Reduction in a linear lambda-calculus with applications to operational semantics

Citation for published version:

Digital Object Identifier (DOI):
10.1007/978-3-540-32033-3_17

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Term Rewriting and Applications

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Reduction in a Linear Lambda-calculus
with Applications to Operational Semantics

Alex Simpson
LFCS, School of Informatics
University of Edinburgh, UK

Abstract. We study beta-reduction in a linear lambda-calculus derived from
Abramsky’s linear combinatory algebras. Reductions are classified depending
on whether the redex is in the computationally active part of a term (“surface”
reductions) or whether it is suspended within the body of a thunk (“internal”
reductions). If surface reduction is considered on its own then any normalizing
term is strongly normalizing. More generally, if a term can be reduced to surface
normal form by a combined sequence of surface and internal reductions then ev-
ery combined reduction sequence from the term contains only finitely many sur-
face reductions. We apply these results to the operational semantics of Lily, a
second-order linear lambda-calculus with recursion, introduced by Bierman, Pitts
and Russo, for which we give simple proofs that call-by-value, call-by-name and
call-by-need contextual equivalences coincide.

1 Introduction

The language Lily was introduced by Bierman, Pitts and Russo in [3]. It is a typed
lambda-calculus based on a second-order intuitionistic linear type theory with recur-
sion. What makes it interesting from a programming language perspective is that, fol-
lowing ideas of Plotkin [10], the language is able to encode a remarkably rich range of
datatype constructs (eager products, lazy products, coproducts, polymorphism, abstract
types, recursive types, etc.). Furthermore, its linearity makes it potentially useful for
modelling single-threadedness and other state and resource-related concepts, cf. [7].

The main achievement of [3] was to establish direct operational techniques for rea-
soning about Lily up to contextual equivalence. Such techniques include useful ex-
tensionality properties, and a powerful framework for establishing program equalities
using an adaptation (based on [8]) of Reynolds’ relational parametricity (first introduced
in [11]). In order to get this machinery to work, the authors of [3] need to first establish
one key result about Lily, a result which pervades all further developments in their
paper. This result, the so-called Strictness Theorem, asserts the (surprising at first sight)
fact that call-by-name and call-by-value operational semantics for Lily both give rise
to the same notion of contextual equivalence.

The outline proof of the Strictness Theorem in [3] makes rather heavy use of the
well-stocked armoury of known operational techniques. In particular it uses Howe’s
method [4] to obtain a version of Mason and Talcott’s citu theorem [6]. The starting point
for the research in this paper was the realisation that basic techniques from rewriting

could be applied to obtain an alternative, self-contained and essentially simple proof of the
Strictness Theorem.

In Sec. 2, we review Lily and its operational semantics. Then, in Secs. 3 and 4, we
present our alternative proof of the Strictness Theorem. We translate Lily into a simple
untyped linear lambda-calculus containing: linear lambda abstractions, \( \lambda x.M \); non-
linear lambda abstractions, \( \lambda!x.M \), which require their arguments to be suspended as
“thunks”; and thunks themselves, \(!M\). We study beta-reduction in this untyped calculus,
making the restriction that, as thunks are considered suspended, reductions should not
take place within a thunk. This restricted relation, which we call surface reduction,
turns out to be extremely well behaved: as well as the expected confluence property,
it holds that every normalizing term is strongly normalizing. The Strictness Theorem
for Lily follows easily from this latter fact, using straightforward simulations under
surface reduction of call-by-name and call-by-value evaluation for Lily.

In Sec. 5, we (temporarily) turn attention away from Lily and take a deeper look
at our untyped linear lambda-calculus and its reduction properties. In order to obtain
a conversion relation between terms that is a congruence, it is necessary to consider
also reductions inside thunks. We call such reductions internal reductions, and we call
arbitrary reductions (either surface or internal) combined reductions. As well as the ex-
pected confluence properties (for both internal and combined reductions), we show that
internal reductions can always be postponed until after surface reductions. Further, we
show that if a term reduces (under combined reduction) to a surface normal form then
any sequence of combined reductions contains only finitely many surface reductions.

Next, in Sec. 6, we return to Lily and show that the results of Sec. 5 again have
applications to operational semantics. We use them to establish the equivalence of the
call-by-name operational semantics of Lily with an implementation-oriented call-by-
need semantics. Once again, the equivalence of these two semantics had previously
been established by the authors of [3], but with an intricate and lengthy proof (private
communication). Our proof turns out to be relatively straightforward.

Finally, in Sec. 7, we observe that our untyped linear lambda-calculus is exactly
the lambda-calculus counterpart of Abramsky’s linear combinatory algebras, presented
in [1]. This connection makes us believe that the linear lambda-calculus introduced in
this paper is rather natural. Accordingly, it is plausible that the properties of reduction
established in Secs. 3 and 5 may turn out to have other applications, perhaps again in
the area of operational semantics, but possibly more widely.

# 2 Lily and its Operational Semantics

In this section, we review the language Lily, a typed \( \lambda \)-calculus, based on second-
order intuitionistic linear type theory with recursion, introduced in [3].

The language of types for Lily contains just three type constructors: linear func-
tion space \( \sigma \rightarrow \tau \); linear “exponentials” \(,!\sigma\), used to type thunks; and universally quant-
tified types \( \forall \alpha.\sigma \), used for polymorphism. Types \( \sigma,\tau,\ldots \) are thus built up from type
variables \( \alpha,\beta,\ldots \), according to the grammar:

\[
\sigma ::= \alpha \mid \sigma \rightarrow \tau \mid !\tau \mid \forall \alpha.\tau
\]
Fig. 1. Typing rules for Lily.

As usual, $\alpha$ is bound in $\forall \alpha. \tau$. We write $\text{ftv}(\sigma)$ for the set of free type variables in $\sigma$ (and below apply the same notation to terms and contexts in the evident way). If $\text{ftv}(\sigma) = \emptyset$ then $\sigma$ is said to be closed.

Although simple, the above language of types is remarkably rich. For example, the other type constructors of intuitionistic linear logic can all be encoded: non-linear (intuitionistic) function space, $\sigma \rightarrow \tau$, using Girard’s $! \sigma \multimap \tau$; tensor, $\otimes$, product, $\&$ and sum, $\oplus$. One can also encode basic ground types (booleans, natural numbers, etc.), and existentially quantified types $\exists \alpha. \sigma$, and, due to the recursion operator in Lily, arbitrary recursive types. These encodings are due to Plotkin [10], see [3] for details.

The term language of Lily is the expected typed $\lambda$-calculus associated with the above types, together with a recursion operator.¹ Raw terms $s, t, \ldots$ are built from term variables $x, y, \ldots$ according to the grammar:

$$t ::= x \mid \lambda x : \sigma. t \mid s(t) \mid !t \mid \text{let } !x = s \text{ in } t \mid \Lambda \alpha. t \mid t(\sigma) \mid \text{rec } x : \sigma. t.$$

Here, $x$ is bound in $\lambda x : \sigma. t$, in $\text{let } !x = s \text{ in } t$ and in $\text{rec } x : \sigma. t$, and $\alpha$ is bound in $\Lambda \alpha. t$. We write $\text{fv}(t)$ for the set of free variables in a term $t$. We identify terms up to $\alpha$-equivalence.

The typing rules for Lily are based on Barber and Plotkin’s DILL [2]. We use $\Gamma, \Delta, \ldots$ to range over “contexts”, which are finite functions from term variables to types. We write $\Gamma \not\# \Delta$ to say that the domains of $\Gamma$ and $\Delta$ are disjoint. The typing rules manipulate sequents $\Gamma ; \Delta \vdash t : \sigma$ where $\Gamma \not\# \Delta$. Here, $\Gamma$ types the “intuitionistic” variables appearing in the term $t$, which have no restriction on how they occur, and $\Delta$ types the “linear” variables, each of which occurs exactly once in $t$, not within the

¹ We depart from [3] by building an explicit recursion operator into Lily, instead of incorporating recursion within thunks. This is an inessential difference.

² For simplicity, we place an inessential restriction in the typing rules ensuring that the term $\text{let } !x = s \text{ in } t$ is well typed only when $x$ does not occur free in $s$. 

s → s′
\[ s(t) \rightarrow s′(t) \]

\[ t \rightarrow t′ \]
\[ t(\sigma) \rightarrow t′(\sigma) \]
\[ (\Lambda x. t)(\sigma) \rightarrow t(\sigma/\alpha) \]

\[ s \rightarrow s′ \]
\[ \text{let } !x = s \text{ in } t \rightarrow \text{let } !x = s′ \text{ in } t \]

\[ t \rightarrow_{vl} t′ \]
\[ (\lambda x: \sigma. s)(t) \rightarrow_{vl} (\lambda x: \sigma. s)(t′) \]
\[ (\lambda x: \sigma. s)(v) \rightarrow_{vl} s[v/x] \]
\[ (\lambda x: \sigma. s)(t) \rightarrow_{nm} s[t/x] \]

Fig. 2. Call-by-value and Call-by-name Evaluation for Lily

scope of a ! or rec operator. The typing rules are presented in Fig. 1. In them, a comma always denotes a disjoint union of contexts and a dash denotes the empty context. We write \( t : \tau \) to mean that the sequent \( \vdash t : \tau \) is derivable, where \( \tau \) is a closed type (\( t \) is necessarily a closed term).

Following [3], we define two operational semantics for Lily, one using a call-by-value evaluation of function application, and one using call-by-name. In both cases, the operational semantics reduces terms to values \( v, \ldots \), which are terms of the form:

\[ v ::= \lambda x: \sigma. t \mid !t \mid \Lambda \alpha. t \]

In contrast to [3], we give the operational semantics in a small-step style. This facilitates our proofs, but only in an inessential way, the equivalence of big-step and small-step definitions being anyway easy to establish.

Figure 2 defines two small-step evaluation relations \( t \rightarrow_{vl} t′ \) and \( t \rightarrow_{nm} t′ \) between Lily terms. The call-by-value (or strict) relation \( t \rightarrow_{vl} t′ \) is inductively defined by the two specific \( \rightarrow_{vl} \) rules for application together with all rules written using the neutral \( \rightarrow \) notation. Similarly, the call-by-name (or non-strict) relation \( t \rightarrow_{nm} t′ \) is defined by the specific \( \rightarrow_{nm} \) rule for application together with the neutral rules. Note that both operational semantics are deterministic.

Our interest lies in the operational semantics of Lily programs, i.e. of closed terms of closed type. It is easily seen that if \( t : \sigma \) and \( t \rightarrow_{vl} t′ \) then \( t′ : \sigma \) (and similar if \( t \rightarrow_{nm} t′ \)). Also, by induction on the structure of \( t \), one sees that if \( t : \sigma \) then \( t \) does not reduce under \( \rightarrow_{vl} \) if and only if \( t \) is a value (and similar for \( \rightarrow_{nm} \)). We write \( t \downarrow_{vl} \) (resp. \( t \downarrow_{nm} \)) for the “termination” property: there exists a value \( v \) such that \( t \rightarrow^{*}_{vl} v \) (resp. \( t \rightarrow^{*}_{nm} v \)), where, as usual, \( R^+ \) (resp. \( R^\ast \)) denotes the reflexive-transitive (resp. transitive) closure of the relation \( R \).

The program below shows that sometimes call-by-name evaluation terminates when call-by-value does not (cf. [3, Example 2.2]).

\[ (\lambda f: \forall \alpha. \alpha \rightarrow \forall \alpha. \alpha. \lambda x: \forall \alpha. \alpha. f(x))(\text{rec } g: \forall \alpha. \alpha \rightarrow \forall \alpha. \alpha. g) \]  

This program has type \( \forall \alpha. \alpha \rightarrow \forall \alpha. \alpha \). An important insight of [3], is that the most useful notion of contextual equivalence for Lily is obtained by only observing termi-
nation for programs of exponential type !τ. The restriction to such observations corresponds to observing termination at ground types (such as booleans, naturals, etc.), it yields desirable extensionality properties for contextual equivalence, and it is crucial to the correctness of Plotkin’s [10] encodings of datatype constructions in Lily.

The key result of [3] that underpins its entire study of contextual equivalence for Lily is the “Strictness Theorem”.

Theorem 2.1 (Strictness Theorem [3]). If \( t : !\tau \) then \( t \downarrow_{\text{vl}} \) if and only if \( t \downarrow_{\text{nm}} \).

When termination observations are restricted to exponential types, it follows immediately from the theorem that both call-by-value and call-by-name operational semantics induce the same contextual equivalence.

We remark that the Strictness Theorem is stated in the most general form possible: the result holds for no types other than exponential types, as simple adaptations of (1) readily show. This suggests that any proof of Theorem 2.1 has to uncover some crucial property of exponential types. The machinery used in [3] to this end has already been mentioned in Section 1. In this paper, we shall instead prove Theorem 2.1 using surprisingly elementary techniques from rewriting, translating Lily into a very simple untyped linear \( \lambda \)-calculus in which (the appropriate notion of) \( \beta \)-reduction simulates both call-by-value and call-by-name operational semantics. This untyped linear \( \lambda \)-calculus includes explicit thunks, and it is the treatment of these thunks that will reflect the all-important behaviour of Lily at exponential type.

### 3 A Linear Lambda-calculus and Surface Reduction

In this section, we introducxe our untyped linear \( \lambda \)-calculus. Its main ingredients are: applications \( MN \); linear lambda abstractions, \( \lambda x.M \); non-linear lambda abstractions, \( \lambda !x.M \), which require their arguments to be suspended as thunks; and thunks themselves, !M. Formally, raw terms \( M, N, \ldots \) are built up from variables \( x, y, \ldots \) according to the grammar:

\[
M ::= x \mid MN \mid \lambda x.M \mid \lambda !x.M \mid !M.
\]

The variable \( x \) is bound in both \( \lambda x.M \) and \( \lambda !x.M \). We write \( \equiv \) for syntactic equality of terms modulo \( \alpha \)-equivalence.

We say that \( x \) is linear in \( M \) if \( x \) occurs free exactly once in \( M \) and, moreover, this free occurrence of \( x \) does not lie within the scope of a \( ! \) operator in \( M \). A term \( M \) is said to be linear if, in every subterm of \( M \) the form \( \lambda x.M' \), it holds that \( x \) is linear in \( M' \).

Henceforth, we consider linear terms only.

In Fig. 3, we define a version of \( \beta \)-reduction for our calculus. The important points are the two types of redex, and that no reduction occurs under the scope of a \( ! \) operator. The latter restriction reflects the idea that thunks are suspended computations. We call the reduction defined in Fig. 3 surface reduction. It is easily shown that when \( M \) is linear and \( M \rightarrow N \) then \( N \) is linear. From now on, all similar observations about linearity will be omitted. All operations we consider will respect the linearity of terms.

---

\(^3\) The theorem as stated here is easily shown to be equivalent to the original [3, Theorem 2.3].
A term is said to be in **surface normal form** if there is no surface reduction from the term. Trivially, any term $!M$ is in surface normal form. A **reduction sequence** from $M$ is a finite or infinite sequence $M \equiv M_0 \rightarrow M_1 \rightarrow M_2 \rightarrow \ldots$. A **completed reduction sequence** is a reduction sequence that is either infinite or is finite with the last term in the sequence in surface normal form.

The linearity restriction on terms combines with the disallowance of reduction within thunks to ensure that the basic well-behavedness properties of surface reduction are almost trivial to establish. The main, though very simple, results of this section are Corollaries 3.3 and 3.4 below. (Only the latter is used in the proof of Theorem 2.1.)

**Lemma 3.1.**

1. If $M \rightarrow M'$ then $M[N/x] \rightarrow M'[N/x]$.
2. If $N \rightarrow N'$ and $x$ is linear in $M$ then $M[N/x] \rightarrow M[N'/x]$.

**Proposition 3.2.** If $M \rightarrow L$ and $M \rightarrow L'$ then either $L \equiv L'$ or there exists $N$ such that $L \rightarrow N$ and $L' \rightarrow N$.

**Proof.** By induction on the structure of $M$, considering all possible cases for $M \rightarrow L$ and $M \rightarrow L'$. We consider only the two redex cases.

If $M \equiv (\lambda x. M_1)(M_2) \rightarrow M_1[M_2/x] \equiv L$ and $L \neq L'$ then either $L' \equiv (\lambda x. L'_1)(M_2)$ where $M_1 \rightarrow L'_1$ or $L' \equiv (\lambda x. M_1)(L'_2)$ where $M_2 \rightarrow L'_2$. In the first case, we have $L \rightarrow L'_1[M_2/x]$, by Lemma 3.1.1, and also $L' \rightarrow L'_1[L'_2/x]$. In the second, we have $L \rightarrow M_1[L'_2/x]$, by Lemma 3.1.2, and also $L' \rightarrow M_1[L'_2/x]$.

If $M \equiv (\lambda x. M_1)(!M_2) \rightarrow M_1[M_2/x] \equiv L$ and $L \neq L'$ then $L' \equiv (\lambda x. L'_1)(!M_2)$ where $M_1 \rightarrow L'_1$. Thus $L \rightarrow L'_1[M_2/x]$, by Lemma 3.1.1, and also $L' \rightarrow L'_1[M_2/x]$. \qed

**Corollary 3.3 (Confluence).** If $M \rightarrow^* M_1$ and $M \rightarrow^* M_2$ then there exists $N$ such that $M_1 \rightarrow^* N$ and $M_2 \rightarrow^* N$.

**Corollary 3.4 (Uniform normalization).** If $M \rightarrow^* V$ is a $k$-step reduction sequence, where $V$ is in surface normal form, then every reduction sequence from $M$ has at most $k$ steps, and every completed reduction sequence has exactly $k$ steps and terminates with $V$. In particular, if a term is normalizing under surface reduction then it is strongly normalizing.
4 Proof of the Strictness Theorem

The proof is based on a simple translation of Lily into the untyped linear \(\lambda\)-calculus of Sec. 3. The translation uses an untyped recursion construct, defined by:

\[
\mu x.M \overset{\text{def}}{=} \lambda x.\mu x.M[x(x)/x](\lambda x.\mu x.M[x(x)/x]).
\]

Observe that \(\mu x.M \rightarrow M[(\mu x.M)/x]\).

To every raw term \(t\) of Lily, we define a raw term \(t^*\), in the grammar from Sec. 3, by induction on the structure of \(t\). In the definition, we make use of a distinguished variable \(u\), used as a dummy translation for types.

\[
x^* = \overset{\text{def}}{=} x \quad (\text{let } !x = s \text{ in } t)^* = \overset{\text{def}}{=} (\lambda x.t^*)(s^*)
\]

\[
(\lambda x:\sigma.t)^* = \overset{\text{def}}{=} \lambda x.t^* \\
(A\alpha.t)^* = \overset{\text{def}}{=} \lambda !w.t^* \\
(s(t))^* = \overset{\text{def}}{=} s^*t^* \\
(!t)^* = \overset{\text{def}}{=} !t^* \\
(\text{rec } x:\sigma.t)^* = \overset{\text{def}}{=} \mu x.t^*.
\]

The four lemmas below are straightforward.

**Lemma 4.1.** \((s[t/x])^* \equiv s^*[t^*/x]\).

**Lemma 4.2.** If \(\Gamma; \Delta \vdash t : \sigma\) then the raw term \(t^*\) is linear.

**Lemma 4.3.** If \(t_1 \rightarrow_{vl} t_2\) then \(t_1^* \rightarrow t_2^*\).

**Lemma 4.4.** If \(t_1 \rightarrow_{nm} t_2\) then \(t_1^* \rightarrow t_2^*\).

**Corollary 4.5.** If \(t : !\tau\) then the following are equivalent:

1. \(t \downarrow_{vl}\),
2. \(t \downarrow_{nm}\),
3. \(t^*\) is surface normalizing.

**Proof.** To show that 1 implies 3, suppose that \(t \downarrow_{vl}\). Then there exists \(v\) with \(v : !\tau\) such that \(t \rightarrow_{vl} v\). As \(v : !\tau\), it holds that \(v \equiv !t'\). By Lemma 4.3, \(t^* \rightarrow (t')^* \equiv !(t'^*)\). But \(!t'^*)\) is in surface normal form, hence \(t^*\) is surface normalizing.

For the converse, suppose \(t \not\downarrow_{vl}\). Then there exists an infinite sequence of call-by-value evaluation steps \(t \equiv t_0 \rightarrow_{vl} t_1 \rightarrow_{vl} t_2 \rightarrow_{vl} \ldots\). Whence, by Lemma 4.3, \(t^*\) has an infinite surface reduction sequence. Thus, by Corollary 3.4, \(t^*\) is not normalizing under surface reduction.

The equivalence of 2 and 3 is shown in the same way, using Lemma 4.4. \(\square\)

Theorem 2.1 is immediate from the corollary. Note that the point that fails for Lily programs \(t : \sigma\) of arbitrary type is that it is not in general the case that \(t \downarrow_{vl}\) (or \(t \downarrow_{nm}\)) implies that \(t^*\) is surface normalizing, because, apart from at exponential type, Lily values do not necessarily translate to surface normal forms, indeed not even to surface normalizing terms (for example, \(\lambda x:\sigma. \text{rec } y:\tau. y\)).

It is worth remarking that the techniques of this section can similarly be used to show that variant operational semantics for Lily, in which evaluation takes place under \(\Lambda-\) and/or \(\lambda-\)abstractions, also give rise to the same contextual equivalence.
In this section, we undertake a deeper study of reduction in our untyped linear λ-calculus. While surface reduction is computationally motivated, the disallowance of reduction inside thunks means that the conversion relation induced by surface reduction is not a congruence. To obtain a conversion relation that is a congruence, it is necessary to consider reduction inside thunks.

We implement reduction inside thunks using *internal reduction*, defined in Figure 4. *Combined reduction* \( M \Rightarrow M' \) is defined by: \( M \Rightarrow M' \) if \( M \rightarrow M' \) or \( M \rightarrow M' \). Note that it is possible that both \( M \rightarrow M' \) and \( M \rightarrow M' \) (for example, \( \Omega(!\Omega) \rightarrow \Omega(!\Omega) \) and \( \Omega(!\Omega) \rightarrow \Omega(!\Omega) \), where \( \Omega \text{ def } \mu x. x \), using the notation of Section 4). Accordingly, when we consider mixed reduction sequences containing both surface and internal reductions, we shall assume that each step comes with a distinguished status (as surface or internal).

The main technical effort of this section will go into the proof of Propositions 5.1 and 5.2 below.

**Proposition 5.1 (Confluence).**
1. If \( M \rightarrow^* M_1 \) and \( M \rightarrow^* M_2 \) then there exists \( N \) such that \( M_1 \rightarrow^* N \) and \( M_2 \rightarrow^* N \).
2. If \( M \Rightarrow^* M_1 \) and \( M \Rightarrow^* M_2 \) then there exists \( N \) such that \( M_1 \Rightarrow^* N \) and \( M_2 \Rightarrow^* N \).

By the proposition, the conversion relation defined by \( M =_\beta M' \) if there exists \( N \) such that \( M \Rightarrow^* N \) and \( M' \Rightarrow^* N \) is an equivalence relation. It is, moreover, a congruence. Thus surface and internal reduction together provide an oriented decomposition of the natural \( \beta \)-conversion between terms of the untyped linear calculus. The next result exhibits natural structure within this decomposition.

**Proposition 5.2 (Internal Postponement).** If \( M \Rightarrow^* N \), by a reduction sequence containing \( k \) surface reductions, then there exists \( L \) such that \( M \rightarrow^* L \rightarrow^* N \), where the surface reduction sequence \( M \rightarrow^* L \) contains at least \( k \) reductions.

The proofs of the two propositions above make use of the (standard) technology of parallel reduction relations. Before giving these, we apply Proposition 5.2 to derive further properties of and interactions between surface, internal and combined reduction. The main result of the section is Theorem 5.5 below.
Lemma 5.3. If $M \to N$ and $M \to M'$ then there exists $N'$ such that $M' \to N'$.

*Proof.* By induction on the derivation of $M \to N$. We consider one case.

Suppose $M \equiv (\lambda!x.M_1)((!M_2) \to M_2[x/M]) \equiv N$. Either $M' \equiv (\lambda!x.M_1)((!M_2)$ where $M_1 \to M'_1$, or $M' \equiv (\lambda!x.M_1)((!M'_2)$ where $M_2 \to M'_2$. In the first case, $M' \to M'_1[M_2/x]$. In the second, $M' \to M'_1[M'_2/x]$. ∎

Corollary 5.4. If $V$ is in surface normal form then:

1. $V \to N$ implies $N$ is in surface normal form.
2. $M \to V$ implies $M$ is in surface normal form.

*Proof.* Statement 1 follows from Proposition 5.2, and statement 2 from Lemma 5.3. ∎

Theorem 5.5. If $M \Rightarrow^* V$, where $V$ is in surface normal form, then each infinite $\Rightarrow$ reduction sequence from $M$ contains only finitely many $\to$ reductions.

*Proof.* By Proposition 5.2, there exists $U$ such that $M \Rightarrow^* U \Rightarrow^* V$. By Corollary 5.4.2, $U$ is in surface normal form. Let $k$ be the number of reductions in the sequence $M \Rightarrow^* U$. We show that every $\Rightarrow$ reduction sequence from $M$ contains at most $k$ surface reductions. Consider any reduction sequence $M \Rightarrow^* N$ with $l$ surface reductions. By Proposition 5.2, there exists $L$ such that $M \Rightarrow^* L$ with at least $l$ reductions. But, by Corollary 3.4, any $\Rightarrow$ reduction sequence from $M$ has at most $k$ reductions. Thus indeed $l \leq k$. ∎

We now turn to the proofs of Propositions 5.1 and 5.2, which use the parallel versions of combined and internal reduction defined in Figs. 5 and 6 respectively.
Lemma 5.6.

1. \( M \not\equiv M \) and \( M \not\rightarrow M \).
2. If \( M \Rightarrow M' \) then \( M \not\equiv M' \). Conversely, if \( M \not\equiv M' \) then \( M \Rightarrow M' \).
3. If \( M \rightarrow M' \) then \( M \not\rightarrow M' \). Conversely, if \( M \not\rightarrow M' \) then \( M \rightarrow M' \).
4. If \( M \not\rightarrow M' \) then \( M \not\equiv M' \).

Lemma 5.7.

1. If \( M \not\equiv M' \) and \( N \not\equiv N' \) then \( M[N/x] \not\equiv M'[N'/x] \).
2. If \( M \not\rightarrow M' \) and \( N \not\rightarrow N' \) then \( M[N/x] \not\rightarrow M'[N'/x] \).

Lemma 5.8.

1. If \( M \not\equiv M_1 \) and \( M \not\equiv M_2 \) then there exists \( N \) such that \( M_1 \not\equiv N \) and \( M_2 \not\equiv N \).
2. If \( M \not\rightarrow M_1 \) and \( M \not\rightarrow M_2 \) then there exists \( N \) such that \( M_1 \not\rightarrow N \) and \( M_2 \not\rightarrow N \).

Proof. The proof, which is by induction on the structure of \( M \), is a routine analysis of all possible cases, cf. [9].

Proposition 5.1 is a straightforward consequence the last lemma.

Sub-lemma 5.9. If \( M \not\rightarrow M', N \not\equiv N' \) and \( N \Rightarrow N' \) then there exists \( L \) such that \( M[N/x] \rightarrow L \not\rightarrow M'[N'/x] \).

Proof. By a straightforward induction on the derivation of \( M \not\rightarrow M' \).

Lemma 5.10. If \( M \not\equiv M' \) then there exists \( L \) such that \( M \Rightarrow L \not\equiv M' \).

Proof. By induction on the derivation of \( M \not\equiv M' \). The most interesting case is when \( M \equiv (\lambda x.M_1)(!M_2) \not\equiv M'_1[M'_2/x] \equiv M' \), where \( M_1 \not\equiv M'_1 \) and \( M_2 \not\equiv M'_2 \). Then, by induction hypothesis, there exist \( L_1, L_2 \) such that \( M_1 \Rightarrow L_1 \not\rightarrow M'_1 \) and \( M_2 \Rightarrow L_2 \not\rightarrow M'_2 \). By Sub-lemma 5.9, there exists \( L \) such that \( L_1[M_2/x] \Rightarrow L \not\rightarrow M'_1[M'_2/x] \).

Thus \( M \equiv (\lambda x.M_1)(!M_2) \rightarrow M_1[M_2/x] \rightarrow L \not\rightarrow M'_1[M'_2/x] \equiv M' \), as required.

Lemma 5.11. If \( M \not\rightarrow L \rightarrow N \) then there exists \( L' \) such that \( M \rightarrow L' \not\equiv N \).

Proof. By induction on the derivation of \( L \rightarrow N \). We consider two cases.

If \( L \equiv (\lambda x.L_1)(!L_2) \rightarrow L_1[L_2/x] \equiv N \), then \( M \equiv (\lambda x.M_1)(!M_2) \) where \( M_1 \not\rightarrow L_1 \) and \( M_2 \not\equiv L_2 \). Thus \( M \rightarrow M_1[M_2/x] \) and, by Lemmas 5.6 and 5.7, we have that \( M_1[M_2/x] \not\equiv L_1[L_2/x] \equiv N \). Hence the result holds with \( L' \equiv L_1[M_2/x] \).

If \( L \equiv L_1L_2 \rightarrow N_1L_2 \equiv N \), where \( L_1 \rightarrow N_1 \), then \( M \equiv M_1M_2 \) where \( M_1 \not\rightarrow L_1 \) and \( M_2 \not\rightarrow L_2 \). By induction hypothesis, there exists \( L'_1 \) such that \( M_1 \rightarrow L'_1 \not\equiv N_1 \).

Thus \( M \rightarrow L'_1M_2 \not\equiv N_1L_2 \), hence the result holds with \( L' \equiv L'_1M_2 \).
Proof (of Proposition 5.2). We have a reduction sequence $M \Rightarrow^* N$, possibly consisting of both $\rightarrow$ and $\rightarrow^*$ rewrites. This can equally well be viewed as a sequence of $\rightarrow$ and $\rightarrow^*$ rewrites. We begin by associating a complexity measure to any such reduction sequence of $\rightarrow$ and $\rightarrow^*$ rewrites. To do this, first assign to to each $\rightarrow^*$ rewrite in the sequence the number of $\rightarrow$ rewrites that occur to the right of it. We thus obtain a sequence of numbers, one for each $\rightarrow^*$ rewrite, which we write in ascending order (equivalently, we write in sequence starting with the rightmost $\rightarrow^*$ rewrite and working leftwards). For example, the rewrite sequence

$$M \equiv M_0 \rightarrow^* M_1 \rightarrow^* M_2 \rightarrow M_3 \rightarrow^* M_4 \rightarrow^* M_5 \rightarrow M_6 \rightarrow^* M_7 \equiv N$$

gets assigned the sequence $0, 1, 3, 3$. This sequence is our complexity measure.

Now take the sequence of $\rightarrow$ and $\rightarrow^*$ rewrites reducing $M$ to $N$. If this sequence does not contain a subsequence $M_i \rightarrow^* M_{i+1} \rightarrow M_{i+2}$, then we have $M \Rightarrow^* M' \rightarrow^* N$, and hence $M \Rightarrow^* M' \Rightarrow^* N$ as required.

Otherwise, select a two-step subsequence $M_i \rightarrow^* M_{i+1} \rightarrow M_{i+2}$. Using Lemma 5.11 followed by 5.10, replace this with a sequence $M_i \rightarrow M' \rightarrow^* M'' \rightarrow^* M_{i+2}$. One thus obtains a new reduction sequence from $M$ to $N$ containing the same number of $\rightarrow^*$ rewrites and at least as many $\rightarrow$ rewrites (possibly more). However, because the identified $\rightarrow^*$ rewrite is shifted to the right, the complexity measure of the new sequence is below that of the original in the lexicographic ordering. Thus by repeatedly selecting two-step subsequences, we repeatedly reduce the complexity measure until we obtain a reduction sequence $M \Rightarrow^* M' \Rightarrow^* N$ containing at least as many surface rewrites as the original sequence. Therefore $M \Rightarrow^* M' \Rightarrow^* N$, as required. \square

6 Call-by-need Operational Semantics for Lily

In the Lily expressions let !x = s in t and rec x : σ. t, the variable x may occur zero, one or several times in t. Because of this, the natural implementation mechanism is call-by-need, whereby the evaluation of the terms substituted for such variables is shared. (In contrast, in an application $(\lambda x : \sigma. t)(s)$, the variable x occurs exactly once in t, and there is no call for sharing.) An operational semantics implementing such a call-by-need evaluation strategy is presented in [3], and the authors have proved that the call-by-need semantics does not affect the notion of contextual equivalence (private communication). In this section, we outline a straightforward proof of this result.

Again, rather than using the big-step operational semantics of [3], which is based on [5, 12], it is convenient for our purposes to use a small-step version, following [13]. We use $S, \ldots$ to range over variable/frame stacks, which are sequences of items of two forms: (i) $\langle F \rangle$, where $F$ is an “evaluation frame”,

$$F ::= (-)(t) \mid \text{let } !x = (-) \text{ in } t \mid (-)(\sigma) ;$$

(ii) or $\langle x \rangle$, for a variable x. We use $H$ to range over heaps, which are finite sequences of assignments of the form $[x \mapsto t]$, with all variables x distinct.

The call-by-need evaluation relation is defined in Fig. 7. It implements a single-step relation of the form $(S, t, H) \rightarrow_{\text{mut}} (S', t', H')$. Roughly, this is interpreted as saying
that the Lily term built up from \( t \) using the nested evaluation frames in \( S \) evaluates in a single step to the term built from \( t' \) using the frames in \( S' \). In Fig. 7, when we write \([x \mapsto t]H\), we assume that \( x \) is not in the domain of \( H \). We treat heaps \( H \) as functions, writing \( H(x) \) for the value assigned to \( x \), and writing \( H[v/x] \) for the heap obtained from \( H \) by replacing the existing term assigned to \( x \) (which is assumed to be in the domain of \( H \)) with \( v \).

The call-by-need evaluation of a Lily program \( t : \sigma \) starts off with the configuration \((\varepsilon, t, \varepsilon)\) (where \( \varepsilon \) is the empty sequence) and then proceeds deterministically according to the rules in Fig. 7. Either an infinite sequence of \( \rightarrow_{nd} \) reductions results, or the evaluation terminates in a configuration of the form \((\varepsilon, v, H)\) for some (possibly open) value \( v \). If the latter case holds, we write \( t \downarrow_{nd} \). The main result of this section states that, for programs of arbitrary type, the call-by-need semantics terminates if and only if the call-by-name semantics does.

**Theorem 6.1.** If \( t : \sigma \) then \( t \downarrow_{nd} \) if and only if \( t \downarrow_{nm} \).

The sharing of recursion implemented in Fig. 7, introduces cycles into the heap, and this makes it hard to give a direct operational proof of the equivalence of call-by-name and call-by-need, see [12] for discussion. This difficulty has, in fact, been overcome by the authors of [3], but their proof is highly involved (private communication).

We give a significantly simpler proof that call-by-name and call-by-need coincide. First, we define an almost trivial translation of Lily into itself, which serves the purpose of “padding out” the call-by-name semantics (sic) for the purpose of facilitating its comparison with the call-by-need semantics. The remaining step is to prove that the “almost trivial” translation really is trivial. For this last step, we again translate into the untyped linear \( \lambda \)-calculus of Sec. 3, this time applying Theorem 5.5.

The almost trivial translation from Lily to itself, is the identity everywhere, except for the translation of thunks, which are padded with a dummy recursion, acting as delay.

\[
(ls) \overset{!}{=} def !((rec z : \tau. s)) \quad z \notin \text{fv}(s).
\]
Here, we are translating well-typed terms \( \Gamma; \Delta \vdash t : \sigma \), to well-typed terms \( \Gamma; \Delta \vdash t^\dagger : \sigma \), and the type \( \tau \) introduced above is determined by this requirement.

**Lemma 6.2.** If \( t : \sigma \) then \( t \downarrow \text{nd} \) if and only if \( t^\dagger \downarrow \text{nm} \).

To prove Lemma 6.2, one shows that the call-by-name evaluation of \( t^\dagger \) simulates the call-by-need evaluation of \( t \). Crucially, the padding of thunks ensures that rule 8 of Fig. 7 always corresponds to a \( \rightarrow \text{nm} \) reduction for the term generated from \( t^\dagger \) by inserting it in the context determined by \( F \) and substituting, for each variable \( x \) with associated heap assignment \([x \mapsto s]\), a term \( \text{rec } x : \sigma. s_0 \), where \( s_0 \) is the term originally assigned to \( x \) when it was first added to the heap. More details are given in Appendix A.

Theorem 6.1 now follows from the lemma below, which is an easy application of Theorem 5.5.

**Lemma 6.3.** If \( t : \sigma \) then \( t \downarrow \text{nm} \) if and only if \( t^\dagger \downarrow \text{nm} \).

**Proof.** We give another translation from \texttt{Lily} into our untyped linear \( \lambda \)-calculus.

\[
\begin{align*}
    x^\dagger &= _{\text{def}} x \\
    (\lambda x : \sigma. t)^\dagger &= _{\text{def}} !(\lambda x.t)^\dagger(s^\dagger) \\
    (s(t))^\dagger &= _{\text{def}} (\lambda w.w(t^\dagger))(s^\dagger) \\
    (t)^\dagger &= _{\text{def}} !(t^\dagger) \\
    (\text{rec } x : \sigma. t)^\dagger &= _{\text{def}} \mu x. t^\dagger
\end{align*}
\]

It is easily established that, for \( t : \sigma \) we have that \( t \downarrow \text{nm} \) if and only if \( t^\dagger \) is surface normalizing. However, we have \( (t^\dagger)^\dagger \rightarrow^{*} t^\dagger \). Therefore, by Theorem 5.5, \( t^\dagger \) is surface normalizing if and only if \( (t^\dagger)^\dagger \) is. Thus indeed \( t \downarrow \text{nm} \) if and only if \( t^\dagger \downarrow \text{nm} \). \( \square \)

More generally, a similar application of Theorem 5.5 shows that call-by-name termination is preserved by the congruence relation on \texttt{Lily} terms generated by the call-by-name reductions. In other words, the natural “conversion relation” on \texttt{Lily} terms is correct with respect to contextual equivalence. Of course, the use of rewriting methods for establishing such simple results goes back to [9].

### 7 Linear Combinatory Algebras

The aim of this short final section is to demonstrate that our untyped linear \( \lambda \)-calculus is the \( \lambda \)-calculus counterpart of Abramsky’s linear combinatory algebras, see [1]. This gives some evidence that our calculus arises reasonably naturally, independently of its applications to operational semantics.

**Definition 7.1.** A \(!\)-applicative structure is an algebra \((A, \cdot, !)\) where \( \cdot \) is a binary operation on the set \( A \) and \( ! \) is a unary operation.

As is standard, we usually omit the “application” operation ‘\( \cdot \)’, using a simple juxtaposition \( xy \) for \( x \cdot y \). Application associates to the left (i.e. \( xyz = (xy)z \)).
Definition 7.2 ([1]). A \textit{linear combinatory algebra} is a $!$-applicative structure $(A, \cdot, !)$ in which there exist elements $I, B, C, K, W, D, \delta, F \in A$ satisfying:

\begin{align*}
I x &= x \\
W x(!y) &= x(!y)(!y) \\
B x y z &= x(yz) \\
D(!x) &= x \\
C x y z &= xzy \\
\delta(!x) &= !!x \\
K x(!y) &= x \\
F(!x)(!y) &= !(xy)
\end{align*}

The main result of this section asserts that linear combinatory algebras are characterized by a form of combinatory completeness in which the forms of implicit $\lambda$-abstraction available correspond to the two forms $\lambda x. M$ and $\lambda !x. M$ of our untyped linear $\lambda$-calculus. Moreover, the equalities associated with the implicit abstractions agree with the two redex forms in Fig. 3.

A $!$-applicative polynomial over a set $A$ is a syntactic expression built up using elements of $A$ as constants, variables $x, y, \ldots$, and operator symbols $\cdot$ and $!$. Any $!$-applicative structure $(A, \cdot, !)$ induces an evident equality relation between polynomials.

We say that a variable $x$ is \textit{linear} in a $!$-applicative polynomial $e$, if it occurs exactly once, and not within the scope of a $!$-operator symbol. We write $\text{vars}(e)$ for the set of variables occurring in $e$, and $\text{linvars}(e)$ for the set of variables that are linear in $e$.

Theorem 7.3 (Linear combinatory completeness). For any $!$-applicative structure $(A, \cdot, !)$, the following are equivalent.

1. $(A, \cdot, !)$ is a linear combinatory algebra.
2. For any $!$-applicative polynomial $e$ over $A$,
   \begin{enumerate}
   \item if $x \in \text{linvars}(e)$ then there exists a polynomial $\lambda^* x. e$ with $\text{vars}(\lambda^* x. e) = \text{vars}(e) - \{x\}$ and $\text{linvars}(\lambda^* x. e) = \text{linvars}(e) - \{x\}$ such that the equality $(\lambda^* x. e)(x) = e$ holds;
   \item there exists a polynomial $\lambda^! x. e$ with $\text{vars}(\lambda^! x. e) = \text{vars}(e) - \{x\}$ and $\text{linvars}(\lambda^! x. e) = \text{linvars}(e) - \{x\}$ such that $(\lambda^! x. e)(!x) = e$.
   \end{enumerate}

It follows easily from the theorem that the closed linear terms of our untyped $\lambda$-calculus, considered modulo $=_{\beta}$ (see Sec. 5), themselves form a linear combinatory algebra.

References

A Outline proof of Lemma 6.2

The main technical lemma we need is Lemma A.1 below. This concerns configurations 
(S, t, H) arrived at by a sequence ((e, s, e) →* nd (S, t, H) for some program s : σ. Given 
such a sequence, and any term t′ with fv(t′) contained in the domain of H, (the term t 
is one such), we define t′[H] as follows. If H = e then t′[H] = def t. If H = [x \mapsto u]H′ 
then t′[H] = def t′([(\text{rec } x : \sigma^†, u_0)/x])[H′], where u_0 is the first value assigned to 
x in a heap occurring along the sequence (e, s, e) →* nd (S, t, H), and σ^† is the appropriate 
type. Here t′[H] is an abuse of notation since the value does not solely depend on H. In 
fact, for any two heaps H_1, H_2 occurring in the sequence (e, s, e) →* nd (S, t, H) and 
containing fv(t′), it holds that t′[H_1] ≡ t′[H_2]. Also, for any term t′ we define [S]t′ as 
follows. If S = e then [S]t′ = def t′. If S = S′ ((\neg) (s′)) then [S]t′ = def [S′](t′(s)). If 
S = S′ (let !x = – in s′) then [S]t′ = def [S′](\text{let !x = t′ in s′}). If S = S′ ((\neg) (s′)) 
then [S]t′ = def [S′](t′(σ′)). If S = S′ (x) then [S]t′ = def [S′](x). Finally, we call 
reductions number 4–8, in Fig. 7, active, and the others passive.

Lemma A.1. Suppose s : σ and ((e, s, e) →* nd (S, t, H).

1. If x is declared in H then \langle x[H] \rangle \uparrow →_\text{nm} ((H(x))[H]) \uparrow.
2. If S = S_0 (x) S_1 then \langle x[H] \rangle \uparrow →_\text{nm} (((S_1)[t])[H]) \uparrow.
3. If (S, t, H) →_\text{nd} (S′, t′, H′), where S = S_0 S_1 and S′ = S_0 S_1, then it holds that 
   (((S_1)[t])[H]) \uparrow →_\text{nm} (((S_1)[t′])[H′]) \uparrow. Moreover, if the call-by-need reduction step is 
   active then the call-by-name sequence contains at least one reduction.

All three statements are proved simultaneously, by induction on the length of the reduction 
sequence ((e, s, e) →* nd (S, t, H). For space reasons, we omit the details.

Proof (of Lemma 6.2). If t \downarrow_\text{nd} then it follows easily from Lemma A.1.3 that t\uparrow \downarrow_\text{nm}.

If t \uparrow_\text{nd} then there exists an infinite →_\text{nd} reduction sequence from (e, t, e). Because 
the four passive reductions either strictly reduce the size of the term component in 
a configuration, or retain the same term and reduce the size of the stack, the infinite 
sequence cannot contain infinitely many consecutive passive reductions. Therefore, 
it must contain infinitely many active reductions. Thus, again by Lemma A.1.3, t\uparrow has an 
infinite →_\text{nm} reduction sequence. So indeed t\uparrow \downarrow_\text{nm}. □