Flow-Directed Lightweight Closure Conversion

JEFFREY MARK SISKIND
NEC Research Institute, Inc.

This paper presents a lightweight closure-conversion method that is driven by the results of whole-program interprocedural flow, reachability, points-to, and escape analyses. The method has been implemented and evaluated as part of a complete SCHEME compiler. When compared with a baseline closure-conversion method that does no optimization, as well as conventional closure-conversion methods that only do optimizations that do not rely on interprocedural analysis, this method significantly increases the degree of closure, variable-slot, parent-slot, closure-pointer–slot, variable, parent-parameter, parent-passing, closure-pointer, variable-spilling, and parent-spilling elimination. It also significantly increases the degree of parent-slot, closure-pointer–slot, and parent-parameter compression and reduces the number of indirections per variable reference. Finally, code produced by this compiler runs significantly faster than code produced by other state-of-the-art SCHEME compilers.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*procedures, functions, and subroutines*; D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*

General Terms: Experimentation, Languages, Performance

Additional Key Words and Phrases: Closure conversion, compiler construction, global optimization/flow analysis, higher-order functional languages

1. INTRODUCTION

This paper describes the lightweight closure-conversion process used by STALIN. STALIN is an aggressively optimizing whole-program compiler for SCHEME. Experiments described in section 4 illustrate that code produced by STALIN is regularly several times faster, and often an order of magnitude or two faster, than code produced by other state-of-the-art SCHEME compilers such as SCHEME->C, GAMBIT-C, BIGLOO, and CHEZ. As also illustrated in section 4, much of this speedup depends on whole-program analysis and lightweight closure conversion. While the techniques described in this paper are formulated in terms of SCHEME, and have been implemented and evaluated for SCHEME, they should be generally

This research was supported, in part, by a Samuel and Miriam Wein Academic Lectureship and by the Natural Sciences and Engineering Research Council of Canada. Part of this research was performed while the author was at the University of Toronto Department of Computer Science, the Technion Department of Electrical Engineering, and the University of Vermont Department of Electrical Engineering and Computer Science. I wish to thank Ken Anderson, Robert Cartwright, Henry Cejtin, Matthias Felleisen, Cormac Flanagan, Suresh Jagannathan, Jacob Katzenelson, Shriram Krishnamurthi, David McAllester, Richard O'Keefe, Stephen Weeks, and Andrew Wright for many useful discussions pertaining to the material presented in this paper.

applicable to any lexically-scoped higher-order language, such as COMMON LISP, ML, HASKELL, SMALLTALK, DYLAN, etc.

STALIN is a batch-mode compiler. Unlike traditional interactive SCHEME implementations, STALIN does not provide a read-eval-print loop or the ability to eval or load new code into a running program. STALIN compiles the SCHEME source program into a single executable, indirectly via C. The behaviour of this executable is equivalent to loading the source program into a virgin interactive SCHEME implementation and terminating its execution. Any computation results from evaluating top-level expressions. Thus STALIN is intended more for application delivery and production research runs than program development.

STALIN performs numerous whole-program analyses and optimizations. First, it does polyvariant interprocedural flow and reachability analysis. The results of this analysis support interprocedural points-to, escape, and lifetime analyses, lightweight closure and CPS conversion, automatic in-lining, region-based storage management, unboxing, and program-specific and program-point-specific low-level representation selection and code generation with on-the-fly coercion between different representations. This paper describes only the interprocedural points-to and escape analyses, automatic in-lining, and lightweight closure conversion. Companion papers [Siskind 2000a; 2000b; 2000c; 2000d] describe the remaining analyses and optimizations.

The remainder of this paper is organized as follows. Section 2 gives an overview of the lightweight closure-conversion process used by STALIN by way of comparison to alternate implementations that do little or no optimization. Section 3 presents the lightweight closure-conversion process in detail. Section 4 presents experimental results. Section 5 concludes with a discussion of related work.

2. OVERVIEW

To understand the optimizations that STALIN performs during closure conversion, it is useful to compare the lightweight closure-conversion process performed by STALIN with both a baseline implementation that does no optimization as well as a conventional implementation[1] that does limited optimization. For simplicity, the baseline, conventional, and lightweight implementations are presented here using a linked closure representation.[2]

The baseline implementation has the following characteristics:

(1) Each procedure has a *closure*.

(2) The closure for each procedure contains a *variable slot* for each variable bound by that procedure.

(3) The closure for each procedure contains a *parent slot*, a pointer to the closure for the immediate lexically surrounding procedure. The parent slot of the closure for the top-level procedure contains a null pointer.

[1]Throughout this paper, I use the term *conventional* implementation to refer to optimizations that can be performed without interprocedural analysis, regardless of whether any actual implementation performs any or all of these optimizations.

[2]The lightweight closure-conversion process used by STALIN is not particular to a linked closure representation. It can be applied to display and flat closure representations as well.

(4) A procedure object contains a *closure-pointer slot*, a pointer to the closure for the immediate lexically surrounding procedure. The closure-pointer slot for the top-level procedure contains a null pointer.

(5) A procedure object contains a *code-pointer slot*, a pointer to the code object for that procedure.

(6) The code object for a procedure has a *variable parameter* for each variable bound by that procedure.

(7) The code object for a procedure has a *parent parameter*, a pointer to the closure for the immediate lexically surrounding procedure. The parent parameter for the top-level procedure will be bound to the null pointer.

(8) Procedure calls indirect to the code object pointed to by the target procedure object.

(9) *Variable passing*: A procedure call passes each argument to its corresponding variable parameter in the code object.

(10) *Parent passing*: A procedure call passes the closure pointer of the target procedure object to the parent parameter of the code object.

(11) Each code object contains a *closure pointer*, a local variable that holds a pointer to its closure. Each code object begins with a preamble that allocates a closure and stores a pointer to that closure in the closure pointer.

(12) *Variable spilling*: The preamble spills each variable parameter into the corresponding variable slot of the closure.

(13) *Parent spilling*: The preamble spills the parent parameter into the parent slot of the closure.

(14) Variables are referenced[3] indirectly via the closure pointer.

The baseline, conventional, and lightweight implementations differ in the following ways:

2.1 Direct procedure calls

Baseline and conventional implementations generate code like the following for a lambda expression `(lambda (x1) ...)`:

```
t1.tag = ONE_ARGUMENT_PROCEDURE;
t1.closure = closure;
t1.code = &p47;

OBJECT p47(CLOSURE parent, OBJECT x1)
{ CLOSURE closure;
  ⋮
  /* slots: CLOSURE parent; OBJECT x1; */
  closure = allocate_closure47();
  closure.parent = parent;
  closure.x1 = x1;
  ...}
```

[3]Throughout this paper, I use the term *reference* to mean access or assignment.

and code like the following for a procedure call (`(`e_1 e_2`)`):

```
t3 = e1;
t4 = e2;
if (t3.tag!=ONE_ARGUMENT_PROCEDURE) error();
t5 = *(t3.code)(t3.closure, t4);
```

Here, a procedure object p has three slots: its type tag p`.tag`, its closure-pointer slot p`.closure`, and its code-pointer slot p`.code`.[4] The type tag of a procedure object encodes its arity and a call to a procedure indirects through its code-pointer slot. In contrast, STALIN always generates direct procedure calls. This is possible because STALIN does whole-program interprocedural flow analysis and can determine all potential targets of each call site. STALIN uses the type tag of a procedure object to encode the identity of the procedure, rather than its arity, and dispatches on the type tag to a collection of direct procedure calls, one for each potential target. Thus, for example, STALIN generates code like the following[5] for a lambda expression `(lambda (x1) ...)`:

```
t1.tag = PROCEDURE47;
t1.closure = closure;

OBJECT p47(CLOSURE parent, OBJECT x1)
{ CLOSURE closure;
  ⋮
  /* slots: CLOSURE parent; OBJECT x1; */
  closure = allocate_closure47();
  closure.parent = parent;
  closure.x1 = x1;
  ...}
```

and code like the following for a procedure call (`(`e_1 e_2`)`, when e_1 can take on several potential objects, including the procedures 47 and 63:

[4]Throughout this paper, I am deliberately vague as to whether objects are boxed or unboxed and uniformly use $o.s$ to denote a reference to slot s of object o rather than distinguishing between $o.s$ and o`->`s. Furthermore, when objects are boxed, I am also deliberately vague as to whether slots, like the type tag, are represented in the object or its handle, and whether they are somehow compressed or encoded, say, in unused addresses or address bits in the handle. See Siskind [2000c] for a discussion of such representation issues.

[5]The actual C code generated by STALIN is fully typed. Each procedure has a corresponding closure of a distinct C type and parent slots, closure-pointer slots, parent parameters, and closure pointers are all typed appropriately to the type of closure that they contain. C variables and slots that can hold objects of multiple types, such as `t3` in this example, are represented as C unions. The code generated by STALIN contains the appropriate casts at each reference. For simplicity, throughout this paper, all such type declarations and casts are eliminated.

```
t3 = e1;
t4 = e2;
switch (t3.tag)
{ case PROCEDURE47: t5 = p47(t3.closure, t4);
  case PROCEDURE63: t5 = p63(t3.closure, t4);
  ⋮
  default: error();}
```

The default branch of the dispatch is eliminated when flow analysis determines that the target can only be a procedure of the correct arity. Furthermore, the dispatch itself is eliminated when flow analysis determines that there is only one potential target.[6] This yields code like the following:

```
t3 = e1;
t4 = e2;
t5 = p47(t3.closure, t4);
```

Using direct procedure calls, and encoding the identity of the target procedure in the type tag, allows Stalin to eliminate the code-pointer slot of procedure objects.

2.2 Parent-slot, closure-pointer–slot, parent-parameter, parent-passing, and parent-spilling elimination

The baseline implementation has a parent slot, a closure-pointer slot, a parent parameter, parent passing, and parent spilling for every procedure. Conventional implementations eliminate the parent slot, closure-pointer slot, parent parameter, parent passing, and parent spilling for the top-level procedure because the pointers are always null. Conventional implementations also eliminate the parent slot of a closure if that slot is never accessed. Stalin carries this optimization further, eliminating the closure-pointer–slot, parent-parameter, parent-passing, and parent-spilling for a procedure when the parent parameter is never accessed. Situations where the parent parameter is never accessed occur even more frequently in Stalin than in conventional implementations because parent parameters are used primarily to access variable slots of closures of lexically surrounding procedures and Stalin performs more aggressive variable and variable-slot elimination, as described below. Furthermore, Stalin is able to perform more aggressive parent-slot elimination than conventional implementations for the same reason. Experiments reported in section 4 show that, in practice, almost all parent slots, closure-pointer slots, parent parameters, parent passing, and parent spilling are eliminated. Closure-pointer–slot, parent-parameter, parent-passing, and parent-spilling elimination creates a complication. Since some procedures have closure-pointer slots, parent parameters, and parent passing, while others do not, the code generated for a procedure call will vary depending on the potential targets. This is possible both because flow analysis determines the potential targets and because the use of dispatch to direct procedure calls allows a target-specific call sequence to be generated when there is more than one target. For example, Stalin generates code like the following for a lambda expression `(lambda (x1) ...)` that has a parent parameter:

[6] When flow analysis determines that the target cannot be a procedure of the correct arity, the dispatch is eliminated and a direct call to the error handler is generated.

```
t1.tag = PROCEDURE47;
t1.closure = closure;[1]

OBJECT p47(CLOSURE parent,[2] OBJECT x1)
{ CLOSURE closure;
  ⋮
  /* slots: CLOSURE parent;[3] OBJECT x1; */
  closure = allocate_closure47();
  closure.parent = parent;[4]
  closure.x1 = x1;
  ...}
```

code like the following for a lambda expression `(lambda (x2) ...)` that does not have a parent parameter:

```
t2.tag = PROCEDURE63;

OBJECT p63(OBJECT x2)
{ CLOSURE closure;
  ⋮
  /* slots:  OBJECT x2; */
  closure = allocate_closure63();
  closure.x2 = x2;
  ...}
```

and code like the following for a procedure call $(e_1\ e_2)$, where the potential targets include procedure 47, which has a parent parameter, and procedure 63, which does not:

```
t3 = e1;
t4 = e2;
switch (t3.tag)
{ case PROCEDURE47: t5 = p47(t3.closure,[5] t4);
  case PROCEDURE63: t5 = p63(t4);
  ⋮
  default: error();}
```

Note that the parent slot (box 3 above), closure-pointer slot (box 1 above), parent parameter (box 2 above), and parent spilling (box 4 above) that are present in procedure 47 are absent from procedure 63. Also note that the calling sequence for procedure 47 contains parent passing (box 5 above) which is absent from the calling sequence for procedure 63. Such per-target differentiation is possible because multiple-target calls generate dispatches to direct procedure calls.

2.3 Eliminating indirection through the closure pointer

The baseline implementation always references variables indirectly through the closure pointer. Conventional implementations eliminate such indirection in two situ-

ations. First, because `closure.parent` must alias `parent`, a free variable reference can proceed via the parent parameter instead of via the closure pointer. The baseline implementation would generate the following code:

$$\texttt{closure.}\underbrace{\texttt{parent.}\ldots\texttt{.parent}}_{n}\texttt{.}x$$

for a free reference to a variable x that is bound n levels up while a conventional implementation would generate the following code:

$$\underbrace{\texttt{parent.}\ldots\texttt{.parent}}_{n}\texttt{.}x$$

instead. Second, if a variable is never assigned, a bound access can proceed via its variable parameter rather than via the closure pointer. The baseline implementation would generate `closure.`x to access the local variable x while a conventional implementation would simply generate x. STALIN applies this optimization in a wider set of circumstances because it uses a much more complex notion of what constitutes a free vs. bound reference, as described below.

2.4 Eliminating fictitious variables

The baseline implementation has a slot, a parameter, passing, and spilling for each variable. The slot, parameter, passing, and spilling for a variable can be eliminated when a variable can hold only a single concrete object. Such a variable is called *fictitious*. STALIN approximates the fictitious property, with the results of flow analysis, as any variable that can hold only a single abstract object where that abstract object contains a single concrete object. The abstract interpretation used by STALIN during flow analysis treats `()`, `#t`, `#f`, the end-of-file object, and procedures created by a given lambda expression as distinct abstract objects. All of these except abstract procedures always contain a single concrete object. Abstract procedures contain single concrete procedures when the procedure has no closure-pointer slot, due to closure-pointer–slot elimination, as described below. Furthermore, the concrete aggregate objects in abstract aggregate objects such as pairs, strings, vectors, symbols, continuations, and procedures are indistinguishable when their identity is not important and the components of those aggregate objects are fictitious or unaccessed. Such a collection of indistinguishable concrete objects can be treated as a single concrete object. Finally, continuations can be indistinguishable in certain circumstances that require a must-alias property to hold. The method used by STALIN to approximate this must-alias property is the crux of the lightweight closure-conversion process and is described in detail in section 3.12. Experiments reported in section 4 show that closure-pointer–slot elimination, in practice, allows most procedures to be fictitious and the variables holding such procedures to be eliminated. Not only does the notion of fictitious variables affect code generation, by eliminating the slots, parameters, passing, and spilling for such variables, as well as references to such variables, it also impacts the lightweight closure-conversion process itself. Free references to fictitious variables are ignored, allowing a greater degree of parent-slot, closure-pointer–slot, parent-parameter, parent-passing, and parent-spilling elimination.

2.5 Eliminating variables that aren't accessed

The baseline implementation has a slot, a parameter, passing, and spilling for each variable. Conventional implementations eliminate the slot, parameter, passing, and spilling for a non-global variable that is never accessed. STALIN carries this optimization further. First, STALIN eliminates all unaccessed variables, not just non-global ones. Second, only reached, non-fictitious accesses, as determined by flow and reachability analysis, count as accesses. An access might be fictitious, even if the variable itself is not fictitious, if the access is in the consequent or alternate of a conditional. For example, an access to `x` in e_1 in

```
(if (eq? x 'a) e1 e2)
```

would be fictitious even if `x` itself were not. Third, as for the case of eliminating fictitious variables, this optimization not only affects code generation, by eliminating the slots, parameters, passing, and spilling for such variables, as well as assignments to such variables, it also impacts the lightweight closure-conversion process itself. Free assignments to unaccessed variables are ignored, allowing a greater degree of parent-slot, closure-pointer–slot, parent-parameter, parent-passing, and parent-spilling elimination.

2.6 Variable-slot elimination

The baseline implementation has a slot and spill for each variable. Furthermore, all variable references are to variable slots via the closure pointer. Conventional implementations eliminate the slot and spill for a variable that is never freely referenced, compiling references to such a variable to the parameter instead of the slot. STALIN carries this optimization further. First, only reached free references, as determined by flow and reachability analysis, count as free references. Second, only non-fictitious free accesses, as determined by flow analysis, count as free accesses. Third, free assignments to hidden variables don't count as free assignments. Hiding will be described below. Fourth, slots are eliminated for globalized variables. Globalization will be described below. Fifth, slots and spills are eliminated for hidden variables. Sixth, the slot and spill for a variable can be eliminated, and references to such a variable can be compiled to the parameter instead of the slot, if two conditions are met. First, all references to the variable must be from within the C function that binds that variable. With the current STALIN code generator, this happens when the variable reference is in-lined in the same procedure where the variable is bound. Second, at every reference to the variable, the slot that would be accessed by the baseline implementation must alias the corresponding parameter in the most recent active invocation of the procedure that binds that variable. The method used by STALIN to approximate this must-alias property is the crux of the lightweight closure-conversion process and is described in detail in section 3.12.

2.7 Closure elimination, closure-pointer elimination, and parent-slot compression

The baseline implementation has a closure and closure pointer for every procedure. And the parent slot for a procedure always points to the closure for the immediate lexically surrounding procedure (or is null if there is none). Conventional implementations eliminate a closure, and the corresponding closure pointer, when all of its variable slots are eliminated. Conventional implementations also perform *parent-*

slot compression: whenever the parent slot of one closure would point to another closure but is only used to access the parent slot of the later closure, the parent slot of the former closure can point to the closure that the parent slot of the later closure points to. Parent-slot compression is necessary when a closure is eliminated, since all parent slots of other closures that would point to an eliminated closure must instead point to the closure that the parent slot of the eliminated closure would have pointed to or be eliminated if the parent slot of the eliminated closure was eliminated. Parent-slot compression reduces indirection in variable reference. Stalin performs more aggressive closure elimination, closure-pointer elimination, and parent-slot compression than conventional implementations because these optimizations are driven by variable and variable-slot elimination and Stalin performs more aggressive variable and variable-slot elimination. Experiments reported in section 4 show that, in practice, almost all closures and closure pointers are eliminated. When they are not, the number of indirections needed to reference free variables is almost always reduced to very low levels.

2.8 Closure-pointer–slot and parent-parameter compression

In the baseline implementation, the closure-pointer slot and parent parameter for a procedure always point to the closure for the immediate lexically surrounding procedure (or are null if there is none). Conventional implementations perform *closure-pointer–slot and parent-parameter compression*: whenever the parent parameter would point to a closure but is only used to access the parent slot of that closure, the closure-point-slot and parameter parameter can point to the closure that the parent slot of the original closure points to. Closure-pointer–slot and parent-parameter compression reduce indirection in variable reference. Stalin performs more aggressive closure-pointer–slot and parent-parameter compression than conventional implementations because these optimizations are driven by variable and variable-slot elimination and Stalin performs more aggressive variable and variable-slot elimination. Experiments reported in section 4 show that, in practice, the number of indirections needed to reference free variables is almost always reduced to very low levels.

2.9 Globalization

The baseline implementation generates slots in closures for all variables. Conventional implementations treat variables that are bound at the top level specially, compiling them as global variables rather than as slots of a closure. Stalin generalizes this optimization. A variable can be *globalized* when it can have at most one live instance. Stalin approximates this one-live-instance property. A variable has at most one live instance if the procedure that binds that variable is not called more than once. And a procedure p is not called more than once if all procedures that can call p, either directly or indirectly, have a single reached call site. A variable also has at most one live instance if the procedure that binds that variable does not call itself directly or indirectly through a non-tail call and all accesses to that variable must alias the most recently created instance. The method used by Stalin to approximate this must-alias property is the crux of the lightweight closure-conversion process and is described in detail in section 3.12. Globalization impacts code generation. Globalized variables need a form of spilling but do not need parameters

or passing. Furthermore, globalized variables are referenced without indirection. Globalization also impacts the lightweight closure-conversion process itself. Since globalized variables do not require variable slots and are referenced without access to parent parameters or parent slots, globalization allows more aggressive parent-slot, closure-pointer-slot, and parent parameter compression and closure, parent-slot, closure-pointer–slot, parent-parameter, parent-passing, closure-pointer, and parent-spilling elimination.

2.10 Hiding

The situation often arises where a variable slot always points to the closure containing that slot. Such a situation arises in the following example:

```
(define (f x)
 (define (g y) ... x ...)
 (define (h z) ... x ...)
 ...)
```

assuming that there are no assignments to the variables `g` and `h` (other than those implicit in the definitions). Here, `f` requires a closure because `x` is freely referenced. And `g` and `h` both take the closure for `f` as their parent parameter to allow them to reference `x`. Thus the procedure objects for `g` and `h` will have the closure for `f` as their closure-pointer slot. Furthermore, the variables slots for `g` and `h` in the closure for `f` will always contain the procedure objects for `g` and `h` respectively. Since STALIN eliminates code-pointer slots of procedure objects, and since the type tags for the slots `g` and `h` can be eliminated because they contain known procedures, the only information in the variable slots for `g` and `h` is their closure-pointer slot. But these will always point to the closure that contains those slots. So such variable slots, as well as the corresponding parameters, passing, and spilling, are redundant and can be eliminated as *hidden*. A conventional implementation would compile a bound access to a variable x as x and a free access to a variable x that is bound n levels up as:

$$\underbrace{\texttt{parent.}\ldots.\texttt{parent}}_{n}.x$$

When x is hidden, STALIN compiles a bound access to x as an access to the closure pointer `closure` and a free access to x that is bound n levels up as:

$$\underbrace{\texttt{parent.}\ldots.\texttt{parent}}_{n}$$

STALIN generalizes the hiding optimization even further. A variable is hidden not only when it must point to the closure that contains its slot, but also when it must point to a specific closure for a procedure that lexically surrounds the procedure whose closure contains that slot. In this situation, if x is a variable that must point to a closure m levels up, STALIN compiles a bound access to x as:

$$\underbrace{\texttt{parent.}\ldots.\texttt{parent}}_{m}$$

and a free access to x that is bound n levels up as:

$$\underbrace{\texttt{parent.....parent}}_{m+n}$$

Sound application of this optimization requires that the variable slot to be hidden must point to the correct ancestor closure. The method used by STALIN to approximate this must-alias property is the crux of the lightweight closure-conversion process and is described in detail in section 3.12. Hiding impacts code generation. Hidden variables do not need slots, parameters, passing, or spilling. Hiding also impacts the lightweight closure-conversion process itself. Since hidden variables do not require variable slots, are never assigned, and are accessed by accessing a wider-scope closure than the closure of the procedure that binds that variable, hiding allows more aggressive parent-slot, closure-pointer–slot, and parent-parameter compression and closure, parent-slot, closure-pointer–slot, parent-parameter, parent-passing, closure-pointer, and parent-spilling elimination.

3. LIGHTWEIGHT CLOSURE CONVERSION

Closure conversion is often formulated as a source-to-source transformation, often as a series of small transformations. In contrast, the approach to lightweight closure conversion taken in this paper is to view closure conversion as an analysis phase that annotates the source program with certain properties and relations followed by a code-generation phase that uses those annotations to control aspects of the code being generated. In this case, the code generator itself is rather straightforward. The annotations control the presence or absence of fragments of the procedure call and entry sequences: variable parameters and slots, parent parameters and slots, closures, closure pointers, and closure-pointer slots, variable and parent spilling, and variable and parent passing. The annotations also control the code generated for variable references. In contrast, the analysis phase is rather complex. This section presents that analysis phase.

Because the analysis phase is complex, I will start with an overview. The input to the analysis phase is the abstract-syntax tree of the program. The output consists of the following annotations:

LOCAL(x)	x will be allocated as a local variable.
GLOBAL(x)	x will be allocated as a global variable.
HIDDEN(x)	x will be allocated as a hidden closure slot.
SLOTTED(x)	x will be allocated as a closure slot.
HASCLOSURE(p)	p will have a closure and closure pointer.
HASPARENTSLOT(p)	p will have a parent slot.
PARENTSLOT(p)	The parent slot for p will point to the closure for PARENTSLOT(p).
HASPARENTPARAMETER(p)	p will have a parent parameter and closure-pointer slot.
PARENTPARAMETER(p)	The parent parameter and closure-pointer slot for p will point to the closure for PARENTPARAMETER(p).

The first four annotations are properties that apply to variables. The remaining five

annotations apply to abstract procedures. At most one of the first four properties will be true of any given variable. That property indicates how that variable is represented. If none of these four properties are true of some variable, then that variable is eliminated and not explicitly represented at run time. This can happen either because the variable is not accessed or because the variable holds a compile-time determinable object. As for the five abstract-procedure annotations, three are properties and two are functions. The properties HasClosure(p), HasParentSlot(p), and HasParentParameter(p) indicate whether closure, parent-slot, or closure-pointer–slot and parent-parameter elimination apply to a given abstract procedure p. ParentSlot(p) and ParentParameter(p) are functional annotations that indicate parent-slot and closure-pointer-slot or parent-parameter compression. If a given abstract procedure p has a parent slot, then ParentSlot(p) indicates which abstract procedure creates the closure that the parent slot of p points to. Likewise, if a given abstract procedure p has a closure-pointer–slot or parent parameter, then ParentParameter(p) indicates which abstract procedure creates the closure that the closure-pointer–slot or parent parameter of p points to.

Before producing the ultimate annotations that drive the closure-conversion aspects of code generation from the undecorated abstract-syntax tree, Stalin produces a number of intermediate annotations. First, Stalin performs flow analysis (described in section 3.2). This determines the potential values of expressions, variables, and other locations. As part of flow analysis, Stalin performs reachability analysis (described in section 3.3). This determines which expressions are reached and which variables are accessed and assigned. Among other things, this allows Stalin to ignore unreached accesses when deciding whether a variable is accessed, to ignore unreached references when deciding whether a variable is freely referenced, and to eliminate unaccessed variables. Next, Stalin determines which components of aggregate objects are accessed and assigned (described in section 3.4). This allows Stalin to eliminate unaccessed components of aggregate objects, and in turn, to eliminate an aggregate object itself when all of its components are eliminated and the identity of the object is unimportant. Next, Stalin computes the call graph (described in section 3.5). Because all call sites are higher-order in Scheme, this requires the output of flow analysis to do effectively. Next, Stalin uses the call graph to determine which abstract procedures are not called more than once (described in section 3.6). This information supports globalization of variables, since variables can be globalized when they have only one live instance and this is trivially true when the abstract procedure that binds a variable is not called more than once. Next, Stalin computes the free variables for each abstract procedure (described in section 3.7). This computation uses the results of flow and reachability analysis to ignore unreached references. The free-variable computation is used, in part, to support variable-slot elimination. Next, Stalin does points-to analysis, driven by the output of flow analysis, to determine which objects can point to other objects (described in section 3.8). Points-to analysis supports escape analysis (described in section 3.9) which determines which objects can be accessed after their creating procedure returns. Escape analysis is used to support must-alias analysis which in turn supports variable-slot elimination, globalization, hiding, and determining when continuations are fictitious. Next, Stalin determines which abstract procedures to in-line (described in section 3.10). In-lining decisions are driven by

the results of flow analysis and are used to support variable-slot elimination. Finally, STALIN determines which procedures are reentrant (described in section 3.11). This is important since variables that are bound by reentrant procedures cannot be globalized because they can have more that one live instance.

STALIN makes a single pass through the above analyses. Since each analysis depends on prior analyses, the order in which the analyses can be performed is fairly tightly constrained. But there are no circularities in these analyses. Thus a single pass suffices.[7]

The above analyses are precursors to the analyses that are directly associated with lightweight closure conversion. In fact, the above analyses support other analyses and optimizations that STALIN performs in addition to lightweight closure conversion. After performing the above analyses, STALIN performs the analyses to directly support lightweight closure conversion. First, STALIN performs must-alias analysis (described in section 3.12). This is used to justify variable-slot elimination, globalization, and hiding and to determine when continuations are fictitious. Next, STALIN determines which abstract locations are fictitious, i.e. always contain the same compile-time determinable concrete object (described in section 3.13). Next, STALIN determines which variable references are trivial and can be ignored (described in section 3.14). Next, STALIN determines which variable slots can be eliminated (described in section 3.15). Next, STALIN determines which variables can be globalized (described in section 3.16). Next, STALIN determines the ancestor relation (described in section 3.17). Next, STALIN determines which variables can be hidden (described in section 3.18). Finally, STALIN completes the lightweight closure-conversion process by determining the representation for each variable and the static backchain for each abstract procedure (described in section 3.19). Unlike the analyses in sections 3.2 through 3.11, the analyses in sections 3.12 through 3.19 are circular, requiring the computation of a least fixpoint.

Because the analyses described in this section are fairly complex and numerous, I have adopted several conventions in terminology and notation. I often define properties or relations that apply to expressions e or expression invocations $\hat{e}$. In all such cases, the property or relation is intended to apply to e or $\hat{e}$ itself, not any subexpressions of e or $\hat{e}$ and not any expressions in procedures that might be called by e or $\hat{e}$.

I often define a base relation along with its transitive and reflexive-transitive variants. When doing so, the symbol for the transitive variant will contain a '$+$' symbol, while the symbol for the reflexive-transitive variant will contain a '$*$' symbol. Often, the $+$ or $*$ symbol will be overlayed on the symbol for the base relation, as in $\oplus$ or $\circledast$ respectively. This is to distinguish between an inherently transitive or reflexive-transitive relation, and $\circ^+$ or $\circ^*$, which I use to denote the transitive or reflexive-transitive closures of the $\circ$ relation respectively. The reason for such distinction is that $\circ^+$ or $\circ^*$ might be less precise than $\oplus$ or $\circledast$ respectively.

I often give English names to relations. When doing so, the English name for

[7] Actually, in one little way, flow analysis depends on the fictitious property which is computed after flow analysis. `(eq?` e_1 e_2`)` can be determined not to yield `#f` when $\beta(e_1)$ and $\beta(e_2)$ each contain the same single fictitious abstract object. STALIN eliminates the circularity in this case by using a weaker noncircular approximation to the fictitious property during flow analysis.

the base relation will contain the term 'direct(ly)' while the English name for the transitive variant will contain the term 'proper(ly).' The English name for the reflexive-transitive variant will not contain any special term.

I often overload the notation for properties or relations to refer to both *concrete* properties or relations, i.e. ones that apply to concrete entities, and *abstract* properties or relations, i.e. ones that apply to abstract entities. (As will be described later, abstract entities are taken to be sets of concrete entitites.) Associated with each such pair of properties or relations is an *abstraction lemma* which states that the abstract property or relation holds between some abstract entities if the concrete property or relation holds between some concrete entities in those abstract entities.[8] Note that the converse might not be true. For example, suppose that k_1 and k_2 are concrete entities in the abstract entity σ_1 and that k_3 and k_4 are concrete entities in the abstract entity σ_2. And suppose that the concrete relations $k_1 \circ k_3$ and $k_2 \circ k_4$ hold. Then, by the abstraction lemma, the abstract relation $\sigma_1 \circ \sigma_2$ holds even though the concrete relations $k_1 \circ k_4$ and $k_2 \circ k_3$ might not hold. Thus abstraction can introduce a degree of approximation.

I often define a pair of properties or relations: the first being an underlying property or relation with the second being an approximation to the first. The need for approximations follows from the fact that the underlying properties and relations are often undecidable. Underlying properties and relations, as well as syntactic properties and relations that do not need approximations, are written without an overbar. In contrast, a property or relation written with an overbar, as in $\overline{\circ}$, denotes an approximation to the corresponding underlying property or relation $\circ$. The soundness of the analysis, and thus the soundness of the lightweight closure-conversion process in general, follows from the fact that the approximations are *conservative*. Associated with each approximation is a *conservative approximation lemma* which states that the approximation holds between some abstract entities if the underlying property or relation holds between those abstract entities.[9] Note that it is trivially possible to derive sound approximations by taking $\overline{\circ}$ to be always true. This corresponds to disabling all optimization. On the other hand, it is not possible to derive a tight approximation because the underlying relations are undecidable. Thus it is always possible to derive tighter approximations. The approximations presented in this paper are a good compromise in that they empirically yield good results, yet can be computed efficiently.

Most of the analyses described in this paper are presented in three stages: concrete properties or relations, followed by abstract properties or relations, followed, in turn, by conservative approximations to the abstract properties or relations. Ac-

[8] In some cases, particularly for the fictitious property described in section 3.13, the polarity is reversed. In such cases, the abstraction lemma states that the abstract property or relation does not hold between some abstract entitites if the concrete property or relation does not hold between some concrete entitites in those abstract entitites.

[9] In some cases, particularly for the must-alias, fictitious, localizable, globalizable, and hideable properties described in sections 3.12, 3.13, 3.15, 3.16, and 3.18 respectively, the polarity is reversed. In such cases, the conservative approximation lemma states that approximation does not hold between some abstract entities if the underlying property or relation does not hold between those abstract entities. In such cases, it is trivially possible to derive sound approximations by taking $\overline{\circ}$ to be always false.

e	$\Rightarrow$	`(quote` k`)`	constant
	|	x	access
	|	`(set!` x e`)`	assignment
	|	`(`e_0 $e_1 \ldots e_n$`)`	call
	|	`(`q $e_1 \ldots e_n$`)`	primcall
	|	`(lambda (`$x_1 \ldots x_n$`)` e`)`	lambda
	|	`(if` e_0 e_1 e_2`)`	conditional

Fig. 1. The STALIN abstract syntax.

cordingly, most of the analyses have two sources of imprecision: the abstraction lemma mapping the concrete properties or relations to abstract properties or relations and the conservative approximation lemma mapping abstract properties or relations to approximations of those properties or relations.

Because the analysis leading to lightweight closure conversion is complex and involves numerous intermediate program annotations, the terminology and notation used in the remainder of this section may appear to be quite cumbersome. To help guide the reader, I have summarized the terminology and notation in a glossary in appendix A.

3.1 Preprocessing

STALIN prepends a standard prelude to every source program that defines the builtin procedures from R4RS [Clinger and Rees 1991]. STALIN then macro expands this extended source program and converts it into an abstract-syntax tree that is essentially of the form specified in figure 1. This macro expansion is done using essentially the same rewrite rules as given in the appendix of R4RS. The result of this macro expansion is a single closed lambda expression that contains the entire source program, including the standard prelude. Running the program corresponds to evaluating this lambda expression to yield a procedure and calling this procedure. All subsequent analyses are performed on the abstract-syntax tree and take the form of computing properties of, and relations between, nodes in this tree and abstract objects and locations.

In the abstract syntax, I refer to the subexpression of an assignment as its *source*, the first subexpression of a call or primcall as its *callee*, any remaining subexpressions of a call or primcall as its *arguments*, any variables of a lambda expression as its *parameters*, the number of arguments of a call or primcall or the number of parameters of a lambda expression as its *arity*, the subexpression of a lambda expression as its *body*, and the subexpressions of a conditional as its *antecedent*, *consequent*, and *alternate* respectively. Note that the macro-expansion process produces abstract-syntax trees where the body of each lambda expression consists of a single expression and the alternate of a conditional is always present.

Throughout this paper, I use the symbol x to denote variables, the symbol q to denote primitives, the symbol e to denote expressions, the symbol p to denote lambda expressions, and the symbol p_0 to denote the top-level lambda expression that contains the entire source program including the standard prelude. Furthermore, I use the symbols X, E, A, S, C, C_i, R, R_i, and P to denote the set of all variables, expressions, accesses, assignments, calls, calls of arity i, primcalls, primcalls of arity i, and lambda expressions in the source program respectively.

Table I. The STALIN primitives.

`pair?`	`cons`	`car`	`cdr`
`set-car!`	`set-cdr!`	`not`	`boolean?`
`eq?`	`null?`	`symbol?`	`symbol->string`
`string->symbol`	`number?`	`real?`	`integer?`
`exact?`	`inexact?`	`=`	`<`
`>`	`<=`	`>=`	`zero?`
`positive?`	`negative?`	`max`	`min`
`+`	`*`	`-`	`/`
`quotient`	`remainder`	`floor`	`ceiling`
`truncate`	`round`	`exp`	`log`
`sin`	`cos`	`tan`	`asin`
`acos`	`atan`	`sqrt`	`expt`
`exact->inexact`	`inexact->exact`	`char?`	`char->integer`
`integer->char`	`string?`	`make-string`	`string`
`string-length`	`string-ref`	`string-set!`	`vector?`
`make-vector`	`vector`	`vector-length`	`vector-ref`
`vector-set!`	`procedure?`	`apply`	`call/cc`
`input-port?`	`output-port?`	`open-input-file`	`open-output-file`
`close-input-port`	`close-output-port`	`read-char`	`peak-char`
`eof-object?`	`char-ready?`	`write-char`	

Throughout this paper, I use equality between expressions or between variables to denote intensional rather than extensional equality, i.e. `eq?` rather than `equal?`. This also affects notions derived from equality such as set membership. Thus I use $e_1 = e_2$ to mean that e_1 and e_2 denote the same expression in the source program, not that they have the same form. Similarly, I use $x_1 = x_2$ to mean that x_1 and x_2 denote the same variable in the source program, not that they have the same name. This obviates the need for alpha renaming and the complexity of expression indices as used by Steckler and Wand [1997].

Table I lists essentially the primitives available in STALIN.[10] Note that primitives are not first-class objects. A primitive can only appear as the first subexpression of a primcall. Primitives are taken to be disjoint from variables. This allows distinguishing primcalls from calls by whether or not the first subexpression is a primitive. The names of primitives are intentionally similar to the names of some builtin R4RS procedures because they implement the same functionality. User programs, however, never contain primcalls. Instead, they call procedures defined in the standard prelude which contain the corresponding primcalls. This is essentially η expansion. As an optimization, STALIN replaces certain calls in the user program that are known to be to certain procedures that are defined in the standard prelude with the corresponding primcalls, thus performing η reduction. Since the analyses described in this paper are formulated independently of this optimization, it will be safely ignored.

For the sake of expository simplicity, the abstract syntax presented in figure 1, as well as all of the analyses presented in this paper, make three crucial simplifications. First, the primitive `apply` is not supported. Second, variable-arity procedures (i.e. 'rest' arguments) are not supported. Third, the primitive `call/cc` cannot be passed a continuation as the first argument. The actual implementation does not

[10] Note that STALIN does not currently implement complex and rational numbers.

impose these restrictions.

Throughout this paper, I use the following functions applied to nodes in the abstract-syntax tree:

SOURCE(e)	Denotes the source subexpression of an assignment e.
$x(e)$	Denotes the variable in an access or assignment e.
CALLEE(e)	Denotes the callee subexpression of a call or primcall e.
ARGUMENTS(e)	Denotes the set of argument subexpressions of a call or primcall e.
ARGUMENT$_i$(e)	Denotes the i^{th} argument subexpression of a call or primcall e.
PARAMETER$_i$(e)	Denotes the i^{th} parameter of a lambda expression e.
ARITY(e)	Denotes the arity of a call, primcall, or lambda expression e.
BODY(e)	Denotes the body subexpression of a lambda expression e.
$p(x)$	Denotes the lambda expression in which x is bound.
$p(e)$	Denotes the narrowest lambda expression that properly contains e.

Furthermore, if $p_2 = p(p_1)$, I say that p_1 is *directly nested in* p_2, denoted $p_1 \prec p_2$. Let $\prec^+$ and $\prec^*$ be the transitive and reflexive-transitive closures of $\prec$ respectively. If $p_1 \prec^+ p_2$ or $p_1 \prec^* p_2$, I say that p_1 is *properly nested in* or *nested in* p_2 respectively. Finally, I say that an expression e is *in tail position*, denoted INTAILPOSITION(e), if and only if it is

—the body of a lambda expression or

—the consequent or alternate of a conditional that is in tail position.

Note that this definition of tail position agrees with the definition given in R5RS [Kelsey et al. 1998]. The above annotations all denote simple syntactic properties of the source program and do not involve any approximation.

3.2 Flow analysis

After macro expansion, STALIN performs whole-program interprocedural flow analysis. Flow analysis determines the potential values of expressions. The lightweight closure-conversion process uses the results of flow analysis in numerous ways. Knowing the potential targets of call sites allows building a call graph which in turn allows determining which procedures are reentrant or are called more than once. This supports globalization of variables. Flow analysis also supports escape analysis which forms the basis of a crucial must-alias criterion for variable-slot elimination, globalization, hiding, and determining when continuations are fictitious. Flow analysis is actually a precursor, directly or indirectly, to almost all of the subsequent analyses that support lightweight closure conversion.

Flow analysis is an abstract interpretation that partitions the set of all concrete objects into a set of abstract objects that is finite for a given program and the set of all concrete locations into a set of abstract locations that is also finite for a given program. Thus each abstract object is a set of concrete objects and each abstract location is a set of concrete locations. Concrete interpretation, i.e. evaluation, associates concrete locations with variable instances, expression invocations,

the slots of concrete pairs, and the elements of concrete strings and vectors. Abstract interpretation, i.e. flow analysis, associates abstract locations with variables, expressions, the slots of abstract pairs, and the elements of abstract strings and vectors. Throughout this paper, I use the symbol k to denote concrete objects, the symbol l to denote concrete locations, the symbol σ to denote abstract objects, and the symbol β to denote abstract locations.

A concrete location can be viewed as the set of all concrete objects that that concrete location can have as its value. Similarly, an abstract location can be viewed as the set of all abstract objects that that abstract location can have as its value. The principal relation produced by flow analysis is $\sigma \mathrel{\overline{\in}} \beta$.

Flow analysis obeys the following abstraction lemma:

LEMMA 1. *For all concrete objects k, concrete locations l, abstract objects σ, and abstract locations β, if, in some state in some execution of the program, $k \in \sigma$, $l \in \beta$, and $k \in l$, then $\sigma \in \beta$.*

Flow analysis also obeys the following conservative approximation lemma:

LEMMA 2. *For all abstract objects σ and abstract locations β, if $\sigma \in \beta$, then $\sigma \mathrel{\overline{\in}} \beta$.*

Different flow analyses differ in the way that they group concrete objects and locations into abstract objects and locations as well as how they approximate $\in$ as $\mathrel{\overline{\in}}$. The lightweight closure-conversion process described in this paper works for any flow analysis that meets the following criteria:

(1) All concrete procedures created by a given lambda expression are in the same abstract procedure.

(2) All concrete locations associated with different instances of a given variable are in the same abstract location.

(3) All concrete locations associated with different invocations of a given expression are in the same abstract location.

(4) If an abstract object contains a concrete procedure then it does not contain any concrete non-procedures.

(5) If an abstract object contains a concrete pair then it does not contain any concrete non-pairs.

(6) If an abstract object contains a concrete string then it does not contain any concrete non-strings.

(7) If an abstract object contains a concrete vector then it does not contain any concrete non-vectors.

(8) If an abstract object contains a concrete symbol then it does not contain any concrete non-symbols.

(9) If an abstract object contains a concrete continuation then it does not contain any concrete non-continuations.

(10) All concrete locations associated with the `car` slots of different concrete pairs in the same abstract pair are in the same abstract location.

(11) All concrete locations associated with the `cdr` slots of different concrete pairs in the same abstract pair are in the same abstract location.

(12) All concrete locations associated with all of the elements of different concrete strings in the same abstract string are in the same abstract location.

(13) All concrete locations associated with all of the elements of different concrete vectors in the same abstract vector are in the same abstract location.

(14) All concrete locations associated with the print-name–string slot of different concrete symbols in the same abstract symbol are in the same abstract location.

(15) All concrete continuations created from different invocations of a given expression are in the same abstract continuation.

Criteria (4) through (9) imply that it is meaningful to talk about abstract procedures, pairs, strings, vectors, symbols, and continuations respectively. Criteria (10) through (14) imply that it is meaningful to talk about the slots and elements of abstract pairs, strings, vectors, and symbols respectively. Criteria (1) through (3) imply that the flow analysis is monovariant (i.e. computed with 0-CFA [Shivers 1988; 1990; 1991a; 1991b; Heintze 1993; 1994]). STALIN actually performs a novel polyvariant analysis [Siskind 2000b] which requires enhancements to the lightweight closure-conversion analyses presented here. For expository simplicity, these enhancements are not presented. They complicate the presentation of the analyses but do not add any conceptual novelty.

As a result of criterion (1) above, there is a one-to-one correspondence between lambda expressions and abstract procedures. Thus I often use the symbol p to denote either a lambda expression or an abstract procedure. Furthermore, the following properties and functions are meaningful because of the above constraints on flow analysis:

$\beta(x)$	Denotes the abstract location associated with x.
$\beta(e)$	Denotes the abstract location associated with the result of evaluating e.
PAIR?(σ)	True if σ is an abstract pair.
CAR(σ)	Denotes the abstract location containing the `car` slots of the concrete pairs in the abstract pair σ.
CDR(σ)	Denotes the abstract location containing the `cdr` slots of the concrete pairs in the abstract pair σ.
STRING?(σ)	True if σ is an abstract string.
STRING-REF(σ)	Denotes the abstract location containing the elements of the concrete strings in the abstract string σ.
VECTOR?(σ)	True if σ is an abstract vector.
VECTOR-REF(σ)	Denotes the abstract location containing the elements of the concrete vectors in the abstract vector σ.
SYMBOL?(σ)	True if σ is an abstract symbol.
SYMBOL->STRING(σ)	Denotes the abstract location containing the print-name strings of the concrete symbols in the abstract symbol σ.
CONTINUATION?(σ)	True if σ is an abstract continuation.
$e(\sigma)$	Denotes the call to `call/cc` where the concrete continuations in the abstract continuation σ were created.

3.3 Reachability analysis

As part of performing flow analysis, STALIN also performs reachability analysis. Reachability analysis determines which expressions are reached and, in turn, which variables are accessed or assigned. Reachability analysis supports lightweight closure conversion in numerous ways. Unaccessed variables can be eliminated. Unreached free references can be ignored when determining whether variables slots can be eliminated. Like flow analysis, reachability analysis is actually a precursor, directly or indirectly, to almost all of the subsequent analyses that support lightweight closure conversion.

Before defining the concrete reachability properties, the following auxiliary definitions are needed:

Definition 1. Calling a procedure created by evaluating a lambda expression p creates an *expression invocation* $\hat{e}$ for each expression e where $p(e) = p$ and creates a *variable instance* $\hat{x}$ for each variable x where $p(x) = p$.

Each expression invocation has program points just before and just after that expression invocation. Assignment, call, and primcall invocations also have intermediate program points. Assignment invocations have an intermediate program point just after the source subexpression invocation but before the actual assignment takes place. Call invocations have an intermediate program point just after the callee and argument subexpression invocations but before the actual procedure call takes place. Primcall invocations have an intermediate program point just after the argument subexpression invocations but before the actions associated with the primitive take place.

Given the above, the concrete reachability properties are defined as follows:

Definition 2. An expression invocation $\hat{e}$ is *reached*, denoted $\text{REACHED}(\hat{e})$, when control flows to the program point just before $\hat{e}$. An expression invocation $\hat{e}$ *returns*, denoted $\text{RETURNS}(\hat{e})$, when control flows to the program point just after $\hat{e}$. An assignment, call, or primcall invocation $\hat{e}$ is *executed*, denoted $\text{EXECUTED}(\hat{e})$, when control flows to the intermediate program point in $\hat{e}$. A variable instance $\hat{x}$ is *accessed*, denoted $\text{ACCESSED}(\hat{x})$, when some access invocation $\hat{e}$ to x is reached. A variable instance $\hat{x}$ is *assigned*, denoted $\text{ASSIGNED}(\hat{x})$, when some assignment invocation $\hat{e}$ to x is executed. A call invocation $\hat{e}$ is *successful* if it is executed and the arity of the call site equals the arity of the target procedure or continuation. A primcall invocation $\hat{e}$ is *successful* if it is executed and the arity of the call site is allowed by the primitive.

Given the above, the abstract reachability properties are defined as follows:

Definition 3. An expression e is *reached*, *returns*, or is *executed*, denoted $\text{REACHED}(e)$, $\text{RETURNS}(e)$, or $\text{EXECUTED}(e)$, if some invocation $\hat{e}$ of e is reached, returns, or is exectuted in some execution of the program respectively. A variable x is *accessed* or *assigned*, denoted $\text{ACCESSED}(x)$ or $\text{ASSIGNED}(x)$, if some instance $\hat{x}$ of x is accessed or assigned in some execution of the program respectively.

Flow analysis computes $\overline{\text{REACHED}}$, $\overline{\text{RETURNS}}$, $\overline{\text{EXECUTED}}$, $\overline{\text{ACCESSED}}$, and $\overline{\text{ASSIGNED}}$ as approximations to REACHED, RETURNS, EXECUTED, ACCESSED, and

ASSIGNED respectively. In particular, flow analysis directly produces the approximations $\overline{\text{REACHED}}$ and $\overline{\text{RETURNS}}$. $\overline{\text{EXECUTED}}$, $\overline{\text{ACCESSED}}$, and $\overline{\text{ASSIGNED}}$ are derived from $\overline{\text{REACHED}}$ and $\overline{\text{RETURNS}}$ by the following:[11]

$$\overline{\text{EXECUTED}}(e) \triangleq \left\{ \begin{array}{l} e \in S \wedge \overline{\text{RETURNS}}(\text{SOURCE}(e)) \vee \\ \left[\begin{array}{l} e \in C \wedge \overline{\text{RETURNS}}(\text{CALLEE}(e)) \wedge \\ (\forall e' \in \text{ARGUMENTS}(e)) \overline{\text{RETURNS}}(e') \end{array} \right] \vee \\ e \in R \wedge (\forall e' \in \text{ARGUMENTS}(e)) \overline{\text{RETURNS}}(e') \end{array} \right\}$$

$$\overline{\text{ACCESSED}}(x) \triangleq (\exists e \in A) x(e) = x \wedge \overline{\text{REACHED}}(e)$$

$$\overline{\text{ASSIGNED}}(x) \triangleq (\exists e \in S) x(e) = x \wedge \overline{\text{EXECUTED}}(e)$$

Reachability analysis obeys the following abstraction lemma:

LEMMA 3. *For all expressions e and invocations $\hat{e}$ of e, if* REACHED$(\hat{e})$*, then* REACHED(e)*, if* RETURNS$(\hat{e})$*, then* RETURNS(e)*, and if* EXECUTED$(\hat{e})$*, then* EXECUTED(e)*. For all variables x and instances $\hat{x}$ of x, if* ACCESSED$(\hat{x})$*, then* ACCESSED(x) *and if* ASSIGNED$(\hat{x})$*, then* ASSIGNED(x)*.*

Reachability analysis also obeys the following conservative approximation lemma:

LEMMA 4. *For all expressions e, if* REACHED(e)*, then* $\overline{\text{REACHED}}(e)$*, if* RETURNS$(e)$*, then* $\overline{\text{RETURNS}}(e)$*, and if* EXECUTED$(e)$*, then* $\overline{\text{EXECUTED}}(e)$*. For all variables x, if* ACCESSED(x)*, then* $\overline{\text{ACCESSED}}(x)$ *and if* ASSIGNED(x)*, then* $\overline{\text{ASSIGNED}}(x)$*.*

3.4 Approximating the abstract aggregate access and assignment properties and relations

After flow and reachability analysis, STALIN approximates the abstract aggregate accesses and assigns relations and the abstract aggregate accessed and assigned properties. Just as a variable or variable instance can be accessed or assigned, components of concrete and abstract aggregate objects can be accessed or assigned. And just as STALIN can eliminate unaccessed variables and perform certain other optimizations on unassigned variables, STALIN can eliminate unaccessed components of abstract aggregate objects. And just as STALIN can eliminate a closure when all of its variable slots are eliminated, STALIN can eliminate an aggregate object when

[11] STALIN actually uses a more precise notion of $\overline{\text{ACCESSED}}(x)$. A variable is accessed if some access to that variable is accessed. And a reached expression is accessed in the following cases:

—An access is accessed.

—The source of an assignment is accessed if its destination is accessed.

—The callee of a call is accessed.

—An argument of a call to a procedure is accessed if the corresponding parameter is accessed.

—An argument of a primcall is accessed.

—The body of a lambda expression is accessed if some call to that lambda expression is accessed.

—The antecedant of a conditional is accessed.

—The consequent and alternate of an accessed conditional are accessed.

—The body of the top-level lambda expression is accessed.

The objective of this more complex analysis is to treat x_1 as unaccessed when x_2 is unaccessed in code like `(let ((`x_2 x_1`)) ...)`.

all of its components are eliminated. To do this, STALIN needs a precise notion of what are all of the different kinds of aggregate objects and their components and under what circumstances these components are accessed and assigned.

In addition to procedures, whose closures are handled by other mechanisms, SCHEME has four kinds of aggregate objects: pairs, strings, vectors, and symbols. Pairs have two components: their `car` and `cdr` slots. Vectors and strings have two kinds of components: their length slot and their elements. Symbols have one component: their print-name–string slot. Additionally, all objects have a type-tag slot. Each of these components is accessed or assigned by a disjoint set of primitives. The `car` and `cdr` primitives access the `car` and `cdr` slots respectively. The `set-car!` and `set-cdr!` primitives assign the `car` and `cdr` slots respectively. The `string-length` and `vector-length` primitives access the string and vector length slots respectively. The `string-ref` and `vector-ref` primitives access string and vector elements respectively. The `string-set!` and `vector-set!` primitives assign string and vector elements respectively. The `symbol->string` primitive accesses the print-name–string slot. And the `not`, `boolean?`, `pair?`, `null?`, `symbol?`, `number?`, `real?`, `integer?`, `char?`, `string?`, `vector?`, `procedure?`, `input-port?`, `output-port?`, and `eof-object?` primitives access the type-tag slot. Note that it is not possible to assign length, print-name–string, or type-tag slots.

STALIN is able to eliminate abstract locations that contain a single compile-time–determinable concrete object. Such locations are called *fictitious*. This optimization will be discussed in detail in section 3.19. What is relevant here is that if none of the components of an abstract aggregate object are ever accessed then all of the concrete objects in that abstract object are indistinguishable. And a location that contains only indistinguishable objects can be treated as fictitious and eliminated. But in order for this to be sound, one other condition must be met. In SCHEME, aggregate objects have identity in addition to components. Even if the components of an object are never accessed, it is still possible to check the identity of an object with the `eq?` primitive. In order to treat the concrete aggregate objects in an abstract aggregate object as indistinguishable, two conditions must hold: all of its components must be unaccessed and its identity must never be checked with the `eq?` primitive.

SCHEME nominally has a fifth kind of aggregate object: continuations. But continuations don't have components that can be accessed or assigned, except for a type-tag slot. Nonetheless, it is convenient to treat continuations as accessed when they are called. This allows the concrete continuations in an abstract continuation to be treated as indistinguishable when their type-tag slot is never accessed and they are never called. Note that it is not necessary to determine that the identity of an abstract continuation is never checked in order to treat its concrete continuations as indistinguishable since R4RS does not define `eq?`-ness for continuations.

Similarly, it is convenient to treat procedures as accessed when they are called. This allows the concrete procedures in an abstract procedure to be treated as indistinguishable when their type-tag slot is never accessed and they are never called. Again, note that it is not necessary to determine that the identity of an abstract procedure is never checked in order to treat its concrete procedures as indistinguishable since R4RS does not define `eq?`-ness for procedures.

STALIN is able to do one further optimization. It can eliminate the code for prim-

itive predicates when flow analysis determines that a primcall to such a predicate will always return `#t` or always return `#f`.

The concrete aggregate accesses[12] and assigns relations are defined as follows:

Definition 4. A one-argument primcall invocation $\hat{e}$ to `not`, `boolean?`, `pair?`, `null?`, `symbol?`, `number?`, `real?`, `integer?`, `char?`, `string?`, `vector?`, `procedure?`, `input-port?`, `output-port?`, or `eof-object?` *type-tag accesses* a concrete object k passed as the argument, denoted TypeTagAccesses$(\hat{e}, k)$, when $\hat{e}$ is executed, unless flow analysis has determined that the invocation must return `#t` or must return `#f`. A two-argument primcall invocation $\hat{e}$ to `eq?` *eq? accesses* a concrete object k passed as one of the arguments, denoted Eq?Accesses$(\hat{e}, k)$, when $\hat{e}$ is executed, unless flow analysis has determined that the invocation must return `#t` or must return `#f`. A one-argument primcall invocation $\hat{e}$ to `car` *car accesses* a concrete pair k passed as the argument, denoted CarAccesses$(\hat{e}, k)$, when $\hat{e}$ is executed. A one-argument primcall invocation $\hat{e}$ to `cdr` *cdr accesses* a concrete pair k passed as the argument, denoted CdrAccesses$(\hat{e}, k)$, when $\hat{e}$ is executed. A one-argument primcall invocation $\hat{e}$ to `string-length` *string-length accesses* a concrete string k passed as the argument, denoted StringLengthAccesses$(\hat{e}, k)$, when $\hat{e}$ is executed. A two-argument primcall invocation $\hat{e}$ to `string-ref` *string-ref accesses* a concrete string k passed as the first argument, denoted StringRefAccesses$(\hat{e}, k)$, when $\hat{e}$ is executed and an exact in-bounds integer is passed as the second argument. A one-argument primcall invocation $\hat{e}$ to `vector-length` *vector-length accesses* a concrete vector k passed as the argument, denoted VectorLengthAccesses$(\hat{e}, k)$, when $\hat{e}$ is executed. A two-argument primcall invocation $\hat{e}$ to `vector-ref` *vector-ref accesses* a concrete vector k passed as the first argument, denoted VectorRefAccesses$(\hat{e}, k)$, when $\hat{e}$ is executed and an exact in-bounds integer is passed as the second argument. A one-argument primcall invocation $\hat{e}$ to `symbol->string` *symbol-to-string accesses* a concrete symbol k passed as the argument, denoted SymbolToStringAccesses$(\hat{e}, k)$, when $\hat{e}$ is executed. A call invocation $\hat{e}$ to a concrete continuation k *continuation accesses* k, denoted ContinuationAccesses$(\hat{e}, k)$, when $\hat{e}$ is successful. A call invocation $\hat{e}$ to a concrete procedure k *procedure accesses* k, denoted ProcedureAccesses$(\hat{e}, k)$, when $\hat{e}$ is successful. A two-argument primcall invocation $\hat{e}$ to `set-car!` *car assigns* a concrete pair k passed as the first argument, denoted CarAssigns$(\hat{e}, k)$, when $\hat{e}$ is executed. A two-argument primcall invocation $\hat{e}$ to `set-cdr!` *cdr assigns* a concrete pair k passed as the first argument, denoted CdrAssigns$(\hat{e}, k)$, when $\hat{e}$ is executed. A three-argument primcall invocation $\hat{e}$ to `string-set!` *string-ref assigns* a concrete string k passed as the first argument, denoted StringRefAssigns$(\hat{e}, k)$, when $\hat{e}$ is executed, an exact in-bounds integer is passed as the second argument, and a character is passed as the third argument. A three-argument primcall invocation $\hat{e}$ to `vector-set!` *vector-ref assigns* a concrete vector k passed as the first argument, denoted VectorRefAssigns$(\hat{e}, k)$, when $\hat{e}$ is executed and an exact in-bounds integer is passed as the second argument.

Given the above, the concrete aggregate accessed and assigned properties are

[12]Note that since Stalin does not currently implement rational and complex numbers, the primitives `rational?` and `complex?` are not included in the notion of type-tag access.

defined as follows:

Definition 5. A concrete object k is *type-tag*, *eq?*, *car*, *cdr*, *string-length*, *string-ref*, *vector-length*, *vector-ref*, *symbol-to-string*, *continuation*, or *procedure accessed*, or *car*, *cdr*, *string-ref*, or *vector-ref assigned*, denoted TypeTagAccessed(k), Eq?Accessed(k), CarAccessed(k), CdrAccessed(k), StringLengthAccessed(k), StringRefAccessed(k), VectorLengthAccessed(k), VectorRefAccessed(k), SymbolToStringAccessed(k), ContinuationAccessed(k), ProcedureAccessed(k), CarAssigned(k), CdrAssigned(k), StringRefAssigned(k), or VectorRefAssigned(k), if some expression invocation $\hat{e}$ type-tag, eq?, car, cdr, string-length, string-ref, vector-length, vector-ref, continuation, or procedure accesses or car, cdr, string-ref, or vector-ref assigns k respectively.

Given the above, the abstract aggregate accesses and assigns relations are defined as follows:

Definition 6. An expression e *type-tag*, *eq?*, *car*, *cdr*, *string-length*, *string-ref*, *vector-length*, *vector-ref*, *continuation*, or *procedure accesses*, or *car*, *cdr*, *string-ref*, or *vector-ref assigns* an abstract object σ, denoted TypeTagAccesses(e, σ), Eq?Accesses(e, σ), CarAccesses(e, σ), CdrAccesses(e, σ), StringLengthAccesses(e, σ), StringRefAccesses(e, σ), VectorLengthAccesses(e, σ), VectorRefAccesses(e, σ), SymbolToStringAccesses(e, σ), ContinuationAccesses(e, σ), ProcedureAccesses(e, σ), CarAssigns(e, σ), CdrAssigns(e, σ), StringRefAssigns(e, σ), or VectorRefAssigns(e, k), if some invocation $\hat{e}$ of e type-tag, eq?, car, cdr, string-length, string-ref, vector-length, vector-ref, continuation, or procedure accesses or car, cdr, string-ref, or vector-ref assigns a concrete object $k \in \sigma$ in some execution of the program respectively.

Given the above, the abstract aggregate accessed and assigned properties are defined as follows:

Definition 7. An abstract object σ is *type-tag*, *eq?*, *car*, *cdr*, *string-length*, *string-ref*, *vector-length*, *vector-ref*, *symbol-to-string*, *continuation*, or *procedure accessed*, or *car*, *cdr*, *string-ref*, or *vector-ref assigned*, denoted TypeTagAccessed(σ), Eq?Accessed(σ), CarAccessed(σ), CdrAccessed(σ), StringLengthAccessed(σ), StringRefAccessed(σ), VectorLengthAccessed(σ), VectorRefAccessed(σ), SymbolToStringAccessed(σ), ContinuationAccessed(σ), ProcedureAccesses(σ,), CarAssigned(σ), CdrAssigned(σ), StringRefAssigned(σ), or VectorRefAssigned(σ), if some expression invocation $\hat{e}$ type-tag, eq?, car, cdr, string-length, string-ref, vector-length, vector-ref, continuation, or procedure accesses or car, cdr, string-ref, or vector-ref assigns a concrete object $k \in \sigma$ in some execution of the program respectively.

Stalin approximates the abstract aggregate accesses and assigns relations as

follows:

$$\overline{\textsc{TypeTagAccesses}}(e,\sigma) \triangleq \left(\begin{array}{l} e \in R_1 \wedge \overline{\textsc{Executed}}(e) \wedge \\ \textsc{Callee}(e) \in \left\{\begin{array}{l}\texttt{not},\\ \texttt{boolean?},\\ \texttt{pair?},\\ \texttt{null?},\\ \texttt{number?},\\ \texttt{real?},\\ \texttt{integer?},\\ \texttt{char?},\\ \texttt{string?},\\ \texttt{vector?},\\ \texttt{procedure?},\\ \texttt{input-port?},\\ \texttt{output-port?},\\ \texttt{eof-object?}\end{array}\right\} \wedge \\ \sigma \mathrel{\overline{\in}} \beta(\textsc{Argument}_1(e)) \wedge \\ \texttt{\#t} \mathrel{\overline{\in}} \beta(e) \wedge \texttt{\#f} \mathrel{\overline{\in}} \beta(e) \end{array}\right)$$

$$\overline{\textsc{Eq?Accesses}}(e,\sigma) \triangleq \left[\begin{array}{l} e \in R_2 \wedge \overline{\textsc{Executed}}(e) \wedge \\ \textsc{Callee}(e) = \texttt{eq?} \wedge \\ \left(\begin{array}{l}\sigma \mathrel{\overline{\in}} \beta(\textsc{Argument}_1(e)) \vee \\ \sigma \mathrel{\overline{\in}} \beta(\textsc{Argument}_2(e))\end{array}\right) \wedge \\ \texttt{\#t} \mathrel{\overline{\in}} \beta(e) \wedge \texttt{\#f} \mathrel{\overline{\in}} \beta(e) \end{array}\right]$$

$$\overline{\textsc{CarAccesses}}(e,\sigma) \triangleq \left(\begin{array}{l} e \in R_1 \wedge \overline{\textsc{Executed}}(e) \wedge \\ \textsc{Callee}(e) = \texttt{car} \wedge \\ \sigma \mathrel{\overline{\in}} \beta(\textsc{Argument}_1(e)) \end{array}\right)$$

$$\overline{\textsc{CdrAccesses}}(e,\sigma) \triangleq \left(\begin{array}{l} e \in R_1 \wedge \overline{\textsc{Executed}}(e) \wedge \\ \textsc{Callee}(e) = \texttt{cdr} \wedge \\ \sigma \mathrel{\overline{\in}} \beta(\textsc{Argument}_1(e)) \end{array}\right)$$

$$\overline{\textsc{StringLengthAccesses}}(e,\sigma) \triangleq \left(\begin{array}{l} e \in R_1 \wedge \overline{\textsc{Executed}}(e) \wedge \\ \textsc{Callee}(e) = \texttt{string-length} \wedge \\ \sigma \mathrel{\overline{\in}} \beta(\textsc{Argument}_1(e)) \end{array}\right)$$

$$\overline{\textsc{StringRefAccesses}}(e,\sigma) \triangleq \left(\begin{array}{l} e \in R_2 \wedge \overline{\textsc{Executed}}(e) \wedge \\ \textsc{Callee}(e) = \texttt{string-ref} \wedge \\ \sigma \mathrel{\overline{\in}} \beta(\textsc{Argument}_1(e)) \wedge \\ \textbf{number} \mathrel{\overline{\in}} \beta(\textsc{Argument}_2(e)) \end{array}\right)$$

$$\overline{\textsc{VectorLengthAccesses}}(e,\sigma) \triangleq \left(\begin{array}{l} e \in R_1 \wedge \overline{\textsc{Executed}}(e) \wedge \\ \textsc{Callee}(e) = \texttt{vector-length} \wedge \\ \sigma \mathrel{\overline{\in}} \beta(\textsc{Argument}_1(e)) \end{array}\right)$$

$$\overline{\textsc{VectorRefAccesses}}(e,\sigma) \triangleq \left(\begin{array}{l} e \in R_2 \wedge \overline{\textsc{Executed}}(e) \wedge \\ \textsc{Callee}(e) = \texttt{vector-ref} \wedge \\ \sigma \mathrel{\overline{\in}} \beta(\textsc{Argument}_1(e)) \wedge \\ \textbf{number} \mathrel{\overline{\in}} \beta(\textsc{Argument}_2(e)) \end{array}\right)$$

$$\overline{\textsc{SymbolToStringAccesses}}(e,\sigma) \triangleq \begin{pmatrix} e \in R_1 \wedge \overline{\textsc{Executed}}(e) \wedge \\ \textsc{Callee}(e) = \texttt{symbol->string} \wedge \\ \sigma \mathrel{\overline{\in}} \beta(\textsc{Argument}_1(e)) \end{pmatrix}$$

$$\overline{\textsc{ContinuationAccesses}}(e,\sigma) \triangleq \begin{pmatrix} e \in C_1 \wedge \overline{\textsc{Executed}}(e) \wedge \\ \sigma \mathrel{\overline{\in}} \beta(\textsc{Callee}(e)) \end{pmatrix}$$

$$\overline{\textsc{ProcedureAccesses}}(e,\sigma) \triangleq \begin{pmatrix} e \in C \wedge \overline{\textsc{Executed}}(e) \wedge \\ \sigma \mathrel{\overline{\in}} \beta(\textsc{Callee}(e)) \wedge \\ \textsc{Arity}(e) = \textsc{Arity}(\sigma) \end{pmatrix}$$

$$\overline{\textsc{CarAssigns}}(e,\sigma) \triangleq \begin{pmatrix} e \in R_2 \wedge \overline{\textsc{Executed}}(e) \wedge \\ \textsc{Callee}(e) = \texttt{set-car!} \wedge \\ \sigma \mathrel{\overline{\in}} \beta(\textsc{Argument}_1(e)) \end{pmatrix}$$

$$\overline{\textsc{CdrAssigns}}(e,\sigma) \triangleq \begin{pmatrix} e \in R_2 \wedge \overline{\textsc{Executed}}(e) \wedge \\ \textsc{Callee}(e) = \texttt{set-cdr!} \wedge \\ \sigma \mathrel{\overline{\in}} \beta(\textsc{Argument}_1(e)) \end{pmatrix}$$

$$\overline{\textsc{StringRefAssigns}}(e,\sigma) \triangleq \begin{pmatrix} e \in R_3 \wedge \overline{\textsc{Executed}}(e) \wedge \\ \textsc{Callee}(e) = \texttt{string-set!} \wedge \\ \sigma \mathrel{\overline{\in}} \beta(\textsc{Argument}_1(e)) \wedge \\ \textbf{number} \mathrel{\overline{\in}} \beta(\textsc{Argument}_2(e)) \wedge \\ \textbf{char} \mathrel{\overline{\in}} \beta(\textsc{Argument}_3(e)) \end{pmatrix}$$

$$\overline{\textsc{VectorRefAssigns}}(e,\sigma) \triangleq \begin{pmatrix} e \in R_3 \wedge \overline{\textsc{Executed}}(e) \wedge \\ \textsc{Callee}(e) = \texttt{vector-set!} \wedge \\ \sigma \mathrel{\overline{\in}} \beta(\textsc{Argument}_1(e)) \wedge \\ \textbf{number} \mathrel{\overline{\in}} \beta(\textsc{Argument}_2(e)) \end{pmatrix}$$

Stalin approximates the abstract aggregate accessed and assigned properties as follows:

$$\overline{\textsc{TypeTagAccessed}}(\sigma) \triangleq (\exists e \in R_1)\overline{\textsc{TypeTagAccesses}}(e,\sigma)$$

$$\overline{\textsc{Eq?Accessed}}(\sigma) \triangleq (\exists e \in R_2)\overline{\textsc{Eq?Accesses}}(e,\sigma)$$

$$\overline{\textsc{CarAccessed}}(\sigma) \triangleq (\exists e \in R_1)\overline{\textsc{CarAccesses}}(e,\sigma)$$

$$\overline{\textsc{CdrAccessed}}(\sigma) \triangleq (\exists e \in R_1)\overline{\textsc{CdrAccesses}}(e,\sigma)$$

$$\overline{\textsc{StringLengthAccessed}}(\sigma) \triangleq (\exists e \in R_1)\overline{\textsc{StringLengthAccesses}}(e,\sigma)$$

$$\overline{\textsc{StringRefAccessed}}(\sigma) \triangleq (\exists e \in R_2)\overline{\textsc{StringRefAccesses}}(e,\sigma)$$

$$\overline{\textsc{VectorLengthAccessed}}(\sigma) \triangleq (\exists e \in R_1)\overline{\textsc{VectorLengthAccesses}}(e,\sigma)$$

$$\overline{\textsc{VectorRefAccessed}}(\sigma) \triangleq (\exists e \in R_2)\overline{\textsc{VectorRefAccesses}}(e,\sigma)$$

$$\overline{\textsc{SymbolToStringAccessed}}(\sigma) \triangleq (\exists e \in R_1)\overline{\textsc{SymbolToStringAccesses}}(e,\sigma)$$

$$\overline{\textsc{ContinuationAccessed}}(\sigma) \triangleq (\exists e \in C_1)\overline{\textsc{ContinuationAccesses}}(e,\sigma)$$

$$\overline{\textsc{ProcedureAccessed}}(\sigma) \triangleq (\exists e \in C)\overline{\textsc{ProcedureAccesses}}(e,\sigma)$$

$$\overline{\textsc{CarAssigned}}(\sigma) \triangleq (\exists e \in R_2)\overline{\textsc{CarAssigns}}(e,\sigma)$$

$$\overline{\textsc{CdrAssigned}}(\sigma) \stackrel{\triangle}{=} (\exists e \in R_2)\overline{\textsc{CdrAssigns}}(e,\sigma)$$
$$\overline{\textsc{StringRefAssigned}}(\sigma) \stackrel{\triangle}{=} (\exists e \in R_3)\overline{\textsc{StringRefAssigns}}(e,\sigma)$$
$$\overline{\textsc{VectorRefAssigned}}(\sigma) \stackrel{\triangle}{=} (\exists e \in R_3)\overline{\textsc{VectorRefAssigns}}(e,\sigma)$$

The aggregate access and assignment properties and relations obey the following abstraction lemma:

Lemma 5. *For all expressions e, invocations $\hat{e}$ of e, abstract objects σ, and concrete objects $k \in \sigma$,*
if $\textsc{TypeTagAccesses}(\hat{e},k)$, *then* $\textsc{TypeTagAccesses}(e,\sigma)$,
if $\textsc{Eq?Accesses}(\hat{e},k)$, *then* $\textsc{Eq?Accesses}(e,\sigma)$,
if $\textsc{CarAccesses}(\hat{e},k)$, *then* $\textsc{CarAccesses}(e,\sigma)$,
if $\textsc{CdrAccesses}(\hat{e},k)$, *then* $\textsc{CdrAccesses}(e,\sigma)$,
if $\textsc{StringLengthAccesses}(\hat{e},k)$, *then* $\textsc{StringLengthAccesses}(e,\sigma)$,
if $\textsc{StringRefAccesses}(\hat{e},k)$, *then* $\textsc{StringRefAccesses}(e,\sigma)$,
if $\textsc{VectorLengthAccesses}(\hat{e},k)$, *then* $\textsc{VectorLengthAccesses}(e,\sigma)$,
if $\textsc{VectorRefAccesses}(\hat{e},k)$, *then* $\textsc{VectorRefAccesses}(e,\sigma)$,
if $\textsc{SymbolToStringAccesses}(\hat{e},k)$, *then* $\textsc{SymbolToStringAccesses}(e,\sigma)$,
if $\textsc{ContinuationAccesses}(\hat{e},k)$, *then* $\textsc{ContinuationAccesses}(e,\sigma)$,
if $\textsc{ProcedureAccesses}(\hat{e},k)$, *then* $\textsc{ProcedureAccesses}(e,\sigma)$,
if $\textsc{CarAssigns}(\hat{e},k)$, *then* $\textsc{CarAssigns}(e,\sigma)$,
if $\textsc{CdrAssigns}(\hat{e},k)$, *then* $\textsc{CdrAssigns}(e,\sigma)$,
if $\textsc{StringRefAssigns}(\hat{e},k)$, *then* $\textsc{StringRefAssigns}(e,\sigma)$, *and*
if $\textsc{VectorRefAssigns}(\hat{e},k)$, *then* $\textsc{VectorRefAssigns}(e,\sigma)$.
For all σ and $k \in \sigma$,
if $\textsc{TypeTagAccessed}(k)$, *then* $\textsc{TypeTagAccessed}(\sigma)$,
if $\textsc{Eq?Accessed}(k)$, *then* $\textsc{Eq?Accessed}(\sigma)$,
if $\textsc{CarAccessed}(k)$, *then* $\textsc{CarAccessed}(\sigma)$,
if $\textsc{CdrAccessed}(k)$, *then* $\textsc{CdrAccessed}(\sigma)$,
if $\textsc{StringLengthAccessed}(k)$, *then* $\textsc{StringLengthAccessed}(\sigma)$,
if $\textsc{StringRefAccessed}(k)$, *then* $\textsc{StringRefAccessed}(\sigma)$,
if $\textsc{VectorLengthAccessed}(k)$, *then* $\textsc{VectorLengthAccessed}(\sigma)$,
if $\textsc{VectorRefAccessed}(k)$, *then* $\textsc{VectorRefAccessed}(\sigma)$,
if $\textsc{SymbolToStringAccessed}(k)$, *then* $\textsc{SymbolToStringAccessed}(\sigma)$,
if $\textsc{ContinuationAccessed}(k)$, *then* $\textsc{ContinuationAccessed}(\sigma)$,
if $\textsc{ProcedureAccessed}(k)$, *then* $\textsc{ProcedureAccessed}(\sigma)$,
if $\textsc{CarAssigned}(k)$, *then* $\textsc{CarAssigned}(\sigma)$,
if $\textsc{CdrAssigned}(k)$, *then* $\textsc{CdrAssigned}(\sigma)$,
if $\textsc{StringRefAssigned}(k)$, *then* $\textsc{StringRefAssigned}(\sigma)$, *and*
if $\textsc{VectorRefAssigned}(k)$, *then* $\textsc{VectorRefAssigned}(\sigma)$.

The aggregate access and assignment properties and relations also obey the following conservative approximation lemma:

Lemma 6. *For all expressions e and abstract objects σ,*
if $\textsc{TypeTagAccesses}(e,\sigma)$, *then* $\overline{\textsc{TypeTagAccesses}}(e,\sigma)$,
if $\textsc{Eq?Accesses}(e,\sigma)$, *then* $\overline{\textsc{Eq?Accesses}}(e,\sigma)$,
if $\textsc{CarAccesses}(e,\sigma)$, *then* $\overline{\textsc{CarAccesses}}(e,\sigma)$,

if $\textsc{CdrAccesses}(e,\sigma)$, *then* $\overline{\textsc{CdrAccesses}}(e,\sigma)$,
if $\textsc{StringLengthAccesses}(e,\sigma)$, *then* $\overline{\textsc{StringLengthAccesses}}(e,\sigma)$,
if $\textsc{StringRefAccesses}(e,\sigma)$, *then* $\overline{\textsc{StringRefAccesses}}(e,\sigma)$,
if $\textsc{VectorLengthAccesses}(e,\sigma)$, *then* $\overline{\textsc{VectorLengthAccesses}}(e,\sigma)$,
if $\textsc{VectorRefAccesses}(e,\sigma)$, *then* $\overline{\textsc{VectorRefAccesses}}(e,\sigma)$,
if $\textsc{SymbolToStringAccesses}(e,\sigma)$, *then* $\overline{\textsc{SymbolToStringAccesses}}(e,\sigma)$,
if $\textsc{ContinuationAccesses}(e,\sigma)$, *then* $\overline{\textsc{ContinuationAccesses}}(e,\sigma)$,
if $\textsc{ProcedureAccesses}(e,\sigma)$, *then* $\overline{\textsc{ProcedureAccesses}}(e,\sigma)$,
if $\textsc{CarAssigns}(e,\sigma)$, *then* $\overline{\textsc{CarAssigns}}(e,\sigma)$,
if $\textsc{CdrAssigns}(e,\sigma)$, *then* $\overline{\textsc{CdrAssigns}}(e,\sigma)$,
if $\textsc{StringRefAssigns}(e,\sigma)$, *then* $\overline{\textsc{StringRefAssigns}}(e,\sigma)$, *and*
if $\textsc{VectorRefAssigns}(e,\sigma)$, *then* $\overline{\textsc{VectorRefAssigns}}(e,\sigma)$.
For all σ,
if $\textsc{TypeTagAccessed}(\sigma)$, *then* $\overline{\textsc{TypeTagAccessed}}(\sigma)$,
if $\textsc{Eq?Accessed}(\sigma)$, *then* $\overline{\textsc{Eq?Accessed}}(\sigma)$,
if $\textsc{CarAccessed}(\sigma)$, *then* $\overline{\textsc{CarAccessed}}(\sigma)$,
if $\textsc{CdrAccessed}(\sigma)$, *then* $\overline{\textsc{CdrAccessed}}(\sigma)$,
if $\textsc{StringLengthAccessed}(\sigma)$, *then* $\overline{\textsc{StringLengthAccessed}}(\sigma)$,
if $\textsc{StringRefAccessed}(\sigma)$, *then* $\overline{\textsc{StringRefAccessed}}(\sigma)$,
if $\textsc{VectorLengthAccessed}(\sigma)$, *then* $\overline{\textsc{VectorLengthAccessed}}(\sigma)$,
if $\textsc{VectorRefAccessed}(\sigma)$, *then* $\overline{\textsc{VectorRefAccessed}}(\sigma)$,
if $\textsc{SymbolToStringAccessed}(\sigma)$, *then* $\overline{\textsc{SymbolToStringAccessed}}(\sigma)$,
if $\textsc{ContinuationAccessed}(\sigma)$, *then* $\overline{\textsc{ContinuationAccessed}}(\sigma)$,
if $\textsc{ProcedureAccessed}(\sigma)$, *then* $\overline{\textsc{ProcedureAccessed}}(\sigma)$,
if $\textsc{CarAssigned}(\sigma)$, *then* $\overline{\textsc{CarAssigned}}(\sigma)$,
if $\textsc{CdrAssigned}(\sigma)$, *then* $\overline{\textsc{CdrAssigned}}(\sigma)$,
if $\textsc{StringRefAssigned}(\sigma)$, *then* $\overline{\textsc{StringRefAssigned}}(\sigma)$, *and*
if $\textsc{VectorRefAssigned}(\sigma)$, *then* $\overline{\textsc{VectorRefAssigned}}(\sigma)$.

3.5 Approximating the abstract call-graph properties and relations

After approximating the abstract aggregate access and assignment properties and relations, Stalin approximates the abstract call-graph properties and relations.

The concrete direct call-graph properties and relations are defined as follows:

Definition 8. If k_1 and k_2 are concrete procedures that were created by evaluating p_1 and p_2 respectively, the call invocation $\hat{e}$ was created by calling k_1, the callee of $\hat{e}$ is k_2, and $\hat{e}$ is successful, then k_2 is *called* and $\hat{e}$ and k_1 *directly call* k_2, denoted $\textsc{Called}(k_2)$, $\hat{e} \rhd k_2$, and $k_1 \rhd k_2$ respectively. Furthermore, if e is in tail position, then $\hat{e}$ and k_1 *directly tail call* k_2, denoted $\hat{e} \rhd_t k_2$ and $k_1 \rhd_t k_2$ respectively. And if e is not in tail position, then $\hat{e}$ and k_1 *directly non-tail call* k_2, denoted $\hat{e} \rhd_{\overline{t}} k_2$ and $k_1 \rhd_{\overline{t}} k_2$ respectively. And if e is in tail position and p_1 is in-lined in p_2, denoted $p_1 \hookrightarrow^* p_2$, then $\hat{e}$ and k_1 *directly self-tail call* k_2, denoted $\hat{e} \rhd_{st} k_2$ and $k_1 \rhd_{st} k_2$ respectively. And if either e is not in tail position or p_1 is not in-lined in p_2, then $\hat{e}$ and k_1 *directly non-self-tail call* k_2, denoted $\hat{e} \rhd_{\overline{st}} k_2$ and $k_1 \rhd_{\overline{st}} k_2$ respectively.

Given the above, the concrete transitive call-graph relations are defined as follows:

Definition 9. If $n \geq 2$ and $k_1, \ldots, k_n$ are concrete procedures that were created

by evaluating $p_1, \ldots, p_n$ respectively, the call invocations $\widehat{e_1}, \ldots, \widehat{e_{n-1}}$ were created by calling $k_1, \ldots, k_{n-1}$ respectively, the callee of each $\widehat{e_1}, \ldots, \widehat{e_{n-1}}$ is $k_2, \ldots, k_n$ respectively, and $\widehat{e_1}, \ldots, \widehat{e_{n-1}}$ are successful, then $\widehat{e_1}$ and k_1 *properly call* k_n, denoted $\widehat{e_1} \Rrightarrow k_n$ and $k_1 \Rrightarrow k_n$ respectively. Furthermore, if $e_1, \ldots, e_{n-1}$ are all in tail position, then $\widehat{e_1}$ and k_1 *properly tail call* k_n, denoted $\widehat{e_1} \Rrightarrow_t k_n$ and $k_1 \Rrightarrow_t k_n$ respectively. And if one or more of $e_1, \ldots, e_{n-1}$ are not in tail position, then $\widehat{e_1}$ and k_1 *properly non-tail call* k_n, denoted $\widehat{e_1} \Rrightarrow_{\overline{t}} k_n$ and $k_1 \Rrightarrow_{\overline{t}} k_n$ respectively. And if $e_1, \ldots, e_{n-1}$ are all in tail position, and each $p_1, \ldots, p_{n-1}$ is in-lined in $p_n, p_1, \ldots, p_{n-2}$ respectively, then $\widehat{e_1}$ and k_1 *properly self-tail call* k_n, denoted $\widehat{e_1} \Rrightarrow_{st} k_n$ and $k_1 \Rrightarrow_{st} k_n$ respectively. And if either one or more of $e_1, \ldots, e_{n-1}$ are not in tail position or one or more of $p_1, \ldots, p_{n-1}$ are not in-lined in $p_n, p_1, \ldots, p_{n-2}$ respectively, then $\widehat{e_1}$ and k_1 *properly non-self-tail call* k_n, denoted $\widehat{e_1} \Rrightarrow_{\overline{st}} k_n$ and $k_1 \Rrightarrow_{\overline{st}} k_n$ respectively.

Given the above, the concrete reflexive-transitive call-graph relations are defined as follows:

Definition 10. If $k_1 = k_2$ or $k_1 \Rrightarrow k_2$ then k_1 *calls* k_2, denoted $k_1 \Rrightarrow k_2$. If $k_1 = k_2$ or $k_1 \Rrightarrow_t k_2$ then k_1 *tail calls* k_2, denoted $k_1 \Rrightarrow_t k_2$. If $k_1 = k_2$ or $k_1 \Rrightarrow_{\overline{t}} k_2$ then k_1 *non-tail calls* k_2, denoted $k_1 \Rrightarrow_{\overline{t}} k_2$. If $k_1 = k_2$ or $k_1 \Rrightarrow_{st} k_2$ then k_1 *self-tail calls* k_2, denoted $k_1 \Rrightarrow_{st} k_2$. If $k_1 = k_2$ or $k_1 \Rrightarrow_{\overline{st}} k_2$ then k_1 *non-self-tail calls* k_2, denoted $k_1 \Rrightarrow_{\overline{st}} k_2$.

Given the above, the abstract call-graph properties and relations are defined as follows:

Definition 11. An abstract procedure p is *called*, denoted $\textsc{Called}(p)$, if some concrete procedure $k \in p$ is called in some execution of the program. A call e *directly calls* an abstract procedure p, denoted $e \rhd p$, if some call invocation $\hat{e}$ of e directly calls some concrete procedure $k \in p$ in some execution of the program. An abstract procedure p_1 *directly calls* an abstract procedure p_2, denoted $p_1 \rhd p_2$, if some concrete procedure $k_1 \in p_1$ directly calls some concrete procedure $k_2 \in p_2$ in some execution of the program. Likewise for the relations $\rhd_t$, $\rhd_{\overline{t}}$, $\rhd_{st}$, $\rhd_{\overline{st}}$, $\Rrightarrow$, $\Rrightarrow_t$, $\Rrightarrow_{\overline{t}}$, $\Rrightarrow_{st}$, $\Rrightarrow_{\overline{st}}$, $\Rrightarrow$, $\Rrightarrow_t$, $\Rrightarrow_{\overline{t}}$, $\Rrightarrow_{st}$, and $\Rrightarrow_{\overline{st}}$.

Note that the direct and proper tail, non-tail, self-tail, and non-self-tail call relations are all independent because it is possible for an abstract procedure to both directly tail and non-tail call another abstract procedure by different paths.

Stalin approximates the abstract call-graph properties and relations as follows:

$$
\begin{aligned}
\overline{\textsc{Called}}(p) &\triangleq (\exists e \in C) e \mathrel{\overline{\rhd}} p \\
e \mathrel{\overline{\rhd}} p &\triangleq \overline{\textsc{Executed}}(e) \wedge p \mathrel{\overline{\in}} \beta(\textsc{Callee}(e)) \wedge \textsc{Arity}(e) = \textsc{Arity}(p) \\
p_1 \mathrel{\overline{\rhd}} p_2 &\triangleq (\exists e \in C) p(e) = p_1 \wedge e \mathrel{\overline{\rhd}} p_2 \\
e \mathrel{\overline{\rhd_t}} p &\triangleq e \mathrel{\overline{\rhd}} p \wedge \textsc{InTailPosition}(e) \\
p_1 \mathrel{\overline{\rhd_t}} p_2 &\triangleq (\exists e \in C) p(e) = p_1 \wedge e \mathrel{\overline{\rhd_t}} p_2 \\
e \mathrel{\overline{\rhd_{\overline{t}}}} p &\triangleq e \mathrel{\overline{\rhd}} p \wedge \neg\textsc{InTailPosition}(e) \\
p_1 \mathrel{\overline{\rhd_{\overline{t}}}} p_2 &\triangleq (\exists e \in C) p(e) = p_1 \wedge e \mathrel{\overline{\rhd_{\overline{t}}}} p_2
\end{aligned}
$$

$$
\begin{aligned}
e\ \overline{\rhd_{st}}\ p &\triangleq e\ \overline{\rhd}\ p \wedge \textsc{InTailPosition}(e) \wedge p \hookrightarrow^* p(e) \\
p_1\ \overline{\rhd_{st}}\ p_2 &\triangleq (\exists e \in C) p(e) = p_1 \wedge e\ \overline{\rhd_{st}}\ p_2 \\
e\ \overline{\rhd_{\overline{st}}}\ p &\triangleq e\ \overline{\rhd}\ p \wedge (\neg \textsc{InTailPosition}(e) \vee p \not\hookrightarrow^* p(e)) \\
p_1\ \overline{\rhd_{\overline{st}}}\ p_2 &\triangleq (\exists e \in C) p(e) = p_1 \wedge e\ \overline{\rhd_{\overline{st}}}\ p_2 \\
p_1\ \overline{\Rrightarrow}\ p_2 &\triangleq p_1\ \overline{\rhd}^+\ p_2 \\
p_1\ \overline{\Rrightarrow_t}\ p_2 &\triangleq p_1\ \overline{\rhd_t}^+\ p_2 \\
p_1\ \overline{\Rrightarrow_{\overline{t}}}\ p_2 &\triangleq (\exists p, p' \in P) p_1\ \overline{\Rrightarrow}\ p \wedge p\ \overline{\rhd_{\overline{t}}}\ p' \wedge p'\ \overline{\Rrightarrow}\ p_2 \\
p_1\ \overline{\Rrightarrow_{st}}\ p_2 &\triangleq p_1\ \overline{\rhd_{st}}\ p_2 \vee (\exists p \in P) p_1\ \overline{\rhd_t}\ p \wedge p \hookrightarrow p_1 \wedge p\ \overline{\Rrightarrow_{st}}\ p_2 \\
p_1\ \overline{\Rrightarrow_{\overline{st}}}\ p_2 &\triangleq p_1\ \overline{\rhd_{\overline{st}}}\ p_2 \vee (\exists p \in P) \left[\begin{array}{l} p_1\ \overline{\rhd_t}\ p \wedge p \hookrightarrow p_1 \wedge p\ \overline{\Rrightarrow_{\overline{st}}}\ p_2 \vee \\ (p_1\ \overline{\rhd_{\overline{t}}}\ p \vee p_1\ \overline{\rhd_t}\ p \wedge p \not\hookrightarrow^* p_1) \wedge p\ \overline{\Rrightarrow}\ p_2 \end{array} \right] \\
e\ \overline{\Rrightarrow}\ p &\triangleq (\exists p' \in P) e\ \overline{\rhd}\ p' \wedge p'\ \overline{\Rrightarrow}\ p \\
e\ \overline{\Rrightarrow_t}\ p &\triangleq (\exists p' \in P) e\ \overline{\rhd_t}\ p' \wedge p'\ \overline{\Rrightarrow_t}\ p \\
e\ \overline{\Rrightarrow_{\overline{t}}}\ p &\triangleq (\exists p' \in P) e\ \overline{\rhd_{\overline{t}}}\ p' \wedge p'\ \overline{\Rrightarrow}\ p \vee e\ \overline{\rhd_t}\ p' \wedge p'\ \overline{\Rrightarrow_{\overline{t}}}\ p \\
e\ \overline{\Rrightarrow_{st}}\ p &\triangleq e\ \overline{\rhd_{st}}\ p \vee (\exists p' \in P) e\ \overline{\rhd_t}\ p' \wedge p' \hookrightarrow p(e) \wedge p'\ \overline{\Rrightarrow_{st}}\ p_2 \\
e\ \overline{\Rrightarrow_{\overline{st}}}\ p &\triangleq e\ \overline{\rhd_{\overline{st}}}\ p \vee (\exists p' \in P) \left[\begin{array}{l} e\ \overline{\rhd_t}\ p' \wedge p' \hookrightarrow p(e) \wedge p'\ \overline{\Rrightarrow_{\overline{st}}}\ p \vee \\ (e\ \overline{\rhd_{\overline{t}}}\ p' \vee e\ \overline{\rhd_t}\ p' \wedge p' \not\hookrightarrow^* p(e)) \wedge p'\ \overline{\Rrightarrow}\ p \end{array} \right] \\
p_1\ \overline{\Rrightarrow}\ p_2 &\triangleq p_1 = p_2 \vee p_1\ \overline{\Rrightarrow}\ p_2 \\
p_1\ \overline{\Rrightarrow_t}\ p_2 &\triangleq p_1 = p_2 \vee p_1\ \overline{\Rrightarrow_t}\ p_2 \\
p_1\ \overline{\Rrightarrow_{\overline{t}}}\ p_2 &\triangleq p_1 = p_2 \vee p_1\ \overline{\Rrightarrow_{\overline{t}}}\ p_2 \\
p_1\ \overline{\Rrightarrow_{st}}\ p_2 &\triangleq p_1 = p_2 \vee p_1\ \overline{\Rrightarrow_{st}}\ p_2 \\
p_1\ \overline{\Rrightarrow_{\overline{st}}}\ p_2 &\triangleq p_1 = p_2 \vee p_1\ \overline{\Rrightarrow_{\overline{st}}}\ p_2
\end{aligned}
$$

The call-graph properties and relations obey the following abstraction lemma:

LEMMA 7. *For all expressions e, invocations $\hat{e}$ of e, abstract procedures p, p_1, and p_2, and concrete procedures $k \in p$, $k_1 \in p_1$, and $k_2 \in p_2$: if* $\textsc{Called}(k)$, *then* $\textsc{Called}(p)$, *if* $\hat{e} \rhd k$, *then* $e \rhd p$, *if* $\hat{e} \rhd_t k$, *then* $e \rhd_t p$, *if* $\hat{e} \rhd_{\overline{t}} k$, *then* $e \rhd_{\overline{t}} p$, *if* $\hat{e} \rhd_{st} k$, *then* $e \rhd_{st} p$, *if* $\hat{e} \rhd_{\overline{st}} k$, *then* $e \rhd_{\overline{st}} p$, *if* $k_1 \rhd k_2$, *then* $p_1 \rhd p_2$, *if* $k_1 \rhd_t k_2$, *then* $p_1 \rhd_t p_2$, *if* $k_1 \rhd_{\overline{t}} k_2$, *then* $p_1 \rhd_{\overline{t}} p_2$, *if* $k_1 \rhd_{st} k_2$, *then* $p_1 \rhd_{st} p_2$, *if* $k_1 \rhd_{\overline{st}} k_2$, *then* $p_1 \rhd_{\overline{st}} p_2$, *if* $\hat{e} \Rrightarrow k$, *then* $e \Rrightarrow p$, *if* $\hat{e} \Rrightarrow_t k$, *then* $e \Rrightarrow_t p$, *if* $\hat{e} \Rrightarrow_{\overline{t}} k$, *then* $e \Rrightarrow_{\overline{t}} p$, *if* $\hat{e} \Rrightarrow_{st} k$, *then* $e \Rrightarrow_{st} p$, *if* $\hat{e} \Rrightarrow_{\overline{st}} k$, *then* $e \Rrightarrow_{\overline{st}} p$, *if* $k_1 \Rrightarrow k_2$, *then* $p_1 \Rrightarrow p_2$, *if* $k_1 \Rrightarrow_t k_2$, *then* $p_1 \Rrightarrow_t p_2$, *if* $k_1 \Rrightarrow_{\overline{t}} k_2$, *then* $p_1 \Rrightarrow_{\overline{t}} p_2$, *if* $k_1 \Rrightarrow_{st} k_2$, *then* $p_1 \Rrightarrow_{st} p_2$, *if* $k_1 \Rrightarrow_{\overline{st}} k_2$, *then* $p_1 \Rrightarrow_{\overline{st}} p_2$, *if* $k_1 \Rrightarrow k_2$, *then* $p_1 \Rrightarrow p_2$, *if* $k_1 \Rrightarrow_t k_2$, *then* $p_1 \Rrightarrow_t p_2$, *if* $k_1 \Rrightarrow_{\overline{t}} k_2$, *then* $p_1 \Rrightarrow_{\overline{t}} p_2$, *if* $k_1 \Rrightarrow_{st} k_2$, *then* $p_1 \Rrightarrow_{st} p_2$, *and if* $k_1 \Rrightarrow_{\overline{st}} k_2$, *then* $p_1 \Rrightarrow_{\overline{st}} p_2$.

The call-graph properties and relations also obey the following conservative approximation lemma:

LEMMA 8. *For all expressions e and abstract procedures p, p_1, and p_2: if*

$\textsc{Called}(k)$, *then* $\overline{\textsc{Called}}(p)$, *if* $e \rhd p$, *then* $e \mathbin{\overline{\rhd}} p$, *if* $e \rhd_t p$, *then* $e \mathbin{\overline{\rhd_t}} p$, *if* $e \rhd_{\overline{t}} p$, *then* $e \mathbin{\overline{\rhd_{\overline{t}}}} p$, *if* $e \rhd_{st} p$, *then* $e \mathbin{\overline{\rhd_{st}}} p$, *if* $e \rhd_{\overline{st}} p$, *then* $e \mathbin{\overline{\rhd_{\overline{st}}}} p$, *if* $p_1 \rhd p_2$, *then* $p_1 \mathbin{\overline{\rhd}} p_2$, *if* $p_1 \rhd_t p_2$, *then* $p_1 \mathbin{\overline{\rhd_t}} p_2$, *if* $p_1 \rhd_{\overline{t}} p_2$, *then* $p_1 \mathbin{\overline{\rhd_{\overline{t}}}} p_2$, *if* $p_1 \rhd_{st} p_2$, *then* $p_1 \mathbin{\overline{\rhd_{st}}} p_2$, *if* $p_1 \rhd_{\overline{st}} p_2$, *then* $p_1 \mathbin{\overline{\rhd_{\overline{st}}}} p_2$, *if* $e \Rrightarrow p$, *then* $e \mathbin{\overline{\Rrightarrow}} p$, *if* $e \Rrightarrow_t p$, *then* $e \mathbin{\overline{\Rrightarrow_t}} p$, *if* $e \Rrightarrow_{\overline{t}} p$, *then* $e \mathbin{\overline{\Rrightarrow_{\overline{t}}}} p$, *if* $e \Rrightarrow_{st} p$, *then* $e \mathbin{\overline{\Rrightarrow_{st}}} p$, *if* $e \Rrightarrow_{\overline{st}} p$, *then* $e \mathbin{\overline{\Rrightarrow_{\overline{st}}}} p$, *if* $p_1 \Rrightarrow p_2$, *then* $p_1 \mathbin{\overline{\Rrightarrow}} p_2$, *if* $p_1 \Rrightarrow_t p_2$, *then* $p_1 \mathbin{\overline{\Rrightarrow_t}} p_2$, *if* $p_1 \Rrightarrow_{\overline{t}} p_2$, *then* $p_1 \mathbin{\overline{\Rrightarrow_{\overline{t}}}} p_2$, *if* $p_1 \Rrightarrow_{st} p_2$, *then* $p_1 \mathbin{\overline{\Rrightarrow_{st}}} p_2$, *if* $p_1 \Rrightarrow_{\overline{st}} p_2$, *then* $p_1 \mathbin{\overline{\Rrightarrow_{\overline{st}}}} p_2$, *if* $p_1 \Rrightarrow p_2$, *then* $p_1 \mathbin{\overline{\Rrightarrow}} p_2$, *if* $p_1 \Rrightarrow_t p_2$, *then* $p_1 \mathbin{\overline{\Rrightarrow_t}} p_2$, *if* $p_1 \Rrightarrow_{\overline{t}} p_2$, *then* $p_1 \mathbin{\overline{\Rrightarrow_{\overline{t}}}} p_2$, *if* $p_1 \Rrightarrow_{st} p_2$, *then* $p_1 \mathbin{\overline{\Rrightarrow_{st}}} p_2$, *and if* $p_1 \Rrightarrow_{\overline{st}} p_2$, *then* $p_1 \mathbin{\overline{\Rrightarrow_{\overline{st}}}} p_2$.

3.6 Approximating the abstract called-more-than-once property

After approximating the abstract call-graph properties and relations, Stalin approximates the abstract called-more-than-once property. This information is used to determine which variables can be globalized. A variable can be globalized if it can have at most one live instance. A variable will have only one instance, hence at most one live instance, if the abstract procedure that binds that variable is not called more than once.

The concrete called-more-than-once property is defined as follows:

Definition 12. A concrete procedure k is *called more than once*, denoted $\textsc{CalledMoreThanOnce}(k)$, if there are two or more successful call invocations that target k.

Given the above, the abstract called-more-than-once property is defined as follows:

Definition 13. An abstract procedure p is *called more than once*, denoted $\textsc{CalledMoreThanOnce}(p)$, if some concrete procedure $k \in p$ is called more than once in some execution of the program.

Stalin approximates the property $\textsc{CalledMoreThanOnce}(p)$ as follows:

$$\overline{\textsc{CalledMoreThanOnce}}(p) \stackrel{\triangle}{=} \left(\begin{array}{l} p \neq p_0 \wedge \\ \left\{ \begin{array}{l} \left[(\exists e \in C) e \mathbin{\overline{\rhd}} p \wedge \overline{\textsc{CalledMoreThanOnce}}(p(e)) \right] \vee \\ (\exists e_1, e_2 \in C) e_1 \neq e_2 \wedge e_1 \mathbin{\overline{\rhd}} p \wedge e_2 \mathbin{\overline{\rhd}} p \end{array} \right\} \end{array} \right)$$

The called-more-than-once property obeys the following abstraction lemma:

Lemma 9. *For all abstract procedures p and concrete procedures $k \in p$, if* $\textsc{CalledMoreThanOnce}(k)$, *then* $\textsc{CalledMoreThanOnce}(p)$.

The called-more-than-once property also obeys the following conservative approximation lemma:

Lemma 10. *For all abstract procedures p, if* $\textsc{CalledMoreThanOnce}(p)$, *then* $\overline{\textsc{CalledMoreThanOnce}}(p)$.

3.7 Approximating the abstract free-in relation

After approximating the abstract called-more-than-once property, Stalin approximates the abstract free-in relation. This information is used to support to support

parent-slot, closure-pointer–slot, and parent-parameter compression and elimination. If an abstract procedure p_1 binds a variable x that is free in an abstract procedure p_2, then the closure for p_1 must be accessible to p_2 via the static backchain. Conversely, if an abstract procedure p_1 does not bind any variables that are free in an abstract procedure p_2, then the closure for p_1, if it exists, need not be accessible to p_2, allowing parent-slot, closure-pointer–slot, and parent-parameter compression and elimination.

Conventional implementations use a purely syntactic definition of the free-in relation: a variable x is free in an abstract procedure p that doesn't bind x if there are any references to x nested in p. STALIN uses a definition that is more precise in two ways. First, a variable is not free if it is never accessed, even if it is freely assigned. This is sound because STALIN eliminates unaccessed variables and assignments to unaccessed variables. Second, unreached accesses and unexecuted assignments to a variable are ignored. A variable can be free only if it has reached free accesses or executed free assignments. This is sound because STALIN eliminates unreached accesses and unexecuted assignments.

The concrete free-in relation is defined as follows:

Definition 14. A variable x is *free in* some procedure p, denoted $\textsc{FreeIn}(x, p)$, if p is properly nested in $p(x)$, x is accessed, and there is some reached access e or executed assignment e that references x such that $p(e)$ is nested in p.

STALIN approximates the relation $\textsc{FreeIn}(x, p)$ as follows:

$$\overline{\textsc{FreeIn}}(x,p) \triangleq \left(\begin{array}{l} p \prec^+ p(x) \wedge \overline{\textsc{Accessed}}(x) \wedge \\ \left\{ \begin{array}{l} \left[(\exists e \in A) \overline{\textsc{Reached}}(e) \wedge x(e) = x \wedge p(e) \prec^* p \right] \vee \\ \left[(\exists e \in S) \overline{\textsc{Executed}}(e) \wedge x(e) = x \wedge p(e) \prec^* p \right] \end{array} \right\} \end{array} \right)$$

The free-in relation obeys the following conservative approximation lemma:

LEMMA 11. *For all abstract procedures p and variables x, if* $\textsc{FreeIn}(x, p)$, *then* $\overline{\textsc{FreeIn}}(x, p)$.

3.8 Approximating the abstract points-to relation

After approximating the abstract free-in relation, STALIN approximates the abstract points-to relation. The abstract points-to relation is used to approximate the abstract escape relation which forms the basis of a must-alias analysis that is used to support variable-slot elimination, globalization, hiding, and determining when continuations are fictitious.

The concrete points-to relation is defined as follows:

Definition 15. A concrete object k *directly points to* a concrete location l, denoted $k \rightsquigarrow l$, if l is a slot or element of k. A concrete location l *directly points to* a concrete object k, denoted $l \rightsquigarrow k$, if k is the value of l. A concrete object k_1 *points to* a concrete object k_2, denoted $k_1 \rightsquigarrow\!\!\!\!\!\!\rightsquigarrow k_2$, if $k_1 \rightsquigarrow^* k_2$. A concrete object k *points to* a concrete location l, denoted $k \rightsquigarrow\!\!\!\!\!\!\rightsquigarrow l$, if $k \rightsquigarrow^* l$. A concrete location l *points to* a concrete object k, denoted $l \rightsquigarrow\!\!\!\!\!\!\rightsquigarrow k$, if $l \rightsquigarrow^* k$. A concrete location l_1 *points to* a concrete location l_2, denoted $l_1 \rightsquigarrow\!\!\!\!\!\!\rightsquigarrow l_2$, if $l_1 \rightsquigarrow^* l_2$.

Given the above, the abstract points-to relation is defined as follows:

Definition 16. An abstract object σ *directly points to* an abstract location β, denoted $\sigma \rightsquigarrow \beta$, if some concrete object in $k \in \sigma$ directly points to some concrete location $l \in \beta$ in some state in some execution of the program. An abstract location β *directly points to* an abstract object σ, denoted $\beta \rightsquigarrow \sigma$, if some concrete location $l \in \beta$ directly points to some concrete object $k \in \sigma$ in some state in some execution of the program. An abstract object σ_1 *points to* an abstract object σ_2, denoted $\sigma_1 \rightsquigarrow\!\!\!\!\!\rightsquigarrow \sigma_2$, if some concrete object $k_1 \in \sigma_1$ points to some concrete object $k_2 \in \sigma_2$ in some state in some execution of the program. An abstract object σ *points to* an abstract location β, denoted $\sigma \rightsquigarrow\!\!\!\!\!\rightsquigarrow \beta$, if some concrete object $k \in \sigma$ points to some concrete location $l \in \beta$ in some state in some execution of the program. An abstract location β *points to* an abstract object σ, denoted $\beta \rightsquigarrow\!\!\!\!\!\rightsquigarrow \sigma$, if some concrete location $l \in \beta$ points to some concrete object $k \in \sigma$ in some state in some execution of the program. An abstract location β_1 *points to* an abstract location β_2, denoted $\beta_1 \rightsquigarrow\!\!\!\!\!\rightsquigarrow \beta_2$, if some concrete location $l_1 \in \beta_1$ points to some concrete location $l_2 \in \beta_2$ in some state in some execution of the program.

Stalin approximates the $\rightsquigarrow$ relation as follows:

$$\beta \overline{\rightsquigarrow} \sigma \triangleq \sigma \overline{\in} \beta$$

$$\sigma \overline{\rightsquigarrow} \beta \triangleq \left[\begin{array}{l} \left\{\begin{array}{l} \sigma \in P \wedge \overline{\textsc{ProcedureAccessed}}(\sigma) \wedge \\ \left[(\exists x \in X)\overline{\textsc{Accessed}}(x) \wedge \beta = \beta(x) \wedge \overline{\textsc{FreeIn}}(x,\sigma)\right] \end{array}\right\} \vee \\ \textsc{Pair?}(\sigma) \wedge \left(\begin{array}{l} \textsc{Car}(\sigma) = \beta \wedge \overline{\textsc{CarAccessed}}(\sigma) \vee \\ \textsc{Cdr}(\sigma) = \beta \wedge \overline{\textsc{CdrAccessed}}(\sigma) \end{array}\right) \vee \\ \left(\begin{array}{l} \textsc{String?}(\sigma) \wedge \textsc{String-Ref}(\sigma) = \beta \wedge \\ \overline{\textsc{StringRefAccessed}}(\sigma) \end{array}\right) \vee \\ \left(\begin{array}{l} \textsc{Vector?}(\sigma) \wedge \textsc{Vector-Ref}(\sigma) = \beta \wedge \\ \overline{\textsc{VectorRefAccessed}}(\sigma) \end{array}\right) \vee \\ \left(\begin{array}{l} \textsc{Symbol?}(\sigma) \wedge \textsc{Symbol->String}(\sigma) = \beta \wedge \\ \overline{\textsc{SymbolToStringAccessed}}(\sigma) \end{array}\right) \end{array}\right]$$

Stalin then approximates the $\rightsquigarrow\!\!\!\!\!\rightsquigarrow$ relation, denoted $\overline{\rightsquigarrow\!\!\!\!\!\rightsquigarrow}$, as $\overline{\rightsquigarrow}^*$.

The points-to relation obeys the following abstraction lemma:

Lemma 12. *For all pairs of abstract objects σ_1 and σ_2, concrete objects $k_1 \in \sigma_1$, concrete objects $k_2 \in \sigma_2$, pairs of abstract locations β_1 and β_2, concrete locations $l_1 \in \beta_1$, and concrete locations $l_2 \in \beta_2$, if, in some state in some execution of the program, $k_1 \rightsquigarrow l_1$, $l_1 \rightsquigarrow k_1$, $k_1 \rightsquigarrow\!\!\!\!\!\rightsquigarrow k_2$, $k_1 \rightsquigarrow\!\!\!\!\!\rightsquigarrow l_2$, $l_1 \rightsquigarrow\!\!\!\!\!\rightsquigarrow k_2$, or $l_1 \rightsquigarrow\!\!\!\!\!\rightsquigarrow l_2$, then $\sigma_1 \rightsquigarrow \beta_1$, $\beta_1 \rightsquigarrow \sigma_1$, $\sigma_1 \rightsquigarrow\!\!\!\!\!\rightsquigarrow \sigma_2$, $\sigma_1 \rightsquigarrow\!\!\!\!\!\rightsquigarrow \beta_2$, $\beta_1 \rightsquigarrow\!\!\!\!\!\rightsquigarrow \sigma_2$, or $\beta_1 \rightsquigarrow\!\!\!\!\!\rightsquigarrow \beta_2$ respectively.*

The points-to relation also obeys the following conservative approximation lemma:

Lemma 13. *For all pairs of abstract objects σ_1 and σ_2 and pairs of abstract locations β_1 and β_2, if $\sigma_1 \rightsquigarrow \beta_1$, $\beta_1 \rightsquigarrow \sigma_1$, $\sigma_1 \rightsquigarrow\!\!\!\!\!\rightsquigarrow \sigma_2$, $\sigma_1 \rightsquigarrow\!\!\!\!\!\rightsquigarrow \beta_2$, $\beta_1 \rightsquigarrow\!\!\!\!\!\rightsquigarrow \sigma_2$, or $\beta_1 \rightsquigarrow\!\!\!\!\!\rightsquigarrow \beta_2$, then $\sigma_1 \overline{\rightsquigarrow} \beta_1$, $\beta_1 \overline{\rightsquigarrow} \sigma_1$, $\sigma_1 \overline{\rightsquigarrow\!\!\!\!\!\rightsquigarrow} \sigma_2$, $\sigma_1 \overline{\rightsquigarrow\!\!\!\!\!\rightsquigarrow} \beta_2$, $\beta_1 \overline{\rightsquigarrow\!\!\!\!\!\rightsquigarrow} \sigma_2$, or $\beta_1 \overline{\rightsquigarrow\!\!\!\!\!\rightsquigarrow} \beta_2$ respectively.*

3.9 Approximating the abstract escape relation

After approximating the abstract points-to relation, Stalin approximates the abstract escape relation. The abstract escape relation forms the basis of a must-alias

analysis that is used to support variable-slot elimination, globalization, hiding, and determining when continuations are fictitious.

Before defining the concrete escape relation, the following auxiliary definitions are needed:

Definition 17. A reached lambda expression invocation *creates* a concrete procedure. A successful primcall invocation to `cons` *creates* a concrete pair. A successful primcall invocation to `string` or `make-string` *creates* a concrete string. A successful primcall invocation to `vector` or `make-vector` *creates* a concrete vector. A successful primcall invocation to `string->symbol` *creates* a symbol. A successful primcall invocation to `call/cc` *creates* a continuation. An expression invocation that type-tag, eq?, car, cdr, string-length, string-ref, vector-length, vector-ref, symbol-to-string, or continuations accesses a concrete object *accesses* that object. An expression invocation that car, cdr, string-ref, or vector-ref assigns a concrete object *assigns* that object. A reached access invocation $\hat{e}$ *accesses* a concrete procedure k, if k binds $\hat{x}(\hat{e})$. An executed assignment invocation $\hat{e}$ *assigns* a concrete procedure k, if k binds $\hat{x}(\hat{e})$.

Given the above, concrete escape relation is defined as follows:

Definition 18. A concrete object k *escapes* escapes an expression invocation $\hat{e}$, denoted $k \Uparrow \hat{e}$, if k is created after $\hat{e}$ is reached but before $\hat{e}$ returns and is accessed after $\hat{e}$ returns.

Given the above, the abstract escape relation is defined as follows:

Definition 19. An abstract object σ *escapes* an expression e, denoted $\sigma \Uparrow e$, if some concrete object $k \in \sigma$ escapes some invocation $\hat{e}$ of e in some execution of the program.

The escape relation will be needed only when e is the body of a lambda expression. This leads to the following definition:

Definition 20. An abstract object σ *escapes* an abstract procedure p, denoted $\sigma \Uparrow p$, if σ escapes $\textsc{Body}(p)$.

STALIN approximates the relation $\sigma \Uparrow p$ as follows:

$$\sigma \overline{\Uparrow} p \triangleq \left\{ \begin{array}{l} \overline{\textsc{Returns}}(\textsc{Body}(p)) \wedge \beta(\textsc{Body}(p)) \mathrel{\overline{\rightsquigarrow}} \sigma \vee \\ \left(\begin{array}{l} \exists\ e \in S, \\ \quad e' \in E, \\ \quad p' \in P \end{array} \right) \left[\begin{array}{l} \overline{\textsc{Executed}}(e) \wedge \overline{\textsc{Returns}}(e') \wedge \\ \overline{\textsc{Accessed}}(x(e)) \wedge p(e') \mathrel{\overline{\triangleright}} p \mathrel{\overline{\triangleright}} p(e) \wedge \\ \beta(\textsc{Source}(e)) \mathrel{\overline{\rightsquigarrow}} \sigma \wedge \beta(e') \mathrel{\overline{\rightsquigarrow}} p' \wedge \\ p' \prec^{+} p(x(e)) \wedge \\ (\exists e'' \in A) \left(\begin{array}{l} \overline{\textsc{Reached}}(e'') \wedge x(e'') = x(e) \wedge \\ p(e'') \prec^{*} p' \end{array} \right) \end{array} \right] \vee \\ \left(\begin{array}{l} \exists\ e \in C_1, \\ \quad p' \in P, \\ \quad \sigma' \mathrel{\overline{\in}} \beta(\textsc{Callee}(e)) \end{array} \right) \left(\begin{array}{l} \overline{\textsc{Executed}}(e) \wedge p' \mathrel{\overline{\triangleright}} p \mathrel{\overline{\triangleright}} p(e) \wedge \\ \textsc{Continuation?}(\sigma') \wedge \\ \beta(\textsc{Argument}_1(e)) \mathrel{\overline{\rightsquigarrow}} \sigma \wedge \\ p' \mathrel{\overline{\in}} \beta(\textsc{Argument}_2(e(\sigma'))) \end{array} \right) \vee \\ \left(\begin{array}{l} \exists\ e \in C_2, \\ \quad e' \in E, \\ \quad \sigma' \mathrel{\overline{\in}} \beta(\textsc{Argument}_1(e)) \end{array} \right) \left[\begin{array}{l} \overline{\textsc{Executed}}(e) \wedge \overline{\textsc{Returns}}(e') \wedge \\ p(e') \mathrel{\overline{\triangleright}} p \mathrel{\overline{\triangleright}} p(e) \wedge \textsc{Pair?}(\sigma') \wedge \\ \left(\begin{array}{l} \textsc{Callee}(e) = \texttt{set-car!} \vee \\ \textsc{Callee}(e) = \texttt{set-cdr!} \end{array} \right) \wedge \\ \beta(\textsc{Argument}_2(e)) \mathrel{\overline{\rightsquigarrow}} \sigma \wedge \\ \beta(e') \mathrel{\overline{\rightsquigarrow}} \sigma' \end{array} \right] \vee \\ \left(\begin{array}{l} \exists\ e \in C_3, \\ \quad e' \in E, \\ \quad \sigma' \mathrel{\overline{\in}} \beta(\textsc{Argument}_1(e)) \end{array} \right) \left(\begin{array}{l} \overline{\textsc{Executed}}(e) \wedge \overline{\textsc{Returns}}(e') \wedge \\ p(e') \mathrel{\overline{\triangleright}} p \mathrel{\overline{\triangleright}} p(e) \wedge \beta(e') \mathrel{\overline{\rightsquigarrow}} \sigma' \wedge \\ \textsc{Vector?}(\sigma') \wedge \\ \textsc{Callee}(e) = \texttt{vector-set!} \wedge \\ \textbf{number} \mathrel{\overline{\in}} \beta(\textsc{Argument}_2(e)) \wedge \\ \beta(\textsc{Argument}_3(e)) \mathrel{\overline{\rightsquigarrow}} \sigma \end{array} \right) \end{array} \right\}$$

Note that calls to `string-set!` do not cause their third argument to escape because that argument must always be a character and characters are not created.

The escape relation obeys the following abstraction lemma:

LEMMA 14. *For all abstract objects σ, expressions e, concrete objects $k \in \sigma$, and invocations $\hat{e}$ of e, if $k \Uparrow \hat{e}$, then $\sigma \Uparrow e$.*

The escape relation also obeys the following conservative approximation lemma:

LEMMA 15. *For all abstract objects σ and abstract procedures p, if $\sigma \Uparrow p$, then $\sigma \overline{\Uparrow} p$.*

3.10 Deciding which procedures to in-line

After approximating the abstract escape relation, STALIN decides which procedures to in-line. The resulting in-lined-in relation is important since an accessed, non-fictitious, non-global variable slot can be eliminated only if all reached accesses and executed assignments are in-lined in the procedure that binds that variable.

STALIN in-lines procedures that have a unique call site. STALIN translates self tail calls as `goto`s. Such self tail calls do not count as call sites when making in-lining decisions. Moreover, the notion of self tail call used by STALIN is sensitive to in-lining decisions.

Definition 21. A call e is a *unique call site* of p, denoted $\textsc{UniqueCallSite}(e, p)$, if e directly non-self-tail calls p and there is no other call e' that directly non-self-

tail calls p. If e is the unique call site of p then p is *directly in-lined in* $p(e)$, denoted $p \hookrightarrow p(e)$. Let $\hookrightarrow^*$ denote the reflexive-transitive closure of $\hookrightarrow$. If $p_1 \hookrightarrow^* p_2$, then p_1 is *in-lined in* p_2.

The above definition must be approximated, because the direct non-self-tail call relation must be approximated. Furthermore, the above definition is circular. The directly–in-lined-in relation $\hookrightarrow$ depends on the notion of unique call site, which depends on the notion of direct non-self-tail call, which depends on the relation $\hookrightarrow^*$, which depends back on the relation $\hookrightarrow$. To approximate the in-lining relation and break this circularity, STALIN finds a least fixed point to approximations of the above definitions. In other words, it finds a minimal relation $\hookrightarrow$ that satisfies the following constraints:

$$\begin{aligned} e \mathrel{\overline{\rhd_{st}}} p &\stackrel{\triangle}{=} e \mathrel{\overline{\rhd}} p \wedge (\neg \textsc{InTailPosition}(e) \vee p(e) \not\hookrightarrow^* p) \\ \overline{\textsc{UniqueCallSite}}(e, p) &\stackrel{\triangle}{=} e \mathrel{\overline{\rhd_{st}}} p \wedge \neg(\exists e' \in E) e' \neq e \wedge e' \mathrel{\overline{\rhd_{st}}} p \\ p_1 \hookrightarrow p_2 &\stackrel{\triangle}{=} (\exists e \in C) p_2 = p(e) \wedge \overline{\textsc{UniqueCallSite}}(e, p_1) \end{aligned}$$

It may seem overly restrictive to in-line only procedures that have a unique call site. STALIN overcomes this restriction by doing a polyvariant flow analysis and splitting procedures, assigning different copies to different call sites. As a result of such splitting, individual copies might have a unique call site, and thus might be in-lined, even if the original procedure would have multiple call sites under a monovariant analysis. The details of the polyvariant flow analysis and splitting procedure are beyond the scope of this paper. They are discussed in detail in Siskind [2000b].

3.11 Approximating the reentrant property

After deciding which procedures to in-line, STALIN approximates the reentrant property. The reentrant property is used to justify globalization. A variable can be globalized when it has at most one live instance. Variables bound by reentrant procedures can have more than one live instance, one for each reentrant invocation. Thus non-reentrancy is one requirement for globalization.

A procedure is recursive if it can call itself. A procedure is reentrant if it can have more than one live activation record. Since tail-merged calls do not create new activation records, not all recursive procedures are reentrant. In an implementation that merged all tail calls, a procedure would be reentrant only if it properly non-tail called itself. STALIN, however, does not merge all tail calls. It only merges self tail calls. Because of this, the concrete reentrant property is defined as follows:

Definition 22. A concrete procedure k is *reentrant*, denoted $\textsc{Reentrant}(k)$, if k properly non-self-tail calls itself.

Given the above, the abstract reentrant property is defined as follows:

Definition 23. An abstract procedure p is *reentrant*, denoted $\textsc{Reentrant}(p)$, if some concrete procedure $k \in p$ is reentrant in some execution of the program.

STALIN approximates the property $\textsc{Reentrant}(p)$, denoted $\overline{\textsc{Reentrant}}(p)$, as $p \mathrel{\overline{\rhd\!\!\rhd_{st}}} p$.

The reentrant property obeys the following abstraction lemma:

LEMMA 16. *For all abstract procedures p and concrete procedures $k \in p$, if* REENTRANT(k)*, then* REENTRANT(p)*.*

The reentrant property also obeys the following conservative approximation lemma:

LEMMA 17. *For all abstract procedures p, if* REENTRANT(p)*, then* $\overline{\text{REENTRANT}}(p)$*.*

3.12 Approximating the abstract must-alias properties

Sound lightweight closure conversion requires determining a number of abstract must-alias properties. These are defined as follows:

Definition 24. A variable x *must alias*, denoted MUSTALIAS(x), if every reached access invocation $\hat{e}$ to x accesses the instance $\hat{x}$ of x bound by the most recent active invocation of some concrete procedure $k \in p(x)$. An abstract continuation σ *must alias*, denoted MUSTALIAS(σ), if for every successful call invocation $\hat{e}$ to some concrete continuation $k \in \sigma$, k was created by the most recent active invocation of $e(\sigma)$. An abstract procedure p *must alias*, denoted MUSTALIAS(p), either if p has no parent parameter or if for every successful call invocation $\hat{e}$ to some concrete procedure $k \in p$, the most recent active invocation of PARENTPARAMETER(p) when k is called is the same as the most recent active invocation of PARENTPARAMETER(p) when k was created.

The variable must-alias property is used to justify variable-slot elimination and globalization. The abstract-continuation must-alias property is used to determine when continuations are fictitious. The abstract-procedure must-alias property is used to justify hiding.

The variable must-alias property can be violated in only two ways: either

a. some reference is to an instance in an invocation that has already returned or

b. some reference is to an instance in a recursive invocation that is not the current invocation.

Condition (a) occurs only when the following situation arises:

```
(lambda (...x...)
 ⋮
 (lambda (...) ... x_e or (set! x ...)_e ...)_p ...)_p(x)
```

Here, a nontrivial reference e to x is nested in p, which is, in turn, properly nested in $p(x)$, and p escapes $p(x)$. Note that $p(e)$ could be the same as p but, in order for p to escape $p(x)$, p cannot be the same as $p(x)$. Calling p after p escapes $p(x)$ allows a reference to x after $p(x)$ returns.

Condition (b) occurs only when the following situation arises:

```
(lambda (...x'...)
 ⋮
 (lambda (...x...)
  ...x'_e'...
  (lambda (...) ... x_e or (set! x ...)_e ...)_p ...)_p(x) ...)_p(x')
```

Here, $p(x)$ is nested in $p(x')$, $p(x)$ is recursive, a nontrivial reference e to x is nested in p, which is, in turn, properly nested in and properly called by $p(x)$, and an access e' to x' is nested in $p(x)$, where $\beta(e')$ points to p. Note that $p(x)$ can be the same as $p(x')$, $p(e')$ can be the same as $p(x)$, $p(e)$ can be the same as p, and e' can be nested anywhere in $p(x)$, including inside p. Passing p, which contains a reference to x, created on one invocation of $p(x)$, to a recursive invocation, and calling p in that recursive invocation, allows an access to x in other than the most recent invocation. Such passing can only happen via an x'. Note that p cannot be the same as $p(x)$ in order for e to access a different instance of x than the one in the most recent invocation.

STALIN approximates the property $\textsc{MustAlias}(x)$ as follows:

$$\overline{\textsc{MustAlias}}(x) \triangleq$$
$$\left[\begin{array}{l} \neg(\exists e \in A \cup S, p \in P)\left(\begin{array}{l} x(e) = x \wedge p(e) \prec^* p \prec^+ p(x) \wedge \\ \overline{\textsc{NonTrivialReference}}(e) \wedge p \overline{\Uparrow} p(x) \end{array}\right) \wedge \\ \neg(\exists e \in A \cup S, e' \in A, p \in P)\left(\begin{array}{l} \overline{\textsc{NonTrivialReference}}(e) \wedge \\ p(e') \prec^* p(x) \prec^* p(x(e')) \wedge \\ p(e) \prec^* p \prec^+ p(x) \wedge \\ p(x) \overline{\Rightarrow} p(x) \wedge x(e) = x \wedge \\ \beta(e') \overline{\not\rightsquigarrow} p \wedge p(x) \overline{\Rightarrow} p \end{array}\right) \end{array}\right]$$

Note that the above approximation uses the property $\overline{\textsc{NonTrivialReference}}(e)$ which will be defined in section 3.14.

The abstract-continuation must-alias property can be violated in only two ways: either

a. the continuation is accessed after the expression invocation that created that continuation returns or

b. the continuation is accessed when there is an active recursive invocation of the expression that created that continuation that is not the invocation that created that continuation.

Condition (a) occurs only when the following situation arises:

$$\texttt{(call/cc ...)}_{e(\sigma)} \ \ldots \ e$$

Here, σ escapes $e(\sigma)$ and some expression e accesses σ.

Condition (b) occurs only when the following situation arises:

$$\begin{array}{l} \texttt{(lambda (...) ...}\, e \,\texttt{...)}_{p(e)} \\ \vdots \\ \texttt{(lambda (...}\, x \,\texttt{...)} \\ \quad \vdots \\ \quad \texttt{(lambda (}x'\texttt{) ...}\, x_{e'} \,\texttt{...)}_{p} \\ \quad \vdots \\ \quad \texttt{(lambda (...) ... (call/cc ...)}_{e(\sigma)} \ \texttt{...)}_{p(e(\sigma))} \ \texttt{...)}_{p(x)} \end{array}$$

Here, a call e accesses the abstract continuation σ, p is passed σ by virtue of the fact that it is called by the call $e(\sigma)$ to `call/cc`, p is properly nested in $p(x)$,

$p(e(\sigma))$ is nested in $p(x)$, p and $p(e(\sigma))$ properly call each other, and an access e' to x is nested in p, where $\beta(e')$ points to σ.

Stalin approximates the property $\textsc{MustAlias}(\sigma)$ as follows:

$$\overline{\textsc{MustAlias}}(\sigma) \triangleq$$
$$\left\{ \begin{array}{l} \neg \left[\begin{array}{l} \left(\overline{\textsc{TypeTagAccessed}}(\sigma) \vee \overline{\textsc{ContinuationAccessed}}(\sigma)\right) \wedge \\ (\exists p \,\overline{\in}\, \beta(\textsc{Argument}_1(e(\sigma)))) \left(\begin{array}{l} p \in P \wedge \sigma \,\overline{\in}\, \beta(\textsc{Parameter}_1(p)) \wedge \\ \sigma \overline{\Uparrow} p \end{array} \right) \end{array} \right] \wedge \\ \neg \left(\begin{array}{l} \exists\ e \in C, \\ \quad e' \in A, \\ \quad p \,\overline{\in}\, \beta(\textsc{Argument}_1(e(\sigma))) \end{array} \right) \left[\begin{array}{l} p \,\overline{\Bbbrack}\, p(e(\sigma)) \,\overline{\Bbbrack}\, p \wedge p \,\overline{\Bbbrack}\, p(e) \wedge \\ \beta(e') \,\overline{\rightsquigarrow}\, \sigma \wedge p \prec^+ p(x(e')) \wedge \\ p(e(\sigma)) \prec^* p(x(e')) \wedge \\ \left(\begin{array}{l} \overline{\textsc{TypeTagAccesses}}(e, \sigma) \vee \\ \overline{\textsc{ContinuationAccesses}}(e, \sigma) \end{array} \right) \end{array} \right] \end{array} \right\}$$

Note that $\overline{\textsc{Eq?Accessed}}(p)$ is not needed in the above since R4RS does not specify the conditions for equality between continuations.

The abstract-procedure must-alias property can be violated in only two ways: either

a. p is accessed after $\textsc{ParentParameter}(p)$ returns or

b. p is accessed when there is an active recursive invocation of $\textsc{ParentParameter}(p)$ that is not the invocation that created p.

Condition (a) occurs only when the following situation arises:

```
(lambda (...)
 ⋮
 (lambda (...) ...)_p ...)_ParentParameter(p)
⋮
e
```

Here, p escapes $\textsc{ParentParameter}(p)$ and some expression e accesses p.

Condition (b) occurs only when the following situation arises:

```
(lambda (...) ... e ...)_p(e)
⋮
(lambda (... x ...) ... (lambda (...) ... x_e' ...)_p' ...)_p(x)
```

Here $p(e')$ is nested in p' which is in turn nested in $p(x(e'))$, $\textsc{ParentParameter}(p)$ is nested in p', $\beta(e')$ points to p, p' is recursive, p' calls $p(e)$, and e accesses p.

STALIN approximates the property MUSTALIAS(p) as follows:

$$\overline{\text{MUSTALIAS}}(p) \triangleq$$
$$\left\{ \begin{array}{ll} \neg \left[\begin{array}{l} \text{HASPARENTPARAMETER}(p) \wedge p \overline{\Uparrow} \text{PARENTPARAMETER}(p) \wedge \\ \left(\overline{\text{TYPETAGACCESSED}}(p) \vee \overline{\text{PROCEDUREACCESSED}}(p)\right) \end{array} \right] \wedge & \\ \neg(\exists e \in C, e' \in A, p' \in P) & \left[\begin{array}{l} p(e') \prec^* p' \prec^* p(x(e')) \wedge p' \overline{\Bbbk} p' \wedge \\ \beta(e') \overline{\rightsquigarrow} p \wedge p' \overline{\Bbbk} p(e) \wedge \\ \text{HASPARENTPARAMETER}(p) \wedge \\ \text{PARENTPARAMETER}(p) \prec^* p' \wedge \\ \left(\overline{\text{TYPETAGACCESSES}}(e, p) \vee e \overline{\rhd} p\right) \end{array} \right] \end{array} \right\}$$

Note that the above approximation uses the property HASPARENTPARAMETER(p) and the function PARENTPARAMETER(p) which will be defined in section 3.19. Also note that $\overline{\text{EQ?ACCESSED}}(p)$ is not needed in the above since R4RS does not specify the conditions for equality between procedures.

The must-alias properties obey the following conservative approximation lemma:

LEMMA 18. *For all variables x, if* $\neg$MUSTALIAS(x)*, then* $\neg\overline{\text{MUSTALIAS}}(x)$*. For all abstract continuations σ, if* $\neg$MUSTALIAS(σ)*, then* $\neg\overline{\text{MUSTALIAS}}(\sigma)$*. For all abstract procedures p, if* $\neg$MUSTALIAS(p)*, then* $\neg\overline{\text{MUSTALIAS}}(p)$*.*

3.13 Approximating the abstract fictitious property

STALIN is able to eliminate locations, and the code to manipulate such locations, when they are known to always hold the same concrete object. Such data is called *fictitious*.

The concrete fictitious property is defined as follows:

Definition 25. A concrete location l is *fictitious*, denoted FICTITIOUS(l), if it always contains the same concrete object.

Given the above, the abstract fictitious property is defined as follows:

Definition 26. An abstract location β is *fictitious*, denoted FICTITIOUS(β), if every $l \in \beta$ is fictitious in every execution of the program.

STALIN approximates the fictitious property by declaring an abstract location to be fictitious if it contains a single abstract object that, in turn, contains a single concrete object. The particular abstract interpretation that STALIN uses puts each of the concrete objects `()`, `#t`, `#f`, and **eof-object** into abstract objects that contain only that concrete object. Furthermore, if an abstract procedure p is not procedure accessed or does not have a parent parameter then all of the concrete procedures $k \in p$ are indistinguishable so p can be treated as if it contained a single concrete procedure. Additionally, if all of the components of some abstract pair, string, vector, or external symbol σ are never accessed, or contain fictitious locations, then all of the concrete objects $k \in \sigma$ are indistinguishable so σ can be treated as if it contained a single concrete object. Finally, STALIN compiles a call to an abstract continuation σ as a `goto` when

a. the call to σ is in-lined in the call to `call/cc` that created σ and

b. σ must alias.

This is described in greater detail in Siskind [2000a]. When this occurs for all calls to σ, or when σ is not continuation accessed, all of the concrete continuations $k \in \sigma$ are indistinguishable so σ can be treated as if it contained a single concrete continuation.

STALIN approximates the property FICTITIOUS(β) as follows:

$$
\begin{array}{l}
\overline{\textsc{Fictitious}}(\sigma) \triangleq \\
\left(\begin{array}{l}
\neg\overline{\textsc{TypeTagAccessed}}(\sigma)\wedge \\
\left\{\begin{array}{l}
\sigma = \texttt{()} \vee \sigma = \texttt{\#t} \vee \sigma = \texttt{\#f} \vee \sigma = \textbf{eof-object}\vee \\
\sigma \in P \wedge \left(\begin{array}{l} \neg\overline{\textsc{ProcedureAccessed}}(\sigma)\vee \\ \neg\textsc{HasParentParameter}(\sigma)) \end{array}\right) \vee \\
\left[\begin{array}{l} \textsc{Pair?}(\sigma) \wedge \neg\overline{\textsc{Eq?Accessed}}(\sigma)\wedge \\ \left(\overline{\textsc{Fictitious}}(\textsc{Car}(\sigma)) \vee \neg\overline{\textsc{CarAccessed}}(\sigma)\right) \wedge \\ \left(\overline{\textsc{Fictitious}}(\textsc{Cdr}(\sigma)) \vee \neg\overline{\textsc{CdrAccessed}}(\sigma)\right) \end{array}\right] \vee \\
\left(\begin{array}{l} \textsc{String?}(\sigma) \wedge \neg\overline{\textsc{Eq?Accessed}}(\sigma)\wedge \\ \neg\overline{\textsc{StringLengthAccessed}}(\sigma) \wedge \neg\overline{\textsc{StringRefAccessed}}(\sigma) \end{array}\right) \vee \\
\left[\begin{array}{l} \textsc{Vector?}(\sigma) \wedge \neg\overline{\textsc{Eq?Accessed}}(\sigma)\wedge \\ \neg\overline{\textsc{VectorLengthAccessed}}(\sigma)\wedge \\ \left(\begin{array}{l} \overline{\textsc{Fictitious}}(\textsc{Vector-Ref}(\sigma))\vee \\ \neg\overline{\textsc{VectorRefAccessed}}(\sigma) \end{array}\right) \end{array}\right] \vee \\
\left(\begin{array}{l} \textsc{Symbol?}(\sigma) \wedge \neg\overline{\textsc{Eq?Accessed}}(\sigma)\wedge \\ \neg\overline{\textsc{SymbolToStringAccessed}}(\sigma) \end{array}\right) \vee \\
\left(\begin{array}{l} \textsc{Continuation?}(\sigma)\wedge \\ \left\{\begin{array}{l} \neg\overline{\textsc{ContinuationAccessed}}(\sigma)\vee \\ \left[\begin{array}{l} \overline{\textsc{MustAlias}}(\sigma)\wedge \\ (\forall e \in C_1) \\ \overline{\textsc{ContinuationAccesses}}(e,\sigma) \rightarrow p(e) \hookrightarrow^* p(e(\sigma)) \end{array}\right] \end{array}\right\} \end{array}\right) \end{array}\right\} \end{array}\right) \\
\overline{\textsc{Fictitious}}(\beta) \triangleq ||\beta|| = 1 \wedge (\forall\sigma \mathrel{\overline{\in}} \beta)\overline{\textsc{Fictitious}}(\sigma)
\end{array}
$$

Note that the above approximation uses the property HASPARENTPARAMETER(p) which will be defined in section 3.19.

The fictitious property obeys the following abstraction lemma:

LEMMA 19. *For all abstract locations β and concrete locations $l \in \beta$, if* $\neg$FICTITIOUS(l)*, then* $\neg$FICTITIOUS(β)*.*

The fictitious property also obeys the following conservative approximation lemma:

LEMMA 20. *For all abstract locations β, if* $\neg$FICTITIOUS(β)*, then* $\neg\overline{\textsc{Fictitious}}(\beta)$*.*

3.14 Approximating the abstract nontrivial-reference property

STALIN does not generate any code for unreached accesses, unexecuted assignments, and assignments to fictitious or hidden variables. Furthermore, an access to a variable which can only yield a single compile-time determinable object need not generate code to access that variable and can instead generate code to return that object as a constant. All other references are called *nontrivial*. Only nontrivial references count as free references.

The concrete nontrivial-reference property is defined as follows:

Definition 27. An access invocation $\hat{e}$ is *nontrivial*, denoted NonTrivialReference($\hat{e}$), if it is reached and returns an object that is not known at compile time. An assignment invocation $\hat{e}$ is *nontrivial*, denoted NonTrivialReference($\hat{e}$), if it is executed, $x(e)$ is not hidden, and there is a subsequent nontrivial access to $\hat{x}(\hat{e})$.

Given the above, the abstract nontrivial-reference property is defined as follows:

Definition 28. A reference e is *nontrivial*, denoted NonTrivialReference(e), if some invocation $\hat{e}$ of e is nontrivial in some execution of the program.

Stalin approximates the property NonTrivialReference(e) as follows:

$$\overline{\textsc{NonTrivialReference}}(e) \triangleq \left[\begin{array}{l} \left(e \in A \wedge \overline{\textsc{Reached}}(e) \wedge \neg\overline{\textsc{Fictitious}}(\beta(e))\right) \vee \\ \left(\begin{array}{l} e \in S \wedge \overline{\textsc{Executed}}(e) \wedge \overline{\textsc{Accessed}}(x(e)) \wedge \\ \neg\overline{\textsc{Fictitious}}(\beta(x(e))) \wedge \neg\textsc{Hidden}(x(e)) \end{array} \right) \end{array} \right]$$

Note that the above approximation uses the property Hidden(x) which will be defined in section 3.19.

The nontrivial-reference property obeys the following abstraction lemma:

Lemma 21. *For all references* e *and invocations* $\hat{e}$ *of* e*, if* NonTrivialReference($\hat{e}$)*, then* NonTrivialReference(e).

The nontrivial-reference property also obeys the following conservative approximation lemma:

Lemma 22. *For all references* e*, if* NonTrivialReference(e)*, then* $\overline{\textsc{NonTrivialReference}}(e)$.

3.15 Approximating the abstract localizable property

In order to eliminate a variable slot and represent a variable as a C local variable it must be *localizable*.

The abstract localizable property is defined as follows:

Definition 29. A variable x is *localizable*, denoted Localizable(x), if x must alias and all nontrivial references to x are in-lined in the procedure that binds x.

Stalin approximates the property Localizable(x) as follows:

$$\overline{\textsc{Localizable}}(x) \triangleq \left[\begin{array}{l} \overline{\textsc{MustAlias}}(x) \wedge \\ \neg(\exists e \in A \cup S) x(e) = x \wedge \overline{\textsc{NonTrivialReference}}(e) \wedge p(e) \not\rightsquigarrow^* p(x) \end{array} \right]$$

The localizable property obeys the following conservative approximation lemma:

Lemma 23. *For all variables* x*, if* $\neg$Localizable(x)*, then* $\neg\overline{\textsc{Localizable}}(x)$.

3.16 Approximating the abstract globalizable property

In order to represent a variable as a C global variable it must be *globalizable*.

The abstract globalizable property is defined as follows:

Definition 30. A variable x is *globalizable*, denoted $\textsc{Globalizable}(x)$, if it can have at most one live instance.

If the procedure that binds a variable is not called more than once then that variable can have at most one live instance. Also, if the procedure that binds the variable is not reentrant and the variable must alias then that variable can have at most one live instance.

Stalin approximates the property $\textsc{Globalizable}(x)$ as follows:

$$\overline{\textsc{Globalizable}}(x) \stackrel{\triangle}{=} \left(\begin{array}{l} \neg\overline{\textsc{CalledMoreThanOnce}}(p(x)) \vee \\ \neg\overline{\textsc{Reentrant}}(p(x)) \wedge \overline{\textsc{MustAlias}}(x) \end{array} \right)$$

The globalizable property obeys the following conservative approximation lemma:

Lemma 24. *For all variables x, if* $\neg\textsc{Globalizable}(x)$*, then* $\neg\overline{\textsc{Globalizable}}(x)$.

3.17 Approximating the abstract ancestor relation

Certain expressions, such as free references to slotted and hidden variables, as well as lambda expressions that contain such free references, access the parent parameter and parent slots of procedures to get a pointer to the requisite closure.

The abstract ancestor relation is defined as follows:

Definition 31. A procedure p_1 is an *ancestor* of a procedure p_2, denoted $\textsc{Ancestor}(p_1, p_2)$, if some reached expression in p_2 accesses a pointer to the closure for p_1.

Stalin approximates the relation $\textsc{Ancestor}(p_1, p_2)$ as follows:

$$\overline{\textsc{Ancestor}}(p_1, p_2) \stackrel{\triangle}{=}$$
$$(\exists x \in X) \left\{ \begin{array}{l} \textsc{Slotted}(x) \wedge p(x) = p_1 \wedge \overline{\textsc{FreeIn}}(x, p_2) \vee \\ \left[\begin{array}{l} \textsc{Hidden}(x) \wedge (p(x) = p_2 \vee \overline{\textsc{FreeIn}}(x, p_2)) \\ (\exists p \in P)\beta(x) = \{p\} \wedge \overline{\textsc{Ancestor}}(p_1, p) \end{array} \right] \end{array} \right\}$$

Note that the above approximation uses the properties $\textsc{Slotted}(x)$ and $\textsc{Hidden}(x)$ which will be defined in section 3.19.

The ancestor relation obeys the following conservative approximation lemma:

Lemma 25. *For all abstract procedures p_1 and p_2, if* $\textsc{Ancestor}(p_1, p_2)$*, then* $\overline{\textsc{Ancestor}}(p_1, p_2)$.

3.18 Approximating the abstract hideable property

In order to eliminate a variable slot and have all accesses to that variable access some closure record on the static backchain it must be *hideable*.

The abstract hideable property is defined as follows:

Definition 32. A variable x is *hideable*, denoted $\textsc{Hideable}(x)$, if

a. the abstract location of x contains a single abstract procedure p,

b. for all nontrivial accesses e to x, $p(e)$ is nested in every ancestor of p, and

c. p must alias.

STALIN approximates the property $\textsc{Hideable}(x)$ as follows:

$$\overline{\textsc{Hideable}}(x) \triangleq$$
$$(\exists p \in P) \left[\begin{array}{l} \beta(x) = \{p\} \wedge \overline{\textsc{MustAlias}}(p) \wedge \\ (\forall e \in A, p' \in P) \\ \left(\begin{array}{l} x(e) = x \wedge \overline{\textsc{Ancestor}}(p', p) \wedge \\ \overline{\textsc{NonTrivialReference}}(e) \end{array} \right) \rightarrow p(e) \prec^* p' \end{array} \right]$$

The hideable property obeys the following conservative approximation lemma:

LEMMA 26. *For all variables x, if $\neg\textsc{Hideable}(x)$, then $\neg\overline{\textsc{Hideable}}(x)$.*

3.19 Lightweight closure conversion

It is now possible to describe the lightweight closure-conversion process. Recall that the output of lightweight closure conversion consists of the following annotations:

$\textsc{Local}(x)$	x will be allocated as a local variable.
$\textsc{Global}(x)$	x will be allocated as a global variable.
$\textsc{Hidden}(x)$	x will be allocated as a hidden closure slot.
$\textsc{Slotted}(x)$	x will be allocated as a closure slot.
$\textsc{HasClosure}(p)$	p will have a closure and closure pointer.
$\textsc{HasParentSlot}(p)$	p will have a parent slot.
$\textsc{ParentSlot}(p)$	The parent slot for p will point to the closure for $\textsc{ParentSlot}(p)$.
$\textsc{HasParentParameter}(p)$	p will have a parent parameter and closure-pointer slot.
$\textsc{ParentParameter}(p)$	The parent parameter and closure-pointer slot for p will point to the closure for $\textsc{ParentParameter}(p)$.

Unaccessed and fictitious variables are eliminated. The ones that remain are made either local, global, hidden, or slotted. Localizable variables can be made local. Globalizable variables can be made global. Hideable variables can be made hidden. Variables that are neither localizable, globalizable, nor hideable must be made slotted. It is possible for more than one of $\overline{\textsc{Localizable}}(x)$, $\overline{\textsc{Globalizable}}(x)$, and $\overline{\textsc{Hideable}}(x)$ to be true of a given variable x. In this situation, making a variable local is given preference to making a variable global since access to local variables is faster, especially since current C compilers do intraprocedural optimization but not interprocedural optimization.

$$\textsc{Local}(x) \triangleq \overline{\textsc{Accessed}}(x) \wedge \neg\overline{\textsc{Fictitious}}(\beta(x)) \wedge \overline{\textsc{Localizable}}(x)$$
$$\textsc{Global}(x) \triangleq \left(\begin{array}{l} \overline{\textsc{Accessed}}(x) \wedge \neg\overline{\textsc{Fictitious}}(\beta(x)) \wedge \\ \overline{\textsc{Globalizable}}(x) \wedge \neg\textsc{Local}(x) \end{array} \right)$$
$$\textsc{Hidden}(x) \triangleq \left(\begin{array}{l} \overline{\textsc{Accessed}}(x) \wedge \neg\overline{\textsc{Fictitious}}(\beta(x)) \wedge \\ \overline{\textsc{Hideable}}(x) \wedge \neg\textsc{Local}(x) \wedge \neg\textsc{Global}(x) \end{array} \right)$$

$$\textsc{Slotted}(x) \triangleq \begin{pmatrix} \overline{\textsc{Accessed}}(x) \wedge \neg\overline{\textsc{Fictitious}}(\beta(x)) \wedge \\ \neg\textsc{Local}(x) \wedge \neg\textsc{Global}(x) \wedge \neg\textsc{Hidden}(x) \end{pmatrix}$$

An abstract procedure has a closure if it binds some slotted variable.

$$\textsc{HasClosure}(p) \triangleq (\exists x \in X) p(x) = p \wedge \textsc{Slotted}(x)$$

An abstract procedure p has a parent slot if some abstract procedure p' is an ancestor of p and p is an ancestor of some abstract procedure p'' that is directly nested in p.

$$\textsc{HasParentSlot}(p) \triangleq (\exists p' \in P, p'' \prec p)\overline{\textsc{Ancestor}}(p, p'') \wedge \overline{\textsc{Ancestor}}(p', p)$$

The parent slot of an abstract procedure p is the narrowest such procedure p'.

$$\textsc{ParentSlot}(p) \triangleq \min_{\prec^*} \{p' \in P | (\exists p'' \prec p)\overline{\textsc{Ancestor}}(p, p'') \wedge \overline{\textsc{Ancestor}}(p', p)\}$$

An abstract procedure p has a parent parameter and closure-pointer slot if it has an ancestor.

$$\textsc{HasParentParameter}(p) \triangleq (\exists p' \in P)\overline{\textsc{Ancestor}}(p', p)$$

The parent parameter and closure-pointer slot of an abstract procedure p is the narrowest such ancestor.

$$\textsc{ParentParameter}(p) \triangleq \min_{\prec^*} \{p' \in P | \overline{\textsc{Ancestor}}(p', p)\}$$

The optimizations described in section 2 can now be formulated in terms of the properties and relations enumerated above.

Parent-slot, closure-pointer–slot, parent-parameter, parent-passing, and parent-spilling elimination: The parent slot and spilling for x are eliminated when $\neg\textsc{HasParentSlot}(x)$. The closure-pointer slot, parent parameter, and parent passing for x are eliminated when $\neg\textsc{HasParentParameter}(x)$.

Eliminating indirection through the closure pointer: A bound access to x can proceed via its variable parameter rather than via the closure pointer if $\neg\overline{\textsc{Assigned}}(x)$.

Eliminating fictitious variables: The slot, parameter, passing, and spilling for x are eliminated when $\overline{\textsc{Fictitious}}(\beta(x))$.

Eliminating variables that aren't accessed: The slot, parameter, passing, and spilling for x are eliminated when $\neg\overline{\textsc{Accessed}}(x)$.

Variable-slot elimination: The slot and spilling for x are eliminated when $\neg\textsc{Slotted}(x)$.

Closure elimination, closure-pointer elimination, and parent-slot compression: The closure and closure-pointer for p are eliminated when $\neg\textsc{HasClosure}(p)$. When the parent slot for p is not eliminated, it is compressed to $\textsc{ParentSlot}(p)$.

Closure-pointer–slot and parent-parameter compression: When the closure-pointer slot and parent parameter for p are not eliminated, they are compressed to $\textsc{ParentParameter}(p)$.

Globalization: A variable x is globalized when $\textsc{Global}(x)$.

Hiding: A variable x is hidden when $\textsc{Hidden}(x)$.

The soundness of lightweight closure conversion is summarized by the following theorem:

THEOREM 1. *For all variables* x*,*

a. at most one of LOCAL(x), GLOBAL(x), HIDDEN(x), *and* SLOTTED(x) *are true,*

b. if there is a nontrivial access to x*, then one of* LOCAL(x), GLOBAL(x), HIDDEN(x), *or* SLOTTED(x) *is true,*

c. if LOCAL(x)*, then* x *is localizable,*

d. if GLOBAL(x)*, then* x *is globalizable,*

e. if HIDDEN(x)*, then* x *is hideable, and*

f. if SLOTTED(x)*, then* HASCLOSURE($p(x)$).

For all nontrivial references e*, if* SLOTTED($x(e)$) *and* $p(x(e)) \neq p(e)$*, then*

g. HASPARENTPARAMETER($p(e)$) *and*

h. there exists $i \geq 0$ *such that*

i. for all $0 \leq j < i$*,*

HASPARENTSLOT(PARENTSLOT^j(PARENTPARAMETER($p(e)$))) *and*

j. $p(x(e)) =$ PARENTSLOT^i(PARENTPARAMETER($p(e)$)).

The approximations given in sections 3.12 through 3.19 are circular. To resolve this circularity, STALIN uses prioritized circumscription [McCarthy 1980]. It first finds a solution to the approximations with a set of hidden variables such that there is no solution to the approximations with a superset of that set of hidden variables. Fixing this set of hidden variables, it then finds a solution to the approximations with a set of abstract objects that are fictitious and a set of pairs of procedures p_1 and p_2 for which p_1 is not an ancestor of p_2 such that there is no solution to the approximations with a superset of those fictitious variables and those pairs of procedures p_1 and p_2 for which p_1 is not an ancestor of p_2.

4. EXPERIMENTAL RESULTS

The lightweight closure-conversion method described in this paper has been implemented as part of the STALIN compiler[13] for SCHEME. As originally implemented, lightweight closure conversion is a necessary component of STALIN. In order to evaluate the lightweight closure-conversion method while keeping all other aspects of the compilation process invariant, STALIN was modified to allow it to use two other closure-conversion methods. The first, the *baseline* method, is as described in section 2. In this method, there is no variable, parameter, slot, passing, spilling, closure, or closure-pointer elimination, no slot or parameter compression, no globalization, and no hiding. This method, however, still uses direct procedure calls and eliminates indirection through the closure pointer for free references, though not for bound accesses of unassigned variables. This is an overly conservative method. The second, the *conventional* method, is an attempt to model common practice in other non–whole-program compilers for SCHEME and similar lexically-scoped higher-order

[13]The version of the compiler used to run the experiments described in this paper, as well as all of the benchmark code, is available from `ftp://ftp.nj.nec.com/pub/qobi/stalin-0.9.tar.Z`.

languages. In this method, the only optimizations performed are those that can be driven by simple syntactic analysis of individual top-level procedures without flow analysis.

More precisely, the baseline method was implemented by taking all of the properties and relations as defined in section 3 with the following exceptions:

$$\begin{aligned}
\overline{\textsc{Reached}}(e) &\triangleq \textbf{true}\\
\overline{\textsc{Returns}}(e) &\triangleq \textbf{true}\\
\overline{\textsc{Fictitious}}(\beta) &\triangleq \textbf{false}\\
\overline{\textsc{Accessed}}(x) &\triangleq \textbf{true}\\
\overline{\textsc{Assigned}}(x) &\triangleq \textbf{true}\\
\overline{\textsc{Localizable}}(x) &\triangleq \textbf{false}\\
\overline{\textsc{Globalizable}}(x) &\triangleq \textbf{false}\\
\overline{\textsc{Hideable}}(x) &\triangleq \textbf{false}
\end{aligned}$$

And the conventional method was implemented by taking all of the properties and relations as defined in section 3 with the following exceptions:

$$\begin{aligned}
\overline{\textsc{Reached}}(e) &\triangleq \textbf{true}\\
\overline{\textsc{Returns}}(e) &\triangleq \textbf{true}\\
\overline{\textsc{Fictitious}}(\beta) &\triangleq \textbf{false}\\
\overline{\textsc{Accessed}}(x) &\triangleq x \text{ is defined at the top level} \vee (\exists e \in A) x(e) = x\\
\overline{\textsc{Assigned}}(x) &\triangleq x \text{ is defined at the top level} \vee (\exists e \in S) x(e) = x\\
\overline{\textsc{Localizable}}(x) &\triangleq \neg(\exists e \in A \cup S) \left(\begin{array}{l} x(e) = x \wedge p(e) \neq p(x(e)) \wedge \\ \overline{\textsc{NonTrivialReference}}(e) \end{array} \right)\\
\overline{\textsc{Globalizable}}(x) &\triangleq \left(\begin{array}{l} x \text{ is bound by a } \texttt{let} \text{ expression} \wedge \\ \text{every surrounding expression is also a } \texttt{let} \end{array} \right)\\
\overline{\textsc{Hideable}}(x) &\triangleq \textbf{false}
\end{aligned}$$

The above definitions for $\overline{\textsc{Reached}}$, $\overline{\textsc{Accessed}}$, and $\overline{\textsc{Assigned}}$ were used only during closure conversion. The full definition of these properties, along with the full whole-program interprocedural flow analysis, was used for all other analyses, optimization, and code generation performed by the compiler.

The code generator needed some modification to support the baseline and conventional methods. As originally implemented, the code generator never generated slots, elements, parameters, locals, or globals for fictitious locations. Because such locations were always eliminated under the lightweight method. With the baseline and conventional methods, however, fictitious locations are not eliminated. Thus the code generator was modified so that dummy C `int`s were used wherever a fictitious object was needed.

Twenty SCHEME benchmarks[14] were compiled using each of the baseline, conventional, and lightweight methods. All runs were done on one processor of an unloaded Dell PowerEdge 6350 with four 450MHz Pentium II Xeon processors with 512K L2 cache and 1G main memory running Red Hat 6.0 with kernel 2.2.10/SMP and EGCS 2.91.66. Tables II, III, and IV give the variable-, procedure-, and reference-elimination statistics for these runs respectively. All entries are static counts except for the last column in table IV which gives run time in CPU seconds. The first, second, and third lines for each benchmark show the results for the baseline, conventional, and lightweight methods respectively. Table II shows the total number of variables, the number of variables for which $\overline{\text{ACCESSED}}(x)$ or $\overline{\text{ASSIGNED}}(x)$ is false or $\overline{\text{FICTITIOUS}}(\beta(x))$ is true, the number of variables that are eliminated, and the number of variables for which $\text{LOCAL}(x)$, $\text{GLOBAL}(x)$, $\text{HIDDEN}(x)$, and $\text{SLOTTED}(x)$ are true. Table III shows the total number of procedures, the number of procedures for which $\text{HASCLOSURE}(p)$, $\text{HASPARENTSLOT}(p)$, and $\text{HASPARENTPARAMETER}(p)$ are true, and the amount of parent parameter and slot compression, the number of procedures for which $\text{HASPARENTSLOT}(p)$ and $\text{HASPARENTPARAMETER}(p)$ is true but for which $\text{PARENTSLOT}(p)$ and $\text{PARENTPARAMETER}(p)$ do not equal $p(p)$ respectively. Table IV shows the average number of indirections (i.e. C `->` operators) per reference, the total number of accesses, the number of nontrivial accesses, the total number of assignments, the number of nontrivial assignments, and the `user+system` CPU time for the benchmarks as reported by the `time` command. Note that the variable, procedure, and reference counts include variables, procedures, and references in the standard prelude as well as those that are introduced by macro expanding the source program, except where such variables, procedures, and references are eliminated by a simple syntax-directed dead-code–elimination prepass.

These results show some important trends. First, flow-directed reachability analysis allows a substantial increase in the number of variables that are determined to be unassigned or unaccessed. Second, in all of the benchmarks, the lightweight method eliminates many more variables, accesses, and assignments and substantially reduces the number of slotted variables, the number of procedures that require closures, parent parameters, and parent slots, and the average number of indirections per reference, as compared with both the baseline and conventional methods. In half of the benchmarks, the lightweight method eliminates all slotted variables and, with that, all closures, parent parameters, parent slots, and indirection during variable reference. Third, in all of the benchmarks, the code produced by the lightweight method runs substantially faster than the code produced by both the baseline and conventional methods.

Table V compares the run times for these twenty benchmarks compiled with STALIN using the lightweight method as compared with SCHEME->C, GAMBIT-C,

[14]The version of `boyer` that was used was obtained from the SCHEME repository. Andrew Wright provided the versions of `graphs` and `lattice` that were used. Saumya Debray provided the versions of `nucleic2`, `matrix`, `earley`, `scheme`, and `conform` that were used. William Clinger provided the versions of `nboyer`, `sboyer`, and `dynamic` that were used. Bruno Haible provide the version of `fannkuch` that was used. Richard O'Keefe provided the versions of `integ`, `gold`, and `sort` that were used. The `simplex`, `em-functional`, `em-imperative`, and `nfm` benchmarks were written by the author.

Table II. Variable-elimination statistics for twenty SCHEME benchmarks. The first, second, and third row for each benchmark give the statistics for the baseline, conventional, and lightweight methods respectively. All entries are static counts.

	vars	not assigned	not accessed	fictitious	eliminated	local	global	hidden	slotted
boyer	2193	0	0	0	0	0	0	0	2193
	2193	1315	393	0	393	953	240	0	607
	2193	1937	2056	2106	2109	80	4	0	0
graphs	2473	0	0	0	0	0	0	0	2473
	2473	1589	536	0	536	977	247	0	713
	2473	2209	2209	2260	2306	142	16	2	7
lattice	2258	0	0	0	0	0	0	0	2258
	2258	1380	384	0	384	992	251	0	631
	2258	1998	2043	2116	2126	117	4	1	10
nucleic2	3212	0	0	0	0	0	0	0	3212
	3212	2055	658	0	658	1286	473	0	795
	3212	2673	2450	2658	2724	406	65	2	15
matrix	2617	0	0	0	0	0	0	0	2617
	2617	1711	417	0	417	1144	274	0	782
	2617	2329	2067	2254	2261	307	10	3	36
earley	2899	0	0	0	0	0	0	0	2899
	2899	1962	556	0	556	1104	244	0	995
	2899	2579	2228	2357	2366	530	2	0	1
scheme	2928	0	0	0	0	0	0	0	2928
	2928	1951	704	0	704	950	489	0	785
	2928	2438	1248	1763	1815	1014	14	0	85
conform	2711	0	0	0	0	0	0	0	2711
	2711	1737	528	0	528	1108	309	0	766
	2711	2324	2051	2290	2297	388	12	2	12
nboyer	2380	0	0	0	0	0	0	0	2380
	2380	1480	432	0	432	1004	287	0	657
	2380	2087	2041	2149	2153	219	8	0	0
sboyer	2384	0	0	0	0	0	0	0	2384
	2384	1483	433	0	433	1006	288	0	657
	2384	2090	2042	2151	2155	221	8	0	0
dynamic	7822	0	0	0	0	0	0	0	7822
	7822	6513	905	0	905	4510	460	0	1947
	7822	7071	2169	3932	3978	3698	59	0	87
fannkuch	2158	0	0	0	0	0	0	0	2158
	2158	1287	388	0	388	934	234	0	602
	2158	1907	2072	2098	2103	53	2	0	0
simplex	2240	0	0	0	0	0	0	0	2240
	2240	1346	441	0	441	938	239	0	622
	2240	1970	2145	2174	2180	50	10	0	0
em-functional	3628	0	0	0	0	0	0	0	3628
	3628	2582	716	0	716	1368	325	0	1219
	3628	3190	2282	2666	2672	944	12	0	0
em-imperative	2807	0	0	0	0	0	0	0	2807
	2807	1820	604	0	604	1062	304	0	837
	2807	2426	2168	2358	2366	429	12	0	0
nfm	4481	0	0	0	0	0	0	0	4481
	4481	3347	733	0	733	2043	274	0	1431
	4481	3932	2150	3123	3156	1315	10	0	0
integ	2104	0	0	0	0	0	0	0	2104
	2104	1243	360	0	360	939	237	0	568
	2104	1859	2017	2045	2047	52	4	0	1
gold	2185	0	0	0	0	0	0	0	2185
	2185	1310	381	0	381	961	246	0	597
	2185	1926	2038	2077	2083	97	5	0	0
sort	2192	0	0	0	0	0	0	0	2192
	2192	1318	388	0	388	950	244	0	610
	2192	1935	2049	2110	2119	71	2	0	0
rrr	2755	0	0	0	0	0	0	0	2755
	2755	1812	547	0	547	1188	262	0	758
	2755	2426	2175	2331	2343	375	36	0	1

Table III. Procedure-elimination statistics for twenty SCHEME benchmarks. The first, second, and third row for each benchmark give the statistics for the baseline, conventional, and lightweight methods respectively. All entries are static counts.

	procedures	closures	parent slots	parent parameters	parent slot compression	parent parameter compression
boyer	2285	1132	34	110	6	31
	2285	450	22	82	2	23
	2285	0	0	0	0	0
graphs	2351	1330	109	270	35	83
	2351	536	87	240	33	77
	2351	6	4	32	2	25
lattice	2181	1195	47	143	8	43
	2181	476	30	111	2	31
	2181	4	2	14	0	9
nucleic2	2868	1427	137	462	27	64
	2868	583	68	297	25	58
	2868	7	1	30	0	23
matrix	2533	1432	136	368	30	96
	2533	582	101	315	23	80
	2533	24	15	110	8	65
earley	2640	1536	241	489	53	121
	2640	676	218	443	42	95
	2640	1	0	9	0	8
scheme	2773	1404	444	1495	163	629
	2773	573	326	1313	120	567
	2773	49	9	103	0	54
conform	2626	1414	214	576	31	155
	2626	582	156	432	22	102
	2626	5	0	40	0	25
nboyer	2466	1243	109	319	27	85
	2466	503	44	216	18	66
	2466	0	0	0	0	0
sboyer	2467	1243	109	319	27	85
	2467	503	44	216	18	66
	2467	0	0	0	0	0
dynamic	6830	4656	845	3532	112	810
	6830	1463	535	2238	86	643
	6830	87	27	94	0	5
fannkuch	2109	1133	43	105	11	30
	2109	460	35	89	10	27
	2109	0	0	0	0	0
simplex	2249	1201	67	179	26	78
	2249	490	62	167	25	76
	2249	0	0	0	0	0
em-functional	3458	2097	437	1071	59	212
	3458	920	343	948	51	196
	3458	0	0	0	0	0
em-imperative	2761	1543	273	639	75	187
	2761	662	233	585	69	177
	2761	0	0	0	0	0
nfm	4170	2548	554	1613	98	348
	4170	1077	431	1432	90	336
	4170	0	0	0	0	0
integ	2045	1092	38	100	7	24
	2045	438	26	79	7	20
	2045	1	0	1	0	0
gold	2100	1132	54	134	16	42
	2100	452	41	110	15	36
	2100	0	0	0	0	0
sort	2139	1153	43	124	7	31
	2139	466	35	110	7	30
	2139	0	0	0	0	0
rrr	2600	1452	160	432	47	149
	2600	559	126	388	42	141
	2600	1	0	1	0	0

Table IV. Reference-elimination statistics for twenty SCHEME benchmarks. The first, second, and third row for each benchmark give the statistics for the baseline, conventional, and lightweight methods respectively. All entries are static counts, except for the last column which is in CPU seconds.

	average indirections per reference	accesses	nontrivial accesses	assignments	nontrivial assignments	run time
boyer	1.147	4402	4402	483	263	149.740
	0.357	4402	4402	483	263	106.450
	0.000	4402	212	483	13	94.820
graphs	1.690	4604	4604	478	264	73.470
	1.038	4604	4604	478	264	47.560
	0.044	4604	501	478	1	15.270
lattice	1.164	4311	4311	472	260	332.790
	0.385	4311	4311	472	260	239.490
	0.026	4311	281	472	4	53.480
nucleic2	1.316	7659	7659	751	539	28.340
	0.314	7659	7659	751	539	20.570
	0.006	7659	1147	751	50	17.510
matrix	1.425	5280	5280	500	288	488.060
	0.616	5280	5280	500	288	257.770
	0.065	5280	729	500	2	100.380
earley	1.862	5246	5246	531	320	84.120
	1.384	5246	5246	531	320	42.660
	0.003	5246	1113	531	3	22.720
scheme	1.743	5823	5823	572	524	444.220
	0.742	5823	5823	572	524	257.750
	0.020	5823	3280	572	48	235.630
conform	1.336	5144	5144	573	392	527.740
	0.555	5144	5144	573	392	289.690
	0.032	5144	981	573	20	198.320
nboyer	1.257	4898	4898	514	313	35.310
	0.428	4898	4898	514	312	27.540
	0.000	4898	630	514	32	23.700
sboyer	1.257	4908	4908	515	314	33.130
	0.426	4908	4908	515	313	23.880
	0.000	4908	638	515	32	17.250
dynamic	1.322	16602	16602	945	793	57.670
	0.473	16602	16602	945	793	31.950
	0.008	16602	6555	945	106	28.430
fannkuch	1.267	3995	3995	465	251	525.030
	0.478	3995	3995	465	251	511.530
	0.000	3995	146	465	1	52.750
simplex	3.529	4289	4289	500	282	15.320
	2.233	4289	4289	500	282	11.190
	0.000	4289	334	500	23	2.210
em-functional	1.770	6505	6505	642	440	70.440
	0.933	6505	6505	642	440	30.920
	0.000	6505	2109	642	16	7.360
em-imperative	2.300	5239	5239	584	384	18.130
	1.361	5239	5239	584	384	9.400
	0.000	5239	1271	584	25	2.410
nfm	1.651	8282	8282	728	549	18.600
	0.843	8282	8282	728	549	15.500
	0.000	8282	2400	728	7	5.030
integ	1.162	3938	3938	455	245	4.390
	0.298	3938	3938	455	245	3.540
	0.001	3938	167	455	0	1.460
gold	1.448	4160	4160	469	259	16.990
	0.706	4160	4160	469	259	6.110
	0.000	4160	351	469	3	5.520
sort	1.187	4114	4114	470	258	60.650
	0.440	4114	4114	470	258	31.780
	0.000	4114	204	470	4	12.500
rrr	2.804	5319	5319	545	337	748.320
	1.201	5319	5319	545	337	493.390
	0.000	5319	1109	545	13	143.100

Bigloo, and Chez. For all but three of the benchmarks, Stalin significantly outperforms the other compilers. Note that many of the compilers were run with options that violate R4RS semantics. In particular, Scheme->C was run with `-On` for many of the benchmarks, Gambit-C was run with `standard-bindings` and `extended-bindings` for all of the benchmarks and with `fixnum` for many of the benchmarks, and Bigloo was run with `-farithmetic` for many of the benchmarks. This, in essence, manually gives these compilers information that can only be determined by whole-program interprocedural flow analysis. Information that Stalin soundly and automatically determines on its own. When Scheme->C, Gambit-C, and Bigloo are run without these options, the ratio of run times relative to Stalin are even more favourable to Stalin than the results presented in table V.

5. CONCLUSION

A number of researchers have pursued work along the lines of the work presented in this paper. Deutsch [1990], Emami and Hendren [1994], and Jagannathan et al. [1998] present aliasing analyses that are based on flow analysis. Henglein [1992], Wand and Steckler [1994], Steckler [1994b], Steckler [1994c], Steckler [1994a], Minamide et al. [1996], and Steckler and Wand [1997] present alternate approaches to lightweight closure conversion. The techniques of Steckler and Wand, in particular, are incomparable to the techniques presented in this paper. Cases can be constructed where the approach of Steckler and Wand performs optimizations that the approach presented in this paper does not perform, and vice versa.

Whole program analysis, in particular flow, reachability, points-to, and escape analysis, when used to support the lightweight closure-conversion method described in this paper, offers significant reduction in

—variable parameters and variable slots,

—parent parameters and parent slots,

—closures, closure pointers, and closure-pointer slots,

—variable parameter and parent parameter spilling,

—variable parameter and parent parameter passing, and

—indirection in variable reference.

Furthermore, it results in substantial improvements in compiled-code speed.

REFERENCES

Clinger, W. and Rees, J. 1991. *Revised*^{4} *Report on the Algorithmic Language* Scheme.

Deutsch, A. 1990. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Proceedings of the* 17$^{\text{th}}$ *Annual ACM Symposium on Principles of Programming Languages*. 157–168.

Emami, M. and Hendren, R. G. L. J. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the 1994 SIGPLAN Conference on Programming Language Design and Implementation*. 242–256.

Heintze, N. 1993. Set based analysis of ml programs. Tech. Rep. CMU CS 93-193, School of Computer Science, Carnegie-Mellon University. July.

Heintze, N. 1994. Set based analysis of ml programs. In *Proceedings of the 1994 ACM Conference on* Lisp *and Functional Programming*.

Table V. The ratio of CPU times to run the twenty benchmarks under Scheme->C (15mar93), Gambit-C (3.0), Bigloo (2.0e), and Chez (5.0a) relative to Stalin (0.9). All runs were done on one processor of an unloaded Dell PowerEdge 6350 with four 450MHz Pentium II Xeon processors with 512K L2 cache and 1G main memory running Red Hat 6.0 with kernel 2.2.10/SMP and egcs 2.91.66. All compilers were run with settings to give maximal speed at the expense of minimal safety. Scheme->C was run with `-Ob -Og -Ot -O3 -fomit-frame-pointer -freg-struct-return`. Scheme->C was given the additional option `-On` for the `boyer`, `graphs`, `lattice`, `matrix`, `earley`, `scheme`, `conform`, `nboyer`, `sboyer`, `dynamic`, `fannkuch`, and `rrr` benchmarks. Gambit-C was run with `r4rs-scheme`, `block`, `standard-bindings`, `extended-bindings`, `not safe`, `not interrupts-enabled`, and `-O3 -fomit-frame-pointer -freg-struct-return -D___SINGLE_HOST`. Gambit-C was given the additional option `fixnum` for the same benchmarks that Scheme->C was given the `-On` option. Bigloo was run with `-call/cc -unsafe -Obench -O6 -fstack -copt "-O3 -fomit-frame-pointer -freg-struct-return"`. Bigloo was given the additional option `-farithmetic` for the same benchmarks that Scheme->C was given the `-On` option. Chez was run with `optimize-level 3`. Stalin was run with `-Ob -Om -On -Or -Ot -copt -O3 -copt -fomit-frame-pointer -copt -freg-struct-return`. Stalin was given the additional options `-clone-size-limit 0 -do-not-index-allocated-structure-types-by-expression -do-not-index-constant-structure-types-by-expression -treat-all-symbols-as-external` for the `scheme` benchmark, the additional option `-do-not-index-allocated-structure-types-by-expression` for the `dynamic` benchmark, and the additional option `-d` for those benchmarks that were not given `-On` under Scheme->C. Times are relative best-of-three-successive `user+system` CPU times as reported by the `time` command. Runs where Stalin performs worse than other implementations are highlighted in bold.

	Scheme->C / Stalin	Gambit-C / Stalin	Bigloo / Stalin	Chez / Stalin
`boyer`	1.611	4.095	1.331	2.225
`graphs`	3.742	8.764	18.112	2.036
`lattice`	2.493	3.123	1.740	1.795
`nucleic2`	12.391	44.851	9.962	8.420
`matrix`	1.773	2.948	1.545	1.889
`earley`	1.106	3.668	1.454	1.844
`scheme`	**0.555**	**0.816**	**0.345**	**0.422**
`conform`	1.204	2.765	1.025	1.544
`nboyer`	3.527	4.145	5.437	3.972
`sboyer`	1.228	2.406	1.963	1.647
`dynamic`	**0.366**	1.620	**0.479**	**0.518**
`fannkuch`	1.104	1.075	**0.742**	3.677
`simplex`	3.789	30.426	5.108	4.441
`em-functional`	6.993	18.503	4.868	3.758
`em-imperative`	6.680	24.768	6.755	5.141
`nfm`	3.645	163.917	6.683	7.950
`integ`	17.828	44.953	15.485	9.931
`gold`	13.578	47.564	12.758	8.391
`sort`	3.410	12.467	2.449	1.569
`rrr`	1.289	10.804	1.020	5.010

Henglein, F. 1992. Simple closure analysis. Tech. Rep. D-193, Department of Computer Science, University of Copenhagen. Mar.

Jagannathan, S., Thiemann, P., Weeks, S. T., and Wright, A. K. 1998. Single and loving it: Must-alias analysis for higher-order languages. In *Proceedings of the 1998 Symposium on Principles of Programming Languages*.

Kelsey, R., Clinger, W., and Rees, J. 1998. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation 11,* 1 (Sept.).

McCarthy, J. 1980. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence 13,* 1–2, 27–39.

Minamide, Y., Morrisett, J. G., and Harper, R. 1996. Typed closure conversion. In *Proceedings of the 1996 Symposium on Principles of Programming Languages*. 271–283.

Shivers, O. 1988. Control flow analysis in Scheme. In *Proceedings of the 1988 SIGPLAN Conference on Programming Language Design and Implementation*. 164–174.

Shivers, O. 1990. Data-flow analysis and type recovery in Scheme. In *Topics in Advanced Language Implementation*. The MIT Press. Appears as Technical Report CMU-CS-90-115.

Shivers, O. 1991a. Control-flow analysis of higher-order languages or taming lambda. Ph.D. thesis, School of Computer Science, Carnegie-Mellon University.

Shivers, O. 1991b. The semantics of Scheme control-flow analysis. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. 190–198. Appears as Technical Report CMU-CS-91-119.

Siskind, J. M. 2000a. Flow-directed lightweight CPS conversion. In preparation.

Siskind, J. M. 2000b. Flow-directed polyvariance. In preparation.

Siskind, J. M. 2000c. Flow-directed representation selection. In preparation.

Siskind, J. M. 2000d. Flow-directed storage management. In preparation.

Steckler, P. A. 1994a. Correct higher-order program transformations. Ph.D. thesis, College of Computer Science, Northeastern University.

Steckler, P. A. 1994b. Correct separate and selective closure conversion. Tech. rep., Laboratory for the Foundations of Computer Science, Edinburgh University. Oct.

Steckler, P. A. 1994c. Tracking available values for lightweight closures. Tech. rep., College of Computer Science, Northeastern University. Mar.

Steckler, P. A. and Wand, M. 1997. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems 19,* 1 (Jan.), 48–86.

Wand, M. and Steckler, P. A. 1994. Selective and lightweight closure conversion. In *Proceedings of the* 21$^{\text{nd}}$ *Annual ACM Symposium on Principles of Programming Languages*. 435–445.

A. GLOSSARY

A.1 Syntactic terminology

source	the first subexpression of an assignment
callee	the first subexpression of a call or primcall
argument	any subexpressions of a call or primcall except the first
parameters	the variables of a lambda expression
arity	the number of arguments of a call or primcall or the number of parameters of a lambda expression
body	the subexpression of a lambda expression
antecedent	the first subexpression of a conditional
consequent	the second subexpression of a conditional
alternate	the third subexpression of a conditional
x	a variable
q	a primitive
e	an expression
p	a lambda expression, also an abstract procedure
p_0	the top-level lambda expression that contains the entire source program
X	the set of all variables in the source program
E	the set of all expressions in the source program
A	the set of all accesses in the source program
S	the set of all assignments in the source program
C	the set of all calls in the source program
C_i	the set of all calls of arity i in the source program
R	the set of all primcalls in the source program
R_i	the set of all primcalls of arity i in the source program
P	the set of all lambda expressions in the source program
$\textsc{Source}(e)$	the source subexpression of an assignment e
$x(e)$	the variable in an access or assignment e
$\textsc{Callee}(e)$	the callee subexpression of a call or primcall e
$\textsc{Arguments}(e)$	the set of argument subexpressions of a call or primcall e
$\textsc{Argument}_i(e)$	the i^{th} argument subexpression of a call or primcall e
$\textsc{Parameter}_i(e)$	the i^{th} parameter of a lambda expression e
$\textsc{Arity}(e)$	the arity of a call, primcall, or lambda expression e
$\textsc{Body}(e)$	the body subexpression of a lambda expression e
$p(x)$	the lambda expression in which x is bound
$p(e)$	the narrowest lambda expression that properly contains e
$p_1 \prec p_2$	p_1 is directly nested in p_2
$p_1 \prec^+ p_2$	p_1 is properly nested in p_2
$p_1 \prec^* p_2$	p_1 is nested in p_2
$\textsc{InTailPosition}(e)$	e is in tail position

A.2 Flow analysis terminology

k	a concrete object
l	a concrete location
σ	an abstract object
β	an abstract location
$k \in l$	l can take on k as its value
$\sigma \in \beta$	some $l \in \beta$ can take on some $k \in \sigma$ as its value in some execution of the program
$\beta(x)$	the abstract location associated with x
$\beta(e)$	the abstract location associated with the result of evaluating e
PAIR?(σ)	σ is an abstract pair
CAR(σ)	the abstract location containing the `car` slots of the concrete pairs in the abstract pair σ
CDR(σ)	the abstract location containing the `cdr` slots of the concrete pairs in the abstract pair σ
STRING?(σ)	σ is an abstract string
STRING-REF(σ)	the abstract location containing the elements of the concrete strings in the abstract string σ
VECTOR?(σ)	σ is an abstract vector
VECTOR-REF(σ)	the abstract location containing the elements of the concrete vectors in the abstract vector σ
SYMBOL?(σ)	σ is an abstract symbol
SYMBOL->STRING(σ)	the abstract location containing the print-name strings of the concrete symbols in the abstract symbol σ
CONTINUATION?(σ)	σ is an abstract continuation
$e(\sigma)$	the call to `call/cc` where the concrete continuations in the abstract continuation σ were created

A.3 Reachability analysis terminology

$\hat{e}$	an invocation of e
$\hat{x}$	an instance of x
$\textsc{Reached}(\hat{e})$	control flows to the program point just before $\hat{e}$
$\textsc{Returns}(\hat{e})$	control flows to the program point just after $\hat{e}$
$\textsc{Executed}(\hat{e})$	control flows to the intermediate program point in an assignment, call, or primcall invocation $\hat{e}$
$\textsc{Accessed}(\hat{x})$	some access invocation $\hat{e}$ to x is reached
$\textsc{Assigned}(\hat{x})$	some assignment invocation $\hat{e}$ to x is reached
successful	the call invocation $\hat{e}$ is executed and the arity of the call site equals the arity of the target procedure or continuation or the primcall invocation $\hat{e}$ is executed and the arity of the call site is allowed by the primitive
$\textsc{Reached}(e)$	some invocation $\hat{e}$ of e is reached in some execution of the program
$\textsc{Returns}(e)$	some invocation $\hat{e}$ of e returns in some execution of the program
$\textsc{Executed}(e)$	some invocation $\hat{e}$ of an assignment, call, or primcall e is executed in some execution of the program
$\textsc{Accessed}(e)$	some instance $\hat{x}$ of x is accessed in some execution of the program
$\textsc{Assigned}(e)$	some instance $\hat{x}$ of x is assigned in some execution of the program

A.4 Concrete aggregate access and assignment terminology—I

$\textsc{TypeTagAccesses}(\hat{e}, k)$	the primcall invocation $\hat{e}$ accesses the type tag slot of k
$\textsc{Eq?Accesses}(\hat{e}, k)$	the primcall invocation $\hat{e}$ accesses the identity of k
$\textsc{CarAccesses}(\hat{e}, k)$	the primcall invocation $\hat{e}$ accesses the `car` slot of the concrete pair k
$\textsc{CdrAccesses}(\hat{e}, k)$	the primcall invocation $\hat{e}$ accesses the `cdr` slot of the concrete pair k
$\textsc{StringLengthAccesses}(\hat{e}, k)$	the primcall invocation $\hat{e}$ accesses the length slot of the concrete string k
$\textsc{StringRefAccesses}(\hat{e}, k)$	the primcall invocation $\hat{e}$ accesses an element of the concrete string k
$\textsc{VectorLengthAccesses}(\hat{e}, k)$	the primcall invocation $\hat{e}$ accesses the length slot of the concrete vector k
$\textsc{VectorRefAccesses}(\hat{e}, k)$	the primcall invocation $\hat{e}$ accesses an element of the concrete vector k
$\textsc{SymbolToStringAccesses}(\hat{e}, k)$	the primcall invocation $\hat{e}$ accesses the print-name–string slot of the concrete symbol k
$\textsc{ContinuationAccesses}(\hat{e}, k)$	the call invocation $\hat{e}$ calls the concrete continuation k
$\textsc{ProcedureAccesses}(\hat{e}, k)$	the call invocation $\hat{e}$ successfully calls the concrete procedure k
$\textsc{CarAssigns}(\hat{e}, k)$	the primcall invocation $\hat{e}$ assigns the `car` slot of the concrete pair k
$\textsc{CdrAssigns}(\hat{e}, k)$	the primcall invocation $\hat{e}$ assigns the `cdr` slot of the concrete pair k
$\textsc{StringRefAssigns}(\hat{e}, k)$	the primcall invocation $\hat{e}$ assigns an element of the concrete string k
$\textsc{VectorRefAssigns}(\hat{e}, k)$	the primcall invocation $\hat{e}$ assigns an element of the concrete vector k

A.5 Concrete aggregate access and assignment terminology—II

TypeTagAccessed(k)	the type-tag slot of k is accessed
Eq?Accessed(k)	the identity of k is accessed
CarAccessed(k)	the `car` slot of the concrete pair k is accessed
CdrAccessed(k)	the `cdr` slot of the concrete pair k is accessed
StringLengthAccessed(k)	the length slot of the concrete string k is accessed
StringRefAccessed(k)	an element of the concrete string k is accessed
VectorLengthAccessed(k)	the length slot of the concrete vector k is accessed
VectorRefAccessed(k)	an element of the concrete vector k is accessed
SymbolToStringAccessed(k)	the print-name–string slot of the concrete symbol k is accessed
ContinuationAccessed(k)	the concrete continuation k is called
ProcedureAccessed(k)	the concrete procedure k is called
CarAssigned(k)	the `car` slot of the concrete pair k is assigned
CdrAssigned(k)	the `cdr` slot of the concrete pair k is assigned
StringRefAssigned(k)	an element of the concrete string k is assigned
VectorRefAssigned(k)	an element of the concrete vector k is assigned

A.6 Abstract aggregate access and assignment terminology—I

$\textsc{TypeTagAccesses}(e, \sigma)$	some invocation $\hat{e}$ of the primcall e accesses the type-tag slot of some $k \in \sigma$
$\textsc{Eq?Accesses}(e, \sigma)$	some invocation $\hat{e}$ of the primcall e accesses the identity of some $k \in \sigma$
$\textsc{CarAccesses}(e, \sigma)$	some invocation $\hat{e}$ of the primcall e accesses the `car` slot of some concrete pair $k \in \sigma$
$\textsc{CdrAccesses}(e, \sigma)$	some invocation $\hat{e}$ of the primcall e accesses the `cdr` slot of some concrete pair $k \in \sigma$
$\textsc{StringLengthAccesses}(e, \sigma)$	some invocation $\hat{e}$ of the primcall e accesses the length slot of some concrete string $k \in \sigma$
$\textsc{StringRefAccesses}(e, \sigma)$	some invocation $\hat{e}$ of the primcall e accesses an element of some concrete string $k \in \sigma$
$\textsc{VectorLengthAccesses}(e, \sigma)$	some invocation $\hat{e}$ of the primcall e accesses the length slot of some concrete vector $k \in \sigma$
$\textsc{VectorRefAccesses}(e, \sigma)$	some invocation $\hat{e}$ of the primcall e accesses an element of some concrete vector $k \in \sigma$
$\textsc{SymbolToStringAccesses}(e, \sigma)$	some invocation $\hat{e}$ of the primcall e accesses the print-name–string slot of some concrete symbol $k \in \sigma$
$\textsc{ContinuationAccesses}(e, \sigma)$	some invocation $\hat{e}$ of the call e calls some concrete continuation $k \in \sigma$
$\textsc{ProcedureAccesses}(e, \sigma)$	some invocation $\hat{e}$ of the call e successfully calls some concrete procedure $k \in \sigma$
$\textsc{CarAssigns}(e, \sigma)$	some invocation $\hat{e}$ of the primcall e assigns the `car` slot of some concrete pair $k \in \sigma$
$\textsc{CdrAssigns}(e, \sigma)$	some invocation $\hat{e}$ of the primcall e assigns the `cdr` slot of some concrete pair $k \in \sigma$
$\textsc{StringRefAssigns}(e, \sigma)$	some invocation $\hat{e}$ of the primcall e assigns an element of some concrete string $k \in \sigma$
$\textsc{VectorRefAssigns}(e, \sigma)$	some invocation $\hat{e}$ of the primcall e assigns an element of some concrete vector $k \in \sigma$

A.7 Abstract aggregate access and assignment terminology—II

TypeTagAccessed(σ)	the type-tag slot of some $k \in \sigma$ is accessed
Eq?Accessed(σ)	the identity of some $k \in \sigma$ is accessed
CarAccessed(σ)	the `car` slot of some concrete pair $k \in \sigma$ is accessed
CdrAccessed(σ)	the `cdr` slot of some concrete pair $k \in \sigma$ is accessed
StringLengthAccessed(σ)	the length slot of some concrete string $k \in \sigma$ is accessed
StringRefAccessed(σ)	an element of some concrete string $k \in \sigma$ is accessed
VectorLengthAccessed(σ)	the length slot of some concrete vector $k \in \sigma$ is accessed
VectorRefAccessed(σ)	an element of some concrete vector $k \in \sigma$ is accessed
SymbolToStringAccessed(σ)	the print-name–string slot of some concrete symbol $k \in \sigma$ is accessed
ContinuationAccessed(σ)	some concrete continuation $k \in \sigma$ is called
ProcedureAccessed(σ)	some concrete procedure $k \in \sigma$ is called
CarAssigned(σ)	the `car` slot of some concrete pair $k \in \sigma$ is assigned
CdrAssigned(σ)	the `cdr` slot of some concrete pair $k \in \sigma$ is assigned
StringRefAssigned(σ)	an element of some concrete string $k \in \sigma$ is assigned
VectorRefAssigned(σ)	an element of some concrete vector $k \in \sigma$ is assigned

A.8 Concrete call-graph terminology—I

$\textsc{Called}(k)$	the concrete procedure k is called
$\hat{e} \rhd k$	the call invocation $\hat{e}$ directly calls the concrete procedure k
$\hat{e} \rhd_t k$	the call invocation $\hat{e}$ directly tail calls the concrete procedure k
$\hat{e} \rhd_{\overline{t}} k$	the call invocation $\hat{e}$ directly non-tail calls the concrete procedure k
$\hat{e} \rhd_{st} k$	the call invocation $\hat{e}$ directly self-tail calls the concrete procedure k
$\hat{e} \rhd_{\overline{st}} k$	the call invocation $\hat{e}$ directly non-self-tail calls the concrete procedure k
$k_1 \rhd k_2$	the concrete procedure k_1 directly calls the concrete procedure k_2
$k_1 \rhd_t k_2$	the concrete procedure k_1 directly tail calls the concrete procedure k_2
$k_1 \rhd_{\overline{t}} k_2$	the concrete procedure k_1 directly non-tail calls the concrete procedure k_2
$k_1 \rhd_{st} k_2$	the concrete procedure k_1 directly self-tail calls the concrete procedure k_2
$k_1 \rhd_{\overline{st}} k_2$	the concrete procedure k_1 directly non-self-tail calls the concrete procedure k_2

A.9 Concrete call-graph terminology—II

$\hat{e} \Rrightarrow k$	the call invocation $\hat{e}$ properly calls the concrete procedure k
$\hat{e} \Rrightarrow_{t} k$	the call invocation $\hat{e}$ properly tail calls the concrete procedure k
$\hat{e} \Rrightarrow_{\overline{t}} k$	the call invocation $\hat{e}$ properly non-tail calls the concrete procedure k
$\hat{e} \Rrightarrow_{st} k$	the call invocation $\hat{e}$ properly self-tail calls the concrete procedure k
$\hat{e} \Rrightarrow_{\overline{st}} k$	the call invocation $\hat{e}$ properly non-self-tail calls the concrete procedure k
$k_1 \Rrightarrow k_2$	the concrete procedure k_1 properly calls the concrete procedure k_2
$k_1 \Rrightarrow_{t} k_2$	the concrete procedure k_1 properly tail calls the concrete procedure k_2
$k_1 \Rrightarrow_{\overline{t}} k_2$	the concrete procedure k_1 properly non-tail calls the concrete procedure k_2
$k_1 \Rrightarrow_{st} k_2$	the concrete procedure k_1 properly self-tail calls the concrete procedure k_2
$k_1 \Rrightarrow_{\overline{st}} k_2$	the concrete procedure k_1 properly non-self-tail calls the concrete procedure k_2
$k_1 \Rrightarrow k_2$	the concrete procedure k_1 calls the concrete procedure k_2
$k_1 \Rrightarrow_{t} k_2$	the concrete procedure k_1 tail calls the concrete procedure k_2
$k_1 \Rrightarrow_{\overline{t}} k_2$	the concrete procedure k_1 non-tail calls the concrete procedure k_2
$k_1 \Rrightarrow_{st} k_2$	the concrete procedure k_1 self-tail calls the concrete procedure k_2
$k_1 \Rrightarrow_{\overline{st}} k_2$	the concrete procedure k_1 non-self-tail calls the concrete procedure k_2

A.10 Abstract call-graph terminology—I

$\textsc{Called}(p)$	some concrete procedure $k \in p$ is called
$e \rhd p$	some invocation e of the call e directly calls some concrete procedure $k \in p$
$e \rhd_t p$	some invocation e of the call e directly tail calls some concrete procedure $k \in p$
$e \rhd_{\overline{t}} p$	some invocation e of the call e directly non-tail calls some concrete procedure $k \in p$
$e \rhd_{st} p$	some invocation e of the call e directly self-tail calls some concrete procedure $k \in p$
$e \rhd_{\overline{st}} p$	some invocation e of the call e directly non-self-tail calls some concrete procedure $k \in p$
$p_1 \rhd p_2$	some concrete procedure $k_1 \in p_1$ directly calls some concrete procedure $k_2 \in p_2$
$p_1 \rhd_t p_2$	some concrete procedure $k_1 \in p_1$ directly tail calls some concrete procedure $k_2 \in p_2$
$p_1 \rhd_{\overline{t}} p_2$	some concrete procedure $k_1 \in p_1$ directly non-tail calls some concrete procedure $k_2 \in p_2$
$p_1 \rhd_{st} p_2$	some concrete procedure $k_1 \in p_1$ directly self-tail calls some concrete procedure $k_2 \in p_2$
$p_1 \rhd_{\overline{st}} p_2$	some concrete procedure $k_1 \in p_1$ directly non-self-tail calls some concrete procedure $k_2 \in p_2$

A.11 Abstract call-graph terminology—II

$e \Rrightarrow p$	some invocation e of the call e properly calls some concrete procedure $k \in p$
$e \Rrightarrow_t p$	some invocation e of the call e properly tail calls some concrete procedure $k \in p$
$e \Rrightarrow_{\overline{t}} p$	some invocation e of the call e properly non-tail calls some concrete procedure $k \in p$
$e \Rrightarrow_{st} p$	some invocation e of the call e properly self-tail calls some concrete procedure $k \in p$
$e \Rrightarrow_{\overline{st}} p$	some invocation e of the call e properly non-self-tail calls some concrete procedure $k \in p$
$p_1 \Rrightarrow p_2$	some concrete procedure $k_1 \in p_1$ properly calls some concrete procedure $k_2 \in p_2$
$p_1 \Rrightarrow_t p_2$	some concrete procedure $k_1 \in p_1$ properly tail calls some concrete procedure $k_2 \in p_2$
$p_1 \Rrightarrow_{\overline{t}} p_2$	some concrete procedure $k_1 \in p_1$ properly non-tail calls some concrete procedure $k_2 \in p_2$
$p_1 \Rrightarrow_{st} p_2$	some concrete procedure $k_1 \in p_1$ properly self-tail calls some concrete procedure $k_2 \in p_2$
$p_1 \Rrightarrow_{\overline{st}} p_2$	some concrete procedure $k_1 \in p_1$ properly non-self-tail calls some concrete procedure $k_2 \in p_2$
$p_1 \Rrightarrow p_2$	some concrete procedure $k_1 \in p_1$ calls some concrete procedure $k_2 \in p_2$
$p_1 \Rrightarrow_t p_2$	some concrete procedure $k_1 \in p_1$ tail calls some concrete procedure $k_2 \in p_2$
$p_1 \Rrightarrow_{\overline{t}} p_2$	some concrete procedure $k_1 \in p_1$ non-tail calls some concrete procedure $k_2 \in p_2$
$p_1 \Rrightarrow_{st} p_2$	some concrete procedure $k_1 \in p_1$ self-tail calls some concrete procedure $k_2 \in p_2$
$p_1 \Rrightarrow_{\overline{st}} p_2$	some concrete procedure $k_1 \in p_1$ non-self-tail calls some concrete procedure $k_2 \in p_2$

A.12 Called-more-than-once, points-to, escape, in-lining, and reentrancy terminology

$\textsc{CalledMoreThanOnce}(k)$	the concrete procedure k is called more than once
$\textsc{CalledMoreThanOnce}(p)$	some concrete procedure $k \in p$ is called more than once
$\textsc{FreeIn}(x, p)$	x is free in p
$k \rightsquigarrow l$	k directly points to l
$l \rightsquigarrow k$	l directly points to k
$k_1 \rightsquigarrow\!\!\!\!\!\rightsquigarrow k_2$	k_1 points to k_2
$k \rightsquigarrow\!\!\!\!\!\rightsquigarrow l$	k points to l
$l \rightsquigarrow\!\!\!\!\!\rightsquigarrow k$	l points to k
$l_1 \rightsquigarrow\!\!\!\!\!\rightsquigarrow l_2$	l_1 points to l_2
$\sigma \rightsquigarrow \beta$	some $k \in \sigma$ directly points to some $l \in \beta$
$\beta \rightsquigarrow \sigma$	some $l \in \beta$ directly points to some $k \in \sigma$
$\sigma_1 \rightsquigarrow\!\!\!\!\!\rightsquigarrow \sigma_2$	some $k_1 \in \sigma_1$ points to some $k_2 \in \sigma_2$
$\sigma \rightsquigarrow\!\!\!\!\!\rightsquigarrow \beta$	some $k \in \sigma$ points to some $l \in \beta$
$\beta \rightsquigarrow\!\!\!\!\!\rightsquigarrow \sigma$	some $l \in \beta$ points to some $k \in \sigma$
$\beta_1 \rightsquigarrow\!\!\!\!\!\rightsquigarrow \beta_2$	some $l_1 \in \beta_1$ points to some $l_2 \in \beta_2$
$k \Uparrow \hat{e}$	k escapes $\hat{e}$
$\sigma \Uparrow e$	some $k \in \sigma$ escapes some invocation $\hat{e}$ of e in some execution of the program
$\sigma \Uparrow p$	σ escapes $\textsc{Body}(p)$
$\overline{\textsc{UniqueCallSite}}(e, p)$	the call e is the unique call site of p
$p_1 \hookrightarrow p_2$	p_1 is directly in-lined in p_2
$p_1 \hookrightarrow^* p_2$	p_1 is in-lined in p_2
$\textsc{Reentrant}(k)$	the concrete procedure k is reentrant
$\textsc{Reentrant}(p)$	some concrete procedure $k \in p$ is reentrant in some execution of the program

A.13 Internal lightweight closure-conversion terminology

MustAlias(x)	every reached access invocation $\hat{e}$ to x accesses the instance $\hat{x}$ bound by the most recent active invocation of some concrete procedure $k \in p(x)$
MustAlias(σ)	for every successful call invocation $\hat{e}$ to some concrete continuation $k \in \sigma$, k was created by the most recent active invocation of $e(\sigma)$
MustAlias(p)	either p has no parent parameter or for every successful call invocation $\hat{e}$ to some concrete procedure $k \in p$, the most active invocation of ParentParameter(p) when k is called is the same as the most active invocation of ParentParameter(p) when k was created
Fictitious(l)	l always contains the same concrete object
Fictitious(β)	every $l \in \beta$ is fictitious in every execution of the program
NonTrivialReference($\hat{e}$)	an access invocation e is reached and returns an object that is not known at compile time or an assignment invocation e is executed, $x(e)$ is not hidden, and there is a subsequent nontrivial access to $\hat{x}(\hat{e})$
NonTrivialReference(e)	some invocation $\hat{e}$ of the reference e is nontrivial in some execution of the program
Localizable(x)	x must alias and all nontrivial references to x are in-lined in the procedure that binds x
Globalizable(x)	x can have at most one live instance
Ancestor(p_1, p_2)	some reached expression in p_2 accesses a pointer to the closure for p_1
Hideable(x)	the abstract location of x contains a single abstract procedure p, for all nontrivial accesses e to x, $p(e)$ is nested in every ancestor of p, and p must alias

A.14 Output lightweight closure-conversion terminology

LOCAL(x)	x will be allocated as a local variable
GLOBAL(x)	x will be allocated as a global variable
HIDDEN(x)	x will be allocated as a hidden closure slot
SLOTTED(x)	x will be allocated as a closure slot
HASCLOSURE(p)	p will have a closure and closure pointer
HASPARENTSLOT(p)	p will have a parent slot
PARENTSLOT(p)	The parent slot for p will point to the closure for PARENTSLOT(p)
HASPARENTPARAMETER(p)	p will have a parent parameter and closure-pointer slot
PARENTPARAMETER(p)	The parent parameter and closure-pointer slot for p will point to the closure for PARENTPARAMETER(p)