Detecting Global Variables in Denotational Specifications

DAVID A. SCHMIDT
Iowa State University

Sufficient criteria are given for replacing all occurrences of the store argument in a Scott–Strachey denotational definition of a programming language by a single global variable. The criteria and transformation are useful for transforming denotational definitions into compilers and interpreters for imperative machines, for optimizing applicative programs, and for judging the suitability of semantic notations for describing imperative languages. An example transformation of a semantics of a **repeat**-loop language to one which uses a global store variable is given to illustrate the technique.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*denotational semantics*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*lambda calculus and related systems*

General Terms: Languages, Theory

Additional Key Words and Phrases: Single-threading

1. INTRODUCTION

As the descriptional methods of denotational semantics become better understood, the task of automatically converting a denotational definition of a language into a processor coded in a low level imperative language also becomes feasible. Early work in the area emphasized generality [7, 10], and the generated processors were not practical to use because they were too large and too slow. Of the restrictions suggested to make the task easier, the most promising has been the use of combinators [2] as a semantic notation. Each combinator maps to a sequence of instructions in the target language, making the task of processor generation a matter of syntax directed translation. Unfortunately, it is not clear what underlying properties the combinators should possess to make them useful for the task, and so no ideal set has been agreed upon [3, 6, 9, 11–13, 18, 25, 26].

This paper examines one critical property that any denotational definition of a sequential programming language needs to possess: the sequential processing of that language's semantic store argument. Sufficient criteria are defined for statically checking that a store argument in a denotational definition is handled

This work was supported by S. E. R. C. Grant GRB67706.
Author's current address: Department of Computer Science, Iowa State University, Ames, IA 50011.

in a fashion which allows it to be modeled as a global variable in any implementation of the definition. This property is important in producing efficient implementations, as it encourages the natural introduction of side-effect-producing commands and control structures into the target language. The stated criteria are general enough to apply to any style of denotational definition—direct, continuation, or combinator—as they are defined upon typed lambda calculus expressions, which serve as the "assembly language" for all forms of denotational notation. The method is also applicable to developing optimizing compilers for applicative languages.

2. SINGLE-THREADING

The sequential processing property to be studied is called *single-threading* [21, 23]. It is best introduced by an example. The syntax of a simple **repeat**-loop language is given in Figure 1. A denotational semantics for the language maps the syntactic structures to mathematical entities such as functions and numbers. These entities are defined in Figure 2 using semantic algebras. Each algebra describes a domain of objects and the operations which utilize the domain. Operations upon primitive domains are also taken as primitive (e.g., *plus* in the algebra of natural numbers *Nat*), but all others are defined as functions using Strachey-style lambda calculus notation (e.g., *update* in the algebra *Store*). The semantic algebras are utilized in Figure 3 to give meaning to the syntax via the valuation functions. (All abstraction forms ($\lambda x.M$) are *strict*—call by value-like—and the function domains to which they belong are *lifted* [17].)

Aside from the mathematical insight that a denotational definition provides, the main benefit of such a definition is that the semantics notation can be treated operationally as well. Figure 4 shows an operational-like interpretation of a sample program when started with an *empty* store. The valuation equations are handled as rewriting rules [5], simplifying the program plus its input store to an output store. The operations of the semantic algebras are treated as encapsulated functions, and so the steps taken in the evaluation of, say, (*access*[x] s1) to *zero*, are not explicitly shown.

The reader who has some experience with denotational semantics will recognize that the definition of Figure 3 appears reasonable for an imperative language because the semantic store argument is treated in a sequential fashion when passed as a parameter. This is confirmed in the example of Figure 4, where apparent multiple copies of stores in lines (4), (7), and (8) have the same value—a single shared copy would suffice. Further, store updating via the *update* operation only occurs in those configurations where the store is not otherwise accessed. This suggests that the individual instances of the store argument can be replaced by access rights to a single global variable, and the *update* operation can be made to cause a side effect. A semantic definition whose store argument can be replaced by access rights to a single global variable while preserving operational properties is said to be *single-threaded* (in its store). In Section 3, sufficient criteria for single-threading are introduced.

The example will be transformed to make use of a variable to represent the store arguments. First, the store semantic algebra of Figure 1 is converted to first

P: Program
C: Command
E: Expression
I: Identifier
N: Numeral
P ::= C.
C ::= I := E | C_1; C_2 | **if** E **then** C_1 **else** C_2 | **repeat** C **until** E
E ::= $E_1 + E_2$ | I | N

Fig. 1. Abstract syntax.

I. t: Tr (truth values)
   $true$: Tr
   $false$: Tr
   not: $Tr \to Tr$
II. n: Nat (natural numbers)
   err: Nat
   $zero$: Nat
   one: Nat
   . . .
   $plus$: $Nat \to Nat \to Nat$
   $equals$: $Nat \to Nat \to Tr$
III. i: Id = Identifier
   [A]: Id
   [B]: Id
   . . .
   $equalid$: $Id \to Id \to Tr$
IV. s: $Store = Id \to Nat$
   $empty$: $Store$
   $empty = \lambda i . err$
   $access$: $Id \to Store \to Nat$
   $access\ i\ s = s(i)$
   $update$: $Id \to Nat \to Store \to Store$
   $update\ i\ n\ s = \lambda i' . i'\ equalid\ i \to n \,\Box\, s(i')$

Fig. 2. Semantic algebras.

P: Program $\to$ $Store \to Store$
  P[C.] = **C**[C]
C: Command $\to$ $Store \to Store$
  C[I := E] = $\lambda s . update$[I](**E**[E]s)s
  C[C_1; C_2] = **C**[C_2] $\circ$ **C**[C_1]
  C[**if** E **then** C_1 **else** C_2] = $\lambda s.$(**E**[E]s $equals\ zero \to$ **C**[C_1] $\Box$ **C**[C_2])s
  C[**repeat** C **until** E] = f $\circ$ **C**[C]
   where $f = \lambda s.(not$ (**E**[E]s $equals\ zero) \to f \circ$ **C**[C] $\Box$ $skip)s$
E: Expression $\to$ $Store \to Nat$
  E[$E_1 + E_2$] = λs.(**E**[E_1]s) $plus$ (**E**[E_2]s)
  E[I] = $\lambda s.access$ [I]s
  E[N] = $\lambda s.$**N**[N]
N: Numeral $\to$ Nat (omitted)

Fig. 3. Valuation functions.

(1) $\mathbf{P}[\mathrm{x} := 0; \mathrm{y} := \mathrm{x} + 1.]\ \mathit{empty}$
(2) $= \mathbf{C}[\mathrm{x} := 0; \mathrm{y} := \mathrm{x} + 1]\ \mathit{empty}$
(3) $= \mathbf{C}[\mathrm{y} := \mathrm{x} + 1](\mathbf{C}[\mathrm{x} := 0]\ \mathit{empty})$
(4) $= \mathbf{C}[\mathrm{y} := \mathrm{x} + 1](\mathit{update}[\mathrm{x}](\mathbf{E}[0]\ \mathit{empty})\ \mathit{empty})$
(5) $= \cdots = \mathbf{C}[\mathrm{y} := \mathrm{x} + 1](\mathit{update}[\mathrm{x}]\ \mathit{zero}\ \mathit{empty})$
(6) $= \mathbf{C}[\mathrm{y} := \mathrm{x} + 1]\ \mathrm{s1}$, where $\mathrm{s1} = \lambda i'.\, i'\ \mathit{equalid}[\mathrm{x}] \rightarrow \mathit{zero}\ \square\ \mathit{empty}(i')$
(7) $= \mathit{update}[\mathrm{y}](\mathbf{E}[\mathrm{x} + 1]\ \mathrm{s1})\ \mathrm{s1}$
(8) $= \cdots = \mathit{update}[\mathrm{y}]((\mathit{access}[\mathrm{x}]\ \mathrm{s1})\ \mathit{plus}\ (\mathbf{E}[1]\ \mathrm{s1}))\ \mathrm{s1}$
(9) $= \cdots = \mathit{update}[\mathrm{y}](\mathit{zero}\ \mathit{plus}\ \mathit{one})\ \mathrm{s1}$
(10) $= \mathit{update}[\mathrm{y}](\mathit{one})\ \mathrm{s1}$
(11) $= \mathrm{s2}$, where $\mathrm{s2} = \lambda i'.\, i' \mathit{equalid}[\mathrm{y}] \rightarrow \mathit{one}\ \square\ \mathrm{s1}(i')$

Fig. 4. An example derivation.

```
IV.′ s: Store = (Id × Nat)*
       empty: Store
         empty = ⟨ ⟩
       access: Id → Store → Nat
         access i s = let lookup i′s′ =
                                 null s′ → err
                                □ let (j, n) = hd s′ in
                                   i′ equalid j → n
                                     □ lookup i′(tl s′)
                             in lookup i s
       update: Id → Nat → Store → Store
         update i n s = (i, n) cons s
```

Fig. 5. Result of defunctionalization.

order form by converting the functional data domain to a nonfunctional one. The step demonstrates that a store is indeed implementable as a data structure on a conventional machine. There exist many choices of first order representations of functions, and for simplicity, lists of *Nat*, *Id* ordered pairs are used. The operations in the *Store* algebra are altered to deal with the new representation. The mechanics of this step have been documented elsewhere [4, 8, 20, 24] and are not given here. Figure 5 contains the new semantic algebra for the store; all other parts of the definition remain the same.

Introduction of the global variable occurs in two steps. First, the *Store* algebra is converted to a SIMULA-like class; a single *Store*-typed variable resides within. The operations of the class have permission to access the variable directly, and in the case of the *Store*-valued operations *empty* and *update*, to alter the variable's value using a newly introduced assignment construct $\leftarrow$. This transformation means that the store is no longer needed as an explicit argument to the class' operations. Second, all explicit occurrences of *Store*-typed variables are explicitly removed from the valuation functions; even the lambda abstractions on stores disappear. The results are shown in Figure 6. Although the stores have been removed, access rights to the store variable must be remembered. One method for doing so is within the type system of the semantics notation. For example,

$$\mathbf{C}[\mathrm{I} := \mathrm{E}] = \mathit{update}[\mathrm{I}]\ \mathbf{E}[\mathrm{E}]$$

IV.'' **var** s: $Store = (Id \times Nat)^*$
  $empty = (s \leftarrow \langle\ \rangle)$
  $access\ i = (\ldots$ as in Figure 5$\ldots)$
  $update\ i\ n = (s \leftarrow (i, n)\ cons\ s)$

P: Program $\rightarrow Store \rightarrow Store$
  $\mathbf{P}[\mathrm{C}\,.\,] = \mathbf{C}[\mathrm{C}]$

C: Command $\rightarrow Store \rightarrow Store$
  $\mathbf{C}[\mathrm{I} := \mathrm{E}] = update[\mathrm{I}]\ \mathbf{E}[\mathrm{E}]$
  $\mathbf{C}[\mathrm{C}_1;\ \mathrm{C}_2] = \mathbf{C}[\mathrm{C}_2] \circ \mathbf{C}[\mathrm{C}_1]$
  $\mathbf{C}[\textbf{if } \mathrm{E} \textbf{ then } \mathrm{C}_1 \textbf{ else } \mathrm{C}_2] = \mathbf{E}[\mathrm{E}]\ equals\ zero \rightarrow \mathbf{C}[\mathrm{C}_1] \square \mathbf{C}[\mathrm{C}_2]$
  $\mathbf{C}[\textbf{repeat } \mathrm{C} \textbf{ until } \mathrm{E}] = f \circ \mathbf{C}[\mathrm{C}]$
    where $f = not(\mathbf{E}[\mathrm{E}]\ equals\ zero) \rightarrow f \circ \mathbf{C}[\mathrm{C}] \square\ skip$

E: Expression $\rightarrow Store \rightarrow Nat$
  $\mathbf{E}[\mathrm{E}_1 + \mathrm{E}_2] = \mathbf{E}[\mathrm{E}_1]\ plus\ \mathbf{E}[\mathrm{E}_2]$
  $\mathbf{E}[\mathrm{I}] = access[\mathrm{I}]$
  $\mathbf{E}[\mathrm{N}] = \mathbf{N}[\mathrm{N}]$

Fig. 6. Result of variable introduction.

$\mathbf{P}[x := 0; y := x + 1\,.\,]empty$
$= \mathbf{P}[\mathrm{x} := 0; \mathrm{y} := \mathrm{x} + 1\,.\,]$, and $s0 = \langle\ \rangle$
$= \mathbf{C}[\mathrm{x} := 0; \mathrm{y} := \mathrm{x} + 1]$
$= \mathbf{C}[\mathrm{y} := \mathrm{x} + 1](\mathbf{C}[\mathrm{x} := 0])$
$= \mathbf{C}[\mathrm{y} := \mathrm{x} + 1](update[\mathrm{x}]\ \mathbf{E}[0])$
$= \cdots = \mathbf{C}[\mathrm{y} := \mathrm{x} + 1](update[\mathrm{x}]\ zero)$
$= \mathbf{C}[\mathrm{y} := \mathrm{x} + 1]$, and $s1 = \langle([\mathrm{x}], zero)\rangle$
$= update[\mathrm{y}]\ \mathbf{E}[\mathrm{x} + 1]$
$= \cdots = update[\mathrm{y}]((access[\mathrm{x}])\ plus\ \mathbf{E}[1])$
$= \cdots = update[\mathrm{y}](zero\ plus\ one)$
$= update[y]\ one$
$= (\ )$, and $s2 = \langle([\mathrm{y}], one), ([\mathrm{x}], zero)\rangle$

Fig. 7. The evaluation revisited.

is typed as (let $S = Store$, $I = Id$, $N = Nat$):

$$\mathbf{C}[\mathrm{I} := \mathrm{E}]^{S \rightarrow S} = ((((update^{I \rightarrow N \rightarrow S \rightarrow S}[\mathrm{I}]^{I})^{N \rightarrow S \rightarrow S}(\mathbf{E}[\mathrm{E}]^{S \rightarrow N})^{N})^{S \rightarrow S})^{S})^{S \rightarrow S}.$$

Coercions such as $(\mathbf{E}[\mathrm{E}]^{S \rightarrow N})^{N}$ denote access rights to the store variable which have been granted; here, $\mathbf{E}[\mathrm{E}]$ is allowed to evaluate as if it had an explicit store parameter. An alternative is to explicitly represent the access rights by dummy placeholders ( ); the previous equation would appear as

$$\mathbf{C}[\mathrm{I} := \mathrm{E}] = \lambda(\ ).update[\mathrm{I}]\ (\mathbf{E}[\mathrm{E}](\ ))\ (\ ).$$

The two formats are equivalent. The evaluation of Figure 4 is repeated using the new notation in Figure 7.

3. CRITERIA FOR SINGLE THREADING

Criteria which are sufficient for ensuring that a global variable can be introduced into a denotational definition are now given. The proof of sufficiency of the criteria is dependent upon the use of a subtree replacement system [5, 15] as an operational semantics for the semantic notation. An operator definition $fx_1 \ldots x_n = M$ is treated as a rewrite rule (δ-rule [2]) $fx_1 \ldots x_n > M$. An instance $R = (fN_1 \ldots N_n)$ in an expression E can be rewritten to M instantiated with arguments N_i for each occurrence of x_i in M. The result is substituted into E. A fundamental rewrite rule for the lambda calculus-based semantic notation is the binding rule (β-rule) $(\lambda x.M)N > [N/x]M$: argument N is substituted for all free occurrences of identifier x in M.

There exist a number of strategies for applying the rewrite rules to an expression. The one used in this paper is a form of call by value. A subexpression is *active* if it is not properly contained within an abstraction. Second, an expression is in *restricted normal form* (rnf) if it is closed (has no unbound variables) and no active subexpression of it matches the left-hand side of a rewrite rule. A *redex* is an active, closed expression that matches the left-hand side of a rewrite rule and whose proper, active subexpressions are in rnf. The call-by-value rewrite system evaluates an expression by rewriting its redexes until no more exist. If the original expression was closed, the final result (if one exists) will be in rnf.

The rule set induced from a Strachey-style denotational definition is confluent [2, 5, 15], and the strategy of choosing closed, active redexes with rnf arguments is sufficient for attaining rnfs for closed expressions [15]. Since redexes must be closed expressions, the confluence problems that other call-by-value systems possess [16] are avoided.

Two assumptions about the denotational definitions studied are made. First, the semantic data types (domain expressions) are built only from a set of primitive types (e.g., *Tr*, *Nat*, ...) and the function space constructor $\rightarrow$. Recursive types are allowed. This keeps the single-threading criteria simpler; product spaces are added later in the paper. Second, the *Store* data type, the candidate for conversion, is first order (nonfunctional). This can be accomplished by using the transformation noted in Section 2.

The criteria for single threading may now be given. Recalling Figure 4, an occurrence of an identifier s:*Store* in an expression E is a form of access right to the store's current value; a *Store*-typed redex is an update right. No update right may be a redex in an expression that also contains access rights. Say that a *Store*-typed identifier is a *trivial* *Store*-typed expression; all other *Store*-typed expressions are *nontrivial* (*Store*-typed expressions may be nested).

Definition 2.1. A closed expression F is single-threaded (in its store argument) if all subexpressions E of F possess the properties:

A. Noninterference
   (i) If E is *Store*-typed, then if E contains multiple, disjoint active occurrences of *Store*-typed expressions, then they are the same trivial identifier.
   (ii) If E is not *Store*-typed, all occurrences of active *Store*-typed expressions in E are the same trivial identifier.

B. Immediate evaluation
   (i) If $E = (\lambda x.M)$: *Store* $\rightarrow D$, then all active *Store*-typed identifiers in M are occurrences of x.

(a) $\mathbf{C}[\mathrm{C}_1;\, \mathrm{C}_2] = \lambda s.\mathbf{C}[\mathrm{C}_2](\mathbf{C}[\mathrm{C}_1]s)$
(b) $\mathbf{E}[\mathrm{E}_1 + \mathrm{E}_2] = \lambda s.\mathbf{E}[\mathrm{E}_1]s \mathit{ plus } \mathbf{E}[\mathrm{E}_2]s$
(c) $\mathbf{P}[\textbf{procedure } \mathrm{C} \textbf{ using } \mathrm{I}] = \lambda n.\lambda s.\mathbf{C}[\mathrm{C}](\mathit{update}[\mathrm{I}]n\ s)$
where $\mathbf{P}$: Procedure $\rightarrow \mathit{Nat} \rightarrow \mathit{Store} \rightarrow \mathit{Store}$
(d) $\mathbf{M}[\textbf{plus1}] = \lambda n.\mathit{succ}\ n$
where $\mathbf{M}$: Operator $\rightarrow \mathit{Nat} \rightarrow \mathit{Nat}$
(e) $\mathbf{C}[\mathrm{C}_1 \textbf{ op } \mathrm{C}_2] = \lambda s.\mathbf{C}[\mathrm{C}_1]s \mathit{ combine } \mathbf{C}[\mathrm{C}_2]s$
(f) $\mathbf{E}[\textbf{valof } \mathrm{C} \textbf{ is } \mathrm{E}] = \lambda s.\mathbf{E}[\mathrm{E}](\mathbf{C}[\mathrm{C}]s)$
(g) $\mathbf{Q}[\textbf{procedure } \mathrm{C}] = \lambda s.(\lambda s'.\mathbf{C}[\mathrm{C}]s$
where $\mathbf{Q}$: Procedure $\rightarrow \mathit{Store} \rightarrow (\mathit{Store} \rightarrow \mathit{Store})$
(h) $\mathbf{C}[\textbf{call } \mathrm{P}(\mathrm{E})] = \lambda s.\mathbf{F}[\mathrm{P}](\lambda n.\mathbf{E}[\mathrm{E}]s)s$,
where $\mathbf{F}$: Procedure $\rightarrow (\mathit{Nat} \rightarrow \mathit{Nat}) \rightarrow \mathit{Store} \rightarrow \mathit{Store}$

Fig. 8. Examples.

(ii) If $E = (\lambda x.M)$: $C \rightarrow D$, and C is not *Store*-typed, then M contains no active *Store*-typed expressions.

It is informative to view examples of compliance with and violation of the requirements, as these provide intuition as to why the criteria are the proper ones. See Figure 8. Expressions (a)–(d) are single-threaded. In particular, expression(a) complies with property (Ai) of Definition 2.1 for its *Store*-typed expressions are nested, not disjoint. The call-by-value reduction strategy forces a lock-stepped evaluation. Expression (b) shows the proper use of multiple access rights, complying with property (Aii) of Definition 2.1, and expression (c) complies with property (Bi), of Definition 2.1, for the occurrence of the store variable s is properly bound to the enclosing λs, guaranteeing that the call-time store will be used with the procedure upon invocation. Expression (d) clearly satisfies property (Bii). Property (Ai) is violated by expression (e), for the disjoint active occurrences of $\mathbf{C}[\mathrm{C}_1]s$, and $\mathbf{C}[\mathrm{C}_2]s$ suggest that C_1 and C_2 will each need local stores to properly complete the noninterfering evaluation. Expression (f) violates property (Aii), which creates a problem when it is used in an expression such as $[(\textbf{valof } (\mathrm{C}) \textbf{ is } \mathrm{E}_1) + \mathrm{E}_2]$: a local store for E_1 is needed. Expression (g) violates property (Bi), for the procedure object saves its declaration-time store for evaluation and ignores its call-time one. Finally, expression (h) violates property (Bii), for the "thunk" $(\lambda n.\mathbf{E}[\mathrm{E}]s)$ created at procedure call time saves the call-time store for later use. All of expressions (e)–(h) can be intuitively seen as difficult to implement; Definition 2.1 provides the means for systematically detecting all such expressions.

Definition 2.1 needs to be supported by an additional restriction to ensure that an expression which uses (δ–) operators is indeed single-threaded; for example, expression (c) of Figure 9 is single-threaded provided that the operators *update* and that $\mathbf{C}[\mathrm{C}]$ both act in a single-threaded fashion. Say that an operator f, defined as $fx_1 \ldots x_n = e$, is *acceptable* if $\lambda x_1 . \ldots \lambda x_n . e$ is single-threaded. The operations of the *Store* algebra are taken as acceptable by definition, since they are assumed to be primitive and deal with the internal representation of the store.

The main results are as follows:

THEOREM 2.2. *An expression E which is single-threaded in its store argument and uses acceptable operators can be implemented by converting the Store algebra into a variable-containing class and by transforming E as directed in Section* 2.

Replace: $\mathbf{I} \to \mathbf{L}$
Replace = *replace* ∘ *subst*
where $subst(E, \nu) = [\nu/(\ )]E$,
that is, all active occurrences of ( ) in E are replaced by ν,
and
$replace\ (f) = f$
$replace(x) = x$
$replace\ ((\ )) = s : Store$
$replace(\lambda x.\ M) = \lambda x.replace\ M$
$replace(\lambda(\ ).M) = \lambda s : Store.replace\ M$
$replace((MN)) = (replace\ M\ replace\ N)$

Convert: $\mathbf{L} \to \mathbf{I}$
Convert = *init* ∘ *convert*
where $init(E) = ([(\ )/k]E, k)$,
that is, $[(\ )/k]E$ is E with the active occurrences of rnf *Store*-typed constant(s) k replaced by ( ).
and
$convert(f) = f$
$convert(x\colon C) = x\colon C,$ if $C \neq Store$
$convert(s : Store) = (\ )$
$convert(\lambda x : C.M) = \lambda x : C.convert\ M,$ if $C \neq Store$
$convert(\lambda s : Store.M) = \lambda(\ ).convert\ M$
$convert((MN)) = (convert\ M\ convert\ N)$

Fig. 9. Definitions of *Replace* and *Convert.*

COROLLARY 2.3. *A denotational definition whose semantic algebras use acceptable operators and whose valuation equations are acceptable can be implemented with a Store-containing class.*

The proof of Theorem 2.2 is an interpreter equivalance proof and is outlined in Section 4. Corollary 2.3 can be used to justify both interpreters and compilers for a language defined denotationally; these options are discussed in Section 5.

4. THE IMPLEMENTATION PROOF

The proof of Theorem 2.2 requires a precise formulation of the semantic notation, the intermediate notation with classes, the respective semantic algebras, rewriting rules, and replacement strategies. This task requires more space than what is available here, so the reader is referred to a companion paper [22] for the details. The main ideas are given here and are straightforward.

If an expression M rewrites in one step to an expression N, abbreviate the action by stating $M > N$. If M writes to N in zero or more steps, say $M \geq N$. The following two properties of single-threaded expressions are critical:

LEMMA 3.1. *If expression E is single-threaded and $E \geq E'$, then all abstractions in E' are single-threaded.*

PROOF. By induction on the number of rewrites to reach E' from E. The basis is immediate. For the inductive step, let $E \geq E''$, and $E'' > E'$ by the contraction of a redex R in E''. Those abstractions in E' disjoint from R in E''

remain single-threaded. If $R = (\lambda x \cdot M)N$ is a β-redex, consider $[N/x]M$: if N is *Store*-typed, then it will not be substituted for x into the body of any abstraction in M, for to do so would imply that the abstraction does not have property B of Definition 2.1. Hence all abstractions in M remain as they were—single threaded. In the case that N is not *Store*-typed, since N is in rnf, it contains no active *Store*-typed subexpressions (this fact requires a small, separate proof), so N's substitution for x maintains properties A and B in $[N/x]M$. On the other hand, if $R = (fN_1 \ldots N_m)$ is a δ-redex, then since f is an acceptable operator and all of $N_1, \ldots, N_m$ are in rnf, reasoning similar to that just given shows that $[N_m/x_m] \ldots [N_1/x_1]M$ contains only single-threaded abstractions. □

LEMMA 3.2. *If E is single threaded, $E \geq E'$, and there exist multiple, disjoint occurrences of active Store-typed expressions in E', then all are in rnf and possess the same value.*

PROOF. The proof is again an inductive one, and the basis is immediate. For the inductive step, say that $E \geq E'' > E'$ and let R be the redex contracted. If R is a *Store*-typed redex, by the inductive hypothesis it must be the only one. Regardless of R's internal structure, by Lemma 3.1, by property A of Definition 2.1, and by the definition of acceptable δ-operator, E' has the needed property. If R is non*Store*-typed, then consider β-redex $R = (\lambda x \cdot M)N$. If N is non*Store*-typed, then no new, active *Store*-typed expressions are found in $[N/x]M$, by Lemma 3.1 and property (Bii) of Definition 2.1. If N is *Store*-typed, then by Lemma 3.1 and properties (Aii) and (Bi) of Definition 2.1, all active *Store*-typed expressions in M are occurrences of x; hence no new clashes are introduced by $[N/x]M$. The reasoning is similar for a δ-redex. □

Another useful property of a single-threaded expression E is that it can be written (α-converted [2]) in a format which uses but one *Store*-typed identifier s. The lone identifier suffices throughout any derivation from E, that is, no *Store*-typed identifiers ever need to be renamed when doing rewritings. Because of this fact and Lemma 3.2, an obvious way to efficiently simulate the evaluation of a single-threaded expression E is to replace the active *Store*-typed subexpressions in E by occurrences of a marker ( ). Since the subexpressions are in rnf and have the same value, a global variable ν holds one copy of the value. The simulation represents an expression E' by a pair (E'', ν), where E'' is E' with active store values ν replaced by ( ). Evaluation proceeds as before, but whenever a *Store*-typed redex R in E' is contracted to an rnf value, the result is placed in ν and R is replaced by ( ) in E' (cf. Figure 7). Let **L** represent the original semantic notation and **I** be the one using the markers and global variable. A map *Convert*: **L** $\longrightarrow$ **I** translates an **L**-expression to an **I**-expression, and *Replace*: **I** $\longrightarrow$ **L** reinserts the variable's value in place of active markers to build an **L**-expression. The functions are defined in Figure 9. The operational equivalence proof is formalized as follows:

THEOREM 3.3. *For a single-threaded expression F the following holds: Convert$(F) \geq M'$ (in* **I**), *M' is in rnf, and Replace(M') equals F', iff $F \geq F'$(in* **L**) *and F' is in rnf.*

The proof is a standard tracking proof [4, 27] of the **L** system's rewrites by the **I** system, where the induction step is the commutative diagram:

$$\begin{tikzcd} F \arrow[r, "(\text{in } \mathbf{L})"] \arrow[d, "Convert"'] & F' \\ M \arrow[r, ">(\text{in } \mathbf{I})"] & M' \arrow[u, "Replace"'] \end{tikzcd}$$

Lemma 3.2 is the key to making the diagram commute.

5. APPLICATIONS AND EXTENSIONS

The primary application for the criteria of Definition 2.1 was mentioned in the introduction—the development of a processor for programs written in a language defined by a denotational definition. Processors can be either interpreters or compilers. An interpreter uses the valuation equations to direct the processing steps taken by an abstract machine which uses representations of the semantic notation and the global variable. The evaluation history of a program on the interpreter resembles the sequence seen in Figure 7. If a compiler-oriented viewpoint is taken, the compiler treats the transformed denotational equations as a translation scheme. A program is completely translated to an expression in the semantic notation and then evaluated. A structural induction over the syntax of the input language justifies Corollary 2.3 and the compiler viewpoint.

Another use of the criteria is in the optimization of applicative programs. In conventional computer architectures, typical implementations of applicative programs are inefficient and waste memory. The single-threading criteria complements data flow analysis [14] for detecting situations in which destructive updates apply and storage can be better managed.

It is also possible to use Definition 2.1 to judge if a proposed combinator set for defining implementable sequential languages is useful: map each combinator to its underlying lambda expression and check for single-threadedness. Application of the criteria to existing systems [13, 26] makes apparent that the implicit sequential handling of the store is a key feature of the combinators.

An obvious extension to the results of Section 2 is to admit product domains into the type system of the semantic notation. This can be handled in a number of ways; one easy approach is to admit the pairing constructor $(a, b){:}A \times B$, for $a{:}A$ and $b{:}B$, into the notation along with its associated subscripting operations *fst* and *snd*. A pair is in rnf iff its components are. Treat any pair (M, N), where either M is *Store*-typed or N is *Store*-typed, as also being "*Store*-typed." Definition 2.1 and its consequences carry through as before. By using tuples, multiple global variables can be created. Since disjoint unions can be simulated with tuples, Definition 2.1 can deal with them as well. Finally, by creating local semantic algebras as needed, local variables within fixed scopes of the semantic definition are possible. These extensions suggest that the basic single-threadedness property has wide and useful application.

6. CONCLUSIONS

Simple, statically checkable criteria for determining the suitability of an imperative implementation of a function expression's store argument have been given. The criteria are useful for directing the implementation of denotational definitions as well as detecting possible optimizations in functional programs. The criteria are also helpful for analyzing the suitability of combinator notations for specifying implementations of languages. Finally, the hybrid imperative/applicative notation seen in Figure 6 provides insight into the relationship between functional and imperative languages and shows a useful transition between the two.

The example in Section 1 was in direct semantics style [23], as this format is most removed from sequential imperative languages. Most implementation-oriented notations use continuation style—now it is easy to see why. One of the primary features of a continuation semantics is its tail-recursive passing of stores. In any configuration of a partially interpreted program expression in continuation form, there is exactly one active occurrence of a store. In fact, a properly written continuation semantics definition of a programming language will contain no occurrences of *Store*-typed expressions at all! Such a definition is trivially single threaded. The results reported here may be viewed as a generalization of conventional continuation style sequentiality.

ACKNOWLEDGMENT

I would like to thank Neil Jones for introducing me to denotational semantics.

REFERENCES

1. BJØRNER, D. Rigorous development of interpreters and compilers. In D. Bjørner and C. Jones (Eds.), *Formal Specification and Software Development*. Prentice-Hall, Englewood Cliffs, N.J., 1982, pp. 271–320.
2. CURRY, H. B., AND FEYS, R. *Combinatory Logic, Vol. 1*. Elsevier North-Holland, New York, 1958.
3. GOGUEN, J., AND PARSAYE-GHOMI, K. Algebraic denotational semantics using parameterized abstract modules. In *Lecture Notes in Computer Science 107: Colloquium on Formalization of Programming Concepts* (Pensicola, Spain). Springer-Verlag, New York, 1981, pp. 292–309.
4. HOARE, C. A. R. Proof of correctness of data representations. *Acta Inf. 1* (1972), 271–281.
5. HUET, G., AND OPPEN, D. Equations and rewrite rules: A survey. In R. Book (Ed.), *Formal Language Theory: Perspectives and Open Problems*. Academic Press, New York, 1980, pp. 349–405.
6. JONES, C. B. The meta language. In D. Bjørner and C. Jones (Eds.), *Formal Specification and Software Development*. Prentice-Hall, Englewood Cliffs, N.J., 1982, pp. 271–320.
7. JONES, N. D. (Ed.) *Lecture Notes in Computer Science 94: Semantics Directed Compiler Generation*. Springer-Verlag, New York, 1980.
8. JONES, N. D., MUCHNICK, S. S., AND SCHMIDT, D. A. A universal compiler. Tech. Rep. DAIMI IR-17, Computer Science Dept., Univ. Aarhus, Denmark, 1979.
9. MOSSES, P. D. Making denotational semantics less concrete. Presented at the workshop on semantics of programming languages, Bad Honnef, Germany, 1977.
10. MOSSES, P. D. SIS—Reference manual and user's guide. Tech. Rep. DAIMI MD-30, Computer Science Dep., Univ. Aarhus, Denmark, 1979.
11. MOSSES, P. D. A constructive approach to compiler correctness. In N. D. Jones (Ed.) *Lecture Notes in Computer Science* 94: *Semantics Directed Compiler Generation*. Springer-Verlag, New

York, 1980, pp. 189–210.

12. MOSSES, P. D. A semantic algebra for binding constructs. In *Lecture notes in Computer Science 107: Formalization of Programming Concepts* (Pensicola, Spain). Springer-Verlag, New York, 1981.
13. MOSSES, P. D. Abstract semantic algebras! In *Formal Description of Programming Concepts* II. Elsevier North-Holland, New York, 1983.
14. MYCROFT, A. Abstract interpretation and optimizing transformations for applicative programs. Ph.D. Dissertation, Computer Science Dep., Univ. Edinburgh, Scotland, 1981.
15. O'DONNELL, M. *Lecture Notes in Computer Science 58: Computing in Systems Described by Equations.* Springer-Verlag, New York, 1977.
16. PLOTKIN, G. LCF considered as a programming language. *Theor. Comput. Sci. 5*(1977), 223–255.
17. PLOTKIN, G. The category of complete partial orders. In *Proceedings, Summer School on Foundations of Artificial Intelligence and Computer Science* (Institute di Scienze dell' Informazione, Universita di Pisa), 1978.
18. RAOULT, J.-C., AND SETHI, R. Properties of a notation for combining functions. In *Lecture Notes in Computer Science 140: Ninth Colloquium on Automata, Languages, and Programming* (Aarhus, Denmark). Springer-Verlag, New York, 1982, pp. 429–441.
19. RAOULT, J.-C., AND SETHI, R. The global storage needs of a subcomputation. In *Proceedings of the 11th ACM Symp. Principles of Programming Languages* (Salt Lake City, Utah, 1984), pp. 148–157.
20. REYNOLDS, J. C. Definitional interpreters for higher order programming languages. In *Proceedings of the 25th ACM Nat. Conf.*, (Boston, 1972), pp. 717–740.
21. SCHMIDT, D. A. Denotational semantics as a programming language. Internal Rep. CSR-100, Computer Science Dep., Univ. Edinburgh, Scotland, 1982.
22. SCHMIDT, D. A. Detecting global variables in denotational specifications (extended version). Tech. Rep. 84-3, Computer Science Dep., Iowa State Univ., Ames, Iowa, 1984.
23. STOY, J. *Denotational Semantics.* MIT Press, Cambridge, Mass., 1977.
24. WAND, M. First order identities as a defining language. *Acta Inf. 14*(1980), 337–357.
25. WAND, M. Semantics-directed computer architecture. In *Proceedings of the 9th ACM Symp. Principles of Programming Languages* (Albuquerque, N. Mex., 1982), pp. 234–241.
26. WAND, M. Deriving target code as a representation of continuation semantics. *ACM Trans. Prog. Lang. Syst. 4*(1982), 496–517.
27. WEGNER, P. Programming language semantics. In R. Rustin (Ed.), *Formal Semantics of Programming Languages.* Prentice-Hall, Englewood Cliffs, N.J., 1972, pp. 149–248.

Received September 1983; revised September 1984; accepted October 1984