Director Strings as Combinators

RICHARD KENNAWAY and RONAN SLEEP
University of East Anglia

A simple calculus (the Director String Calculus—DSC) for expressing abstractions is introduced, which captures the essence of the "long reach" combinators introduced by Turner. We present abstraction rules that preserve the *applicative structure* of the original lambda term, and that cannot increase the number of subterms in the translation.

A translated lambda term can be reduced according to the evaluation rules of DSC. If this terminates with a DSC normal form, this can be translated into a lambda term using rules presented below. We call this process of abstracting a lambda term, reducing to normal form in the space of DSC terms, and translating back to a lambda term an *implementation.*

We show that our implementation of the lambda calculus is correct: For lambda terms with a normal form that contains no lambdas (*ground terms*), the implementation is shown to yield a lambda calculus normal form. For lambda terms whose normal forms represent functions, it is shown that the implementation yields lambda terms that are beta-convertible in zero or more steps to the normal form of the original lambda term. In this sense, our implementation involves *weak* reduction according to Hindley et al. [9].

Categories and Subject Descriptors: D.1.1 [**Programming Techniques**]: Applicative Functional Programming; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*; D.3.2 [**Programming Languages**]: Language Classification—*applicative languages*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages

General Terms: Languages, Theory

Additional Key Words and Phrases: Combinators, eta optimization, implementation of functional languages, lambda calculus, weak reduction

1. INTRODUCTION

For a number of reasons, ranging from use in specification and software prototyping to exploitation of new ideas for parallel architectures, increasing interest has been shown in languages whose semantics are based on the lambda calculus. Examples include Iswim, LISP, KRC, HOPE, ML, and Miranda. We ignore the various deviations all practical implementations make from the lambda calculus model, and concentrate on the lambda calculus spirit of such languages.

Work by Turner and others [5, 23, 24] has shown it is possible to implement lambda languages by translating a lambda term into combinatory form,

This work was partially supported by the UK Science and Engineering Research Council as part of the Distributed Computing Systems program.

Authors' address: School of Information Systems, University of East Anglia, Norwich NR4 7TJ U.K.

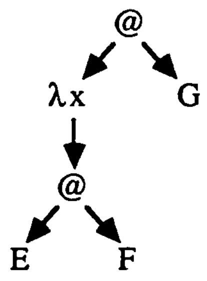

Fig. 1. $(\lambda x.EF)G$.

and reducing the result to normal form according to the rules for combinator reduction.

We consider one such scheme formally. First we present the basic insight behind the "long reach" combinators introduced by Turner [24], and discussed in a short note by Dijkstra [8]. This leads to the idea of representing abstractions by trees of application nodes in which each application is associated with a string of *directors*. Each director is associated with a level of abstraction. Director strings encode the distribution subtrees associated with substituting in an abstraction. The idea is initially expressed using informal abstraction rules, which are illustrated using familiar examples.

The remainder of the paper develops these ideas in a more formal way. Section 2 introduces the Director String calculus, shows how to move between the worlds of lambda terms and DS terms, and presents some basic results. Section 3 discusses the relation between DSC reduction and beta-reduction. Section 4 discusses evaluation strategies for DS terms. Section 5 discusses the use of the eta-rule and the removal of redundant directors. Section 6 is devoted to remarks about pragmatic issues, such as the space requirements of abstraction and the run-time consequences of abstraction. Section 7 discusses related work. The conclusion summarizes the main results and identifies issues left unresolved.

1.1 Long Reach Combinators

The practicality of Turner's implementation technique depends critically on the introduction of some "long reach" combinators described in [23]. The basic idea behind these long-reach combinators can be explained in terms of Figure 1, which represents a lambda term that is a beta redex.

The symbol @ is the usually implicit application constructor for lambda terms. By the beta conversion rule of the lambda calculus, the term illustrated converts to the term:

$$(E'F') = (EF)[x := G] = (E[x := G]F[x := G])$$

in which every free occurrence of x in both E and F is replaced with G (possibly after some alpha conversion to resolve variable clashes).

Usually we regard beta conversion as an atomic action. However, for the purpose of practical implementations we may be interested in expressing beta-conversion in terms of smaller steps. One basis for this is provided by the above diagram, in which it is clear that the substitution $[x := G]$ will have to be conveyed to both E and F, E alone, F alone, or neither depending on the free occurrences of x in E and F. This suggests it may be useful to encode abstractions such as $(\lambda x.(EF))$ in a variety of ways, depending on the pattern of free occurrences of x in E and F.

1.2 Informal Abstraction Rules

As an introduction to the more detailed technical presentation, we develop a set of informal abstraction rules by considering the three forms of a lambda term E in Λ, the set of lambda terms:

$$\begin{array}{rll} E ::= & (EE) & \text{an } \textit{application} \\ & \lambda v.E & \text{an } \textit{abstraction} \\ & v & \text{a } \textit{variable} \\ v \in & \text{VARS} & \end{array}$$

We might add some collection of atoms, with rewrite rules, to provide built-in arithmetic, for example, but for our present purposes there is no great interest in doing so. We shall use pure lambda calculus throughout this paper.

To avoid the technical complications associated with variable name clashes, we adopt the usual variable convention, that all lambda abstractions in a lambda expression bind distinct variables, which are also distinct from the free variables of the expression. Alpha conversions are assumed to be performed implicitly to ensure that this condition is maintained throughout. We may omit redundant parentheses, assuming left-associative application.

To abstract a variable from an application we have the following four rules.

$$\begin{array}{ll} \lambda x.(E_x F_x) & \rightarrow \wedge((\lambda x.E_x)(\lambda x.F_x)) \\ \lambda x.(E_x F) & \rightarrow /((\lambda x.E_x)F) \\ \lambda x.(E F_x) & \rightarrow \backslash(E(\lambda x.F_x)) \\ \lambda x.(EF) & \rightarrow -(EF) \end{array}$$

These express the notion of "moving abstraction through application," leaving a *directing symbol* behind and transmitting the abstraction process to the rator/rand as appropriate. Subscripts encode the freeness or otherwise of the variable x in a term. For example, the third rule can be read as "the abstraction of x from a (rator rand) combination in which x occurs free in the rand but not in the rator can be encoded as a *send to the right* director symbol, followed by the application of the rator to an encoding of the abstraction of the rand with respect to x." Note that similar rules may be devised for any binary operator which might be added to the lambda calculus. The basic insight can be generalized to n-ary operators, by using bit strings to indicate the various directing patterns.

For an abstraction (the case $\lambda x.\lambda y.E$) there is no rule. This ensures abstraction takes place innermost out. First y will be abstracted from E, then x will be abstracted from the result. Such multiple abstractions lead to strings of directors, called *director strings*, appearing at the nodes of the syntax tree. In the completed translation, the number of directors at a given node will be equal to the number of variables free in the subtree rooted at that node and bound by lambda-abstractions above that node.

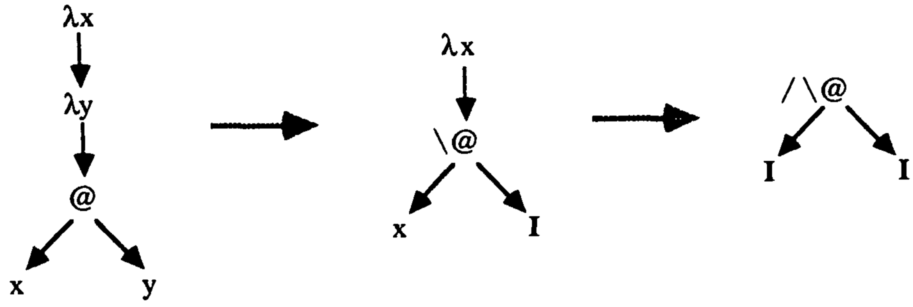

Fig. 2. Translation of $\lambda x.\lambda y.xy$.

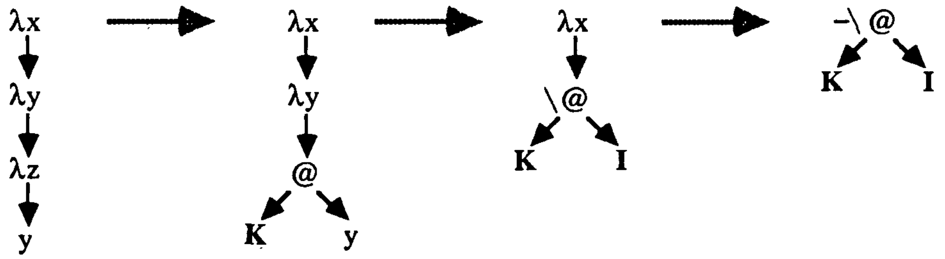

Fig. 3. Translation of $\lambda x.\lambda y.\lambda z.y$.

For a variable, there are two cases:

$$\lambda x.x \rightarrow I$$

$$\lambda x.y \rightarrow (Ky) \qquad \textit{(for any variable } y \textit{ other than } x\textit{)}$$

These rules use the usual I and K combinators to express variable abstraction. The formal presentation of the DS calculus will introduce new notations that remove the need for I and K.

The *director symbols* {^, /, \, −} have the natural interpretations *send both ways, send to the left* (i.e., the rator) *only, send to the right* (i.e., the rand) *only,* and *send neither way,* respectively.

1.3 Examples

The abstraction process outlined above can be depicted as a series of tree transformations in which each step removes one lambda, and all occurrences of its bound variable.

(i) $\lambda x.\lambda y.xy$. Each of the steps depicted in Figure 2 represents more than one application of the informal abstraction rules. The lack of a rule for the case $\lambda x.\lambda y.E$ prevents the outermost abstraction overtaking the inner one, thus enforcing innermost abstraction.

(ii) $\lambda x.\lambda y.\lambda z.y$. (See Figure 3.)

(iii) $\lambda f.\lambda x.f(fx)$—the "twice" function. (See Figure 4.)

(iv) $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$—Curry's fixed-point combinator. (See Figure 5).

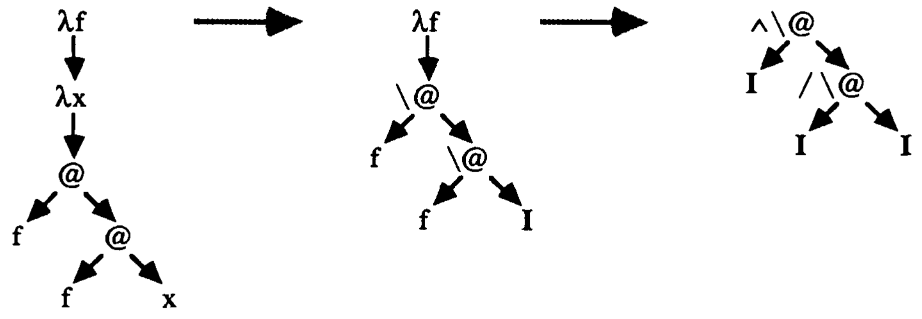

Fig. 4. Translation of $\lambda f.\lambda x.f\,(fx)$.

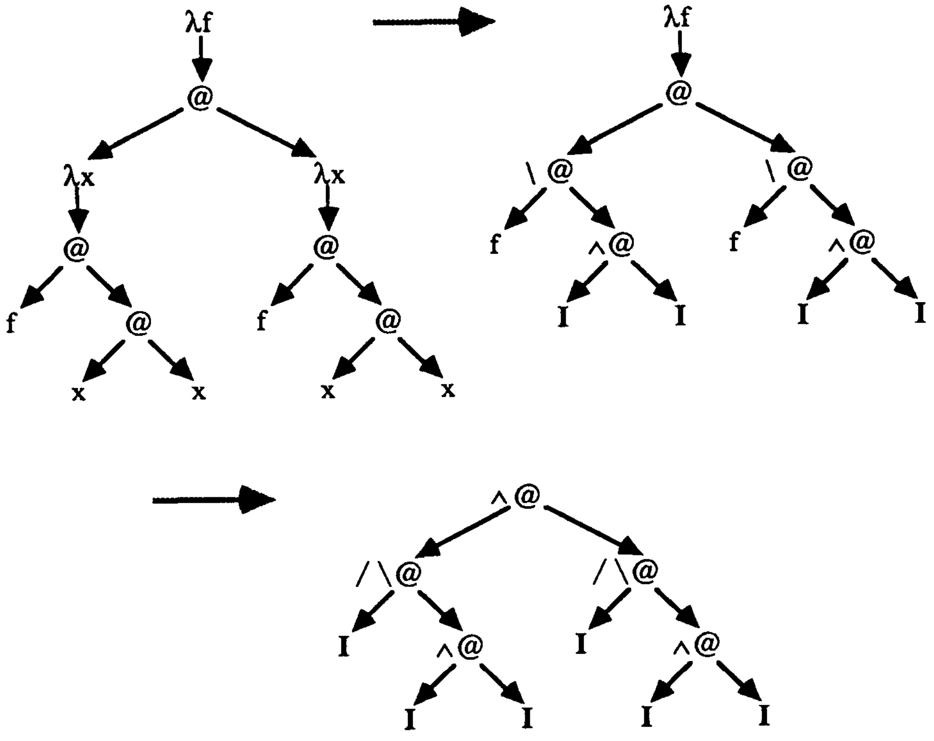

Fig. 5. Translation of $\lambda f.(\lambda x.f\,(xx))(\lambda x.f\,(xx))$.

From these examples we see that a tree representation of a lambda expression may be encoded as a tree in which all the abstraction operators and associated variables are removed, and we are left with a tree of application nodes annotated with director strings, with leaves K or I. Note that unlike the other examples, (ii) required the introduction of a new @ node. This does not happen with the full notation introduced in the next section, which preserves the "applicative structure" exactly.

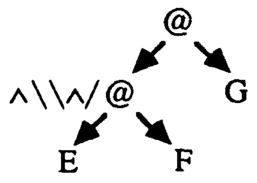

Fig. 6. A director string expression.

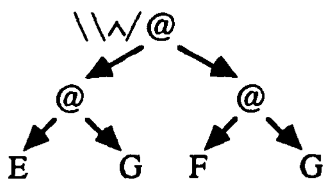

Fig. 7. The result of reducing Figure 6.

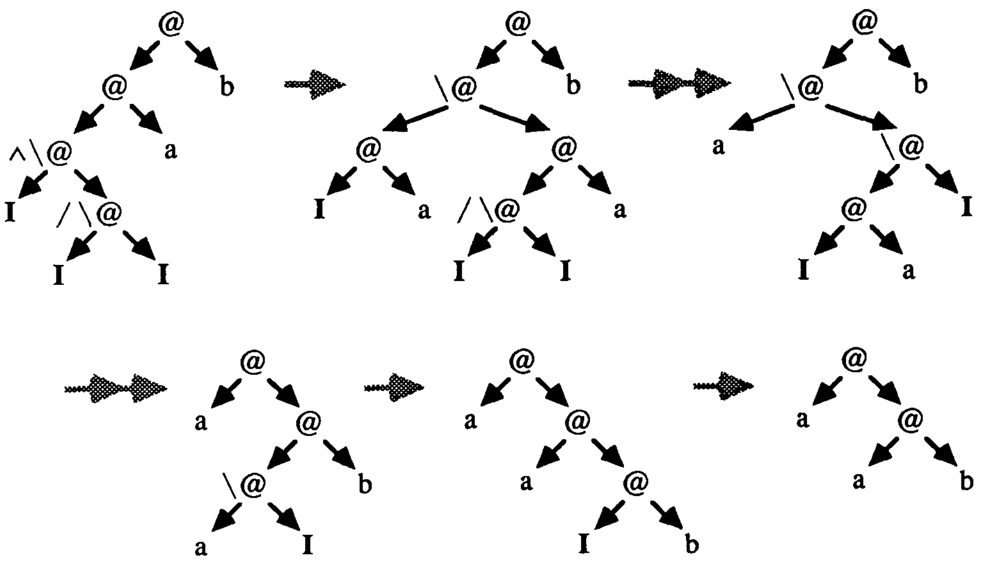

Fig. 8. Computation of $(\lambda f.\lambda x.f\,(fx))ab$.

1.4 Computation with Directors

To evaluate an expression such as that shown in Figure 6 we look at the leading director of the string. Since it is ^, the argument G must be sent to both E and F. The remainder of the director string appears at the top of the resulting expression, shown in Figure 7.

Figure 8 shows an example of a simple computation: the expression $(\lambda f.\lambda x.f\,(fx))$ ab. The lambda expression $\lambda f.\lambda x.f\,(fx)$ was translated in Example 1.3 (iii) above.

2. THE DIRECTOR STRING CALCULUS

The director string calculus, or DSC, is a term rewrite system formalizing the constructions above. The terms of this calculus (*DS terms*) are constructed from the set of symbols $\{\wedge, /, \backslash, -, !, \#, \Delta\}$, together with variables, and pair and triple constructors. The first six of these symbols are called *directors*, of which the first four are binary and the last two are unary. The symbol Δ is a *hole*, indicating a place in an expression. Unary directors are used to express abstractions such as $\lambda x.\lambda y.\lambda z.y$. Informally, "#" means *discard*, and "!" means *insert here*. Unary director strings that arise from translating lambda terms contain at most one "!" symbol.

As there will never be any confusion between unary and binary directors, we could reduce the number of different directors by overloading "$\wedge$" and "$-$", using them in place of "!" and "#". However, for expository purposes, it is clearer to use one symbol for one thing. Similarly, we might write the familiar symbol I instead of Δ, as Δ will behave in a manner very similar to the I combinator. But as the correspondence is not exact, we use the notation we have described.

A *director string* is a string of directors, all of the same arity. DIR is the set of all director strings, and is the union of DIR_1 (strings of unary directors) and DIR_2 (strings of binary directors). In what follows, d, e, f ... will represent individual directors, and D, D_1, D_2 ... will represent director strings. In the director string dD, d is called the *leading director*. $@_2$ and $@_1$ denote the empty binary and unary director string, respectively. Where the context makes the meaning clear, we may drop the subscript. ATOMS is the set of all variables, together with the symbol Δ.

2.1 Director String (DS) Terms

The set DST of DS terms is defined inductively as follows:

$$\begin{aligned}
&D_1 \in \mathrm{DIR}_1, && a \in \mathrm{ATOMS} \Rightarrow (D_1, a) \in \mathrm{DST}\\
&D_1 \in \mathrm{DIR}_1, && E \in \mathrm{DST} \Rightarrow (D_1, E) \in \mathrm{DST}\\
&D_2 \in \mathrm{DIR}_2, && E_1, E_2 \in \mathrm{DST} \Rightarrow (D_2, E_1, E_2) \in \mathrm{DST}
\end{aligned}$$

This is a slightly sugared six-sorted term algebra. The six sorts are unary and binary directors, unary and binary director strings, atoms, and DS terms. (,) and (,,) are used as pair- and triple-forming operators, and juxtaposition is used to prefix directors to director strings. To avoid notational clutter, we may abbreviate $(!, \Delta)$ (which in the DS calculus replaces the I combinator used previously) to $!\Delta$, and omit the $@_1$ or $@_2$ terminating a nonempty director string.

2.2 Mixed Lambda Calculus and Director String Terms

We map lambda-calculus terms into a system DSλ of mixed λ-calculus and director string terms, by replacing every occurrence of a variable v by $(@_1, v)$ and every application $(E\ F)$ by $(@_2, E, F)$. This embeds the λ-calculus into a system DSλ generated by the syntactic rules:

$$\begin{gathered}
E \in \mathrm{DST} \Rightarrow E \in \mathrm{DS\lambda T}\\
x \in \mathrm{VAR},\ E \in \mathrm{DST} \Rightarrow \lambda x.E \in \mathrm{DS\lambda T}
\end{gathered}$$

Thus $\lambda f.\lambda x.(f\,(xx))$ converts to the mixed term:

$$\lambda f.\lambda x.(@_2,\ (@_1,\ f),\ (@_2,\ (@_1,\ x),\ (@_1,\ x)))$$

We will identify lambda terms with their DSλ images from now on, although we may still write the conventional textual representation.

We now define a translation from lambda terms in DSλT to DST, which removes all lambdas and bound variables. The abstraction rules are expressed using the notational convention that x occurs free in E_x but not in E. D_1 and D_2 range over unary and binary director strings, respectively.

$$\begin{aligned}
\lambda x.(D_1,\ x) &\to (!D_1,\ \Delta)\\
\lambda x.(D_1,\ E) &\to (\#D_1,\ E)\\
\lambda x.(D_2,\ E_x,\ F_x) &\to ({\wedge}D_2,\ (\lambda x.E_x),\ (\lambda x.F_x)) \qquad (*)\\
\lambda x.(D_2,\ E_x,\ F) &\to (/D_2,\ (\lambda x.E_x),\ F)\\
\lambda x.(D_2,\ E,\ F_x) &\to (\backslash D_2,\ E,\ (\lambda x.F_x))\\
\lambda x.(D_2,\ E,\ F) &\to (-D_2,\ E,\ F)
\end{aligned}$$

The rule $(*)$ presents an opportunity for optimizing the translation that will be discussed in Section 5.1.

Remark. If the first rule is replaced by the rules:

$$\begin{aligned}
\lambda x.(D_1,\ E_x) &\to (!D_1,\ (\lambda x.E_x)) \qquad (\text{if } E_x \neq x)\\
\lambda x.(@_1,\ x) &\to (!,\ \Delta)
\end{aligned}$$

we obtain a translation from the whole of DSλT onto DST. However this is not needed for dealing with lambda terms, in which unary directors are initially only attached to variables (a property that remains true throughout the translation process).

Examples. The λ-expression $\lambda f.\lambda x.f\,(fx)$ (example (iii) of Section 1.3) is mapped into DSλT as $\lambda f.\lambda x.\ (@_2,\ (@_1,\ f),\ (@_2,\ (@_1,\ f),\ (@_1,\ x)))$. The translation of this term into DST yields the following reduction sequence:

$$\begin{aligned}
&\lambda f.\lambda x.(@_2,\ (@_1,\ f),\ (@_2,\ (@_1,\ f),\ (@_1,\ x)))\\
&\quad\to \lambda f.(\backslash,\ (@_1,\ f),\ \lambda x.(@_2,\ (@_1,\ f),\ (@_1,\ x)))\\
&\quad\to \lambda f.(\backslash,\ (@_1,\ f),\ (\backslash,\ (@_1,\ f),\ \lambda x.(@_1,\ x)))\\
&\quad\to \lambda f.(\backslash,\ (@_1,\ f),\ (\backslash,\ (@_1,\ f),\ !\Delta))\\
&\quad\to ({\wedge}\ \backslash,\ \lambda f.(@_1,\ f),\ \lambda f.(\backslash,\ (@_1,\ f),\ !\Delta))\\
&\quad\to ({\wedge}\ \backslash,\ !\Delta,\ (/\ \backslash,\ \lambda f.(@_1,\ f),\ !\Delta))\\
&\quad\to ({\wedge}\ \backslash,\ !\Delta,\ (/\ \backslash,\ !\Delta,\ !\Delta))
\end{aligned}$$

The other examples of Section 1.3 translate thus:

	Lambda term	*DS term*
(i)	$\lambda x.\lambda y.xy$	$(/\ \backslash,\ !\Delta,\ !\Delta)$
(ii)	$\lambda x.\lambda y.\lambda z.y$	$(\#!\#,\ \Delta)$
(iv)	$\lambda f.(\lambda x.f\,(xx))(\lambda x.f\,(xx))$	$({\wedge},\ (/\ \backslash,\ !\Delta,\ ({\wedge},\ !\Delta,\ !\Delta)),$ $(/\ \backslash,\ !\Delta,\ ({\wedge},\ !\Delta,\ !\Delta)))$

$!\Delta$ appears instead of **I**, and the introduction of a **K** combinator in example (ii) is avoided. In fact, the translation preserves the "applicative structure" of the original lambda term.

2.2.1 PROPOSITION. *The abstraction rules are confluent and terminating. Every lambda term has a unique normal form with respect to the rules, and this normal form is a* DS *term.*

PROOF. It is trivial to verify that the rules are *commutative*—that is, if $E \rightarrow F_1$ and $E \rightarrow F_2$ by reduction of different redexes, then there is a G such that $F_1 \twoheadrightarrow G$ and $F_2 \twoheadrightarrow G$. It immediately follows (see, e.g., [16]) that the rules are confluent. Each application of an abstraction rule replaces a lambda abstraction by a finite set of smaller lambda abstractions. Such a process must terminate. The existence of unique normal forms follows from Newman's Lemma [1].

Since every term containing a lambda abstraction contains a redex with respect to these rules (for example, any innermost abstraction must be such a redex), a normal form cannot contain any lambda abstractions. It is therefore a DS term. □

Note that we cannot simply apply the general theorem of [16] that a regular combinatory reduction system (CRS) is confluent, since the requirement that x occur free in E_x and F_x makes these rules fall outside the scope of Klop's definition of a CRS.

2.2.2 *Definition.* For a lambda expression E, let $\mathbf{F}(E)$ be the normal form of E with respect to the abstraction rules. By the preceding proposition, $\mathbf{F}(E)$ is a DS term.

For a DSλ expression E, let $\mathbf{AS}(E)$ (the applicative structure of E) be the object resulting from (i) replacing every bound variable in E by Δ, (ii) removing all directors from E, and (iii) replacing every subterm of the forms $\lambda x.F$ or $(@_1, F)$ by F.

Thus the applicative structure of the twice function (Example 1.3 (iii)) is shown in Figure 9, which is also the applicative structure of its translation into DST. This is a general fact.

2.2.3 PROPOSITION. *For a lambda expression E,* $\mathbf{AS}(\mathbf{F}(E)) = \mathbf{AS}(E)$.

PROOF. Each of the translation rules preserves **AS**, therefore, so does **F**. □

Remark. Examples of abstraction techniques that do not preserve applicative structure are lambda-lifting [13], balancing [4], and the eta-optimized translation described in Section 5. The advantages of such abstraction techniques must be judged by pragmatic measures such as speed on a particular machine using particular benchmarks. We think any practical abstraction techniques will disturb applicative structure to some extent.

However, preservation of applicative structure appears to be a useful property in reasoning about abstraction algorithms. For example, let $|E|$ be the size of a

$$\begin{tikzcd}
& @ \arrow[dl] \arrow[dr] & & \\
\Delta & & @ \arrow[dl] \arrow[dr] & \\
& \Delta & & \Delta
\end{tikzcd}$$

Fig. 9. Applicative structure of $\lambda f.\lambda x.f\,(fx)$.

mixed term E as measured by number of $@_1$'s, $@_2$'s, lambdas, and directors. Because $\mathbf{F}$ preserves applicative structure, any difference in size between $\mathbf{F}(E)$ and E must be solely due to the removal of lambdas and addition of directors. However, the addition of directors does not introduce new subterms. It follows that $\mathbf{F}(E)$ contains at most the number of subterms in E. Given details of the representations of E and its abstraction, we can begin to reason about the relation between the space required to store both representations. In particular, if we observe that the maximum length of a director string introduced by $\mathbf{F}$ must be proportional to $|E|$, we can immediately deduce a quadratic bound on the space complexity of the translation. A more detailed analysis is performed in [15].

2.3 Conversion Rules for DS Terms

Evaluation of DS terms is defined by the following rewrite rules. $E_1, E_2 \ldots$ stand for arbitrary DS terms.

$$\begin{array}{rcl}
(@_2,\ ({\wedge}D,\ E_1,\ E_2),\ E_3) & \rightarrow & (D,\ (@_2,\ E_1,\ E_3),\ (@_2,\ E_2,\ E_3)) \\
(@_2,\ (/D,\ E_1,\ E_2),\ E_3) & \rightarrow & (D,\ (@_2,\ E_1,\ E_3),\ E_2) \\
(@_2,\ (\backslash D,\ E_1,\ E_2),\ E_3) & \rightarrow & (D,\ E_1,\ (@_2,\ E_2,\ E_3)) \\
(@_2,\ (-D,\ E_1,\ E_2),\ E_3) & \rightarrow & (D,\ E_1,\ E_2) \\
(@_2,\ (!D,\ E_1),\ E_2) & \rightarrow & (D,\ (@_2,\ E_1,\ E_2)) \\
(@_2,\ (\#D,\ E_1),\ E_2) & \rightarrow & (D,\ E_1) \\
(@_2,\ \Delta,\ E_1) & \rightarrow & E_1
\end{array}$$

These constitute the conversion rules of the Director String Calculus. The empty director strings $@_2$ and $@_1$ can be viewed as "activators" for DS reduction, with the interpretation "transmit argument." For example, the LHS of the second rule can be read: "Transmit E_3 to $(/D, E_1, E_2)$." The last rule corresponds to the rule for the **I** combinator.

2.3.1 THEOREM. *DSC is confluent.*

PROOF. The rewrite rules form a regular term rewrite system. Confluence follows by a standard theorem [16]. □

2.4 Translating DS Terms to Lambda Terms.

We shall now consider how to translate DS expressions back into the lambda calculus. The obvious way is to run backwards the translation from lambda calculus to DSC. This suggests the following set of rules:

$$\begin{array}{rll}
(!D, (\lambda x.E)) & \rightarrow \lambda x.(D, E) & \\
(\#D, E) & \rightarrow \lambda x.(D, E) & (x \text{ a new variable}) \\
(\wedge D, (\lambda x.E), (\lambda y.F)) & \rightarrow \lambda z.(D, E[x := z], F[x := z]) & (z \text{ a new variable}) \\
(/D, (\lambda x.E), F) & \rightarrow \lambda x.\ (D, E, F) & \\
(\backslash D, E, (\lambda x.F)) & \rightarrow \lambda x.\ (D, E, F) & \\
(-D, E, F) & \rightarrow \lambda x.\ (D, E, F) & (x \text{ a new variable}) \\
\Delta & \rightarrow \lambda x.x & (x \text{ a new variable})
\end{array}$$

In the rules for /- and \-elimination, no variable clash can occur between x and, respectively, F or E, since the '#', '∧', '–', and 'Δ' rules always create new variables.

Remark. The operation of the backwards translation may be visualized as follows: First, Δ symbols are converted to the form $\lambda x.x$, with a distinct variable for every Δ. This introduces λ's, which enable other rules which gradually percolate their way up throughout the structure. A pair of lambdas appearing at the children of a binary director string beginning with a ∧ are merged.

2.4.1 THEOREM. *Every DS expression E has a unique normal form with respect to these rules. Denote it by* $\mathbf{B}(E)$. $\mathbf{B}(E)$ *is a lambda expression. For any DS expression containing no directors or holes,* $\mathbf{B}(E) = E$. *For any lambda expression* E, $\mathbf{B}(\mathbf{F}(E)) = E$.

PROOF. Each application of any of the rules removes one hole or one director. Therefore the rules are terminating. The rules are left-linear and unambiguous, therefore confluent. Thus every DS expression E has a unique normal form with respect to these rules.

Since every redex contains a director or a hole, a DS expression E containing neither must be a normal form, and $\mathbf{B}(E) = E$.

We prove by induction on E that for a lambda expression E, $\mathbf{B}(\mathbf{F}(E)) = E$.

Case (i). $E = \lambda x_1 \ldots \lambda x_n.x$.

$$\begin{array}{rll}
\mathbf{B}(\mathbf{F}(E)) & = \mathbf{B}((D, \Delta)) & \text{where } D = \#^{i-1}!\#^{n-i} \text{ if } x = x_i, \\
& & \text{or } \#^n \text{ if } x \text{ is not equal to any } x_i \\
& = E \quad \text{(up to alpha-equivalence)} &
\end{array}$$

Case (ii). $E = \lambda x_1 \ldots \lambda x_n.(FG)$. First consider $\mathbf{F}(E)$. By confluence of the **F**-rules, we can choose to compute $\mathbf{F}(E)$ by first reducing the **F**-redex $\lambda x_n.(FG)$. This gives an expression of the form (d_n, F', G'). d_n is "∧", "/", "\", or "–", according to whether x_n is free in both F and G, F only, G only, or neither. F' is F if x_n is not free in F, otherwise F' is $\lambda x_n.F$; and G' is defined similarly. Next we reduce the **F**-redex $\lambda x_{n-1}.(d_n, F', G')$, giving an expression $(d_{n-1}d_n, F'', G'')$. Continuing, we eliminate the whole of the abstraction string $\lambda x_1 \ldots \lambda x_n$, obtaining an expression of the form $(D, \lambda y_1 \ldots \lambda y_i.F, \lambda z_1 \ldots \lambda z_j.G)$. $y_1 \ldots y_i$ are the members of $x_1 \ldots x_n$ occurring free in F, and $z_1 \ldots z_j$ bears the same relation to G. We thus find that

$$\begin{aligned}\mathbf{F}(E) &= \mathbf{F}((D, \lambda y_1 \ldots \lambda y_i.F, \lambda z_1 \ldots \lambda z_j.G)) \\ &= (D, \mathbf{F}(\lambda y_1 \ldots \lambda y_i.F), \mathbf{F}(\lambda z_1 \ldots \lambda z_j.\ G)).\end{aligned}$$

Hence:

$$\mathbf{B}(\mathbf{F}(E)) = \mathbf{B}((D, \mathbf{F}(\lambda y_1 \ldots \lambda y_i.F), \mathbf{F}(\lambda z_1 \ldots \lambda z_j.G)).$$

By confluence of the **B**-rules, we can evaluate the right-hand side by first finding the **B**-normal forms of the subexpressions $\mathbf{F}(\lambda y_1 \ldots \lambda y_i.F)$ and $\mathbf{F}(\lambda z_1 \ldots \lambda z_j.G)$. But by induction these are respectively equal to $\lambda y_1 \ldots \lambda y_i.F$ and $\lambda z_1 \ldots \lambda z_j.G$. Thus:

$$\begin{aligned}&\mathbf{B}((D, \mathbf{F}(\lambda y_1 \ldots \lambda y_i.F), \mathbf{F}(\lambda z_1 \ldots \lambda z_j.G)) \\ &\quad = \mathbf{B}((D, \mathbf{B}(\mathbf{F}(\lambda y_1 \ldots \lambda y_i.F)), \mathbf{B}(\mathbf{F}(\lambda z_1 \ldots \lambda z_j.G))) \\ &\quad = \mathbf{B}((D, \lambda y_1 \ldots \lambda y_i.F, \lambda z_1 \ldots \lambda z_j.G))\end{aligned}$$

We can now apply the **B**-rules to eliminate D, reversing the string of **F**-reductions with which we began:

$$\mathbf{B}((D, \lambda y_1 \ldots \lambda y_i.F, \lambda z_1 \ldots \lambda z_j.G)) = \lambda x_1 \ldots \lambda x_n.(FG) = E \qquad \Box$$

The rules given for **B** handle all DS terms which can be produced by applying **F** to some lambda term, but do not suffice to handle the full generality of arbitrary DS expressions. A simple counterexample is $(\wedge, x, y)$. The rule for $\wedge$-elimination expects the two descendant terms to back-translate to lambda abstractions. To handle such situations we define an extended set of rules, on the basis of our intuition of the meaning of directors.

When an argument z is supplied to $(\wedge, x, y)$, the resulting expression $(@, (\wedge, x, y), z)$ reduces by the DSC conversion rules to $(@, (@, x, z), (@, y, z))$. Thus $(\wedge, x, y)$ may be regarded as the equivalent of $\lambda z.\ (@, (@, x, z), (@, y, z))$. Similar arguments for the other directors produce the rules below for elimination of "$\wedge$", "/", and "\", which subsume the earlier rules. In these rules, E must have no directors at its root. On the right-hand sides of the rules, x is a new variable not occurring on the left-hand side. If E is a lambda abstraction, E^x denotes its body, with x substituted for the bound variable of the abstraction; otherwise, E^x is $(@, E, x)$. Notice that in either case, $E^x[x := G]$ is beta-equivalent to (EG) (since x, being a new variable, cannot be free in E). This will be relevant when we prove properties of the translation below. F^x bears the same relation to F as E^x to E.

2.4.2 *General Rules for Translating a DS Term to a Lambda Term.*

$$\begin{aligned}(\#D, E) &\rightarrow \lambda x.(D, E) \\ (!D, E) &\rightarrow \lambda x.(D, E^x) \\ (\wedge D, E, F) &\rightarrow \lambda x.(D, E^x, F^x) \\ (/D, E, F) &\rightarrow \lambda x.(D, E^x, F) \\ (\backslash D, E, F) &\rightarrow \lambda x.(D, E, F^x) \\ (-D, E, F) &\rightarrow \lambda x.(D, E, F) \\ \Delta &\rightarrow \lambda x.x\end{aligned}$$

To make clear the use of the notation E^x, the fourth of these rules could otherwise be written as the following set:

$$\begin{aligned}
&(/D, \lambda y.E, F) && \rightarrow \lambda x.(D, E[y := x], F)\\
&(/D, (@_1, E), F) && \rightarrow \lambda x.(D, (@_2, (@_1, E), x), F)\\
&(/D, (@_2, E_1, E_2), F) && \rightarrow \lambda x.(D, (@_2, (@_2, E_1, E_2), x), F)
\end{aligned}$$

The rules are consistent with the previous set of rules that dealt only with DS terms of the form $\mathbf{F}(E)$ where E is a lambda term. If the abbreviations E^x and F^x are expanded out a regular CRS is obtained, so by a theorem of Klop [16] the existence and uniqueness of normal forms, and hence the well-definedness of **B**, still hold.

3. DSC REDUCTION COMPARED WITH LAMBDA CALCULUS REDUCTION

To relate DS reduction to beta reduction, we construct the joint system containing the DSλ terms defined in Section 2.2, and having as its reduction rules both the DS rules and beta reduction. DS rules can be directly read as applying to DSλ terms; beta reduction requires that the substitution operation be extended to DSλ, which is done in the obvious way, *viz.* $(D, E, F)[x := G] = (D, E[x := G], F[x := G])$, and similarly for unary directors. We call the combined reduction relation DSβ-reduction. Note that this defines a regular CRS.

3.1 THEOREM. *If E and F are DS-equivalent DS expressions, then* $\mathbf{B}(E)$ *and* $\mathbf{B}(F)$ *are beta-equivalent.*

PROOF. It is sufficient to prove this for the case where $E \rightarrow F$ by a single DS reduction. We prove it by means of two lemmas.

LEMMA 1. *Let E and F be DSλ terms,* $E \rightarrow F$ *by a single* β *reduction, and* $E \rightarrow E'$ *by a single* **B**-*step. Then there is an* F' *such that* $E' \rightarrow_\beta F' \leftarrow^*_{\mathbf{B}} F$. *Hence* $\mathbf{B}(E)$ *and* $\mathbf{B}(F)$ *are beta-equivalent.*

PROOF. Note that the **B**-rules are disjoint from the rule of beta reduction. Appealing to standard theory of CRSs [16], we obtain F' such that $E' \rightarrow^*_\beta F' \leftarrow^*_{\mathbf{B}} F$. Noting also that the **B** rules are right-linear (they do not duplicate any of their metavariables), we deduce that the sequence $E' \rightarrow^*_\beta F'$ has unit length.

The claim that $\mathbf{B}(E)$ and $\mathbf{B}(F)$ are beta-equivalent follows by applying the construction to each step in the **B**-sequence from E to $\mathbf{B}(E)$. We obtain an expression F'' and reductions $\mathbf{B}(E) \rightarrow_\beta F'' \leftarrow^*_{\mathbf{B}} F$. Since $\mathbf{B}(E)$ is in **B**-normal form, so is F''; hence $F'' = \mathbf{B}(F)$. □

LEMMA 2. *Let E and F be DSλ terms,* $E \rightarrow F$ *by a single DS reduction, and* $E \rightarrow E'$ *by a single* **B**-*step. Then there is an* F' *such that either* $E' \rightarrow_{DS} F' \leftarrow^*_{\mathbf{B}} F$ *or* $E' \rightarrow_\beta F' \leftarrow_\beta \leftarrow_\beta F$.

PROOF. *Case* (i). Suppose that the DS redex reduced in $E \rightarrow_{DS} F$ is E itself. The proofs for each of the different forms of DS redex are much alike: We shall prove the cases of the rules for "∧", "–", and "Δ".

"$\wedge$": Let $E = (@_2, (\wedge D, E_1, E_2), E_3)$, $F = (D, (@_2, E_1, E_3), (@_2, E_2, E_3))$. If the **B**-redex reduced in E to obtain E' is contained in E_1, E_2, or E_3, then it is disjoint from the DS redex, and we argue as for Lemma 1. For example, if the **B**-redex is in E_1 and its reduction yields E_1', then $E' = (@_2, (\wedge D, E_1', E_2), E_3)$ and $F' = (D, (@_2, E_1', E_3), (@_2, E_2, E_3))$.

Otherwise, the **B**-redex is $(\wedge D, E_1, E_2)$. For this to be a **B**-redex, E_1 and E_2 must have the respective forms $\lambda x.E_1'$ and $\lambda x.E_2'$. Then $E' = (@_2, \lambda x.(D, E_1', E_2'), E_3)$ and $F = (D, (@_2, \lambda x.E_1', E_3), (@_2, \lambda x.\ E_2', E_3))$. Taking $F' = (D, E_1'[x := E_3], E_2'[x := E_3])$, we have:

$$E' \to_\beta F' \leftarrow_\beta \leftarrow_\beta F$$

"$-$": Let $E = (@_2, (-D, E_1, E_2), E_3)$, $F = (D, E_1, E_2)$. The **B**-redex reduced in E to obtain E' must be contained in E_1, E_2, or E_3, and the result follows again from disjointness.

"Δ": Let $E = (@_2, \Delta, E_1)$, $F = E_1$. The **B**-redex reduced in E to obtain E' must be contained in E_1, and we again use disjointness.

Case (ii). Suppose that the DS redex is a proper subexpression of E. We proceed by induction on the structure of E. The possible forms of E are $\lambda x.E_1$, (D, E_1, E_2), and (D, E_1) (x and Δ are clearly impossible).

(a) $E = \lambda x.E_1$, $F = \lambda x.F_1$, E_1 DS-reduces to F_1. Since E is not a **B**-redex, $E' = \lambda x.E_1'$, where E_1 **B**-reduces to E_1'. By induction there is an F_1' such that F **B**-reduces to F_1' and E_1' and F_1' are β-equivalent. Take $F' = \lambda x.F_1'$.

(b) $E = (D, E_1, E_2)$. Let the DS-redex be within E_1. Then $E_1 \to_{\mathrm{DS}} F_1$ and $F = (D, F_1, E_2)$. If the **B**-redex reduced in E to obtain E' is within E_1, then we apply induction; if within E_2 then the result is trivial. Otherwise it is E itself, and its reduction eliminates the leading director of D. There are four cases; we prove the case of $\wedge$.

$D = \wedge D'$. E_1 and E_2 must have the respective forms $\lambda x.E'_1$ and $\lambda x.E_2'$. $E' = \lambda x.(D', E_1', E_2')$. F_1 has the form $\lambda x.F_1'$, where $E_1' \to_{\mathrm{DS}} F_1'$. Take $F' = \lambda x.(D', F_1', E_2')$, and it is clear that $F \to_{\mathbf{B}} F'$ and $E' =_\beta F'$.

(c) $E = (D, E_1)$. Similar to case (b).

We now prove the theorem. Let $E \to_{\mathrm{DS}} F$ and $E \to_{\mathbf{B}} E'$. We argue by induction on the longest **B**-reduction sequence from E to $\mathbf{B}(E)$ (which exists since **B**-reduction is strongly normalizing). Lemma 2 gives us an F' such that either $E' \to_{\mathrm{DS}} F' \leftarrow^*_{\mathbf{B}} F$ or $E' \to_\beta F' \leftarrow_\beta \leftarrow_\beta F$. In the former case, the theorem follows by induction, and in the latter case, it follows from Lemma 1. □

The converse does not hold in general. If a lambda-expression E beta-reduces to F, $\mathbf{F}(E)$ and $\mathbf{F}(F)$ may not be DS-convertible. For an example of this, take $E = \lambda x.((\lambda y.y)x)$. This beta-reduces to $F = \lambda x.x$. However, $\mathbf{F}(E) = (\backslash, !\Delta, !\Delta)$ and $\mathbf{F}(F) = !\Delta$, which are both DS normal forms. In translating E, the subexpression $((\lambda y.y)x)$ is first replaced by $(@_2, !\Delta, x)$, which is a DS-redex. But when x is abstracted from this DS-redex it is no longer a DS-redex. This is generally the case: The result of abstracting a variable x from a DS-expression is never a

DS redex, as it will have a nonempty director string at its root. In the sense of Hindley et al. [9], DSC reduction is *weak.*

3.2 THEOREM. *Let E and F be lambda expressions, and let $E \rightarrow F$ by a reduction of a beta redex R of E. If R has no free variables that are bound in E, and R is not the body of a lambda abstraction, then* $\mathbf{F}(E) \rightarrow^* \mathbf{F}(F)$ *in DSC.*

PROOF. As with the last theorem, we prove this by means of lemmas about DSλ.

LEMMA 1. *Let E and F be DSλ terms, $\rightarrow_{\mathrm{DS}} F$ by reduction of a DS redex R, and $E \rightarrow_{\mathbf{F}} E'$. Then there is an F' such that $E' \rightarrow_{\mathrm{DS}} F'$, and $F \rightarrow^*_{\mathbf{F}} F'$. Hence if E and F are DS-equivalent DSλ terms, then $\mathbf{F}(E)$ and $\mathbf{F}(F)$ are DS-equivalent DS terms.*

PROOF. The first part follows from the disjointness of DS-reduction and $\mathbf{F}$-reduction, and right-linearity of $\mathbf{F}$-reduction. The second follows by induction on the length of an $\mathbf{F}$-reduction sequence from E to $\mathbf{F}(E)$. □

LEMMA 2. *Let E and F be DSλ terms, $E \rightarrow_\beta F$ by reduction of a beta redex R satisfying the restriction, and $E \rightarrow_{\mathbf{F}} E'$. Then there exists F' such that either $E' \rightarrow_\beta F' \leftarrow^*_{\mathbf{F}} F$ or $E' \rightarrow_{DS} F' \rightarrow^*_\beta F$. In either case, the beta-reductions satisfy the restriction.*

PROOF. If R is a DS redex, then this follows from the disjointness of DS-reduction and $\mathbf{F}$-reduction. Let R be a beta redex. We proceed by induction on the structure of E.

Case (i). R is the whole of E. The residual of R by the $\mathbf{F}$-reduction will be the whole of E', which clearly satisfies the restriction.

E has the form $(\lambda x.E_1)E_2$ and $F = E_1[x := E_2]$. If the $\mathbf{F}$-redex reduced in E to obtain E' is contained in E_1 or E_2, then it is disjoint from R and the result follows. Otherwise, the $\mathbf{F}$-redex is $\lambda x.E_1$. We proceed by cases of the form of E_1. For illustration we prove the case $E_1 = (D, E_{11x}, E_{12x})$.

$$\begin{aligned}
E &= (\lambda x.(D, E_{11x}, E_{12x}))E_2\\
F &= (D, E_{11x}[x := E_2], E_{12x}[x := E_2])\\
E' &= ({}^\wedge D, \lambda x.E_{11x}, \lambda x.E_{12x})E_2\\
&\rightarrow_{\mathrm{DS}} (D, (@_2, \lambda x.E_{11x}, E_2), (@_2, \lambda x.E_{12x}, E_2))\\
&\rightarrow_\beta \rightarrow_\beta F
\end{aligned}$$

So we can take $F' = F$.

Case (ii). R is not the whole of E. We proceed by cases of the form of E. The possible forms are $\lambda x.E_1$, (D, E_1, E_2), and (D, E_1).

(a) $E = \lambda x.E_1$, $E_1 \rightarrow_\beta F_1$ by reduction of R, $F = \lambda x.F_1$, and $E \rightarrow_{\mathbf{F}} E'$. If the $\mathbf{F}$-redex is contained in E_1, then the result follows by induction. Otherwise, the $\mathbf{F}$-redex is E. It will be disjoint from R, unless R is E_1. But by hypothesis, R is not the body of a lambda abstraction, so is not E_1. The lemma follows, except possibly for the claim that the residual of R by the $\mathbf{F}$-reduction satisfies the restriction.

To prove this, we must analyze the possible forms of E_1, which are (D, E_{11}, E_{12}) or (D, E_{11}) (for any other form, either E would not be an **F**-redex, or E_1 could not contain any beta redex). There are then 12 cases, depending on the freeness of x in E_1 or E_2, whether D is unary or binary, and whether R is in E_{11} or E_{12}. As an example, we consider the cases where D is binary, and x is free in E_1 and not E_2. We have:

$$\begin{aligned} E &= \lambda x.(D, E_{11}, E_{12}) \\ F &= \lambda x.(D, F_{11}, E_{12}) \quad \text{or} \quad \lambda x.(D, E_{11}, F_{12}) \qquad \text{(depending on where } R \text{ is)} \\ E' &= (/D, \lambda x.E_{11}, E_{12}) \\ F' &= (/D, \lambda x.F_{11}, E_{12}) \quad \text{or} \quad (/D, \lambda x.E_{11}, F_{12}) \end{aligned}$$

If R is in E_{12}, then its residual in E' satisfies the restriction. If R is in E_{11}, then its residual in E' satisfies the restriction, unless $R = E_{11}$. But x is free in E_{11}, but (by the restriction on R) not free in R, so $R \neq E_{11}$.

(b) $E = (D, E_1, E_2)$. Suppose R is contained in E_1 (the case of E_2 is similar). If the **F**-redex is contained in E_1, we argue by induction; if contained in E_2, the result follows by disjointness. These are the only possibilities, since E is not an **F**-redex.

(c) $E = (D, E_1)$. E is not an **F**-redex, so R and the **F**-redex are in E_1, and the result follows by induction. □

PROOF OF THEOREM. We proceed by induction, first on the number of director strings in the rator of R (which for brevity we call the *size* of R), and then on the length of the longest **F**-reduction sequence of E to $\mathbf{F}(E)$.

By Lemma 2, we obtain F' such that either $E' \to_\beta F' \leftarrow^*_{\mathrm{F}} F$ or $E' \to_{\mathrm{DS}} F' \to^*_\beta F$. In the first case, $\mathbf{F}(E)$ and $\mathbf{F}(F)$ will be DS-equivalent iff $\mathbf{F}(E')$ and $\mathbf{F}(F')$ are. The beta redex reduced in $E' \to_\beta F'$ will have the same size as R, and the **F**-reduction from E' to $\mathbf{F}(E)$ will be one step shorter than the sequence for E. So induction applies to establish the DS-equivalence of $\mathbf{F}(E)$ and $\mathbf{F}(F)$.

In the second case, if the sequence $F' \to^*_\beta F$ is empty, then E' and F are DS-equivalent, and Lemma 1 implies the DS-equivalence of $\mathbf{F}(E)$ and $\mathbf{F}(F)$. Otherwise, the 1 or 2 beta redexes in this sequence are smaller than R (having, e.g., the form $(@_2, \lambda x.E_{11}, E_2)$, if R is $(@_2, \lambda x.(@_2, E_{11}, E_{12}), E_2)$). So induction applies, and $\mathbf{F}(E)$ and $\mathbf{F}(F)$ are DS-equivalent. □

3.3 THEOREM. *If the DS expression E has a normal form containing no directors and no holes, then it is also the lambda calculus normal form of* $\mathbf{B}(E)$. *If the normal form of a lambda expression E contains no lambdas, then it is also the DSC normal form of* $\mathbf{F}(E)$.

PROOF. Let E be a DS expression with DS normal form F containing no directors and no holes. Then $F = \mathbf{B}(F)$. $\mathbf{B}(F)$ contains no lambdas, and is therefore a lambda calculus normal form, which, by the preceding theorem, is β-equivalent to $\mathbf{B}(E)$.

Let E be a lambda expression with normal form F containing no lambdas. By a standard result (see [1]), E reduces to F by a sequence of leftmost-outermost beta reductions. Since the normal form of F contains no lambdas, the leftmost-outermost redex of E cannot be in the body of any lambda abstraction, so it satisfies the restriction of Theorem 3.2. Therefore $\mathbf{F}(E)$ reduces to $\mathbf{F}(F)$ by DS reduction. As F contains no lambdas, $F = \mathbf{F}(F)$, and $\mathbf{F}(F)$ is a DS normal form. □

These theorems state the soundness and (for expressions having "ground" normal forms) completeness of the following method of evaluating lambda expressions: translate the expression into DSC, evaluate it to normal form in DSC, and translate back to lambda calculus. The result need not be a normal form, but is guaranteed to be obtainable from the original lambda expression by beta-reduction (soundness). If the original expression is of ground type (that is, having a normal form containing no lambdas), then the lambda calculus normal form is obtained (completeness for ground terms).

4. EVALUATION STRATEGIES

A *strategy* for a rewrite system such as DSC is a function S that maps every term t of the system not in normal form to a nonempty set of finite nonempty reduction sequences, each beginning with t. It is *deterministic* if $S(t)$ always contains a single reduction sequence, whenever it is defined. S is a *one-step* strategy if $S(t)$ always consists only of one-step sequences.

Some well-known strategies are the following:

(i) The *leftmost-outermost* strategy is a one-step deterministic strategy that selects the leftmost-outermost redex.

(ii) The *parallel-outermost* strategy reduces all the outermost redexes of a term. This is a multistep deterministic strategy.

(iii) In a regular combinatory reduction system, a *needed* redex in a given term is a redex, some residual of which must be reduced in any reduction of that term to normal form. A formal definition of the concept is given for regular TRSs in [11] and for lambda-calculus in [2, 18]. *Needed reduction* is the strategy which at each step reduces any needed redex. This is a nondeterministic one-step strategy.

(iv) For any strategy S, *quasi-S* is the strategy

$$\text{quasi-}S(t) = \{R \cdot R' \mid \exists t'.\, R \text{ is a reduction sequence from } t \text{ to } t' \in S(t')\}$$

Thus quasi-S is S "diluted" by extra arbitrary reduction steps.

Given a strategy S and a term t_0, a reduction sequence *generated by S* from t_0 is a sequence $t_0 - R_0 \rightarrow t_1 - R_1 \rightarrow t_2 - R_2 \rightarrow \ldots$ where R_i is a member of $S(t_i)$. S is *normalizing* if every such sequence terminates with a normal form whenever t_0 has a normal form. A *substrategy* of S is a strategy S' such that for all t not in normal form, $\varnothing \subset S'(t) \subseteq S(t)$. Clearly, if S is normalizing then so is every substrategy of S. In particular, if quasi-S is normalizing, so is S.

It is well known [1] that strategies quasi-(i) and quasi-(ii) are normalizing for lambda calculus. Barendregt et al. [2] showed that quasi-(iii) is also normalizing for lambda calculus. Huet and Lévy [11] showed that quasi-(iii) is normalizing for regular TRSs, and that quasi-(ii) is a substrategy of quasi-(iii). As DSC is a regular TRS, these are normalizing strategies for DSC.

THEOREM. *Quasi-leftmost-outermost reduction is normalizing for DSC.*

PROOF. It is clear that the leftmost-outermost redex has exactly one residual by reduction of any other redex, and that residual is leftmost-outermost in the resulting term. Therefore the leftmost-outermost redex is needed, and quasi-lefmost-outermost reduction is a substrategy of quasi-needed reduction. □

5. VARIATIONS ON THE TRANSLATION

5.1 η-optimization

At the cost of losing the property of preserving applicative structure, we can make a slight optimization of the translation from lambda calculus to DSC, by taking advantage of the lambda calculus rule of η-reduction. This rule takes an expression $\lambda x.(Ex)$ to E, if x is not free in E. To accommodate this optimization, we may replace the rule (*) in Section 2.2 by the rules:

$$\begin{array}{lll} \lambda x.(D, E, F_x) & \rightarrow (\backslash D, E, (\lambda x.F_x)) & \text{if } D \neq @_2 \\ \lambda x.(@_2, E, F_x) & \rightarrow (\backslash, E, (\lambda x.F_x)) & \text{if } F_x \neq x \\ \lambda x.(@_2, E, x) & \rightarrow E & \end{array}$$

The third of these rules is the one that performs the optimization. The first and second rules are versions of the rule removed, restricted to those cases not covered by the third rule.

A further optimization can be made for the expression $\lambda x.x$. Instead of translating this to $(!, \Delta)$, we can instead use simply Δ (as may be verified by comparing the results when either of these is applied to an argument). This has the effect, compared with the former rules for **F**, that no unary director string will be generated that ends with a ! director. Such directors are redundant.

The resulting system is regular, and therefore Church Rosser. We denote this optimized translation by $\mathbf{F}_\eta$. The original **F** translation to DSC, applied to $\lambda x.(Ex)$, would give $(\backslash @_2, E, !\Delta)$, whereas $\mathbf{F}_\eta$ returns just E.

η-redexes appear infrequently in practical functional programs. However, they often arise in the course of translating to DSC. An example is the translation of $\lambda f.\lambda x.f\,(fx)$. One step of the translation gives the expression

$$\lambda f.(\backslash,(@_1, f), \lambda x.(@_2, (@_1, f), (@_1, x)))$$

which contains the η-redex $\lambda x.(@_2, (@_1, f), (@_1, x))$. The η-optimized translation then gives

$$\lambda f.(\backslash, (@_1, f), (@_1, f))$$

and the final translation is

$$(\wedge \backslash, \Delta, \Delta)$$

This is smaller than the translation given by **F**, which is $(\wedge \backslash, !\Delta, (/ \backslash, !\Delta, !\Delta))$.

An η-redex will arise whenever a lambda abstraction $\lambda x.E$ contains a subexpression of the form (Fx) with x not free in F. $\mathbf{F}_\eta(E)$ is never bigger than $\mathbf{F}(E)$, and may be significantly smaller. On average, the optimization makes a linear improvement.

5.2 Redundant Directors

Some DS expressions contain directors that are in a sense redundant. Consider a DS expression $(\wedge, (-, E, F), G)$. The '$\wedge$' director will send an incoming argument to both the subexpressions $(-, E, F)$ and G. However, the argument is then discarded on the left. Clearly it might as well not have been sent there at all, and the expression is equivalent to $(\backslash, (@, E, F), G)$. Similarly, if the director string at the root of G begins with '$-$', then an incoming argument is discarded on both sides, and might as well have been discarded immediately. Another instance of redundancy was noted in the previous section: a '!' at the end of a unary director string can be omitted. The following rules formalize this "pruning" of redundant directors. There is some trivial overlap between them.

Rule (i) Given an expression (D, E, F), let D contain at least n '$\wedge$' or '/' directors, of which the nth is d. Let E have a director string (unary or binary) at its root of length at least n, of which the nth is either '$-$' or '#'. Then the '$-$' or '#' may be deleted and d replaced by '\' (if d was '$\wedge$') or '$-$' (if d was '/').

Rule (i′) As rule (i), but considering '$\wedge$' and '\' directors in D, '$-$' and '#' directors at the root of F, and replacing d by '/' or '$-$'.

Rule (ii) Given an expression (D, E), let D contain at least n '!' directors. Let E have a director string (unary or binary) at its root of length at least n, of which the nth is either '$-$' or '#'. Then the '$-$' or '#' may be deleted and the '!' replaced by '#'.

Rule (iii) Delete any occurrence of '!' as the last director of a unary director string.

Rule (iv) Replace $(@_1, E)$ by E for nonatomic E.

Rule (v) Replace $(D, (D', E))$ by $(D \circ D', E)$, where $D \circ D'$ is defined by:

$$\begin{aligned}
@_1 \circ D' &= D' \\
D \circ @_1 &= D \\
!D \circ dD' &= d(D \circ D') \qquad (d = \text{'!' or '\#'}) \\
\#D \circ D' &= \#(D \circ D')
\end{aligned}$$

Note that for a lambda expression E, $\mathbf{F}(E)$ contains no directors subject to rules (i), (i′), or (ii). This results from the occurrence checks written into the $\mathbf{F}$ rules. If these checks were omitted, the resulting (ambiguous) translation rules would be able to yield DS expressions with redundant directors. The DS expressions obtained by this generalized translation would still be subject to the theorems concerning the correspondence between DS reduction and beta reduction. We remark (but do not prove) that for any DS expression E, $\mathbf{F}_\eta(\mathbf{B}(E))$ is a normal form of E with respect to the pruning rules. Normal forms are not unique. For example, $(!, (-, E, F))$ can be pruned to $(-, E, F)$ or to $(\#, (@_2, E, F))$.

5.3 Minimal Directors

We can define another translator of lambda calculus to DSC that does not use the '/' and '\' directors. When abstracting x from a term $(E_x F)$, the translation $\mathbf{F}$ would use a '/' director to avoid sending an incoming argument to where it is not needed. However, we could use a '^' director to send to both E_x and F. A '−' or '#' director will then be used to discard that argument from F. A similar change is made to the rule abstracting x from (EF_x). This translation uses more directors, but a smaller set of directors. Which is preferable for implementation will depend on the details of the machine.

6. COMPLEXITY OF TRANSLATION AND EXECUTION

6.1 Translation

Turner's translation of lambda calculus into combinators causes in the worst case a quadratic expansion [14]. Director strings are essentially a clearer representation of Turner's translation, and have the same complexity. Run-length encoding of the director strings [15] reduces the size of the translation to $O(n \log(k + 2))$ for lambda expressions containing n (occurrences of) atoms and k lambdas. The time taken to make the compacted translation is, however, the same as for the original translation—proportional to the size of the uncompacted translation, which is quadratic. The same reference shows that the quadratic time penalty can be avoided by use of Hughes' supercombinators [12].

6.2 Execution

Hirokawa has shown [10] that when a lambda expression is translated into Turner's combinators, the resulting expression can be evaluated in time proportional to the time required to evaluate the lambda expression by Wadsworth's method [25]. The constant of proportionality is a small constant times the maximum arity of the functions appearing in the program. Director strings are closely related to Turner's combinators, and the result applies to director strings as well. The overhead introduced by the counting representation is also linear.

This result for complexity of execution depends on executing the rules by graph reduction, not term reduction. That is, for the ^ rule:

$$(@_2, ({}^\wedge D, E_1, E_2), E_3) \rightarrow (D, (@_2, E_1, E_3), (@_2, E_2, E_3))$$

we do not make two copies of the subexpression E_3, but two pointers to the original E_3. This is a standard technique for implementing functional languages based on term rewriting [19]. Not only the space is reduced, but also the time, since a shared subexpression need only be evaluated once. As we saw in Section 4, a needed reduction strategy, such as leftmost-outermost reduction, will only reduce a subexpression if its value is needed to compute the normal form. Thus leftmost-outermost graph reduction is a *lazy* implementation of DSC. The results of Staples [21] imply that it is an optimal evaluation strategy for DSC—the reduction sequence to normal form will contain as few steps as possible.

This does not imply that lambda calculus is optimally implemented by translating to DSC and applying leftmost-outermost graph reduction. The translation removes some possibilities for sharing of computations, since, as shown in Section 3, beta redexes having free variables translate to DS terms that are not redexes. This constrains the possible orders of evaluation.

7. RELATIONS WITH OTHER WORK

7.1 Revesz's Rules for Lambda Calculus

The rules for evaluating DS expressions are related to a system of rules for lambda calculus proposed by Revesz [20]:

(rl) $(\lambda x.x)Q \rightarrow Q$

(r2) $(\lambda x.P)Q \rightarrow P$ (if x is not free in P)

(r3) $(\lambda x.\lambda y.P)Q \rightarrow \lambda z.((\lambda x.P[y := z])Q)$
(if $x \neq y$, x is free in P, and z is a new variable)

(r4) $(\lambda x.(P_1P_2))Q \rightarrow ((\lambda x.P_1)Q)((\lambda x.P_2)Q)$ (if x is free in P_1P_2)

To see the connection with director strings, we will derive a similar set of rules for evaluating lambda terms from our rules for DSC. If we take a beta-redex and translate it into DSC by **F**, the resulting DS term will be a DS redex. We can reduce this redex and translate the resulting expression back to lambda-calculus by **B**. This process is equivalent to applying the following rules directly to the lambda term. We adopt the notation of E_x and F_x as before. P and Q are ordinary metavariables.

(dl) $(\lambda x.\lambda x_1 \ldots \lambda x_n.x)Q \rightarrow Q$

(d2) $(\lambda x.\lambda x_1 \ldots \lambda_n.P)Q \rightarrow P$ (if x is not free in P)

(d3a) $(\lambda x.\lambda x_1 \ldots \lambda x_n.(E_xF_x))Q \rightarrow \lambda x_1 \ldots \lambda x_n.((\lambda x.E_x)Q)((\lambda x.F_x)Q)$ $(n \geq 0)$

(d3b) $(\lambda x.\lambda x_1 \ldots \lambda x_n.(E_xF))Q \rightarrow \lambda x_1 \ldots \lambda x_n.((\lambda x.E_x)Q)F$ $(n \geq 0)$

(d3c) $(\lambda x.\lambda x_1 \ldots \lambda x_n.(EF_x))Q \rightarrow \lambda x_1 \ldots \lambda x_n.E((\lambda x.F_x)Q)$ $(n \geq 0)$

There are two main differences between these rules and Revesz's. Firstly, rule (r4) corresponds to three rules (d3a), (d3b), and (d3c). This reflects our more detailed analysis of the occurrences of the bound variable in the rator and is not of great significance. If we based the rules on the modified translation described in Section 5.3 that does not use the '/' or '\' directors, these three rules would be a single rule corresponding more closely to rule (r4). The more important difference lies in the fact that a single application of one of Revesz's rules to a beta-redex pushes the incoming argument past the outermost node of the body of the rator. The director-based rules push the argument past an arbitrarily long string of abstraction nodes as well. Klop has noted [17] that as a result, leftmost-outermost reduction is not normalizing for Revesz's rules, due to the possibility that two incoming arguments can continually "overtake" each other without making progress.

An example is an expression of the form $(\lambda x.((\lambda y.P)Q))R$, where x is free in P and Q, and y is free in P. With Revesz's rules, this expression gives the following reduction sequence:

$$\begin{aligned}(\lambda x.((\lambda y.P)Q))R &\rightarrow ((\lambda x.\lambda y.P)R)((\lambda x.Q)R) && \text{(r4)}\\ &\rightarrow (\lambda z.((\lambda x.P[y := z])R))((\lambda x.Q)R) && \text{(r2)}\end{aligned}$$

Writing P' for $P[y:=z]$, Q' for R, and R' for $((\lambda x.Q)R)$, this is $(\lambda z.((\lambda x.P')Q'))R'$, which has the same form as the starting expression (note that z is free in P'). An infinite reduction sequence follows. To avoid the loop, one should at this point reduce not the leftmost-outermost redex (the whole expression) but the redex $(\lambda x.P[y := z])R$.

With the rules (d1)–(d3c) based on DS reduction, we obtain the following sequence. For illustrative purposes, we assume that P has the form P_1P_2, with x free in both P_1 and P_2.

$$\begin{array}{rcll} (\lambda x.((\lambda y.P_1P_2)Q))R & \rightarrow & ((\lambda x.\lambda y.P_1P_2)R)((\lambda x.Q)R) & \text{(d3a)} \\ & \rightarrow & (\lambda y.(\lambda x.P_1R)(\lambda x.P_1R))((\lambda x.Q)R) & \text{(d3a)} \end{array}$$

The second step here has performed the work of the second step in the previous reduction sequence, together with the reduction of the inner redex which we saw was necessary to make progress. In DS, we have (supposing for illustration that y is also free in both P_1 and P_2):

$$\begin{array}{l} (@_2, (\wedge, (\wedge\wedge, P_1', P_2'), Q'), R') \\ \qquad \rightarrow (@_2, (@_2, (\wedge\wedge, P_1', P_2'), R'), (@_2, Q', R')) \\ \qquad \rightarrow (\wedge, (@_2, (@_2, P_1', R'), (@_2, P_2', R')), (@_2, Q', R')) \end{array}$$

7.2 SKIM

Director strings have been implemented on the SKIM2 machine [22]. This is a successor to SKIM [5], which was a hardware implementation of Turner's combinators. SKIM2 is microcoded, and can be programmed to support Turner's combinators or director strings. Stoye [22] reports significant, although not dramatic improvements in performance using director strings.

7.3 Categorical Combinators

Categorical combinators [7] are another variable-free combinator system into which lambda calculus can be translated. They can be developed from the de Bruijn indexing technique [3], where variables in a lambda-expression are replaced by numerical indexes that identify the binding lambda. Categorical combinators are used as the machine code of the Categorical Abstract Machine, or CAM [6, 7], and a version of ML called CAML has been implemented on the CAM. The relationship between categorical combinators and director strings is not entirely clear, and we only give some brief remarks.

7.3.1 *Translation.* The translation into categorical combinators is linear. This is so regardless of whether one assumes that a de Bruijn index occupies unit or $O(\log n)$ space, since either assumption must apply equally to the variables that the indexes replace.

7.3.2 *Execution.* Several rule-systems for evaluating categorical combinator expressions have been discussed in the literature. For comparisons with director strings, we shall mention here only the "weak" rules of [6, 7].

When a closed lambda expression M has been translated to a categorical combinator expression M', M' is evaluated by applying it to the empty environment and using the weak rules. M' itself is already in normal form with respect to the weak rules; only when an environment is supplied can any evaluation be performed. The weak rules distribute the environment downwards from the root of M' to the leaves. When this distribution reaches a subexpression of M' corresponding to a beta redex of M, the rand of the redex is added to the environment, and this augmented environment is distributed over the rator. For example, the evaluation of the term $(\lambda x.(\lambda y.EF)G))H$ in the empty environment—which we denote by $[\![(\lambda x.(\lambda y.EF)G))H]\!](\,)$—proceeds thus (retaining lambda calculus notation rather than translating to combinators):

$$\begin{aligned}[\![(\lambda x.(\lambda y.EF)G))H]\!](\,) &\to [\![(\lambda y.EF)G]\!](x\colon [\![H]\!])\\ &\to [\![EF]\!](x\colon [\![H]\!],\, y\colon [\![G]\!])\\ &\to ([\![E]\!](x\colon [\![H]\!],\, y\colon [\![G]\!]))([\![F]\!](x\colon [\![H]\!],\, y\colon [\![G]\!]))\end{aligned}$$

and we see how the combined environment is distributed as a unit. This contrasts with DS reduction, which, described in terms of environments, will, in effect, first distribute the environment over the unreduced redex, and then proceed to reduce the now modified redex. Choosing some pattern of occurrence of x and y for illustration, the DS reduction might begin:

$$\begin{aligned}&(@_2,\, (\wedge,\, (/\,/,\, E',\, F'),\, G'),\, H')\\ &\quad\to (@_2,\, (@_2,\, (\wedge/,\, E',\, F'),\, H'),\, (@_2,\, G',\, H'))\\ &\quad\to (@_2,\, (/,\, (@_2,\, E',\, H'),\, F'),\, (@_2,\, G',\, H'))\\ &\quad\to (@_2,\, (@_2,\, (@_2,\, E',\, H'),\, (@_2,\, G',\, H')),\, F')\end{aligned}$$

H' and $(@_2, G', H')$ are here being distributed separately over E'. However, with director strings, the environment need not be sent where it is not needed, as with the subterm F' above.

Thus, categorical combinators have the advantage over director strings that nested beta redexes are in effect reduced simultaneously, but the disadvantage that arguments to beta redexes are distributed everywhere, not just to the places where they are needed.

A further difference is that the implementation of the weak rules in the CAM imposes a strict evaluation order, whereas machines like SKIM2 use leftmost-outermost reduction to achieve lazy evaluation. There are well-known techniques for encoding nonstrict functions in a strict language ([6] shows how it is done on the CAM), but the encoding loses laziness—the ability to evaluate an argument at most once, no matter how many times it is required, and not at all if it is not required.

A precise comparison of theoretical performance is difficult; practical tests must decide the matter.

8. SUMMARY AND CONCLUSION

The basic insight underlying the Director String Calculus is present in Turner's S', B', and C' "long reach" combinators [24]. Dijkstra [8] independently suggested the idea of annotating the application subterms with strings. We have formalized these ideas, and shown that our implementation of the lambda calculus based on DSC is correct up to beta-convertibility. We have also shown that for ground terms (those whose normal form contains no lambdas) DSC reduction is strong, i.e., the normal form obtained by the DSC implementation is the lambda calculus normal form. Many of our proofs have been simplified by designing our rule systems to be regular.

There are a number of reasons for believing that DSC reduction provides a basis for implementing lambda languages:

(a) Practical implementations (e.g., SKIM) are in use already.

(b) Although DSC reduction is weak, this is sufficient for evaluating ground terms.

(c) For nonground terms, strong (beta-) reduction can lead to a phenomenon called *code expansion*, in which the size of a function body grows considerably with little if any accompanying benefit. For example the lambda calculus normal form of the term ((twice twice) (twice twice)) **where** twice = $\lambda f.\lambda x.f\,(fx)$ contains 256 subterms, which are applications. The DSC normal form of the translation of this expression is much smaller.

In practice many optimizations of the DSC implementation can be made. During abstraction, the use of eta-optimization can lead to significantly more compact code. During reduction, advantage may be taken of large, frequently used abstraction patterns (e.g., the supercombinators of Hughes [12]) and specialized machine code may be written to achieve the same overall effect.

We have shown that our abstraction rules preserve the *applicative structure* of the original lambda term, and cannot increase the number of subterms in the translation. This may be a useful law for reasoning about DSC-like implementations.

ACKNOWLEDGMENTS

David Turner first showed that combinator implementations of lambda languages were practicable [23, 24]. Peter Buneman pointed out that DSC abstraction preserves applicative structure. Many helpful comments were made by P.-L. Curien, members of the Declarative Systems Project at UEA, and the referees. Jan Willem Klop and Henk Barendregt taught us about reduction theory and the lambda calculus. The work was partially supported by the UK Science and Engineering Research Council as part of the Distributed Computing Systems program.

REFERENCES

1. BARENDREGT, H. P. *The Lambda Calculus. Studies in Logic and the Foundations of Mathematics 103*, North-Holland, Amsterdam, 1984.
2. BARENDREGT, H. P., KENNAWAY, J. R., KLOP, J. W., AND SLEEP, M. R. Needed reduction and spine strategies for the lambda calculus. Rep. CS-R8621, Centre for Mathematics and Computer Science, Mathematical Centre, Amsterdam, 1986.
3. DE BRUIJN, N. D. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, *Indag. Math. 34* (1972), 381–392.
4. BURTON, F. W. A linear space translation of functional programs to Turner combinators. *Inf. Process Lett. 14*, 5 (1982), 201–204.
5. CLARKE, T. J. W., GLADSTONE, P. J. S., MACLEAN, C. D., AND NORMAN, A. SKIM—the S, K, I reduction machine. In *Proceedings of the LISP-80 Conference* (Stanford, Calif., 1980).

6. Cousineau, G., Curien, P.-L., and Mauny, M. The categorical abstract machine. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture. Lecture Notes in Computer Science 201.* Springer, Berlin, 1986, pp. 50–64.
7. Curien, P.-L. *Categorical Combinators, Sequential Algorithms, and Functional Programming.* Pitman, London, 1986.
8. Dijkstra, E. W. A mild variant of combinatory logic. Unpublished note. EWD735, 1980.
9. Hindley, J. R., Lercher, B., and Seldin, J. P. *Introduction to Combinatory Logic.* Cambridge University Press, London, 1972.
10. Hirokawa, S. Complexity of the combinator reduction machine. *Theor. Comput. Sci. 41,* 2–3 (1985), 289–304.
11. Huet, G., and Lévy, J. J. Call by need computations in non-ambiguous linear term rewriting systems. Rapport Laboria 359, IRIA, 1979.
12. Hughes, R. J. M. The design and implementation of programming languages. Ph.D. dissertation, Programming Research Group, Oxford Univ., 1984.
13. Johnsson, T. Lambda-lifting: Transforming programs to recursive equations. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture. Lecture Notes in Computer Science 201.* Springer, Berlin, 1986, pp. 190–203.
14. Kennaway, J. R. The complexity of a translation of lambda calculus to Turner's combinators. Rep. CSA/13/1984, School of Information Systems, Univ. of East Anglia, 1982.
15. Kennaway, J. R., and Sleep, M. R. Variable abstraction in $O(n \log n)$ space. *Inf. Process. Lett. 24* (1987), 343–349.
16. Klop, J. W. Combinatory reduction systems. Ph.D. dissertation, Mathematical Centre Tracts 127, Mathematical Centre, Amsterdam, 1980.
17. Klop, J. W. Term rewriting systems. In *Notes for the Workshop on Reduction Machines* (Ustica, 1985).
18. Lévy, J. J. Optimal reductions in the lambda-calculus. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism.* J. P. Seldin and J. R. Hindley, Eds. Academic Press, London, 1980, pp. 159–191.
19. Peyton Jones, S. L. *The Implementation of Functional Languages.* Prentice-Hall, Englewood Cliffs, N.J., 1987.
20. Revesz, G. Axioms for the theory of lambda conversion. *SIAM J. Comput. 14,* 2 (1985), 373–382.
21. Staples, J. Optimal evaluations of graph-like expressions. *Theor. Comput. Sci. 10,* (1980), 297–316.
22. Stoye, W. Director strings on SKIM. In *Proceedings of the Functional Programming Workshop* (Göteborg, 1985).
23. Turner, D. A. A new implementation technique for applicative languages. *Softw. Pract. Exper. 9* (1979), 31–49.
24. Turner, D. A. Another algorithm for bracket abstraction. *J. Symbol. Logic 44,* 2 (1979), 267–270.
25. Wadsworth, C. P. Semantics and pragmatics of the lambda calculus. Ph.D. dissertation, Programming Research Group, Oxford Univ., 1971.

Received August 1982; revised November 1986 and March 1988; accepted April 1988