Detection of single-threading properties in combinator notations

Dean Roger Lass
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons

Recommended Citation
https://lib.dr.iastate.edu/rtd/9543
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

University Microfilms International
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600
Detection of single-threading properties in combinator notations

Lass, Dean Roger, Ph.D.

Iowa State University, 1991
Detection of single-threading properties in combinator notations

by

Dean Roger Lass

A Dissertation Submitted to the Graduate Faculty in Partial Fulfillment of the Requirements for the Degree of DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved: Members of the Committee:

Signature was redacted for privacy.

Signature was redacted for privacy.

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa
1991
# TABLE OF CONTENTS

**ACKNOWLEDGEMENTS** ........................................... v

**CHAPTER 1. BACKGROUND** ................................. 1

  - Problem Motivation ..................................... 1
    - A Functional Programming Example .................. 1
    - Semantics-Directed Compiler Generation .......... 2
  - Single-threading Detection and Transformation .... 7
    - Single-threading Detection in Typed \( \lambda \)-Calculus .. 7
    - The Global Variable Transformation ................ 10
  - Goals of Our Research .................................. 14

**CHAPTER 2. TML SINGLE-THREADING CRITERIA** ........ 16

  - The TML Combinator Language .......................... 16
  - Our TML Variant ......................................... 20
  - The Single-Threading Criteria ......................... 24
    - Acceptable \( f_i \) Operators ......................... 25
    - \( s_0 \)-trivial Sets .................................. 29
    - \( S \)-consistent Expressions ......................... 33
    - Single-threading and s-t Expressions .............. 36
  - Sufficiency of the s-t Criteria ....................... 42
CHAPTER 3. GENERALIZED COMBINATOR NOTATIONS . . 69

Introduction ........................................... 69
The Combinator Notation ............................... 69
Acceptable Rewriting Rules ............................. 72
Static Criteria for Acceptability ..................... 76
  Criteria for First-Order Combinators ............... 77
  Criteria for Higher-Order Combinators ............. 82
Comparison to the TML Work .......................... 88

CHAPTER 4. IMPLEMENTATION OF THE TML CRITERIA . 92

The PSI/DAOS System .................................. 92
  The PSI System ....................................... 92
  The DAOS System .................................... 97
Single-Threading and Globalization Implementations . 98
Testing of the Implementation ........................ 103

CHAPTER 5. FUTURE WORK ............................. 108

BIBLIOGRAPHY ......................................... 111
LIST OF FIGURES

Figure 1.1: An Example of a Denotational Semantics Definition ........ 4
Figure 1.2: The Defunctionalized Store Domain ......................... 11
Figure 1.3: Semantic Equations After Global Variable Introduction ... 12
Figure 1.4: The Globalized Store Domain ................................. 12
Figure 2.1: The TML Metalanguage ....................................... 17
Figure 2.2: Semantic Valuation Functions in TML ........................ 19
Figure 2.3: Syntax of the TML Variant .................................. 21
Figure 2.4: Types of TML Expressions ................................. 22
Figure 2.5: TML Rewriting Rules ........................................ 23
Figure 2.6: GVS Combinator Rewriting Rules ............................. 59
Figure 3.1: The Combinator Language .................................... 70
Figure 3.2: Rule Set for Computing $2^k$ ................................ 79
Figure 3.3: hoc Rules for Computing $2^k$ ................................ 88
Figure 4.1: Test Programs for Execution Timings .......................... 105
LIST OF TABLES

4.1 Execution Timing Results .......................... 105
ACKNOWLEDGEMENTS

I am indebted to my advisor, David A. Schmidt, for his assistance, guidance and support during my studies. I also gratefully extend thanks to all of my committee members for their seemingly boundless patience.
CHAPTER 1. BACKGROUND

Problem Motivation

Our research is focused on certain inefficiency problems that have been noted in implementations of functional programming languages, and in the area of semantics-directed compiler generation. We have sought to develop techniques which can be used to recognize occurrences of these inefficiencies and make implementation transformations which alleviate them.

A Functional Programming Example

The inefficiency we are discussing can be illustrated with a simple example, a recursive definition of a function to compute the value $2^k$:

```latex
let exp2 m n ⇒ if (eq0 m) n (exp2 (pred m) (plus n n))
in exp2 k 1
```

A typical implementation of this function will make use of a stack of environment records to handle the recursive calls. Each time a recursive call is made, the values of $m$ and $n$ stored in the current environment record are used to calculate $(\text{pred } m)$ and $(\text{plus } n n)$, which will be the values of $m$ and $n$, respectively, in a new environment record that is then pushed on the stack. In a computation of $exp2 k 1$, the stack will eventually contain $k + 1$ environment records, each with storage cells for $m$ and $n$. 
But there is an alternative method of implementation which reduces the storage overhead. Because of the tail-recursive form of the function expression, we could safely use two global storage cells to store the values of \( m \) and \( n \) during evaluation. To see this, note that once we have computed \( \text{pred} \ m \) and \( \text{plus} \ n \ n \) as the next values for \( m \) and \( n \), we have no more need for their current values, and that when the recursion stops (when \( m = 0 \)) the value of \( n \) at that point is returned as the result. Because of this property, we could store the values in two global storage cells, and simply update the values in these cells instead of creating new storage on the stack. This will conserve storage space and also reduce the execution time overhead required for manipulation of the environment stack.

It is this opportunity for optimization in functional expressions that we wish to be able to detect and exploit. When an expression has some argument which can be implemented in this way (i.e., by sequential, destructive updating of a global storage cell) we will say that the expression is single-threaded in that argument.

**Semantics-Directed Compiler Generation**

Much research has been done in recent years on methods for the automatic generation of compilers from formal specifications of the syntax and semantics of programming languages [2, 10, 11, 16]. This can be seen as an extension of earlier work which showed that parsers can be automatically generated from a specification of a language's syntax.

The type of formal specification in which we are particularly interested, denotational semantics [33, 36], uses a formal notation (a metalanguage) to specify the semantics of a programming language. The underlying viewpoint is mathematical,
in that programs and the values that they manipulate are considered to be abstract objects (numbers, functions, etc.) in various mathematical domains.

Because of this viewpoint, the metalanguage should be a notation that lends itself to the description of mathematical functions, which naturally leads to the use of notations that are similar to functional programming languages. In the earliest work in denotational semantics, the most typical choice for the semantic metalanguage was some version of the $\lambda$-calculus [7], but in recent years there has been increasing interest in various combinator-based notations [5, 7, 14, 23, 30, 37].

The semantics of a programming language is defined by a collection of valuation functions, which map each syntactic construct to an expression in the metalanguage. Figure 1.1 shows (a portion of) a typical semantic definition for a simple sequential language. For example, there is a valuation function $C$ which maps a command in the language to its denotation as a $Store \rightarrow Store$ function, which is described by a $\lambda$-calculus expression. The semantics is “compositional” in the sense that the semantics of a complex construct is built up from the semantics of its subparts. This concept is illustrated in the equation $C[C_1;C_2] = \lambda s. C[[C_2] C[C_1]] s$, which defines the meaning of the sequential composition of commands $C_1$ and $C_2$ in terms of the individual meanings of the two commands.

Given the semantic definition of a language, a simple compiler for the language can be generated. The compiler uses the semantic definition to translate a source program to its metalanguage representation. At that point we can “execute” the source program by using an interpreter to evaluate the metalanguage expression. But as was noted earlier, the semantic metalanguages are usually very similar to functional programming languages, so the type of implementation problems described
Semantic Algebra for Store Domain (Store = Identifier → Nat)

empty : Store
    empty = λi. error

access : Identifier → Store → Nat
    access i s = s(i)

update : Identifier → Nat → Store → Store
    update i n s = λi'. (i' = i) → n || s(i)

P: Program → Store → Store

P[C.] = C[C]

C: Command → Store → Store

C[I := E] = λs. update I[I] (E[E] s) s
C[C₁ ; C₂] = λs.C[C₂](C[C₁] s)
C[if B then C₁ else C₂] = λs. (B[B] s → C[C₁] || C[C₂]) s
C[while B do C] = fix (λf. λs. B[B] s → f(C[C] s) || s)

E: Expression → Store → Nat

E[E₁ + E₂] = λs. (E[E₁] s) plus (E[E₂] s)
E[I] = λs. access I[I] s

B: Boolean-expression → Store → Bool

B[E₁ = E₂] = λs. (E[E₁] s) equals (E[E₂] s)

Figure 1.1: An Example of a Denotational Semantics Definition
in the earlier exp2 example can cause our generated compiler to perform poorly.

For example, consider the translation of a single if-then-else command under the semantic definition in Figure 1.1:

\[
C[\text{if } X = Y \text{ then } A := B \text{ else } A := C] = \\
\lambda s.((\lambda s.((\lambda s.\text{access } X \ s) \ s) \text{ equals } ((\lambda s.\text{access } Y \ s) \ s)) \ s \\
\rightarrow (\lambda s.\text{update } A ((\lambda s.\text{access } B \ s) \ s) \ s) \\
\| (\lambda s.\text{update } A ((\lambda s.\text{access } C \ s) \ s) \ s)) \ s
\]

If we have an interpreter that performs λ-calculus β-reductions, we can apply this expression to a store argument and reduce to normal form, which gives us the effect of executing the if-then-else command. But the efficiency of this "execution" will largely depend on how efficiently our λ-calculus interpreter handles the numerous bindings required during β-reduction.

If each binding requires a copying of the store argument, there will be a significant execution time overhead (note, for instance, that three successive bindings are required before the store argument becomes available to the two access operations in the conditional test). Another concern involves expressions in which the store argument is bound to multiple instances of the store variable at the same expression level. For instance, if we let \( s_0 \) represent the program store argument, one step in executing our example command will be the reduction:

\[
(\lambda s.((\lambda s.\text{access } X \ s) \ s) \text{ equals } ((\lambda s.\text{access } Y \ s) \ s)) \ s_0 \\
\Rightarrow (\lambda s.\text{access } X \ s) \ s_0 \text{ equals } ((\lambda s.\text{access } Y \ s) \ s_0)
\]

If a faithful implementation of the intended semantics requires that separate copies of the store be created for the two instances of \( s_0 \) in the rewritten expression, this
will also have a detrimental effect on the performance of the system.

Fortunately, an analysis of the semantic definition for our example will reveal that, like the arguments \(m\) and \(n\) in the earlier \(exp2\) example, the store in our simple language can be safely implemented with global storage, i.e., the semantics is single-threaded in the \(Store\) argument. (The analysis is done in a structural induction style, and if each semantic equation is found to have the single-threading property individually, then we conclude that the property holds for the semantics as a whole.)

In dealing with semantic definitions, we are interested in looking for single-threading properties in the nonprimitive semantic domains (such as the \(Store\) domain in our example) which are used in all programs of the defined language. Detection of the single-threading property in such situations has a much larger potential benefit than the detection of single-threaded arguments in a particular program such as the \(exp2\) example, for two reasons. First, because the store argument in a typical program will require a considerably larger amount of storage than the two integer arguments in \(exp2\), the potential savings in binding and storage overhead through the use of global storage is much greater. Second, since it is the semantic definition itself that displays the single-threading property, we know that the globalization of the program store can be done for every program written in the language. Thus, the global storage method can be implemented through an transformation of the semantic equations, and we avoid having to check each program individually for the single-threading property.

In the next two sections, our discussion of single-threading detection and transformation focuses primarily on the issues involved in applying the work to semantic definitions.
Single-threading Detection and Transformation

Earlier, we described the single-threading property in operational terms; namely, that if an expression is single-threaded in a given argument, then we can use destructively updated global storage to implement that argument during evaluation of the expression. In this section, we informally discuss the criteria developed by Schmidt for the detection of single-threading in typed $\lambda$-calculus expressions, and how the expressions can be transformed to take advantage of a global storage implementation.

Single-threading Detection in Typed $\lambda$-Calculus

In [31], Schmidt developed syntactic criteria which are sufficient to detect single-threaded arguments in typed $\lambda$-calculus expressions. (The detection of a single-threaded argument is reducible to the halting problem, so no absolute necessary and sufficient criteria can be given. Thus, some arguments which have the single-threading property may not be detectable by the criteria.) The criteria are designed to check whether an expression has two properties, termed noninterference and immediate evaluation, which insure that a global storage implementation can be safely used for the argument.

These two properties are designed to insure that during the evaluation of an expression, there is no point at which it is necessary to have duplicate copies of the single-threaded argument available (noninterference), or at which we discard the current value of the argument and retrieve an earlier value (immediate evaluation). If the expression satisfies these restrictions, then the argument can be safely implemented by a single global storage location which is destructively updated.

To illustrate these properties, we use the semantic equations given earlier (Fig-
ure 1.1), which are single-threaded in the Store argument, and some other example equations which are not.

The noninterference property concerns conflicts between active subexpressions (those subexpressions which are not properly contained within a \( \lambda \)-abstraction) of the argument type, in this case those of type Store. The idea is to insure that if there are disjoint subexpressions of type Store, they all represent the same store state. In Schmidt's work, this is enforced by the requirement that in such a case, all of these disjoint subexpressions are the same store identifier.

For example, in the semantic equation for the assignment statement,

\[
\mathcal{C}[I := E] = \lambda s. \text{update } \mathcal{I}[I] \ (\mathcal{E}[E] \ s) \ s
\]

there are three subexpressions of type Store to consider: the abstraction body update \( \mathcal{I}[I] \ (\mathcal{E}[E] \ s) \ s \) and the two (disjoint) occurrences of the store identifier \( s \). Since the subexpression \( \mathcal{E}[E] \ s \) reduces to a value of type Nat and we assume (for the purposes of the single-threading analysis) that this reduction has no side-effect on the store, both occurrences of the store variable could be replaced by a reference to a global location, and we can be sure that the same store state will be used in both cases, which is the desired semantic effect. Further, since the abstraction body is not disjoint from the two occurrences of \( s \), the changing of the store state by the application of the update operator creates no conflict, since both occurrences of \( s \) disappear in the process of reducing the update application, leaving us with a single reference to a new store state.

As an example of a violation of the noninterference property, consider the semantic equation:

\[
\mathcal{C}[C_1 \land C_2] = \lambda s. (\mathcal{C}[C_1] \ s) \ \text{combine} \ (\mathcal{C}[C_2] \ s)
\]
The idea here is that we have a command construct $C_1 \land C_2$ in which the intended semantics is to execute both $C_1$ and $C_2$ (the order is unspecified) using the same initial store value $s$. Since each command can potentially change the store, we then take the two new stores and somehow combine them into a single new store state which is the result of the construct as a whole. The important point here is that we cannot use a single global store and get the desired semantics, because no matter which of the two subcommands we evaluate first, we must retain a copy of the original store $s$ for use in evaluating the other command.

The immediate evaluation property is intended to prevent "hiding" of store states in $\lambda$-abstractions. In Schmidt's criteria, if we have the expression $\lambda x. M : T \rightarrow T'$ and $T = \text{Store}$, then all active identifiers of type $\text{Store}$ in $M$ must be $x$, and if $T \neq \text{Store}$, then there can be no active subexpressions of type $\text{Store}$ in $M$. All the semantic equations in Figure 1.1 have this property. A simple example that violates this property is the semantic equation:

$$D[\text{procedureC}] = \lambda s . (\lambda s'. C[\_s] s)$$

where $D : \text{Procedure} \rightarrow \text{Store} \rightarrow (\text{Store} \rightarrow \text{Store})$

The intended semantics here is that at the declaration of procedure $C$, the store which is current at declaration time ($s$) is bound into a $\lambda$-abstraction of type $\text{Store} \rightarrow \text{Store}$. A call of the procedure would then involve applying this abstraction to the call-time store ($s'$), which the procedure ignores in favor of the hidden declaration-time store. Clearly, a single global store is not sufficient to implement this kind of effect.
The Global Variable Transformation

Once we have determined that a set of semantic equations is single-threaded in a type $S$, we can apply a two-step transformation to the semantic definition which will implement the global variable strategy.

The first step is to modify the definition of the semantic domain which we have determined to be single-threaded. This step is usually referred to as defunctionalization, a term that derives from the fact that nonprimitive semantic domains in a denotational definition are usually described as function spaces, and operations which modify the domain are thus higher-order functions. For example, in the semantic algebra for $\text{Store}$ given in Figure 1.1, we have

$$\text{Store} = \text{Identifier} \to \text{Nat}$$
$$\text{update} : \text{Identifier} \to \text{Nat} \to \text{Store} \to \text{Store}$$

But the manipulation and storage of function closures is usually far less efficient than the handling of data structures such as arrays or lists. So while there are formal reasons that the functional description is useful in denotational definitions, when we want to implement the definition we should seek a simpler, nonfunctional representation. Since the store structure in our example simply associates identifiers with values, we can change the definition to employ some first-order data structure, say a list of name/value pairs, and then operations on the store become list manipulation operations, which are more efficiently implementable. Figure 1.2 shows a defunctionalized version of the $\text{Store}$ algebra which uses a list representation.

Defunctionalization can be performed even in the absence of single-threadedness, and has been used to some extent in previous compiler generation systems [27, 28].
\[ Store = (\text{Identifier} \times \text{Nat})^* \]

\[ \text{empty} : \text{Store} \]
\[ \text{empty} = [] \]

\[ \text{access} : \text{Identifier} \rightarrow \text{Store} \rightarrow \text{Nat} \]
\[ \text{access} ~ i ~ s = \text{let lookup} ~ i' ~ s' = \]
\[ \begin{align*}
    (s' = []) & \rightarrow \text{error} \\
    \text{let} ~ (j, n) = \text{hd} ~ s' & \text{ in } (i' = j) \rightarrow n \\
    \text{in lookup} ~ i' ~ (\text{tl} ~ s') & \text{ in lookup} ~ i ~ s
\end{align*} \]

\[ \text{update} : \text{Identifier} \rightarrow \text{Nat} \rightarrow \text{Store} \rightarrow \text{Store} \]
\[ \text{update} ~ i ~ n ~ s = \text{cons} ((i, n), s) \]

Figure 1.2: The Defunctionalized \textit{Store} Domain

But since single-threadedness holds for our store domain, we can go a step further and “globalize” the defunctionalized store, and also change the way in which the semantics equations reference the store. The identifiers representing the store in the equations are replaced by a marker symbol which represents a right to use the store. Figure 1.3 shows our set of semantic equations after replacement of the \( s \) identifiers with the marker symbol ( ).

The defunctionalized semantic algebra for \textit{Store} is also transformed further. A global variable will be introduced to hold the list of name/value pairs representing the store. The operations on the store will also use the () marker to denote an access right to the store. One significant difference in the new implementation is that operations which modify the store do so as a side-effect, merely returning the () marker as the “result” of the modification. Figure 1.4 shows the globalized \textit{Store} domain.

These new semantic equations may not appear to be any different from the
\( \mathcal{P} \): Program \( \rightarrow \text{Store} \rightarrow \text{Store} \)

\( \mathcal{P}[C.] = C[C] \)

\( \mathcal{C} \): Command \( \rightarrow \text{Store} \rightarrow \text{Store} \)

\[
C[I:=E] = \lambda(). \text{update } I[I] (E[E] ()) ()
\]

\[
C[C_1;C_2] = \lambda(). C[C_2][C[C_1] ()]
\]

\[
C[\text{if } B \text{ then } C_1 \text{ else } C_2] = \lambda(). (B[B] () \rightarrow C[C_1] [C[C_2]]) ()
\]

\[
C[\text{while } B \text{ do } C] = \text{fix} (\lambda f. \lambda(). B[B] () \rightarrow f(C[C] ()) []())
\]

\( \mathcal{E} \): Expression \( \rightarrow \text{Store} \rightarrow \text{Nat} \)

\[
\mathcal{E}[E_1 + E_2] = \lambda(). (\mathcal{E}[E_1] () \text{ plus } (\mathcal{E}[E_2] ()])
\]

\[
\mathcal{E}[I] = \lambda(). \text{access } I[I] ()
\]

\( \mathcal{B} \): Boolean-expression \( \rightarrow \text{Store} \rightarrow \text{Bool} \)

\[
B[E_1 = E_2] = \lambda(). (\mathcal{E}[E_1] ()) \text{ equals } (\mathcal{E}[E_2] ())
\]

Figure 1.3: Semantic Equations After Global Variable Introduction

\[\text{Store} = (\text{Identifier} \times \text{Nat}) \text{ list}\]

\[\text{var store\_cell: Store = (Identifier \times Nat) list}\]

\[\text{empty: Store}
\]

\[
\text{empty} = () \quad \text{with side-effect store\_cell := []}
\]

\[\text{access: Identifier \rightarrow Store \rightarrow Nat}\]

\[
\text{access } i () = \text{let lookup } i' s' =
\]

\[
(s' = []) \rightarrow \text{error}
\]

\[
\text{let } (j, n) = \text{hd } s'
\]

\[
\text{in } (i' = j) \rightarrow n
\]

\[
\text{in lookup } i' (\text{tl } s')
\]

\[\text{in lookup } i \text{ store\_cell}\]

\[\text{update: Identifier \rightarrow Nat \rightarrow Store \rightarrow Store}\]

\[
\text{update } i \ n () = ()
\]

\[\text{with side effect store\_cell := cons((i,n), store\_cell)}\]

Figure 1.4: The Globalized Store Domain
equations given in Figure 1.1, except for the textual substitution of () for the variable s. The point is that the program store, which in realistic programs is typically a large data structure, is no longer explicitly present in the metalanguage expression. This means that if we use the simple-minded compilation strategy of translating a program to its semantic representation in the metalanguage and then interpreting the metalanguage expression, the interpreter can perform the normal bindings and substitutions associated with expression evaluation on a simple marker value. The presence of this marker is used to indicate the right to access the global store variable, and these accesses are handled as a side-effect of the normal interpretation process. The hope is that this will result in a more efficient implementation in terms of both time and space usage.

Finally, we should note that the single-threading detection and transformation methods discussed here in regard to semantic equations can be applied in a similar way to the expressions which define individual functional programs. This is because of the structural similarity of the two situations, namely, sets of interrelated equations written in a functional notation of some sort. The key in both cases is an ability to analyze the notation used in the definitions in order to detect the relevant single-threading properties.

In the semantics-directed compiler generation case, it seems that the benefits of the single-threading analysis may be twofold. By analyzing the semantic definition, we detect single-threading properties which apply to every program in the defined language. But since in the compilation process we translate each individual program to an expression in the same notation used for the semantic definition, we can then further analyze this expression for single-threading properties unique to that
program's structure, possibly finding more possibilities for optimization at this level.

Goals of Our Research

Much of the previous work on the detection of single-threading properties has employed various forms of dataflow analysis [4, 6, 8, 9, 12, 15, 35] to examine the possible run-time contexts in which a data object may appear. A significant problem with this approach is that the flow analysis requires exponential time in the worst case [6].

In Schmidt's work [31] on the typed $\lambda$-calculus, a linear-time static analysis based on an examination of the types of subexpressions was developed. Since runtime contexts are ignored, the static analysis is not as powerful as the use of dataflow analysis; some occurrences of single-threading which can be detected by dataflow analysis will be missed by the static analysis. But the reduced time requirements for the static analysis make it an attractive option.

One goal of our research was to extend Schmidt's ideas to the development of static single-threading analyses for combinator notations, which have been drawing increasing attention in recent years for their usefulness in the definition and implementation of programming languages [5, 7, 14, 23, 30, 37]. We have developed static analyses (described in Chapters 2 and 3) for both a particular combinator language, TML [23], and for a generalized combinator notation. In the generalized case, our static analysis criteria are also almost totally independent of the choice of reduction strategy, which is a significant difference from most earlier research.

Our second goal was to produce an implementation of our single-threading criteria for TML in order to judge the feasibility and usefulness of the criteria. This
was done in the framework of the PSI/DAOS system [1, 22], which is a compiler generation system based on the use of TML as a semantic metalanguage. This work is described in Chapter 4.
CHAPTER 2. TML SINGLE-THREADING CRITERIA

The TML Combinator Language

Our work began by developing static analysis criteria for the detection of single-threaded expressions in a particular combinator language. The language used is a variant of TML (Two-level MetaLanguage), which was developed by Hanne and Flemming Nielson [23]. TML was intended to be used in the study of various issues in the area of denotational semantics, particularly issues of compiler generation from semantic specifications. For the purposes of their own work, the Nielsons were particularly interested in making an explicit distinction between compile-time and run-time bindings. TML is therefore a combination of λ-calculus and combinator notations, which specify compile-time and run-time entities, respectively. In addition, TML is also the semantic metalanguage of a working compiler generation system, the PSI system [22], which we are using as the testbed for an implementation of our single-threading work.

Figure 2.1 shows the type structure and expression syntax of TML. Because of the Nielsens's desire to make the distinction between compile-time and run-time objects explicit, the type structure underlying the metalanguage is split into compile-time and run-time types, denoted by ct and rt, respectively, in Figure 2.1. In both parts of the type structure, we have base types (denoted by $A_i$ and $B_i$), product types,
Types:
\[ \text{ct ::= } A_i | ct_1 \times \cdots \times ct_k | ct_1 + \cdots + ct_k \quad (k \geq 2) \]
\[ | \text{ct}_1 \rightarrow \text{ct}_2 | \text{rec } X_i . \text{ct} | X_i | \text{rt}_1 = \text{rt}_2 \]
\[ \text{rt ::= } B_i | rt_1 \times \cdots \times rt_k | rt_1 \pm \cdots \pm rt_k \quad (k \geq 2) \]
\[ | \text{rt}_1 \Rightarrow \text{rt}_2 | \text{rec } Y_i . \text{rt} | Y_i \]

Expression Forms:
\[ e ::= f_i \]
\[ | (e_1, \ldots, e_k) | e \downarrow i \quad \text{ct} \rightarrow \text{constants} \]
\[ | \text{in } i e | \text{is } i e | \text{out } i e \quad \text{ct} \rightarrow \text{product types} \]
\[ | \text{lam } x . e | e_1 e_2 | x \quad \text{ct} \rightarrow \text{sum types} \]
\[ | \text{mkrec } e | \text{unrec } e \quad \text{ct} \rightarrow \text{function types} \]
\[ | e \rightarrow e_1, e_2 \quad \text{ct} \rightarrow \text{recursive types} \]
\[ | \text{fix } e \]
\[ | \text{tuple } (e_1, \ldots, e_k) | \text{take}_i \quad \text{rt} \rightarrow \text{product types} \]
\[ | \text{in } i | \text{case } (e_1, \ldots, e_k) \quad \text{rt} \rightarrow \text{sum types} \]
\[ | \text{curry } e | \text{apply} \quad \text{rt} \rightarrow \text{function types} \]
\[ | \text{mkrec } | \text{unrec} \]
\[ | \text{cond } (e_1, e_2, e_3) \]
\[ | e_1 \parr e_2 \]
\[ | \text{id} \]
\[ | \text{const } e \]

Figure 2.1: The TML Metalanguage
sum types, function types, and recursive types with their associated type variables. (The underlining of symbols in the \( \texttt{rt} \) type notation is done to help distinguish the two levels of types.) In the compile-time type structure we also have run-time functions \( (\texttt{rt}_1 \rightarrow \texttt{rt}_2) \), which reflects the fact that one of the actions of the compile-time phase of a compiler is the construction of run-time code.

The expression syntax of TML corresponds to the two-level type structure. Expressions of compile-time type are written in a \( \lambda \)-calculus-like syntax, while a combinator notation is used for run-time entities. For both compile-time and run-time types, we have expression forms for defining and using entities of product, sum, function and recursive types. The \( \texttt{f}_i \) expressions represent compile-time constants, which include both base type values and function-typed operators. In particular, note that since run-time functions are also considered to be compile-time entities, a run-time operator (such as an addition operator on run-time integers) would also be considered an example of an \( \texttt{f}_i \) in the TML framework. In the expression \( \texttt{fix} \ e \), the subexpression \( e \) must be (compile-time or run-time) function-typed. The \( \texttt{id} \) combinator is used to distribute values in run-time expressions by passing them unchanged. The \( \texttt{const} \ e \) expression, when applied to an argument, will ignore the argument and reduce to its subexpression \( e \). It is primarily used to introduce an \( \texttt{f}_i \) operator into an expression in a situation where well-formedness typing rules would not allow us to simply write \( \texttt{f}_i \) itself. We also have conditional expressions for both levels and the \( \Box \) combinator for composing run-time functions.

To illustrate how the compile-time and run-time notation mix, Figure 2.2 shows TML versions of the semantic equations given earlier in Figure 1.1 for the \( C \) valuation function. The new equations are primarily made up of the combinator notation. The
exceptions are the use of the \textit{fix} and \textit{\lambda}-abstraction in the while-command equation, and the compile-time application (\textit{update } \mathcal{I}[I]) in the assignment statement. In the first case, the \textit{\lambda}-variable \textit{f} must be of type \textit{Store} \rightarrow \textit{Store}, so we are taking the fixpoint of a run-time functional, which is allowed by the inclusion of run-time functions as compile-time objects. In the assignment case, we note that in the two-level type system, the appropriate type for the \textit{Store} algebra's \textit{update} operation is:

\[
\textit{update} : \textit{Identifier} \rightarrow \textit{Nat} \rightarrow \textit{Store} \rightarrow \textit{Store}
\]

This makes explicit the fact that the identifier in the assignment command can be processed at compile-time, hence the compile-time application embedded in the TML expression. The fact that the other two arrows in the type description are underlined makes it clear that the \textit{Nat} and \textit{Store} arguments needed to complete the assignment evaluation will not be available until run-time.
Our TML Variant

Because we are concerned with detecting single-threaded run-time arguments, our focus was on developing single-threading criteria for analyzing TML's combinator notation, and we will deal no further with the compile-time portion of the language. To simplify the development, we have also omitted the sum types and recursive types allowed in the full version of TML, along with the combinators used to operate on these domains. Finally, in the Nielson's work, the combinator portion of TML was only intended to specify expressions which have a function type (the expressions should be thought of as run-time code waiting for input). But developing a single-threading analysis and proving its correctness requires the ability to discuss the run-time behavior of expressions when they are applied to arguments. To facilitate this, we added expression forms for base type values, product type values and applications. Rewriting rules for the evaluation of expressions were added (these were given only implicitly by the Nielsons), along with the extra combinator expression if(e_1, e_2), which is used to represent the intermediate step in the reduction of a cond(e_1, e_2, e_3) application.

Figure 2.3 gives the syntax of the variant of TML used in our research, Figure 2.4 gives the typing rules for expressions and Figure 2.5 lists the rewriting rule schemes. Some other differences from the version of TML described in the previous section should be noted. We still want to have the fixpoint operator available, but in keeping with the usual practice in combinator notations, we have dropped the subexpression e. Since we have added expression forms for base type and product type values, we restrict the f_i constants to being n-ary functions. As is implied by Figure 2.5, the f_i operators are assumed to be defined separately, with their own
Types:
\[ t ::= B_i \quad \text{(a run-time base type)} \]
\[ t_1 \times \cdots \times t_k \quad (k \geq 2) \]
\[ t_1 \rightarrow \cdots \rightarrow t_k \quad (k \geq 2, \text{with } \rightarrow \text{ right-associative}) \]

Expression Forms:
\[ e ::= v \quad \text{a normal form value of some base type } B_i \]
\[ (e_1, \ldots, e_k) \quad \text{a product type value } (k \geq 2) \]
\[ f_i[t_1 \rightarrow \cdots \rightarrow t_n \rightarrow t_{n+1}] \quad \text{an } n\text{-ary operator } (n \geq 1) \]
\[ \text{id}[t] \]
\[ \text{const}[t] e \]
\[ \text{fix}[t] \]
\[ \text{cond}(e_1, e_2, e_3) \]
\[ \text{if}(e_1, e_2) \]
\[ \text{curry } e \]
\[ \text{tuple}(e_1, \ldots, e_k) \quad (k \geq 2) \]
\[ \text{take}_i[t_1 \times \cdots \times t_k] \quad (k \geq 2 \text{ and } 1 \leq i \leq k) \]
\[ e_1 \mathcal{O} e_2 \]
\[ \text{apply}[t_1 \rightarrow t_2] \]

Figure 2.3: Syntax of the TML Variant

set of rewriting rules (like the Store domain operators in Figure 1.1). Finally, note that the expression forms for \( f_i \), id, const, fix, take\(_i\), and apply now include a type specification in brackets. These specifications are also part of the full version of TML, but were omitted from the earlier description because they are optional subexpressions. But the single-threading analysis is dependent on knowing the types of all subexpressions, and in order to do full type checking these expression forms must be required to include this extra information. (In the other expression forms, the type of the expression can be determined from the types of the subexpressions.)

In the remainder of this chapter the reader should assume that, unless otherwise stated, when we refer to “TML” we are referring to the version described in this
1. \( v : B_i \) if \( v \) is a value from the base type \( B_i \)
2. \( (e_1, \ldots, e_k) : t_1 \times \cdots \times t_k \) if \( e_i : t_i \) for \( 1 \leq i \leq k \)
3. \( f_i[t_1 \rightarrow \cdots \rightarrow t_n \rightarrow t_{n+1}] : t_1 \rightarrow \cdots \rightarrow t_n \rightarrow t_{n+1} \)
4. \( \text{id}[t] : t \rightarrow t \)
5. \( \text{const}[t] e : t \rightarrow t' \) if \( e : t' \)
6. \( \text{fix}[t] : (t \rightarrow t) \rightarrow t \)
7. \( \text{cond}(e_1, e_2, e_3) : t_1 \rightarrow t_2 \)
   if \( e_1 : t_1 \rightarrow \text{Bool} \) and \( e_2 : t_1 \rightarrow t_2 \) and \( e_3 : t_1 \rightarrow t_2 \)
8. \( \text{if}(e_1, e_2) : \text{Bool} \rightarrow t_1 \rightarrow t_2 \) if \( e_1, e_2 : t_1 \rightarrow t_2 \)
9. \( \text{curry} e : t_1 \rightarrow t_2 \rightarrow t_3 \) if \( e : (t_1 \times t_2) \rightarrow t_3 \)
10. \( \text{tuple}(e_1, \ldots, e_k) : t \rightarrow (t_1 \times \cdots \times t_k) \)
    if \( e_i : t \rightarrow t_i \) for \( 1 \leq i \leq k \)
11. \( \text{take}_i[t_1 \times \cdots \times t_k] : (t_1 \times \cdots \times t_k) \rightarrow t_i \)
    if \( 1 \leq i \leq k \)
12. \( e_1 \square e_2 : t_1 \rightarrow t_3 \)
    if \( e_1 : t_2 \rightarrow t_3 \) and \( e_2 : t_1 \rightarrow t_2 \)
13. \( \text{apply}[t_1 \rightarrow t_2] : ((t_1 \rightarrow t_2) \times t_1) \rightarrow t_2 \)
14. \( (e_1, e_2) : t_2 \)
    if \( e_1 : t_1 \rightarrow t_2 \) and \( e_2 : t_1 \)

Figure 2.4: Types of TML Expressions
1. \((f_i[t] x_1 \cdots x_n)\)
   will rewrite if \(t = t_1 \rightarrow \cdots \rightarrow t_n \rightarrow t_{n+1}\),
   \(\forall i, 1 \leq i \leq n, x_i : t_i\), and the application matches the
   left hand side of one of the rewriting rules that define \(f_i\).

2. \((id[t] x) \Rightarrow x\)

3. \((\text{const}[t] e x) \Rightarrow e\)

4. \((\text{fix}[t] x) \Rightarrow x (\text{fix}[t] x)\)

5. \((\text{cond}(e_1, e_2, e_3) x) \Rightarrow ((\text{if}(e_2, e_3) (e_1 x)) x)\)

6. \((\text{if}(e_1, e_2) \text{true} x) \Rightarrow (e_1 x)\)

7. \((\text{if}(e_1, e_2) \text{false} x) \Rightarrow (e_2 x)\)

8. \((\text{curry} e x y) \Rightarrow (e (x, y))\)

9. \((\text{tuple}(e_1, \ldots, e_k) x) \Rightarrow \{(e_1 x), \ldots, (e_k x)\}\)

10. \((\text{take}[i] [t_1 \times \cdots \times t_k] (x_1, \ldots, x_k)) \Rightarrow x_i\)

11. \((e_1 \square e_2 x) \Rightarrow (e_1 (e_2 x))\)

12. \((\text{apply}[t_1 \rightarrow t_2] [f, x]) \Rightarrow (f x)\)

   \text{Figure 2.5: TML Rewriting Rules}
section, not to the full TML as defined by the Nielsons.

The Single-Threading Criteria

Before describing the single-threading analysis, we first introduce some terminology. For a base type $S$, we let $\text{nfs} S$ stand for the set of normal forms of type $S$.

**Definition 1**  
Let $S$ be an arbitrary base type. A type $t$ is $S$-typed iff $t = S$, or $t = t_1 \times \cdots \times t_k$ and $\exists i$, $1 \leq i \leq k$, such that $t_i$ is $S$-typed. An expression $e$ is $S$-typed iff $e : t$ and $t$ is $S$-typed.

Some of the TML expressions are combinators which contain other TML expressions as proper subexpressions. We will refer to these as *higher order combinators*, abbreviated as $\text{hoc}$. Specifically, these are the $\text{const}$, $\text{cond}$, $\text{if}$, $\text{curry}$, $\text{tuple}$, and $\Box$ (composition) combinators. The intent of the nesting of subexpressions in these combinators is to allow control of the stage in a reduction at which the embedded subexpressions will come into play. For example, the $\text{tuple}$ combinator contains several function subexpressions, all of which are to be applied to the same argument. The rewriting rule for $\text{tuple}$ shows that this argument must first be given to the $\text{tuple}$ expression, which then distributes the argument to the subexpressions.

To emphasize this notion, we say that an expression is *inactive* if it is a proper subexpression of a $\text{hoc}$; otherwise it is *active*. A redex in our system is an active expression which matches the left-hand side of one of the rewriting rules and in which all the proper subexpressions are in normal form. The requirement that a redex must be an active expression enforces the desired behavior of the $\text{hoc}$ expressions.
Acceptable \( f_i \) Operators

While our primary concern is with developing single-threading criteria for the TML combinator notation, our definitions and proofs must also take into account the behavior of the \( f_i \) operators. We make some reasonable simplifying assumptions about the form of the rewriting rules that define them, and also define certain further restrictions which the rewriting rules must meet for the operator to be considered acceptable for use in a single-threaded expression. These assumptions and restrictions will enable us to prove that expressions that satisfy our criteria have the single-threading property, and also that a global variable implementation of the single-threaded expression will faithfully preserve the reduction behavior of the original expression.

As we saw in the example in Figure 1.1, operations on the basic semantic domains in a denotational semantics definition are typically defined in a different style and notation than the semantic valuation functions. This is the role of the \( f_i \) operators in TML. An underlying semantic algebra is used as the representation for the domain, and the definitions of the operations on the domain make use of this algebra.

For example, say that we wish to define a domain that represents a program store, which maintains a record of identifier-value bindings. We might choose an array indexed by the identifier set as the representation for objects of type \( Store \), and the definitions for the \textit{access} and \textit{update} operators would make use of some amount of notation associated with array objects in order to define the effect of the \( Store \) operations. We might write the definitions as:

\[
\text{access} : \text{Identifier} \rightarrow \text{Store} \rightarrow \text{Store}
\]
access $i s \Rightarrow s[i]$

update : Identifier $\rightarrow$ Nat $\rightarrow$ Store $\rightarrow$ Store

update $i n s \Rightarrow s'$ where $s'$ is the result of the assignment $s[i] \leftarrow n$

Since the notation to describe the array assignment underlying the update operation does not fit smoothly into the usual rewriting rule notation, we use a new variable $s'$ which does not appear among the left hand side arguments, and an auxiliary description which describes how $s'$ is derived from those arguments.

Having the auxiliary description standing apart from the rewriting rule for update causes no problem for our single-threading analysis. The fact that the variables of type Store on the left hand and right hand sides of the rule are different, and that there is only one variable of this type on each side, conveys all the information we need. Indeed, the fact that the references to the underlying array representation have been removed from the rewriting rule gives us the proper level of abstraction. When trying to determine if the rules are single-threaded in Store, we want to view Store as a primitive type, and concern ourselves with how Store objects are manipulated by the rewriting process, not with the details of how access and update are actually implemented. In the access rule given above, the presence of the array notation $s[i]$ would require our single-threading criteria to take that notation into account in some way. This would create a problem, in that the detection of single-threading in some type $S$ would depend to some extent on the implementation of the type, which we want to avoid. It would be better for our purposes if the access rule was written in the style of the update rule:

access $i s \Rightarrow n$ where $n = s[i]$
In keeping with these observations, we will view each rule for an operator \( f_i[t_1 \rightarrow \cdots \rightarrow t_{n+1}] \) as consisting of both a rewriting rule and zero or more auxiliary descriptions. The rewriting rule will have the form \( (f_i \ a_1 \cdots a_n) \Rightarrow B \). The argument subexpressions \( a_1, \ldots, a_n \) in the left hand side have the syntax:

\[
\begin{align*}
a & ::= v & \text{a normal form value of some base type} \\
   & | x & \text{a variable, which may have any type} \\
   & | \{a_1, \ldots, a_k\} & (k \geq 2)
\end{align*}
\]

The syntax of the right hand side \( B \) is an extension of the argument syntax:

\[
\begin{align*}
B & ::= v & \text{a normal form value of some base type} \\
   & | x & \text{a variable, which may have any type} \\
   & | f_j & (j \neq i) \\
   & | \{B_1, \ldots, B_k\} & (k \geq 2) \\
   & | (B_1 \ B_2)
\end{align*}
\]

As is usual, we require that if the definition of operator \( f_i \) has more than one rewriting rule, they are mutually exclusive; that is, no redex can match more than one of the rewriting rules.

For each rewriting rule, let \( NV \) be the (possibly empty) set of variables which appear in \( B \) but not in the left hand side. For each \( x \in NV \), we will assume that there is an auxiliary description \( d_x(a_1, \ldots, a_n) = D \) which defines the value of \( x \) in terms of the arguments \( a_1, \ldots, a_n \). The exact format of the description \( D \) is unimportant to us, and we will simply assume that it is some sort of mixture of mathematical notation and text, as we used in the update and access examples. The only important property we assume \( d_x \) to have is an “equals-for-equals” property:
that if \( \forall i, 1 \leq i \leq n, e_i \) and \( e'_i \) represent the same value, then \( d_x(e_1, \ldots, e_n) \) and \( d_x(e'_1, \ldots, e'_n) \) "describe" the same value.

These descriptions will be employed later in this chapter in the proofs of correctness for the global variable transformation. But they play no part in the single-threading analysis, and will not be mentioned further in the course of that development.

We now give the requirements which an \( f_i \) operator must satisfy in order to be acceptable for use in a single-threaded expression.

**Definition 2** An operator \( f_i[t_1 \rightarrow \cdots \rightarrow t_{n+1}] \) is acceptable (with respect to a base type \( S \)) iff

1. \( \forall i, 1 \leq i \leq n - 1, t_i \) is not \( S \)-typed.
2. If \( t_{n+1} \) is \( S \)-typed then \( t_n \) is \( S \)-typed.
3. For each rewriting rule \( (f_j \ a_1 \cdots a_n) \Rightarrow B \),
   (a) If an operator \( f_j \) appears as a subexpression of \( B \), then \( f_j \) is acceptable.
   (b) If \( B \) is \( S \)-typed, then if \( B \) contains multiple, disjoint subexpressions of type \( S \), they are all the same variable or normal form value of type \( S \).
   (c) If \( B \) is not \( S \)-typed, then every subexpression of type \( S \) in \( B \) must be a subexpression of \( a_n \).
   (d) No subexpression of \( a_n \) can be a normal form value of type \( S \).
4. Any expression \( (f_i \ e_1 \cdots e_n) \), where \( \forall i, 1 \leq i \leq n, e_i \) is in normal form and \( e_n \) is \( S \)-typed, must match the left-hand side of one of \( f_i \)'s rewriting rules.
Part 3(d) of the definition is motivated by the global variable transformation we wish to apply. Suppose we had two rewrite rules for some \( f_j \) in which argument \( a_n \) is of type \( S \), and one rule uses a variable in this position while the other uses a particular normal form value of type \( S \). In the global variable transformation, we would replace both of these arguments by the () global variable marker, and then the possibility arises that both rules can match the same redexes.

The other parts of the definition are designed to make the \( f_j \) operators behave in a single-threaded manner. Their purpose should become clear as they are used in the correctness proofs of the single-threading criteria.

\( s_0 \)-trivial Sets

When applying Schmidt's single-threading criteria to typed \( \lambda \)-calculus expressions to test for single-threading in a type \( S \), there are times when one must verify that some or all subexpressions of type \( S \) are simply variables of type \( S \), and in fact are the same variable. This is done to ensure that all of the relevant subexpressions refer to a unique object of type \( S \), which is a necessary property for the correctness of the global storage transformation we seek to make.

In dealing with our combinator notation, the situation is complicated by the absence of variables representing the run-time object whose manipulation we are trying to analyze. For example, in the equations in Figure 2.2, there is no variable representing the Store argument on which these functions operate. Another complication is the frequent use of combinators such as \( \text{id} \), which simply passes its argument through unchanged, or \( \text{take}_i \), which projects one element out of a product type argument. We can think of such expressions as being "trivial" with respect to their arguments,
in the sense that they use an argument without changing it.

In developing our single-threading criteria, it is useful to be able to recognize expressions with this property. Such expressions also play an important role in proofs of the correctness of our criteria, in which we must show that when expressions which satisfy the criteria are evaluated, the evaluation will proceed in a way that enables us to safely use our global storage strategy. What proves useful to our development is the idea of a set of expressions which has the property that (i) all normal form subexpressions of the type $S$ which appear in the expressions are a particular normal form value from $S$, and (ii) that the set is closed under rewriting. This motivates the following definition:

**Definition 3** Let $S$ be a base type, and let $s_0 \in nfS$. A set $U \subseteq Expression$ is said to be $s_0$-trivial if:

1. $s_0 \in U$;

2. if expression $e \in U$, then

   (a) all active normal form subexpressions of type $S$ in $e$ are $s_0$;

   (b) $e \Rightarrow e'$ implies that $e' \in U$.

In our single-threading analysis, we must be able to identify whether an expression belongs to an $s_0$-trivial set for a given $s_0$. To enable us to do this, we define the set $T_{s_0}$ for $s_0 \in nfS$, and then show that such a set is $s_0$-trivial.

**Definition 4** Let $S$ be a base type and let $s_0 \in nfS$. The set $T_{s_0}$ is defined inductively:

1. (a) $s_0 \in T_{s_0}$
(b) if \( T \) is a base type, \( T \neq S \), and \( t \in nfT \), then \( t \in T_{S0} \)

2. \((e_1, \ldots, e_k) \in T_{S0} \) iff \( \forall i, 1 \leq i \leq k, e_i \in T_{S0} \).

3. \( f_i[t_1 \to \cdots \to t_{n+1}] \in T_{S0} \) iff

   (a) \( f_i \) is acceptable

   (b) if \( t_{n+1} \) is \( S \)-typed, then in each of \( f_i \)'s rewriting rules (each of the form

   \((f_i \ a_1 \cdots a_n) \Rightarrow B\)) every subexpression of type \( S \) in \( B \) is a subexpression

   of \( a_n \).

4. \( \text{id}[t] \in T_{S0} \) for all types \( t \).

5. \( \text{const}[t] e \in T_{S0} \) iff \( e \in T_{S0} \).

6. \( \text{fix}[t] \in T_{S0} \) iff \( t \) is not \( S \)-typed.

7. \( \text{cond}(e_1, e_2, e_3) \in T_{S0} \) iff \( \forall i, 1 \leq i \leq 3, e_i \in T_{S0} \).

8. \( \text{if}(e_1, e_2) \in T_{S0} \) iff \( e_1 \in T_{S0} \) and \( e_2 \in T_{S0} \).

9. \( \text{curry} e \in T_{S0} \) iff \( e \in T_{S0} \).

10. \( \text{tuple}(e_1, \ldots, e_k) \in T_{S0} \) iff \( \forall i, 1 \leq i \leq k, e_i \in T_{S0} \).

11. \( \text{take}_i[t_1 \times \cdots \times t_k] \in T_{S0} \) for all types \( t_i \).

12. \( e_1 \square e_2 \in T_{S0} \) iff \( e_1 \in T_{S0} \) and \( e_2 \in T_{S0} \).

13. \( \text{apply}[t_1 \to t_2] \in T_{S0} \) for all types \( t_1 \) and \( t_2 \).

14. \( (e_1 \ e_2) \in T_{S0} \) iff \( e_1 \in T_{S0} \) and \( e_2 \in T_{S0} \).
Before giving the proof that the $T_{s_0}$ set is $s_0$-trivial, we first note that $T_{s_0}$ is actually a very large subset of all TML expressions, since the only expressions explicitly excluded by our definition are the other $nfs$ values other than $s_0$, $f_i$ operators which modify values of type $S$, certain fix operators and larger expressions built up from these expressions. There is also a large overlap between sets based on different normal form values, e.g., $T_{s_0}$ and $T_{s_1}$, since many expressions such as $id[t]$ are in every such set, regardless of the normal form value or type in question.

In the definition given later for our single-threading criteria (Definition 8) we will require that some subexpressions be checked to insure that they are in the set $T_{s_n}$ for any $s_n \in nfs$. To see that this checking is possible, note that the only part of the $T_{s_0}$ definition that actually depends on the value $s_0$ is part 1(a). So if we have an expression which satisfies the definition for $T_{s_0}$ and also contains no $nfs$ subexpressions at all, it would also satisfy the analogous definition for a set $T_{s_n}$, where $s_n$ is an arbitrary member of $nfs$.

**Lemma 1** Let $S$ be a base type and let $s_0 \in nfs$. Then the set $T_{s_0}$ defined according to Definition 4 is $s_0$-trivial.

**Proof:** We must show that the set $T_{s_0}$ has properties 1 and 2 of Definition 3. $T_{s_0}$ has property 1 by part 1(a) of Definition 4.

$T_{s_0}$ has property 2(a): Let expression $e \in T_{s_0}$. The proof is by induction on the structure of $e$. If $e$ is a base value $v$, the result follows from part 1 of Definition 4 and the fact that $e$ has no proper subexpressions. If $e = f_i[t_1 \to \cdots \to t_{n+1}]$, then since $S$ is a base type and $f_i$ is function-typed, $e \not\in S$, and because $e$ has no proper subexpressions, the result follows. If $e = id[t], fix[t], take_i[t_1 \times \cdots \times t_k]$, or
apply[@1 \rightarrow t_2], then \( e \not\in S \) and \( e \) has no proper subexpressions, and the result follows. If \( e \) is one of the hoc expressions, then \( e \not\in S \), and there are no active proper subexpressions to consider. Finally, if \( e = (e_1, \ldots, e_k) \) or \( (e_1 \ e_2) \), then \( e \not\in n_f S \) and any possible active \( n_f S \) subexpression must occur in one of the proper subexpressions \( e_i \) of \( e \). Since by Definition 4 these subexpressions must also be in \( T_{S0} \), the result follows by induction.

\( T_{S0} \) has property 2(b): In this case, we assume that \( e \in T_{S0} \) is a redex and examine the TML rewriting rules. If \( e = (f_i[t] \ x_1 \ldots x_n) \), then \( e \in T_{S0} \) implies that \( f_i[t] \) and \( x_1, \ldots, x_n \) are all in \( T_{S0} \), and the result follows from the requirement in part 3 of Definition 4 that the rewriting rule produces a contractum in \( T_{S0} \). In all the other rules, \( e \in T_{S0} \) implies that all subexpressions in the redex are also in \( T_{S0} \). Thus, in each rule, the right-hand side is built from \( T_{S0} \) expressions in a way that satisfies Definition 4, and the desired result holds.

\( S \)-consistent Expressions

The next important notion in our development is that of an \( S \)-consistent expression. This is similar to the notion of trivial expressions discussed in the last section, but less restrictive.

The notion of a trivial expression (with respect to a normal form value \( s_0 \in n_f S \)) required that all normal form values of type \( S \) in the expression be \( s_0 \), and that this property be preserved during reduction of the expression. When considered in light of the global storage implementation we hope to use for single-threaded expressions, such a trivial expression would allow for the replacement of normal form subexpressions of type \( S \) by a pointer to the global storage location, since all of these
subexpressions refer to the same normal form value. The trivial expression would also have the property that reduction of the expression would never cause us to alter the value in global storage. This property becomes useful in defining our notion of $S$-consistent expressions.

An $S$-consistent expression has the property that all active, disjoint subexpressions of type $S$, whether in normal form or not, represent the same value. The motivation behind this property is that if we perform the globalization transformation on an $S$-consistent expression and then reduce the expression, the value in the global storage may change, but not in a way that causes a conflict between disjoint global storage pointers.

A few examples will help illustrate the point. Let:

\[ e_1 = (access \ I) : Store \to Nat \]
\[ e_2 = \text{tuple}(\text{id}[S], \text{id}[S]) : Store \to (Store \times Store) \]
\[ e_3 = ((update \ I) \ 3) : Store \to Store \]

and let $s_0 \in nfS$ represent the current $Store$ value. Of these three expressions, $e_1$ and $e_2$ are both in the set $T_{s_0}$, while $e_3$ is not.

The pair expression

\[ \{(e_1 \ s_0), (e_2 \ s_0)\} \]

is considered $S$-consistent. The underlying idea is that all expressions of type $S$ which are in normal form are the same (the two instances of $s_0$), and the non-normal form subexpressions of type $S$ ($(e_1 \ s_0)$ and $(e_2 \ s_0)$) are both in $T_{s_0}$. Thus, if we think of $s_0$ as representing a pointer to a global $Store$ variable, rather than the $Store$ value itself, it makes no difference whether we reduce $(e_1 \ s_0)$ or $(e_2 \ s_0)$ first, since the one
we reduce will not change the global store, and the other one’s $s_0$ pointer will still refer to the intended store state.

However, in the expression

\[(e_1 s_0), (e_3 s_0)\]

reducing $(e_3 s_0)$ first will change the global store, and thus affect the result of the access operation. Since the "obvious" intent of the expression is that $e_1$ and $e_3$ should be applied to the same store state, we would not be able to implement this expression safely using the global storage method. To help us recognize this problem, we want to define the property of $S$-consistency in such a way that our first example will be $S$-consistent, but the second is not.

Finally, if we have expressions of type $S$ which are nested rather than disjoint, such as in

\[(update I 4 (update I 3 s_0))\]

then no conflicts will arise in the use of a global variable implementation, because only the innermost expression can be in normal form, and thus be a pointer to the global store. In this example, a call-by-value reduction strategy requires us to reduce the inner $update$ application first. This alters the global store pointed to by $s_0$, and the subexpression $(update I 3 s_0)$ is replaced by a pointer to this new store. Only then can the outer $update$ be reduced. Thus the use of the global variable correctly implements the intended semantics of the TML expression.

These ideas lead us to the following definitions, which make use of the concept of an $s_0$-trivial set:
Definition 5 Let $s_0 \in nfS$, and let $U$ be an $s_0$-trivial set. An expression $e$ is $U$-consistent iff

1. all active normal form subexpressions of type $S$ in $e$ are $s_0$.

2. if $e$ has multiple, disjoint, active subexpressions of type $S$, then they are all in $U$.

We note for later use that by this definition, any subexpression of an $U$-consistent expression is also $U$-consistent.

Definition 6 Expression $e$ is $S$-consistent iff it is $U$-consistent for some $s_0$-trivial set $U$ for some $s_0 \in nfS$.

Single-threading and s-t Expressions

As we discussed in the last section, an $S$-consistent expression has the property that it can be evaluated safely using the global variable implementation for its arguments of type $S$. Since this is the key property we have in mind when we ask whether an expression is single-threaded in $S$, we can define single-threadedness in terms of $S$-consistency.

Definition 7 Expression $e_0$ is single-threaded in $S$ iff for every reduction sequence $e_0 \Rightarrow e_1 \Rightarrow \ldots \Rightarrow e_n$, $n \geq 0$, for all $0 \leq i \leq n$, $e_i$ is $S$-consistent.

We now give sufficient criteria for determining when a TML expression will be single-threaded in a type $S$. Since satisfying the criteria implies that an expression is single-threaded, but the opposite direction does not necessarily hold, we will use the term $s$-t to refer to an expression that satisfies the criteria.
Definition 8 A TML expression is s-t (with respect to a type S) iff:

1. Any base type value v is s-t.

2. \( [e_1, \ldots, e_k] \) is s-t iff it is S-consistent and \( \forall i, 1 \leq i \leq k, e_i \) is s-t.

3. \( f_i[t_1 \rightarrow \cdots \rightarrow t_{n+1}] \) is s-t iff it is acceptable.

4. \( id[t] \) is s-t for any type t.

5. \( const[e] \) is s-t iff \( e \) is s-t and \( e \) has no active S-typed subexpressions.

6. \( fix[t] \) is s-t iff \( t \) is not S-typed.

7. \( cond(e_1, e_2, e_3) \) is s-t iff

   (a) \( \forall i, 1 \leq i \leq 3, e_i \) is s-t and has no active S-typed subexpressions.

   (b) \( e_1 \in T_{sn} \) for any \( sn \in nfS \)

8. if \( (e_1, e_2) \) is s-t iff both \( e_1 \) and \( e_2 \) are s-t and have no active S-typed subexpressions.

9. \( curry \ e \) is s-t iff

   (a) \( e \) is s-t and has no active S-typed subexpressions

   (b) \( e : (t_1 \times t_2) \rightarrow t_3 \) and \( t_1 \) is not S-typed.

10. \( tuple(e_1, \ldots, e_k) \) is s-t iff

    (a) \( \forall i, 1 \leq i \leq k, e_i \) is s-t and has no active S-typed subexpressions

    (b) if \( tuple(e_1, \ldots, e_k) : t \rightarrow (t_1 \times \cdots \times t_k) \) and \( t \) is S-typed, then
i. no proper subtype of $t$ is of the form $t'_1 \rightarrow \cdots \rightarrow t'_n \rightarrow t'_{n+1}$ with $t'_n$ $S$-typed.

ii. either

A. $k = 2$, $e_2 \in T s_n$ for any $s_n \in n f S$ and $e_1 : S \rightarrow T$ where $T$ is not $S$-typed, or

B. $\forall i, 1 \leq i \leq k, e_i \in T s_n$ for any $s_n \in n f S$

11. $\text{take}_{i}[t_1 \times \cdots \times t_k]$ is s-t for any type $t_1 \times \cdots \times t_k$.

12. $e_1 \uplus e_2$ is s-t iff both $e_1$ and $e_2$ are s-t and have no active $S$-typed subexpressions.

13. $\text{apply}[t_1 \rightarrow t_2]$ is s-t iff $t_2$ $S$-typed implies $t_1$ $S$-typed.

14. $(e_1 \ e_2)$ is s-t iff it is $S$-consistent and both $e_1$ and $e_2$ are s-t.

Most parts of the s-t criteria are fairly straightforward, primarily consisting of inductive requirements that proper subexpressions be s-t and that no $S$-typed subexpressions be hidden in the inactive subexpressions of the $hoc$ combinators. The requirements for the tuple combinator, on the other hand, are more involved and some examples will be helpful in explaining the motivation for the restrictions given.

Recall that the function application form $(e_1 \ e_2)$ is not a part of the original TML notation; it was added for the purpose of being able to study the rewriting behavior of TML expressions. In the pure TML notation, the application of a function to an argument is specified by a composition of a tuple expression (to pair the function with its argument) with an apply combinator (which performs the actual function application). For example, if we have the successor function, $\text{succ} : Nat \rightarrow Nat$, the
expression

\[
\text{apply □ tuple (const succ, id)}
\]

represents the \text{succ} function waiting to be applied to an argument. The reduction when an argument is provided will proceed:

\[
(\text{apply □ tuple (const succ, id) 5}) \Rightarrow \text{apply(tuple (const succ, id) 5)}
\]

\[
\Rightarrow \text{apply((\text{const succ 5}, (id 5))))
\]

\[
\Rightarrow \text{apply((succ, 5))}
\]

\[
\Rightarrow (\text{succ 5})
\]

\[
\Rightarrow 6
\]

Another example of this type of construction is in the semantic equation for the assignment statement given in Figure 2.2. There the \text{update} function is nested inside two \text{apply □ tuple} combinations which in the course of reduction will present it successively with its \text{Nat} and \text{Store} arguments and apply it in a curried style.

But this two-combinator pattern also presents a danger for our single-threading work, because it allows us to use \text{tuple} to pair a function with its argument, without giving it immediately to an \text{apply} combinator. The unevaluated function application can then be passed around just like any other product type value and manipulated in ways that can cause problems.

To illustrate this, consider a simplified example. Let \(s_{A=2}\) represent a \text{Store} with the identifier A bound to the value 2, let \text{updateA3 : Store \rightarrow Store} be a function which will take a \text{Store} argument and change the value of A to 3, and let \text{accessA : Store \rightarrow Nat} be a function which retrieves the value of A from the \text{Store}.
Now consider the expressions:

\[
e_0 = (\text{tuple}(\text{const } \text{updateA3}, \text{id}) s_{A=2})
\]

\[
e_1 = (\text{updateA3, } s_{A=2})
\]

\[
e_2 = (\text{accessA, } s_{A=2})
\]

Note that \(e_0\) reduces to \(e_1\), and that we can also obtain \(e_2\) from \(e_0\) by substituting \(\text{accessA}\) for \(\text{updateA3}\) in \(e_0\).

None of these expressions by itself causes a problem if we are concerned with single-threading behavior in the \(\text{Store}\) domain. All will behave nicely in the proper context. The problem comes if the subexpressions are extracted and combined in other patterns. For example, consider the expression (and reduction):

\[
(\text{tuple}(\text{apply, take}_2) e_1) \rightarrow^* (s_{A=3}, s_{A=2})
\]

In this example, we give our \(e_1\) pair as an argument to another \text{tuple} expression which then applies the \text{updateA3} function to \(s_{A=2}\) but also pulls \(s_{A=2}\) out as the second component of its result. This results in a \(\text{Store} \times \text{Store}\) pair that is not \(S\)-consistent.

The problem is not simply because \text{updateA3} is a function which changes the \(\text{Store}\) value. The expression \(e_2\) has the same form as \(e_1\) but \text{accessA} cannot change the \(\text{Store}\). But if we construct the expression

\[
(\text{tuple}(\text{apply, apply} \Box \text{tuple}(\text{const } \text{updateA3, take}_2)) e_2)
\]

we have an expression which takes our seemingly harmless \(e_2\) expression as an argument, and may reduce (depending on the order in which redexes are chosen for
rewriting) to:
\[(\text{access} \ A \ s_{A=2}), (\text{update} \ A \ s_{A=2})]\)

This expression is also not $S$-consistent, due to the fact that there are two subexpressions of type $S$, $(\text{update} \ A \ s_{A=2})$ and $s_{A=2}$, which are disjoint, and $(\text{update} \ A \ s_{A=2})$ reduces to $s_{A=3} \neq s_{A=2}$.

The real problem with these examples is that the Store is given to a function as its argument but then is used for other purposes before the function has been applied. This is related to the "immediate evaluation" property discussed in the first chapter (see page 9). The tuple combinator is the root of the problem, since it is the only combinator that can create new instances of disjoint subexpressions, which can cause $S$-consistency to fail. But we do need to allow tuple expressions which pair functions and arguments together, including Store-changing functions like update, in order to specify expressions such as the assignment semantics equation in Figure 2.2.

Our restrictions will allow $S$-changing functions to be used only in the kind of $(\text{function, argument})$ pairing that the apply combinator expects, otherwise requiring that all subexpressions of the combinator be in the $s_0$-trivial set $T_{s_0}$ of any $s_0 \in nfS$ (i.e., they are not capable of changing the current $S$ domain value). This is the intent of part 10(b) of the s-t criteria. In addition, to prevent the sort of troublesome expressions described in the examples given above, part 10(b)(i) of the s-t criteria will prevent any product type argument in which some subcomponent is an $S$-changing function from being used as an argument to another tuple expression.

These restrictions rule out many expressions which could be used with perfect safety, but also could cause $S$-consistency failure, depending on the context of their use. Since we want satisfaction of the criteria to guarantee the safety of a global
variable implementation, we must take the cautious approach. But the allowance of the form described in part 10(b)(ii)(A) does allow the use of $S$-changing functions in a controlled manner.

**Sufficiency of the s-t Criteria**

We now develop the proofs that show that the criteria of Definition 8 are indeed sufficient for the detection of the single-threading property, i.e., if an expression meets the s-t criteria, then it has the reduction sequence property described in Definition 7. In the remainder of this chapter, “s-t” should be taken to mean “s-t with respect to base type $S$” unless stated otherwise.

The first lemma shows that if an s-t expression is an n-ary function with an $S$-typed result, then its $n$th argument must be $S$-typed. If we think of $S = Store$, this can be viewed as a sort of “orderly behavior” restraint, in that a function which will produce a new store as (part of) its result must take the current store as (part of) an argument. It may still produce a new store that is unrelated to its argument, but we can view this as a modification of the store, rather than a spontaneous creation of a new store. The utility of the lemma in our proof development is that in certain cases it allows us to deduce something about type $t_n$ from our knowledge of type $t_{n+1}$, or vice versa.

**Lemma 2** Let $e$ be s-t, with type $t_1 \rightarrow \cdots \rightarrow t_n \rightarrow t_{n+1}$, where $n \geq 1$. If $t_{n+1}$ is $S$-typed then $t_n$ is $S$-typed.

**Proof:** By induction on the structure of $e$. The cases are:

1. $e$ is a base type value. Since $e$ is not function-typed, the result is trivial.
2. \( e \) is a product type value. Since \( e \) is not function-typed, the result is trivial.

3. \( e = f_i[t_1 \rightarrow \cdots \rightarrow t_{n+1}] \). Since \( e \) is s-t, \( f_i \) is acceptable, and part 2 of Definition 2 gives us the result.

4. \( e = \text{id}[t] : t \rightarrow t \). In this case \( t_n \) and \( t_{n+1} \) are identical, so the result holds.

5. \( e = \text{const}[t] e' : t \rightarrow t' \) where \( e' : t' \). For \( e \) to be s-t, \( e' \) must have no active \( S \)-typed subexpressions, so \( t' \) cannot be \( S \)-typed, and the result is trivially true.

6. \( e = \text{fix}[t] : (t \rightarrow t) \rightarrow t \). Since \( e \) is s-t, \( t \) cannot be \( S \)-typed, so the result is trivially true.

7. \( e = \text{cond}(e_1, e_2, e_3) : t_1 \rightarrow t_2 \). Since \( e \) being s-t implies that both \( e_2 \) and \( e_3 \) are s-t, and \( t_1 \rightarrow t_2 \) is also the type of \( e_2 \) and \( e_3 \), the result follows by induction.

8. \( e = \text{if}(e_1, e_2) : \text{Bool} \rightarrow t_1 \rightarrow t_2 \). Since \( e \) being s-t implies that both \( e_1 \) and \( e_2 \) are s-t, and \( t_1 \rightarrow t_2 \) is also the type of \( e_1 \) and \( e_2 \), the result follows by induction.

9. \( e = \text{curry} e' : t_1 \rightarrow t_2 \rightarrow t_3 \), where \( e' : (t_1 \times t_2) \rightarrow t_3 \). Since \( e \) s-t implies \( e' \) s-t, we assume that the result holds inductively for \( e' \), so that \( t_3 \) \( S \)-typed implies that \((t_1 \times t_2)\) is \( S \)-typed. But \( e \) s-t also requires that \( t_1 \) not be \( S \)-typed, so it must be that \( t_3 \) \( S \)-typed implies \( t_2 \) \( S \)-typed, which is the desired result.

10. \( e = \text{tuple}(e_1, \ldots, e_k) : t \rightarrow (t_1 \times \cdots \times t_k) \). Since \( e \) is s-t, so is \( e_i : t \rightarrow t_i \), for \( 1 \leq i \leq k \). By induction, we assume the result holds for each \( e_i \). Then
(\(t_1 \times \cdots \times t_k\)) \(S\)-typed implies that for some \(i\), \(1 \leq i \leq k\), \(t_i\) is \(S\)-typed, which implies that \(t\) is \(S\)-typed, and we have the result.

11. \(e = \text{take}_i [t_1 \times \cdots \times t_k] : (t_1 \times \cdots \times t_k) \to t_i\). Since \(t_i\) is a subtype of \((t_1 \times \cdots \times t_k)\), The result follows immediately from the definition of \(S\)-typed.

12. \(e = e_1 \circ e_2 : t_1 \to t_3\). Since \(e\) is \(s\)-t, so are \(e_1 : t_2 \to t_3\) and \(e_2 : t_1 \to t_2\). By induction, we assume that the result holds for \(e_1\) and \(e_2\), so \(t_3\) \(S\)-typed implies that \(t_2\) is \(S\)-typed, which in turn implies that \(t_1\) is \(S\)-typed, and we have the result.

13. \(e = \text{apply}[t_1 \to t_2] : ((t_1 \to t_2) \times t_1) \to t_2\). Since \(e\) is \(s\)-t, it is required that if \(t_2\) is \(S\)-typed, then \(t_1\) is \(S\)-typed, which means that \(((t_1 \to t_2) \times t_1)\) is \(S\)-typed, and we have the result.

14. \(e = (e_1 e_2)\). If \(e : t_1 \to \cdots \to t_n \to t_{n+1}\), then for some type \(t_0\), \(e_1 : t_0 \to t_1 \to \cdots \to t_n \to t_{n+1}\) and \(e_2 : t_0\). Since \(e\) is \(s\)-t, \(e_1\) is also \(s\)-t. By induction, we assume that the result holds for \(e_1\), and this gives us the result for \(e\).

The next lemma tells us that when an \(s\)-t expression is reduced, the expression at each step in the reduction has the property that all combinators and \(f_i\) operators in the expression are \(s\)-t.

Lemma 3 Let \(e\) be \(s\)-t and let \(e \Rightarrow e'\). Then every combinator and \(f_i\) operator occurring in \(e'\) is \(s\)-t.

Proof: Let \(f\) be a subexpression of \(e'\) that is a combinator or an \(f_i\) operator. To see that \(f\) is \(s\)-t, first notice that none of the combinator rewriting rules except the rule for \(\text{cond}(e_1, e_2, e_3)\) introduces a new combinator or \(f_i\) as part of the contractum
(the rule for \( \text{fix} \) only embeds the redex combinator in the contractum). Also, recall that we are assuming that the rewriting rules of our \( f_i \) operators do not contain combinators in their contractums. Thus there are three cases to consider for \( f \):

1. \( f \) was present as a subexpression in the original s-t expression \( e \), and thus is s-t itself.

2. \( f \) is an \( f_i \) operator which was introduced as part of the contractum when a redex \( (f_j \ a_1 \ldots a_n) \) was contracted at an earlier step in the reduction from \( e \) to \( e' \). But then we can assume inductively that the earlier \( f_j \) was s-t, and thus acceptable, and then by part 3(a) of Definition 2, \( f_j \) is also acceptable, and thus is s-t.

3. \( f \) is a if\((e_1, e_2)\) combinator which was introduced at an earlier step in the reduction from \( e \) to \( e' \) by the contraction of a cond redex. But since the cond combinator must have been s-t, and the if combinator is built from the s-t subexpressions of the cond combinator, then the if combinator must also be s-t. ■

In some of the rewriting rules, a variable in the left hand side of the rule appears more than once in the right hand side, for example:

\[
(\text{tuple}(e_1, \ldots, e_k) \ x) \Rightarrow (e_1 \ x, \ldots, (e_k \ x))
\]

In our S-consistency proofs, this could cause complications because the copies of \( x \) are disjoint. There is the possibility of a redex in which the subexpression represented by \( x \) is the application \((e_1 \ e_2)\) and both \((e_1 \ e_2)\) and \( e_2 \) are of type S. In the redex, \((e_1 \ e_2)\) and \( e_2 \) are not disjoint, but in the contractum we will have \((e_1 \ e_2)\) disjoint
from $e_2$ because of the multiple copies of the application. The next two lemmas will enable us to show that this situation cannot arise.

Lemma 4 Let $e$ be s-t, let $e \Rightarrow e'$ and let $x = (x_1, x_2)$ be a subexpression of $e'$. If $x$ is S-typed and both $x_1$ and $x_2$ are in normal form, then $x$ is a redex.

Proof: By examining the rewriting rules, it can be seen that if $x_1$ is a id, const, fix, cond, tuple, take, $e_1 \Box e_2$ or apply expression, then $x$ will be a redex. If $x$ is (if ($e_1, e_2$) true), (if ($e_1, e_2$) false) or (curry $e_1 e_2$), then $x$ is not S-typed. The final possibility is that $x = ((f_1 [t_1 \rightarrow \cdots \rightarrow t_n \rightarrow t_{n+1}] e_1 \cdots e_{k-1}) e_k)$ where $n \geq 1$. If $k < n$, then $x$ is function-typed, not S-typed. So suppose that $k = n$ and $t_{n+1}$ is S-typed. By Lemma 3, we know that $f_i$ is s-t and thus acceptable. Then part 2 of Definition 2 shows us that $t_n$ and $e_k$ must be S-typed, and part 4 of the definition shows that $x$ is a redex.  

Lemma 5 Let $e$ be s-t, let $e \Rightarrow e'$ and let $x$ be a subexpression of $e'$. Let $s_0 \in n \in S$ and let $U$ be an $s_0$-trivial set. If $x$ is $U$-consistent and in normal form, then every subexpression of type $S$ in $x$ is $s_0$.

Proof: By the TML typing rules, the only expression forms which can have type $S$ are normal form values and ($e_1 e_2$) applications. Now suppose that $x$ has a subexpression $x' = (x_1, x_2)$ that is of type $S$. Since $x$ is in normal form, $x'$, $x_1$ and $x_2$ must all be in normal form. But then by Lemma 4, $x'$ must be a redex, which is a contradiction. So all subexpressions of type $S$ in $x$ must be normal form values, and the assumption that $x$ is $U$-consistent gives us the result.

The next lemma shows us that when we are proving that $S$-consistency is preserved during the contraction of a redex, our reasoning about the consistency prop-
Lemma 6 Let e be s-t, let e \rightarrow^{*} e' and let x be a subexpression of e'. If x is in normal form and is not S-typed, then x contains no active S-typed subexpressions.

Proof: By induction on the structure of x. By the definition of S-typed, the possible type of x is restricted to three possibilities:

1. x : T, where T is a base type and T \neq S. In this case x has no proper subexpressions, so the result is obvious.

2. x : t_1 \times \cdots \times t_k, where \forall i, 1 \leq i \leq k, t_i is not S-typed. Then x = (x_1, \ldots, x_k) with x_i : t_i. By induction, we conclude that \forall i, 1 \leq i \leq k, x_i has no active S-typed subexpressions, and the result follows.

3. x : t_1 \rightarrow \cdots \rightarrow t_k, with k \geq 2. If x is some f_i or a combinator, then x has no active subexpressions. If x = (x_1 x_2), then for some type t_0, x_1 : t_0 \rightarrow t_1 \rightarrow \cdots \rightarrow t_k and x_2 : t_0. Since x is in normal form, both x_1 and x_2 are in normal form. But if this is true, then x_1 must be either:

   (a) if (e_1, e_2), in which case x_2 : \text{Bool},

   (b) \text{curry } e, in which case Lemma 3 tells us that x_1 is s-t, and so x_2 cannot be S-typed, or

   (c) (f_i [t'_1 \rightarrow \cdots \rightarrow t'_n \rightarrow t'_{n+1}] e_1 \cdots e_j). Since k \geq 2, it must be that j \leq n-2, and x_2 : t'_h where h \leq n-1. Using Lemma 3 again, we see that f_i is s-t, thus acceptable, and so by part 1 of Definition 2 we have that x_2 is not S-typed.
So neither $x_1$ nor $x_2$ is $S$-typed, and by induction we conclude that neither contains any active $S$-typed subexpressions, which gives us the result for $x$. ■

The next lemma shows that if an expression has certain properties, we can conclude that it is a member of the set $T_{S0}$. This will be useful in proving that $S$-consistency is preserved during the contraction of an $S$-typed tuple redex.

**Lemma 7** Let $s_0 \in nfS$ and let $T_{S0}$ be defined as in Definition 4. Let $e$ be an expression in which all subexpressions of type $S$ are $s_0$, all $f_i$ operators are in $T_{S0}$ and all combinators are $s$-$t$. Then $e \in T_{S0}$.

**Proof:** By induction on the structure of $e$. First we note that the three properties assumed for $e$ are also true for any proper subexpression of $e$, so the inductive hypothesis is that all proper subexpressions of $e$ are in $T_{S0}$.

1. $e$ is a base type value. The result follows immediately from our assumption that all subexpressions of type $S$ are $s_0$.

2. $e = (e_1, \ldots, e_k)$. The result follows immediately from the inductive hypothesis.

3. $e = f_i[t_1 \rightarrow \cdots \rightarrow t_{n+1}]$. The result follows immediately from our assumptions about $e$.

4. $e = \text{id}[t]$, and $\text{id}[t] \in T_{S0}$ for all types $t$.

5. $e = \text{const}[t] \cdot$. The result follows immediately from the inductive hypothesis.

6. $e = \text{fix}[t]$. By our assumptions about $e$, we know that $\text{fix}[t]$ is $s$-$t$. This requires that $t$ not be $S$-typed, which gives us that $e \in T_{S0}$. 
7. $e = \text{cond}(e_1, e_2, e_3)$. The result follows immediately from the inductive hypothesis.

8. $e = \text{if}(e_1, e_2)$. The result follows immediately from the inductive hypothesis.

9. $e = \text{curry } e$. The result follows immediately from the inductive hypothesis.

10. $e = \text{tuple}(e_1, \ldots, e_k)$. The result follows immediately from the inductive hypothesis.

11. $e = \text{take}_i[t_1 \times \cdots \times t_k]$, and $\text{take}_i[t_1 \times \cdots \times t_k] \in T_{S0}$ for all types $t_i$.

12. $e = e_1 \square e_2$. The result follows immediately from the inductive hypothesis.

13. $e = \text{apply}[t_1 \rightarrow t_2]$, and $\text{apply}[t_1 \rightarrow t_2] \in T_{S0}$ for all types $t_1$ and $t_2$.

14. $e = (e_1 e_2)$. The result follows immediately from the inductive hypothesis.

The next lemma shows that in the reduction of an $s$-$t$ expression, no reduction step will cause a previously inactive $S$-typed expression to become active. This assures us that “hidden” $S$-typed subexpressions will not be exposed during reduction, which simplifies our proofs that $S$-consistency is preserved during the reduction by allowing us to ignore inactive subexpressions in the redex which become active in the contractum.

**Lemma 8** Let $e$ be $s$-$t$, and let $e \Rightarrow e'$. Let $R$ be a redex in $e'$, and let $R \Rightarrow R'$. If $e''$ is an inactive subexpression of $R$ and an active subexpression of $R'$, then $e''$ is not $S$-typed.

**Proof:** By our assumptions about the rewriting rules for $f_i$ operators, no hoc combinators appear explicitly in the rewriting rules for any $f_i$. So if $R$ is a $f_i$ redex, any
hoc expressions in the redex must be arguments that will be bound to variables in the $f_i$'s rewriting rule, and will either appear unchanged or not appear at all in the contractum. Thus there will be no inactive subexpressions in the redex that become active in the contractum.

By examining the rewriting rules for the combinators, we can see that here, too, expressions bound to the argument variables of the redex will either appear unchanged or not appear at all in the contractum, so no inactive subexpressions in the arguments will become active in the contractum. In the cases of the rewriting rules for the hoc combinators, where we do have inactive subexpressions of the combinators becoming active in the contractum, we appeal to Lemma 3. From this we conclude that the hoc combinator in the redex is s-t, which tells us that there are no $S$-typed subexpressions hidden in the inactive subexpressions of the combinator, and we are done.

We next give two lemmas which demonstrate the relationship between the $S$-consistency properties of a redex and its contractum. Lemma 9 gives the result for non-$S$-typed redexes, and Lemma 10 gives the analogous result for redexes which are $S$-typed.

**Lemma 9** Let $e$ be s-t, let $e \Rightarrow e'$, and let $R$ be a redex in $e'$. If $R$ is not $S$-typed and is $U$-consistent for some $s_0$-trivial set $U$ for some $s_0 \in nfS$, and $R \Rightarrow R'$, then $R'$ is $U$-consistent.

**Proof:** First we note that by our assumption that $e$ is s-t, the results of Lemma 3 and Lemma 8 may be used. Specifically, this allows us to assume that every combinator and $f_i$ occurring in $R$ is s-t, and that no inactive $S$-typed subexpression in $R$ has become active in $R'$. To complete the proof, we examine the possible redex cases (the numbering of the cases corresponds to the listing of the rewriting rules in Figure 2.5).
1. \( R = (f_1 \ a_1 \cdot a_n) \), with \( f_i \) s-t and therefore acceptable. Then by part 3(c) of Definition 2, every subexpression of type \( S \) in \( R' \) must be a subexpression of \( a_n \). Since \( R \), and therefore \( a_n \), is \( U \)-consistent, and \( a_n \) must be in normal form, by using Lemma 5 we can conclude that all subexpressions of type \( S \) in \( R' \) are the same normal form value, so \( R' \) is \( U \)-consistent.

2. Since \((id[t] \ x)\) is \( U \)-consistent, so is \( x \), and \( R' \) is \( U \)-consistent.

3. Since \((const[t] \ e \ x)\) is \( U \)-consistent, so is \( e \), and \( R' \) is \( U \)-consistent.

4. Since \((fix[t] \ x)\) is \( U \)-consistent, so is \( x \). By the TML typing rules, \( x \) must have type \( t \rightarrow t \), so \( x \) is not \( S \)-typed, and by Lemma 6 we see that \( x \) has no active \( S \)-typed subexpressions. And since \( R = (fix[t] \ x) \) is not \( S \)-typed, we conclude that \( R' \) contains no active \( S \)-typed subexpressions, so \( R' \) is trivially \( U \)-consistent.

5. Since \((cond(e_1, e_2, e_3) \ x)\) is \( U \)-consistent, so is \( x \). So in examining \( R' \), the only way in which \( U \)-consistency could be violated would be if there was a conflict between two subexpressions of type \( S \) which were not disjoint in \( x \), but are now disjoint in \( R' \) because of the fact that there are two disjoint occurrences of \( x \). But by using Lemma 5 we can see that all subexpressions of type \( S \) in \( x \) are the same normal form value, and we can conclude that \( R' \) is still \( U \)-consistent.

6. Since \((if(e_1, e_2) \ true \ x)\) is \( U \)-consistent, so is \( x \), and \( R' \) is \( U \)-consistent.

7. This case is handled exactly like case 6.

8. Since \((curry e x y)\) is \( U \)-consistent, \( x \), \( y \) and \( R' \) are all \( U \)-consistent.
9. Since \( \text{tuple}(e_1,\ldots,e_k, x) \) is \( \mathcal{U} \)-consistent, so is \( x \). Since \( R \) is not \( S \)-typed, neither is \( \{ (e_1 x),\ldots,(e_k x) \} \), which means that none of the \( (e_i x) \) subexpressions is \( S \)-typed. So the only possibility for newly disjoint subexpressions of type \( S \) arises because of the \( k \) copies of \( x \), and by using Lemma 5 as we did in case 5 we can conclude that \( R' \) is \( \mathcal{U} \)-consistent.

10. Since \( \text{take}_i[t_1 \times \cdots \times t_k] (x_1,\ldots,x_k) \) is \( \mathcal{U} \)-consistent, so is \( x_i \), and the contractum is \( \mathcal{U} \)-consistent.

11. Since \( (e_1 \bowtie e_2, x) \) is \( \mathcal{U} \)-consistent, so is \( x \). Because \( R \) is not \( S \)-typed, \( R' \) as a whole is not \( S \)-typed. The subexpression \( (e_2 x) \) might be \( S \)-typed, but it is not disjoint from \( x \), so the fact that \( x \) is \( \mathcal{U} \)-consistent gives us the result.

12. Since \( \text{apply}[t_1 \rightarrow t_2] (f, x) \) is \( \mathcal{U} \)-consistent, both \( f \) and \( x \) are, and the contractum is \( \mathcal{U} \)-consistent. 

**Lemma 10** Let \( e \) be \( s \)-t, let \( e \Rightarrow e' \), and let \( R \) be a redex in \( e' \). If \( R \) is \( S \)-typed and is \( S \)-consistent, and \( R \Rightarrow R' \), then \( R' \) is \( S \)-consistent.

**Proof:** As in the previous lemma, we can assume the results of Lemma 3 and Lemma 8, and we examine the possible redex cases:

1. \( R = (f_i x_1 \cdots x_n) \). By Lemma 3 we know that \( f_i \) is \( s \)-t and therefore acceptable. From this and the fact that \( R \) is \( S \)-typed, we can conclude by parts 1 and 2 of Definition 2 that \( x_n \) is \( S \)-typed, but that \( \forall i, 1 \leq i \leq n-1, x_i \) is not \( S \)-typed. So by Lemma 6 the only active \( S \)-typed subexpressions in \( R \) are in \( x_n \). From this and part 3(b) of Definition 2, we conclude that if there are multiple, disjoint subexpressions of type \( S \) in \( R' \), they are either...
(a) all the same subexpression $x'_n$ of $x_n$, that subexpression which was substituted for the single variable allowed by part 3(b) of the acceptability definition, or

(b) all the same normal form value of type $S$.

In case (a), since $R$ is $S$-consistent, so is $x'_n$. And since $x'_n$ must be in normal form, by Lemma 5, all subexpressions of type $S$ in $x'_n$ are the same normal form value. So in either case, if there are multiple, disjoint subexpressions of type $S$ in $R'$, they are actually all the same normal form value of type $S$, and $R'$ is $S$-consistent.

2. Since $(\text{id}[t] x)$ is $S$-consistent, so is $x$, and $R'$ is $S$-consistent.

3. Since $(\text{const}[t] e x)$ is $S$-consistent, so is $e$, and $R'$ is $S$-consistent.

4. This case cannot occur, because if $(\text{fix}[t] x)$ is $S$-typed, then $t$ is $S$-typed. But then $\text{fix}[t]$ is not $s$-t, which contradicts the result of Lemma 3.

5. Since $(\text{cond}(e_1, e_2, e_3) x)$ is $S$-consistent, so is $x$. In $R'$, the only possibility for newly disjoint subexpressions of type $S$ arises because of the two disjoint occurrences of $x$, But by using Lemma 5 we can see that all subexpressions of type $S$ in $x$ are the same normal form value, and we can conclude that $R'$ is $S$-consistent.

6. Since $(\text{if}(e_1, e_2) \text{true} x)$ is $S$-consistent, so is $x$, and $R'$ is $S$-consistent.

7. This case is handled exactly like case 6.

8. Since $(\text{curry} e x y)$ is $S$-consistent, $x, y$ and $R'$ are all $S$-consistent.
9. Since \((\text{tuple}(e_1, \ldots, e_k) \ z)\) is \(S\)-consistent, so is \(x\). This means that there is some \(s_0 \in \mathcal{NF} S\) such that \(x\) is \(U\)-consistent for the \(s_0\)-trivial set \(U = T_{s_0}\). Since \(R\) is \(S\)-typed, and we have observed that the \(\text{tuple}(e_1, \ldots, e_k)\) combinator, which has the type \(t \rightarrow (t_1 \times \cdots \times t_k)\), must be \(s\)-t, Lemma 2 tells us that \(t\) and therefore \(x\) must be \(S\)-typed. This means that the combinator must satisfy parts (b)(i) and either (b)(ii)(A) or (b)(ii)(B) of the \(s\)-t definition for \(\text{tuple}\).

Now suppose that \(x\) has a subexpression of the form \(f_i[t'_1 \rightarrow \cdots \rightarrow t'_n \rightarrow t'_{n+1}]\). From part (b)(i), we can conclude that \(t'_{n+1}\) is not \(S\)-typed. Since \(f_i\) is \(s\)-t, this means that \(t'_{n+1}\) is also not \(S\)-typed, since otherwise we would have a contradiction to Lemma 2. Then since \(f_i\) is \(s\)-t and thus acceptable, \(f_i \in T_{s_0}\).

Since \(x\) is \(T_{s_0}\)-consistent and is in normal form, by Lemma 5 we can conclude that all subexpressions of type \(S\) in \(x\) are \(s_0\). And by Lemma 3, we know that all combinators in \(x\) are \(s\)-t. This gives us the necessary conditions for the use of Lemma 7, and we see that \(x \in T_{s_0}\).

To show that \(R'\) is \(S\)-consistent, we note that the only possibilities for disjoint subexpressions of type \(S\) in \(R'\) are those in the \(k\) copies of \(x\), and those of the \((e_i \ x)\) applications which are of type \(S\). Since \(x \in T_{s_0}\), the disjoint copies of \(x\) cause no conflict. Now if the \(\text{tuple}\) combinator satisfies (b)(ii)(A), then \((e_1 \ x) \not\in S\) and \((e_2 \ x) \in T_{s_0}\), so \(R'\) is \(S\)-consistent. If the combinator satisfies (b)(ii)(B), then \(\forall i, 1 \leq i \leq k, (e_i \ x) \in T_{s_0}\), and \(R'\) is \(S\)-consistent.

10. Since \((\text{take}_i [t_1 \times \cdots \times t_k] (x_1, \ldots, x_k))\) is \(S\)-consistent, so is \(x_i\), and the contractum is \(S\)-consistent.

11. Since \((e_1 \| e_2 \ x)\) is \(S\)-consistent, so is \(x\). \(R'\) as a whole is \(S\)-typed, and the
subexpression \( (e_2 \, x) \) might be \( S \)-typed, but they are not disjoint from \( x \) or from each other, so the fact that \( x \) is \( S \)-consistent gives us the result.

12. Since \( \text{apply}[t_1 \to t_2] (f, x) \) is \( S \)-consistent, both \( f \) and \( x \) are, and the contractum is \( S \)-consistent. 

We can now use these results for redex/contractum \( S \)-consistency to show that in the reduction of an \( s-t \) expression, every step in the reduction is an \( S \)-consistent expression. First we show that an \( s-t \) expression is itself \( S \)-consistent, then that this property is preserved at each reduction step.

**Lemma 11** If an expression \( e \) is \( s-t \), then \( e \) is \( S \)-consistent.

**Proof:** There are four cases, based on the possible forms of the expression \( e \).

1. \( e \) is a base type value \( v \). If \( v = s_0 \in n f S \), then \( e \) is \( \mathcal{U} \)-consistent for \( \mathcal{U} = T_{s_0} \). If \( v \notin n f S \), then \( e \) has no subexpressions of type \( S \), and is trivially \( \mathcal{U} \)-consistent for \( \mathcal{U} = T_{s_n} \), where the choice of \( s_n \in n f S \) is arbitrary. So \( e \) is \( S \)-consistent.

2. If \( e \) is \( \text{const}[t'] \), \( \text{cond}(e_1, e_2, e_3) \), if \( (e_1, e_2) \), \( \text{curry} \, e' \), \( \text{tuple} \, (e_1, \ldots, e_k) \), or \( e_1 \, \mathrel{\#} \, e_2 \), then \( e \) is not of type \( S \), and all proper subexpressions of \( e \) are inactive. Thus, \( e \) is trivially \( \mathcal{U} \)-consistent for \( \mathcal{U} = T_{s_n} \), where the choice of \( s_n \in n f S \) is arbitrary, and \( e \) is \( S \)-consistent.

3. If \( e \) is \( f_1[t_1 \to \cdots \to t_n \to t_{n+1}] \), \( id[t] \), \( \text{fix}[t] \), \( \text{take}[t_1 \times \cdots \times t_k] \), or \( \text{apply}[t_1 \to t_2] \), then \( e \) is not of type \( S \), and \( e \) has no proper subexpressions. Thus, \( e \) is trivially \( \mathcal{U} \)-consistent for \( \mathcal{U} = T_{s_n} \), where the choice of \( s_n \in n f S \) is arbitrary, and \( e \) is \( S \)-consistent.
4. If \( e \) is \( (e_1, \ldots, e_k) \) or \( (e_1 \ e_2) \), then \( e \) is \( S \)-consistent by the requirements of Definition 8. \( \blacksquare \)

**Lemma 12** Let \( e_0 \) be an \( s-t \) expression. If \( e_0 \Rightarrow e_1 \Rightarrow \cdots \Rightarrow e_n \), then \( \forall i, 0 \leq i \leq n, e_i \) is \( S \)-consistent.

**Proof:** By induction on \( i \), the number of rewriting steps. The basis, \( i = 0 \), follows directly from Lemma 11. Now suppose that \( i \geq 0 \) and that the result is true for rewriting steps \( 0, 1, \ldots, i \) and that the \( i + 1 \)th step, \( e_i \Rightarrow e_{i+1} \), is made by the reduction \( R \Rightarrow R' \) of redex \( R \) in \( e_i \). Assume that \( e_i \) (and thus \( R \)) are \( \mathcal{U} \)-consistent for \( \mathcal{U} = T_{S_0} \), where \( s_0 \in n f S \). There are two cases to consider, based on the type of \( R \):

1. \( R \) is not \( S \)-typed. By Lemma 9, the contractum \( R' \) is also \( \mathcal{U} \)-consistent. So the reduction from \( e_i \) to \( e_{i+1} \) has merely replaced one \( \mathcal{U} \)-consistent subexpression with another, and \( e_{i+1} \) is \( \mathcal{U} \)-consistent, and therefore \( S \)-consistent.

2. \( R \) is \( S \)-typed, and Lemma 10 tells us that \( R' \) will be \( S \)-consistent. If \( R' \) is in fact still \( \mathcal{U} \)-consistent then the same argument that was used in case 1 gives us the result. But it may be that \( R' \) is \( \mathcal{U}' \)-consistent, where \( \mathcal{U}' \) is a \( s_1 \)-trivial set for some \( s_1 \in n f S, s_1 \neq s_0 \). In that case, there were no active subexpressions of type \( S \) in \( e_i \) that were disjoint from \( R \). Because if there had been, then \( e_i \) being \( \mathcal{U} \)-consistent would imply that \( R \in \mathcal{U} \), and since \( \mathcal{U} \) is an \( s_0 \)-trivial set, \( R' \) would also be in \( \mathcal{U} \), which is not the case. So there are no active subexpressions of type \( S \) in \( e_{i+1} \) which are disjoint from \( R' \), which means that \( e_{i+1} \) is \( \mathcal{U}' \)-consistent and thus \( S \)-consistent. \( \blacksquare \)
Lemma 12 shows that we have the conditions required by Definition 7, so we have proved the sufficiency of our single-threading criteria for TML expressions, which we state as the following theorem:

**Theorem 1** If an expression $e$ is s-t with respect to type $S$, then $e$ is single-threaded in $S$.

**Correctness of the Global Variable Transformation**

Now that we have shown that a TML expression which satisfies our s-t criteria has the desired single-threading property, we need to show that the global variable transformation described in the examples of Chapter 1 will give a faithful implementation of the normal TML evaluation behavior.

To do this we define a modified TML rewriting system which represents a global variable implementation of the type $S$. We will call this new system GVS, for “Global Variable $S$”. We will define two functions:

\[
\text{convert} : \text{TML} \rightarrow \text{GVS}
\]

\[
\text{replace} : \text{GVS} \rightarrow \text{TML}
\]

which are used to convert expressions from one rewriting system to the other. The correctness of the implementation is demonstrated by showing that if we have a TML expression $e$ which reduces (in the TML system) to a normal form expression $e'$, then if we convert $e$ to a GVS expression $g$, reduce $g$ to a normal form GVS expression $g'$ and replace the global variable into $g'$, we also obtain the expression $e'$. This property can be expressed by the commutative diagram:
The GVS Rewriting System

The syntax of GVS expressions is the same as the syntax given for TML expressions in Figure 2.3, except that we add one new value, the () global variable access marker, to the set \( \text{nfs} \) of normal form values of the type \( S \). This is the only \( \text{nfs} \) value allowed for use in a GVS expression, the other \( \text{nfs} \) values being reserved for use in the global variable cell. The typing rules for GVS expressions are the same as those given for TML expressions in Figure 2.4.

The rewriting system for GVS works with configurations. A GVS configuration is a pair \((e, \sigma)\), where \(e\) is a GVS expression and \(\sigma\) is the global variable cell. We write \(\text{val}(\sigma)\) to represent the contents of the global variable, and \(\text{val}(\sigma) \in \text{nfs} - \{()\}\). The rewriting rules for GVS combinators are shown in Figure 2.6. These rules are a straightforward modification of the TML rules in Figure 2.5. Since none of these rewriting steps can modify the global variable, we have \(\sigma\) in both the left hand and right hand configurations. Also, since all of these rules are actually polymorphically typed rule schemes, none of the variables are replaced by the () marker. In the reduction of a GVS expression the () marker will be substituted for the appropriate variable just as any other expression would.
1. $(\langle \text{id}[i] \ x \rangle, \sigma) \Rightarrow (x, \sigma)$
2. $(\langle \text{const}[i] \ e \ x \rangle, \sigma) \Rightarrow (e, \sigma)$
3. $(\langle \text{fix}[i] \ x \rangle, \sigma) \Rightarrow (\langle x \ (\text{fix}[i] \ x) \rangle, \sigma)$
4. $(\langle \text{cond}(e_1, e_2, e_3) \ x \rangle, \sigma) \Rightarrow (\langle (\text{if}(e_2, e_3) \ (e_1 x)) \ x \rangle, \sigma)$
5. $(\langle \text{if}(e_1, e_2) \ \text{true} \ x \rangle, \sigma) \Rightarrow (\langle e_1 x \rangle, \sigma)$
6. $(\langle \text{if}(e_1, e_2) \ \text{false} \ x \rangle, \sigma) \Rightarrow (\langle e_2 x \rangle, \sigma)$
7. $(\langle \text{curry} \ e \ x \ y \rangle, \sigma) \Rightarrow (\langle (e \ (x, y)) \rangle, \sigma)$
8. $(\langle \text{tuple}(e_1, \ldots, e_k) \ x \rangle, \sigma) \Rightarrow (\langle (e_1 x), \ldots, (e_k x) \rangle, \sigma)$
9. $(\langle \text{take}_i [t_1 \times \cdots \times t_k] \ (x_1, \ldots, x_k) \rangle, \sigma) \Rightarrow (x_i, \sigma)$
10. $(\langle (e_1 \ \square \ e_2) \ x \rangle, \sigma) \Rightarrow (\langle (e_1 \ (e_2 x)) \rangle, \sigma)$
11. $(\langle \text{apply}[t_1 \rightarrow t_2] \ (f, x) \rangle, \sigma) \Rightarrow (\langle (f \ x) \rangle, \sigma)$

Figure 2.6: GVS Combinator Rewriting Rules
The rewriting rules for the $f_i$ operators will undergo more noticeable changes. Recall from our earlier discussion of the structure of $f_i$ operator definitions (see page 27) that for an operator $f_i[t_1 \to \cdots \to t_{n+1}]$, each rewriting rule, of the form $(f_i \cdot a_1 \cdots a_n) \Rightarrow B$, is accompanied by a (possibly empty) set of auxiliary descriptions $\{d_x(a_1, \ldots, a_n) = D \mid x \in NV\}$, where $NV$ is the set of variables that appear only in the right hand side of the rewriting rule. To derive the GVS $f_i$ rules from the TML rules, both the rewriting rules and their auxiliary descriptions must be taken into account. For example, in the access rule given on page 26, the $s$ variable in the left hand side should be replaced by the () marker. But there is also an occurrence of $s$ in the auxiliary description, and this represents a use of the global Store variable’s value to derive the value of the variable $n$. So the proper substitution for this occurrence of $s$ is $val(\sigma)$, and the GVS rule will be:

$$access \ i \ () \Rightarrow n \quad \text{where} \ n = val(\sigma)[i]$$

In order for an $f_i$ operator to be used in an s-t TML expression, it must satisfy the acceptability requirements of Definition 2. Since the intent of the GVS system is to represent an implementation of single-threaded expressions, we will assume that all the $f_i$ rules used meet these requirements. Given a rewriting rule $(f_i \cdot a_1 \cdots a_n) \Rightarrow B$, by part 1 of Definition 2 we know that none of $a_1, \ldots, a_{n-1}$ are S-typed, and by Lemma 6 we can conclude that no redex argument which will match against one of these $a_i$ will contain an active S-typed subexpression. So even if $a_i$ is a pattern variable, e.g., $\langle x, (y, z) \rangle$, we know that none of the variables in the pattern represents an argument of type $S$. Thus we will never need to insert a () marker in any of $a_1, \ldots, a_{n-1}$.

If $a_n$ is not S-typed, then we also know that $B$ is not S-typed (this follows
from part 2 of Definition 2), and by the same reasoning as for \( a_1, \ldots, a_{n-1} \), we can conclude that no insertions of \( () \) are needed in these subexpressions. So in this case, we simply add a \( \sigma \) on both sides to obtain our GVS rewriting rule:

\[
((f_i \ a_1 \cdots a_n), \sigma) \Rightarrow (B, \sigma)
\]

Finally, since no \( () \) substitutions were made in the rewriting rule, none of the \( a_i \) arguments used in the bodies of the \( d_x \) descriptions need to be changed, and so all of the descriptions remain as they were in the TML rule.

If \( a_n \) is \( S \)-typed, there are two types of transformations which must be made on the rewriting rule and its auxiliary descriptions. These transformations are denoted by:

\[
\begin{align*}
()\text{-sub}(e) &\equiv \text{Substitute } () \text{ for each active variable or normal form value of type } S \text{ in } e. \\
\sigma\text{-sub}(D) &\equiv \text{For each variable or normal form value } e \in D, \text{ where } e : S \text{ and } e \notin \text{NV}, \\
&\quad \text{substitute } \text{val}(\sigma) \text{ (the expression, not the actual contents of } \sigma). 
\end{align*}
\]

For a rewriting rule \((f_i \ a_1 \cdots a_n) \Rightarrow B\) with auxiliary descriptions \(d_x(a_1, \ldots, a_n) = D\) for \( x \in \text{NV} \), we first change the subexpression \( a_n \) in the left hand side of the rewriting rule to \( a'_n = ()\text{-sub}(a_n) \). For the right hand side of the rewriting rule we first set \( \text{val}(\sigma') \) equal to any variable or normal form value of type \( S \) which appears in \( B \) (we will show later that even though we allow a choice, \( \text{val}(\sigma') \) will still be uniquely defined). Then we let \( B' = ()\text{-sub}(B) \), and the new rewriting rule is:

\[
((f_i \ a_1 \cdots a'_n), \sigma) \Rightarrow (B', \sigma')
\]
To reflect the fact that the descriptions should now be expressed in terms of the value of the global variable instead of explicit subexpressions of type $S$ from the left hand side of the rewriting rule, for each $d_x$ we change $D$ to $\sigma$-sub($D$).

By reviewing the derivation of the GVS rewriting rules, keeping in mind that part 3(d) of Definition 2 requires that no acceptable $f_i$ rule in TML uses a $nfS$ value in the left hand side arguments, it should be clear that there is a one-to-one correspondence between the TML and GVS rules. This will mean that if $R$ is a TML redex, and $G = convert(R)$ is the corresponding GVS redex ($convert$ will be defined shortly), that $G$ will match the GVS rewriting rule that was derived from the TML rule that matches $R$.

The Correctness Proofs

The functions $convert$ and $replace$ which map expressions between TML and GVS are quite simple:

$convert(e) = ((s_i)/e, \sigma)$ where $s_i$ is the unique active $nfS$ value appearing in $e$, and $val(\sigma) = s_i$.

$replace((g, \sigma)) = [s_i/()]g$ where $s_i = val(\sigma)$

We assume that the substitutions $[(s_i)/s_i]$ and $[s_i/()]$ perform substitutions only on active occurrences of $s_i$ and $()$, respectively. Note that $convert$ is not well-defined if there are two or more different active $nfS$ values in $e$. But in the current context, we are concerned only with the reduction behavior of TML expressions which are s-t. Our earlier results show that these expressions will be $S$-consistent at every step of their reduction, which gives us the unique $s_i$ property necessary to use $convert$.
It should also be obvious that since our two functions only substitute () for normal form values, and vice versa, the mappings will give expressions which are structurally equivalent in terms of redexes, active/inactive subexpressions and the types of all subexpressions.

In the statements of Lemma 13 and Lemma 14 that follow, assume that the TML expression \( e \) is the result of a reduction \( e_0 \Rightarrow e \), where \( e_0 \) was s-t. This allows us to assume (by Lemma 12) that \( e \) is \( S \)-consistent.

**Lemma 13** \( \text{replace} \circ \text{convert}(e) = e \)

**Proof:** Since \( e \) is \( S \)-consistent, \( \text{convert}(e) \) is well-defined. Because the question of whether a subexpression is active is solely a question of its position within a larger expression, the substitutions used in \( \text{convert} \) and \( \text{replace} \) will produce substituted subexpressions which are still active. Then it is easy to see that

\[
\text{replace} \circ \text{convert}(e) = \text{replace}([^/()_{si}]e, \sigma) \quad \text{with} \quad \text{val}(\sigma) = s_i
\]

\[= [s_i/()]^e
\]

\[= e
\]

**Lemma 14** Let \( R \) be a redex in \( e \) and let \( \langle G, \sigma \rangle = \text{convert}(R) \). If \( \langle G, \sigma \rangle \Rightarrow \langle G', \sigma' \rangle \) then \( R \Rightarrow \text{replace}(\langle G', \sigma' \rangle) \).

**Proof:** Suppose that \( R \) is a combinator redex in TML, i.e., it matches one of the rewriting rules 2-12 in Figure 2.5. By Lemma 3 and the s-t criteria in Definition 8 we can conclude that any active \( nfs \) subexpressions in \( R \) must be contained in expressions that have matched to the variables in the rewriting rule. Since \( e \) is \( S \)-consistent, so is \( R \), so each of the \( nfs \) values are the same unique value, say \( s_i \).
From these facts and an examination of the rewriting rules we can see that the right hand side will have the same property. So if \( R \Rightarrow R' \) in TML, all \( nfs \) subexpressions in \( R' \) are also \( s_i \).

By the definition of convert, \( \langle G, \sigma \rangle \) will have () in every position that \( R \) had \( s_i \), and \( val(\sigma) = s_i \). Since the GVS rewriting rule (from Figure 2.6) that will be used is just the TML rule with \( \sigma \) added on both sides, we have \( \sigma = \sigma' \), and it is easy to see that \( \langle G', \sigma' \rangle = convert(R') \). But then \( replace(\langle G', \sigma' \rangle) = replace \circ convert(R') = R' \), and we have the result for the combinator case.

Now suppose that \( R \) is a \( f_i \) redex, \((f_i e_1 \cdots e_n)\). By part 1 of Definition 2 and Lemma 6 there are no active \( nfs \) values contained in \( e_1, \ldots, e_{n-1} \). If \( e_n \) is also not \( S \)-typed, then by Lemma 6 it has no \( nfs \) subexpressions and \( convert(R) = \langle R, \sigma \rangle \) (where \( val(\sigma) \) is undefined). Also, \( e_n \) not \( S \)-typed implies that \( R \) is not \( S \)-typed by using part 2 of Definition 2. The GVS rewriting rule in this case is just the TML rule with \( \sigma \) added on both sides, so if \( R \Rightarrow R' \) in TML, then \( \langle R, \sigma \rangle \Rightarrow \langle R', \sigma \rangle \) in GVS. Since \( R \) is not \( S \)-typed, neither is \( R' \), and by part 3(c) of Definition 2 and our earlier observation about \( e_n \), \( R' \) can contain no \( nfs \) subexpressions. Also, since \( R' \) has no type \( S \) subexpressions, the auxiliary descriptions used with the TML and GVS rules are identical, so any subexpressions in \( R' \) which are defined through the descriptions will have the same definition in \( \langle R', \sigma \rangle \). This means that \( replace(\langle R', \sigma \rangle) = R' \), and we have the result.

If \( e_n \) is \( S \)-typed, then since it is in normal form and \( S \)-consistent, by Lemma 5 every subexpression of type \( S \) in \( e_n \) is the same \( nfs \) value, say \( s_i \). This means that \( convert(R) = \langle G, \sigma \rangle \), with \( G = [(]/s_i]R \) and \( val(\sigma) = s_i \). Let \( R \Rightarrow R' \) by the TML rewriting rule \((f_i a_1 \cdots a_n) \Rightarrow B\) and let \( \langle G, \sigma \rangle \Rightarrow \langle G', \sigma' \rangle \) in GVS by the
corresponding derived GVS rule.

If $R$ (and thus $G$, $G'$ and $R'$) are not $S$-typed, then in the right hand side of the TML rewriting rule used to contract $R$, the only subexpressions of type $S$ which are allowed are subexpressions of $a_n$. Thus in $R'$ all subexpressions of type $S$ will be $s_i$. Also, from the syntax allowed for TML rewriting rule arguments, plus part 3(d) of Definition 2, we can conclude that all subexpressions of type $S$ in the right hand side of the rewriting rule must be variables, although not necessarily the same variable. In the GVS rewriting rule derived from the TML rule, the right hand side of the rule will have () in place of these variables, and $val(\sigma')$ will be equal to some one of these variables. But by the fact that all subexpressions of type $S$ in $e_n$ must be some unique $s_i$, we can see that the choice of which variable was used to set $val(\sigma')$ is immaterial; when we rewrite $(G, \sigma) \Rightarrow (G', \sigma')$ by this rule, we will have $G' = [() / s_i] R'$ and $val(\sigma') = s_i$. Finally, in the auxiliary descriptions accompanying the rules, any variable in $R'$ which was defined in terms of $s_i$ in the TML rule will be defined in terms of $val(\sigma) = s_i$ in the GVS rule, so the definitions are the same. From all this we can see that $R' = replace((G', \sigma))$ and we have the result for this case.

If $R$ (and thus $G$, $G'$ and $R'$) are $S$-typed, then it follows from part 3(b) of Definition 2 that in the right hand side of the TML rule used to contract $R$, there is a unique variable or normal form value of type $S$. Call this subexpression $s_k$. In the corresponding GVS rule, occurrences of $s_k$ will be replaced by () and $val(\sigma')$ will be set equal to $s_k$. So when we rewrite $(G, \sigma) \Rightarrow (G', \sigma')$ by this rule, we will have $G' = [() / s_k] R'$ and $val(\sigma') = s_k$. Finally, in the auxiliary descriptions accompanying the rules, any variable in $R'$ which was defined in terms of $s_i$ in the TML rule will
be defined in terms of \( \text{val}(\sigma) = s_i \) in the GVS rule, so the definitions are the same. From all this we can see that \( R' = \text{replace}(\langle G', \sigma \rangle) \) and we have shown the result for this final case. 

Using the two previous results, we now give the proof of the commutative property described by the diagram on page 58. Before doing this we must make one further assumption about the relationship between the TML and GVS rewriting systems. Because of the presence of product types, at some points during the reduction of expressions we can have disjoint redexes. This can be seen by examining the TML rewriting rule for the tuple combinator, where the contraction of the tuple redex produces \( k \) disjoint redexes. In dealing with TML by itself, it was not necessary to specify any ordering for the reduction of these new redexes. But in showing that a GVS reduction provides a faithful simulation of a TML expression, we will assume that in such a situation the two systems will evaluate the disjoint redexes in the same order, although we do not care what the particular ordering is.

**Lemma 15** Let \( e \) be a TML expression which is s-t. If \( \text{convert}(e) = \langle g, \sigma \rangle \) and \( \langle g, \sigma \rangle \overset{*}{\Rightarrow} \langle g', \sigma' \rangle \) in GVS, then \( e \overset{*}{\Rightarrow} \text{replace}(\langle g', \sigma' \rangle) \) in TML.

**Proof:** By induction on the number of rewriting steps.

**Basis** (0 steps): In this case \( \langle g', \sigma' \rangle = \langle g, \sigma \rangle \) and since \( e \) is s-t and therefore S-consistent (by Lemma 11), we use Lemma 13 to obtain

\[
\text{replace}(\langle g, \sigma \rangle) = \text{replace}(\text{convert}(e)) = \text{replace} \circ \text{convert}(e) = e
\]

Since \( e \overset{*}{\Rightarrow} e \) in 0 steps is clearly true, we have the result.

**Induction:** Assume the result holds for \( n \) rewriting steps, with \( n \geq 0 \), so that \( \langle g, \sigma \rangle \overset{R}{\Rightarrow} \langle g', \sigma' \rangle \). Suppose that \( R \) is a redex in \( \text{replace}(\langle g', \sigma' \rangle) = e' \). Since \( e \overset{R}{\Rightarrow} e' \),
by Lemma 12 we know that $e'$ is $S$-consistent, so every $nfS$ subexpression in $e'$ is the same value, say $s_i$. From this we can conclude that $g' = ([()s_i]e'$ and $val(\sigma') = s_i$.

Let $\text{convert}(R) = ([()s_i]R, \sigma')$. In TML $R$ will match some rewriting rule and there will be a reduction $R \Rightarrow R'$. By the use of Lemma 9 and Lemma 10 we know that since $R$ was $S$-consistent for $s_i$, $R'$ will also be $S$-consistent for some $s_j$, where $s_i$ and $s_j$ are possibly the same. Because of the way in which the GVS rewriting rules were derived, there will be an analogous rule in GVS and we will have the reduction $\langle([()s_i]R, \sigma') \Rightarrow ([()s_j]R', \sigma'') \rangle$, where $val(\sigma'') = s_j$. Then, by Lemma 14, we know that

$$R \Rightarrow \text{replace}([([()s_j]R', \sigma'')] = [val(\sigma'')/s_j]R' = R'$$

in TML, so $e' \Rightarrow e'' = [R/R']e'$.

Now, using the same GVS reduction step, and recalling that $g' = ([()/s_i]e'$, the matching step in the GVS reduction can be expressed as

$$\langle g', \sigma' \rangle \Rightarrow \langle ([()s_j]R' / ([()s_i]R)[()s_i]e', \sigma'') \rangle = \langle ([()s_j][()s_i][R'/R]e', \sigma'') \rangle$$

To finish our result, we need to show that

$$[R/R']e' = e'' = \text{replace}\langle([()s_j][()s_i][R'/R]e', \sigma''\rangle) = [s_j/()]([()s_j][()s_i][R'/R]e'$$

The first and last of these expressions are unequal only if $s_i \neq s_j$ and $e'$ contains an occurrence of $s_i$ which is disjoint from $R$. But if the latter is true, this would imply, since $e'$ is $S$-consistent, that $R$ is a member of a $s_i$-trivial set. And then, by part 2 of Definition 3, $R'$ is a member of the same set, and so it must be that $s_i = s_j$, and this gives us the result. □
As was discussed earlier, mapping expressions from one system to the other using the `convert` and `replace` functions preserves their redex and active/inactive subexpression structure, and the derived GVS rewriting rules are also structurally equivalent to their corresponding TML rules. So it is clear that for the expressions $e$ and $convert(e)$, their respective TML and GVS reductions will exhibit the same termination behavior. Combined with the commutativity result, this shows us that the global variable transformation gives a faithful implementation of the normal TML reduction behavior:

**Theorem 2** If $e$ is a TML expression which is $s$-t, then $e \triangleright e'$, $e'$ in normal form, iff $convert(e) \triangleright \langle g, \sigma \rangle$, $\langle g, \sigma \rangle$ is in normal form and $replace((g, \sigma)) = e'$. 
CHAPTER 3. GENERALIZED COMBINATOR NOTATIONS

Introduction

In this chapter we extend the ideas underlying the work on single-threading analysis for TML to the problem of arbitrary combinator sets and their associated rewriting rules.

The single-threading criteria which are developed are more general, in the sense that we no longer assume a partially fixed set of combinators and rewriting rules such as we had in the TML work. But this generality also has a tradeoff, in that the new criteria cannot be applied to polymorphically typed expressions, which was allowed in the case of TML. The final section of this chapter will discuss these issues further.

The Combinator Notation

We will work with a generalized typed combinator language, which is described in Figure 3.1. The only types allowed are primitive types and function types, and the expressions consist only of constants, combinator expressions and applications. Combinator expressions \( f[e_1, \ldots, e_k] \) where \( k > 0 \) are called higher-order combinators (hoes). These kinds of combinators appear in the TML and CAML combinator sets (e.g., the curry or tuple combinators in TML). We say that an expression is inactive if it appears inside the brackets of a hoc; otherwise, it is active. If \( k = 0 \), we say that
Types:
\[ t ::= p \quad \text{(a primitive type)} \]
\[ t_1 \rightarrow \cdots \rightarrow t_k \quad (k \geq 2, \text{with } \rightarrow \text{ right-associative}) \]

Expression Forms:
\[ e ::= c \quad \text{(a constant)} \]
\[ f[e_1, \ldots, e_k] \quad \text{(a combinator } f, \text{ with } k \geq 0) \]
\[ (e_1 \ e_2) \quad \text{(application)} \]

Expression Types
\[ c : p \quad \text{for some primitive type } p \]
\[ f[e_1, \ldots, e_k] : t_1 \rightarrow \cdots \rightarrow t_n \rightarrow t_{n+1} \quad \text{where } n \geq 1, \text{ and the type of the combinator may depend on the types of } e_1, \ldots, e_k \]
\[ (e_1 \ e_2) : t_2 \quad \text{where } e_1 : t_1 \rightarrow t_2 \text{ and } e_2 : t_1 \]

Figure 3.1: The Combinator Language

\[ f[\ ] \] is a first-order combinator (foc) and write it as just \( f \).

A rewriting rule for a combinator has the form:

\[ f[x_1, \ldots, x_k] \ a_1 \ldots a_n \Rightarrow B \]

where \( k \geq 0, n > 0, \) each \( x_i \) is a typed variable name, and each \( a_i \) may be a typed variable name or a constant. No variable name may be mentioned twice in the left hand side of a rule, and any variable used in the right hand side must also appear on the left hand side. If a combinator \( f \) has multiple rewriting rules, the values for \( k \) and \( n \) must be the same for each of \( f \)'s rules.

We assume that some notion of redex is defined. This definition must include the requirement that a redex is an expression:

\[ f[e_1, \ldots, e_k] \ v_1 \ldots v_n \]

that matches the left hand side of a rewriting rule, and we do not permit an inactive
expression to be a redex, since in our framework the purpose of a *hoc* is to protect inactive expressions from reduction in order to allow us some control over the reduction sequence. But for the moment we place no further restrictions on the form of a redex; as we shall see, different definitions of "redex" will have an effect on the outcome of our single-threading analysis.

As usual, we define the *contractum* of a redex to be the expression that results from the substitution into $B$ of those expressions that match variables in the left hand side of the rule. Let a *context* $C[\ ]$ be an expression with a "hole" in it [3]. If redex $r$ contracts to $r'$, then an expression $e = C[r]$ reduces to $e' = C[r']$, which we write $e \Rightarrow e'$. We use the usual notions of *reduction sequence*, *stage of a reduction*, and *normal form* [3].

For a primitive type $S$, let $nfS$ be the set of normal forms of $S$. We will assume that for any primitive type $S$, the cardinality of $nfS$ is greater than one.

As in Chapter 2, we will use the concepts of $s_0$-trivial sets and $U$-consistent and $S$-consistent expressions, and the notion that the single-threading property involves the preservation of $S$-consistency during expression reduction. The discussion on pages 29-36 will remind the reader of the motivations for these ideas, and we will make use of the same definitions for each (Definitions 3, 5, 6 and 7).

Since we are no longer dealing with a partially fixed set of combinators as we did in the TML work, we can give no equivalent to the definition of the $T_{s_0}$ sets in TML (Definition 4). We will simply assume that given a particular set of combinators and some normal form value $s_0 \in nfS$, we are able to define some set of expressions which satisfies Definition 3. As in the TML work, we would want these $s_0$-trivial sets to be as large as possible, since this will increase the number of single-threaded expressions
which can be successfully identified by our criteria. But there is one difference in
the use of the $s_0$-trivial sets that should be noted. In the TML s-t criteria, it was
useful to have the $T_{s_0}$ sets include many expressions which make no use of the type
$S$; for example, the combinator $\text{id}[:t \rightarrow t]$ is in $T_{s_0}$ for any $s_0 \in n\!f\!S$, regardless of what the type $t$ is. In the generalized criteria developed in this chapter, however, it is sufficient to have a definition of $s_0$-trivial sets which deals only with expressions of type $S$ or $S \rightarrow S$.

Acceptable Rewriting Rules

Since the notion of single-threadedness involves the preservation of a consistency property during expression reduction, the behavior of the rewriting rules used is a key concern. This was somewhat obscured in the TML work because of the need to deal with both fixed (the combinators) and variable (the $f_i$ operators) portions of the TML language. The acceptability requirements for $f_i$ operators (Definition 2) are clearly based on the form of the operators' rewriting rules. But what is less obvious (until one examines the sufficiency proofs) is that the s-t criteria for the combinators (Definition 8) are motivated by the associated rewriting rules, and the need to guarantee that the use of these rules will preserve $S$-consistency.

In our generalized situation, we have no fixed combinators, so our criteria will be defined in terms of the acceptability of combinators' rewriting rules. We first give a general, sufficient condition for a rewriting rule to be acceptable to a type $S$, then show that the use of an acceptable rule on an $S$-consistent redex produces an $S$-consistent contractum.

This initial acceptability definition makes use of the notion of the context in
which a redex occurs, and thus would require the use of flow analysis techniques. In the next section, we give static analysis criteria, which ignore context, and show that rewriting rules that satisfy the static criteria will also satisfy our initial acceptability definition.

Definition 9 A rewriting rule $L \Rightarrow R$ is acceptable to a type $S$ iff

1. if $L$ is of type $S$, then for every redex of the rule, if the redex is $S$-consistent the contractum is also $S$-consistent.

2. if $L$ is not of type $S$, then for every redex of the rule, if the redex is $U$-consistent for some $s_0$-trivial set $U$ for some $s_0 \in n_fS$, then

   (a) the contractum is also $U$-consistent, and

   (b) if the contractum contains an active subexpression $e$ of type $S$ and $e \notin U$,

   then no context $C[ ]$ in which the redex can appear may have an active expression of type $S$ that is disjoint from $[ ]$.

These conditions are chosen because they allow us to prove:

Theorem 3 If expression $e$ is $S$-consistent, and $e \Rightarrow e'$ by a rewriting rule that is acceptable to $S$, then $e'$ is $S$-consistent.

Proof: Let $e = C[r]$, where $r$ is a redex. Let the contractum of $r$ be $r'$. Since $e$ is $U$-consistent for some $s_0$-trivial set $U$, where $s_0 \in n_fS$, both $r$ and $C[ ]$, taken individually, are also $U$-consistent. There are two cases to consider:

1. $r$ is of type $S$: If there is an active subexpression of type $S$ in $C[ ]$ which is disjoint from $[ ]$, then since $e$ is $U$-consistent, we must have $r \in U$. But then
since $\mathcal{U}$ is an $s_0$-trivial set, $r' \in \mathcal{U}$, and by our assumption that the rewriting rule is acceptable we have that $r'$, and therefore $e'$, are both $\mathcal{U}$-consistent. If there is no active subexpression of type $S$ disjoint from $[]$, then the acceptability of the rewriting rule gives us the result directly.

2. $r$ is not of type $S$: if $C[\ ]$ has no active subexpressions of type $S$ disjoint from $[]$, then the result is immediate from the fact that the rewriting rule is acceptable to $S$ and part 2(a) of Definition 9. Now assume that there is an active subexpression of type $S$ disjoint from $[]$, and consider whether there are any active subexpressions of type $S$ in $r'$. If there are none, then the result follows from the fact that $C[\ ]$ is $\mathcal{U}$-consistent. If $r'$ contains an active subexpression of type $S$ that is not in $\mathcal{U}$, then we have a violation of part 2(b) of Definition 9, which contradicts our assumption that the rewriting rule which was used is acceptable. The remaining case is that all active subexpressions of type $S$ in $r'$ are in $\mathcal{U}$, and then since $C[\ ]$ is $\mathcal{U}$-consistent, we clearly have that $e'$ is $\mathcal{U}$-consistent. \[\qed\]

As a corollary of Theorem 3, if we have a set of combinators and rewriting rules in which all the rules are acceptable to $S$, an easy induction using the theorem shows that any expression $e_0$ which is $S$-consistent is also single-threaded in $S$.

Since the acceptability of a rewriting rule (Definition 9) is based on the notions of "$s_0$-trivial set", "redex" and "context", a given rewriting rule may be acceptable in one system but not acceptable in another. For example, consider the following typed combinators:

\[\text{pred} : M \rightarrow M\]
plus : N → N → N
exp2 : M → N → N
double : M → N → N

and the rewriting rule:

double m n ⟹ exp2 (pred m) (plus n n)

If this rule is used in a language where (i) for all \( m_0 \in nfM \), the only \( m_0 \)-trivial set is \( \mathcal{U} = \{m_0\} \), (ii) a redex is an expression which matches some rewriting rule and in which all arguments are in normal form, and (iii) the only context in which a redex of this rule can appear is the empty context \([\ ]\), then the rule is acceptable to \( M \).

On the other hand, if a redex need not have its arguments in normal form, then the double rule is not necessarily acceptable. For example, suppose we have the combinator dereference : \( M \rightarrow N \) and the redex:

\((\text{double } m_0 \ (\text{dereference } m_0))\)

The redex is \( \mathcal{U} \)-consistent for \( \mathcal{U} = \{m_0\} \), but the contractum:

\(\text{exp2 } (\text{pred } m_0) \ (\text{plus } (\text{dereference } m_0) \ (\text{dereference } m_0))\)

is not, because the subexpression \((\text{pred } m_0)\), which is of type \( M \), is disjoint from the two other occurrences of \( m_0 \), but \((\text{pred } m_0) \notin \mathcal{U}\).

Further, if a redex of the double rule can appear in a nonempty context, then acceptability might fail again. For example, let \( \text{succ} : M \rightarrow M, m_0 \in nfM, n_0 \in nfN, \) and \( C[\ ] = \text{double } m_0 [\ ] \). The expression \( C[\text{double } m_0 \ n_0] \) is \( \mathcal{U} \)-consistent for \( \mathcal{U} = \{m_0\} \), but \( C[\text{exp2 } (\text{pred } m_0) \ (\text{plus } n_0 \ n_0)] \) is not.
Due to these subtleties, studies of single-threading detection usually assume some restrictions on the rewriting system in order to simplify the analysis: first-order systems with eager evaluation from left-to-right are most common [8, 9, 12, 29]. Flow analysis techniques can be used to help identify the contexts in which a redex may appear [4, 6, 8, 9, 12, 15, 35], while other methods ignore contexts and use static analyses to enforce restrictive conditions on the rewrite rules [31, 34, 38]. In the next section, we give two such static analyses.

Static Criteria for Acceptability

The criteria given in Definition 9 for checking the acceptability of a rewriting rule will in some cases require an examination of the possible contexts in which a redex matching the rule may occur. This presents a significant problem, in that the flow analysis techniques used to identify these contexts usually take exponential time [6].

In this section we give two static analysis criteria for acceptability, one for first-order combinator sets and the other for higher-order combinator sets. Since these criteria ignore contexts, they can be checked in linear time. The drawback to the static analysis approach is that the criteria are more restrictive than before; in some cases a rewriting rule will be considered unacceptable under the static criteria even though a context analysis would show it to be acceptable under Definition 9. For example, the rule given in the previous section for the double combinator will not be considered acceptable to $M$ under the static criteria.
Criteria for First-Order Combinators

The first criteria guarantees acceptability (for a type S) for rewriting rules that use only first-order combinators and use eager evaluation for expressions of type S. Assume, for a type S, that s_0-trivial sets satisfying Definition 3 are defined. We require that a redex be an expression \((f \; e_1 \ldots e_n)\) such that

1. it matches the left hand side of a rewriting rule \(f \; a_1 \ldots a_n \Rightarrow B\)

2. for all \(1 \leq i \leq n\),
   
   (a) if \(e_i\) is of type S, then it is in normal form
   
   (b) if \(e_i\) is not of type S, then it has no active subexpressions of type S

Note that evaluation in this system is eager only in S, the type under consideration for single-threadedness; expressions of other types may be evaluated lazily. Under this definition of redex, we obtain:

**Theorem 4**  A rewriting rule \(f \; a_1 \ldots a_n \Rightarrow B\) is acceptable to S if

1. if \(B\) is of type S, then if we assume that every \(a_i\) of type S is in the same \(s_0\)-trivial set \(U\), where the choices of \(s_0 \in \text{nfS}\) and \(U\) are arbitrary, then \(B\) is S-consistent.

2. if \(B\) is not of type S, then every subexpression of type S in \(B\) is some \(a_i\).

**Proof:** We show that if a rewriting rule satisfies the conditions of the theorem, it also satisfies the conditions of our original acceptability definition (Definition 9). There are two cases to consider, based on the type of \(B\).
B is of type $S$: Suppose condition 1 of the theorem holds, and let $(f \ e_1 \cdots e_n)$ be an $S$-consistent redex which matches the rewriting rule. Then the redex is $U$-consistent for some $s_0$-trivial set $U$ for some $s_0 \in nfs$. By our requirements for redexes, every $e_i$ that is of type $S$ must be in normal form, and so each such $e_i$ must be $s_0$. Thus, for every $a_i$ of type $S$ in the rewriting rule, the corresponding $e_i$ in the redex is in the same $s_0$-trivial set, and condition 1 of the theorem tells us that the contractum will be $S$-consistent, so we have satisfied condition 1 of Definition 9.

B is not of type $S$: Suppose condition 2 of the theorem holds, and let $(f \ e_1 \cdots e_n)$ be a redex which matches the rewriting rule, with the redex $U$-consistent for some $s_0$-trivial set $U$ for some $s_0 \in nfs$. By condition 2, every subexpression of type $S$ in the contractum will be some $e_i$. By our requirements for redexes, each of these $e_i$ must be in normal form, and so they must all be $s_0$. So the contractum is also $U$-consistent, and condition 2(a) of Definition 9 is satisfied. Finally, since each $e_i \in U$, condition 2(b) of Definition 9 is vacuously satisfied, and we are done. $\blacksquare$

As an example of the difference in power between the criteria of Definition 9 and Theorem 4, consider the system of rewriting rules shown in Figure 3.2. In this system, an initial expression $(\text{exp2 } k \ 1)$ will reduce to the value $2^k$. (In these rules, both types $M$ and $N$ should be considered to be integers. We use different type names in order to analyze them separately for single-threadedness.)

Under both acceptability criteria, all of the rules will be verified as being acceptable to $N$. But if we check for acceptability to $M$, the criteria will give different answers, because of the presence of the subexpression $(\text{pred } m)$. This subexpression represents the value "$m - 1$" and we will assume that our definition of the $m_0$-trivial sets for this rule system recognizes that for all $m_0 \in M$, $(\text{pred } m_0) \Rightarrow m' \neq m_0,$
Rule set: (let $m \in M$, $n \in N$, and $f, g \in N \rightarrow N$)

- $\exp2 : M \rightarrow N \rightarrow N$
  - $\exp2 m n \Rightarrow (\text{if} \ (\text{eq0} m) \ I \ (\text{double} m)) \ n$

- $\text{double} : M \rightarrow N \rightarrow N$
  - $\text{double} m n \Rightarrow \exp2 (\text{pred} m) \ (\text{plus} n n)$

- $I : N \rightarrow N$
  - $I n \Rightarrow n$

- $\text{if} : \text{Bool} \rightarrow (N \rightarrow N) \rightarrow (N \rightarrow N) \rightarrow (N \rightarrow N)$
  - if $\text{true} f g \Rightarrow f$
  - if $\text{false} f g \Rightarrow g$

Primitive rules on type $M$:

- $\text{eq0} m \Rightarrow \ldots$
- $\text{pred} m \Rightarrow \ldots$

Primitive rule on type $N$:

- $\text{plus} n_1 n_2 \Rightarrow \ldots$

Figure 3.2: Rule Set for Computing $2^k$
and so for an \( m_0 \)-trivial set \( \mathcal{U} \), \((\text{pred } m_0) \notin \mathcal{U}\).

When checking the double rewriting rule by the criteria in Definition 9, this subexpression will cause us to consider the contexts in which a redex \((\text{double } m \ n)\) can occur. But in this rule system, the reduction of an initial expression \((\text{exp2 } k \ 1)\) with \( k > 0 \) follows the pattern:

\[
\text{exp2 } k \ 1 \Rightarrow (\text{if } (\text{eq0 } k) \ 1 \ (\text{double } k)) \ 1 \\
\Rightarrow (\text{if } \text{false} \ 1 \ (\text{double } k)) \ 1 \\
\Rightarrow \text{double } k \ 1 \\
\Rightarrow \text{exp2 } (\text{pred } k) \ (\text{plus } 1 \ 1) \\
\vdots
\]

So when a double redex does occur, it is in an empty context, so condition 2(b) of Definition 9 is satisfied and the rewriting rule is acceptable to \( M \). But when using the criteria of Theorem 4, the rule fails condition 2 because \((\text{pred } m)\) is not one of the argument subexpressions in the left hand side of the rule.

Thus far, all of the rewriting rules which we have considered have been monomorphic. Often, a functional program is written as a set of rule schemes, that is, a set of rewriting rules with polymorphic typing [13]. Rule schemes present a problem for a single-threading analysis; for example, the rule scheme:

\[
\text{comp } f^{\alpha \rightarrow \beta} \ g^{\beta \rightarrow \gamma} \ a^{\alpha} \Rightarrow g \ (f \ a)
\]

satisfies the criteria of Theorem 4 and is acceptable to type \( S \) when \( \alpha = \beta = \gamma = S \), but the rule fails to satisfy the criteria when \( \alpha = \beta = S \) and \( \gamma = T \neq S \). Thus, one instance of a rule scheme may be acceptable and another instance may not.
But we can process a collection of rule schemes in the following way. Assume that all of the rule schemes have the property that all type variables on the right hand side also appear on the left hand side, that the typing of the rule schemes conforms to the usual Hindley-Milner typing [13], and that the initial expression to be reduced is monotyped. Then we can instantiate the rule schemes with the types used by the initial expression, generating a finite set of monotyped rewriting rules, and this rule set can then be checked for acceptability. For example, for the rule schemes:

\[
\text{comp } f^{a \rightarrow b} \ g^{b \rightarrow c} \ a^a \Rightarrow g (f \ a)
\]

\[
\text{K } a^a \ b^b \Rightarrow a
\]

\[
\text{I } a^a \Rightarrow a
\]

and an initial expression:

\[
\text{comp I (comp (K 2) I) true}
\]

such that \(2 \in N\) and \(\text{true} \in B\), the resulting set of monotyped rules is:

\[
\text{comp } f^{B \rightarrow B} \ g^{B \rightarrow N} \ a^B \Rightarrow g (f \ a)
\]

\[
\text{comp } f^{B \rightarrow N} \ g^{N \rightarrow N} \ a^B \Rightarrow g (f \ a)
\]

\[
\text{I } a^B \Rightarrow a
\]

\[
\text{I } a^N \Rightarrow a
\]

\[
\text{K } a^N \ b^B \Rightarrow a
\]

These rules can then be verified, using the conditions in Theorem 4, as being acceptable to types \(B\) and \(N\).
Criteria for Higher-Order Combinators

We next consider systems with higher-order combinators. These systems frequently use arguments of higher-order type, and value-passing combinators like I and K often appear as inactive arguments to hocs. The criteria in Theorem 4 must be extended in order to take arguments like I and K into account.

For example, suppose we have variables \( f \in N \to M \), \( g \in N \to N \) and \( n \in N \) and a combinator \( I \in N \to N \). If we assume that for all \( n_0 \in nfN \) and any \( n_0 \)-trivial set \( U \), \((I \ n_0) \in U\), then the rewriting rule

\[
\text{do-this}[f, g] \ n \Rightarrow \text{do-that} \ (f \ n) \ (g \ n)
\]

is acceptable to type \( N \) when \( g \) is instantiated by I. But if \( g \) is instantiated by, say, \( \text{succ} \in N \to N \), and \((\text{succ} \ n_0) \notin U\) for any \( n_0 \)-trivial set \( U \), then the rule is not acceptable.

The need to identify operators like I, which are "trivial" with respect to any possible argument of a type \( S \), leads to the following definition:

**Definition 10** An expression \( e \in S \to S \) is an \( S \)-trivial operator iff for all \( s_i \in nfS \), \((e \ s_i) \in U\) for any \( s_i \)-trivial set \( U \).

Given a set of rewriting rules for hocs, we assume that (i) for every hoc \( f \), if \( f \) takes \( k \) inactive arguments, then each rewriting rule for \( f \):

\[
f[x_{f1}, \ldots, x_{fk}] \ a_1 \ldots a_n \Rightarrow B
\]

uses the same variable names \( x_{f1}, \ldots, x_{fk} \) for the inactive arguments, and (ii) no variable name stands for an inactive argument in the left hand sides of distinct rewriting rules for distinct combinators.
We use the following definition to identify "well-behaved" rewriting rules and expressions:

**Definition 11** For a set of rewriting rules, let $X$ be some subset of all the variables used to denote inactive arguments to hors, and assume that all the variables in $X$ represent $S$-trivial operators. Then a rewriting rule $f[x_{j1},\ldots,x_{jk}] a_1 \ldots a_n \Rightarrow B$ is $X$-$S$-safe iff

1. (a) if $B$ is of type $S$ and we assume that all $a_i$ of type $S$ are in the same $s_i$-trivial set $U$, where the choices of $s_i \in n_f S$ and $U$ are arbitrary, then $B$ is $S$-consistent.

   (b) if $B$ is not of type $S$, then all active subexpressions of type $S$ in $B$ are either some $a_i$ of type $S$ or an application $(x_{fj} a_i)$, where $x_{fj} \in X$.

2. No $x_{fi}, 1 \leq i \leq k$, is of type $S$.

3. No inactive expression in $B$ has type $S$.

4. For any occurrence of a hoc $h[e_{h1},\ldots,e_{hk}]$ in $B$, if the variable $x_{hi} \in X$, then the corresponding $e_{hi}$ is an $S$-trivial operator.

We will also say that an expression $B$ is $X$-$S$-safe if $B$ satisfies conditions 3 and 4 given above.

As in the criteria for first-order combinators, we assume that a redex is an expression that matches the left hand side of a rewriting rule, has all of its active arguments of type $S$ in normal form, and has no active arguments of type $T \neq S$ which contain an active subexpression of type $S$. 
The following theorem states the key result for systems with higher-order combinators:

**Theorem 5** Let \( f[x_1, \ldots, x_k][a_1 \ldots a_n \Rightarrow B] \) be a rewriting rule that is \( X\text{-}S\)-safe. If all redexes that match the rule are \( X\text{-}S\)-safe, then the rule is acceptable to \( S \) and produces contractums which are \( X\text{-}S\)-safe.

**Proof:** Let \( r = f[e_{f_1}, \ldots, x_{f_k}] \quad v_1 \ldots v_n \) be a redex that matches the rule, and let \( r' = [e_{f_1}/x_{f_1}]v_j/a_j B \) be its contractum. We will show that the rewriting rule satisfies the conditions of Definition 9. There are two cases, based on the type of the rule's left hand side, which is the type of \( r \):

1. The left hand side of the rule is of type \( S \): Assume that \( r \) is \( S\)-consistent. We must show that \( r' \) is also \( S\)-consistent, i.e., that there is some \( s_0 \in n f S \) and \( s_0\)-trivial set \( U \) such that \( r' \) is \( U\)-consistent. To do this, we show that \( r' \) has the two properties required by Definition 5. The proof is by induction on the structure of \( B \). The cases are:

   (a) \( B = a_i \): By our definition of redex, if \( a_i \) is of type \( S \), then the corresponding \( v_i \) that binds to \( a_i \) must be a normal form, say \( s_0 \in n f S \). Since \( r' \) is simply the normal form expression \( s_0 \), we have that \( r' \) is \( U\)-consistent for \( U = \{s_0\} \). If \( a_i \) is not of type \( S \), then the corresponding \( v_i \) that binds to \( a_i \) has no active subexpressions of type \( S \), and \( r' \) is \( S\)-consistent for any choice of \( s_i \in n f S \).

   (b) \( B = x_{f_i} \): By condition 2 of Definition 11, \( x_{f_i} \) cannot be of type \( S \), so this case cannot occur.
(c) $B = c$: Since $r' = c$ is simply a normal form constant of type $S$, $r'$ is $\mathcal{U}$-consistent for the set $\mathcal{U} = \{c\}$.

(d) $B = f'[e_f]'_1, \ldots, e_f]'_k]$: No combinator can be of type $S$, so this case cannot occur.

(e) $B = (e_1, e_2)$: In this case we have $r' = (e_1', e_2')$, where for $m \in 1..2$, $e_m' = [e_f_i/x_f_i][v_j/a_j]e_m$. By induction, we assume that $e_1'$ and $e_2'$ are both $S$-consistent, say for $nfS$ values $s_1$ and $s_2$, respectively. We must consider whether it is possible that $s_1 \neq s_2$.

We observe that by the structure of $B$ and our restrictions on redexes, any normal form value of type $S$ in $r'$ must be some constant $c$ that appeared in $B$, or be some $v_j$ that was substituted for $a_j$ in $B$. But there can be no constant of type $S$, say $s_j$, in $B$, because then by an arbitrary choice of some $s_j \neq s_i$, we can violate condition 1(a) of Definition 11, which contradicts our assumption that our rewriting rule was $X$-$S$-safe. So all of the normal form values in $r'$ come from the redex $r$, and since $r$ is $S$-consistent it must be that $s_1 = s_2$. Thus, all active normal form values of type $S$ in $r'$ are the same value, and this gives us property 1 of Definition 5. Since $e_1'$ and $e_2'$ are individually $S$-consistent, the only way in which property 2 of Definition 5 could be violated for $r'$ is if we have two disjoint subexpressions of type $S$, one contained in $e_1'$ and the other in $e_2'$, which are not both in the same $s_j$-trivial set. Since we know that all normal form values of type $S$ in $r'$ are the same, at least one of these subexpressions (call it $e_3$) would have to not be in normal form. But then, by our restrictions on redexes and condition 2 of Definition 11, we see that $e_3$
must have been present in \( B \), rather than being introduced by substitution of the redex subexpressions into \( B \). And if it was present in \( B \), then by reasoning like that used in the previous paragraph, we can produce a contradiction to our assumption that the rewriting rule is \( X-S \)-safe. So property 2 of Definition 5 also holds, and \( r' \) is \( S \)-consistent.

2. The left hand side of the rule is not of type \( S \): Assume that redex \( r \) is \( U \)-consistent for some \( s_0 \in n f S \) and \( s_0 \)-trivial set \( U \). Since the rewriting rule and redex \( r \) are both \( X-S \)-safe, and any \( a_i \) of type \( S \) in \( r \) must be in normal form, all active subexpressions of type \( S \) in \( r' \) are either \( a_i = s_0 \), or \( (z f_j \ a_i) \) where \( z f_j \) is a \( S \)-trivial operator. This means that all active subexpressions of type \( S \) in \( r' \) are in \( U \), so \( r' \) is \( U \)-consistent and condition 2(b) of Definition 9 is vacuously satisfied, which completes the proof that the rewriting rule is acceptable to \( S \).

Finally, to show that the contractum \( r' \) is an \( X-S \)-safe expression, we note that (i) any subexpression of an \( X-S \)-safe expression is also \( X-S \)-safe, and (ii) the substitution of one \( X-S \)-safe expression by another preserves \( X-S \)-safeness. Then it follows that, since both the redex \( r \) and the rewriting rule are \( X-S \)-safe, \( r' \) is also \( X-S \)-safe.

The following results are easy corollaries of Theorem 5:

**Corollary 1** If \( e \) is \( X-S \)-safe and \( S \)-consistent, and \( e \Rightarrow e' \) by an \( X-S \)-safe rewriting rule, then \( e' \) is \( X-S \)-safe and \( S \)-consistent.

**Proof:** If \( r \) is a redex in \( e \), then \( r \) is also \( X-S \)-safe and \( S \)-consistent. Since the substitution of one \( X-S \)-safe (respectively, \( S \)-consistent) subexpression by another
preserves $X$-$S$-safeness (respectively, $S$-consistency), by Theorem 5 it follows that $e'$ is $X$-$S$-safe and $S$-consistent.

**Corollary 2** Suppose we have a system in which all rewriting rules are $X$-$S$-safe. If $e_0$ is $X$-$S$-safe and $S$-consistent, then $e_0$ is single-threaded in $S$.

**Proof:** By an easy induction, using Corollary 1.

We can implement a two-pass algorithm that verifies that a set of rewriting rules are all $X$-$S$-safe for an appropriate $X$. Starting with $X = \{}$, the first pass of the algorithm attempts to verify condition 1 of Definition 11 for each of the rewriting rules, adding as many $x_j$ variables as necessary to the set $X$. Given the set $X$ built as a result of the first pass, the second pass of the algorithm verifies the remaining conditions of the definition.

Figure 3.3 shows a set of hoc rewriting rules for computing $2^k$. Say that we wish to verify that the rules are $X$-$N$-safe for an appropriate set $X$, and that the set of $N$-trivial operators is just $\{I \in N \rightarrow N\}$. The first pass of the two-pass algorithm calculates a minimal set $X$ necessary for condition 1 to hold. The rule for $\text{exp2}$ satisfies condition 1 regardless of the value of $X$. The rule for $S$ requires that $x_2$ be in $X$, because in order to satisfy condition 1(a), the right hand side of the rule must be $N$-consistent. This is only possible if $(x_2, n)$ is in the same $n_1$-trivial set as $n$, and this is only possible if $x_2$ is an $N$-trivial operator. The remaining rules satisfy condition 1 with no extra requirements on $X$. Hence, we let $X$ be the set $\{x_2\}$ and try to verify conditions 2-4.

Conditions 2 and 3 are easy to verify, since the typing of the rules makes it clear that no inactive subexpression has type $N$. Condition 4 is verified by noting that the only instantiation of $S$ is in the rule for $\text{exp2}$, and it is indeed the case that
Let \( m \in M, n \in N, x_1 \in N \rightarrow N \rightarrow N, \) and \( f, g, x_2, x_3, x_4 \in N \rightarrow N. \)

\[
\exp_2 m n \Rightarrow \text{if } [I, \text{comp } (\exp_2 \text{ (pred } m))] S[\text{plus, } I)] (\text{eq0 } m) n
\]
\[
S[x_1, x_2] n \Rightarrow x_1 n (x_2 n)
\]
\[
\text{if } [x_3, x_4] \text{ true } \Rightarrow x_3
\]
\[
\text{if } [x_3, x_4] \text{ false } \Rightarrow x_4
\]
\[
\text{comp } f g n \Rightarrow f(g n)
\]
\[
I n \Rightarrow n
\]
\[
\text{eq0 } m \Rightarrow \ldots
\]

Figure 3.3: \textit{hoc} Rules for Computing \( 2^k \)

the argument corresponding to \( x_2, I, \) is an \( N \)-trivial operator. Hence, the rules are \( \{x_2\} \cdot N \)-safe.

Given an initial expression \( e \) to be reduced (e.g., \( e = \exp_2 3 1 \)), we verify that the expression is \( \{x_2\} \cdot N \)-safe and \( N \)-consistent. This verification, as well as the two-pass algorithm itself, can be done in linear time, assuming that membership in the \( S \)-trivial operator set is decidable in linear time. And because of the result of Corollary 2, we can conclude that when reduced by this system of rewriting rules, \( e \) is single-threaded in \( N, \) in the sense of Definition 7.

Comparison to the TML Work

We have noted that our single-threading criteria for the generalized combinator notation are based on the same four underlying definitions for \( s_0 \)-trivial sets and \( U \)-consistent, \( S \)-consistent and single-threaded expressions which were used to develop the criteria for TML. Despite this fact, the TML criteria as given in Chapter 2 are
not what would be obtained if we used the criteria of this chapter as the starting point for an analysis of the TML notation.

In some cases, similar restrictions are actually enforced, but in ways that obscure the similarities. One example of this can be seen in the acceptability requirements for TML \( f_i \) operators and the s-t criteria for the curry combinator. In both cases, the TML criteria prohibit expressions with type \( t_1 \rightarrow \cdots \rightarrow t_n \rightarrow t_{n+1} \) from having \( t_i \) be \( S \)-typed for any \( i < n \). (For the moment, think of “\( S \)-typed” as meaning simply “of type \( S \).”) This was used (by way of Lemma 6) to guarantee that, in a redex, arguments which are not \( S \)-typed will contain no active \( S \)-typed subexpressions. In the work in this chapter, on the other hand, we simply assume that redexes will have an analogous property, and use this assumption in our proofs.

But in other cases there are real differences in the two sets of criteria. We note three examples:

1. In the TML s-t criteria for curry \( e : t_1 \rightarrow t_2 \rightarrow t_3 \), we require that \( t_1 \) not be \( S \)-typed. But under our generalized criteria, the rewriting rule for curry is accepted as \( X-S \)-safe without any such restriction.

2. For the TML expression \( e_1 \Box e_2 \) to be s-t, it is sufficient for \( e_1 \) and \( e_2 \) to be s-t individually. But if we look at the associated rewriting rule in the generalized framework (think of the rule as \( \text{compose}[e_1, e_2] \ x \Rightarrow (e_1 \ (e_2 \ x)) \)) with typing \( e_1 : S \rightarrow T \) and \( e_2 : S \rightarrow S \), where \( T \neq S \), then the \( X-S \)-safe definition would require \( e_2 \) to be an \( S \)-trivial operator.

3. If we make the obvious generalization of part 1(a) of our \( X-S \)-safe definition to
TML's concept of $S$-typed, then for the TML tuple rewriting rule:

$$(\text{tuple}(e_1, \ldots, e_k) \ x) \Rightarrow (e_1 \ x, \ldots, e_k \ x)$$

we would have to require that each $e_i$ be an $S$-trivial operator in order for the rule to be $X$-$S$-safe. This is a more restrictive requirement than in the TML criteria, where we allow nontrivial $S$-operators in certain limited situations.

One obvious reason for the differences in the criteria developed in Chapters 2 and 3 is the difference in the combinator notations being examined. In the TML work, the fact that a portion of the expression syntax and rewriting rules is fixed (the combinators) while the remainder (the $f_i$ operators) is variable naturally led to a two-part ($s$-$t$ expressions and acceptable $f_i$ operators) definition of the single-threading criteria. Since the fixed combinators are the basic building blocks of TML expressions, the examination of various example expressions during the course of the criteria development uncovered some common patterns of use for the combinators. Knowledge of these patterns enabled us to closely tailor the criteria to the TML language.

But in the generalized combinator notation, the lack of any fixed combinators gave a situation more like the $f_i$ operators of TML, in which the the criteria focus primarily on the form of the rewriting rules. The increased generality prevents us from making some of the fine distinctions in the criteria that could be made with the TML combinators.

Finally, our feeling is that the fact that TML has product types, while the generalized notation does not, may have affected the work. The problems discussed in pages 38-42, which led to the final formulation of $s$-$t$ criteria for the tuple combinator,
are a direct result of the fact that the rewriting rule for tuple passes multiple copies of the redex argument to the inactive subexpressions of the combinator, and that this can lead in many cases to a violation of the desired consistency properties. But we saw that because of certain desirable expression patterns which we want to allow, we had to carefully specify the s-t criteria for tuple in order to obtain useful criteria that were also sound with respect to the preservation of S-consistency. It would be useful to formalize an extension of this chapter's generalized criteria to include product types, and examine whether the new criteria become too restrictive in this case.
CHAPTER 4. IMPLEMENTATION OF THE TML CRITERIA

The PSI/DAOS System

In order to examine the effectiveness of our TML s-t criteria and the global variable transformation, we have programmed a test implementation of it, using as our starting point the PSI/DAOS compiler generation system. (PSI and DAOS are Danish acronyms for "Programming Languages and their Implementation" and "Dynamic Automatic Translator System", respectively.) This system, developed by Hanne Riis Nielson and Flemming Nielson and their students, uses TML as the semantic metalanguage of a compiler generation system.

The PSI System

The Nielsens have been studying issues concerning the automatic construction of optimizing compilers from formal language definitions [16, 17, 18, 19, 21, 24, 26], and the PSI system is meant to act as a test bed system for their work. The PSI system [22, 25] provides facilities for parsing TML expressions and type specifications and giving interpretations for the expressions and types. (The PSI system is based on the full two-level version of TML, as given in the first section of Chapter 2. In the current discussion, it is this TML to which we are referring, unless specifically stated otherwise.) The system is implemented in the Standard ML programming
language (abbreviated SML), and expressions and type specifications are parsed to internal SML values which preserve the syntactic structure of the expression or type. Another SML datatype is used to represent the interpretations of the TML types, and an evaluator function is defined which takes an argument that represents the syntactic structure of a TML expression and applies an expression interpretation to it. For example, if we want to define the program store as a run-time function from identifiers to integers, we can specify this by the type specification

$$\text{B-Ide} \rightarrow \text{B-Num}$$

The run-time type parser of PSI will convert this to the internal representation

$$\text{Srf} (\text{Srb} (\text{SrbString}), \text{Srb} (\text{SrbInt}))$$

A store value that corresponds to this type will be a SML value that contains an embedded SML function. For instance, a store in which the identifier “abc” is bound to the integer 7 (and all other identifiers are undefined) would be represented as

$$\text{Irf} (\text{fn} \ \text{Irb(} \text{IrbString} \text{“abc”) } \Rightarrow \text{Irb(} \text{IrbInt 7)})$$

An example using a TML expression would be

$$\text{tuple} (\text{const} [\text{S-Store}] \text{ f-update}, \text{id} [\text{S-Store}])$$

which PSI’s expression parser would convert to the SML value

$$\text{Sertu} ([\text{Sercn} (\text{Sef} (“f-update”), \text{notyp}), \text{rttyp} (\text{Srv} “\text{S-Store”})], \text{notyp})$$

The notyp and rttyp components of this structure represent the (incomplete) information given in the TML expression concerning the types of the subexpressions.
The particular version of TML to be used can be varied by setting system flags before compiling the SML files that define the PSI system (the full TML described in the first section of Chapter 2 is TMLm, the most general version of TML). The compile-time types and their corresponding expressions are fixed, but the run-time types and expressions can be varied, depending on what the user feels is necessary for the intended use of the system. For example, if the user wants to give a definition for a programming language in which functions are not first-class run-time data objects, a version of TML (TMLs) can be constructed in which the definition of the run-time types does not include a function space, and the curry and apply combinators are not defined. Similarly, the interpretations of compile-time types and expressions are fixed, while the run-time interpretations can be varied. The compile-time expressions are interpreted as SML functions, and both compile-time and run-time types are interpreted as SML types.

One of the most important features of the PSI system is the ability to give different interpretations for run-time expressions. In the “standard” interpretation, the interpretations are SML functions, so that the evaluated TML expression is an executable SML function. Consider just the

\[
\text{Sercn (Sef ("f-update", notyp), rtyp (Srv "S-Store"))}
\]

portion of the expression example given earlier. The interpretation of the \( f_i \) operator f-update will be an SML function (packaged within an Icft datatype constructor, in a way similar to the embedding of the store function example) which has been stored in a list of \( f_i \) interpretations, and the evaluator function will extract that function as the meaning of the Sef subexpression. The meaning of the const combinator will be specified to PSI as the function:
fn subexpr ⇒ let val Icf f = subexpr
                   in Icf (fn irt ⇒ Irf f)
                   end

The evaluator function will apply this const function to the f-update function (the argument subexpr), stripping the Icf constructor from it and embedding it (as the value f) inside the expression

Icf (fn irt ⇒ Irf f)

The result is that the meaning of the original TML expression

const [S-Store] f-update

is a function which, when applied to a run-time store argument (irt), will ignore the argument and return the f-update function (f) as its result. (The fact that f is now packaged inside a Irf constructor rather than Icf is a minor consequence of TML's two-level typing system, which allows a run-time → run-time function such as f-update to be viewed as both a compile-time piece of code and a run-time data object.)

Since evaluation under the standard interpretation produces an executable SML function, it can be viewed as specifying an interpreter for TML expressions, using SML as the intermediate language. But other types of interpretations can be defined which give different effects. In a "coding" interpretation, one can define an abstract machine by specifying an instruction set and instruction interpreter function (as a separate SML module), and by defining the interpretation of run-time TML expressions to be sequences of these instructions, the TML evaluator produces a program for the abstract machine. Other interpretations can be defined which have the effect
of performing various types of dataflow analysis. By applying different interpretations to a single TML expression, it is hoped that the effect of an optimizing compiler can be achieved. For example, once a TML expression is parsed to the internal representation, one or more dataflow interpretations could be applied. The resulting dataflow information could be used to make optimizing transformations on the expression structure, and then a coding interpretation could be applied, resulting in a more efficient executable version of the expression.

The process of using PSI to install a TML version consists of the following general steps:

1. Define the type structure of the desired TML version.

2. Define the expression structure, based on the defined types.

3. Define an interpretation for the types.

4. Define an interpretation for the expressions.

5. Apply the evaluator generator to the expression interpretation, producing an evaluator function for TML expressions.

It should be noted that these steps are mostly automatic. The user simply sets some system flags, which are used by a preprocessor to select appropriate portions from a collection of PSI files, and then feeds the resulting code to the SML compiler. Once the TML version is constructed, the user can give definitions for various \( f_i \) operators (specifying their names, types and interpretations), and use the system to evaluate TML expressions.
The DAOS System

The DAOS system [1] extends the PSI system by providing features that allow for the specification of a denotational language definition. Methods are provided to specify semantic domains, primitive operations (the \texttt{f}_i operators of TML), and source language syntax and semantics. Using these specifications, the DAOS system can compile programs in the defined language into TML expressions, which can then be evaluated by the PSI system in the manner described earlier.

For example, consider the following input to DAOS:

\begin{verbatim}
RDomain ["S-Store = B-Ide \rightarrow B-Num"];
SFunc ("f-plus", 
[B-Num * B-Num \rightarrow B-Num"]);
IFunc ("f-plus", Icft (fn Irp [x, y] => 
let val Irb (Irblnt nl) = x
val Irb (Irblnt n2) = y
in Irb (Irblnt (nl + n2))
end));

TySem ("CC", "COM", 
[S-Store \rightarrow S-Store");

Syn (3, "c = '(' c ';'; c )' | COM ';'; COM;");

Sem ("CC", "COM", "COM ; COM", 
("s-CC s-3) \# (s-CC s-1)");
\end{verbatim}

These instructions specify, in order:

- A run-time domain S-Store, implemented as a function from identifiers to integers.
- An addition operator on run-time integers, called f-plus, whose type is defined by the SFunc function, and whose meaning is defined by the IFunc function.
• A semantic function CC which maps commands from the syntactic domain COM to their meanings as S-Store \( \rightarrow \) S-Store functions.

• The syntax and semantics of command sequencing. (In the Sem definition, "#" represents TML's \( ^{\odot} \) composition combinator, and the subexpressions \( (s\text{-CC} s\text{-1}) \) and \( (s\text{-CC} s\text{-3}) \) represent the application of the semantic function CC to the first and third lexical units of the abstract syntax, which are the commands before and after the semicolon, respectively.)

DAOS is designed in such a way as to enable interactive experimentation with the language definition. The pieces of the syntax and semantics definitions are given incrementally and may be redefined interactively, and source programs in the defined language can be compiled from incomplete language definitions, as long as the program does not make use of the undefined portions of the language.

**Single-Threading and Globalization Implementations**

In order to test our single-threading criteria and global variable transformation, several changes and additions were made to the PSI/DAOS files.

Since use of the s-t criteria depends on knowing the types of certain subexpressions, a typechecker function was defined in PSI. In PSI, the parameterized datatype ('a Sex) that is used for the internal representation of parsed TML expressions provides a position to attach extra information (represented by the type variable 'a) to each subexpression. For instance, the datatype constructor representing the TML expression const \( [t] e \) is

\[
\text{Sercn of 'a Sex \ast 'a}
\]
where 'a Sex represents the subexpression e. The type of information to be held in the 'a field is defined when installing PSI, and in the version we use it is defined so as to hold whatever type information is given by the type specifications attached to some combinators (the [f] in our const example). Once an expression is parsed to the 'a Sex form, a function wfSex is applied, which propagates the partial typing information so that each subexpression is correctly typed. In our original version of PSI, obtained from the Neilsons, wfSex was left unimplemented, so we added code for it and several auxiliary functions. We also made some modifications in the parts of DAOS that use the semantic rules of a language definition to map an abstract syntax tree to a TML expression, so that now the TML expression is typechecked by wfSex before being given to PSI's evaluator function. This enables us to apply the s-t checker to individual expressions, as well as semantic rules.

In order to check a set of semantic rules for single-threading, they must also be typechecked. Since they are specified to DAOS by TML expressions, the rules are also represented internally as 'a Sex values. But an additional complication in this case is that these representations may also contain subexpressions representing arbitrary semantic or syntactic entities. For example, in the semantic rule for an assignment statement given in Figure 2.2:

\[
C[I := E] = apply \square tuple (apply \square tuple (const (update I[I]), \varepsilon[E]), id)
\]

we have the embedded subexpressions \(\varepsilon[E]\) and \(I[I]\). In the 'a Sex representation of the rule, these will be represented as compile-time applications, respectively:
Secap (Secva ("s-EE", notyp), Secva ("s-3", notyp), notyp)
Secap (Secva ("s-LEX", notyp), Secva ("s-1", notyp), notyp)

For the rule to be correctly typechecked, all of the notyp values must be replaced by valid TML types, and the type attached to the Secap constructor must be such that it will agree with the type of whatever constructor immediately surrounds it. The solution implemented is that if the first Secva is not an s-LEX application, then the string given must represent one of the semantic functions for the language which is being defined. Since the type of this function will have been specified to DAOS, we look up this type and use it as the type of both the Secap and the first Secva. If the first Secva is an s-LEX application, we assume (in the current implementation) that the Secap represents an occurrence of an identifier in the syntax, and use the base type value Scb(ScbString) as the type of the Secap and first Secva. In both cases, we use a dummy Scb(ScbUnit) type as the type of the second Secva, and our typechecker is programmed so that it will not do its usual typechecking actions within these two forms of the Secap structure (normally it would verify that the first Secva has a function type whose domain matches the type of the second Secva). (A future improvement to the system would allow for s-LEX applications which represent other syntactic units besides identifiers, e.g., integer or Boolean constants. But this will require changes in several different modules of PSI and DAOS, and has not yet been implemented.)

Functions were programmed which check whether types and expressions satisfy the definitions of S-typed types and s\(\beta\)-trivial and s-t expressions. The s-t checker function is the top-level function, and takes three arguments:

1. The (typechecked) expression to be tested
2. The type representation of the domain being tested for single-threading

3. A list of any $f_i$ operators which are not trivial with respect to values of the domain (e.g., the update operator on the Store domain)

The s-t checker makes use of the $S$-typed and $s_0$-trivial checkers to give its result, which is simply a Boolean value. To test a list of semantic rules for single-threading, we simply map the s-t checker onto the list, producing a list of Boolean values, and if all of these are true, then the semantics as a whole is single-threaded in the specified domain.

Because of the differences between full TMLm and our TML variant, which was used as the basis for the definitions, the current implementations of the $s_0$-trivial and s-t checker functions are somewhat incomplete. In order to produce a working first version of the implementation, these functions will return default "true" results or raise errors in some cases. Since the behavior of $f_i$ operators in our implementation is specified by giving a SML function rather than through rewriting rules, the s-t checker makes the assumption that all $f_i$ operators are s-t, and that they are $s_0$-trivial if they do not appear in the list of nontrivial operators mentioned above. Also, since the first examples tested were either semantic rules or program expressions in which no subexpressions of the type $S$ can appear, there is currently no implementation of the definitions of $U$-consistent and $S$-consistent expressions.

Once a semantics definition has been found to be single-threaded, say in the domain S-Store, a new semantics can be given which specifies a global variable implementation for S-Store. The only changes which need to be made in the semantics definition are:
• Use R_Domain to change the type specification of the S-Store domain. In keeping with the idea of using a () marker to represent an access right to the store, we can make the definition

\[
R\_Domain \["S-Store = B\_Unb"\]
\]

which will mean that the representation of the store at run-time will be the structured SML value \(Irb(IrbUnb())\).

• Define an SML variable of the desired type to be used as the actual global store value. For instance, if we want to use a list of (identifier, number) pairs as the implementation, we add the command

\[
\text{val storegv: } ((\text{string} \times \text{int}) \text{ list}) \text{ ref } = \text{ ref } [\ ]
\]

to the semantics file, creating an empty list variable to represent the initial store.

• Give new interpretations for any \(f_i\) operators on the S-Store domain, e.g., \(f\)-update and \(f\)-access. The new interpretations will be functions that wait for the () marker argument, but then ignore it and instead act directly on the variable storegv. In the case of an operator like \(f\)-update, which is supposed to produce a modified store as its result, the new interpretation will change the global store variable as a side-effect, then return the () marker as its result.

Although the implementation cannot make an automatic transformation to a global variable implementation (we note that, depending on the domain being checked, there could be more than one reasonable choice for the type of the global variable), the
changes the user must make in the semantics definition are relatively few. Assuming, as we did in the description above, that actual manipulation of the domain's representation is done only through $f_i$ operators, none of the semantic rules or valuation functions need be changed. Also, the interpretations of the relevant $f_i$ operators are not required in order to do the single-threading check. One could give only the types of the $f$-update and $f$-access operators, delaying the specification of their interpretations until it is determined whether the S-Store domain is indeed single-threaded.

The primary drawback of the current state of the system is that we cannot give only the modified portions of the semantics definition when making the global variable transformation. Because our implementation of the $s$-$t$ checking process modifies some of the PSI/DAOS variables in which information from the original semantics definition is stored, the user must give the system the entire semantics definition again, with the necessary globalization changes included. Since the semantics definition would normally be given as a file of SML commands, this is not really that difficult, requiring only some minor editing of the file, which is then given as input to the system again. It also appears that this problem can be eliminated by some rearrangement of the modular structure of the system.

Testing of the Implementation

We have tested our implementation on a semantics definition for a small programming language. The language defined is a more complete version of the one shown in Figures 1.1 and 2.2 (the semantics is, of course, specified in TML rather than $\lambda$-calculus).

The single-threading checker verifies that these semantic rules are single-threaded
in the Store domain, and several example programs in the language have been compiled and run using both the functional Store and two different global Store implementations. We have also tested the single-threading checker successfully on a few examples of individual semantics rules which violate the s-t criteria.

Execution timing comparisons have been made on seven example programs from the language in order to compare the efficiency of the functional and the global Store implementations. The seven programs used are shown in Figure 4.1, and a summary of the timing results is shown in Table 4.1.

The results shown are for a global implementation using an array variable to implement the store. Because SML array variables are restricted to having integer indices, and also to avoid having to use an environment to map variable identifiers to locations in the array, we used an array indexed from 0 to 9, and all identifiers used were simply the letter "v" followed by a single digit (i.e., v0, ... ,v9), with the digit serving as the index into the store array. Because our small language does not allow numeric constants, initial store configurations were constructed for the programs.

The timing figures given are the average total execution times (in seconds) for N consecutive runs of the program on the same initial store value. In programs 1 through 5, this store had variables v1 through v6 bound to the values 1 through 6, respectively, and N = 10,000. For programs 6 and 7 the initial store was the same, with the exception that v6 was initialized to 100 and 200, respectively, in order to control the number of times the while-statement would repeat. Because of their longer execution times, the timing values shown for programs 6 and 7 are total times for a smaller number of runs. The last column of the table shows the ratio of the execution times for the two different Store implementations.
1. \( v_9 := v_3 + v_5 + v_2 \).
2. \( v_7 := v_1; v_8 := v_2 \).
3. \( v_9 := v_3 + v_5 + v_2 + v_1 + v_6 + v_4 \).
4. \( v_7 := v_1; v_8 := v_2; v_9 := v_6 + v_6 + v_6 + v_6 + v_6 + v_6 \).
5. \( v_7 := v_6; v_8 := v_6; v_9 := v_6; v_7 := v_6; \\
   v_0 := v_1 + v_2 + v_3 + v_4 + v_5 + v_6 + v_7 + v_8 + v_9 \).
6. \( v_0 := v_1; \) while \( v_0 = v_1 \) do if \( v_2 = v_6 \) then \( v_0 := v_3 \) else \( v_2 := v_2 + v_1 \).
7. Same as number 6.

Figure 4.1: Test Programs for Execution Timings

<table>
<thead>
<tr>
<th>Program</th>
<th>N</th>
<th>Array Store</th>
<th>Function Store</th>
<th>Array/Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10000</td>
<td>18.63</td>
<td>17.70</td>
<td>1.05</td>
</tr>
<tr>
<td>2</td>
<td>10000</td>
<td>11.98</td>
<td>9.04</td>
<td>1.33</td>
</tr>
<tr>
<td>3</td>
<td>10000</td>
<td>36.54</td>
<td>36.53</td>
<td>1.00</td>
</tr>
<tr>
<td>4</td>
<td>10000</td>
<td>47.28</td>
<td>54.59</td>
<td>0.87</td>
</tr>
<tr>
<td>5</td>
<td>10000</td>
<td>81.57</td>
<td>99.67</td>
<td>0.82</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>31.28</td>
<td>40.64</td>
<td>0.77</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>31.54</td>
<td>50.20</td>
<td>0.63</td>
</tr>
</tbody>
</table>

Table 4.1: Execution Timing Results
The fact that the first three programs show no improvement in run-time efficiency is probably due to the small number of store operations performed, plus the fact that the string and arithmetic operations required to convert the identifier string "vn" to the integer array index n cause an extra overhead for the access and update operations in the array implementation. But in the later examples, increasing numbers of updates to the store will cause the functional store to grow in "size" (i.e., in the number of identifier comparisons required to find a match), while the time required for an access to the array store, once the index value is calculated, remains constant.

The example programs were also timed using a list of (string, integer) pairs as the global store variable. In this case, the global variable implementation results in slower, not faster, execution. There appear to be two reasons for this:

1. Since the example programs tested were compiled through the PSI/DAOS system using a "standard" interpretation, the compiled programs are actually SML functions, and their execution time will be affected by the underlying SML system used. In our case, the run-time system on which the SML compiler is based has been designed to make the manipulation of function closures (such as our Identifier → Number store) much more efficient than in more traditional compilers, so changing from a functional implementation to the use of a list data structure will not by itself give the speedup in execution time that we might expect.

2. In the functional implementation, the store is represented by what amounts to a nested if-then-else construction, and each update operation can be viewed as adding a new level to the nesting. An access operation is then a sequence of
comparisons, until we reach the topmost level at which the identifier in question is defined (its most recent update). The list implementation is very similar in structure; each update adds a new (identifier, number) pair to the front of the list, and an access is implemented by a recursive function which searches sequentially from the front of the list. This similarity in structure would lead us to expect similar execution times for the two implementations, and it may be that the recursive nature of the list access operation is the key difference that causes consistently slower performance for the list implementation.

Even though we have tested only a few small examples, it can be seen that the detection of a single-threaded domain and its transformation into a global data structure can, in some cases, have a significant effect on execution efficiency. But the degree of improvement (indeed, whether there is an improvement) is dependent on the type of global data structure used, the pattern of usage of the globalized domain by the program being optimized, and also on properties of the compiler generation system being used.
CHAPTER 5. FUTURE WORK

In Chapters 2 and 3 we have given a general definition of the single-threading property, based on the notion of preserving a consistency property of an expression during a reduction sequence. We have developed two sets of sufficient criteria for the detection of single-threading, one for a variant of the TML combinator language and one for generalized combinator languages. In each case, the criteria can be checked in linear time, as opposed to the exponential time requirements of dataflow analysis-based methods.

The criteria for generalized combinator languages require only that the language be (polymorphically) typed and that we use eager evaluation on the domain being considered for single-threading. Other domains may be evaluated lazily, and evaluation order (e.g., left to right, parallel) is unimportant.

Some issues in the theoretical work which merit further study are:

- In the development of the single-threading criteria for TML we assumed eager evaluation on all domains. The criteria and their correctness proofs should be reexamined to see whether we can relax this restriction to allow partially lazy evaluation, as in the generalized case.

- The addition of product types to the generalized notation should be studied. It is unclear at this point whether the problems that were encountered in the TML
work (finding suitable s-t conditions for the tuple combinator) would carry over into the generalized situation and, if so, how they should be handled.

• The differences between the two sets of criteria, as discussed in the final section of Chapter 3, should be examined more closely. In general, it would be desirable to have a generalized criteria which could be specialized in some systematic way to obtain a more specific criteria that is tailored to a given combinator language such as TML, and to know that no power was lost by this specialization.

• Finally, the notion of the "power" of a single-threading criteria (which we used casually just now) is not well-defined. Since we can only give sufficient, but not necessary, criteria for single-threading, it would be useful to be able to characterize what kinds of single-threaded expressions a given criteria fails to detect. For example, it appears that the specialized criteria given in Chapter 2 will be more powerful than the generalized criteria of Chapter 3 in detecting single-threaded TML expressions, but it would be desirable to verify this more formally. Also, we have suggested that these static, syntactic criteria should be useful in practice because they can be checked much more efficiently than the traditional dataflow analysis methods. But have we also lost a significant amount of detection power in obtaining a faster analysis?

There is also much room for further work in making practical use of our single-threading criteria. The current test implementation in the PSI/DAOS system should be made more complete and user-friendly, and more extensive examples should be tested to help in determining how useful the ideas will actually be in practice.

Another direction of research is the application of static single-threading detec-
tion methods to the problem of optimizing individual functional programs written in a combinator notation. This application introduces some additional complications, such as the presence of multiple arguments of the same type but with different names, the ordering of the arguments in a function definition and the extraction of multiple global variables from a single program. We believe that the ideas presented here can be extended successfully, but there is much work that could be done in this direction.
BIBLIOGRAPHY


