J. Functional Programming 4 (4): 395–433, October 1994

Lilac: a functional programming language based on linear logic

IAN MACKIE

Department of Computing, Imperial College of Science, Technology and Medicine,

180 Queen's Gate, London SW7 2BZ, UK

(E-mail: im@doc.ic.ac.uk*)*

Abstract

We take Abramsky's term assignment for Intuitionistic Linear Logic (the linear term calculus) as the basis of a functional programming language. This is a language where the programmer must embed explicitly the resource and control information of an algorithm. We give a type reconstruction algorithm for our language in the style of Milner's $\mathcal{W}$ algorithm, together with a description of the implementation and examples of use.

Capsule Review

Since the introduction of linear logic, researchers have been investigating the possibility of a functional programming language based on the institutionistic fragment. This paper gives a clear explanation of how this is possible. It first considers a linear lambda calculus – the basis of any linear functional programming language. An implementation strategy based on the SECD machine is considered for this calculus, as well as a type inference algorithm based on Milner's $\mathcal{W}$ algorithm.

Finally, the paper considers how to extend the calculus into a usable programming language with not only a more pleasant syntax, but also constructs for recursion and datatypes.

1 Introduction

Functional languages have their theoretical foundations in the lambda calculus; indeed, it is the canonical form of such languages. In particular, the Curry-Howard isomorphism establishes a tight relationship between intuitionistic propositional logic and the simply typed λ-calculus. However, the λ-calculus is too abstract and its reduction steps too 'big' to be of use directly as the basis of an implementation.

1. The λ-calculus does not provide any hint about the implementation of β-reduction. Indeed, the primary technique used to implement β-reduction, namely *explicit* substitutions (e.g. see Abadi *et al.*, 1991), takes us out of the realm of the pure λ-calculus.

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

2. The λ-calculus does not give us information regarding argument use; in particular copying of arguments. Thus, one needs sophisticated techniques to extract this information: for example, abstract interpretation (Abramsky & Hankin, 1987):
   - Strictness analysis is concerned with knowing when lazy functions can be safely replaced by eager versions.
   - In-place update analysis is concerned with knowing when it is safe to overwrite a specific data object because it is no longer needed.

This motivates the use of proof terms of Intuitionistic Linear Logic (Girard, 1987) as the foundation of a functional programming language. The linear types have more information about use/reuse than the intuitionistic types:

- discarding an argument corresponds to the *weakening* rule of the logic. Knowing that an argument is discarded tells us that it is *not* strict in that argument;
- copying an argument corresponds to the *contraction* rule of the logic. Knowing that an argument is not copied tells us that we can perform in-place updates safely.

Abramsky (1993) describes a syntax for the proof terms of Intuitionistic Linear Logic: the linear term calculus. That paper also describes an SECD implementation for the linear term calculus, which uses the explicit operational information inherent in the linear term calculus.

In this paper, we concentrate on the practical issues for the linear term calculus, where we investigate the usefulness of the extra information given by the linear types. The main contributions of this paper are:

1. A type reconstruction algorithm for the linear term calculus, mirrored after Milner's $\mathcal{W}$ algorithm, which will infer the most general *linear* type for a given term. This type gives the kind of information suggested above. A similar algorithm has been developed independently by Wadler (1991), and more recently by Lincoln & Mitchell (1992).
2. A concrete realisation of Abramsky's linear term calculus as a programming language, where we include data-types and recursion. We assume that the reader is familiar with the work of Abramsky (1993), which is our primary reference.

We will not address many of the issues regarding garbage collection and in-place updates. In fact, we will only give a straightforward implementation based on an SECD machine. The purpose of this work is to set up a framework and investigate the pragmatics of the language.

Several linear functional languages have already been proposed, for example Holmström (1988), Lafont (1988b), Wadler (1991), Wakeling (1990) and Chirimar, Gunter & Rieke (1991). The work closest to ours is that of Holmström and Chirimar *et al.*, since we deal with the linear constraints fully; we do not have any *non-linear* data types. For example, Lafont (1988b), Wadler (1991) and Wakeling (1990) do not have terms corresponding to the *dereliction* rule of the logic. Our aim is to

make *everything* explicit in the hope that we can gain some insight into the low-level resource usage of a program.

The rest of this paper is organised as follows. Section 2 presents a brief overview of linear logic and gives the term assignment for the Intuitionistic fragment: the linear term calculus, which we are taking as our canonical linear functional programming language. Section 3 gives a natural deduction presentation of intuitionistic linear logic and defines a type reconstruction algorithm for the calculus. In section 4 we introduce our language, Lilac. We give the syntax, implementation and examples of use. Section 5 suggests some extensions to our ideas.

2 Linear logic and the linear term calculus

The linear term calculus is our 'λ-calculus'. Work on standard functional programming can use well established results about the λ-calculus so we must therefore try to set up an equivalent foundation for our calculus. Most of this section is included for completeness only, and is essentially taken from Abramsky (1993).

2.1 Linear logic

It is not our aim to give a treatise on linear logic. Our interests are the computational interpretations of the intuitionistic fragment of linear logic; and besides, there are many interesting introductions available in the literature, for example papers by Girard (1989), Lincoln (1992), Scedrov (1993) and Lafont (1988a). However, we would like our presentation to be in some sense complete, so we shall present a condensed overview of the subject.

Classical linear logic (Girard, 1987) is hailed as a major breakthrough in both computer science and logic. In logic because we have a constructive classical logic–with ordinary classical logic there is no way of extracting an algorithm from a proof. But, most importantly for us, in computer science because it gives an approach for investigation into issues concerning evaluation semantics, memory organisation and the possibility of safe side-effects in functional programming languages (Wadler 1990).

In this paper we shall deal with intuitionistic linear logic, which is a refinement of intuitionistic logic where the structural rules *weakening* and *contraction* are removed and re-introduced as logical rules by the use of a new logical connective–the *exponential.*

The connectives of intuitionistic linear logic and their significance are stated below:

- $\mathbf{I}$: The tensor unit is the identity for the tensor product.
- $A \multimap B$: The linear function space–functions which use their arguments *exactly* once. For example, a bricklayer will build you a wall if supplied with some bricks, but the act of doing so consumes the bricks, i.e. you cannot have two walls. In programming terms this restricts all functions to using their arguments exactly once. A direct consequence of this is that all functions are

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

now *strict*, indicating that we can safely use eager evaluation since we know the argument will be used–the preferred strategy for implementing functional languages efficiently.

- $A \,\&\, B$: The direct product (or *with*) is the data constructor which places a restriction on using either the left or the right projection; but not both. For example, a bricklayer will offer to build a wall or a house extension if supplied with some bricks. The choice is yours, but the construction of one utilises the bricks necessary to build the other. A programming example of this is the conditional, where either the consequent or the alternative will be selected; but not both. Note that lazy evaluation is indicated since it is not known which projection will be selected.
- $A \otimes B$: The tensor product is the data constructor where *both* components of the pair must be used; projections are forbidden. If we give our bricklayer some bricks and mortar, then to produce a wall he will need both–one is no good without the other. Note that eager evaluation should be used since both components are used.
- $A \oplus B$: The direct sum (or *plus*) is the linear version of the logical or. This corresponds to an internal choice, where we will be offered one or the other, but not both. Again, using our friend the bricklayer, if we give him some resources (bricks and mortar) and tell him to get to work, we will either get a straight wall or a crooked one. The outcome depends on how much beer he had for lunch which is out of our control. Of course, we cannot have both since the process of building the wall consumes the resources. A programming example would be some form of non-deterministic 'function' $A \multimap B \oplus C$. Note that eager evaluation is indicated here.
- $!A$: The modality (exponential) 'of course A' is what puts the power back into linear logic. It is this which reflects the non-linear use. This corresponds to giving our bricklayer a credit card and telling him to buy the resources he needs. So, he can now build many walls and also go on a holiday! A programming example would be the functions in a standard environment–we would want to use these functions many times. Note that lazy evaluation should be used here since it is not known how the data will be used.

In particular for this logic we have the following four logical rules to handle the exponential, which we will present here in the *sequent calculus*:

- Of course Right rule:

$$\frac{!\Gamma \vdash A}{!\Gamma \vdash !A}\ \text{(promotion)}$$

  where $!\Gamma$ is a context containing only formulae of the form $!A$.
- Of course Left rules:

$$\frac{\Gamma \vdash B}{\Gamma, !A \vdash B}\ \text{(weakening)}$$

$$\frac{\Gamma, A \vdash B}{\Gamma, !A \vdash B}\ \text{(dereliction)}$$

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

$$\frac{\Gamma, !A, !A \vdash B}{\Gamma, !A \vdash B}\ (\text{contraction})$$

Note that Weakening, Dereliction and Contraction correspond to the use of a resource zero, once and twice respectively; and it is the Contraction rule that gives the *potential* of an infinite number of copies.

2.2 *The linear term calculus*

The linear term calculus is the term assignment to the intuitionistic fragment of linear logic. We begin with the definition of the linear term calculus which we will take as standard. An important point to note is that the syntax is not context free–the linearity constraint must be reflected in the construction of the terms. We adopt the same approach to that of Abramsky (1993) and define an auxiliary syntactic category of patterns $\mathscr{P}_X$, the set of patterns with variables in X; and then define the terms, $\mathscr{T}_X$, having free variables in X.

Definition 2.1 (The Linear Term Calculus)

$$\star, \_ \in \mathscr{P}_\emptyset \qquad \langle x, \_\rangle, \langle \_, x\rangle,\ !x \in \mathscr{P}_{\{x\}} \qquad x \otimes y, x@y \in \mathscr{P}_{\{x,y\}}$$

We can now define $\mathscr{T}_X$, the linear terms with free variables in X, inductively as follows:

- $x \in \mathscr{T}_{\{x\}}$
- $\star \in \mathscr{T}_\emptyset$
- $t \in \mathscr{T}_X, u \in \mathscr{T}_Y, X \cap Y = \emptyset \Longrightarrow t \otimes u, tu \in \mathscr{T}_{X \cup Y}$
- $t, u \in \mathscr{T}_X \Longrightarrow \langle t, u\rangle \in \mathscr{T}_X$
- $t \in \mathscr{T}_X \Longrightarrow \mathtt{inl}(t), \mathtt{inr}(t),\ !t \in \mathscr{T}_X$
- $t \in \mathscr{T}_{X \cup \{x\}}, x \notin X \Longrightarrow \lambda x.t \in \mathscr{T}_X$
- $t \in \mathscr{T}_X, p \in \mathscr{P}_Y, u \in \mathscr{T}_{Y \cup Z}, X \cap Z = Y \cap Z = \emptyset \Longrightarrow \mathtt{let}\ t\ \mathtt{be}\ p\ \mathtt{in}\ u \in \mathscr{T}_{X \cup Z}$
- $t \in \mathscr{T}_X, u \in \mathscr{T}_{Z \cup \{x\}}, v \in \mathscr{T}_{Z \cup \{y\}}, X \cap Z = \{x, y\} \cap Z = \emptyset$
  $\Longrightarrow \mathtt{case}\ t\ \mathtt{of}\ \mathtt{inl}(x) \Rightarrow u \mid \mathtt{inr}(y) \Rightarrow v \in \mathscr{T}_{X \cup Z}$

Remark

1. Throughout this paper we will adopt the form `let` t `be` p `in` u consistently. One should read this as a pattern matching construct, where t is a term that will match against a pattern p. Some other authors (e.g. Benton *et al.* (1992) and Lincoln & Mitchell (1992)) have chosen to give a more intuitive but less systematic syntax.
2. The reader who has tried writing terms in this calculus may have noticed that there are several ways of writing what, at first sight, are the same thing. For example, consider the linear version of the **K**. combinator (**K**. $= \lambda xy.y$); we can write this in two ways:

$$\lambda xy.\mathtt{let}\ x\ \mathtt{be}\ \_\ \mathtt{in}\ y \text{ and } \lambda x.\mathtt{let}\ x\ \mathtt{be}\ \_\ \mathtt{in}\ \lambda y.y$$

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

Operationally, using a leftmost outermost evaluation strategy, the first version takes two arguments, one at a time, then discards the first one. The second version states that after the first argument has been provided, it will discard it, then wait for the second argument. These terms have the same type, but surely the second version is better from a programming point of view–the sooner we can reclaim the data space the better. Technically, these terms are related via what is known as a commutative conversion; the same proof (program) can be given in different ways. The real proofs are those modulo these conversions. A discussion on these issues can be found in Benton *et al.* (1992), and for a more complete set of equations for the multiplicative fragment see Mackie *et al.* (1993).

2.3 Operational semantics of the linear term calculus

It is important to have a model of the calculus which can be used as a basis for proving properties about any implementation. We choose to give an operational semantics rather than any other formalism for the following reason: an operational semantics provides us with intuitions, removed from any mathematical structure or specific implementation. It is generally regarded that the best formalism in which to present a semantics depends upon who will be using it. Operational semantics is biased towards the user of the language, hence is sufficient in the spirit of this work.

We present the operational semantics in the style advocated by Plotkin (1981), and more recently for Standard ML by Milner, Tofte & Harper (1990). The operational semantics take on the form of an inductive definition. We write $t \Downarrow u$ for the evaluation relation ('t converges to u') which formalises our notion of a computation step.

We will use meta-variables c and d to range over *canonical* forms, which are:

$$\star \quad < t, u > \quad !t \quad c \otimes d \quad \lambda x.t \quad \mathtt{inl}(c) \quad \mathtt{inr}(d)$$

The operational semantics are given in Fig. 1; taken from Abramsky (1993).

3 Linear type reconstruction

This section develops a type reconstruction algorithm, $\mathscr{L}$, for the linear term calculus which we are taking to be our canonical functional programming language. Our algorithm is in a similar style to that of Milner's algorithm $\mathscr{W}$. We begin by presenting the type assignment to the linear term calculus in natural deduction form. These rules suggest the type reconstruction algorithm which we will go on to prove to be both sound and complete with respect to these rules. (The full proofs are given in Mackie, 1991). The reader is referred to the work of Milner (1978), Damas & Milner (1982) and Damas (1985) for background to this work.

Work on linear type inference is not new:

- The work of Lincoln & Mitchell (1992) is very close to ours. The presentation of the term assignment corresponds to their Nat2 system. In particular, in

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

$$\frac{}{\star \Downarrow \star} \qquad \frac{t \Downarrow \star \quad u \Downarrow c}{\texttt{let } t \texttt{ be } \star \texttt{ in } u \Downarrow c}$$

$$\frac{t \Downarrow c \quad u \Downarrow d}{t \otimes u \Downarrow c \otimes d} \qquad \frac{t \Downarrow c \otimes d \quad u[c/x, d/y] \Downarrow e}{\texttt{let } t \texttt{ be } x \otimes y \texttt{ in } u \Downarrow e}$$

$$\frac{}{\lambda x.t \Downarrow \lambda x.t} \qquad \frac{t \Downarrow \lambda x.v \quad u \Downarrow c \quad v[c/x] \Downarrow d}{tu \Downarrow d}$$

$$\frac{}{\langle t, u \rangle \Downarrow \langle t, u \rangle}$$

$$\frac{t \Downarrow \langle v, w \rangle \quad v \Downarrow c \quad u[c/x] \Downarrow d}{\texttt{let } t \texttt{ be } \langle x, \_ \rangle \texttt{ in } u \Downarrow d} \qquad \frac{t \Downarrow \langle v, w \rangle \quad w \Downarrow c \quad u[c/y] \Downarrow d}{\texttt{let } t \texttt{ be } \langle \_, y \rangle \texttt{ in } u \Downarrow d}$$

$$\frac{t \Downarrow c}{\texttt{inl}(t) \Downarrow \texttt{inl}(c)} \qquad \frac{u \Downarrow d}{\texttt{inr}(u) \Downarrow \texttt{inr}(d)}$$

$$\frac{t \Downarrow \texttt{inl}(c) \quad u[c/x] \Downarrow d}{\texttt{case } t \texttt{ of inl}(x) \Rightarrow u \mid \texttt{inr}(y) \Rightarrow v \Downarrow d} \qquad \frac{t \Downarrow \texttt{inr}(c) \quad v[c/y] \Downarrow d}{\texttt{case } t \texttt{ of inl}(x) \Rightarrow u \mid \texttt{inr}(y) \Rightarrow v \Downarrow d}$$

$$\frac{}{!t \Downarrow !t} \qquad \frac{t \Downarrow !v \quad v \Downarrow c \quad u[c/x] \Downarrow d}{\texttt{let } t \texttt{ be } !x \texttt{ in } u \Downarrow d}$$

$$\frac{t \Downarrow !t' \quad u \Downarrow c}{\texttt{let } t \texttt{ be } \_ \texttt{ in } u \Downarrow c} \qquad \frac{t \Downarrow !t' \quad u[!t'/x, !t'/y] \Downarrow c}{\texttt{let } t \texttt{ be } x@y \texttt{ in } u \Downarrow c}$$

Fig. 1. Operational semantics of the linear term calculus.

that paper, they independently prove the existence of the type reconstruction algorithm.

- The work of Wadler (1991) and Wakeling (1990) tries to enforce a linear type discipline onto the λ-calculus; they have the typed λ-calculus as the term assignment to linear logic with the consequence of having the ability to have many types for the same term. Our approach has a term corresponding to each of the structural rules which gives us a unique type for each term. Another advantage of our approach is that we have 'extra knowledge' about our programs. Wadler points out that in a *call-by-need* régime dereliction leads to us to not knowing how many pointers we have to a data item; see the following example from (Wadler 1991):

$$\lambda f x.f(fx) : !(\alpha \multimap \alpha) \multimap \alpha \multimap \alpha$$

The claim is that, since dereliction was needed to type each occurrence of f,

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

it has a linear type $\alpha \multimap \alpha$, but since f really has a non-linear type, there may be more than one pointer to it. The problem is that there is no history in the proof to indicate that contraction and dereliction is used. Our version of this function is:

$$\lambda f x.\texttt{let } f \texttt{ be } g@h \texttt{ in let } g \texttt{ be } !g' \texttt{ in let } h \texttt{ be } !h' \texttt{ in } g'(h'x) : !(\alpha \multimap \alpha) \multimap \alpha \multimap \alpha$$

From this we know exactly how many copies of f are made–since we have recorded the *full* history of the proof in the term we can extract this kind of information–which is not the case in other work mentioned where there are no terms for some of the logical rules. Of course, this information gives nothing more than a reference count–a simple variable count on the original term will provide exactly the same information. Although we will not take advantage of this information in our implementation (see, for example, Chirimar *et al.*, 1991, for a reference counting model) we propose that this information provides a general basis in which more refined and efficient implementations can be developed.

3.1 A new notation

Linear logic is a resource logic: we place a restriction on the use of assumptions–namely use them all exactly once in the linear case. It is imperative that our logical rules reflect this constraint. To capture this we will write judgements in the following way:

$$\Gamma | \Theta \vdash t : \alpha$$

where we call Γ the *before-set* and Θ the *after-set*—indicating that the derivation uses the assumptions only in the set $\Gamma \setminus \Theta$. Note that $\Theta \subseteq \Gamma$, and $\Gamma \setminus \Theta$ is precisely the set of assumptions necessary to type t.

The idea is best explained by an example. Consider the Application rule:

$$\frac{\Gamma | \Delta \vdash t : \alpha \multimap \beta \quad \Delta | \Theta \vdash u : \alpha}{\Gamma | \Theta \vdash tu : \beta} \ (\multimap\text{-elim})$$

This rule states that if we type tu using Γ then Θ will be left over. We give t all of the assumptions, and the remaining Δ are given to u. What are not consumed here are exactly those which are left over in typing tu. The rationale for choosing this new notation will become more apparent when we present the type reconstruction algorithm.

For a substitution R, we will write $R(A|A_1)$ for $RA|RA_1$, and define substitution on judgements by :

$$R((A \mid A_1) \vdash t : \alpha) = (RA \mid RA_1) \vdash t : R\alpha$$

We will write $x : A.\Gamma$ for concatenation of assumptions.

The Natural Deduction presentation (in sequent form) for Intuitionistic Linear

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

Logic using our new notation is given in Figure 2. The natural deduction presentation follows from the Sequent based rules of Abramsky (1993) by standard translations (see, for example, Girard, Lafont & Taylor, 1989).

The only problematic case is the promotion rule which has been studied by Benton *et al.* (1992), Lincoln & Mitchell (1992), O'Hearn (1991) and Wadler (1992). As noted by Benton *et al.* and Wadler, the presentation of natural deduction given here is not closed under substitution. However, since we are working with programs (i.e. *closed* terms) and we are dealing only with outermost reductions (we never never substitute a free variable) these problems never surface. This is a property coming from the operational semantics used and for our application everything works. However, in a more general setting such as a categorical semantics of the language a more delicate treatment of the rules is required, for example that of Benton *et al.* (1992).

3.2 The type reconstruction algorithm $\mathscr{L}$

Our presentation of the algorithm $\mathscr{L}$ will assume that the terms are *syntactically* linear. It is a trivial extension to the algorithm to perform this kind of checking–we just need extra conditions to be satisfied. The reason we refrain from giving these conditions is purely one of presentation–adding these extra checks obscures the real details of the algorithm. The important point is that since our algorithm makes resource consumption explicit it is a very useful place to perform such checks.

We begin with a short note on unification before giving the algorithm.

Unification

We will need the notion of unification in this section, for which we define an algorithm $\mathscr{V}$, a simple extension to the unification algorithm used for Milner's $\mathscr{W}$; based on a variant of Robinson's theorem (Robinson, 1965). Given two linear types τ and τ':

$$\mathscr{V}\ \tau\ \tau' = \begin{cases} U & \text{if } U\tau = U\tau' \\ \text{fail} & \text{otherwise} \end{cases}$$

where U is the *most general unifier*: if V also unifies τ and τ' then V is just a substitution instance of U, i.e. $V = SU$, for some substitution S. The final requirement is that U only involves variables τ and τ'–no new variables are introduced during unification.

The implementation is based on the notion of *disagreement pairs*, which is a straightforward extension of the algorithm for intuitionistic types. We define this algorithm as follows. We write τ, τ' for type variables and A, B, C, D for compound formulae. We assume the following clauses are applied exhaustively from top-to-bottom.

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

$$\frac{}{(x : \alpha.\Gamma \mid \Gamma) \vdash x : \alpha}\ (\text{Axiom})$$

$$\frac{}{(\Gamma \mid \Gamma) \vdash \star : \mathbf{I}}\ (\mathbf{I}\text{-intro}) \qquad \frac{(\Gamma \mid \Delta) \vdash t : \mathbf{I} \quad (\Delta \mid \Theta) \vdash u : \alpha}{(\Gamma \mid \Theta) \vdash \mathtt{let}\ t\ \mathtt{be}\ \star\ \mathtt{in}\ u : \alpha}\ (\mathbf{I}\text{-elim})$$

$$\frac{(\Gamma \mid \Delta) \vdash t : \alpha \quad (\Delta \mid \Theta) \vdash u : \beta}{(\Gamma \mid \Theta) \vdash t \otimes u : \alpha \otimes \beta}\ (\otimes\text{-intro})$$

$$\frac{(\Gamma \mid \Delta) \vdash t : \alpha \otimes \beta \quad (x : \alpha.y : \beta.\Delta \mid \Theta) \vdash u : \gamma}{(\Gamma \mid \Theta) \vdash \mathtt{let}\ t\ \mathtt{be}\ x \otimes y\ \mathtt{in}\ u : \gamma}\ (\otimes\text{-elim})$$

$$\frac{(x : \alpha.\Gamma \mid \Theta) \vdash t : \beta}{(\Gamma \mid \Theta) \vdash \lambda x.t : \alpha \multimap \beta}\ (\multimap\text{-intro}) \qquad \frac{(\Gamma \mid \Delta) \vdash t : \alpha \multimap \beta \quad (\Delta \mid \Theta) \vdash u : \alpha}{(\Gamma \mid \Theta) \vdash tu : \beta}\ (\multimap\text{-elim})$$

$$\frac{(\Gamma \mid \Theta) \vdash t : \alpha \quad (\Gamma \mid \Theta) \vdash u : \beta}{(\Gamma \mid \Theta) \vdash \langle t, u \rangle : \alpha \,\&\, \beta}\ (\,\&\text{-intro})$$

$$\frac{(\Gamma \mid \Delta) \vdash t : \alpha \,\&\, \beta \quad (x : \alpha.\Delta \mid \Theta) \vdash u : \gamma}{(\Gamma \mid \Theta) \vdash \mathtt{let}\ t\ \mathtt{be}\ < x, y >\ \mathtt{in}\ u : \gamma}\ (\,\&\text{-elim : left})$$

$$\frac{(\Gamma \mid \Delta) \vdash t : \alpha \,\&\, \beta \quad (y : \beta.\Delta \mid \Theta) \vdash u : \gamma}{(\Gamma \mid \Theta) \vdash \mathtt{let}\ t\ \mathtt{be}\ < x, y >\ \mathtt{in}\ u : \gamma}\ (\,\&\text{-elim : right})$$

$$\frac{(\Gamma \mid \Theta) \vdash t : \alpha}{(\Gamma \mid \Theta) \vdash \mathtt{inl}(t) : \alpha \oplus \beta}\ (\oplus\text{-intro : left}) \qquad \frac{(\Gamma \mid \Theta) \vdash t : \beta}{(\Gamma \mid \Theta) \vdash \mathtt{inr}(t) : \alpha \oplus \beta}\ (\oplus\text{-intro : right})$$

$$\frac{(\Gamma \mid \Delta) \vdash t : \alpha \oplus \beta \quad (x : \alpha.\Delta \mid \Theta) \vdash u : \gamma \quad (y : \beta.\Delta \mid \Theta) \vdash v : \gamma}{(\Gamma \mid \Theta) \vdash \mathtt{case}\ t\ \mathtt{of}\ \mathtt{inl}(x)\ \Rightarrow\ u \mid \mathtt{inr}(y)\ \Rightarrow\ v : \gamma}\ (\oplus\text{-elim})$$

$$\frac{(\Gamma \mid \Delta) \vdash t : !\alpha \quad (x : \alpha.\Delta \mid \Theta) \vdash u : \beta}{(\Gamma \mid \Theta) \vdash \mathtt{let}\ t\ \mathtt{be}\ !x\ \mathtt{in}\ u : \beta}\ (!\text{-elim : dereliction})$$

$$\frac{(\Gamma \mid \Delta) \vdash t : !\alpha \quad (x : !\alpha.y : !\alpha.\Delta \mid \Theta) \vdash u : \beta}{(\Gamma \mid \Theta) \vdash \mathtt{let}\ t\ \mathtt{be}\ x@y\ \mathtt{in}\ u : \beta}\ (!\text{-elim : contraction})$$

$$\frac{(\Gamma \mid \Delta) \vdash t : !\alpha \quad (\Delta \mid \Theta) \vdash u : \beta}{(\Gamma \mid \Theta) \vdash \mathtt{let}\ t\ \mathtt{be}\ \_\ \mathtt{in}\ u : \beta}\ (!\text{-elim : weakening})$$

$$\frac{(\Gamma \mid \Delta) \vdash t : \alpha}{(\Gamma \mid \Delta) \vdash !t : !\alpha}\ (!\text{-intro : promotion})\ (\Gamma \setminus \Delta \text{ are all } !\text{-type})$$

Fig. 2. Natural deduction formulation of intuitionistic linear logic.

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

$$
\begin{array}{lcl}
\mathscr{D}(\tau, \tau') & = & \text{if } \tau = \tau' \text{ then } \emptyset \text{ else } (\tau, \tau') \\
\mathscr{D}(\mathbf{I}, \mathbf{I}) & = & \emptyset \\
\mathscr{D}(A \otimes B, C \otimes D) & = & \text{if } \mathscr{D}(A, C) = \emptyset \text{ then } \mathscr{D}(B, D) \text{ else } \mathscr{D}(A, C) \\
\mathscr{D}(A \multimap B, C \multimap D) & = & \text{if } \mathscr{D}(A, C) = \emptyset \text{ then } \mathscr{D}(B, D) \text{ else } \mathscr{D}(A, C) \\
\mathscr{D}(A \,\&\, B, C \,\&\, D) & = & \text{if } \mathscr{D}(A, C) = \emptyset \text{ then } \mathscr{D}(B, D) \text{ else } \mathscr{D}(A, C) \\
\mathscr{D}(A \oplus B, C \oplus D) & = & \text{if } \mathscr{D}(A, C) = \emptyset \text{ then } \mathscr{D}(B, D) \text{ else } \mathscr{D}(A, C) \\
\mathscr{D}(!A, !B) & = & \mathscr{D}(A, B) \\
\mathscr{D}(A, B) & = & (A, B)
\end{array}
$$

We can then use the standard iterative algorithm for unification (for example, see Field & Harrison, 1988).

Our substitutions are mappings from types to types. They are associative and idempotent. We will write composition of substitutions by juxtaposition.

To reflect the linearity constraint that all assumptions must be used exactly once, we treat the assumption set as a set of resources–once used, we remove it. To this end our type reconstruction algorithm will return a triple (rather than a pair in the case of Milner's $\mathscr{W}$ algorithm), which will consist of a substitution, a type and the assumptions not yet used.

We will write $R, S, \ldots$ to range over substitutions, $\alpha, \beta, \ldots$ to range over type variables, $A, A', A_1, A_2, \ldots$ to range over assumption sets. We will write id for the identity substitution and substitution over sets is defined element-wise: $RA = \{x : R\alpha | x : \alpha \in A\}$. We assume a function *new* which returns a fresh type variable.

Definition 3.1 (The type reconstruction algorithm $\mathscr{L}$)

$$\mathscr{L}(A, e) = (T, \tau, A')$$

where:

1. If e is the identifier x, and $x : \alpha \in A$ then $T = \mathsf{id}$, $\tau = \alpha$, $A' = A \setminus \{x : \alpha\}$.
2. If e is of the form $\star$, then $T = \mathsf{id}, \tau = \mathbf{I}, A' = A$.
3. If e is of the form `let` t `be` $\star$ `in` u, let

$$
\begin{array}{lcl}
(R, \beta, A_1) & = & \mathscr{L}(A, t) \\
U & = & \mathscr{V}\ \beta\ \mathbf{I} \\
(S, \alpha, A_2) & = & \mathscr{L}(URA_1, u)
\end{array}
$$

then $T = SUR, \tau = \alpha, A' = A_2$.

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

4. If e is of the form $t \otimes u$, let

$$\begin{aligned}(R, \alpha, A_1) &= \mathscr{L}(A, t)\\ (S, \beta, A_2) &= \mathscr{L}(RA_1, u)\end{aligned}$$

then $T = SR, \tau = S\alpha \otimes \beta, A' = A_2$.

5. If e is of the form `let` t `be` $x \otimes y$ `in` u, let

$$\begin{aligned}(R, \epsilon, A_1) &= \mathscr{L}(A, t)\\ U &= \mathscr{V}\ \epsilon\ \alpha \otimes \beta; \alpha, \beta\ \textit{new}\\ (S, \gamma, A_2) &= \mathscr{L}(URA_1.x : \alpha.y : \beta, u)\end{aligned}$$

then $T = SUR, \tau = \gamma, A' = A_2$.

6. If e is of the form $\lambda x.t$, let

$$(R, \beta, A_1) = \mathscr{L}(A.x : \alpha, t); \alpha\ \textit{new}$$

then $T = R, \tau = R\alpha \multimap \beta, A' = A_1$.

7. If e is of the form tu, let

$$\begin{aligned}(R, \sigma, A_1) &= \mathscr{L}(A, t)\\ (S, \alpha, A_2) &= \mathscr{L}(RA_1, u)\\ U &= \mathscr{V}\ (S\sigma)\ (\alpha \multimap \beta); \beta\ \textit{new}\end{aligned}$$

then $T = USR, \tau = U\beta, A' = A_2$.

8. If e is of the form `let` t `be` $< x, \_ >$ `in` u, let

$$\begin{aligned}(R, \epsilon, A_1) &= \mathscr{L}(A, t)\\ U &= \mathscr{V}\ \epsilon\ \alpha\ \&\ \beta; \alpha, \beta\ \textit{new}\\ (S, \gamma, A_2) &= \mathscr{L}(URA_1.x : \alpha, u)\end{aligned}$$

then $T = SUR, \tau = \gamma, A' = A_2$.

9. If e is of the form `let` t `be` $< \_, y >$ `in` u, similar to above.

10. If e is of the form $< t, u >$, let

$$\begin{aligned}(R, \alpha, A_1) &= \mathscr{L}(A, t)\\ (S, \beta, A_2) &= \mathscr{L}(RA, u)\\ \textit{Condition} &: A_2 = A_1\end{aligned}$$

then $T = SR, \tau = S\alpha\ \&\ \beta, A' = A_1 (= A_2)$.

11. If e is of the form `inl`(t), let

$$(R, \alpha, A_1) = \mathscr{L}(A, t)$$

then $T = R, \tau = \alpha \oplus \beta, A' = A_1;\ \beta$ *new*.

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

12. If e is of the form $\texttt{inr}(t)$, similar to above.
13. If e is of the form $\texttt{case}\ t\ \texttt{of}\ \texttt{inl}(x) \Rightarrow u \mid \texttt{inr}(y) \Rightarrow v$, let

$$\begin{array}{lcl} (R, \epsilon, A_1) & = & \mathscr{L}(A, t) \\ U & = & \mathscr{V}\ \epsilon\ \alpha \oplus \beta; \alpha, \beta\ new \\ (S, \rho, A_2) & = & \mathscr{L}(URA_1.x : \alpha, u) \\ (S', \sigma, A_3) & = & \mathscr{L}(SURA_1.y : \beta, v) \\ Condition & : & A_3 = A_2 \\ V & = & \mathscr{V}\ \sigma\ S'\rho \end{array}$$

then $T = VS'SUR, \tau = V\sigma, A' = A_2 (= A_3)$.

14. If e is of the form $\texttt{let}\ t\ \texttt{be}\ !x\ \texttt{in}\ u$, let

$$\begin{array}{lcl} (R, \beta, A_1) & = & \mathscr{L}(A, t) \\ U & = & \mathscr{V}\ \beta\ !\alpha; \alpha\ new \\ (S, \rho, A_2) & = & \mathscr{L}(URA_1.x : \alpha, u) \end{array}$$

then $T = SUR, \tau = \rho, A' = A_2$.

15. If e is of the form $\texttt{let}\ t\ \texttt{be}\ x@y\ \texttt{in}\ u$, let

$$\begin{array}{lcl} (R, \rho, A_1) & = & \mathscr{L}(A, t) \\ U & = & \mathscr{V}\ \rho\ !\alpha; \alpha\ new \\ (S, \beta, A_2) & = & \mathscr{L}(URA_1.x : !\alpha.y : !\alpha, u) \end{array}$$

then $T = SUR, \tau = \beta, A' = A_2$.

16. If e is of the form $\texttt{let}\ t\ \texttt{be}\ \_\ \texttt{in}\ u$, let

$$\begin{array}{lcl} (R, \epsilon, A_1) & = & \mathscr{L}(A, t) \\ U & = & \mathscr{V}\ \epsilon\ !\alpha; \alpha\ new \\ (S, \beta, A_2) & = & \mathscr{L}(URA_1, u) \end{array}$$

then $T = SUR, \tau = \beta, A' = A_2$.

17. If e is of the form $!t$, let

$$\begin{array}{lcl} (R, \alpha, A_1) & = & \mathscr{L}(A, t) \\ condition & : & R(A \setminus A_1) \text{ are all } !\text{-type} \end{array}$$

then $T = R, \tau = !\alpha, A' = A_1$.

We also require that $\mathscr{L}$ will fail if it is not one of the above forms.

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

3.3 Soundness of $\mathscr{L}$

If the algorithm $\mathscr{L}$ succeeds in typing a term e under some assumptions, then we want to be sure that e actually is well typed, i.e. a derivation exists. This is called syntactic soundness of our algorithm and states that our algorithm is safe–it produces no wrong results. The following Lemma will be useful for this result:

Lemma 3.2
If there is a derivation $(A \mid A') \vdash e : \tau$ then, for any substitution S, there is also a derivation for $S(A|A') \vdash e : S\tau$.

Proof
By induction over the length of the derivation of $A|A' \vdash e : \tau$.
There are 17 cases, we will just show four.

1. If the derivation consists of the (Axiom) $: A.x : \tau|A \vdash x : \tau$, then $x : S\tau$ is in $S(A.x : \tau)$, so $S(A.x : \tau|A) \vdash x : S\tau$ is a valid derivation.
2. If the derivation consists of just the (I-intro) step: $A|A \vdash \star : \mathbf{I}$, then $S(A|A) \vdash \star : S\mathbf{I}$ follows trivially.
3. If the last step of the derivation was (I-elim), then by the inductive hypothesis twice, there are two valid derivations $S(A|A_1) \vdash t : S\mathbf{I}$ and $S(A_1|A_2) \vdash u : S\alpha$. By application of the (I-elim) rule, we obtain $S(A|A_2) \vdash \texttt{let}\ t\ \texttt{be}\ \star\ \texttt{in}\ u : S\alpha$, as required.
4. If the last step of the derivation was (!-elim : weakening), then by the inductive hypothesis twice, we have derivations $S(A|A_1) \vdash t : S!\alpha$ and $S(A_1|A_2) \vdash u : S\beta$. Now, since $S!\alpha = !(S\alpha)$ we apply the (!-elim : weakening) rule to obtain the required derivation. □

Theorem 3.3 (Soundness of $\mathscr{L}$)
If $\mathscr{L}(A, e)$ succeeds with (S, τ, A') then there is a derivation of $S(A|A') \vdash e : \tau$.

Proof
By induction on the structure of terms e.
There are 17 cases, we will show just seven.

1. If e is the identifier x, $x : \alpha \in A.x : \alpha$ then $\mathscr{L}(A.x : \alpha, e)$ succeeds immediately with (id, α, A) and there is a derivation consisting of the (Axiom) $: A.x : \alpha|A \vdash x : \alpha$. The result follows since $\mathsf{id}(A|A') = A|A'$.
2. If e is $\star$ then $\mathscr{L}(A, e)$ succeeds immediately with $(\mathsf{id}, \mathbf{I}, A)$ and there is a derivation consisting of the (I-intro) rule: $A|A \vdash \star : \mathbf{I}$. The result follows since $\mathsf{id}(A|A) = A|A$.
3. If e is of the form $\texttt{let}\ t\ \texttt{be}\ x \otimes y\ \texttt{in}\ u$ then $\mathscr{L}(A, t)$ succeeds with (R, τ, A_1), $\mathscr{V}\ \tau\ \alpha \otimes \beta$ succeeds with a substitution U and $\mathscr{L}(URA_1.x : \alpha.y : \beta, u)$ succeeds with (S, γ, A_2). By the inductive hypothesis twice, there are derivations $R(A|A_1) \vdash t : \tau$ and $SUR(A_1.x : \alpha.y : \beta|A_2) \vdash u : \gamma$. By Lemma 3.2 the first derivation can be written as $SUR(A|A_1) \vdash t : SU\tau$, *i.e.* $SUR(A|A_1) \vdash t : S(U\alpha \otimes U\beta)$. By the (⊗-elim) rule there is a derivation $SUR(A_1|A_2) \vdash \texttt{let}\ t\ \texttt{be}\ x \otimes y\ \texttt{in}\ u : \gamma$.

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

4. If e is of the form $< t, u >$ then $\mathscr{L}(A, t)$ succeeds with (R, α, A_1) and $\mathscr{L}(RA, u)$ succeeds with (S, β, A_2). By the inductive hypothesis twice there are derivations $R(A|A_1) \vdash t : \alpha$ and $SR(A|A_2) \vdash u : \beta$. By Lemma 3.2 we can write the first derivation as $SR(A|A_1) \vdash t : S\alpha$. By the (& -intro) rule and the condition that $A_1 = A_2$ there is a derivation $SR(A|A_1) \vdash t : (S\alpha)\ \&\ \beta$.
5. If e is of the form `let` t `be` x@y `in` u then $\mathscr{L}(A, t)$ succeeds with (R, ρ, A_1), $\mathscr{V}\ \rho\ !\alpha$ succeeds with a substitution U and $\mathscr{L}(URA_1.x :!\alpha.y :!\alpha, u)$ succeeds with (S, β, A_2). By the inductive hypothesis twice, there are derivations $R(A|A_1) \vdash t : \rho$ and $SUR(A_1.x :!\alpha.y :!\alpha|A_2) \vdash u : \beta$. By Lemma 3.2 the first derivation can be written as $SUR(A|A_1) \vdash t : SU\rho$ *i.e.* $SUR(A|A_1) \vdash t : S(!U\alpha)$. By the (!-elim : contraction) rule there is a derivation $SUR(A|A_2) \vdash$ `let` t `be` x@y `in` $u : \beta$.
6. If e is of the form `let` t `be` $!x$ `in` u then $\mathscr{L}(A, t)$ succeeds with (R, β, A_1), $\mathscr{V}\ \beta\ !\alpha$ succeeds with a substitution U and $\mathscr{L}(URA_1.x : \alpha, u)$ succeeds with (S, ρ, A_2). By the inductive hypothesis twice, there are derivations $R(A|A_1) \vdash t : \beta$ and $SUR(A_1.x : \alpha|A_2) \vdash u : \rho$. By Lemma 3.2 the first derivation can be written as $SUR(A|A_1) \vdash t : SU\beta$, which rewrites to $SUR(A|A_1) \vdash t : S(!U\alpha)$. By the (!-elim : dereliction) rule there is a derivation $SUR(A|A_2) \vdash$ `let` t `be` $!x$ `in` $u : \rho$.
7. If e is of the form $!t$ then $\mathscr{L}(A, t)$ succeeds with (R, α, A_1) and $R(A \backslash A_1)$ are all !-assumptions. By the inductive hypothesis, there is a derivation $R(A|A_1) \vdash t : \alpha$, and by the (!-intro : promotion) rule there is a derivation $R(A|A_1) \vdash !t :!\alpha$ as required. □

3.4 Completeness of $\mathscr{L}$

If a term can be typed using the inference rules, then we would require our algorithm to also be able to compute the type of this term. The proof given will follow very closely the proof of the completeness of $\mathscr{W}$ in Damas (1985). Note that since we have monomorphic types the notion of a *principle* type follows as an immediate corollary of the completeness theorem.

We will need some new notation, which is taken from Damas (1985). We write

$$dom(S) =_{def} \{\alpha | S\alpha \neq \alpha\}$$

for the domain of a substitution;
and write $R + S$ for the simultaneous composition of R and S; defined as:

$$(R + S)\alpha =_{def} \begin{cases} R\alpha & \text{if } \alpha \in dom(R) \\ S\alpha & \text{otherwise} \end{cases}$$

Additionally, we will call a substitution minimal if it has the smallest domain possible for the required substitution.

Theorem 3.4 (Completeness of $\mathscr{L}$)

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

If there is a derivation $S(A|A_1) \vdash e : \tau$ for some substitution S, then:

1. $\mathscr{L}(A, e)$ succeeds with (R, α, A_1) for some R, α.
2. There exists a substitution T such that: $TR(A|A_1) = S(A|A_1)$ and $T\alpha = \tau$.

Proof

By induction over the structure of linear terms e.

There are 17 cases, we will show just seven.

1. If e is the variable x, then the derivation just consists of the (Axiom) : $S(A.x : \alpha|A) \vdash x : S\alpha$. Now, $\mathscr{L}(A.x : \alpha, x)$ succeeds with (id, α, A) and we must find a substitution T such that $T\mathrm{id}(A.x : \alpha|A) = S(A.x : \alpha|A)$ and $T\alpha = S\alpha$. Let the substitution just be S and we are done.
2. If e is of the form $\star$, then the derivation consists of just the (**I**-intro) rule: $S(A|A) \vdash \star : \mathbf{I}$. Now $\mathscr{L}(A, e)$ succeeds immediately with $(\mathrm{id}, \mathbf{I}, A)$ and since $\mathbf{I}$ contains no type variables (2) follows trivially by letting the required substitution be just S.
3. If e is of the form `let` t `be` $\star$ `in` u, then there is a derivation ending in the (**I**-elim) rule, where the antecedents are: $S(A|A_1) \vdash t : \mathbf{I}$ and $S(A_1|A_2) \vdash u : \gamma$. By the inductive hypothesis, $\mathscr{L}(A, t)$ succeeds with (R_1, β, A_1) and there is a substitution T_1 such that $T_1R_1(A|A_1) = S(A|A_1)$ and $T_1\beta = \mathbf{I}$. To show $U = \mathscr{V}\ \beta\ \mathbf{I}$ is a unifying substitution observe that there is a substitution U_0 which satisfies: $U_0\beta = U_0\mathbf{I}$. Now, since $\mathbf{I}$ is a type constant this must satisfy $U_0\beta = \mathbf{I}$; hence the substitution U_0 is just T_1 from above. Now $\mathscr{V}\ \beta\ \mathbf{I}$ succeeds with some substitution U, which is the most general unifier— *i.e.* we have $XU = U_0$ for some substitution X. Hence $XUR_1(A|A_1) = S(A|A_1)$. Again by the inductive hypothesis, $\mathscr{L}(UR_1A_1, u)$ succeeds with (R_2, a, UR_1A_2), and there is a substitution T_2 satisfying $T_2R_2UR_1(A_1|A_2) = XUR1(A_1|A_2)$ and $T_2\alpha = \gamma$. $\mathscr{L}(A,$ `let` t `be` $\star$ `in` $u)$ succeeds as required with (R_2UR_1, a, A_2). To show clause (2) of the theorem we must exhibit a substitution T which satisfies $TR_2UR_1(A|A_2) = S(A|A_2)$ and $T\alpha = \gamma$. Let this substitution be T_2 from above and we are done.
4. If e is of the form $\mathtt{inl}(t)$, then there is a derivation ending in the ($\oplus$-intro : left) rule : $S(A|A_1) \vdash \mathtt{inl}(t) : \tau \oplus \tau'$. The antecedent of this rule is $S(A|A_1) \vdash t : \tau$. By the inductive hypothesis $\mathscr{L}(A, t)$ succeeds with (R, α, A_1) and there is a substitution T_1 such that $T_1R(A|A_1) = S(A|A_1)$ and $T_1\alpha = \tau$. Now, $\mathscr{L}(A, \mathtt{inl}(t))$ succeeds as required with $(R, \alpha \oplus \beta, A_1)$ where β is a new type variable. To show clause (2) of the theorem all we have to do is show that there is a substitution T such that $TR(A|A_1) = S(A|A_1)$ and $T(\alpha \oplus \beta) = \tau \oplus \tau'$. If we let T be $[\tau'/\beta] + T_1$, then $TR(A|A_1) = S(A|A_1)$ from before since β does not occur in $(A|A_1)$, and $T(\alpha \oplus \beta) = T\alpha \oplus T\beta = T_1\alpha \oplus \tau' = \tau \oplus \tau'$, as required.
5. If e is of the form `let` t `be _ in` u, then there is a derivation ending in the (!-elim : weakening) rule: $S(A|A_2) \vdash$ `let` t `be _ in` $u : \gamma$. The antecedents to this rule are $S(A|A_1) \vdash t :!\alpha$ and $S(A_1|A_2) \vdash u : \gamma$. By the inductive hypothesis, $\mathscr{L}(A, t)$ succeeds with (R_1, τ, A_1) and there is a substitution T_1 satisfying $T_1R_1(A|A_1) = S(A|A_1)$ and $T_1\tau = !\alpha$. To show that $\mathscr{V}\ \tau\ !\epsilon$ is a

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

unifying substitution, let $U_0 = [\epsilon/\alpha] + T_1$. Now, $U_0\tau = T_1\tau = !\alpha$, and $U_0!\epsilon = !(U_0\epsilon) = !\alpha$. So, U_0 is a unifying substitution, therefore $\mathscr{V}\ \tau\ !\epsilon$ succeeds with some substitution U. Since U is the most general unifier, there is a substitution X such that $XU = U_0$, hence we have $XUR_1(A|A_1) = S(A|A_1)$ since e does not occur in $(A|A_1)$. Now, $\mathscr{L}(UR_1A_1, u)$ succeeds by the inductive hypothesis with (R_2, β, UR_1A_2), and there is a substitution T_2 such that $T_2R_2UR_1(A|A_1) = XUR_1(A|A_1)$ and $T_2\beta = \gamma$. Now, $\mathscr{L}(A, \texttt{let}\ t\ \texttt{be}\ \_\ \texttt{in}\ u)$ succeeds as required with (R_2UR_1, β, A_2), all we are left to show is that there is a substitution T such that $TR_2UR_1(A|A_2) = S(A|A_2)$. Let this substitution be just T_2 from above and we are done.

6. If e is of the form `let` t `be` $!x$ `in` u, then there is a derivation ending in the (!-elim : dereliction) rule: $S(A|A_2) \vdash$ `let` t `be` $!x$ `in` $u : \gamma$. The antecedents to this rule are: $S(A|A_1) \vdash t : !\alpha$ and $S(A_1.x : \alpha|A_2) \vdash u : \gamma$. By the inductive hypothesis on the first antecedent, $\mathscr{L}(A, t)$ succeeds with (R_1, β, A_1) and there is a substitution T_1 satisfying $T_1R_1(A|A_1) = S(A|A_1)$ and $T_1\beta = !\alpha$. To show that $\mathscr{V}\ \beta\ !\epsilon$ is a unifying substitution, observe that $U_0 = [\alpha/\epsilon] + T_1$ is a valid substitution satisfying the following: $U_0\beta = U_0!\epsilon$ which simplifies to $T_1\beta = !\alpha$, which is satisfied from above. Because U_0 is a unifying substitution, $\mathscr{V}\ \beta\ !\epsilon$ succeeds with the most general unifier U, such that $XU = U_0$ for some substitution X. Since α is not free in A, we also have $XUR_1(A|A_1) = S(A|A_1)$. By the inductive hypothesis again, $\mathscr{L}(UR_1A_1.x : \alpha, u)$ succeeds with (R_2, ρ, UR_1A_2) and there is a substitution T_2 satisfying $T_2R_2UR_1(A_1.x : \alpha|A_2) = XUR_1(A_1.x : \alpha|A_2)$ and $T_2\rho = \gamma$. Now $\mathscr{L}(A, \texttt{let}\ t\ \texttt{be}\ !x\ \texttt{in}\ u)$ succeeds as required with (R_2UR_1, ρ, A_2). To show clause (2) of the theorem we must exhibit a substitution T satisfying $TR_2UR_1(A|A_2) = S(A|A_2)$ and $T\rho = \gamma$. Let this substitution be just T_2 and we are done.

7. If e is of the form $< t, u >$, then there is a derivation ending in the (& -intro) rule: $S(A|A_1) \vdash < t, u > : \tau\ \&\ \tau'$. The antecedents to this rule are $S(A|A_1) \vdash t : \tau$ and $S(A|A_1) \vdash u : \tau'$. By the inductive hypothesis on the first antecedent, $\mathscr{L}(A, t)$ succeeds with (R_1, α, A_1) and there is a substitution T_1 such that $T_1R_1(A|A_1) = S(A|A_1)$ and $T_1\alpha = \tau$. Again, by the inductive hypothesis on the second antecedent, $\mathscr{L}(R_1A, u)$ succeeds with (R_2, β, R_1A_1), and there is a substitution T_2 satisfying $T_2R_2R_1(A|A_1) = T_1R_1(A|A_1)$ and $T_2\beta = \tau'$. Now, $\mathscr{L}(A, < t, u >)$ succeeds with $(R_2R_1, R_2\alpha\ \&\ \beta, A_1)$ as required. To show clause (2) of the theorem, observe that $T_2R_2\alpha = T_1\alpha$, all $\alpha \in R_1A$. Hence, let the substitution required be T_2, then $TR_2R_1(A|A_1) = S(A|A_1)$ from above, and $T(R_1\alpha\ \&\ \beta) = TR_2\alpha\ \&\ T\beta = \tau\ \&\ \tau'$ as required. □

3.5 Polymorphism in $\mathscr{L}$

Throughout this paper we will work with *monomorphic* types, although we do have an *ad hoc* polymorphic construct in our implementation which we will discuss below.

The reason we refrain from including a `polylet` construct (*cf.* Milner's polymorphic `let` construct) is that it is not clear how one should include such entities into a

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

linear framework. We give the general idea of the problem and suggest one possible solution.

We begin with the archetypical example of a polymorphic function from the λ-calculus. (*cf.* `let` $I = \lambda x.x$ `in` II in Milner's notation.)

$$\texttt{polylet}\ \lambda x.x\ \texttt{be}\ I\ \texttt{in}\ II$$

We can extend our linear term calculus with such a construct, for which we might write:

$$\texttt{polylet}\ !(\lambda x.x)\ \texttt{be}\ I\ \texttt{in let}\ I\ \texttt{be}\ f@g\ \texttt{in let}\ !f\ \texttt{be}\ f'\ \texttt{in let}\ g\ \texttt{be}\ !g'\ \texttt{in}\ f'g'$$

Note first that we needed to promote the function since we are copying it–an intrinsic part of the pragmatics of polymorphism. Polymorphism is very closely linked with Contraction–one does not need polymorphism if one never uses the function more than once!

The basic problem is that the *generalisation* of the type of I will be $\forall\alpha.!(\alpha \multimap \alpha)$ which does *not* type check correctly, i.e. this indicates that we instantiate, then copy–we want the converse $!\forall\alpha.(\alpha \multimap \alpha)$, copy then instantiate.

The solution we propose is to have a `polylet` construct which allows us to write:

$$\texttt{polylet}\ \lambda x.x\ \texttt{be}\ I\ \texttt{in}\ II$$

i.e. we omit the exponential and the `polylet` construction is allowed to use the function any number of times. We feel this is the only reasonable solution when we look at the general case:

$$\texttt{polylet}\ e_1,\ldots,e_n\ \texttt{be}\ x_1,\ldots x_n\ \texttt{in}\ e$$

which would arise in the *standard environment* of a functional programming language for example. It would not be very sensible to have the programmer weakening all the functions in the standard environment before using just one of them. It is for this reason we adopt this solution since it appears to be the most satisfying solution both theoretically and practically. For the moment this has only be implemented in Lilac at the outer level; within the standard environment. The extension of the calculus to allow local polymorphic declarations is one gap in our work.

To summarise, the terms of a `polylet` construct will be typed using $\mathscr{L}$, then *generalised* and Promoted. This, of course, is allowed since we are dealing with programs (closed terms). Lilac will automatically use Contraction, Weakening, Dereliction and *instantiation*–the *linking phase*–while typing the program to be executed.

We therefore have two additional logical rules that we need to add to complete our natural deduction presentation of intuitionistic linear logic:

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash t : \forall\alpha.\tau}\ \text{(generalise)} \qquad \frac{\Gamma \vdash t : \forall\alpha.\tau}{\Gamma \vdash t : \tau[\tau'/\alpha]}\ \text{(specialise)}$$

Where the side condition that α is not free in Γ applies to the (generalise) rule. Note that we do *not* have any additional syntax for polymorphism. The justification

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

of this method comes from the observation that the standard environment is just a series of definitions–if we want to use any of them, then we take a copy and compile that definition into the object code. What we are really performing is a linking phase in the compilation, i.e. we are merely selecting the relevant components of the chosen computation.

4 Lilac: a linear functional programming language

The language of the linear term calculus, as with the λ-calculus, is a very puritanical one. There is a need therefore to place some reasonable syntax around it so that we can investigate the usefulness of the calculus as a programming language. We enrich the linear term calculus with constants and recursion, give corresponding operational semantics and typing rules, then give an SECD machine implementation. Our language Lilac will just be a sugared version of this enriched calculus.

Our implementation will be based on the *call-by-need* strategy. To be more precise, *linear* functions will be implemented as call-by-value: we don't need to build closures for linear functions since we know we will need the value. Non-linear functions will be implemented as call-by-need: we evaluate the argument only if it is needed in the computation and moreover we will share the result once computed. The elegant mechanism which will decide which strategy to use comes directly from the type information: no additional analyses (such as strictness analysis) are required.

Our implementation however will not take advantage of the garbage collection and in-place updates mentioned in the introduction. Details of these issues can be found in the paper by Chirimar *et al.* (1991).

4.1 Extending the linear term calculus

It is possible to program in the pure linear term calculus and implement various data types by coding tricks. We prefer, however, to add several constants to the calculus for both practical and efficiency purposes. Turner (1991) gives a good discussion on the relative merits of adding constants to a pure calculus.

We do not provide *user* defined data-types, but several useful built-in data-types. The extension to user defined data-types is where we feel we can start to see many advantages of working within a linear framework. The ability to have both lazy and eager data structures within a unified system is a rather novel feature. For example we could define binary trees which are lazy in one branch and eager in the other. It is the exponential that gives this power of control. We leave the study of data structures for further work.

Natural numbers and Booleans

We build in the *eager* natural numbers†, nat, defined by the recursive type equation $\mathtt{nat} = \mathbf{I} \oplus \mathtt{nat}$. Our intended interpretation will be $\mathtt{zero} = \mathtt{inl}(\star)$ and $\mathtt{succ} = \mathtt{inr}$, e.g. $2 = \mathtt{inr}(\mathtt{inr}(\mathtt{inl}(\star)))$.

For Boolean values we have the type equation $\mathtt{bool} = \mathbf{I} \oplus \mathbf{I}$, our interpretation being $\mathtt{true} = \mathtt{inl}(\star)$, and $\mathtt{false} = \mathtt{inr}(\star)$.

We provide some standard functions over these types : $+, -, *, \mathtt{div}, \mathtt{mod}, <, =$ for nat and `or, and, not` for bool. The operational behaviour and typing rules for these constants are standard. The conditionals for these two types are derived from the $\oplus$—elim rules given in Fig. 2. The calculus is enriched by a casenat and a casebool construct. The operational semantics are given by:

$$\frac{n \Downarrow \mathtt{zero} \quad u \Downarrow c}{\mathtt{casenat}\ n\ \mathtt{of}\ \mathtt{zero} \Rightarrow u \mid \mathtt{succ}(m) \Rightarrow v \Downarrow c}$$

$$\frac{n \Downarrow \mathtt{succ}(d) \quad v[d/m] \Downarrow c}{\mathtt{casenat}\ n\ \mathtt{of}\ \mathtt{zero} \Rightarrow u \mid \mathtt{succ}(m) \Rightarrow v \Downarrow c}$$

$$\frac{b \Downarrow \mathtt{true} \quad u \Downarrow d}{\mathtt{casebool}\ b\ \mathtt{of}\ \mathtt{true} \Rightarrow u \mid \mathtt{false} \Rightarrow v \Downarrow d}$$

$$\frac{b \Downarrow \mathtt{false} \quad v \Downarrow d}{\mathtt{casebool}\ b\ \mathtt{of}\ \mathtt{true} \Rightarrow u \mid \mathtt{false} \Rightarrow v \Downarrow d}$$

The typing rules for these two additional constructs are given by:

$$\frac{\Gamma \vdash n : \mathtt{nat} \quad \Delta \vdash u : \alpha \quad m : \mathtt{nat}, \Delta \vdash v : \alpha}{\Gamma, \Delta \vdash \mathtt{casenat}\ n\ \mathtt{of}\ \mathtt{zero} \Rightarrow u \mid \mathtt{succ}(m) \Rightarrow v : \alpha}\ (\mathtt{nat—elim})$$

$$\frac{\Gamma \vdash b : \mathtt{bool} \quad \Delta \vdash u : \alpha \quad \Delta \vdash v : \alpha}{\Gamma, \Delta \vdash \mathtt{casebool}\ b\ \mathtt{of}\ \mathtt{true} \Rightarrow u \mid \mathtt{false} \Rightarrow v : \alpha}\ (\mathtt{bool—elim})$$

Eager lists

The type of eager lists in our linear system is given by the recursive type equation $\mathtt{list}(A) = \mathbf{I} \oplus (A \otimes \mathtt{list}(A))$. We write '[ ]' for the empty list, and ':' for the right associative cons operator. We have an abbreviation and write $[1,2,3,4]$ for $1 : 2 : 3 : 4 : [\,]$. The conditional for lists is again a derivative of the $\oplus$—elim rules.

The operational semantics of lists is given by the following:

$$\frac{h \Downarrow c \quad t \Downarrow d}{h : t \Downarrow c : d} \qquad \frac{}{[\,] \Downarrow [\,]}$$

† The *lazy* natural numbers would be defined as $\mathtt{nat} = \mathbf{I} \oplus !\mathtt{nat}$.

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

$$\frac{n \Downarrow [\,] \quad u \Downarrow c}{\texttt{caselist}\ n\ \texttt{of}\ [\,] \Rightarrow u \mid (h : t) \Rightarrow v \Downarrow c}$$

$$\frac{n \Downarrow d : e \quad v[d/h, e/t] \Downarrow c}{\texttt{caselist}\ n\ \texttt{of}\ [\,] \Rightarrow u \mid (h : t) \Rightarrow v \Downarrow c}$$

and the corresponding typing rules are as follows:

$$\frac{}{\vdash [\,] : \texttt{list}(\alpha)} \quad \frac{\Gamma \vdash h : \alpha \quad \Delta \vdash t : \texttt{list}(\alpha)}{\Gamma, \Delta \vdash (h : t) : \texttt{list}(\alpha)}\ (\text{list—intro})$$

$$\frac{\Gamma \vdash n : \texttt{list}(\alpha) \quad \Delta \vdash u : \beta \quad h : \alpha, t : \texttt{list}(\alpha), \Delta \vdash v : \beta}{\Gamma, \Delta \vdash \texttt{caselist}\ n\ \texttt{of}\ [\,] \Rightarrow u \mid (h : t) \Rightarrow v : \beta}\ (\text{list—elim})$$

Lazy lists (streams)

The type of streams is given by the recursive type equation $\texttt{stream}(A) = \mathbf{I} \oplus (A \otimes !\texttt{stream}(A))$. The empty stream is denoted by '{}'; '::' is the cons operator, and again we have an abbreviation and write $\{1,2,3,4\}$ for $1 :: 2 :: 3 :: 4 :: \{\}$. The conditional for streams is again a derivative of the $\oplus$—elim rules.

The operational semantics for streams is as follows:

$$\frac{h \Downarrow c}{h :: t \Downarrow c :: t} \quad \frac{}{\{\,\} \Downarrow \{\,\}}$$

$$\frac{n \Downarrow \{\,\} \quad u \Downarrow c}{\texttt{casestream}\ n\ \texttt{of}\ \{\,\} \Rightarrow u \mid (h :: t) \Rightarrow v \Downarrow c}$$

$$\frac{n \Downarrow d :: e \quad v[d/h, e/t] \Downarrow c}{\texttt{casestream}\ n\ \texttt{of}\ \{\,\} \Rightarrow u \mid (h :: t) \Rightarrow v \Downarrow c}$$

and the typing rules are give by:

$$\frac{}{\vdash \{\,\} : \texttt{stream}(\alpha)} \quad \frac{\Gamma \vdash h : \alpha \quad \Delta \vdash t : \texttt{stream}(\alpha)}{\Gamma, \Delta \vdash (h :: t) : \texttt{stream}(\alpha)}\ (\text{stream—intro})$$

$$\frac{\Gamma \vdash n : \texttt{stream}(\alpha) \quad \Delta \vdash u : \beta \quad h : \alpha, t : \texttt{stream}(\alpha), \Delta \vdash v : \beta}{\Gamma, \Delta \vdash \texttt{casestream}\ n\ \texttt{of}\ \{\,\} \Rightarrow u \mid (h :: t) \Rightarrow v : \beta}\ (\text{stream—elim})$$

The above specifies the operational behaviour of the data types in terms of the connectives of linear logic already given. Our implementation of these data types will have the same operational behaviour of the linear logic connectives as shown, but will be implemented by enriching the calculus with the corresponding constructs; which are derived from existing linear term calculus constructs.

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

Recursion

One of the most important pieces of machinery in a functional programming language is recursion; without some form of recursion we cannot write very interesting programs. In our strongly typed calculus we have no hope of defining a 'Y' recursion combinator from the raw syntax. We must therefore introduce it as a constant to the language.

In addition to the normal typing constraints, we must also take care of the linearity aspect of the language. *e.g.* $Y(f) = f(\cdots f(\bot)\cdots)$ indicates we need many copies of f. Fortunately the exponential of linear logic allows us to copy the function f, so will will need something like: $Y(!f) = f\,!(Y(!f))$. Hence we need to derelict the value f and promote the argument for each recursive call in our programs.

We extend the linear term calculus with recursion which we will write as:

$$\texttt{letrec}\ t\ \texttt{be}\ x\ \texttt{in}\ u$$

which has the usual interpretation:

$$\texttt{letrec}\ t\ \texttt{be}\ x\ \texttt{in}\ u = \texttt{let}\ (\texttt{rec}\ x.t)\ \texttt{be}\ x\ \texttt{in}\ u$$

Note that $\texttt{rec}\ x.t$ can be written as $Y(\lambda x.t)$.

We will present the operational semantics and typing rules just for the `rec` construct, since this is the heart of recursion. There is nothing deep in this; it just economises the presentation. The reason why we needed to introduce the `letrec` construct is for the compilation into SECD codes; to be discussed later.

The operational semantics for this construct is given by:

$$\frac{t[!\texttt{rec}\ x.t/x] \Downarrow c}{\texttt{rec}\ x.t \Downarrow c}$$

The important point is that we *promote* the $\texttt{rec}\ x.t$ at each substitution. This is an application of using the exponential as both a control (we need lazy evaluation here) and resource (we may need to copy this when we substitute) operator.

We have the corresponding typing rule which reflects this operational behaviour:

$$\frac{!\Gamma, x : !\alpha \vdash t : \alpha}{!\Gamma \vdash \texttt{rec}\ x.t : \alpha}$$

Note that we promote the type of x to type the term t. In intuitionistic logic $\texttt{rec}\ x.t$ provides the least solution to the equation $x = t$, note that in intuitionistic linear logic our presentation gives these two values *different* types. An analogy can be made here with the way the function space of intuitionistic logic is translated into linear logic using the translation due to Girard (1987), *i.e.* $(A \Longrightarrow B)^\circ$ becomes $!A^\circ \multimap B^\circ$. The Axiom: $x : A \vdash x : A$, for example, is translated into a Dereliction: $x : !A \vdash \texttt{let}\ x\ \texttt{be}\ x'\ \texttt{in}\ x' : A$. When we perform our recursive call we have to Promote the value, which is possible by the typing rule–the term t is in an !-context.

Examples

We give three simple examples of use.

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

1. The first example is the most trivial case to show how things work. We will show the reduction of 'YI' which is coded as:
   $\texttt{letrec}\ (\texttt{let}\ z\ \texttt{be}\ !z'\ \texttt{in}\ z')\ \texttt{be}\ z\ \texttt{in}\ z = \texttt{rec}\ z.\texttt{let}\ z\ \texttt{be}\ !z'\ \texttt{in}\ z'$
   The reduction proceeds as follows:

$$\begin{array}{ll} & \texttt{rec}\ z.\texttt{let}\ z\ \texttt{be}\ !z'\ \texttt{in}\ z' \\ \Downarrow & \texttt{let}\ (!\texttt{rec}\ z.\texttt{let}\ z\ \texttt{be}\ !z'\ \texttt{in}\ z')\ \texttt{be}\ !z'\ \texttt{in}\ z' \\ \Downarrow & \texttt{rec}\ z.\texttt{let}\ z\ \texttt{be}\ !z'\ \texttt{in}\ z' \\ \vdots & \end{array}$$

   i.e. an infinite computation.
2. The second example is very similar, but based on streams. The function will yield an infinite stream of ones $(1 :: 1 :: 1 :: \cdots)$. The function is coded as:
   $\texttt{letrec}\ (1 :: \texttt{let}\ x\ \texttt{be}\ !x'\ \texttt{in}\ x')\ \texttt{be}\ x\ \texttt{in}\ x = \texttt{rec}\ x.1 :: \texttt{let}\ x\ \texttt{be}\ !x'\ \texttt{in}\ x'$
   The reduction proceeds as follows:

$$\begin{array}{ll} & \texttt{rec}\ x.1 :: \texttt{let}\ x\ \texttt{be}\ x'\ \texttt{in}\ x' \\ \Downarrow & 1 :: \texttt{let}\ (!\texttt{rec}\ x.1 :: \texttt{let}\ x\ \texttt{be}\ !x'\ \texttt{in}\ x')\ \texttt{be}\ !x'\ \texttt{in}\ x' \end{array}$$

   Which terminates by the operational semantics of streams. A forced evaluation of the tail of the stream will perform the dereliction and so on.
3. Finally, we give a coding of the infamous factorial function within this linear framework.
   Recall that in an Intuitionistic framework we would represent this function as follows:

$$fact = \texttt{rec}\ f.\lambda n.\texttt{case}\ n\ \texttt{of}\ 0\ \Rightarrow\ 1 \mid \texttt{succ}(n)\ \Rightarrow\ n * f(n-1)$$

   In the enriched linear term calculus we get:

$$\begin{array}{l} fact\ =\ \texttt{rec}\ f.\lambda n.\ \texttt{let}\ n\ \texttt{be}\ n_1 @ n_2\ \texttt{in} \\ \qquad\qquad \texttt{let}\ n_1\ \texttt{be}\ !n_1'\ \texttt{in} \\ \qquad\qquad \texttt{casenat}\ n_1'\ \texttt{of}\ \ 0\ \Rightarrow \texttt{let}\ f\ \texttt{be}\ \_\ \texttt{in} \\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{let}\ n_2\ \texttt{be}\ \_\ \texttt{in}\ 1 \\ \qquad\qquad\qquad\qquad\quad \texttt{succ}(m) \Rightarrow \texttt{let}\ f\ \texttt{be}\ !f'\ \texttt{in} \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{succ}(m) * f'(n_2) \end{array}$$

   which will get a type : $!(!\texttt{nat} \multimap \texttt{nat})$.

This last example looks a rather complicated way of writing one of the simplest recursive functions. Fundamentally the problem is the excessive number of resource management operations which causes the problems. In our language Lilac we will improve the syntax by allowing nested patterns which will shorten the definition, but it remains a fact that we have to explicitly Contract, Weaken and Derelict many values. It is examples like these that are starting to show that the usefulness of such

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

a language may be limited–no programmer would like to have to write a function in such a complicated way. We will come back to this point in the conclusions, but next we show an alternative way of coding many functions in this calculus in a much simpler way.

Iterators

The calculus is enriched with iterators for the natural numbers, lists and streams. Since all these iterators are essentially the same, we just mention the case for natural numbers. An iterator for the natural numbers gives us the power to repeatedly apply a function to an argument a specified number of times. We enrich the syntax of the linear term calculus with the `iternat` construct:

$$\texttt{iternat}(n,f,b) = \underbrace{(f \circ \cdots \circ f)}_{n} b$$

Whereas in general recursion we do not know how many times we need to *recurse*, here we have the contrary–the iterator tells us exactly how many copies of the function f we need to make. It is for this reason we will work without exponentials: an iterator will take a function *without* an '!' around it and use it many times. We are safe to do this since we will ensure that the term has only free variables of the !-type, to made clear in the typing rule below. It is the iterator that will effectively do the Contraction, Weakening and Dereliction for us.

We can give this an explicit operational semantics as follows:

$$\frac{n \Downarrow \texttt{zero} \quad b \Downarrow c}{\texttt{iternat}(n,f,b) \Downarrow c} \qquad \frac{n \Downarrow \texttt{succ}(m) \quad \texttt{iternat}(m,f,b) \Downarrow c \quad fc \Downarrow d}{\texttt{iternat}(n,f,b) \Downarrow d}$$

with the corresponding typing rule:

$$\frac{!\Gamma \vdash f : \alpha \multimap \alpha \quad \Delta \vdash b : \alpha}{n : \texttt{nat}, !\Gamma, \Delta \vdash \texttt{iternat}(n,f,b) : \alpha}$$

Note that this is a derived rule; we have used the promotion rule for the function f. Note also that this function is the *only* non-linear part of the iterator–the base value b and the number of iterations n are used only once.

The expressive power of the iterator should not be underestimated. The exact class of functions we can represent with this iterator is an open question, but we get at least all of the primitive recursive functions, even *without* using the exponential '!' (Mackie *et al.*, 1993). The following example shows that we can get a cartesian structure from the tensor product, i.e. we can both project and duplicate natural numbers:

- $fst = \lambda x.\texttt{let } x \texttt{ be } u \otimes v \texttt{ in iternat}(v, \lambda z.z, u) : \texttt{nat} \otimes \texttt{nat} \multimap \texttt{nat}$
- $snd = \lambda x.\texttt{let } x \texttt{ be } u \otimes v \texttt{ in iternat}(u, \lambda z.z, v) : \texttt{nat} \otimes \texttt{nat} \multimap \texttt{nat}$
- $copy = \lambda x.\texttt{iternat}(x, \lambda y.\texttt{let } y \texttt{ be } a \otimes b \texttt{ in } (a+1) \otimes (b+1), 0 \otimes 0) : \texttt{nat} \multimap \texttt{nat} \otimes \texttt{nat}$

Note that this works by using the structure of the natural numbers. We can use

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

the same technique for other recursive data types such as lists of natural numbers, lists of lists, etc.

It is evident that there is a relation between `nat` and `!nat`. The exact relationship is beyond the scope of the present paper, but one can find models of linear logic with natural numbers objects such that `!nat` $\simeq$ `nat`. This basic result gives us the power to write programs with natural numbers without the exponential. It would be interesting to investigate just how useful this !-free language would be.

As a final example we will give an iterative version of factorial:

$$fact = \lambda n.snd(\texttt{iternat}(n, \lambda z.\ \texttt{let}\ z\ \texttt{be}\ x \otimes y\ \texttt{in}\ \texttt{let}\ copy(x)\ \texttt{be}\ a \otimes b\ \texttt{in}\ (a+1) \otimes (b * y), 1 \otimes 1))$$

where *snd* and *copy* are defined above. This version of factorial gets a more linear type: `nat` $\multimap$ `nat`.

It is an interesting point that functions which are primitive recursive are much easier to express using the iterator and the defined copying (*copy*) and projection (*fst, snd*) functions than in general recursion; the factorial functions defined exemplify this. One could hypothesise in saying that a Linear Logic is guiding the programmer to use the appropriate level of machinery–it is well known that general recursion is overkill for many algorithms. More solid connections must be left to further work.

4.2 The linear SECD machine

The linear SECD machine is not the most efficient implementation of the linear term calculus. It is, however, our *de facto* standard; all other implementations should give the same results. It is also a nice mathematical account of an implementation.

The machine that we shall present is an extension of the machine presented in Abramsky (1993). We refer the reader to that paper for the exposition of of the unextended language.

The machine is based on a list structured memory where functions are implemented as *call-by-need*–an argument will only be evaluated if it is used and, moreover, we will share this result on successive uses.

Extending the machine

To support the extensions to the calculus, we need to extend the original Linear SECD machine. Our presentation will be of the entire machine; we will follow Henderson (1980) in our presentation–particularly for the implementation of recursion.

The machine is based on four stacks: Stack, Environment, Control and Dump. We will use a list notation for each of the stacks; using ':' for a right associative *cons* operation and '[ ]' for the empty list.

Definition 4.1 (Linear SECD instruction codes)

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

NUM(n)	ADD	SUB	MUL	DIV
NOT	AND	OR	PUSHENV	RET
PUSH	POP	HD	TL	AP
UNIT	UNUNIT	PAIR	UNPAIR	FST
SND	INL	INR	CASE(c_1, c_2)	READ
DUP	MAKEFCL(c)	MAKECCL(c_1, c_2)	MAKEOCL(c)	NIL
DUM	UPD	BOOL(b)	MOD	EQ
LT	CONS	CASELIST(c_1, c_2)	CASENAT(c_1, c_2)	ITER(n, c)
IT(c_1, c_2)	CONSSTREAM	CASESTREAM(c_1, c_2)	CASEBOOL	NILSTREAM
RAP				

where n is a natural number, b is a Boolean, and c, c_1, c_2 are codes.

The following are the values:

Definition 4.2 (Values)

num(n)	bool(b)	env(e)	$\star$	nul
inr(v)	inl(v)	fcl(c, e)	ccl(c_1, c_2, e)	ocl(c, e)
tuple(v_1, v_2)	dummy(ref(v))	list([v])	stream([v])	ocv(v)

where n is a Natural Number; b a Boolean; v, v_1, v_2 are values; c, c_1, c_2 are codes; e is an environment. Note that dummy holds a *reference* to a value. We assume our machine has reference values (*cf.* Standard ML (Milner et al. 1990)) which are required for the efficient coding of recursion and our implementation of call-by-need; we also assume our machine is equipped with basic operations on base data types, e.g. $+, -$, mul, div, mod.

Our stacks have the following types:

- S : list of value
- E : list of value
- C : list of instruction
- D : list of (S,E,C) triples

We now give the transition rules for each of the possible states of the machine. Each rule corresponds to our atomic computation step.

Figure 3 shows the state transitions for the linear SECD machine, and is taken from Abramsky (1990), with a couple of alterations. Figure 4 shows the extensions to the linear SECD machine to handle the enriched linear term calculus. The constants ($C1$) to ($C10$) are straightforward. We do not show all of the compilation rules–just a

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

$$
\begin{array}{lcl}
s, e, \mathrm{UNIT} : c, d & \Longrightarrow & \star : s, e, c, d \\
\star : s, e, \mathrm{UNUNIT} : c, d & \Longrightarrow & s, e, c, d \\
s, e, \mathrm{PUSHENV} : c, d & \Longrightarrow & \mathtt{env}(e) : s, e, c, d \\
\mathtt{env}(v : l) : s, e, \mathrm{HD} : c, d & \Longrightarrow & v : s, e, c, d \\
\mathtt{env}(v : l) : s, e, \mathrm{TL} : c, d & \Longrightarrow & \mathtt{env}(l) : s, e, c, d \\
v : s, e, \mathrm{PUSH} : c, d & \Longrightarrow & s, v : e, c, d \\
s, v : e, \mathrm{POP} : c, d & \Longrightarrow & s, e, c, d \\
 & & \\
s, e, \mathrm{MAKEFCL}(c_1) : c, d & \Longrightarrow & \mathtt{fcl}(c_1, e) : s, e, c, d \\
\mathtt{fcl}(c_1, e_2) : v : s, e, \mathrm{AP} : c, d & \Longrightarrow & [\,], v : e_1, c_1, (s, e, c) : d \\
v : s, e, \mathrm{RET} : c, (s_1, e_1, c_1) : d & \Longrightarrow & v : s_1, e_1, c_1, d \\
 & & \\
v : s, e, \mathrm{INL} : c, d & \Longrightarrow & \mathtt{inl}(v) : s, e, c, d \\
v : s, e, \mathrm{INR} : c, d & \Longrightarrow & \mathtt{inr}(v) : s, e, c, d \\
\mathtt{inl}(v) : s, e, \mathrm{CASE}(c_1, c_2) : c, d & \Longrightarrow & [\,], v : e, c_1, (s, e, c) : d \\
\mathtt{inr}(v) : s, e, \mathrm{CASE}(c_1, c_2) : c, d & \Longrightarrow & [\,], v : e, c_2, (s, e, c) : d \\
 & & \\
v : w : s, e, \mathrm{PAIR} : c, d & \Longrightarrow & \mathtt{tuple}(v, w) : s, e, c, d \\
\mathtt{tuple}(v, w) : s, e, \mathrm{UNPAIR} : c, d & \Longrightarrow & v : w : s, e, c, d \\
s, e, \mathrm{MAKECCL}(c_1, c_2) : c, d & \Longrightarrow & \mathtt{ccl}(c_1, c_2, e) : s, e, c, d \\
\mathtt{ccl}(c_1, c_2, e_1) : s, e, \mathrm{FST} : c, d & \Longrightarrow & [\,], e_1, c_1, (s, e, c) : d \\
\mathtt{ccl}(c_1, c_2, e_1) : s, e, \mathrm{SND} : c, d & \Longrightarrow & [\,], e_1, c_2, (s, e, c) : d \\
 & & \\
s, e, \mathrm{MAKEOCL}(c_1) : c, d & \Longrightarrow & \mathtt{ocl}(c_1, e) : s, e, c, d \\
l :: \mathtt{ocl}(c_1, e_1) : s, e, \mathrm{READ} : c, d & \Longrightarrow & [\,], e_1, c_1, l : (s, e, c) : d \\
\mathtt{ocv}(v) : s, e, \mathrm{READ} : c, d & \Longrightarrow & v : s, e, c, d \\
v : s, e, \mathrm{DUP} : c, d & \Longrightarrow & v : v : s, e, c, d \\
v : s, e, \mathrm{UPD} : c, l : (s_1, e_1, c_1) : d & \Longrightarrow & v : s_1, e_1, c_1, d \\
 & & \text{where } l := \mathtt{ocv}(v)
\end{array}
$$

Fig. 3. SECD machine transitions for the linear term calculus.

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

selection. We assume our machine has built in operations for primitive functions over the Booleans and natural numbers. The selectors ($S1$) to ($S8$) are straightforward and are derived from the general `case` construct in the pure calculus.

For the efficient implementation of recursion on an SECD machine we build a circular environment by placing a *hole* (`ref(nul)`) on the environment stack ($R1$) in preparation for the recursive application. A recursive application ($R2$) *ties the knot* by assigning this hole to the function. Rule ($R3$) is a cleanup rule which extracts the value from the referencing mechanism.

The implementation of iteration is straightforward. We compute the value of the number of iterations and apply the function f that many times to the argument, which is placed at the head of the stack. Rule ($I1$) sets up the stacks so that the base value is computed, and packages up the function and number of iterations required in the ITER instruction. Rules ($I2$) and ($I3$) apply the function to the base value the appropriate number of times. There is scope for optimisation here since we could partially evaluate the function before making each copy.

Compiling the extended language

We define the compilation of the enriched linear term calculus in the style of Henderson (1980) in the following sense: we define a function $\mathscr{C}(t, l, c)$, where t is the term for compilation, l is a list of variables and c is the code to be executed after t; a continuation.

Constants

$$
\begin{aligned}
\mathscr{C}(\texttt{var}(x), l, c) &= \text{PUSHENV} : (\text{LOOKUP}(x, l, \text{HD} : c)) \\
\mathscr{C}(\texttt{let } t \texttt{ be } \star \texttt{ in } u, l, c) &= \mathscr{C}(t, l, \text{UNUNIT} : \mathscr{C}(u, l, c)) \\
\mathscr{C}(\star, l, c) &= \text{UNIT} : c \\
\mathscr{C}(a + b, l, c) &= \mathscr{C}(a, l, \mathscr{C}(b, l, \text{ADD} : c)) \\
\mathscr{C}(\texttt{not}(a), l, c) &= \mathscr{C}(a, l, \text{NOT} : c) \\
\mathscr{C}(\texttt{Number}(n), l, c) &= \text{NUM}(n) : c \\
\mathscr{C}(\texttt{Boolean}(b), l, c) &= \text{BOOL}(b) : c
\end{aligned}
$$

$$
\begin{aligned}
&\mathscr{C}(\texttt{casebool } b \texttt{ of true} \Rightarrow t \mid \texttt{false} \Rightarrow f, l, c) = \\
&\quad \mathscr{C}(b, l, \text{SEL}(\mathscr{C}(t, l, [\text{RET}]), \mathscr{C}(f, l, [\text{RET}])) : c) \\
&\mathscr{C}(\texttt{casenat } v \texttt{ of zero} \Rightarrow c_1 \mid \texttt{succ}(n) \Rightarrow c_2, l, c) = \\
&\quad \mathscr{C}(v, l, \text{CASENAT}(\mathscr{C}(c_1, l, [\text{RET}], \mathscr{C}(c_2, n : l, [\text{POP}, \text{RET}]))) : c)
\end{aligned}
$$

where $\text{LOOKUP}(x, y : t, r) = \text{if } x = y \text{ then } r \text{ else } \text{LOOKUP}(x, t, \text{TL} : r)$.

Lists and Streams

$$
\mathscr{C}([\,], l, c) = \text{NIL} : c
$$

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

$(C1)$	$s, e, \text{NUM}(n) : c, d$	$\Longrightarrow$	$\texttt{num}(n) : s, e, c, d$
$(C2)$	$s, e, \text{BOOL}(b) : c, d$	$\Longrightarrow$	$\texttt{bool}(b) : s, e, c, d$
$(C3)$	$s, e, \text{NIL} : c, d$	$\Longrightarrow$	$\texttt{list}([\,]) : s, e, c, d$
$(C4)$	$v : \texttt{list}(t) : s, e, \text{CONS} : c, d$	$\Longrightarrow$	$\texttt{list}(v : t) : s, e, c, d$
$(C5)$	$s, e, \text{NILSTREAM} : c, d$	$\Longrightarrow$	$\texttt{stream}(\{\,\}) : s, e, c, d$
$(C6)$	$v : \texttt{ocl}(l, e_1) : s, e, \text{CONSSTREAM} : c, d$	$\Longrightarrow$	$\texttt{stream}(v : \texttt{ocl}(l, e_1)) : s, e, c, d$
$(C7)$	$\texttt{num}(n) : \texttt{num}(m) : s, e, \text{ADD} : c, d$	$\Longrightarrow$	$\texttt{num}(m + n) : s, e, c, d$
$(C8)$	$\texttt{num}(n) : \texttt{num}(m) : s, e, \text{LT} : c, d$	$\Longrightarrow$	$\texttt{bool}(n < m) : s, e, c, d$
$(C9)$	$\texttt{bool}(b) : s, e, \text{NOT} : c, d$	$\Longrightarrow$	$\texttt{bool}(\texttt{not}(b)) : s, e, c, d$
$(C10)$	$\texttt{bool}(b_1) : \texttt{bool}(b_2) : s, e, \text{AND} : c, d$	$\Longrightarrow$	$\texttt{bool}(b_1 \text{ and } b_2) : s, e, c, d$
$(S1)$	$\texttt{bool(true)} : s, e, \text{CASEBOOL}(c_1, c_2) : c, d$	$\Longrightarrow$	$[\,], e, c_1, (s, e, c) : d$
$(S2)$	$\texttt{bool(false)} : s, e, \text{CASEBOOL}(c_1, c_2) : c, d$	$\Longrightarrow$	$[\,], e, c_2, (s, e, c) : d$
$(S3)$	$\texttt{list}([\,]) : s, e, \text{CASELIST}(c_1, c_2) : c, d$	$\Longrightarrow$	$[\,], e, c_1, (s, e, c) : d$
$(S4)$	$\texttt{list}(h : t) : s, e, \text{CASELIST}(c_1, c_2) : c, d$	$\Longrightarrow$	$[\,], h : \texttt{list}(t) : e, c_2, (s, e, c) : d$
$(S5)$	$\texttt{stream}(\{\,\}) : s, e, \text{CASESTREAM}(c_1, c_2) : c, d$	$\Longrightarrow$	$[\,], e, c_1, (s, e, c) : d$
$(S6)$	$\texttt{stream}(h : t) : s, e, \text{CASESTREAM}(c_1, c_2) : c, d$	$\Longrightarrow$	$[\,], h : \texttt{stream}(t) : e, c_2, (s, e, c) : d$
$(S7)$	$\texttt{num(zero)} : s, e, \text{CASENAT}(c_1, c_2) : c, d$	$\Longrightarrow$	$[\,], e, c_1, (s, e, c) : d$
$(S8)$	$\texttt{num(succ}(n)) : s, e, \text{CASENAT}(c_1, c_2) : c, d$	$\Longrightarrow$	$[\,], n : e, c_2, (s, e, c) : d$
$(R1)$	$s, e, \text{DUM} : c, d$	$\Longrightarrow$	$s, \texttt{dummy(ref(nul))} : e, c, d$
$(R2)$	$\texttt{fcl}(c_1, e_1) : v : s, \texttt{dummy}(t) : e, \text{RAP} : c, d$	$\Longrightarrow$	$[\,], v :: (\texttt{tl } e_1), c_1, (s, e, c) : d$ where $t := v$
$(R3)$	$\texttt{dummy(ref}(v)) : s, e, c, d$	$\Longrightarrow$	$v : s, e, c, d$
$(I1)$	$\texttt{num}(n) : s, e, \text{IT}(b, f) : c, d$	$\Longrightarrow$	$[\,], e, b, (s, e, \text{ITER}(n, f) : c) : d$
$(I2)$	$s, e, \text{ITER}(\texttt{succ}(n), f) : c, d$	$\Longrightarrow$	$s, e, f : \text{AP} : \text{ITER}(n, f) : c, d$
$(I3)$	$s, e, \text{ITER}(\texttt{zero}, f) : c, d$	$\Longrightarrow$	s, e, c, d

Fig. 4. SECD machine transitions for the extensions.

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

$$\begin{aligned}
\mathscr{C}(\mathtt{h} : \mathtt{t}, l, c) &= \mathscr{C}(t, l, \text{CONS} : \mathscr{C}(h, l, c)) \\
\mathscr{C}(\{\,\}, l, c) &= \text{MAKEOCL}([\text{NILSTREAM}, \text{RET}]) : c \\
\mathscr{C}(\mathtt{h} :: \mathtt{t}, l, c) &= \text{MAKEOCL}(\mathscr{C}(t, l, \mathscr{C}(h, l, [\text{CONSSTREAM}, \text{RET}]))) : c
\end{aligned}$$

$$\begin{aligned}
&\mathscr{C}(\mathtt{caselist}\ v\ \mathtt{of}\ [\,] \Rightarrow c_1 \mid (h : t) \Rightarrow c_2, l, c) = \\
&\quad \mathscr{C}(v, l, \text{CASELIST}(\mathscr{C}(c_1, l, [\text{RET}], \mathscr{C}(c_2, h : t : l, [\text{POP}, \text{POP}, \text{RET}]))) : c) \\
&\mathscr{C}(\mathtt{casestream}\ v\ \mathtt{of}\ \{\,\} \Rightarrow c_1 \mid (h :: t) \Rightarrow c_2, l, c) = \\
&\quad \mathscr{C}(v, l, \text{READ} : \text{CASESTREAM}(\mathscr{C}(c_1, l, [\text{RET}], \mathscr{C}(c_2, h : t : l, [\text{POP}, \text{POP}, \text{RET}]))) : c)
\end{aligned}$$

Note that the only difference between lists and streams is that we place an of-course closure around streams–we do not need any new machinery.

Recursion and Iterator

$$\mathscr{C}(\mathtt{iternat}(n, f, b), l, c) = \mathscr{C}(n, l, \text{IT}(\mathscr{C}(b, l, [\text{RET}]), \mathscr{C}(f, l, [\,])) : c)$$

$$\begin{aligned}
&\mathscr{C}(\mathtt{letrec}\ t\ \mathtt{be}\ x\ \mathtt{in}\ u, l, c) = \\
&\quad \text{DUM} : \mathscr{C}(t, x : l, \text{MAKEFCL}(\mathscr{C}(u, x : l, [\text{POP}, \text{RET}])) : \text{RAP} : c)
\end{aligned}$$

The compilation rules for recursion places a dummy value on the environment stack, followed by code for the recursive function and the application. It is the RAP instruction that we use for the recursive application. It is for this reason we need a `letrec` construct, we need to differentiate between standard applications and recursive function applications

The compilation rule for iteration places the code for the number of iterations, followed by an IT instruction which contains the code for the base value and the function. The extended SECD machine will unpack these codes to apply the function the required number of times; as shown above.

The following compilation rules are for the pure linear term calculus and are taken (modulo notation) from Abramsky (1993).

Functions

$$\begin{aligned}
\mathscr{C}(\lambda x.t, l, c) &= \text{MAKEFCL}(\mathscr{C}(t, x : l, [\text{POP}, \text{RET}])) : c \\
\mathscr{C}(tu, l, c) &= \mathscr{C}(u, l, \mathscr{C}(t, l, \text{AP} : c))
\end{aligned}$$

Products and Sums

$$\begin{aligned}
\mathscr{C}(t \otimes u, l, c) &= \mathscr{C}(t, l, \mathscr{C}(u, l, \text{PAIR} : c)) \\
\mathscr{C}(< t, u >, l, c) &= \text{MAKECCL}(\mathscr{C}(t, l, [\text{RET}]), \mathscr{C}(u, l, [\text{RET}])) : c \\
\mathscr{C}(\mathtt{let}\ t\ \mathtt{be}\ < x, _ >\ \mathtt{in}\ u, l, c) &= \mathscr{C}(t, l, \text{FST} : \text{PUSH} : \mathscr{C}(u, x : l, \text{POP} : c)) \\
\mathscr{C}(\mathtt{let}\ t\ \mathtt{be}\ < _, y >\ \mathtt{in}\ u, l, c) &= \mathscr{C}(t, l, \text{SND} : \text{PUSH} : \mathscr{C}(u, y : l, \text{POP} : c))
\end{aligned}$$

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

$$\begin{aligned}\mathscr{C}(\texttt{inl}(t), l, c) &= \mathscr{C}(t, l, \text{INL} : c)\\ \mathscr{C}(\texttt{inr}(t), l, c) &= \mathscr{C}(t, l, \text{INR} : c)\end{aligned}$$

$$\begin{aligned}&\mathscr{C}(\texttt{let } t \texttt{ be } x \otimes y \texttt{ in } u, l, c) =\\ &\quad \mathscr{C}(t, l, \text{UNPAIR} : \text{PUSH} : \text{PUSH} : \mathscr{C}(u, x : y : l, \text{POP} : \text{POP} : c))\\ &\mathscr{C}(\texttt{case } t \texttt{ of inl}(x) \Rightarrow u \mid \texttt{inr}(y) \Rightarrow v, l, c) =\\ &\quad \mathscr{C}(t, l, \text{CASE}(\mathscr{C}(u, x : l, [\text{POP}, \text{RET}]), \mathscr{C}(v, y : l, [\text{POP}, \text{RET}])) : c)\end{aligned}$$

Exponentials

$$\begin{aligned}\mathscr{C}(!t, l, c) &= \text{MAKEOCL}(\mathscr{C}(t, l, [\text{UPD}])) : c\\ \mathscr{C}(\texttt{let } t \texttt{ be } !x \texttt{ in } u, l, c) &= \mathscr{C}(t, l, \text{READ} : \text{PUSH} : \mathscr{C}(u, x : l, \text{POP} : c))\\ \mathscr{C}(\texttt{let } t \texttt{ be } \_ \texttt{ in } u, l, c) &= \mathscr{C}(u, l, c)\end{aligned}$$

$$\begin{aligned}&\mathscr{C}(\texttt{let } t \texttt{ be } x@y \texttt{ in } u, l, c) =\\ &\quad \mathscr{C}(t, l, \text{DUP} : \text{PUSH} : \text{PUSH} : \mathscr{C}(u, x : y : l, \text{POP} : \text{POP} : c))\end{aligned}$$

4.3 Implementation overview

The principle behind the implementation is standard: we perform syntactical analyses on the programs and generate an abstract syntax tree; then translate this into our canonical functional programming language: the extended linear term calculus. It is this form that we perform type inference upon. We then translate into Linear SECD codes and evaluate the results on an implementation of this abstract machine.

Lilac is a very small language without many of the advanced features found in languages such as Standard ML. Our design philosophy was to keep things simple–a syntactically sparse language in the same spirit as Miranda‡ for example. The key features provided are:

- Script style: the current program is held in a script–a collection of function definitions.
- Pattern matching: on both built-in data structures, and the linear patterns.
- Lists: eager lists and lazy lists (streams).
- General Recursion and Iterators (for the natural numbers).
- Currying.

The BNF of the *abstract syntax* of Lilac is given in Fig. 5. Note that this syntax is ambiguous; but can be disambiguated using standard precedence rules. Functions are defined by a series of equations, with multiple left hand sides corresponding to pattern matching. The only patterns we need to generate selection code for is matching against numbers, Booleans and lists. We insist that patterns are non-overlapping and total. Additionally, the 'let' construct of the enriched linear term

‡ Miranda is a trademark of Research Software Ltd.

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

script ::= { *definition* ; }*

definition ::= [fun | funrec] { *lhs exp* }*

lhs ::= *ID* { *pattern* }*

pattern ::= *ID* | *NUM* | true | false

| [ ] (* *eager lists* *) | { } (* *streams* *)

| *pattern* + *NUM*

| *pattern* : *pattern* (* *eager lists* *)

| *pattern* :: *pattern* (* *streams* *)

| ! *pattern* | _ | *

| (*pattern* @ *pattern*)

| < *pattern* , *pattern* > (* & : *Direct Product* *)

| (*pattern* , *pattern*) (* ⊗ : *Tensor Product* *)

| (*pattern*)

exp ::= if *exp* then *exp* else *exp* end

| let *exp* be *pattern* in *exp* end

| case *exp* of inl(*ID*) ⟹ *exp* inr(*ID*) ⟹ *exp* end

| casenat *exp* of 0 ⟹ *exp* succ(*exp*) ⟹ *exp* end

| caselist *exp* of nil ⟹ *exp exp* : *exp* ⟹ *exp* end

| casestream *exp* of nilstream ⟹ *exp exp* :: *exp* ⟹ *exp* end

| fn *pattern* ⟹ *exp*

| iterlist(*exp*,*exp*,*exp*)

| primrec(*exp*,*exp*,*exp*)

| (*exp* , *exp*) | < *exp* , *exp* >

| *exp binop exp*

| *unop exp*

| *exp exp*

| [*exp_list*] | { *exp_list* }

| *ID* | *NUM* | true | false | *

| (*exp*)

exp_list ::= *exp* , ... , *exp*

binop ::= + | − | mul | div | mod | = | < | : | ::

unop ::= inl | inr | − | ! | not

A few notes on the syntax: we write $\{T\}^*$ for zero or more occurrences of T and alternatives are separated by a '|'. Reserved words are indicated in boldface. NUM corresponds to the Natural Numbers, and ID the Identifiers.

Fig. 5. Lilac abstract syntax.

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

calculus has been included into the language, this allows pattern matching on the right hand side of an equation. To allow a more concise way of expressing many linear constraints together, we allow nesting of the patterns. For example we can write $f(!x@\_) = x$ for

$$f = \lambda x.\texttt{let } x \texttt{ be } y@z \texttt{ in let } z \texttt{ be } \_ \texttt{ in let } y \texttt{ be } !z \texttt{ in } z$$

We also give the syntax of the types *generated* by $\mathscr{L}$ for Lilac; see Fig. 6. We will show in our examples below the types generated for the programs, but note that our language does not allow type declarations.

t	::=	$t \otimes t'$	*basetype*	::=	**I**
	|	$t \multimap t'$		|	`nat`
	|	$t \,\&\, t'$		|	`bool`
	|	$t \oplus t'$		|	`list`(t)
	|	$!t$		|	`stream`(t)
	|	*basetype*			

Fig. 6. Syntax of Lilac types *generated* by $\mathscr{L}$.

Since Lilac is just a sugared version of the enriched calculus we need not say very much about the translation between the two formalisms. We just state that standard techniques have been used for pattern matching etc. (see Field & Harrison, 1988, for example).

4.4 Pragmatics: Using Lilac

This section gives examples of linear functional programs that can be written using Lilac. We begin with some very basic functions which are shown together with the types generated by $\mathscr{L}$:

```
funrec zip (x : xs, y : ys) = let zip be !zp in (x, y) : zp(xs, ys) end
       zip ([ ], [ ]) = let zip be _ in [ ] end ;
       zip : !(list(α), list(β) ⊸ list(α, β))
funrec length (_ : t) = let length be !len in 1 + len(t) end
       length [ ] = let length be _ in 0 end ;
       length : !(list(!α) ⊸ nat)
funrec sum (h : t) = let sum be !s in h + s(t) end
       sum [ ] = let sum be _ in 0 end ;
       sum : !(list(nat) ⊸ nat)
```

Note that the types of *sum* and *length* are different with respect to resources. *length* discards the elements, whereas *sum* uses each element exactly once.

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

Suppose we wanted to write a function to compute the average of the elements of a list. A naive algorithm would be *average* l = *sum* l div *length* l which, first uses two copies of the list, then each copy is used in different ways.

In Lilac if we were to write this naive version then the type of the list that this function would require would be !(`list`(!`nat`)). An example application would be *average* ![!1, !2, !3], which is far from a useful notation.

However, no self-respecting functional programmer would write *average* in this way–a much more worthy solution would traverse the list only once and accumulate the *sum* and *length*. Indeed this solution follows from the above definitions of *sum* and *length* using the unfold/fold program transformation methodology of Burstall & Darlington (1977). We can write the following version of *average* in Lilac:

fun *average* l = let *av* be !f in let fl be (u, v) in u div v end end
funrec *av* [ ] = let *av* be _ in $(0, 0)$ end
av $(h : t)$ = let *av* be !f in let ft be (u, v) in $(h + u, 1 + v)$ end end

which gives *average* a more linear type `list`(`nat`) ⊸ `nat` and can be applied as: *average* $[1, 2, 3]$, which is much more acceptable.

The above is an example of a phenomena that, through practical experience with an implementation, is indicating that there is an evident connection between just how linear a function is and how efficient the algorithm is. This really should come as no surprise–if a function can be written that traverses a data structure only once then we would expect it to be more efficient.

We feel that there is scope for more work here in using the types generated from a program to indicate just how efficient the algorithm may be. One could use the typing algorithm to guide the programmer in using program transformation techniques to achieve more efficient algorithms, i.e. the typing information may help to provide heuristics.

An script showing more examples of Lilac functions is given below. The functions are given with their types as generated by $\mathscr{L}$.

Examples

fun *square* $(!x@!y) = x * y$;
  square : !`nat` ⊸ `nat`
fun $S\ f\ g\ (y@z) = fy(gz)$;
  $S : (!\alpha \multimap \beta \multimap \gamma) \multimap (!\alpha \multimap \beta) \multimap !\alpha \multimap \gamma$
fun $K\ x\ \_ = x$;
  $K : \alpha \multimap !\beta \multimap \alpha$
fun *fst* $< x, \_ >= x$;
  fst : $\alpha \,\&\, \beta \multimap \alpha$
fun *not* true = false
  not false = true ;
  not : `bool` ⊸ `bool`

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

```
funrec ones = 1 :: let ones be !x in x end ;
       ones : stream(nat)
fun assoc (x,(y,z)) = ((x,y),z) ;
    assoc : α ⊗ (β ⊗ γ) ⊸ (α ⊗ β) ⊗ γ
fun insr a = (a,*) ;
    insr : A ⊸ A ⊗ I
fun exch (a,b) = (b,a) ;
    exch : A ⊗ B ⊸ B ⊗ A
```

It is interesting to note that the last three functions give rise to the basic ingredients of a Symmetric Monoidal Category. Hence we can use the $(\mathbf{I}, \otimes, \multimap)$ fragment of Lilac as an *internal language* for such categories. This has been studied in depth in Mackie *et al.* (1993).

Problems with linear programming

We close this section by mentioning some of the fundamental problems that have arisen by practical experience of using Lilac. We feel that these comments are valid for linear functional programming languages in general:

- The most fundamental problem is that of explicitness. Our algorithms become hidden in a wealth of Contractions, Weakenings and Derelictions. There is no harm in the programmer being aware of the linearity aspect, but one would like to think the problem on two levels–the algorithm *first*, then the resource constraints. Our present system blends these two distinct issues into one which gives rise to problems such as debugging–one has to debug not only the algorithm, but also the resource constraints.
- Composition of programs becomes more subtle. Typing is a theory of *plugging compatibility*–within a linear framework this extends to a *resource compatibility*. For two linear functions to compose, they must additionally have the same resource requirements. To illustrate the point, consider the following example:

  $$square(!x@!y) = x * y \ : !\mathtt{nat} \multimap \mathtt{nat}$$

  The composition *square* ∘ *square* is *not* valid since the output of the first *square* can only be used linearly. We are obliged to use promotion: *square* ∘ ! *square* (*cf.* composition in the co-Kleisli category). This generally does not cause any problems, but it does show up most acutely in recursion where one generally has to promote the argument in the recursive call.

5 Extensions and conclusions

Our implementation is really a testbed–to get some experience in programming in such a formalism. The current implementation has several shortcomings and we envisage it being extended substantially. Some possible extensions are:

https://doi.org/10.1017/S0956796800001131 Published online by Cambridge University Press

- Polymorphism. The omission of a polylet construct, as discussed in Section 3, is one of the main technical problems that we have failed to provide a suitable answer. An *ad hoc* solution has been implemented for Lilac, but a more solid foundation needs to be formulated.
- User defined data-types. One of our main conclusions with experimenting with this kind of formalism is that the modality of linear logic can give great power to the programmer to express control over data-structures. Both eager and lazy data-structures are available within a unified framework. We are confident that the extension of Lilac for user defined data-types will make this into a useful programming paradigm.
- Optimisation. The SECD machine implementation is not so efficient—we used it just to get a prototype implementation working; it was already defined for the linear term calculus by Abramsky (1993) and did not need too much work to extend it to our enriched linear term calculus. For a more practical implementation we suggest the a graph reduction style (Peyton Jones, 1987) as a promising alternative, particularly since there are now strong connections with Linear Logic and optimal reduction strategies (Gonthier *et al.*, 1992).

Practical experience of using an implementation has left us with some strong conclusions. A programming language where we have explicit annotations of resource management is beneficial to the programmer in the sense that it gives a good indication of what work is required during execution. However we join the views of Holmström (1988) and Wakeling (1990) in that these annotations obstruct the algorithm–the most important part–which is not a desired property of any programming language. One could then regard Lilac, in some sense, as a failure. However we feel that this is a step in the direction of finding a programming paradigm where the programmer does not need to be aware of the linearity constraints, but we can still reap the benefits of having the linearity information available. To be more precise, a language based on a logic that sits between Intuitionistic and Linear Logic. One such candidate is the current work of Wadler (1993).

We suggest some variations on our theme to counter these problems:

- A language that is not so pedantic about resources. We see the principle advantages of this formalism as providing both eager and lazy data-types within one framework. A language which provides these facilities without the overhead of all the explicitness of Contraction, Weakening and Dereliction would give the gains of our system without the disadvantages. The kind of language we suggest here is one where we annotate functions and data types with control and resource information, but in a more *user friendly* way. To some extent these languages already exist, for example in the early versions of Miranda the programmer could annotate the function to indicate the method of evaluation: eager or lazy. We propose to investigate a similar language, but with a more solid logical foundation. Recent work reported by Wadler (1993) and Boudol (1993) seem to offer promising hope.
- A two-level system. One can see a program as a specification of two distinct entities: the algorithm and the resource management. A framework which

separates these two independent aspects seems as promising compromise. It is, however, unclear for the moment how the pragmatics of such a language would work out; we leave this for further investigation.

- !-free functional programming. When programming with recursive data types and iterators we can eliminate a lot of the problems with the resource constraints since we can copy and project for free; as shown in section 4. This opens up a very exciting area of *modal free linear functional programming*. It would be interesting to see just how far we can get using this formalism, and classify just how many functions we can define. Again we leave this for further investigation.

As a final remark we suggest that such a formalism is an ideal intermediate language rather than a programming language. What now needs to be developed is a translation from the λ-calculus to the linear term calculus which captures as much *linearity* as possible. Such a translation would essentially be performing some form of abstract interpretation. This has been studied by Bierman (1992) and Roversi (1992).

Acknowledgements

This paper is a cut-down and refinement of my MSc Thesis (Mackie, 1991) written under the supervision of Samson Abramsky. My thanks to all the people who helped me during that time–in particular Samson Abramsky, Mark Dawson and Paul Taylor. My thanks also to Chris Hankin, Radha Jagadeesan, David Sands and Hilfred Chau whose help in the preparation of this paper is very much appreciated. I am also grateful for the comments of the referees and Philip Wadler. The name Lilac (LInear LAmbda Calculus) is due to David Sands.

Research supported by a UK SERC studentship and ESPRIT Basic Research Action 6454 'CONFER'.

References

Abadi, M., Cardelli, L., Curien, P.-L. & Lévy, J.-J. (1991) Explicit substitutions. *Journal of Functional Programming* **1**(4), 375–416.

Abramsky, S. (1993) Computational Interpretation of Linear Logic. *Theoretical Computer Science* **111**, 3–57. (Earlier version appeared as Imperial College Technical Report DOC 90/20, October 1990.)

Abramsky, S. & Hankin, C. (Eds.) (1987) *Abstract Interpretation for Declarative Languages*. Ellis Horwood.

Benton, N., Bierman, G., de Paiva, V. & Hyland, M. (1992) Term assignment for intuitionistic linear logic. Technical Report No. 262, University of Cambridge.

Bierman, G. (1992) Type systems, linearity and functional programming. Slides from talk given at CLICS Workshop Aarhus University, Denmark.

Boudol, G. (1993) The lambda-calculus with multiplicities. Preliminary report, INRIA-Sophia Antipolis.

Burstall, R. & Darlington, J. (1977) A transformation system for developing recursive programs. *Journal ACM* **24**(1), 44–67.

Chirimar, J., Gunter, C. & Rieke, J. (1991), Proving memory management invariants for a language based on linear logic, Technical report, University of Pennsylvania, Department of Computer and Information Science.

Damas, L. (1985) Type Assignment in Programming Languages. PhD thesis, University of Edinburgh.

Damas, L. & Milner, R. (1982) Principal type schemes for functional programs. In: *Conference Record of the Ninth Annual ACM Symposium on the Principles of Programming Languages*, pp. 207–212.

Field, A. & Harrison, P. (1988) *Functional Programming*. Addison Wesley.

Girard, J.-Y. (1987) Linear Logic. *Theoretical Computer Science* **50**(1), 1–102.

Girard, J.-Y. (1989) Towards a geometry of interaction. In: J. Gray & A. Scedrov (Eds.), *Categories in Computer Science and Logic: Vol. 92 of Contemporary Mathematics*, American Mathematical Society, pp. 69–108.

Girard, J.-Y., Lafont, Y. & Taylor, P. (1989) *Proofs and Types*. Vol. 7 of *Cambridge Tracts in Theoretical Computer Science*, Cambridge University Press.

Gonthier, G., Abadi, M. & Lévy, J.-J. (1992) The geometry of optimal lambda reduction. In: *Proceedings of ACM Symposium Principles of Programming Languages*, pp. 15–26.

Henderson, P. (1980) *Functional Programming: Applications and Implementation*. Prentice Hall.

Holmström, S. (1988) Linear functional programming. In: T. Johnsson, S. Peyton Jones & K. Karlsson (Eds.), *Proceedings of the Workshop on Implementation of Lazy Functional Languages*, pp. 13–32.

Lafont, Y. (1988a) Introduction to linear logic. Lecture Notes for the Summerschool on Constructive Logics and Category Theory, Isle of Thorns.

Lafont, Y. (1988b) Logiques, Catégories and Machines. PhD thesis, Université de Paris 7.

Lincoln, P. (1992) Linear logic. *ACM SIGACT Notices* **23**(2) 29–37.

Lincoln, P. & Mitchell, J. (1992) Operational aspects of linear lambda calculus. In: *Proceedings 7th IEEE Symposium Logic in Computer Science*, Santa Cruz, CA.

Mackie, I. (1991) Lilac: A functional programming language based on linear logic. Master's thesis, Imperial College of Science, Technology and Medicine, University of London.

Mackie, I., Román, L. & Abramsky, S. (1993) An internal language for autonomous categories. *Journal of Applied Categorical Structures* Vol. 1 pp. 311–343.

Milner, R. (1978) A theory of type polymorphism in programming. *Journal of Computer and System Sciences*.

Milner, R., Tofte, M. & Harper, R. (1990) *The Definition of Standard ML*. MIT Press.

O'Hearn, P. W. (1991) Linear logic and inference control (preliminary report). In: *Proceedings 4th Category Theory and Computer Science conference*, pp. 74–93.

Peyton Jones, S. (1987) *The Implementation of Functional Programming Languages*. Prentice-Hall.

Plotkin, G. (1981) A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University.

Robinson, J. A. (1965) A machine-oriented logic based on the resolution principle. *Journal ACM* **12**(1) 23–41.

Roversi, L. (1992) A compiler from curry-typed-λ-terms to the linear-λ-terms. In: *Proceedings of the IV Convegno italiano di informatica teorica*, World Scientific (to appear).

Scedrov, A. (1993) A brief guide to linear logic. In: G. Rozenberg & A. Salomaa (Eds), *Current Trends in Theoretical Computer Science*. World Scientific.

Turner, R. (1991) *Constructive foundations for Functional Programming*. McGraw Hill.

Wadler, P. (1990) Linear types can change the world! In: M. Broy & C. Jones (Eds.), *Programming Concepts and Methods*. IFIP TC2 Working Conference on Programming Concepts and Methods, North-Holland, Sea of Galilee, Israel.

Wadler, P. (1991) Is there a use for linear logic? In: *Conference on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*. ACM Press, New Haven, CT.

Wadler, P. (1992) There's no substitute for linear logic. Presented at *Workshop on Mathematical Foundations of Programming Language Semantics.*
Wadler, P. (1993) A taste of linear logic. *Mathematical Foundations of Computer Science,* Gdansk. Springer-Verlag.
Wakeling, D. (1990) Linearity and Laziness. PhD thesis, University of York.