Linear Explicit Substitutions

NEIL GHANI, *School of Computer Science, University of Birmingham, e-mail:* `N.Ghani@cs.bham.ac.uk`

VALERIA DE PAIVA, *School of Computer Science, University of Birmingham, e-mail:* `V.dePaiva@cs.bham.ac.uk`

EIKE RITTER, *School of Computer Science, University of Birmingham, e-mail:* `E.Ritter@cs.bham.ac.uk`

Abstract

The $\lambda\sigma$-calculus adds explicit substitutions to the λ-calculus so as to provide a theoretical framework within which the implementation of functional programming languages can be studied. This paper generalises the $\lambda\sigma$-calculus to provide a *linear calculus of explicit substitutions*, called xDILL, which analogously describes the implementation of linear functional programming languages.

Our main observation is that there are non-trivial interactions between linearity and explicit substitutions and that xDILL is therefore best understood as a synthesis of its underlying logical structure and the technology of explicit substitutions. This is in contrast to the $\lambda\sigma$-calculus where the explicit substitutions are independent of the underlying logical structure.

Keywords: λ-calculus, explicit substitutions, linear logic

1 Introduction

This paper combines the technologies of explicit substitutions and linearity in a mathematically consistent way. We start by describing these technologies and then explain the problems which arise when we try to combine them and our proposed solutions.

Why Explicit Substitutions? Traditional functional programming languages are based on the λ-calculus and model computation by β-reduction. In its usual form $(\lambda x.t)u \Rightarrow t[u/x]$ the argument u is substituted into the body of the λ-abstraction in one step. This is highly inefficient from the perspective of implementations as each redex in u may be copied arbitrarily often. Abstract machines avoid this problem by reducing terms in an environment — the contraction of a β-redex creates a new substitution which, instead of being substituted into the body of the λ-abstraction, is added to the existing environment and only evaluated when actually needed.

In order to study such machines, substitution must be made part of the λ-calculus instead of the usual approach where substitutions are meta-theoretic operations. This has led to the development of a variety of *calculi of explicit substitutions*. The canonical calculus of explicit substitutions is the $\lambda\sigma$-calculus of Abadi et al. [1] whose essence may be summarised as follows:

- **Expressions:** A new syntactic category of *explicit substitutions* is added to the λ-calculus to represent the environments in which terms are evaluated. There are new typing rules for constructing explicit substitutions and for applying such substitutions to other terms.
- **Reduction:** Secondly, there is a reduction calculus which contains a modified form of β-reduction which creates an explicit substitution and a set of rewrite rules which describe how such explicit substitutions may be performed. The evaluation of terms in an abstract machine then corresponds to a particular reduction strategy for this reduction calculus.
- **Correctness:** Finally, there is a correctness proof stating that the more refined notion of β- reduction in the $\lambda\sigma$-calculus agrees with that in the λ-calculus. That is, reduction in the $\lambda\sigma$-calculus is confluent and every $\lambda\sigma$-term rewrites to a unique normal form which is a λ-term.

This paper develops a linear calculus of explicit substitutions which conforms to this general paradigm. But first we discuss why we should want such a calculus.

Why Linear Explicit Substitutions? Linear logic [15] has attracted the interest of functional programmers since its inception. Its representation within the syntax of weakening and contraction was seen as permitting a compile-time analysis of *sharing* and *garbage collection*. More concretely, some of the overhead involved in the environment based approach could be avoided by detecting linear usage of variables and directly performing substitutions for such variables. These ideas have led to several abstract machines [16, 17, 18], but all run into difficulties with correctly exploiting linearity information. In particular, Wadler [22] and Chirimar et al. [7] tried to give an operational semantics where terms of linear type have exactly one pointer to them. However, their attempts failed mainly because the linearity inherent in their type system did not actually correspond to linear usage. This difficulty has its roots in the semantics of linear logic: not all terms of type $!A$ may be duplicated but only those whose free variables are also of type $!B$ for some B.

A syntactic way of capturing this difference has been proposed by Barber in his calculus *Dual Intuitionistic Linear Logic* (DILL) [3]. The key feature of DILL is the presence of both *intuitionistic* and *linear* variables. Contraction and weakening apply only to intuitionistic variables, and even linear variables of type $!A$ cannot be thrown away or copied, only passed around. This distinction is a key feature in the design of our linear abstract machine [2].

Just as traditional abstract machines are based on calculi of explicit substitutions, we want to base our abstract linear machines on a linear calculus of explicit substitutions which conforms to the principles outlined above and elsewhere [20]. Our proposed calculus is based upon the explicit substitutions of Abadi et al. [1] and the linear λ-calculus DILL [3] — hence we name our calculus xDILL. The distinction in DILL between intuitionistic and linear variables is reflected in xDILL by the existence of two kinds of explicit substitutions, one for linear variables and one for intuitionistic variables. This distinction is crucial for the abstract machine: in many cases, the linear substitution can be executed directly, whereas the intuitionistic substitutions must be stored in environments as in Krivine's machine [21].

Interactions of Linearity and Explicit Substitutions In the $\lambda\sigma$-calculus the logical structure of the λ-calculus is orthogonal to the explicit substitutions. That is, the inference rules for the explicit substitutions are independent of those for the connectives of the underlying λ-calculus. However, in xDILL there are non-trivial interactions between linearity and explicit substitutions:

- **Let-Substitutions:** The fundamental feature of linear logic is the isomorphism between a linear variable of type $!A$ and an intuitionistic variable of type A. In DILL, this is ensured by the `let`-expression arising from the !-elimination rule. Since xDILL contains terms and substitutions, enforcing this isomorphism requires terms `let` t `be` $!x$ `in` u (where u is a term) *and* explicit substitutions `let` t `be` $!x$ `in` f (where f is an explicit substitution). Similarly, the `let`-expressions associated to the tensor and the unit must be generalised to substitutions. We call such substitutions *let-substitutions*.
  Since `let`-expressions arise from the elimination rule of a logical connective, the structure of explicit substitutions in xDILL depends on the logical structure of the underlying λ-calculus. This differs from the $\lambda\sigma$-calculus where the inference rules for the explicit substitutions are independent of those for the connectives of the underlying λ-calculus.
- **Let-Substitutions and Reduction:** Let-substitutions complicate reduction in xDILL. In particular new commuting conversions are required and normal forms of substitutions are no longer simply environments, *i.e.*, lists of term/variable pairs, but are in general `let`-expressions. This is an instance of a general phenomenon which arises in reduction in calculi containing `let`-expressions [9] and hence is to be expected.
- **Linearity and Reduction:** Finally, the linearity constraints of xDILL mean that some of the rewrite rules of the $\lambda\sigma$-calculus require reformulation. For instance, if $f * t$ represents the application of a substitution f to a term t, then the $\lambda\sigma$-reduction rule
  $$f * (uv) \Rightarrow (f * u)(f * v)$$
  duplicates the free variables in f and hence does not satisfy the crucial property of subject reduction in a linear setting. Our solution is to partition f into that part which applies to u, written f_u, and that part which applies to v, written f_v. We can then use the alternative reduction rule
  $$f * (uv) \Rightarrow (f_u * u)(f_v * v)$$

Remarks

1. It is possible to derive a simpler linear λ-calculus with explicit substitutions by considering a subset of DILL with two function spaces, one which accepts arguments which can be thrown away or duplicated, and one which accepts only

arguments which are used exactly once. This calculus has no let-substitutions, and hence is less complicated. It can be used to derive a linear version of Krivine's machine but for applications like optimal reduction or update-in-place it is necessary to consider terms of type $!A$, which are not part of this subset. Hence we consider in this paper a calculus based on DILL rather than on the subset.

2. It is also possible to construct linear λ-calculi with explicit substitution based on other linear λ-calculi like the one by Bierman et al. [5] and other λ-calculi with explicit substitutions like λx [6] or $\lambda\upsilon$ [4]. This construction follows a general schema: We take a linear λ-calculus and extend it firstly by the term substitutions of the calculus with explicit substitutions (*e.g.*, λx or $\lambda\upsilon$), and secondly by the let-substitutions discussed above. The correctness theorem, which states that the reductions of the linear λ-calculus with explicit substitutions should be refinements of the reductions in the calculus without explicit substitutions, guides the definition of the reduction rules for let- and term substitutions. In this way it is possible to construct a linear λx- or a linear λe_s-calculus.

This paper is structured as follows. We start by recalling the calculi whose synthesis leads to xDILL, namely the traditional calculus of explicit substitutions $\lambda\sigma$ and Barber's term calculus for intuitionistic linear logic DILL. In section 3 we present xDILL, its reduction calculus and its correctness proof and also consider the effect of adding computational commuting conversions to xDILL. In section 4 we discuss alternative ways of combining linearity with explicit substitutions and argue why our calculus is better. Finally, in section 5 we present our conclusions and future directions of research.

2 Towards a linear λ-calculus with Explicit Substitutions

In order to motivate our linear calculus of explicit substitutions, we briefly recall our choices as far as the traditional (cartesian) calculus of explicit substitutions and underlying linear λ-calculus are concerned.

2.1 A Cartesian Calculus of Explicit Substitutions

We briefly recall the variant of the $\lambda\sigma$-calculus with names which we later generalise to the linear setting — see [20] for a full presentation of the calculus and its metatheory. We use the name $\lambda\sigma$ to describe this calculus as it captures the essence of the calculus of Abadi et al [1]. The types of the $\lambda\sigma$-calculus are ground types and function types $G \mid A{\to}B$. The raw expressions of $\lambda\sigma$ are:

$$\begin{array}{rcl} t & ::= & x \mid \lambda x{:}\, A.t \mid tt \mid f * t \\ f & ::= & \langle\rangle_X \mid \langle f, t/x\rangle \mid f; f \end{array}$$

where x is a variable and X is a set of variables. The term $f * t$ represents the application of the explicit substitution f to the term t, while $\langle\rangle_X$ should be thought of as a substitution of variables for themselves — the suffix X encodes typing information into the raw expressions and is discussed in section 2.1.2. The substitution $\langle f, t/x\rangle$ represents the parallel composition of the substitution f with the substitution of the term t for the variable x. Finally, $f; g$ represents the sequential composition of the substitutions f and g which arises naturally in several situations: (i) the composition of substitutions is used during reduction to normal form rather than weak head-normal form; (ii) when modelling instantiation in theorem provers; (iii) as the counterpart of composition in the categorical semantics of λ-calculi; and (iv) as a mechanism for modelling sharing in environment machines via substitutions of the form $\langle u/z\rangle; \langle z/x, z/y\rangle$. [1]

2.1.1 Notational Conventions

Throughout this paper we assume Barendregt's variable convention that bound variables and free variables are distinct and identify α-equivalent terms. This is not an entirely trivial matter in calculi of explicit substitution as the operators $*$ and ; bind variables, *e.g.*, if y is a fresh variable then the expressions $\langle t/x\rangle * u$ and $\langle t/y\rangle * u[y/x]$ are α-equivalent. For details of α-equivalence see subsection 3.3, where we consider the problem for xDILL, which is an extension of the $\lambda\sigma$-calculus.

We write $\langle\langle\rangle_X, t_1/x_1, \ldots, t_n/x_n\rangle$ for $\langle..\langle\langle\rangle_X, t_1/x_1\rangle.., t_n/x_n\rangle$ and, if X is empty, we simply write $\langle t_1/x_1, \ldots, t_n/x_n\rangle$. We use the usual notation of $t[u_1/x_1, \ldots, u_n/x_n]$ to denote the result of actually performing the meta-level substitutions of u_i for x_i in parallel on the term t. A context is a list of the form $x_1 : A_1, \ldots, x_n : A_n$ where the x's are distinct variables and the A's are types — the domain of the context is the set of variables $\{x_1, \ldots, x_n\}$. Given a context Γ and a set of variables X, the context $\Gamma \setminus X$ is obtained by removing X from the domain of Γ.

The $\lambda\sigma$-calculus has term judgements of the form $\Gamma \vdash t : A$ and substitution judgements of the form $\Gamma \vdash f : \Delta$ — these judgements are generated by the inference rules of Table 1. The use of a context on the right-hand side of a substitution judgement can be seen as an implicit conjunction, *i.e.*, f is offering to substitute terms for *all* of the variables mentioned in Δ. The inference rules for declaring variables and the $\rightarrow$-introduction and elimination rules are standard. The inference rules for the substitution judgements of the $\lambda\sigma$-calculus are independent of the connectives of the logic and hence we may say that the explicit substitutions and the logical structure of the $\lambda\sigma$-calculus are orthogonal. In xDILL there is a more synthetic relationship arising from the integration of linearity constraints into the explicit substitutions.

[1] The expression $\langle z/x, z/y\rangle$ is not a $\lambda\sigma$-pre-term as defined above but uses a notational convenience which we explain below.

TABLE 1. Typing Judgements for the $\lambda\sigma$-calculus

- Term Judgements

$$\frac{x\colon A \text{ declared in } \Gamma}{\Gamma \vdash x\colon A} \qquad \frac{\Gamma \vdash f : \Delta \qquad \Delta \vdash t : A}{\Gamma \vdash f * t : A}$$

$$\frac{\Gamma, x\colon A \vdash t\colon B}{\Gamma \vdash \lambda x\colon A.t\colon A \to B} \qquad \frac{\Gamma \vdash t\colon A \to B \qquad \Gamma \vdash u\colon A}{\Gamma \vdash tu\colon B}$$

- Substitution Judgements

$$\frac{}{\Gamma \vdash \langle\rangle_X : \Gamma \setminus X} \qquad \frac{\Gamma \vdash f\colon\Delta \qquad \Gamma \vdash t\colon A}{\Gamma \vdash \langle f, t/x\rangle : \Delta, x : A} \qquad \frac{\Gamma \vdash f\colon\Delta \qquad \Delta \vdash g\colon\Psi}{\Gamma \vdash f; g : \Psi}$$

where in the second rule $x \notin \mathtt{dom}\Delta$.

2.1.2 Typing the substitution $\langle\rangle$

There are several possibilities for typing the empty substitution $\langle\rangle$. For example, any of the following may be considered:

$$\frac{}{\Gamma \vdash \langle\rangle : \Gamma} \qquad \frac{}{\Gamma \vdash \langle\rangle : \_} \qquad \frac{\Gamma' \subseteq \Gamma}{\Gamma \vdash \langle\rangle : \Gamma'}$$

The first rule cannot be used as we require weakening to be admissible. Since β-reduction is to be formulated as $(\lambda x.t)u \Rightarrow \langle u/x\rangle * t$, then for subject reduction to hold, and if $\Gamma \vdash (\lambda x.t)u$, one must have $\Gamma \vdash \langle u/x\rangle : \Gamma, x : A$. As weakening is admissible and reduction is to be defined on raw expressions [2], the following inference rule must be admissible

$$\frac{\Gamma \vdash f : \Gamma'}{\Delta, \Gamma \vdash f : \Delta, \Gamma'}$$

which rules out the second possibility. The last possibility is also problematic as the associated theory of normal forms is unsatisfactory, *e.g.*, the substitution $\langle u/x\rangle; \langle\rangle$ is a normal form since we do not know if $\langle\rangle$ discards or retains the variable x. Our solution uses the annotation of a set of variables X to record those variables which are discarded, and hence we obtain both subject reduction and a decent theory of normal forms. As a result, $\langle\rangle$ is best regarded as a projection. This explicit representation of weakening within calculi of explicit substitution is common and is, for example, present in Abadi et al's $\lambda\sigma$-calculus [1].

[2]For implementational reasons we want to avoid the reductions-in-context approach which is notationally cumbersome and hence inefficient

2.2 Reduction in the $\lambda\sigma$-calculus

In the λ-calculus, β-reduction is an atomic process, *i.e.*, it is performed in one-step. Reduction in the $\lambda\sigma$-calculus refines this process into two distinct stages. Firstly there is the alternative form of β-reduction which creates an explicit substitution

$$(\lambda x : A.t)(u) \Rightarrow \langle u/x \rangle * t$$

and, secondly, a new rewrite relation $\Rightarrow_\sigma$ which controls how such explicit substitutions are performed, *e.g.*, how the term $\langle u/x \rangle * t$ can be evaluated to $t[u/x]$. These two facets of reduction in the $\lambda\sigma$-calculus are formalised by relations $\Rightarrow_\beta$ and $\Rightarrow_\sigma$ as defined in Table 2. We also define $\Rightarrow_{\beta\sigma}$ to be the union of $\Rightarrow_\beta$ and $\Rightarrow_\sigma$. Since the relation $\Rightarrow_\sigma$ is less well known, we describe it in more detail:

- **Reducing Terms:** Rewrite rules (1) and (2) push substitutions through term constructors until the substitution is applied to a variable — (1) performs an α-conversion to avoid variable capture; then rules (3) and (4) actually replace a substitution applied to a variable with the correct term. Given that the rewrite rules (6-10) reduce all substitutions to substitutions of the form $\langle\langle\rangle_X, t_1/x_1, \ldots, t_n/x_n\rangle$, the rewrite rules (1-5) ensure that normal forms for terms contain no substitutions.
- **Reducing Substitutions:** The rewrite rules (6-10) exploit the possibilities for sharing presented by the presence of sequential composition of substitution. For instance, the term $\langle t/x \rangle * \langle\langle\rangle_x, u/y\rangle * v$ can be reduced to its normal form in two different ways. The first approach uses the rules (1-5)

  $$\begin{aligned} \langle t/x \rangle * \langle\langle\rangle_x, u/y\rangle * v &\Rightarrow^*_\sigma \langle t/x \rangle * v[u/y] \\ &\Rightarrow^*_\sigma v[u[t/x]/y] \end{aligned}$$

  This reduction sequence first substitutes u in v and then, for each resulting occurrence of u in v, performs a substitution of t in u. The following reduction sequence is potentially shorter as t is substituted into u exactly once and then resulting term is substituted into v.

  $$\begin{aligned} \langle t/x \rangle * \langle\langle\rangle_x, u/y\rangle * v &\Rightarrow^*_\sigma \langle u[t/x]/y \rangle * v \\ &\Rightarrow^*_\sigma v[u[t/x]/y] \end{aligned}$$

  In fact, these rules ensure that normal forms for substitutions are of the form $\langle\langle\rangle_X, t_1/x_1, \ldots, t_n/x_n\rangle$.

LEMMA 2.1 (Subject Reduction)
Given a typing judgement $\Gamma \vdash t : A$ and a rewrite $t \Rightarrow_{\beta\sigma} t'$, then there is a typing judgement $\Gamma \vdash t' : A$. Similarly, if there is a typing judgement $\Gamma \vdash f : \Delta$ and a rewrite $f \Rightarrow_{\beta\sigma} f'$, then there is a typing judgement $\Gamma \vdash f' : \Delta$.

PROOF. Induction over the structure of expressions. ∎

TABLE 2. Reductions for $\lambda\sigma$

- A modified form of β-reduction

$$(\lambda x\colon A.t)u \quad \Rightarrow_\beta \quad \langle u/x\rangle * t$$

- Application of Substitutions: Let @ be either ; or $*$

$$\begin{array}{llll}
(1) & f * \lambda x.t & \Rightarrow_\sigma & \lambda y.\langle f, y/x\rangle * t \quad y \text{ fresh} \\
(2) & f * (tu) & \Rightarrow_\sigma & (f * t)(f * u) \\
(3) & \langle f, t/x\rangle * x & \Rightarrow_\sigma & t \\
(4) & \langle f, t/x\rangle * y & \Rightarrow_\sigma & f * y \quad (x \neq y) \\
(5) & \langle\rangle * u & \Rightarrow_\sigma & u \\
 & & & \\
(6) & f@(g@u) & \Rightarrow_\sigma & (f; g)@u \\
(7) & f; \langle g, t/x\rangle & \Rightarrow_\sigma & \langle f; g, (f * t)/x\rangle \\
(8) & \langle f, u/x\rangle; \langle\rangle_X & \Rightarrow_\sigma & f; \langle\rangle_{X\setminus x} \quad (x \in X) \\
(9) & \langle f, u/x\rangle; \langle\rangle_X & \Rightarrow_\sigma & \langle f; \langle\rangle_X, u/x\rangle \quad (x \notin X) \\
(10) & \langle\rangle_Y; \langle\rangle_X & \Rightarrow_\sigma & \langle\rangle_{Y\cup X} \\
(11) & \langle\rangle; f & \Rightarrow_\sigma & f \\
(12) & f; \langle\rangle & \Rightarrow_\sigma & f
\end{array}$$

The correctness of the implementation of reduction in the λ-calculus as given by reduction in the $\lambda\sigma$-calculus is now sketched. The key results are that reduction in the $\lambda\sigma$-calculus is confluent and every $\lambda\sigma$-term rewrites to a unique normal form which is a λ-term. These results are proven by defining a reduction preserving embedding of the $\lambda\sigma$-calculus into the λ-calculus — confluence and normalisation can then be lifted from the λ-calculus to the $\lambda\sigma$-calculus. This embedding essentially performs all substitutions embedded within terms — for more details see [20]. The same technique is applied later in this paper to show the corresponding properties of xDILL.

THEOREM 2.2
Every $\lambda\sigma$-calculus term rewrites to a unique β-normal form of the λ-calculus and every $\lambda\sigma$-calculus substitution rewrites to a unique substitution of the form

$$\langle\langle\rangle_X, t_1/x_1, \ldots, t_n/x_n\rangle$$

where each t_i is a β-normal form of the λ-calculus. Hence, the $\lambda\sigma$-calculus is confluent and weakly normalising.

Of course reduction in the $\lambda\sigma$-calculus does not satisfy strong normalisation as Mellies counter-example applies [19].

2.3 Barber's DILL

We use Barber's Dual Intuitionistic Linear Logic (DILL) as our base linear λ-calculus to which explicit substitutions are added. We prefer DILL to ILL [5] as DILL contains the simply typed λ as a subcalculus and hence we have access to both intuitionistic and linear environments. In the case of ILL, one has to encode the simply typed λ-calculus which entails a significant overhead. In this section, we give a brief presentation of the calculus but refer the reader to [3] for details and proofs. The types of DILL are

$$T ::= G \mid I \mid T \otimes T \mid !T \mid T \multimap T$$

where G is a ground type, I is called the *unit*, $A \multimap B$ is the *linear function space*, $A \otimes B$ is called the *tensor* and $!A$ is called an *exponential* or *modality*. The raw expressions of DILL are

$$\begin{array}{rcl} t & ::= & x_L \mid x_I \mid \lambda x{:}\,A.t \mid tt \\ & & \mid \ \mathtt{let}\ t\ \mathtt{be}\ \bullet\ \mathtt{in}\ t \mid \bullet \\ & & \mid \ \mathtt{let}\ t\ \mathtt{be}\ x{\otimes}y\ \mathtt{in}\ t \mid t{\otimes}t \\ & & \mid \ \mathtt{let}\ t\ \mathtt{be}\ !x\ \mathtt{in}\ t \mid\ !t \end{array}$$

where x is a variable. The subscripts I and L are used to indicate whether a variable is to be thought of as linear or intuitionistic — these subscripts are omitted if this causes no confusion. The canonical element of I is denoted by $\bullet$ as the more usual symbol $*$ is used in this paper to represent the application of a substitution to a term. The pre-terms for λ-abstraction and application are standard, while the pre-terms $\bullet, !t$ and $t{\otimes}t'$ follow the pattern of introduction terms in applying a constructor to terms of the relevant component types. Finally the `let`-expressions can be thought of as permitting various cuts, eg the pre-term `let` u `be` $!x$ `in` t is intended to cut a term u of type $!A$ against a variable of type A in the term t. In order to reduce the notational overhead we define a *pattern* by

$$p := \bullet \mid\ !x \mid x \otimes y$$

and write `let` t `be` p `in` u when we wish to treat the `let`-expressions uniformly. A context is of the form $x_1 : A_1, \ldots, x_n : A_n$ where the x's are distinct variables and the A's are types. A DILL context consists of two contexts Γ and Δ containing disjoint sets of variables and is written $\Gamma|\Delta$. We call the variables in Γ *intuitionistic variables*, and the variables in Δ *linear variables*. The typing judgements of DILL are of the form $\Gamma|\Delta \vdash t : A$ and are generated by the inference rules of Table 3.

Weakening for intuitionistic variables is admissible because of the inference rule for typing such variables while contraction of intuitionistic variables follows from the additive way intuitionistic contexts are combined. The structural rule exchange is also admissible. All linear variables occurring in the context of a judgement must occur free exactly once in the associated term. As with traditional λ-calculi, each type constructor is defined by an introduction and elimination rule. The !-type constructor controls the interaction between the linear and intuitionistic parts of the context, and, in particular, explains how linear variables of !-type should be seen as intuitionistic variables.

TABLE 3. Typing Judgements for DILL

$$\Gamma, x{:}\,A, \Gamma' | \_ \vdash x{:}\,A \qquad \Gamma | x{:}\,A \vdash x{:}\,A$$

$$\frac{\Gamma|\Delta, x{:}\,A \vdash t{:}\,B}{\Gamma|\Delta \vdash \lambda x{:}\,A.t{:}\,A \multimap B} \qquad \frac{\Gamma|\Delta_1 \vdash t{:}\,A \multimap B \qquad \Gamma|\Delta_2 \vdash u{:}\,A}{\Gamma|\Delta \vdash tu{:}\,B}$$

$$\Gamma|\_ \vdash \bullet{:}\,I \qquad \frac{\Gamma|\Delta_1 \vdash t{:}\,I \qquad \Gamma|\Delta_2 \vdash u{:}\,A}{\Gamma|\Delta \vdash \texttt{let}\ t\ \texttt{be}\ \bullet\ \texttt{in}\ u{:}\,A}$$

$$\frac{\Gamma|\Delta_1 \vdash t{:}\,A \quad \Gamma|\Delta_2 \vdash u{:}\,B}{\Gamma|\Delta \vdash t \otimes u{:}\,A \otimes B} \qquad \frac{\Gamma|\Delta_1 \vdash t{:}\,A \otimes B \qquad \Gamma|\Delta_2, x{:}\,A, y{:}\,B \vdash u{:}\,C}{\Gamma|\Delta \vdash \texttt{let}\ t\ \texttt{be}\ x \otimes y\ \texttt{in}\ u{:}\,C}$$

$$\frac{\Gamma|\_ \vdash t{:}\,A}{\Gamma|\_ \vdash !t{:}\,!A} \qquad \frac{\Gamma|\Delta_1 \vdash t{:}\,!A \qquad \Gamma, x{:}\,A|\Delta_2 \vdash u{:}\,B}{\Gamma|\Delta \vdash \texttt{let}\ t\ \texttt{be}\ !x\ \texttt{in}\ u{:}\,B}$$

In those rules with multiple premises, Δ_1, Δ_2 are disjoint and Δ is a permutation of Δ_1, Δ_2.

2.4 A Reduction Calculus for DILL

Reduction in DILL consists of a β-redex for each type constructor — in section 3.6 we shall consider the effect of adding *commuting conversions*. Formally, the reduction relation $\Rightarrow_\beta$ is the least congruence on terms containing the basic redexes given in Table 4. The basic results concerning reduction in DILL are summarised in Theorem 2.3 and can be found in [3, 12].

TABLE 4. A Reduction Calculus for DILL

$\texttt{let}\ \bullet\ \texttt{be}\ \bullet\ \texttt{in}\ t$	$\Rightarrow_\beta$	t
$\texttt{let}\ v \otimes u\ \texttt{be}\ x \otimes y\ \texttt{in}\ t$	$\Rightarrow_\beta$	$t[v/x, u/y]$
$(\lambda x{:}\,A.t)u$	$\Rightarrow_\beta$	$t[u/x]$
$\texttt{let}\ !u\ \texttt{be}\ !x\ \texttt{in}\ t$	$\Rightarrow_\beta$	$t[u/x]$

THEOREM 2.3
(i) If there is a typing judgement $\Gamma|\Delta \vdash t : A$ and a rewrite $t \Rightarrow_\beta t'$, then there is also a typing judgement $\Gamma|\Delta \vdash t' : A$.

(ii) The relation $\Rightarrow_\beta$ is strongly normalising and confluent.

PROOF. Subject reduction is proven by induction on typing judgements, strong normalisation is proven by standard reducibility arguments and confluence then follows from local confluence which is established by case analysis. ∎

3 A Linear Calculus of Explicit Substitutions — xDILL

We now combine the $\lambda\sigma$-calculus and DILL to obtain a linear calculus of explicit substitutions which we call xDILL. Before actually presenting xDILL, we discuss the design criteria behind the calculus.

3.1 Design Criteria for xDILL

The calculus xDILL should conform to the general pattern of calculi of explicit substitutions as exemplified by the $\lambda\sigma$-calculus. Hence xDILL should consist of:

- An extension of DILL with a new syntactic category of *explicit substitutions* and associated typing judgements.
- A reduction calculus which decomposes β-reduction in DILL into a modified form of β-reduction and $\Rightarrow_\sigma$-rewrite rules which distribute substitutions over terms.
- An embedding of xDILL into DILL.

In addition, just as the λ-calculus is contained in DILL, so the $\lambda\sigma$-calculus should be contained in xDILL. This containment should be such that $\lambda\sigma$-calculus reductions are mapped to xDILL reductions, thus ensuring that linear abstract machines built from xDILL correctly simulate their intuitionistic counterparts. We may summarise this discussion by the following diagram in which inclusions of subcalculi are given by full lines, while the dotted lines refer to the embeddings involved in the correctness proof of each calculus.

$$
\begin{tikzcd}
\lambda\sigma\text{-calculus} \arrow[r] \arrow[d, dashed, shift right] & \text{xDILL} \arrow[d, dashed, shift right] \\
\lambda\text{-calculus} \arrow[r] \arrow[u, shift right] & \text{DILL} \arrow[u, shift right]
\end{tikzcd}
$$

3.2 The Judgements of xDILL

Formally, the types of xDILL are those of DILL and the raw expressions of xDILL are a strict extension of those of DILL given by

$$
\begin{array}{rcl}
t & ::= & x_L \mid x_I \mid \lambda x{:}A.t \mid tu \mid t\otimes t \mid \bullet \mid\ !t \mid f * t \\
 & & \texttt{let}\ t\ \texttt{be}\ p\ \texttt{in}\ t \\
 & & \\
f & ::= & \langle\rangle_X \mid \langle f, t/x_I\rangle \mid \langle f, t/x_L\rangle \mid f;f \\
 & & \texttt{let}\ t\ \texttt{be}\ p\ \texttt{in}\ f
\end{array}
$$

where x is a variable, X is a set of variables and p a pattern. The notational conventions adopted in Section 2.1.1 for the $\lambda\sigma$-calculus are adopted here. In addition, as in DILL, we may decorate variables with I, L to indicate their intended usage. The terms of xDILL are as expected, *i.e.*, the terms of DILL and the term $f * t$ which represents the application of a substitution to a term. As with $\lambda\sigma$, there is the projection $\langle\rangle_X$ and the sequential composition of substitutions $f; f$. Since DILL has two kinds of variables, intuitionistic and linear, xDILL has separate mechanisms for extending a substitution with the substitution of a term for a linear variable $\langle f, t/x_L\rangle$ and for extending a substitution with the substitution of a term for an intuitionistic variable $\langle f, t/x_I\rangle$. Finally, as discussed in the introduction, we generalise the $\otimes, I$ and !-elimination rules to apply to substitutions as well as terms — this ensures that the isomorphisms underpinning DILL are also valid for xDILL. As we shall see, these substitutions behave differently from the substitutions of the $\lambda\sigma$-calculus and hence we introduce some nomenclature:

- A substitution $\texttt{let}\ t\ \texttt{be}\ p\ \texttt{in}\ f$ is called a *let-substitution*,
- A substitution is a *term substitution* iff it has the same structure as $\lambda\sigma$-substitutions, ie they are given by (where t is an arbitrary xDILL term)

$$g ::= \langle\rangle_X \ \mid\ \langle g, t/x_I\rangle \ \mid\ \langle g, t/x_L\rangle \ \mid\ g; g$$

The contexts of xDILL are the same as those of DILL. The typing judgements of xDILL are of the form $\Gamma|\Delta \vdash t : A$ and $\Gamma|\Delta \vdash f : \Gamma'|\Delta'$ and are given in Table 5. These judgements can be grouped into the following categories:

- **Embedding DILL:** We want DILL to be a subsystem of xDILL and hence all the inference rules of DILL are included in xDILL.
- **Embedding $\lambda\sigma$:** The rules for the substitution $\langle\rangle_X$, extending substitutions and the sequential composition of substitutions are the obvious generalisations of their counterparts in the $\lambda\sigma$-calculus to cover the zonal contexts of DILL and xDILL.
- **Let-Substitutions:** There are three typing judgements for let-substitutions following the same pattern as the three `let`-expressions of DILL.

TABLE 5. xDILL Typing Judgements

The term judgements of xDILL are

$$\frac{\Gamma|\Delta, x\colon A \vdash t\colon B}{\Gamma|\Delta \vdash \lambda x\colon A.t\colon A\multimap B} \qquad \frac{\Gamma|\Delta_1 \vdash t\colon A\multimap B \qquad \Gamma|\Delta_2 \vdash u\colon A}{\Gamma|\Delta \vdash tu\colon B}$$

$$\Gamma|\_ \vdash \bullet\colon I \qquad \frac{\Gamma|\Delta_1 \vdash t\colon I \qquad \Gamma|\Delta_2 \vdash u\colon A}{\Gamma|\Delta \vdash \texttt{let}\ t\ \texttt{be}\ \bullet\ \texttt{in}\ u\colon A}$$

$$\frac{\Gamma|\Delta_1 \vdash t\colon A \quad \Gamma|\Delta_2 \vdash u\colon B}{\Gamma|\Delta \vdash t\otimes u\colon A\otimes B} \qquad \frac{\Gamma|\Delta_1 \vdash t\colon A\otimes B \qquad \Gamma|\Delta_2, x\colon A, y\colon B \vdash u\colon C}{\Gamma|\Delta \vdash \texttt{let}\ t\ \texttt{be}\ x\otimes y\ \texttt{in}\ u\colon C}$$

$$\frac{\Gamma|\_ \vdash t\colon A}{\Gamma|\_ \vdash !t\colon !A} \qquad \frac{\Gamma|\Delta_1 \vdash t\colon !A \qquad \Gamma, x\colon A|\Delta_2 \vdash u\colon B}{\Gamma|\Delta \vdash \texttt{let}\ t\ \texttt{be}\ !x\ \texttt{in}\ u\colon B}$$

$$\frac{\Gamma_1|\Delta_1 \vdash f\colon \Gamma_2|\Delta_2 \qquad \Gamma_2|\Delta_2 \vdash t\colon A}{\Gamma_1|\Delta_1 \vdash f * t\colon A}$$

The substitution judgements of xDILL are

$$\frac{}{\Gamma|\Delta \vdash \langle\rangle_X : \Gamma \setminus X|\Delta}$$

$$\frac{\Gamma|\Delta \vdash f : \Gamma'|\Delta' \qquad \Gamma|\_ \vdash t\colon A}{\Gamma|\Delta \vdash \langle f, t/x_I\rangle\colon \Gamma', x\colon A|\Delta'} \qquad \frac{\Gamma|\Delta_1 \vdash f : \Gamma'|\Delta' \qquad \Gamma|\Delta_2 \vdash t\colon A}{\Gamma|\Delta \vdash \langle f, t/x_L\rangle\colon \Gamma'|\Delta', x\colon A}$$

$$\frac{\Gamma_1|\Delta_1 \vdash f\colon \Gamma_2|\Delta_2 \qquad \Gamma_2|\Delta_2 \vdash g\colon \Gamma_3|\Delta_3}{\Gamma_1|\Delta_1 \vdash f; g\colon \Gamma_3|\Delta_3}$$

$$\frac{\Gamma|\Delta_1 \vdash u\colon A\otimes B \qquad \Gamma|\Delta_2, x\colon A, y\colon B \vdash f\colon \Gamma'|\Delta'}{\Gamma|\Delta \vdash \texttt{let}\ u\ \texttt{be}\ x\otimes y\ \texttt{in}\ f\colon \Gamma'|\Delta'}$$

$$\frac{\Gamma|\Delta_1 \vdash t\colon !A \qquad \Gamma, x\colon A|\Delta_2 \vdash f\colon \Gamma'|\Delta'}{\Gamma|\Delta \vdash \texttt{let}\ t\ \texttt{be}\ !x\ \texttt{in}\ f\colon \Gamma'|\Delta'}$$

$$\frac{\Gamma|\Delta_1 \vdash t\colon I \qquad \Gamma|\Delta_2 \vdash f\colon \Gamma'|\Delta'}{\Gamma|\Delta \vdash \texttt{let}\ t\ \texttt{be}\ \bullet\ \texttt{in}\ f\colon \Gamma'|\Delta'}$$

The rules for substitutions assume x, y are fresh and, where applicable, Δ_1, Δ_2 are disjoint and Δ is a permutation of Δ_1, Δ_2.

Note that weakening and contraction are admissible for intuitionistic variables while linear variables occurring in the context of a judgement must occur free exactly once in the associated term or substitution.

Before we investigate the properties of xDILL any further, we have to deal with α-equivalence. As as already mentioned, in calculi of explicit substitution α-equivalence is not entirely trivial. The next subsection shows how to justify Barendregt's variable convention also for xDILL. It is an extension of the results presented in [20].

3.3 α-equivalence in xDILL

We start by identifying free variables and, for substitutions, the variables for which the substitution provides terms to be substituted.

DEFINITION 3.1
For raw terms t and raw substitutions f we define the sets $\mathtt{FV}(t)$ and $\mathtt{FV}(f)$ of free variables and $\mathtt{SV}(t)$ and $\mathtt{SV}(f)$ of substitution variables by induction over the structure of raw expressions, where for a pattern p the set of variables bound by the `let`-expression is also denoted by p:

$$\begin{array}{lrcl}
(x) & \mathtt{FV}(x) & = & \{x\} \\
(\lambda) & \mathtt{FV}(\lambda x{:}\,A.t) & = & \mathtt{FV}(t) \setminus \{x\} \\
(ts) & \mathtt{FV}(ts) & = & \mathtt{FV}(t) \cup \mathtt{FV}(s) \\
(t\otimes t) & \mathtt{FV}(t\otimes s) & = & \mathtt{FV}(t) \cup \mathtt{FV}(s) \\
(f * t) & \mathtt{FV}(f * t) & = & \mathtt{FV}(f) \cup (\mathtt{FV}(t) \setminus \mathtt{SV}(f)) \\
(\mathtt{let}) & \mathtt{let}\ t\ \mathtt{be}\ p\ \mathtt{in}\ s & = & (\mathtt{FV}(s) \setminus p) \cup \mathtt{FV}(t) \\
(\langle\rangle) & \mathtt{FV}(\langle\rangle_X) & = & X \\
(\langle -,- \rangle) & \mathtt{FV}(\langle f, t/x\rangle) & = & \mathtt{FV}(f) \cup \mathtt{FV}(t) \\
(;) & \mathtt{FV}(f;g) & = & \mathtt{FV}(f) \cup (\mathtt{FV}(g) \setminus \mathtt{SV}(f))
\end{array}$$

$$\begin{array}{lrcl}
(\langle\rangle) & \mathtt{SV}(\langle\rangle_X) & = & \emptyset \\
(\langle -,- \rangle) & \mathtt{SV}(\langle f, t/x\rangle) & = & \mathtt{SV}(f) \cup \{x\} \\
(;\langle\rangle) & \mathtt{SV}(f;\langle\rangle_X) & = & \mathtt{SV}(f) \setminus X \\
(;\langle -,- \rangle) & \mathtt{SV}(f;\langle g, t/x\rangle) & = & \mathtt{SV}(f;g) \cup \{x\} \\
(;;) & \mathtt{SV}(f;(g;h)) & = & \mathtt{SV}((f;g);h)
\end{array}$$

A variable occurring in t is called *bound* in the term t if it is not a free variable in t. A variable occurring in f is called *bound* in the substitution f if it is neither a free variable nor a substitution variable in f.

The definition of α-equivalence requires the definition of renaming.

DEFINITION 3.2
(i) We define the name-changing substitution $f\{y/x\}$ by induction over the structure of f as follows, where V is the set of forbidden variables and $y \notin V$:

$$\begin{array}{rcll}
\langle\rangle_X\{y/x\}_V & = & \langle\rangle_X & \\
(f;g)\{y/x\}_V & = & \left\{\begin{array}{ll} f;(g\{y/x\})_V & \text{if } x \in \mathtt{SV}(g) \\ f\{z/y\}_{V\cup\mathtt{SV}(f)\cup\mathtt{FV}(g)}\{y/x\}_V; & \\ \quad g[z/y]_{V\cup\mathtt{SV}(f)\cup\mathtt{FV}(g)}[y/x]_V & \text{if } x \notin \mathtt{SV}(g) \text{ and } x \in \mathtt{SV}(f) \\ f & \text{if } x \notin \mathtt{SV}(g) \text{ and } x \notin \mathtt{SV}(f) \end{array}\right. & \\
\langle f, t/x\rangle\{y/x\}_V & = & \langle f, t/y\rangle & \\
\langle f, t/z\rangle\{y/x\}_V & = & \langle f\{y/x\}_V, t/z\rangle \text{ if } x \neq y &
\end{array}$$

(ii) We define the renaming of the variable x by the variable y in t or f with forbidden variables V such that $y \notin V$, written $t[y/x]_V$ and $f[y/x]_V$ and $z \notin V \cup \{x, y\}$, by induction over the structure of raw expressions.

$$
\begin{array}{rcl}
x[y/x]_V & = & y \\
z[y/x]_V & = & z \text{ if } z \neq x \\
(\lambda x{:}\, A.t)[y/x]_V & = & \lambda x{:}\, A.t \\
(\lambda y{:}\, A.t)[y/x] & = & \lambda z{:}\, A.t[z/y]_{V \cup \{y\}}[y/x]_V \;\; (\, z \notin V \cup \{y\}) \\
(\lambda z{:}\, A.t)[y/x] & = & \lambda z{:}\, A.t[y/x]_{V \cup \{z\}} \;\; (\, z \notin V \cup \{x, y\}) \\
(tu)[y/x]_V & = & (t[y/x]_V)(u[y/x]_V) \\
(t \otimes u)[y/x]_V & = & (t[y/x]_V) \otimes (u[y/x]_V) \\
(f * t)[y/x] & = & \left\{ \begin{array}{ll} f\{z/y\}_{V \cup \mathtt{SV}(f) \cup \mathtt{FV}(t)}[y/x]_V * & \\ \quad t[z/y]_{V \cup \mathtt{SV}(f) \cup \mathtt{FV}(t)} & \text{if } x \in \mathtt{SV}(f) \\ f\{z/y\}_{V \cup \mathtt{SV}(f) \cup \mathtt{FV}(t)}[y/x]_V * & \\ \quad (t[z/y]_{V \cup \mathtt{SV}(f) \cup \mathtt{FV}(t)})[y/x]_V & \text{otherwise} \end{array} \right. \\
(\mathtt{let}\ t\ \mathtt{be}\ p\ \mathtt{in}\ s)[y/x]_V & = & \left\{ \begin{array}{ll} \mathtt{let}\ t[y/x]_V\ \mathtt{be}\ p\ \mathtt{in}\ s & \text{if } x \in p \\ \mathtt{let}\ t[y/x]\ \mathtt{be}\ p'\ \mathtt{in}\ s[z/y]_{V \cup \{y\}}[y/x]_V & \\ \qquad \text{if } y \in p \text{, and } p' \text{ is } p \text{ with } y \text{ replaced by } z & \\ \mathtt{let}\ t[y/x]_V\ \mathtt{be}\ p\ \mathtt{in}\ s[y/x]_V & \text{otherwise} \end{array} \right. \\
\langle\rangle_{X,x}[y/x]_V & = & \langle\rangle_{X,y} \\
\langle\rangle_X[y/x]_V & = & \langle\rangle_X \text{ if } x \notin X \\
\langle f, t/z\rangle[y/x]_V & = & \langle f[y/x]_V, t[y/x]_V/z\rangle \\
(f; g)[y/x]_V & = & \left\{ \begin{array}{ll} f\{z/y\}_{V \cup \mathtt{SV}(f) \cup \mathtt{FV}(g)}[y/x]_V; & \\ \quad g[z/y]_{V \cup \mathtt{SV}(f) \cup \mathtt{FV}(g)} & \text{if } x \in \mathtt{SV}(f) \\ f\{z/y\}_{V \cup \mathtt{SV}(f) \cup \mathtt{FV}(g)}[y/x]_V; & \\ \quad (g[z/y]_{V \cup \mathtt{SV}(f) \cup \mathtt{FV}(g)})[y/x]_V & \text{otherwise} \end{array} \right.
\end{array}
$$

Note that renaming is not a function from expressions to expressions but only a function from expressions to set of expressions.

Now we can define α-equivalence. Because the operations $*$ and ; do not indicate the bound variable as part of the syntax, we have to determine the bound variables from the terms. This explains the conditions in the second and third clause of the definition.

Definition 3.3

We define α-equivalence in xDILL to be the smallest congruence relation on raw expressions including

$$
\begin{array}{rcl}
\lambda x{:}\, A.t & \equiv_\alpha & \lambda y{:}\, A.t[y/x]_{\mathtt{FV}(t)} \\
f * t & \equiv_\alpha & f\{y/x\}_{\mathtt{FV}(t) \cup \mathtt{SV}(f)} * t[y/x]_{\mathtt{FV}(t) \cup \mathtt{SV}(f)} \text{ (if } x \in \mathtt{SV}(f)) \\
f; g & \equiv_\alpha & f\{y/x\}_{\mathtt{FV}(g) \cup \mathtt{SV}(f)}; g[y/x]_{\mathtt{FV}(g) \cup \mathtt{SV}(f)} \\
 & & \text{(if } x \in (\mathtt{SV}(f) \setminus (\mathtt{SV}(f; g)) \cup (\mathtt{SV}(f) \cap \mathtt{SV}(g)))
\end{array}
$$

The free and substituted variables are the same for any two α-equivalent expressions. Moreover, renaming respects α-equivalence.

LEMMA 3.4
Assume t and t' as well as f and f' are α-equivalent.

(i) t and t' as well as f and f' have the same free variables, *i.e.*, $\mathtt{FV}(t) = \mathtt{FV}(t')$ and $\mathtt{FV}(f) = \mathtt{FV}(f')$;
(ii) f and f' have the same substituted variables, *i.e.*, $\mathtt{SV}(f) = \mathtt{SV}(f')$;
(iii) $t[y/x]_V \equiv_\alpha t'[y/x]_V$, $f[y/x]_V \equiv_\alpha f'[y/x]_V$ and $f\{y/x\}_V \equiv_\alpha f'\{y/x\}_V$.

PROOF. Induction over the structure of f and t. ∎

The last clause of the lemma shows in particular that all expressions which are obtained by renaming are contained in the same α-equivalence class.

Next we show that typing respects α-equivalence. We start by proving that the free and substituted variables are always part of the context used in the typing judgements. Because there is no difference in the behaviour of linear and intuitionistic variables with respect to renaming and α-equivalence, we omit the separation between intuitionistic and linear contexts and write in the rest of this subsection simply Γ for a context $\Gamma'|\Delta'$.

LEMMA 3.5
(i) Assume $\Gamma \vdash t\colon A$ and $\Gamma \vdash f\colon \Delta$. Then the free variables of t and f are contained in the variables declared in Γ, and the substitution variables of f are contained in Δ. Moreover, all variables in Δ are either substitution variables of f or variables contained in Γ.
(ii) Assume $\Gamma \vdash f\colon \Gamma'$, $\Gamma' \vdash g\colon \Delta$ and $x \in \mathtt{SV}(f)$. If x is contained in Δ, then $x \in \mathtt{SV}(f;g)$.

PROOF. Easy induction over the structure of the derivation of f and t. ∎

Now we show that renaming preserves typing.

LEMMA 3.6
Consider a term t, a substitution f, contexts Γ, Δ and sets of variables V and W such that V contains at least the variables declared in Γ and W at least the set of variables declared in Γ and Δ.

(i) $\Gamma \vdash t\colon A$ implies $\Gamma[y/x] \vdash t[y/x]_V\colon A$;
(ii) $\Gamma \vdash f\colon \Delta$ implies $\Gamma[y/x] \vdash f[y/x]_V\colon \Delta$ whenever $x \in \mathtt{SV}(f)$;
(iii) $\Gamma \vdash f\colon \Delta$ implies $\Gamma[y/x] \vdash f[y/x]_V\colon \Delta[y/x]$ if $x \notin \mathtt{SV}(f)$;
(iv) $\Gamma \vdash f\colon \Delta$ implies $\Gamma \vdash f\{y/x\}_W\colon \Delta[y/x]$ if $x \in \mathtt{SV}(f)$;
(v) $\Gamma \vdash f\colon \Delta$ iff $\Gamma \vdash f\{y/x\}_W\colon \Delta$ if $x \notin \mathtt{SV}(f)$.

PROOF. Simultaneous induction over the structure of f and t. ∎

Now we can show the main theorem, which states that typing is invariant under α-equivalence.

THEOREM 3.7
Assume that t_1 and t_2 are two α-equivalent xDILL-terms, and assume that f_1 and f_2 are two α-equivalent substitutions.

(i) If $\Gamma \vdash t_1 : A$, then also $\Gamma \vdash t_2 : A$, and similarly if $\Gamma \vdash f_1 : \Delta$, then also $\Gamma \vdash f_2 : \Delta$.
(ii) If t and s are α-equivalent terms in the λ-calculus, then they are α-equivalent in xDILL, too.

PROOF. Induction over derivations, using the previous lemma as a substitution lemma. ∎

This theorem justifies Barendregt's variable convention: whenever necessary we can rename bound variables without changing the typing.

3.4 Reductions in xDILL

We now equip xDILL with a reduction calculus. Reduction in the $\lambda\sigma$-calculus consists of a modified form of β-reduction $(\lambda x.t)u \Rightarrow \langle u/x\rangle * t$ which creates an explicit substitution and a set of $\Rightarrow_\sigma$-rewrite rules which control how substitutions are evaluated. Reduction in xDILL is made more complex by the presence of let-substitutions and the constraints of a linear framework. We now detail these problems and our proposed solutions.

- **Let-Substitutions:** In xDILL the $\Rightarrow_\sigma$ rewrite rules cannot be used without restriction, eg the rewrite rule $f*!t \;\Rightarrow_\sigma\; !(f*t)$ is unsound [3] and does not satisfy subject reduction. The solution is to restrict some of the $\Rightarrow_\sigma$-rewrite rules to apply only to what we call *term substitutions* and leave the *let-substitutions* unaffected.
- **Commuting Conversions:** The restriction of $\Rightarrow_\sigma$-rules to apply to only term substitutions means that reduction must posses a mechanism for extracting term-substitutions from arbitrary substitutions. This is achieved via the introduction of three new commuting conversions (the symbol @ represents either ; or $*$)

$$\begin{array}{rcl} \langle\rangle_X ; \texttt{let}\ t\ \texttt{be}\ p\ \texttt{in}\ f & \Rightarrow_\sigma & \texttt{let}\ \langle\rangle_X * t\ \texttt{be}\ p\ \texttt{in}\ \langle\rangle_X ; f \\ \langle \texttt{let}\ t\ \texttt{be}\ p\ \texttt{in}\ f, u/x\rangle & \Rightarrow_\sigma & \texttt{let}\ t\ \texttt{be}\ p\ \texttt{in}\ \langle f, u/x\rangle \\ (\texttt{let}\ t\ \texttt{be}\ p\ \texttt{in}\ f)@h & \Rightarrow_\sigma & \texttt{let}\ t\ \texttt{be}\ p\ \texttt{in}\ (f@h) \end{array}$$

  which help ensure that the normal forms of substitutions are of the form

$$\texttt{let}\ t_1\ \texttt{be}\ p_1\ \texttt{in}\ (\texttt{let}\ t_2\ \texttt{be}\ p_2\ \texttt{in}\ \cdots \texttt{let}\ t_m\ \texttt{be}\ p_m\ \texttt{in}\ \langle\langle\rangle_X, u_1/x_1, \ldots, u_n/x_n\rangle)$$

[3] the comonad in the term model interpreting the !-type constructor becomes idempotent.

- **Linearity Constraints:** Some $\lambda\sigma$-rewrite rules are inherently non-linear and hence require reformulation, eg the rewrite rule $f*(uv) \Rightarrow_\sigma (f*u)(f*v)$ duplicates the free variables of f and hence does not satisfy subject reduction in xDILL. Therefore we partition the substitution f into that part which applies to u, denoted f_u, and that part which applies to v, denoted f_v, and replace the above rewrite rule by

$$f*(uv) \Rightarrow_\sigma (f_u * u)(f_v * v)$$

As remarked at the beginning of this section, only term substitutions are distributed through terms and hence this restriction of a substitution needs to be defined only for term substitutions. Formally, we denote the restriction of a term-substitution f to a set of variables X by $f|_X$ and define this to be [4]

$$\begin{array}{lcll}
(\langle\rangle_Y)|_X & = & \langle\rangle_Y & \\
\langle f, t/x\rangle|_X & = & \langle f|_X, t/x\rangle & \text{if } x \in X \setminus x \\
\langle f, t/x\rangle|_X & = & f|_X & \text{if } x \notin X \\
(f;g)|_X & = & f|_{\mathtt{FV}(g|_X)}; g|_X &
\end{array}$$

We write f_u for $f|_{\mathtt{FV}(u)}$.

[4] Since intuitionistic variables can be copied, X may be restricted to only linear variables

The reduction calculus for xDILL is defined in Table 6. By Barendregt's variable con-

TABLE 6. Reductions for xDILL

Let h be either a term or a substitution and @ be ; or $*$ appropriately.

- Modified β-reduction:

$$\begin{array}{lll}
(\lambda x{:}\,A.t)u & \Rightarrow_\beta & \langle u/x_L\rangle * t \\
\mathtt{let}\ \bullet\ \mathtt{be}\ \bullet\ \mathtt{in}\ h & \Rightarrow_\beta & h \\
\mathtt{let}\ (v\otimes u)\ \mathtt{be}\ (x\otimes y)\ \mathtt{in}\ h & \Rightarrow_\beta & \langle v/x_L, u/y_L\rangle @ h \\
\mathtt{let}\ !u\ \mathtt{be}\ !x\ \mathtt{in}\ h & \Rightarrow_\beta & \langle u/x_I\rangle @ h
\end{array}$$

- Normalising Substitutions: In the first two rules, f is a term substitution

$$\begin{array}{llll}
f@(g@u) & \Rightarrow_\sigma & (f;g)@u & \\
f;\langle g, t/x\rangle & \Rightarrow_\sigma & \langle f_g; g, (f_t * t)/x\rangle & \\
\langle\rangle_X;\langle\rangle_Y & \Rightarrow_\sigma & \langle\rangle_{X\cup Y} & \\
\langle f, t/x\rangle;\langle\rangle_X & \Rightarrow_\sigma & f;\langle\rangle_{X\backslash x} & (x\in X) \\
\langle f, t/x\rangle;\langle\rangle_X & \Rightarrow_\sigma & \langle f;\langle\rangle_X, t/x\rangle & (x\notin X)
\end{array}$$

$$\begin{array}{lll}
\langle\rangle_X;\mathtt{let}\ t\ \mathtt{be}\ p\ \mathtt{in}\ f & \Rightarrow_\sigma & \mathtt{let}\ \langle\rangle_X * t\ \mathtt{be}\ p\ \mathtt{in}\ \langle\rangle_X; f \\
\langle \mathtt{let}\ t\ \mathtt{be}\ p\ \mathtt{in}\ f, u/x\rangle & \Rightarrow_\sigma & \mathtt{let}\ t\ \mathtt{be}\ p\ \mathtt{in}\ \langle f, u/x\rangle \\
(\mathtt{let}\ t\ \mathtt{be}\ p\ \mathtt{in}\ f)@h & \Rightarrow_\sigma & \mathtt{let}\ t\ \mathtt{be}\ p\ \mathtt{in}\ (f@h)
\end{array}$$

- Normalising Terms: Let f be a term substitution

$$\begin{array}{lll}
f@\mathtt{let}\ t\ \mathtt{be}\ p\ \mathtt{in}\ h & \Rightarrow_\sigma & \mathtt{let}\ (f_t * t)\ \mathtt{be}\ p\ \mathtt{in}\ f_h@h \\
f*!u & \Rightarrow_\sigma & !(f*u) \\
f*(u\otimes v) & \Rightarrow_\sigma & (f_u*u)\otimes(f_v*v) \\
f*\bullet & \Rightarrow_\sigma & \bullet \\
f*\lambda y{:}\,A.u & \Rightarrow_\sigma & \lambda z{:}\,A.(\langle f, z/y\rangle)*u \\
f*uv & \Rightarrow_\sigma & (f_u*u)(f_v*v) \\
\langle f, t/x\rangle * x & \Rightarrow_\sigma & t \\
\langle f, t/y\rangle * x & \Rightarrow_\sigma & f*x \\
\langle\rangle_X * t & \Rightarrow_\sigma & t
\end{array}$$

The rewrite relation $\Rightarrow$ is defined to be $\Rightarrow_\beta\cup\Rightarrow_\sigma$

vention, the xDILL reduction rules do not allow variable capture. As with reduction in the $\lambda\sigma$-calculus, the redexes in xDILL may be classified as follows:

- **β-redexes:** There are 4 modified β-redexes, one for each connective of xDILL.
- **Substitution Normalisation:** Reduction in $\lambda\sigma$ uses five redexes to normalise substitutions. xDILL contains these redexes and a further three commuting conversions to normalise substitutions containing `let`-expressions to substitutions which are a series of `let`-expressions applied to a term substitution:

$$\mathtt{let}\ t_1\ \mathtt{be}\ p_1\ \mathtt{in}\ (\mathtt{let}\ t_2\ \mathtt{be}\ p_2\ \mathtt{in}\ \cdots \mathtt{let}\ t_n\ \mathtt{be}\ p_n\ \mathtt{in}\ \langle\langle\rangle_X, u_1/x_1,\ldots,u_n/x_n\rangle)$$

- **Term Normalisation:** As with the $\lambda\sigma$-calculus, xDILL contains a redex for pushing substitutions through each term constructor, a pair of redexes for evaluating the result of applying a substitution to a variable and a redex for deleting any residual empty substitutions. As we have explained before, these redexes must be restricted to apply only to term substitutions which are extracted from arbitrary substitutions by the three new commuting conversions mentioned above. Some of these redexes have also been linearised to ensure that reduction satisfies the subject reduction property.

LEMMA 3.8 (Subject reduction)
If $\Gamma|\Delta \vdash t : A$ and $t \Rightarrow t'$, then $\Gamma|\Delta \vdash t' : A$. Similarly, if $\Gamma|\Delta \vdash f : \Gamma'|\Delta'$ and $f \Rightarrow f'$, then $\Gamma|\Delta \vdash f' : \Gamma'|\Delta'$.

PROOF. The proof is by induction on the structure of expressions and we treat two cases:

- For the rule $f{*}!t \Rightarrow_\sigma !(f * t)$, observe that for the left-hand side to type-check, t contains no free linear variables. Thus the typing judgement of f must be of the form $\Gamma|\Delta \vdash f : \Gamma'|\_$. Since f is a term substitution, this forces Δ to be empty and hence the right-hand side type-checks.
- If the left-hand side of $\langle f, t/x\rangle * y \Rightarrow_\sigma f * y$ type-checks, then x is intuitionistic, t contains no linear variables and hence no linear variables are actually discarded by this rewrite rule.

∎

The other obvious property required of xDILL reduction is that it implements substitution.

LEMMA 3.9
If t is a term and $\langle u_1/x_1, \ldots, u_n/x_n\rangle$ is a substitution, then there is a reduction $\langle u_1/x_1, \ldots, u_n/x_n\rangle * t \Rightarrow_\sigma^* t[u_1/x_1, \ldots, u_n/x_n]$.

PROOF. Since the substitution $\langle u_1/x_1, \ldots, u_n/x_n\rangle$ is a term substitution it can be distributed over all term constructors and then evaluated correctly at variables. ∎

3.5 Correctness of xDILL

The correctness proof for the $\lambda\sigma$-calculus constructs a reduction preserving embedding of the $\lambda\sigma$-calculus into the λ-calculus — this embedding performs all explicit substitutions. Normalisation and confluence for the $\lambda\sigma$-calculus can then be derived from the equivalent results for the λ-calculus. The correctness proof for xDILL is essentially the same as we construct a reduction preserving embedding from the terms of xDILL to the terms of DILL.

An xDILL term is called *canonical* iff it is a DILL term, while an xDILL substitution is called *canonical* iff it is of the form

$$\texttt{let } t_1 \texttt{ be } p_1 \texttt{ in } (\cdots \texttt{let } t_n \texttt{ be } p_n \texttt{ in } \langle u_1/x_1, \ldots, u_m/x_m\rangle)$$

where the terms t_i and u_i are canonical. As a notational convenience, we denote multiple `let`-expressions such as the one above by

$$\texttt{let } t_1, \cdots, t_n \texttt{ be } p_1, \cdots, p_n \texttt{ in } u$$

However, we emphasise that such expressions are *not* the parallel `let`-expressions which occur in the literature, eg the pattern p_i may bind some of the free variables of t_{i+1}. The translation of xDILL terms and substitutions into canonical terms and canonical substitutions is given in Table 7.

TABLE 7. Translating terms to Canonical Terms

Let $[\![f]\!] = \texttt{let } \vec{t_i} \texttt{ be } \vec{p_i} \texttt{ in } \langle \vec{u_j}/\vec{x_j}\rangle$

- On Substitutions

$$\begin{array}{rcl}
[\![\langle\rangle_X]\!] & = & \langle\rangle_X \\
[\![\langle f, u/x\rangle]\!] & = & \texttt{let } \vec{t_i} \texttt{ be } \vec{p_i} \texttt{ in } \langle \vec{u_j}/\vec{x}_j, [\![u]\!]/x\rangle \\
[\![f; g]\!] & = & \texttt{let } \vec{t_i} \texttt{ be } \vec{p_i} \texttt{ in } [\![g]\!][\vec{u_j}/\vec{x}_j] \\
[\![(\texttt{let } t \texttt{ be } p \texttt{ in } f)]\!] & = & \texttt{let } [\![t]\!] \texttt{ be } p \texttt{ in } [\![f]\!]
\end{array}$$

- On terms

$$\begin{array}{rcl}
[\![x]\!] & = & x \\
[\![\bullet]\!] & = & \bullet \\
[\![\lambda x\colon A.t]\!] & = & \lambda x\colon A.[\![t]\!] \\
[\![ts]\!] & = & [\![t]\!][\![s]\!] \\
[\![!t]\!] & = & ![\![t]\!] \\
[\![u \otimes v]\!] & = & [\![t]\!] \otimes [\![s]\!] \\
[\![(\texttt{let } t \texttt{ be } p \texttt{ in } u)]\!] & = & \texttt{let } [\![t]\!] \texttt{ be } p \texttt{ in } [\![u]\!] \\
[\![f * t]\!] & = & \texttt{let } \vec{t_i} \texttt{ be } \vec{p_i} \texttt{ in } [\![t]\!][\vec{u_j}/\vec{x}_j]
\end{array}$$

LEMMA 3.10
Given judgements $\Gamma|\Delta \vdash f : \Gamma'|\Delta'$ and $\Gamma|\Delta \vdash t : A$,

- There are rewrites $f \Rightarrow_\sigma^* [\![f]\!]$ and $t \Rightarrow_\sigma^* [\![t]\!]$.
- There are typing judgements $\Gamma|\Delta \vdash [\![f]\!] : \Gamma'|\Delta'$ and $\Gamma|\Delta \vdash [\![t]\!] : A$.
- Given a reduction $t \Rightarrow_\sigma u$, then $[\![t]\!] = [\![u]\!]$. Similarly if $f \Rightarrow_\sigma g$, then $[\![f]\!] = [\![g]\!]$.

- $[\![f]\!]$ and $[\![t]\!]$ are canonical and if h is a canonical, then $[\![h]\!] = h$.
- There is an equality $[\![u[v/x]]\!] = [\![u]\!][[\![v]\!]/x]$.

PROOF. The first four parts of the lemma are proven by induction and using subject reduction for xDILL as established in lemma 3.8. For the final part

$$[\![u[v/x]]\!] = [\![[\![u]\!][[\![v]\!]/x]]\!] = [\![u]\!][[\![v]\!]/x]$$

where the first equality holds because $u \Rightarrow_\sigma^* [\![u]\!]$ and $v \Rightarrow_\sigma^* [\![v]\!]$ and the second equality holds because $[\![u]\!][[\![v]\!]/x]$ is canonical and we have just proven that canonical terms are translated by $[\![.]\!]$ to themselves. ∎

As a corollary we have $\Rightarrow_\sigma$ is confluent and normalising to the canonical terms. Extending these results to the full reduction relation on xDILL requires a characterisation of the interaction between the embedding $[\![.]\!]$ and β-reduction. Firstly we must characterise where a β-reduction can occur within a substitution.

LEMMA 3.11
Given a substitution f such that $f \Rightarrow_\beta f'$, then one of the following holds

- $[\![f]\!] = \langle \vec{u_j}/\vec{x_j}\rangle$, $[\![f']\!] = \langle \vec{v_j}/\vec{x_j}\rangle$ and $u_j \Rightarrow_\beta^* v_j$
- $[\![f]\!] = \texttt{let}\ t\ \texttt{be}\ p\ \texttt{in}\ u$, $[\![f']\!] = \texttt{let}\ t'\ \texttt{be}\ p\ \texttt{in}\ u$ and $t \Rightarrow_\beta^* t'$
- $[\![f]\!] = \texttt{let}\ t\ \texttt{be}\ p\ \texttt{in}\ u$, $[\![f']\!] = \texttt{let}\ t\ \texttt{be}\ p\ \texttt{in}\ u'$ and $u \Rightarrow_\beta^* u'$
- $[\![f]\!] = \texttt{let}\ t\ \texttt{be}\ p\ \texttt{in}\ u$, t is an introduction term and f' is the result of contracting the top-level β-redex

Hence there is a DILL-reduction sequence $[\![f]\!] \Rightarrow_\beta^* [\![f']\!]$.

PROOF. By case analysis of the reduction $f \Rightarrow_\beta f'$. ∎

Finally we can prove confluence and normalisation.

THEOREM 3.12
The relation $\Rightarrow$ is confluent and normalising on both terms and substitutions.

PROOF. Given two terms u and v related in the xDILL equational theory, we have by lemma 3.10 that $[\![u]\!]$ and $[\![v]\!]$ are related in the equational theory generated by β-reduction on DILL. From theorem 2.3, these terms have a common reduct t and by lemma 3.10 both u and v reduce to t. Lemma 3.9 and 3.10 guarantee that all terms t normalise to the β-normal form in DILL of $[\![t]\!]$. The argument for substitutions is analogous.

∎

In fact, the above proof also characterises normal forms for xDILL-terms and xDILL-substitutions.

THEOREM 3.13
Every xDILL term reduces to a unique normal form which is an DILL-normal form and every xDILL substitution reduces to a unique normal form of the form

$$\texttt{let}\ t_1\ \texttt{be}\ p_1\ \texttt{in}\ (\texttt{let}\ t_2\ \texttt{be}\ p_2\ \texttt{in}\ \cdots \texttt{let}\ t_n\ \texttt{be}\ p_n\ \texttt{in}\ \langle\langle\rangle_X, u_1/x_1, \ldots, u_n/x_n\rangle)$$

where the t_i and u_i are xDILL-normal terms.

3.6 Commuting Conversions

Commuting conversions are semantically valid reductions which arise whenever `let`-expressions are used. Commuting conversions may be classified into two forms:

- *Computational Commuting Conversions* which may lead to the discovery of new β-redexes, eg the following reduction rule can create a β-redex when performed on the first argument of an application term.

$$\texttt{let}\ t\ \texttt{be}\ !x\ \texttt{in}\ \lambda y.u \quad \Rightarrow \quad \lambda y.\texttt{let}\ t\ \texttt{be}\ !x\ \texttt{in}\ u$$

- *Non-computational Commuting Conversions* which do not discover new β-redexes, eg the following rewrite rule simply permutes the order in which the `let`-expressions occur.

$$\begin{array}{l}\texttt{let}\ t\ \texttt{be}\ !x\ \texttt{in}\ (\texttt{let}\ t'\ \texttt{be}\ !x'\ \texttt{in}\ u) \Rightarrow \\ \qquad \texttt{let}\ t'\ \texttt{be}\ !x'\ \texttt{in}\ (\texttt{let}\ t\ \texttt{be}\ !x\ \texttt{in}\ u)\end{array}$$

Clearly any reduction machine should be able to detect all β-redexes and hence we now add the computational commuting conversions to reduction in DILL and xDILL and then prove that reduction in xDILL is still confluent and normalising to unique DILL normal forms — later we shall comment briefly on the non-computational commuting conversions. Our first lemma details exactly what are the computational commuting conversions.

LEMMA 3.14
Together with β-reduction, the following DILL redexes are sufficient to locate all β-redexes in DILL.

$$\begin{array}{lll}\texttt{let}\ t\ \texttt{be}\ p\ \texttt{in}\ \lambda x.u & \Rightarrow_{cc} & \lambda x.\texttt{let}\ t\ \texttt{be}\ p\ \texttt{in}\ u \\ \texttt{let}\ (\texttt{let}\ w\ \texttt{be}\ s\ \texttt{in}\ u)\ \texttt{be}\ p\ \texttt{in}\ n & \Rightarrow_{cc} & \texttt{let}\ w\ \texttt{be}\ s\ \texttt{in}\ (\texttt{let}\ u\ \texttt{be}\ p\ \texttt{in}\ n)\end{array} \qquad (3.1)$$

PROOF. The proof rests on an analysis of equality, and specifically η-conversion in DILL [12]. If a term is normal with respect to the computational commuting conversions and β-reduction, then applying any non-computational commuting conversions does not create any new β-redexes. ∎

Before proceeding we must prove that DILL extended with these commuting conversions is still strongly normalising and confluent. Proofs can be found in Barber's thesis [3].

THEOREM 3.15
The relation $\Rightarrow_{cc} \cup \Rightarrow_{\beta}$ on DILL is SN and CR.

The computational commuting conversions of xDILL are only slightly more general than those of DILL. In particular, the presence of let-substitutions means that the second of the computational commuting conversions in equation 3.1 must be generalised to cover substitutions as well as terms. Thus the xDILL computational commuting conversions are

$$\begin{array}{lll} \mathtt{let}\ t\ \mathtt{be}\ p\ \mathtt{in}\ \lambda x.u & \Rightarrow_{cc} & \lambda x.\mathtt{let}\ t\ \mathtt{be}\ p\ \mathtt{in}\ u \\ \mathtt{let}\ (\mathtt{let}\ w\ \mathtt{be}\ s\ \mathtt{in}\ u)\ \mathtt{be}\ p\ \mathtt{in}\ h & \Rightarrow_{cc} & \mathtt{let}\ w\ \mathtt{be}\ s\ \mathtt{in}\ (\mathtt{let}\ u\ \mathtt{be}\ p\ \mathtt{in}\ h) \end{array}$$

where in the second rule h can either be a term *or a substitution*. Once again variable capture is prevented by Barendregt's variable convention. Normalisation and confluence can be shown as for Theorem 3.13.

THEOREM 3.16
The relation $\Rightarrow_{cc} \cup \Rightarrow_{\beta\sigma}$ on xDILL is confluent and normalising.

Hence, there is an xDILL rewrite $f * \lambda x.t \Rightarrow_{\sigma} \lambda y.\langle f, y/x\rangle * t$ when f is a term substitution and also a rewrite $f * \lambda x.t \Rightarrow_{cc} \lambda y.\langle f, y/x\rangle * t$ when f is a let-substitution. Therefore these separate rules can be replaced by $f * \lambda x.t \Rightarrow_{cc} \lambda y.\langle f, y/x\rangle * t$ for arbitrary f — this is a neat implementational trick which saves work in checking whether f is a term substitution or not. Unfortunately, explicit substitutions do not seem to give a full analysis of commuting conversions. In general `let`-expressions satisfy the following generalised commuting conversion [12]

$$C[\mathtt{let}\ t\ \mathtt{be}\ p\ \mathtt{in}\ h] \;\Rightarrow\; \mathtt{let}\ t\ \mathtt{be}\ p\ \mathtt{in}\ C[h]$$

where C is any context not binding the free variables of t. Attempts to analyse commuting conversions via explicit substitutions leads to rewrite rules in the opposite direction and the consequent loss of key properties such as confluence. The behaviour of these commuting conversions is not special to either DILL or xDILL but is an instance of a more general phenomenon, *e.g.*, see recent work on the theory of coproducts [13].

4 Is xDILL correct?

In this section we address the question "Is xDILL the correct way of combining linearity and explicit substitutions?". We have already motivated in the introduction the use of DILL as the underlying linear λ-calculus and $\lambda\sigma$ as the canonical calculus of explicit substitutions. Hence our question may be rephrased "Is xDILL the right way to combine DILL and $\lambda\sigma$?"

Although such a question can never be given an absolutely positive answer, there are several reasons to believe that xDILL is as good a system as we can get.

- **Paradigmatic Criteria:** In section 3 we gave general, paradigmatic criteria concerning what xDILL should look like and what properties xDILL should satisfy. These criteria have been met. For instance, xDILL does indeed consist of a type theory, a reduction calculus and a correctness proof given by an embedding of xDILL back into DILL. In addition, each of the λ-calculus, the $\lambda\sigma$-calculus and DILL sit inside xDILL with the embedding on the $\lambda\sigma$-calculus preserving reductions.

  Even in those areas where new ideas were required, our solutions still conform to general patterns established in other calculi. In particular, the treatment of *let-substitutions* is similar to that found in other λ-calculi containing `let`-expressions, *e.g.*,reduction in xDILL was only affected by the introduction of a few *commuting conversions* which are a well known concept in the literature.
- **Semantic Criteria:** Although beyond the scope of this paper, there are semantic reasons for believing xDILL to be mathematically convincing. In [14], we described a categorical semantics for both cartesian and linear calculi of explicit substitutions based on pre-sheaves. There is a Curry-Howard correspondence between xDILL and its categorical semantics, *i.e.*, these categorical structures form a sound and complete class of models for xDILL.
- **Practical Criteria:** Above all, our motivation for xDILL was the need for sound theoretical foundations for the design and construction of linear abstract machines. The normalisation and confluence results proved in this paper are the basic results required for a sound theoretical foundation for our abstract machines. Having said this, a couple of questions remain
  - Firstly, a reduction machine requires a reduction strategy whereas our presentation of xDILL is equipped with only a rewrite relation
  - Secondly, the restrictions that certain rewrite rules apply only to term substitutions and the need to decompose a substitution in rules such as $f * uv \Rightarrow_\sigma (f_u{*}u)(f_v{*}v)$ means that, a priori, our detecting xDILL-redexes is not a constant time or space operation.

However, these criticisms of xDILL are not valid as xDILL is *not* designed to be an abstract machine. Rather xDILL is a calculus which is detailed enough to be used for typical applications of explicit substitution calculi like designing theorem provers or abstract machines. For the design of abstract machines we need to choose a reduction strategy. xDILL, in accordance with other explicit substitution calculi, makes it possible to specify such a reduction strategy in detail and reason about its efficiency. This addresses the first problem. By introducing an additional tag which indicates whether a substitution is a term substitution or not, the side conditions which state that a substitution has to be a term substitution disappears.

The problem of decomposing substitutions can be overcome in two ways. One way is to indicate as part of the syntax which term has which free variables and then distribute linear substitutions appropriately. This information is available statically, as linear variables cannot be discarded or copied during reduction. This approach is suitable for the design of theorem provers. A variant of this approach is to change the representation of the terms slightly and add a pointer to the occurrence of a bound linear variable as part of the λ-abstraction. In this way a linear β-redex can be executed in constant time. As a consequence it is possible to have only intuitionistic substitutions for which $f_u = f$.

On a technical level, we should mention that the presentation of xDILL found in [11] differs slightly from that found here. Recall, that one half of the isomorphism between a linear variable of type $!A$ and an intuitionistic variable of type A is given by the let-substitution

$$-|z :!A \vdash \texttt{let } z \texttt{ be } !x \texttt{ in } x : x : A|-$$

In [11], we permitted such substitutions to be syntactically placed in parallel with other substitutions. That is, we replaced the typing rule for !-let-substitutions found in this paper with the more general rule

$$\frac{\Gamma|\Delta_1 \vdash f : \Gamma'|\Delta' \qquad \Gamma|\Delta_2 \vdash t :!A}{\Gamma|\Delta \vdash \langle f, t/!x\rangle : \Gamma', x : A|\Delta'}$$

The idea behind this more general rule was that we could use the technology of explicit substitutions to eliminate all `let`-expressions from the calculus — β-reduction for the !-type then becomes the rather elegant

$$\langle f, !t/!x\rangle \Rightarrow \langle f, t/x\rangle$$

Unfortunately, we ran into problems and did not get any substantial pay-offs to justify them. In particular, extracting term-substitutions from arbitrary substitutions proved extremely technical as the paper demonstrates. As we have mentioned before, the same problem is solved for this version of xDILL by the elegant use of a couple of commuting conversions. The simplicity of this solution as opposed to the intricacies required by other formulations provides yet further, if rather technical, evidence that we have got our calculus right.

5 Conclusion and Further work

We have presented a linear λ-calculus with explicit substitutions and derived its major meta-theoretic properties. Our calculus is a synthesis of the logical structure of DILL and the theory of explicit substitutions as occurs in the $\lambda\sigma$-calculus.

There are two principal directions in which we wish to proceed with this research. Firstly we will pursue our overall goal of constructing efficient abstract reduction machines based on linear logic — essentially this amounts to considering reduction strategies for xDILL. There are several important and related problems; firstly a suitable form of recursion must be introduced into xDILL. Secondly we wish to improve upon the efficiency of Girard's translation of intuitionistic logic into intuitionistic linear logic and the theory of *linear decoration* [8] seems a good starting point. Finally, we also wish to apply the theory of optimal reduction to our machines to ensure as much sharing occurs as possible.

In a different direction, we believe that the techniques developed in this paper, are applicable to other λ-calculi. For example, there has recently been an interest in using the modal operator $\Box$ within the context of two-level languages. We believe that our work will be of benefit here and our recent paper [10] provides justification for this view.

Acknowledgements

This research was funded by the EPSRC-grant GR/L28296 under the title "The eXplicit Substitution Linear Abstract Machine".

References

[1] Martin Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jaques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] F.J Alberti. An abstract machine based on linear logic and explicit substitutions. Master's thesis, School of Computer Science, University of Birmingham, 1997.

[3] A. Barber. *Linear Type Theories, Semantics and Action Calculi.* PhD thesis, LFCS, University of Edinburgh, 1997.

[4] Z.-E.A. Benaissa, D. Briaud, P. Lescanne, and J. Rouyer-Degli. $\lambda\upsilon$, a calculus of explicit substitutions which preserves strong normalisation. *Journal of Functional Programming*, 6(5), 1996.

[5] N. Benton, G. Bierman, V. de Paiva, and M. Hyland. A term calculus for intuitionistic linear logic. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 75–90. Springer Verlag, 1993.

[6] R. Bloo and K.H. Rose. Preesrvation of strong normalisation in named lambda calculi with explicit substitution and garbage collection. In *Proc. CSN'95—Computer Science in the Netherlands*, pages 62–72, 1995.

[7] Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6(2):195–244, 1996.

[8] V. Danos, J.-B. Joinet, and H. Schellinx. On the linear decoration of intuitionistic derivations. *Archive for Mathematical Logic*, 33:387–412, 1995.

[9] N. Ghani. *Adjoint Rewriting.* PhD thesis, University of Edinburgh, 1995.

[10] N. Ghani, V. de Paiva, and E. Ritter. Explicit substitutions for constructive necessity. In *Proc. of ICALP'98*, volume 1443 of *Lecture Notes in Computer Science*, pages 743–754, 1998.

[11] N. Ghani, V. de Paiva, and E. Ritter. Linear explicit substitutions. In *Proc. of Westapp'98*, 1998.

[12] Neil Ghani. Adjoint rewriting and the !-type constructor. Manuscript, University of Leicester, 1998.

[13] Neil Ghani. $\beta\eta$-equality for coproducts. In *Second Conference on Typed Lambda Calculus and its Applications*, Edinburgh, April 1995.

[14] N. Ghani, V. de Paiva, and E. Ritter. Categorical models for explicit substitutions. In W. Thomas, editor, *Proc. of FoSSaCS'99*, volume 1578 of *Lecture Notes in Computer Science*. Springer Verlag, 1999.

[15] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[16] Yves Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

[17] Ian Mackie. Lilac: A functional programming language based on linear logic. *Journal of Functional Programming*, 4(4):395–433, 1994.

[18] Ian Mackie. The Geometry of Interaction Machine. In *Proc. of the 22nd Symposium on Principles of Programming Languages*, pages 198–208, 1995.

[19] P.-A. Mellies. Typed λ-calculi with explicit substitution may not terminate. In *Proc. of TLCA'95*, pages 328–334. LNCS 902, 1995.

[20] E. Ritter and V. de Paiva. On explicit substitution and names (extended abstract). In *Proc. of ICALP'97*, LNCS 1256, pages 248–258, 1997.

[21] Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, 1997.

[22] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*. North Holland, April 1990.

Received 23 December 1999