The lambda-context calculus

Murdoch J. Gabbay Stéphane Lengrand

Heriot-Watt University, St-Andrew's University, Scotland

Abstract

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

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

Email address: Murdoch.Gabbay@gmail.com, sl@cs.st-andrews.ac.uk (Murdoch J. Gabbay Stéphane Lengrand).

Contents

1 Introduction

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

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

(We shall use '=' to denote α-equality of terms.)

Contexts arise often in proofs of meta-properties in functional programming. They have been substantially investigated in papers by Pitts on contextual equivalence between terms in λ-calculi (with global state) [24,27]. This work was about proving programs equivalent in all contexts — **contextual equivalence**. The idea is that two programs, represented by possibly-open λ-terms, are equivalent when one can be exchanged for another in code. That code might have binders which bind variables in the scope of the 'hole' within which we are substituting, whence the need for binders.

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

A context calculus can have applications besides aspects of proofs of contextual equivalence.

Consider formalising mathematics in a logical framework based on Higher-Order Logic (**HOL**) [37]. Typically we have a goal and some assumptions and we want a derivation of one from the other. This derivation may be represented by a λ-term (the *Curry-Howard correspondence*). But the derivation is arrived at by stages in which it is *in*complete.

To the right are two derivations of $A{\Rightarrow}B{\Rightarrow}C, A{\Rightarrow}B \vdash A{\Rightarrow}C$. The bottom one is complete, the top one is incomplete.[1] An issue arises because the right-most $[A]^i$ in the bottom (complete) derivation is *discharged*, which means that we have to be able to instantiate ? in a sub-derivation for an assumption which *will be discharged*. Discharge corresponds in the Curry-Howard correspondence precisely to λ-abstraction, and this instantiation corresponds to capturing substitution. Similar issues arise with existential variables [14, Section 2, Example 3].

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

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

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

Solutions include clever control of substitution and evaluation order [30], types to prevent 'bad' α-conversions [28,16,29], explicit labels on meta-variables [14,18], and more [4, Section 2]. More on this in the Conclusions.

In this paper we present a new calculus of contexts, the 'lambda context calculus' (**LCC**). We took our technical ideas for handling α-equivalence not from the literature on context calculi cited above, but from *nominal unification* [36]. This was designed to manage α-equivalence in the presence of holes, in unification — 'unification of contexts of syntax', in other words. Crudely put, we obtained the LCC by allowing λ-abstraction over the holes and adding β-reduction.

This work has similar goals to the N EWcc previously developed by the first author [9]. The LCC has a simpler and more intuitive set of reduction rules. Indeed there is now only *one* non-obvious side-condition, it is on $(\sigma\mathbf{p})$; 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 N EWcc.

The result is a clean and simple system with a powerful hierarchy of context variables, yet which has 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, a Hindley-Milner type system with principal typings and subject reduction, and an applicative characterisation of contextual equivalence.

[1] This example 'borrowed' from [14].

2 Syntax, freshness, reductions

2.1 Syntax

We suppose a countably infinite set of disjoint infinite **sets of variables** $\mathbb{A}_1$, $\mathbb{A}_2$, ..., 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 1 *The syntax of the lambda context calculus (LCC) is given by:*

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

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.

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

Fig. 1. Level $\mathsf{level}(s)$

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

Fig. 2. Free variables $\mathsf{fv}(s)$

2.2 Levels and Free variables

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

Here $\mathsf{max}(i, j)$ is the greater of i and j, and $\mathsf{max}(i, j, k)$ is the greatest of i, j, and k.

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 ... and $\mathsf{level}(s_n) \leq i$', similarly for '$\mathsf{level}(s_1, \ldots, s_n) < i$'; this will be useful later.

Lemma 3 *If* $\mathsf{level}(s) = 1$ *then* $\mathsf{fv}(s)$ *coincides with the usual notion of 'free variables of' for the untyped* λ*-calculus, if we read* $s[a_1 \mapsto t]$ *as* $(\lambda a_1.s)t$*.*

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

2.3 α-equivalence

Definition 4 *A* **congruence** *on LCC 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 5 *Define an* **(atoms) swapping** $(a_i\ b_i)s$ *action inductively by the rules in Figure 4.*

Note that the definition of the swapping action is absolutely uniform, literally

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

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

$$\frac{s\ R\ s'}{s'\ R\ s} \qquad \frac{s\ R\ s' \quad s'\ R\ s''}{s\ R\ s''}$$

Fig. 3. Rules for a congruence

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

Fig. 4. Rules for swapping

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

Fig. 5. Rules for α-equivalence

swapping a_i and b_i in s without regard to binders et cetera. This is very characteristic of the underlying 'nominal' method of this paper [13,36].

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 6 $\mathsf{fv}((a_i\ b_i)s) = (a_i\ b_i)\mathsf{fv}(s)$ *and* $\mathsf{level}((a_i\ b_i)s) = \mathsf{level}(s)$.

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

Definition 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* $j > i$.

Lemma 8 *If* $\mathsf{level}(t) < j$ *then* $b_j \# \mathsf{fv}(t)$.

Proof We work by induction on the definition of $\mathsf{level}(t)$.

- The case of a_i. $\mathsf{level}(a_i) = i$. So suppose that $i < j$. It follows from the definition of fv that $b_j \# \mathsf{fv}(a_i)$.
- The case of tt'. $\mathsf{level}(tt') = \mathsf{level}(t) \cup \mathsf{level}(t')$. $\mathsf{fv}(tt') = \mathsf{fv}(t) \cup \mathsf{fv}(t')$. The result follows by the inductive hypothesis.
- The case of $\lambda a_i.t$. $\mathsf{level}(\lambda a_i.t) = \mathsf{max}(i, \mathsf{level}(t))$, so $i < j$. $\mathsf{fv}(\lambda a_i.t) \subseteq \mathsf{fv}(t)$. The result follows by the inductive hypothesis.
- The case of $s[a_i \mapsto t]$ is similar.

□

Definition 9 *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 $b_j \in \mathsf{fv}(s)$ for $j > i$. For example $\lambda x.X \neq_\alpha \lambda y.X$. This is consistent with a reading of strong variables as unknown terms with respect to weaker variables.
- We can never α-convert variables to variables of *other* levels.

Lemma 10 *If* s *mentions only variables of level 1, then* α*-equivalence collapses to the usual* α*-equivalence on untyped* λ*-terms (plus an explicit substitution).*

Parenthetical note: The definitions above are descended from the notion of α-equivalence for nominal terms from [36]. In the terminology used here, that paper considered a syntax with a hierarchy with just levels 1 and 2 and no abstraction over variables of level 2. However LCC is weaker in the sense that nominal terms include swappings in the syntax of terms. We use swappings as an operation *on* LCC syntax (Definition 5) but we have not included them in the syntax of the LCC itself.

Extending LCC syntax with swappings is future work. To do that it would help to have a better understanding of LCC models (this paper is purely syntactic) and of freshness $a\#t$. Another paper does explore the notion of freshness in the presence of a hierarchy [10].

Theorem 11 *If* $s =_\alpha s'$ *then* $\mathsf{fv}(s) = \mathsf{fv}(s')$ *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 $b_i \# \mathsf{fv}(s)$ then $\mathsf{fv}(\lambda a_i.s) =_\alpha \mathsf{fv}(\lambda b_i.(b_i\ a_i)s)$.
  Suppose $\mathsf{level}(s) \leq i$. By Lemma 6 $\mathsf{level}((b_i\ a_i)s) \leq i$ as well. Then using Lemma 6 we have

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

  It follows by easy set calculations that

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

  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 $b_i \# \mathsf{fv}(s)$ then $\mathsf{fv}(s[a_i \mapsto t]) = \mathsf{fv}(((b_i\ a_i))s[b_i \mapsto t])$.
  Note that

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

  We use the previous part.

□

Theorem 11 guarantees that we can α-convert without changing the levels or sets of free variables. We use these facts (especially in the proof of confluence when we write 'renaming if necessary') without comment henceforth.

2.4 Reductions

Definition 12 *Define the* **reduction relation** *on terms (modulo* α*-equivalence) inductively by the rules in Figure 6.*

In that figure 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 use the following notation:

- We write $\rightsquigarrow^*$ for the transitive reflexive closure of $\rightsquigarrow$.

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

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

$$
\frac{s \rightsquigarrow s'}{\lambda a_i.s \rightsquigarrow \lambda a_i.s'}\,(\mathbf{R}\lambda)
$$

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

Fig. 6. Reduction rules of the LCC

- 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

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

  (Later in Section 6 we extend reduction with rules for a binder $\mathsf{И}$.)
- 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 $\rightsquigarrow (\mathbf{ruleset})$.

This is all standard [33,1].

We take a moment to note two easy but important technical properties: reductions decrease the level of a term, and its set of free variables.

Lemma 13 *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. □

Lemma 14 *If* $s \rightsquigarrow s'$ *then* $\mathsf{fv}(s') \subseteq \mathsf{fv}(s)$. *As a corollary, if* $s \rightsquigarrow^* s'$ *then* $\mathsf{fv}(s') \subseteq \mathsf{fv}(s)$.

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

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

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

  From Lemma 8 we deduce that $\mathsf{fv}(s) \setminus \{a_i\} = \mathsf{fv}(s)$ and so $\mathsf{fv}(s)$ is a subset of $\mathsf{fv}(s[a_i{\mapsto}t])$.
- Suppose that $\mathsf{level}(s, s', t) \leq i$. Then

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

  The subset inclusion follows by easy calculations on sets.

Other cases are no harder. The inductive argument is straightforward, relying on Lemma 13. The corollary is immediate. □

2.5 Example reductions

The LCC is a λ-calculus with explicit substitutions [20]. 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:

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

  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.**
  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:

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

  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 $(\sigma\mathsf{fv})$ 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 LCC (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 $\lambda X.(X[x{\mapsto}\mathcal{W}])\ \mathcal{W}$ can be bound to X by a substitution arriving from some enclosing context:

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

- **Substitutions as terms** $[x{\mapsto}y]$ 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:

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

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 $i < j$.

2.6 Comments on the side-conditions

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

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

  This is why the side-condition is not $a_i \notin \mathsf{fv}(s)$. For example $x \notin \mathsf{fv}(X) = \{X\}$ and so with a *false* version of the rule with a side-condition using $\notin$

instead of # we have reductions

$$X[x{\mapsto}2][X{\mapsto}x] \overset{(\sigma\mathsf{fv}\mathbf{FALSE})}{\rightsquigarrow} X[X{\mapsto}x] \overset{\sigma a}{\rightsquigarrow} x$$

$$X[x{\mapsto}2][X{\mapsto}x] \overset{(\sigma\sigma)}{\rightsquigarrow} X[X{\mapsto}x][x{\mapsto}2[X{\mapsto}x]] \overset{(\sigma\mathbf{a}),(\sigma\mathsf{fv}\mathbf{FALSE})}{\rightsquigarrow\!\!*} 2$$

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

It is unusual for a garbage collection rule to appear in a calculus of explicit substitutions; usually we can 'push substitutions into a term until they reach variables' so we make do with a rule of the form $b[a{\mapsto}t] \rightsquigarrow b$. In the LCC we cannot always push substitutions into a term until they reach variables, because of the conditions on $(\sigma\mathbf{p})$ and $(\sigma\lambda')$. A version of $(\sigma\mathsf{fv})$ does appear in the literature as Bloo's 'garbage collection' [3].

- 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.

  This condition seems to be fundamental for confluence to work; we have not been able to sensibly weaken it, even if we also change other rules to fix what goes wrong when we do. Here is what happens if we drop the side-condition entirely:

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

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

  The term $(\lambda x.X)[y{\mapsto}x]$ does not reduce (more on that in the Conclusions).
- The side-conditions on $(\sigma\sigma)$, $(\sigma\lambda)$, and $(\sigma\lambda')$ implement that a relatively strong substitution can capture but substitution for variables of the same level avoids capture.

  There is no rule

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

  since that would destroy termination of the part of the LCC without λ — and we have managed to get confluence without it.
- There is no rule permitting a weak substitution to propagate under a *stronger* abstraction, *even if* we avoid capture:

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

Such a rule would cause the following problem for confluence:

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

$$(\lambda Y.(xZ))[x{\mapsto}3][Z{\mapsto}\mathcal{W}] \overset{(\sigma\sigma)}{\rightsquigarrow} (\lambda Y.(xZ))[Z{\mapsto}\mathcal{W}][x{\mapsto}3[Z{\mapsto}\mathcal{W}]]$$
$$\overset{(\sigma\mathsf{fv})}{\rightsquigarrow} (\lambda Y.(xZ))[Z{\mapsto}\mathcal{W}][x{\mapsto}3]$$

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 The substitution action

Definition 15 *Let* **(sigma)** *be equal to the set of rewrite rules* other *than* (β)*, namely,*

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

This is the part of the LCC 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 [3,20] 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 **(sigma)**

We show that **(sigma)**-reduction decreases terms with respect to a well-founded ordering based on mapping LCC terms to first-order terms (no variables, no binders), and using a *lexicographic path ordering* [19,1] on them.

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

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

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:

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

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

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 [19,1] that $\ll$ is a well-founded order on first-order terms satisfying the *subterm property*, i.e. if s is a subterm of t then $s \ll t$.

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

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

Theorem 16 *If* $t \overset{\text{(sigma)}}{\rightsquigarrow} u$ *then* $\overline{t} \gg \overline{u}$.

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:

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

□

Corollary 17 (sigma)*-reduction terminates.*

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

Fig. 7. Substitution on terms of the LCC

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

Fig. 8. The catchily-named 'the star' function

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 18 *Call* s (**sigma**)**-normal** *when* $s \overset{\textbf{(sigma)}}{\not\rightsquigarrow}$.

By Corollary 17, any chain of (**sigma**)-reductions must terminate, and by definition it terminates at a (**sigma**)-normal form.

What does that (**sigma**)-normal form look like?

3.2 Calculating (**sigma**)*-normal forms*

Definition 19 *Define a* **substitution action** $s[a_i{:=}t]$ *by the equalities in Figure 7.*

The precedence of which equality in Figure 7 to use is from top to bottom. Also, 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 $c_i\#\mathsf{fv}(t)$.

Lemma 20 $s[a_i{\mapsto}t] \overset{\textbf{(sigma)}}{\rightsquigarrow^*} s[a_i{:=}t]$.

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 $[a_i \mapsto t]$. □

Lemma 21 *If s and t are (**sigma**)-normal then $s[a_i:=t]$ is (**sigma**)-normal.*

Proof By induction on i and then s. □

Definition 22 *Define s^* inductively by the rules in Figure 8.*

Theorem 23 *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 20). The second is that s^* is a (**sigma**)-normal form, which is proved by a routine induction on s, using Lemma 21. □

4 Confluence

Recall from Definition 15 that (**sigma**) is the set of rules defined by

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

It is convenient to define:

Definition 24 *Let (**beta**) be the set $\{(\beta), (\sigma\lambda), (\sigma\lambda'), (\sigma\mathsf{fv})\}$.*

Note that (**sigma**) $\cup$ (**beta**) is equal to the set of all reduction rules of the LCC.

Note that (**sigma**) $\cap$ (**beta**) is non-empty. We mention the technical reasons for this just after Lemma 39 — it seems to be vital for the proofs to work. Understanding the deeper mathematical reasons for this, if any, is future work.

Theorem 25 *$\rightsquigarrow$ 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 25 occupies this section. Before we give the details we comment on the overall design of the proof.

Roughly speaking there are two standard ways to prove confluence:

(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 $t_2 \Rightarrow u$.

(An example of the method is in Subsection 4.2.)

(2) Define for each term s a *normal 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 [33]. But which to use for the LCC?

It seems that reductions using (β) 'want' method 1 above — but **(sigma)**-reductions 'want' method 2. To prove confluence of the LCC, we split the reduction relation into **(sigma)** and **(beta)**; we prove confluence results by methods 1 and 2 above independently, and then we combine them.

4.1 Confluence of **(sigma)**

Lemmas 27 and 29 will be useful; Lemma 27 in this subsection and Lemma 29 in Subsection 4.3. Since these results are closely related so we have put them side-by-side here.

We need this technical lemma in a moment:

Lemma 26 *If* $\mathsf{level}(t) < j$ *then* $t[b_j{:=}u] = t$.

Proof By Lemma 8 if $\mathsf{level}(t) < j$ then $b_j \# \mathsf{fv}(t)$. The result follows from the definition of $[b_j{:=}u]$. □

Lemma 27 *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 $c_k \# \mathsf{fv}(t)$.
- 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 20 we have $(ss')[a_i{\mapsto}t] \rightsquigarrow^* (ss')[a_i{:=}t]$. By Lemma 13 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 26 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 $c_i\#\mathsf{fv}(t)$. 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 Lemma 29:

Lemma 28 *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. □

Lemma 29 *Suppose that* $\mathsf{level}(s,t,u) \leq i$ *and* $a_i \# \mathsf{fv}(u)$ *(which in view of the condition on levels, means just* $a_i \notin \mathsf{fv}(u)$*). 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 28 that $k < i$. Hence,

$$\begin{aligned} s[c_k{\mapsto}s'][a_i{:=}t][b_i{:=}u] &= s[a_i{:=}t][c_k{:=}s'[a_i{:=}t]][b_i{:=}u] \\ &\overset{\text{Lem.27}}{=} s[a_i{:=}t][b_i{:=}u][c_k{:=}s'[a_i{:=}t][b_i{:=}u]] \\ &\overset{\text{ind. hyp.}}{=} s[b_i{:=}u][a_i{:=}t[b_i{:=}u]][c_k{:=}s'[b_i{:=}u][a_i{:=}t[b_i{:=}u]]] \\ s[c_k{\mapsto}s'][b_i{:=}u][a_i{:=}t[b_i{:=}u]] &= s[b_i{:=}u][c_k{:=}s'[b_i{:=}u]][a_i{:=}t[b_i{:=}u]] \\ &\overset{\text{Lem.27}}{=} s[b_i{:=}u][a_i{:=}t[b_i{:=}u]][c_k{:=}s'[b_i{:=}u][a_i{:=}t[b_i{:=}u]]] \end{aligned}$$

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

$\mathsf{fv}(u)$. 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.

□

We now come back to the confluence of **(sigma)**.

Lemma 30 *(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)* $a_i\#\mathsf{fv}(u)$.

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 $\mathsf{level}(s,s',t) \leq i$.

$$\begin{aligned}((ss')[a_i{\mapsto}t])^* &= (s^*[a_i{:=}t^*])(s'^*[a_i{:=}t^*])\\ ((s[a_i{\mapsto}t])(s'[a_i{\mapsto}t]))^* &= (s^*[a_i{:=}t^*])(s'^*[a_i{:=}t^*]).\end{aligned}$$

(4) Recall that we assume that $i < j$. Using Lemma 27

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

(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)

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

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

□

Lemma 31 $s^*[a_i{\mapsto}t^*] \overset{(\mathbf{sigma})}{\rightsquigarrow^*} (s[a_i{\mapsto}t])^*$.

Proof By definition $(s[a_i{\mapsto}t])^* = s^*[a_i{:=}t^*]$. We use Lemma 20. □

Lemma 32 *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.

In this proof, and in this proof only, we shall write $\rightsquigarrow$ for $\overset{(\mathbf{sigma})}{\rightsquigarrow}$ and $\rightsquigarrow^*$ for $\overset{(\mathbf{sigma})}{\rightsquigarrow^*}$.

- **(Rapp)** Suppose $s \rightsquigarrow s'$ so that $st \rightsquigarrow s't$.
  By inductive hypothesis $s' \rightsquigarrow^* s^*$ and by Theorem 23 $t \rightsquigarrow^* t^*$, so also $s't \rightsquigarrow^* s^*t^* = (st)^*$. The case of **(Rapp′)** is similar.
- **(Rλ)** Suppose $s \rightsquigarrow s'$ so that $\lambda a_i.s \rightsquigarrow \lambda a_i.s'$. By inductive hypothesis $s' \rightsquigarrow^* s^*$, and so $\lambda a_i.s' \rightsquigarrow^* \lambda a_i.(s^*) = (\lambda a_i.s)^*$.
- **(Rσ)** Suppose $s \rightsquigarrow s'$ so that $s[a_i{\mapsto}t] \rightsquigarrow s'[a_i{\mapsto}t]$.
  By inductive hypothesis $s' \rightsquigarrow^* s^*$ and by Theorem 23 $t \rightsquigarrow^* t^*$, so also $s'[a_i{\mapsto}t] \rightsquigarrow^* s^*[a_i{\mapsto}t^*]$. By Lemma 31 $s^*[a_i{\mapsto}t^*] \rightsquigarrow^* (s[a_i{\mapsto}t])^*$.
  The case of **(Rσ')** is similar.
- Now suppose that $s \overset{(\mathbf{sigma})}{\rightsquigarrow} s'$ is derived using one of the rules **(σa)**, **(σc)**, **(σp)**, **($\sigma\sigma$)**, **($\sigma\lambda$)**, or **($\sigma\lambda'$)**. In these cases we use Theorem 23 and the relevant part of Lemma 30.

□

Theorem 33 $\overset{(\mathbf{sigma})}{\rightsquigarrow}$ *is confluent.*

Proof By an easy inductive argument using Lemma 32. □

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

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

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

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

Fig. 9. Parallel reduction relation for the LCC

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

Definition 34 *Inductively define the* **parallel reduction relation** $\Longrightarrow$ *(for* (**beta**)*) by the rules in Figure 9.*

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 35 *(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.
(2) By induction on the derivation of $s \Longrightarrow s'$.
(3) By induction on the derivation of $s \overset{(\mathbf{beta})}{\rightsquigarrow}\ s'$.

□

Corollary 36 *If* $s \Longrightarrow s'$ *then* $\mathsf{fv}(s') \subseteq \mathsf{fv}(s)$ *and* $\mathsf{level}(s') \subseteq \mathsf{level}(s)$.

Proof From Lemma 35 and Lemma 14. □

Lemma 37 $\Longrightarrow$ *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 $s \Longrightarrow s_2$.

- **(Pa)** and **(Pa)**. There is nothing to prove.
- **(Pσ)** and **(Pσ)**.
  $s \Longrightarrow s'$ and $t \Longrightarrow t'$ and also $s \Longrightarrow s''$ and $t \Longrightarrow t''$ so that by **(Pσ)** and **(Pσ)**
  $$s'[a_i \mapsto t'] \Longleftarrow s[a_i \mapsto t] \Longrightarrow s''[a_i \mapsto t''].$$
  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
  $$s'[a_i \mapsto t'] \Longrightarrow s'''[a_i \mapsto t'''] \Longleftarrow s''[a_i \mapsto t''].$$
- **(Pσ)** and **(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 **(Pσ)** and **(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 **(P$\sigma\epsilon$)** for **($\sigma\lambda$)** and **(Pσ)**
  $$(\lambda a_i.s')[b_j \mapsto t'] \Longrightarrow \lambda a_i.(s'''[b_j \mapsto t''']) \Longleftarrow \lambda a_i.(s''[b_j \mapsto t'']).$$
- The case of **(P$\sigma\epsilon$)** for **($\sigma\lambda$)** and **(Pσ)** is similar.
- **(Pσ)** and **(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) $a_i \# u''$ so that by **(Pσ)** and **(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 36 $a_i \# u'''$. 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 38 $\overset{(\mathbf{beta})}{\rightsquigarrow}$ *is confluent.*

Proof By Lemma 35 and Lemma 35 and a standard argument [2]. □

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

Lemma 39 *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 $\mathsf{level}(s,t,u) \leq j$:

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

  This can be closed by:

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

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

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

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

By Corollary 36 $\mathsf{level}(s', t', u') \leq i$. By Lemma 20 and Theorem 23,

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

and by Lemma 29 this is equal to $(s')^*[b_i{:=}u'][a_i{:=}t'[b_i{:=}u']]$.

On the other hand (also using Lemma 20 and Lemma **??**):

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

which closes the divergence above.

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

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

This can be closed by:

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

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

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

We know that $b_j\#\mathsf{fv}(t)$ because $c_i\#\mathsf{fv}(t)$ and $i < j$. We then deduce $b_j\#\mathsf{fv}(t')$ using Corollary 36. We use this to justify the $\Longrightarrow$-rewrite which uses $(\sigma\mathsf{fv})$ in a moment.

This can be closed by:

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

□

We promised to explain why $(\sigma) \cap (\beta) \neq \emptyset$. We can now do so by reference to the details of the proof of Lemma 39.

- $(\sigma\lambda) \in$ (**beta**) and $(\sigma\lambda') \in$ (**beta**) because otherwise the two cases of $(\sigma\mathbf{p})$ and (β) above, would not work.
- $(\sigma\mathsf{fv}) \in$ (**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 40 *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 39 then provides s''' such that $t' \rightsquigarrow^* s'''$ and $s'' \Longrightarrow s'''$. □

We now generalise this lemma even further:

Lemma 41 *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 39 (respectively Theorem 33) provides s''' such that $t' \rightsquigarrow^* s'''$ and $s'' \Longrightarrow s'''$ (respectively $s'' \overset{(\mathbf{sigma})}{\rightsquigarrow^*} s'''$). In the latter case, Lemma 37 (respectively Lemma 40) provides s''' such that $t' \rightsquigarrow^* s'''$ and $s'' \Longrightarrow s'''$ (respectively $s'' \overset{(\mathbf{sigma})}{\rightsquigarrow^*} s'''$). □

We can now prove Theorem 25: *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 $s \rightsquigarrow^* s'' \overset{(\mathbf{sigma})}{\rightsquigarrow} t$ or $s \rightsquigarrow^* s'' \overset{(\mathbf{beta})}{\rightsquigarrow} t$. In both cases, the induction hypothesis provides t'' such that $s'' \rightsquigarrow^* t''$ and $t' \rightsquigarrow^* t''$, and then Lemma 41 provides s''' such that $t \rightsquigarrow^* s'''$, and $t'' \Longrightarrow s'''$ or $t'' \overset{(\mathbf{sigma})}{\rightsquigarrow^*} s'''$, 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 LCC.

For convenience, we use the variables of level 1 in the LCC as variables in our λ-calculus; when we translate the λ-calculus into the LCC, 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 [2].

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

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

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

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

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

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 LCC 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 42 $\mathsf{fv}(e) = \mathsf{fv}([\![e]\!])$.

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 43 $[\![e[x{:=}e']]\!] = [\![e]\!][x{:=}[\![e']\!]]$.

Proof We work by induction on the structure of e.

- $[\![x[x{:=}e']]\!] = [\![e']\!] = x[x{:=}[\![e']\!]]$.
- $$\begin{aligned}[\![(e_1e_2)[x{:=}e']]\!] &= [\![e_1[x{:=}e'](e_2[x{:=}e'])]\!] \\ &= [\![e_1[x{:=}e']]\!][\![e_2[x{:=}e']]\!]) \\ &\overset{ind.hyp.}{=} [\![e_1]\!][x{:=}[\![e']\!]]([\![e_2]\!][x{:=}[\![e']\!]]) \\ &= ([\![e_1]\!][\![e_2]\!])[x{:=}[\![e']\!]] \\ &= [\![e_1e_2]\!][x{:=}[\![e']\!]].\end{aligned}$$
- $$\begin{aligned}[\![(\lambda y.e)[x{:=}e']]\!] &= [\![\lambda y.(e[x{:=}e'])]\!] \\ &= \lambda y.[\![e[x{:=}e']]\!] \\ &= \lambda y.([\![e]\!][x{:=}[\![e']\!]]) \\ &= (\lambda y.[\![e]\!])[x{:=}[\![e']\!]] \\ &\overset{\text{Lemma 42}}{=} [\![\lambda y.e]\!][x{:=}[\![e']\!]].\end{aligned}$$
  Here we assume that $y \notin \mathsf{fv}(e')$.

□

Theorem 44 *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 $x \notin \mathsf{fv}(e')$.

$$\begin{aligned}[\![(\lambda x.e)e']\!] &= (\lambda x.[\![e]\!])[\![e']\!] \\ &\rightsquigarrow [\![e]\!][x{\mapsto}[\![e']\!]] \\ &\overset{\text{Lemma 20}}{\rightsquigarrow^*} [\![e]\!][x{:=}[\![e']\!]]. \\ &\overset{\text{Lemma 43}}{=} [\![e[x{:=}e']]\!].\end{aligned}$$

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 45 (Preservation of strong normalisation) *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 44.

It is a fact [2] 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 LCC 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 LCC 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 LCC and observe that if $[\![V]\!]$ is a normal form then so is $\lambda x.[\![V]\!]$.

□

6 A NEW part for the LCC

6.1 Some NEW rules

LCC 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.

Suppose we *really do* want to bind x in $\lambda x.X$. That is, suppose we want to recover the notion of 'local variable' which the traditional λ-calculus (without a hierarchy of levels) identifies with functional abstraction.

We extend the syntax of the LCC as follows:

$$s, t ::= \ldots \mid \mathcal{N}a_i.t.$$

We extend the definition of level and fv (Definition 2) with clauses

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

We extend the definition of congruence (Definition 4) with a clause

$$\frac{s \; R \; s'}{\mathcal{N}a_i.s \; R \; \mathcal{N}a_i.s'}.$$

We extend the definition of swapping (Definition 5) with a clause

$$(a_i \; b_i)\mathcal{N}c.s = \mathcal{N}(a_i \; b_i)c.(a_i \; b_i)s$$

where c is any atom.

We extend the definition of α-equivalence (Definition 9) with a clause

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

Note the difference that in α-equivalence for λ-abstraction we check $b_i\#\mathsf{fv}(s)$ (Definition 7), and in α-equivalence for $\mathsf{И}$-binding we check $b_i \notin \mathsf{fv}(s)$.

Variables bound by $\mathsf{И}$ rename *regardless* of whether stronger variables are present.

For example take x and y to be variables of level 1, and X and Y to be variables of level 2. Then

$$\lambda x.X \neq_\alpha \lambda y.X \qquad \text{but} \qquad \mathsf{И}x.\lambda x.X =_\alpha \mathsf{И}y.\lambda y.X.$$

We add reduction rules

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

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

$\mathsf{И}$ has behaviour similar to that of π-calculus restriction [21].

- $(\mathsf{И}\mathbf{p})$ and $(\mathsf{И}\sigma)$ are reminiscent of scope-extrusion.
- $(\mathsf{И} \notin)$ is reminiscent of 'garbage-collection'.

Here is an example reduction which exploits $\mathsf{И}$:

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

Compare with a pair of related rewrites in the syntax *without* $\mathsf{И}$:

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

So $\mathsf{И}$ binds and this is separated from the functional abstraction implemented by λ.

LCC syntax permits $\mathsf{И}$ also *not* directly above λ, for example in $\mathsf{И}x.x$. This behaves like a constant symbol and indeed $\mathsf{fv}(\mathsf{И}x.x) = \emptyset$ (though note that level information is preserved; $\mathsf{level}(\mathsf{И}a_i.a_i) = i$). The $\mathsf{И}$ which binds x ensures

that it can never be substituted for:

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

(Recall that we take terms up to α-equivalence when discussing reductions.)

We shall make no use of terms of the form $\text{И}x.x$ in the examples to come. In this paper we are only interested in И to restore α-conversion behaviour to λ-abstracted variables which have been abstracted over a scope containing stronger variables. However, we expect a better understanding of И to be important for future work.

6.2 Some false NEW rules

We do not admit a rule

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

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

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

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 symbol'. Then **(И**$\mathbf{pFALSE)}$ (with the other rules of the LCC) 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**$\sigma\mathbf{FALSE)}$:

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

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

confluence:

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

$$(И X.x)[x{\mapsto}2] \overset{(И\sigma\mathbf{FALSE})}{\rightsquigarrow} И X.(x[x{\mapsto}2]) \overset{(\sigma \mathbf{a})}{\rightsquigarrow} 2.$$

Proofs extend smoothly to the calculus extended with rules for И, including confluence and termination of **(sigma)** extended with the rules for И.

7 Hindley-Milner types

Some basic motivation: a type system is a logic (often a decidable logic) on terms, which allows us to reason on terms without having to evaluate them. For example, in ML if a term types has type integer, by Subject Reduction the term will always have type integer no matter how we evaluate it, and if it reduces to a normal form that normal form will *be* an integer. So there is no 'one' type system; it depends what properties we are interested in.

Hindley-Milner typing [7] is a simple and successful polymorphic type system which underlies functional programming languages such as Erlang [39], ML [23], or Haskell [34]. As such it is both a 'working (functional) programmer's' tool and a starting point for more complex schemes. If the LCC interacts well with it, then a hierarchy of variables can be implemented, at least in principle, as an extension of functional programming as we know it in common practice.

Fix infinitely many **type variables** $\alpha, \beta \in \mathrm{TyVar}$. **Types** and **type schemes** are defined by:

$$\tau ::= \alpha \mid (\tau, \tau) \mid \tau \to \tau \qquad \sigma ::= \tau \mid \forall \alpha.\ \sigma.$$

Let a **type substitution**, we generally write S or T, be a function from type variables α to types τ such that $S\alpha = \alpha$ for all but finitely many α. Type substitutions act on types in the standard way. Write $\tau \preceq \sigma$ when there is some substitution S such that

- $\sigma = \forall \alpha_1.\ \cdots \forall \alpha_n.\ \tau'$ (we shall just write $\sigma = \forall \overline{\alpha}.\ \tau$ for this),
- $S\alpha \neq \alpha$ implies that α occurs in $\overline{\alpha}$ (we may say 'S acts only on variables in $\overline{\alpha}$'), and
- $\tau = S\tau'$.

Also write $tyv\,\tau$ for the type variables appearing in τ.

A **type context** is a finite set of pairs $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ such that if $x_i = x_j$ then $i = j$ for $1 \leq i, j \leq n$. Type contexts may be empty. Γ ranges over **type contexts**.

Then typing rules are as follows:

$$\frac{a_i : \sigma \in \Gamma \quad \tau \preceq \sigma}{\Gamma \vdash a_i : \tau}\ (\mathbf{Tya}) \qquad \frac{\Gamma, a_i : \tau \vdash s : \tau'}{\Gamma \vdash \lambda a_i.s : \tau \to \tau'}\ (\mathbf{Ty}\lambda)$$

$$\frac{\Gamma \vdash s' : \tau' \quad \Gamma, a_i : \forall \overline{\alpha}.\ \tau' \vdash s : \tau \quad \overline{\alpha} = tyv\tau' \setminus tyv\Gamma}{\Gamma \vdash s[a_i \mapsto s'] : \tau}\ (\mathbf{Ty}\sigma)$$

$$\frac{\Gamma, n_j : \alpha \vdash s : \tau \quad n_j, \alpha \notin \Gamma}{\Gamma \vdash \mathsf{И} n_j.s : \tau}\ (\mathbf{Ty}\mathsf{И}) \qquad \frac{\Gamma \vdash s : \tau \to \tau' \quad \Gamma \vdash t : \tau}{\Gamma \vdash st : \tau'}\ (\mathbf{Typ})$$

The notation $n_j, \alpha \notin \Gamma$ in (**Ty**$\mathsf{И}$) is shorthand for 'n_j and α do not occur anywhere in the syntax of Γ'. The notation $tyv\Gamma$ in (**Ty**σ) is shorthand for the type variables occurring free in Γ. For example

$$tyv\{c : \forall\alpha.\ \alpha \to \beta\} = \{\beta\}.$$

Abusing notation write $\Gamma \setminus c$ for the typing context obtained from Γ by removing from it $c : \sigma'$ if there is some σ' such that $c : \sigma' \in \Gamma$.

Write $\Gamma, c : \sigma$ for $(\Gamma \setminus c) \cup \{c : \sigma\}$.

We have to be a little careful proving Lemma 46. In 'normal' λ-calculus we can always rename a λ-abstracted variable to avoid any clash with names in the type context, if this is convenient. In LCC this is not the case; for example if x has level 1 and X has level 2, then we *cannot* rename x in $\lambda x.X$. This difficulty is easily surmounted with just a little care, as follows:

If S is a finite set of variables write $\Gamma|_S$ for the typing context Γ restricted to S; for example $\{c : \tau, c' : \tau'\}|_{\{a,c\}} = \{c : \tau\}$.

Lemma 46 (Weakening) *$\Gamma \vdash s : \tau$ if and only if $\Gamma|_{\mathsf{fv}(s)} \vdash s : \tau$.*

As a corollary, if $c \notin \mathsf{fv}(s)$ then if $\Gamma \vdash s : \tau$ then $\Gamma, c : \sigma' \vdash s : \tau$. Here c is any variable.

Proof We prove the first part by induction on the derivation; the corollary is an easy consequence of it:

- The case of (**Tya**). ...is routine.

- The case of (**Ty**λ). Suppose $\Gamma, a_i : \tau \vdash s : \tau'$. By inductive hypothesis

$$(\Gamma, a_i : \tau)|_{\mathsf{fv}(s)} \vdash s : \tau'.$$

By some easy set calculations, and using the corollary if $a_i \notin \mathsf{fv}(s)$,

$$\Gamma|_{\mathsf{fv}(s)\setminus\{a_i\}}, a_i : \tau \vdash s : \tau'.$$

We may then use (**Ty**λ) to deduce that

$$\Gamma|_{\mathsf{fv}} \vdash \lambda a_i.s : \tau \to \tau'$$

as required.
- The case of (**Ty**σ). Suppose that $\Gamma \vdash s' : \tau'$ and $\Gamma, a_i : \forall\overline{\alpha}.\ \tau' \vdash s : \tau$ where $\overline{\alpha} = tyv\,\tau' \setminus tyv\,\Gamma$. By inductive hypothesis

$$\Gamma|_{\mathsf{fv}(s')} \vdash s' : \tau' \qquad \text{and} \qquad (\Gamma, a_i : \forall\overline{\alpha}.\ \tau')|_{\mathsf{fv}(s)} \vdash s : \tau.$$

By elementary calculations on sets and using the corollary if $a_i \notin \mathsf{fv}(s)$ we deduce that

$$\Gamma|_{(\mathsf{fv}(s)\setminus\{a_i\})\cup\mathsf{fv}(s')} \vdash s' : \tau' \quad \text{and} \quad \Gamma|_{(\mathsf{fv}(s)\setminus\{a_i\})\cup\mathsf{fv}(s')}, a_i : \forall\overline{\alpha}.\ \tau' \vdash s : \tau.$$

The result follows observing that $\mathsf{fv}(s[a_i{\mapsto}s']) = (\mathsf{fv}(s) \setminus \{a_i\}) \cup \mathsf{fv}(s')$.
- The case of (**Ty**$\mathcal{N}$). Suppose that $\Gamma, n_j : \alpha \vdash s : \tau$ where n_j and α do not occur in Γ. By inductive hypothesis $(\Gamma, n_j : \alpha)|_{\mathsf{fv}(s)} \vdash s : \tau$. By elementary set calculations and using the corollary if $n_j \notin \mathsf{fv}(s)$ we deduce that $\Gamma|_{\mathsf{fv}(s)}, n_j : \alpha \vdash s : \tau$. The result follows.
- The case of (**Typ**) ...is easy.

□

Theorem 47 (Soundness) *If* $\Gamma \vdash s : \tau$ *and* $s \rightsquigarrow s'$ *then* $\Gamma \vdash s' : \tau$.

Proof We check the rules for $\rightsquigarrow$.

- The case of (β). Suppose that $\Gamma \vdash (\lambda a_i.s)t : \tau'$ is derivable. By following the typing derivation rules we see that for some τ,

$$\Gamma, a_i : \tau \vdash s : \tau' \qquad \text{and} \qquad \Gamma \vdash t : \tau$$

must be derivable. It is then easy to use (**Ty**σ) to deduce that $\Gamma \vdash s[a_i{\mapsto}t] : \tau$, since $\tau \preceq \forall\overline{\alpha}.\ \tau$ for any $\overline{\alpha}$.
- The case of (**σa**) ...is easy.
- The case of (**σfv**). Suppose that $\Gamma \vdash s[a_i{\mapsto}t] : \tau$ and suppose that $a_i \# \mathsf{fv}(s)$. It follows that $a_i \notin \mathsf{fv}(s)$. By following the typing derivation rules we see that

$$\Gamma \vdash t : \tau' \qquad \Gamma, a_i : \forall\overline{\alpha}.\ \tau' \vdash s : \tau$$

where $\overline{\alpha} = tyv\tau' \setminus tyv\Gamma$. Now if $a_i \notin \mathsf{fv}(s)$ then by Lemma 46 also $\Gamma \vdash s : \tau$ and we are done.

- The case of ($\sigma\mathbf{p}$). Suppose that $\Gamma \vdash (s's)[a_i{\mapsto}t] : \tau'$. Then for some τ'' is must be that

$$\Gamma, a_i : \forall\overline{\alpha}.\ \tau'' \vdash s : \tau \quad \Gamma, a_i : \forall\overline{\alpha}.\ \tau'' \vdash s' : \tau \to \tau' \quad \Gamma \vdash t : \tau''.$$

Here $\overline{\alpha} = tyv\tau'' \setminus tyv\Gamma$. Using ($\sigma\mathbf{p}$) it is immediate that

$$\Gamma \vdash (s[a_i{\mapsto}t])(s'[a_i{\mapsto}t]) : \tau'.$$

- The case of ($\sigma\sigma$). Suppose $\Gamma \vdash s[a_i{\mapsto}t][b_j{\mapsto}u] : \tau$. Then for some τ' and τ'',

$$\Gamma, b_j : \forall\overline{\alpha''}.\ \tau'', a_i : \forall\overline{\alpha'}.\ \tau' \vdash s : \tau \qquad \Gamma, b_j : \forall\overline{\alpha''}.\ \tau'' \vdash t : \tau' \qquad \Gamma \vdash u : \tau''.$$

Here

$$\overline{\alpha''} = tyv\tau'' \setminus tyv\Gamma \quad \text{and} \quad \overline{\alpha'} = tyv\tau' \setminus tyv(\Gamma, \forall\overline{\alpha''}.\ \tau'').$$

It is not hard to calculate that $\overline{\alpha'} = \overline{\alpha''}$, so we write $\overline{\alpha}$ for both henceforth. From this we can calculate that

$$\Gamma, a_i : \forall\overline{\alpha}.\ \tau' \vdash s[b_j{\mapsto}u] : \tau \qquad \text{and} \qquad \Gamma \vdash t[b_j{\mapsto}u] : \tau'$$

and from this we can conclude that

$$\Gamma \vdash s[b_j{\mapsto}u][a_i{\mapsto}t[b_j{\mapsto}u]] : \tau$$

as required.

- The cases of ($\sigma\lambda$) and ($\sigma\lambda'$). Suppose $\Gamma \vdash (\lambda n_j.t)[a_i{\mapsto}u] : \tau \to \tau'$ is derivable. By following the typing derivation rules we see that

$$\Gamma \vdash u : \tau'' \quad \text{and} \quad \Gamma, a_i : \forall\overline{\alpha}.\ \tau'', n_j : \tau \vdash t : \tau'$$

must be derivable, where $\alpha = tyv\tau'' \setminus tyv\Gamma$. Without loss of generality we rename elements of α to be disjoint from $tyv\tau$. It is now not hard to derive $\Gamma \vdash \lambda n_j.(t[a_i{\mapsto}u]) : \tau \to \tau'$, using Lemma 46 to weaken $\Gamma \vdash u : \tau''$ to $\Gamma, n_j : \tau \vdash u : \tau''$.

- The case of ($И\mathbf{p}$). Suppose that $\Gamma \vdash (И n_j.s)t : \tau'$ is derivable where (renaming n_j if necessary) we suppose n_j does not occur in t or Γ. Then

$$\Gamma, n_j : \alpha \vdash s : \tau \to \tau' \qquad \text{and} \qquad \Gamma \vdash t : \tau$$

are derivable, where $\alpha \notin \Gamma$. By Lemma 46 also $\Gamma, n_j : \alpha \vdash t : \tau$ is derivable. It is now easy to derive $\Gamma \vdash И n_j.(st) : \tau'$.

- The cases of ($И\sigma$) and ($И \notin$) are easy.

□

We use the following technical lemma in Theorem 49:

Lemma 48 *If* $\Gamma \vdash s : \tau$ *then* $S\Gamma \vdash s : S\tau$.

Proof By induction on the derivation of $\Gamma \vdash s : \tau$. □

Write $(S', \tau') \preceq (S, \tau)$ when there is some T' such that

- $S' = T'S$ (type substitutions are written prefix, so $T'S$ is 'S followed by T''), and
- $\tau' = T'\tau$.

A **type problem** is a pair $(\Gamma \vdash_? s)$. A **solution** to $(\Gamma \vdash_? s)$ is a pair (S, τ) such that $S\Gamma \vdash s : \tau$. A **principal solution** is a solution (S, τ) which is maximal amongst all solutions to $(\Gamma \vdash_? s)$ in the ordering given by $\preceq$.

Theorem 49 *If* $\Gamma \vdash_? s$ *has a solution, then it has a principal solution.*

Proof A principal solution (S, τ) is calculated by the algorithm below — the rules are read bottom-up, and we write $(\Gamma \vdash_{\mathsf{sol}} s)$ for the pair (S, τ) which is being calculated:

$$\frac{(\Gamma, a_i : \alpha \vdash_{\mathsf{sol}} s) = (S, \tau)}{(\Gamma \vdash_{\mathsf{sol}} \lambda a_i.s) = (S, S\alpha \to \tau)} \qquad \frac{\begin{array}{c}(\Gamma \vdash_{\mathsf{sol}} s) = (S, \tau) \qquad S''{=}mgu(S'\tau, \tau' \to \alpha) \\ (S\Gamma \vdash_{\mathsf{sol}} s') = (S', \tau') \;\; \alpha{\notin}S, S', \Gamma, s, s'\end{array}}{(\Gamma \vdash_{\mathsf{sol}} ss') = (S''S'S, S''\alpha)}$$

$$\frac{\begin{array}{c}(\Gamma \vdash_{\mathsf{sol}} s') = (S', \tau') \quad \overline{\alpha}{=}tyv\tau'\backslash tyv(S', \Gamma) \\ (S'\Gamma, a_i : \forall\overline{\alpha}.\ \tau' \vdash_{\mathsf{sol}} s) = (S, \tau)\end{array}}{(\Gamma \vdash_{\mathsf{sol}} s[a_i{\mapsto}s']) = (SS', \tau)}$$

$$\frac{(\Gamma, a_i : \alpha \vdash_{\mathsf{sol}} s) = (S, \tau) \quad \alpha \notin \Gamma}{(\Gamma \vdash_{\mathsf{sol}} \mathsf{И}a_i.s) = (S, \tau)} \qquad \frac{a_i : \forall\overline{\alpha}.\ \tau \in \Gamma}{(\Gamma \vdash_{\mathsf{sol}} a_i) = (\mathbf{Id}, \tau)}$$

In the rule for a_i, **Id** is the identity type substitution, which is such that $\mathbf{Id}\alpha = \alpha$.

In the rule for application $mgu(S'\tau, \tau' \to \alpha)$ is the **most general unifier** of $S'\tau$ and $\tau' \to \alpha$; that is it is a substitution such that $S''S'\tau = S''(\tau' \to \alpha)$ (a *unifier* of $S'\tau$ and $\tau' \to \alpha$) which is *msot general* in the sense that all other unifiers are of the form $S'''S''$ for some S'''. This is standard, see elsewhere [7]. The *mgu* need not exist, for example $mgu(\alpha', \alpha' \to \alpha)$ does not exist. It is part of the condition for using the rule for application above, that $mgu(S'\tau, \tau' \to \alpha)$ does exist. It is a fact that if a unifier of two types exists, then a most general unifier exists.

In the rule for И, consistent with previous usage $\alpha \notin \Gamma$ means 'α does not occur anywhere in the syntax of Γ'.

The typing rules and principal types algorithm above are identical to those of the standard Hindley-Milner; we give our explicit substitution exactly the rules for the ML let construct. [2]

The proofs transfer to this setting unchanged from the original presentations [7,6], because the definitions are identical — except for the case of И, which is new.

It is routine to show by induction that if $(\Gamma \vdash_{\mathsf{sol}} s)$ exists then it is a solution to $\Gamma \vdash_? s$.

It remains to show that if a solution to $(\Gamma \vdash_? s)$ exists, then the algorithm for calculating $(\Gamma_{\mathsf{sol}} s)$ calculates a principal solution. We consider only the cases of a_i and $Иa_i.s$:

- The case of a_i. Suppose that

  $$S'\Gamma \vdash a_i : \tau'.$$

  By the form of the typing rules it must be that $a_i : \forall\overline{\alpha}.\ \tau \in \Gamma$ for some $\overline{\alpha}$ and τ such that $\tau' = S'\tau$. Here we assume (renaming if necessary) that S' acts trivially on $\overline{\alpha}$ and that no variable in $\overline{\alpha}$ occurs free in Γ. We take $T = S'$, and we are done.
- The case of $Иa_i.s$. Suppose that

  $$S'\Gamma \vdash Иa_i.s : \tau'.$$

  By the form of the typing rules it must be that

  $$S'\Gamma, a_i : \alpha \vdash s : \tau'.$$

  By inductive hypothesis $(\Gamma \vdash_{\mathsf{sol}} s)$ calculates a principal solution, write it (S, τ), so there is some T such that $TS = S'$ and $TS\tau = \tau'$, and we are done.

□

In conclusion, the type system is almost identical to a standard Hindley-Milner type system for polymorphic types. The typing rule for explicit substitutions is identical to that of ML let. The hierarchy of variables does not damage the type system in any way. There also seems no *a priori* reason that LCC would not accommodate any other type systems developed for functional programming.

[2] This is one of those ideas which is instantly obvious — in retrospect.

8 Applicative characterisation of contextual equivalence

As our final theoretical investigation we consider contextual equivalence of the LCC.

8.1 Programs, contexts, evaluations, and equivalences

It is convenient to introduce a constant $\top$ such that $\top \not\rightsquigarrow$.

Definition 50 *Call s a* **program** *when* $\mathsf{fv}(s) = \emptyset$.

As is our convention, take x to have level 1 and X to have level 2. For example:

- x and $\lambda x.X$ are not programs; $\mathsf{fv}(x) = \{x\}$ and $\mathsf{fv}(\lambda x.X) = \{X\}$.
- $\lambda x.x$ and $\lambda x.(X[X{\mapsto}x])$ are programs.

If s is a program write

$$s\searrow \quad \text{when} \quad s \rightsquigarrow^* \top$$

and say that s **evaluates**.

Call C a **context** when

- C is a program, and
- $C = \lambda d_l.D$ where $l = \mathsf{level}(D)$ and d_l is the *only* variable of level l in D.

We may abuse notation and call D a **context**.

For example:

- $\lambda X.X$, $\lambda X.\lambda x.(xX)X$, and $\lambda X.\top$ are contexts.
- $\lambda X.x$ is not a context because it is not a program; $\mathsf{fv}(\lambda X.x) = \{x\}$.
- $\lambda x.\lambda X.Xx$ is not a context.
- $\lambda X.\lambda Y.XY$ is not a context.
- $\lambda X.\lambda y.XXy$ is a context.

An **equivalence relation** is a transitive symmetric reflexive relation. Call an equivalence relation $\mathcal{X}$ on programs **contextual** when

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

Here C ranges over contexts and s and t range over programs. Write $=_{ctx}$ for the greatest contextual equivalence, which (abusing notation) we call **contextual equivalence**. We discuss this definition below.

Call an equivalence relation $\mathcal{P}$ on programs **applicative** when

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

Here s, t, and u range over programs. Write $=_{ap}$ for the greatest applicative equivalence and abuse notation calling it **applicative equivalence**.

The main result of this section is

Theorem 51 *$=_{ctx}$ and $=_{ap}$ are equal.*

. . . but we need technical machinery to prove it.

Lemma 52 *Write $s \leftrightarrow t$ for the least equivalence relation containing $\rightsquigarrow$. Then*

$$\leftrightarrow \subseteq =_{ap} \qquad \textit{and} \qquad \leftrightarrow \subseteq =_{ctx}.$$

Also, $\top s_1 \ldots s_n =_{ap} \top t_1 \ldots t_n$ always, for $n > 0$.

Proof Suppose $s \leftrightarrow t$. By confluence Theorem 25

$$su{\searrow} \qquad \text{if and only if} \qquad tu{\searrow},$$

and similarly for Cs and Ct.

For the second part we examine the reduction rules and observe that $\top s_1 \ldots s_n{\searrow}$ is impossible unless $n = 0$. □

When we define $=_{ctx}$ we consider C *applied* to s. From the first part of the technical lemma above and from the definition of context, this is clearly equivalent to a definition in more traditional form $D[d_l{\mapsto}s]$.

In the proofs below we tend to use traditional notation of $D[d_l{\mapsto}s]$ rather than $(\lambda d_l.D)s$.

We can also characterise $=_{ctx}$ as the greatest *congruence* (Definition 4; extended with the rule for $\mathsf{И}$ from Subsection 6.1) such that

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

We do not explore this further because the definition of $=_{ctx}$ which we use is more convenient for the proof-method to follow.

8.2 Proof that $=_{ctx}$ equals $=_{ap}$

Suppose s and t are programs and $s =_{ctx} t$. We want to show that $s =_{ap} t$.

By the coinductive principle by which $=_{ap}$ was defined, it suffices to show that if $s =_{ctx} t$ then:

- If $s\searrow$ then $t\searrow$.
- For any program u it is the case that $su =_{ctx} tu$.

The first part follows directly from our assumption that $s =_{ctx} t$.

Also for any program u, the term $(d_l u)$ is a context (where d_l is a variable which we choose stronger than any variable in s or u). By assumption $(d_l u)[d_l{\mapsto}s] =_{ctx} (d_l u)[d_l{\mapsto}t]$. To take the next step and reduce these to su and tu we need Lemmas 53 and 54.

Lemma 53 *If* $l > \mathsf{level}(s)$ *then* $d_l \# \mathsf{fv}(s)$.

Proof By an easy induction on the definition of $\mathsf{level}(s)$ (Definition 2). □

Lemma 54 *Suppose s and u are any terms (we only care about the case that s and u are programs). Suppose further that d_l is a variable such that $l > \mathsf{level}(s,u)$. Then*

$$(d_l u)[d_l{\mapsto}s] \rightsquigarrow^* su.$$

Proof The reduction path is as follows:

$$\begin{aligned}(d_l u)[d_l{\mapsto}s] &\overset{(\sigma\mathbf{p})}{\rightsquigarrow} (d_l[d_l{\mapsto}s])(u[d_l{\mapsto}s])\\ &\overset{(\sigma\mathbf{a})}{\rightsquigarrow} s(u[d_l{\mapsto}s])\\ &\overset{(\sigma\mathsf{fv})}{\rightsquigarrow} su.\end{aligned}$$

For the reduction with $(\sigma\mathsf{fv})$ we use Lemma 53. □

By Lemma 54 $(d_l u)[d_l{\mapsto}s] \rightsquigarrow^* su$ and $(d_l u)[d_l{\mapsto}t] \rightsquigarrow^* tu$. By Lemma 52 we have that $su =_{ctx} tu$, and we are done.

Now suppose that $s =_{ap} t$. We want to prove that $s =_{ctx} t$. By the coinductive principle by which $=_{ctx}$ was defined, it suffices to show that if $s =_{ap} t$ then:

- If $s\searrow$ then $t\searrow$.
- For any context D it is the case that $Ds =_{ap} Dt$.

We work by induction on the tuple (oc_D, nf_D, si_D) where

- oc_D is the number of occurrences of d_l in the normal form of D and ω otherwise (=the first uncountable ordinal; if D has no normal form there will be nothing to prove),
- nf_D is the least number of $\rightsquigarrow$-reductions to reduce D to its normal form and ω otherwise, and

- si_D is the size of D,

proving that

$$\{(И as.D[d_l \mapsto s], И as.D[d_l \mapsto t]) \mid D, d_l, as\}$$

is a contextual relation, where as varies over possibly empty lists of variables no stronger than d_l. We work by cases on the form of D.

In the the cases below we may implicitly use the first part of Lemma 52 along with confluence and the inductive hypothesis, to suppose that D is in $\rightsquigarrow$-normal form. We also silently strip leading $И$s, writing for example d_l for $И a.d_l$.

- Suppose $D = d_l$. Then $d_l[d_l \mapsto s] \leftrightarrow s =_{ap} t \leftrightarrow d_l[d_l \mapsto t]$. We use the first part of Lemma 52 and the fact that $=_{ap}$ is an equivalence.
- Suppose $D = (\lambda n_j.D')$. Then $D[d_l \mapsto s]$ and $D[d_l \mapsto t]$ cannot evaluate to $\top$ so there is nothing to prove.
- Suppose $D = D'D''$, and suppose d_l occurs in D' and D''. We reason as follows:

$$\begin{aligned}
(D'D'')[d_l \mapsto s] &\leftrightarrow (D'[d_l \mapsto s])(D''[d_l \mapsto s]) \\
&=_{ap} (D'[d_l \mapsto t])(D''[d_l \mapsto s]) && (1)\\
&\leftrightarrow (\lambda x_m.x_m D'')[d_l \mapsto s](D'[d_l \mapsto t]) && (2)\\
&=_{ap} (\lambda x_m.x_m D'')[d_l \mapsto t](D'[d_l \mapsto t]) && (3)\\
&\leftrightarrow (D'D'')[d_l \mapsto t] && (4)
\end{aligned}$$

  Here we choose x_m fresh and stronger than d_l. We justify the steps of this reasoning, which we numbered on the right above:

(1) D' has fewer instances of d_l than $D = D'D''$ since we assumed that d_l also appears in D''. So we use the inductive hypothesis for D' to deduce that $D'[d_l \mapsto s] =_{ap} D'[d_l \mapsto t]$. By assumption $=_{ap}$ is closed under application on the right, and so also $(D'[d_l \mapsto s])(D''[d_l \mapsto s]) =_{ap} (D'[d_l \mapsto t])(D''[d_l \mapsto s])$.

(2) $(\lambda x_m.(x_m D''))[d_l \mapsto s](D'[d_l \mapsto t]) \rightsquigarrow D'D''$ is easy to verify.

(3) By the inductive hypothesis and the fact that $=_{ap}$ is closed under application on the right.

(4) Reversing the reasoning above.

- Suppose $D = D'D''$, and suppose d_l occurs in D' but not D''. We reason as follows:

$$\begin{aligned}
(D'D'')[d_l \mapsto s] &\leftrightarrow (D'[d_l \mapsto s])D'' \\
&=_{ap} (D'[d_l \mapsto t])D'' \\
&\leftrightarrow (D'D'')[d_l \mapsto t]
\end{aligned}$$

- Suppose $D = D'D''$, and suppose d_l does not occur in D' and may or may not occur in D''. We have supposed that D' is a normal form with no

unabstracted variables. If D' is a λ-abstraction then $D'D''$ reduces with (β) so we use the inductive hypothesis. Otherwise, $D'D''$ cannot ever evaluate and there is nothing to prove.

- Suppose $D = D'[n_j \mapsto D'']$. If this is not a normal form, we reduce and use the inductive hypothesis.

  If this is a normal form we reason by cases. If d_l occurs in D' and D'', or in D' and not in D'', we can proceed as we did for applications above.

  Suppose d_l does not occur in D' and does occur in D''. We have supposed this is a normal form, so we work by cases:

  · $D' = \lambda e.E$ for some e and E such that e is not weaker than n_j (so that $(\sigma\lambda)$ does not apply). Then $D'[n_j \mapsto D''][d_l \mapsto u]$ cannot possibly evaluate to $\top$ for any u, so there is nothing to prove.

  · $D' = \mathsf{N}e.E$. We are free to α-convert e and therefore we can use $(\mathsf{N}\sigma)$ to reduce $D'[n_j \mapsto D'']$, contradicting our assumption that $D'[n_j \mapsto D'']$ is a normal form.

  · $D' = n_j$. $n_j[n_j \mapsto D'']$ is not a normal form, because of $(\sigma\mathbf{a})$.

  · $D' = c_k$ for c_k not stronger than n_j. $c_k[n_j \mapsto D'']$ is not a normal form, because of $(\sigma\mathsf{fv})$.

  · $D' = c_k$ for c_k stronger than n_j and (by assumption) not equal to d_l. It is not possible for $c_k[n_j \mapsto D''][d_l \mapsto u]$ to evaluate to $\top$ for any u, so there is nothing to prove.

  · The case of $D' = D'_1[n'_{j'} \mapsto D'_2]$ is similar.

That concludes the proof of Theorem 51.

9 Related work, conclusions, and future work

In a previous conference paper we presented the NEW calculus of contexts [9]. The LCC of this journal paper updates and improves that that work. The LCC is simpler than the NEWcc. Compare the side-condition of $(\sigma\mathbf{a})$ (there is none) with that of $(\sigma\mathbf{a})$ from [9]. 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 [9]. 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 $(\sigma\mathbf{p})$.

But there is a price: the LCC has fewer reductions. Notably $(\sigma\lambda')$ will not reduce $(\lambda a_i.s)[c_k \mapsto u]$ where $k < i$; a rule $(\sigma\lambda')$ in [9] does. However that stronger version seems to be a major source of complexity. Do we *miss* the extra reductions? An insight that was necessary to make this paper is that certain reductions which we thought were important, could be dropped.

Still, it is 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.

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 [36]. A problem is that we do not yet understand the theory of *swappings* for strong variables; the underlying Fraenkel-Mostowski sets model [13] only has (in the terminology of this paper) one level of variable. A semantic model of the hierarchy of variables would be useful and this is current work.

In this paper we cannot deduce $x\#\mathsf{fv}(\lambda x.X)$ even though for every *instance* this does hold (for example $x\#\lambda x.x$ and $x\#\lambda x.y$). Hierarchical nominal rewriting [10] has a more powerful notion of freshness which can prove the equivalent of $x\#\mathsf{fv}(\lambda x.X)$. Note that hierarchical nominal rewriting does 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 [25] or the style of a sequent calculus for Nominal Logic by Cheney [5].

Finally, we cannot reduce $X[x{\mapsto}2][y{\mapsto}3]$ to $X[y{\mapsto}3][x{\mapsto}2]$. Other work [12] gives an *equational* system which can do this, and more.

Desirable and nontrivial meta-properties of the λ-calculus survive in the LCC. The LCC is confluent. It supports a type system in Hindley-Milner style making it possible, in principle at least, to envisage an extension of ML or Haskell with meta-variables based on the LCC's notion of strong and weak variables. Also, contextual equivalence coincides with applicative equivalence — a full treatment will be in a longer paper. This is a surprising result considering that substitution in the LCC can capture.

More related work (not using nominal techniques). The calculi of contexts λm and $\lambda\mathcal{M}$ [30] also have a hierarchy of variables. They use carefully-crafted scoping conventions to manage problems with α-conversion. Other work [28,16,29] uses a type system; connections with this work are unclear. λc of Bognar's thesis contains [4, Section 2] an extensive literature survey on the topic of context calculi.

A separation of abstraction λ and binding $\mathsf{И}$ appears in one other (unpublished) work we know of [32], where they are called q and ν. In this vein there is [17], which manages scope explicitly in a completely different way, just for the fun. Finally, the reduction rules of $\mathsf{И}$ look remarkably similar to π-calculus restriction [21], and it is probably quite accurate to think of $\mathsf{И}$ as a 'restriction in the λ-calculus'.

Hamana takes a semantic approach to meta-variables [15]. We have not developed the semantic theory of the LCC. 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 [13]). 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 [20] for a survey. Our treatment of substitution is simple-minded but still quite subtle because of interactions with the rest of the language. We note that the translation of possibly open terms of the untyped λ-calculus into the LCC preserves strong normalisation. One reduction rule, $(\sigma\mathsf{fv})$, is a little unusual amongst such calculi, though it appears as Bloo's 'garbage collection' [3].

The look and feel of the LCC is squarely that of a λ-calculus with explicit substitutions. All the real cleverness has been isolated in the side-condition of $(\sigma\mathbf{p})$; other side-conditions are obvious given an intuition that strong variables can cause capturing substitution (in the NEWcc [9] complexity spilled over into other rules and into a logic for freshness). M is only necessary when variables of different strengths occur, and the hierarchy of variables only plays a rôle 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 [38], along with suitable equalities and/or derivation rules. Because the LCC admits a Hindley-Milner style type system it certainly admits a simple type system. There should be no problem with writing down a 'context higher-order logic'. This takes the LCC in the direction of calculi of contexts for incomplete proofs [18,14], 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 LCC 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 LCC *cannot* express, but they might be accommodated with a relatively straightforward

extension.

As discussed, the LCC 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

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

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 LCC based on atom from [11] may be possible and useful.

The LCC 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 [8] on *dynamic binding*. We do believe that the LCC 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 [22,26], Template Haskell [31], and Converge [35], 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 LCC 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 LCC to make it into a programming language, the deeper problem is that the LCC has no first-class construct to promote variables between levels. Adding this extension is interesting future work.

In conclusion: the LCC of this paper is simple, 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 LCC does that.

References

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

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

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

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

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

[6] Luis Damas, *Type assignment in programming languages*, Ph.D. thesis, University of Edinburgh, 1985.

[7] Luis Damas and Robin Milner, *Principal type-schemes for functional programs*, POPL '82: Proc. of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, 1982, pp. 207–212.

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

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

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

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

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

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

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

[15] M. Hamana, *Free sigma-monoids: A higher-order syntax with metavariables*, The Second Asian Symposium on Programming Languages and Systems (APLAS 2004), LNCS, vol. 3202, 2004, pp. 348–363.

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

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

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

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

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

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

[22] Eugenio Moggi, Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard, *An idealized metaml: Simpler, and more expressive*, ESOP '99: Proc. of the 8th European Symposium on Programming Languages and Systems, LNCS, vol. 1576, 1999, pp. 193–207.

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

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

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

[26] A. M. Pitts and T. Sheard, *On the denotational semantics of staged execution of open code*, Submitted, 2004.

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

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

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

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

[31] Tim Sheard and Simon Peyton Jones, *Template metaprogramming for Haskell*, ACM SIGPLAN Haskell Workshop 02 (Manuel M. T. Chakravarty, ed.), ACM Press, 2002, pp. 1–16.

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

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

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

[35] Laurence Tratt, *Compile-time meta-programming in converge*, Tech. Report TR-04-11, Department of Computer Science, King's College London, 2002.

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

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

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

[39] Robert Virding, Claes Wikström, and Mike Williams, *Concurrent programming in ERLANG*, 2 ed., Prentice Hall, 1996.