not expect the semantics to be identical; ours it will be phrased in terms of sets and permutation actions (in keeping with the first author's previous work [21]). These have slightly stronger, and in our opinion more desirable, properties than the categories of presheaves which Hamana uses.

Ours is a calculus with explicit substitutions. See [30] for a survey. Our treatment of substitution is simple but still quite subtle because of interactions with the rest of the language. The

[5] $(\sigma\lambda)$ in this paper corresponds with $(\sigma\lambda')$ in [11] and $(\sigma\lambda')$ in this paper corresponds with $(\sigma\lambda)$ in [11] — the rule names have been switched. This maintains consistency, in this paper, with $(\sigma\sigma)$ which distributes a strong substitution under a weaker substitution; now the rule called $(\sigma\lambda)$ also distributes a strong substitution, under a weaker λ-abstraction.

translation of possibly open terms of the untyped λ-calculus into the LamCC preserves strong normalisation. The reduction rule (σfv) is a little unusual amongst such calculi, though it appears as Bloo's 'garbage collection' [5], but the reasons why we include it come naturally from the hierarchy of variables.

The look and feel of the LamCC 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 [11] complexity spilled over into other rules and into a logic for freshness). $\#$ is only necessary when variables of different strengths occur, and the hierarchy of variables only plays a role to trigger side-conditions.

Further work

We discussed above how to add off-the-shelf elements of nominal techniques, if we want to give ourselves more reductions.

Another possibility is 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 $\Rightarrow\colon o \to o \to o$ where o is a type of truth-values [47], along with suitable equalities and/or derivation rules. It is easy to impose a simple type system on terms. Thus, we see no problem with writing down a 'context higher-order logic'. This takes the LamCC in the direction of calculi of contexts for incomplete proofs [26, 22], and also in the direction of giving semantics to existential variables in logic-programming (unpublished work by Lipton and Mariño). The non-trivial work (in no particular order) is to investigate cut-elimination, develop a suitable theory of models, and to check whether and how usefully the LamCC can be used as-is to model incomplete proofs of some theorem-proving system.

One important element is the denotation semantics of the hierarchy of variables — we do not yet have one; this is current work.

Certain specific ideas appear elsewhere in the literature which the LamCC *cannot* express, but they might be accommodated with a relatively straightforward extension.

As discussed, the LamCC can express $[a_i{\mapsto}t]$ using the term $\lambda b_j.(b_j[a_i{\mapsto}t])$ where $i < j$ and $\mathsf{level}(t) \leq j$. However we cannot abstract over a_i. For example consider $\lambda P.\lambda X.\lambda x.(X[x{\mapsto}P])$ and the reduction

$$\begin{array}{lll} (\lambda P.\lambda X.\lambda x.X[x{\mapsto}P])2xy & \rightsquigarrow^* & X[x{\mapsto}P][P{\mapsto}2][X{\mapsto}x][x{\mapsto}y] \\ & \rightsquigarrow & X[x{\mapsto}2][X{\mapsto}x][x{\mapsto}y] \\ & \rightsquigarrow^* & x[x{\mapsto}2][x{\mapsto}y] \\ & \rightsquigarrow^* & 2. \end{array}$$

This is not the intended operational behaviour (if we intended to abstract over the name of x and replace it by y). Variables can be substituted for so it should not possible to pass a variable name as a first-class value. An extension of the LamCC based on atom from [13] may be possible and useful.

The LamCC cannot express 'substitute all variables of level 1 for t in s', which we might write as $s[\star{\mapsto}t]$. This idea appears in work by Dami [9] on *dynamic binding*. We do believe that the LamCC could have something to contribute to dynamic binding and linking, since they seem to have to do with capturing substitution, but that is future work.

Languages for staged computation, for example MetaML [34], Template Haskell [40], and Converge [44], have a hierarchy (of stages) reminiscent of our hierarchy of levels. They offer a program enough control of its own execution that it can suspend its own execution, compose suspended programs into larger (suspended) programs, pass suspended programs as arguments to functions, and evaluate them. This raises issues similar to those surrounding contexts. The LamCC cannot model staged computation because it is a pure rewrite system with no control of evaluation order. Even if we choose some evaluation order on the LamCC to make it into a programming language, the deeper problem is that the LamCC has no first-class construct to promote variables between levels. Adding this extension is interesting future work.

In conclusion, the LamCC of this paper is expressive and yet it remains simple and quite clear, and it has good properties. It seems to hit a technical sweet spot. Often in computer science the trick is to find a useful balance between simplicity and expressivity. Perhaps the LamCC does that.

References

[1] Samson Abramsky, Kohei Honda, and Guy McCusker. A fully abstract game semantics for general references. In *Proc. 13th IEEE Symp. Logic in Comp. Sci.*, pages 334–344. IEEE Computer Society Press, 1998.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[3] Henk P. Barendregt. *The Lambda Calculus: its Syntax and Semantics (revised ed.)*. North-Holland, 1984.

[4] Jon Barwise. An introduction to first-order logic. In Jon Barwise, editor, *Handbook of Mathematical Logic*, pages 5–46. North Holland, 1977.

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

[6] Mirna Bognar. *Contexts in Lambda Calculus*. PhD thesis, Vrije Universiteit Amsterdam, 2002.

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

[8] Ranald A. Clouston and Andrew M. Pitts. Nominal equational logic. *Electronic Notes in Theoretical Computer Science*, 172:223–257, 2007.

[9] Laurent Dami. A lambda-calculus for dynamic binding. *Theoretical Computer Science*, 192(2):201–231, 1998.

[10] Maribel Fernández and Murdoch J. Gabbay. Nominal rewriting (journal version). *Information and Computation*, 205(6):917–965, 2007.

[11] Murdoch J. Gabbay. A NEW calculus of contexts. In *PPDP'05*, pages 94–105. ACM, 2005.

[12] Murdoch J. Gabbay. Hierarchical Nominal Terms and Their Theory of Rewriting. *Electronic Notes in Theoretical Computer Science*, 174(5):37–52, 2007.

[13] Murdoch J. Gabbay and Michael J. Gabbay. a-logic. In *We Will Show Them: Essays in Honour of Dov Gabbay*, volume 1. College Publications, 2005.

[14] Murdoch J. Gabbay and Aad Mathijssen. Nominal Algebra. In *18th Nordic Workshop on Programming Theory*, 2006.

[15] Murdoch J. Gabbay and Aad Mathijssen. Capture-Avoiding Substitution as a Nominal Algebra. *Formal Aspects of Computing*, 20(4-5):451–479, June 2008.

[16] Murdoch J. Gabbay and Aad Mathijssen. Nominal Algebra. *Journal of Logic and Computation*, 2009. In press.

[17] Murdoch J. Gabbay and Aad Mathijssen. A nominal axiomatisation of the lambda-calculus. *Journal of Logic and Computation*, 2009. In press.

[18] Murdoch J. Gabbay and Dominic P. Mulligan. One-and-a-halfth Order Terms: Curry-Howard for Incomplete Derivations. In *Proceedings of* 15^{th} *Workshop on Logic, Language and Information in Computation (WoLLIC 2008)*, volume 5110 of *Lecture Notes in Artificial Intelligence*, pages 180–194, 2008.

[19] Murdoch J. Gabbay and Dominic P. Mulligan. One-and-a-halfth Order Terms: Curry-Howard for Incomplete First-Order Logic Derivations Using One-and-a-Halfth Level Terms. *Information and Computation*, 2009. In press.

[20] Murdoch J. Gabbay and Dominic P. Mulligan. Two-and-a-halfth Order Lambda-calculus. *Electronic Notes in Theoretical Computer Science*, 2009. To appear.

[21] Murdoch J. Gabbay and Andrew M. Pitts. A New Approach to Abstract Syntax with Variable Binding. *Formal Aspects of Computing*, 13(3–5):341–363, 2001.

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

[23] Makoto Hamana. Free sigma-monoids: A higher-order syntax with metavariables. In *The Second Asian Symposium on Programming Languages and Systems (APLAS 2004)*, volume 3202 of *Lecture Notes in Computer Science*, pages 348–363, 2004.

[24] Masatomo Hashimoto and Atsushi Ohori. A typed context calculus. *Theoretical Computer Science*, 266(1-2):249–272, 2001.

[25] Dimitri Hendriks and Vincent van Oostrom. Adbmal. In *CADE*, pages 136–150, 2003.

[26] Gueorgui I. Jojgov. Holes with binding power. In *TYPES*, volume 2646 of *Lecture Notes in Computer Science*, pages 162–181. Springer, 2002.

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

[28] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *PLDI*, pages 24–35. ACM, 1994.

[29] Shinn-Der Lee and Daniel P. Friedman. Enriching the lambda calculus with contexts: toward a theory of incremental program construction. In *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 239–250, New York, NY, USA, 1996. ACM.

[30] Pierre Lescanne. From lambda-sigma to lambda-upsilon: a journey through calculi of explicit substitutions. In *POPL*, pages 60–69. ACM, 1994.

[31] Aad Mathijssen. *Logical Calculi for Reasoning with Binding*. PhD thesis, Technische Universiteit Eindhoven, 2007.

[32] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Information and Computation*, 100(1):41–77, 1992.

[33] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[34] Eugenio Moggi, Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. An idealized metaml: Simpler, and more expressive. In *ESOP '99: Proc. of the 8th European Symposium on Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, pages 193–207, 1999.

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

[36] Andrew M. Pitts and Murdoch J. Gabbay. A metalanguage for programming with bound names modulo renaming. In *MPC2000*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer, 2000.

[37] Masahiko Sato, Takafumi Sakurai, and Rod Burstall. Explicit environments. *Fundamenta Informaticae*, 45:1-2:79–115, 2001.
[38] Masahiko Sato, Takafumi Sakurai, and Yukiyoshi Kameyama. A simply typed context calculus with first-class environments. *Journal of Functional and Logic Programming*, 2002(4):359 – 374, 2002.
[39] Masahiko Sato, Takafumi Sakurai, Yukiyoshi Kameyama, and Atsushi Igarashi. Calculi of meta-variables. In *CSL*, volume 2803 of *Lecture Notes in Computer Science*, pages 484–497, 2003.
[40] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, 2002.
[41] Francois Maurel Sylvain Baro. The qnu and qnuk calculi : name capture and control. Technical report, Université Paris VII, 2003. Extended Abstract, Prépublication PPS//03/11//n16.
[42] Makato Takahashi. Parallel reductions in lambda-calculus. *Information and Computation*, 1(118):120–127, 1995.
[43] Terese. *Term Rewriting Systems*. Number 55 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003.
[44] Laurence Tratt. Compile-time meta-programming in converge. Technical Report TR-04-11, Department of Computer Science, King's College London, 2002.
[45] Christian Urban, Andrew M. Pitts, and Murdoch J. Gabbay. Nominal Unification. *Theoretical Computer Science*, 323(1–3):473–497, 2004.
[46] Johan van Benthem. Modal foundations for predicate logic. *Logic Journal of the IGPL*, 5(2):259–286, 1997.
[47] Johan van Benthem. Higher-order logic. In *Handbook of Philosophical Logic, 2nd Edition*, volume 1, pages 189–244. Kluwer, 2001.

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