Formally Based Profiling for Higher-Order Functional Languages

PATRICK M. SANSOM and SIMON L. PEYTON JONES
University of Glasgow

We present the first source-level profiler for a compiled, nonstrict, higher-order, purely functional language capable of measuring *time* as well as *space* usage. Our profiler is implemented in a production-quality optimizing compiler for Haskell and can successfully profile large applications. A unique feature of our approach is that we give a formal specification of the attribution of execution costs to cost centers. This specification enables us to discuss our design decisions in a precise framework, prove properties about the attribution of costs, and examine the effects of different program transformations on the attribution of costs. Since it is not obvious how to map this specification onto a particular implementation, we also present an implementation-oriented operational semantics, and prove it equivalent to the specification.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*; D.3.2 [**Programming Languages**]: Language Classifications—*applicative languages*; D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*; F.3.2 [**Logics and Meanings of Programs**]: Sematics of Programming Languages—*operational semantics*

General Terms: Languages, Theory

Additional Key Words and Phrases: Attribution of costs, cost centers, cost semantics, execution profiling, program transformation, space profiling, source-level profiling

1. MOTIVATION AND OVERVIEW

Everyone knows the importance of profiling tools: the best way to improve a program's performance is to concentrate on the parts or features of a program that are "eating the lion's share" of the machine resources [Bentley 1982; Ingalls 1972; Knuth 1971]. One would expect profiling tools to be particularly useful for very high-level languages where the mapping from source code to target machine is much less obvious to the programmer than it is for (say) C. Despite this obvious need, profiling tools for such languages are very rare. Why? Because profilers can only readily measure or count low-level execution events, whose relationship to the

An earlier version of this article appeared in Proceedings of the ACM Symposium on Principles of Programming Languages, 1995, under the title "Time and space profiling for non-strict, higher-order functional languages."

This work was partly supported by the Commonwealth Scholarship Commission and the Engineering and Physical Sciences Research Council (grant GR/J12994).

Authors' address: Department of Computer Science, The University of Glasgow, Glasgow G12 8QQ, U.K.; email: {sansom; simonpj}@dcs.glasgow.ac.uk.

original high-level program can be far from obvious.

With this in mind, we have developed a profiler for Haskell, a higher-order nonstrict, purely functional language [Hudak et al. 1992]. We make three main contributions:

(1) *We describe a source-level profiler for a compiled, nonstrict language capable of measuring both execution time and space usage.* Nonstrict languages are usually implemented using some form of *lazy evaluation*, whereby values are computed only when required. For example, if one function produces a list which is consumed by another, execution alternates between producer and consumer in a coroutine-like fashion. Execution of different parts of the program is therefore finely interleaved, which makes it difficult to measure how much time is spent in each "part" of the program.

Our profiler solves this problem; indeed, though the results depend on the *degree* of evaluation, they are entirely independent of the *order* in which the evaluation proceeds. This issue has been independently addressed by Clack et al. [1995]; our new contribution is to deal with a production-quality compiled implementation of a fully featured language (Haskell).

(2) *We support the profiling of large programs by allowing the systematic subsumption of execution costs under programmer-controlled headings, or "cost centers."* Most profilers implicitly attribute costs to the function or procedure which incurred them. This is unhelpful for languages like Haskell, which encourage a modular programming style based on a large number of small functions, for two reasons: first, there are simply too many functions in a large program; and second, it does not help much to be told (say) that the program spends 20% of its time in the heavily used library function `map`. Cost centers allow the programmer to choose an appropriate granularity for profiling, ranging from whole program phases to individual subexpressions in a single function.

(3) *We provide a formal specification of the attribution of execution costs to cost centers.* Higher-order languages make it harder to give the programmer a clear model of where costs are attributed. For example, suppose a function produces a data structure with functions embedded in it. Should the execution costs of one of these embedded functions be attributed to the "part" of the program where it is called or to the part which produced the data structure?

A unique contribution of this article is that we back up our informal description of cost attribution (Section 3.1) with a formal specification, or *cost semantics* (Sections 3.2-3.4). In this framework we prove properties about the cost attribution (Sections 3.5 and 4.3), examine the effects of different program transformations on the attribution of costs (Section 5), and explore the design space in a precise way (Section 8).

While our approach can handle nonstrict languages such as Haskell, it is not restricted to them: it can also accommodate strict languages such as SML, though many of the issues we address here are simplified — see Section 8.4.

From a practical point of view, our technique is relatively straightforward to implement. In Section 4 we describe a full implementation of the profiler in the Glasgow Haskell Compiler, a state-of-the-art compiler for Haskell [Peyton Jones et al. 1993].

As well as an informal description of the implementation (Section 4.1), we present a state-transition system which describes it formally and prove it equivalent to the specification (Section 4.2). In Section 5 we consider program transformation in the presence of profiling. Using the formal specification we establish conditions under which the transformations performed by the compiler do not affect the attribution of costs. This enables us successfully to profile optimized code. The run-time overheads of the profiler are discussed in Section 6. Even though these are quite significant (61%), we believe that they are not excessive in practice.

The profiler is publicly distributed with the compiler, and its design has been significantly influenced by user feedback (Section 7) — a modified cost semantics is introduced in Section 7.2.

In Section 8 we discuss some alternative approaches, including the inheritance of costs (Section 8.3) and the profiling of strict languages (Section 8.4). Finally, we discuss related work (Section 9) and draw some conclusions (Section 10).

2. AN OVERVIEW OF THE PROFILER

We begin with an overview of the profiler, as seen from the programmer's point of view.

2.1 Cost Centers

All profilers attribute execution costs to the "parts" of the program. Most profilers implicitly identify such "parts" with functions or procedures. Since Haskell is an expression-based language, we take a more flexible approach, identifying the "parts" of the program by associating a *cost center*, to which execution costs are attributed, with each expression of interest. For example, consider the following function definition:

```
f x y  = x + (scc "test" factorize y)
```

The `scc` construct explicitly annotates an expression with a cost center, to which the costs of evaluating that expression are attributed. In the example above, the costs of evaluating `factorize y` are attributed to the (arbitrarily named) cost center `"test"`. Associated with each `scc` annotation is an *entry count* which is incremented each time the `scc` is evaluated. This count is equivalent to the function-call-counts of conventional profilers.

From a syntactic point of view `scc` (short for "set-cost-center"[1]) is a language construct, like `let` or `case`, and not a function. The cost center is a literal string, not a computed value; and `scc` has lower precedence than function application, i.e., its scope extends as far to the right as possible.

The `scc` construct annotates an *expression* rather than a *function definition*. This distinction is largely cosmetic: an expression can easily be made into a function by lambda lifting. Nevertheless we have found that it is often convenient to be able to profile subexpressions of the main function of a program without lambda lifting. More importantly, the expression form gives us a convenient language in which to discuss the effect of program transformations (see Section 5).

[1] The irony of this imperative-sounding name is not lost on us.

2.2 Introducing the scc Annotations

Our profiler offers two mechanisms for introducing `scc` annotations into a program: automatically by the compiler or manually by the programmer. These two mechanisms support the two main approaches used in practice by programmers to find performance bugs: bottom-up and top-down.

The bottom-up method is used when profiling a library module, such as an abstract data type. Using a compiler flag, the programmer can annotate every top-level function definition with an eponymous cost center, without altering the source code itself. This annotation can be done for selected modules or indeed for all modules. The latter case is equivalent to the function-based profiling provided by conventional profilers.

The top-down method starts with the main function of the program, which usually contains calls to a number of other functions. Each of these calls is annotated explicitly by the programmer with a suitable `scc`. A run of the program often now reveals that one of these calls is taking a substantial fraction of the total execution time, in which case that function can be annotated in the same way as before. In this fashion the programmer can "home in" on the culprit.

The two methods are complementary. The bottom-up method answers the question "is this abstract data type too slow (regardless of where it is called from)," while the top-down method addresses the question "is this phase of the program too slow (regardless of which abstract data types it uses)."

2.3 Using the Profiler

To profile a program all modules (whether or not they contain `scc` annotations) must be compiled and linked with the `-prof` option. The program can then be run normally, except that it produces some extra output files containing the profiling information. A number of runtime flags can be used to request different profiling outputs.

Any explicit `scc` annotations in the source are ignored when a program is compiled without the `-prof` option.

Figures 1 and 2 give an example of output produced by the profiler. (They show the results of profiling the compiler itself, using the top-down approach, in which the call to each each of the main passes in the compiler was explicitly `scc`'d in the main function.)

The *cost center profile* in Figure 1 reports basic profiling information aggregated over the whole run of the program. For each cost center the profiler reports the following:

`scc`: The number of times the `scc` annotation was evaluated. This count is equivalent to the function-call counts of conventional profilers.

`%time`: The proportion of execution time consumed by evaluation of the expression annotated with the cost center.

`%alloc`: The proportion of heap allocation attributed to evaluation of the expression annotated with the cost center.

`inner`: The number of `scc` annotations evaluated within the scope of this cost center. This provides a reminder that some of the costs of evaluating the expression were attributed to an inner cost center.

```
Tue May 18 17:03 1993 Cost Center Profile
   hsc-0.13  +RTS -H25M -p -RTS  -C -hi ...
total time  =       240.48 secs (12024 ticks@20ms)
total alloc =  619,779,088 bytes
               51,846,277 closures
COST CENTER      scc  %time %alloc  inner
TypeChecker        1   45.4   44.6      0
Renamer            1   25.0   27.0      0
builtinNames       1    7.6   14.4      0
PrintRealC         1    4.5    4.3      0
Core2Core          1    3.9    2.5      0
MAIN               1    2.7    2.7      1
CodeGen            1    1.5    1.1      0
rdModule           1    1.8    1.2      0
Stg2Stg            1    1.1    0.5      0
FlattenAbsC        1    0.7    0.5      0
cvModule           1    0.6    0.5      1
Core2Stg           1    0.6    0.3      0
  ...
```

Fig. 1. Cost Center profile of the Glasgow Haskell compiler.

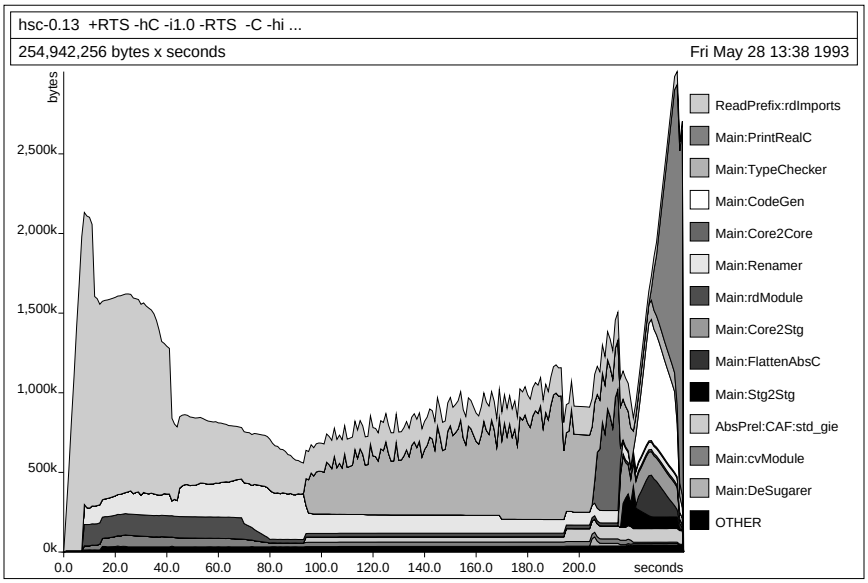


Fig. 2. Heap profile of the Glasgow Haskell compiler.

The *heap profile* in Figure 2 shows the composition of the (live) heap data, by cost center, plotted against time, in the style of Runciman and Wakeling [1993]. As with Runciman and Wakeling [1993], it is possible to break down the contents of the heap by other criteria, such as the type of the heap object or the particular data constructor. It is also possible to restrict the heap profile to a subset of interest, selecting by cost center, type, constructor, etc.

3. A SEMANTICS FOR COST ATTRIBUTION

For a profiler to be useful it must be possible to explain to a programmer the exact way in which execution costs are attributed to the headings under which the profiler reports them — that is, we must give a specification of the profiler. For a higher-order, nonstrict language such as Haskell, we have found that this specification is remarkably slippery. Every time we came up with an informal specification we found new examples for which the specification was inadequate!

This experience eventually led us to develop a formal specification of the way in which our profiler attributes costs. In this section we give an informal model of cost attribution, show how it is inadequate, and then describe our formal model.

An important constraint is that the profiler should do "what the programmer expects" in commonly occurring cases. The formal specification is useful for obscure or difficult cases, but programmers should not need to refer to it most of the time. The formal system plays exactly the same role as the formal semantics of a programming language: it is a guide to implementors and is the final arbiter of obscure cases.

3.1 Informal Cost Attribution

Given an expression `scc` *cc exp*, the general idea is that the cost of evaluating *exp* should be attributed to *cc*. However, consider the following examples:

—`scc "cc1" x+1`. In a nonstrict language `x` may not be evaluated when the evaluation of `x+1` begins: should the cost of evaluating `x` be attributed to `cc1`? Surely not, because if the evaluation order was (legitimately) changed by the compiler so that `x` was already evaluated by the time the `x+1` was started then the cost attribution would change radically.

—`scc "cc2" (f x, g x)`. In a nonstrict language, the second component of the pair (say) might never be evaluated. The profiler should presumably therefore only attribute the cost of the call `(g x)` to `cc2` if the second component of the pair turns out to be required.

—`scc "cc3" (\x -> f x, True)`. Should the costs of evaluating the calls to `f` — and there could be many such calls — be attributed to `cc3` or to the cost centers enclosing the places where the lambda abstraction is applied? We make the former choice; the issue is discussed further in Section 8.1.

Here, then, is an informal specification of our profiler's cost attribution, which tries to answer questions such as those above:

> Given an expression `scc` *cc exp*, the costs attributed to *cc* are the entire costs of evaluating the expression *exp* as far as the enclosing context demands it, including

(a) the cost of evaluating any functions called during the evaluation of *exp* and
(b) the cost of evaluating the bodies of any lambda abstractions in *exp* (however many times they are called),

but excluding

(c) the cost of evaluating the free variables of *exp* and
(d) the cost of evaluating any `scc`-expressions within *exp* or within any function called from *exp*.

This definition has the following consequences:

Cost Centers Scope Statically. If *cc* is the innermost cost center statically enclosing a subexpression *e*, then the costs of evaluating *e* are attributed to *cc*. This simple notion ensures that the profiling results are easy to interpret.

Costs are Aggregated. If no cost center statically encloses an expression, then its costs are dynamically attributed to the caller of the function of which that expression is part. For example, consider the example:

```
my_fun xs = scc "mapper" map square xs
square x = x * x
```

The function `square` does not have an `scc` construct, so its costs are attributed to the cost center of its caller, in this case `"mapper"`. Similarly, the cost of executing this call to the library function `map` is attributed to `"mapper"` as well.

Shared functions are dealt with correctly, of course: other calls to `square` are attributed to the cost center of their callers, not to `"mapper"`.

In short, except where explicit `scc` constructs specify otherwise, the costs of callees are automatically subsumed into the costs of the caller.

Cost Attribution is Independent of the Order of Evaluation. When `my_fun` is called, its argument `xs` may not be fully evaluated, and its further evaluation may ultimately be forced by `map` called from within `my_fun`. Nevertheless, the costs of evaluating `xs` is not attributed to `"mapper"` but rather to the cost center which encloses the producer of `xs`. Similarly, the result of `my_fun` is a list, which may be evaluated fully, partially, or not at all. The costs of whatever evaluation is performed are attributed to `"mapper"`, no more and no less.

In effect, the programmer does not need to understand the program's evaluation order to reason about which costs are attributed to which cost center. This property is formalized in Section 3.5.

The Degree of Evaluation Performed is Unaffected. The result of `my_fun` is evaluated no more and no less than would be the case in an unprofiled program. It follows that the costs attributed to `"mapper"` depend on how much of `my_fun`'s result is evaluated by its caller. The results are independent of the *order* of evaluation; they are *not* independent of the *degree* of evaluation!

Costs are Attributed to Precisely One Cost Center. Thus, the sum of all the time costs attributed to all the cost centers of a program is equal to the total runtime of the program; no time is lost, nor double-counted. An alternative approach is to arrange for costs to be inherited by each cost center in the stack of enclosing cost centers. This is discussed in Section 8.3.

Values:	z	::=	$\lambda x.e$	
		|	$C\ x_1 \cdots x_a$	$a \geq 0$
Expressions:	e	::=	z	
		|	$e\ x$	
		|	x	
		|	`let` x_1`=`$e_1, \ldots, x_n$`=`e_n `in` e	$n>0$
		|	`case` e `of` $\{C_j\ x_{j1} \cdots x_{ja_j}$ `->` $e_j\}_{j=1}^{n}$	$n>0$
		|	`scc` $cc\ e$	
Program:	$prog$	::=	x_1`=`$e_1, \ldots, x_n$`=`e_n	$n>0$

Fig. 3. Language syntax.

Despite our attempt at precision, our definition is still vague, especially when higher-order functions are concerned. For example, what costs are attributed to `"tricky"` in the expression `scc "tricky" (f,g)`, where `f` and `g` are functions defined elsewhere? Are the costs of any calls to the first or second components of this pair attributed to `tricky` or not? What about `scc "tricky" (f x)`, where `f` is a function of two arguments? In order to be able to answer such questions precisely we developed a formal model for cost attribution which we discuss next.

3.2 Language

The language we consider is given in Figure 3. In order to make the theory tractable we restrict ourselves to a rather small language. However, we believe that it is still close enough to Haskell to be meaningful to a programmer.

The language looks at first like no more than a slightly sugared lambda calculus, with mutually recursive `let`s, saturated data constructors, `case`, and `scc`. However, it has an important distinguishing feature: *the argument of a function application is always a simple variable*. A nonatomic argument is handled by first binding it to a variable using a `let` expression. This has a direct operational interpretation: a nonatomic function argument must be constructed in the heap (the `let` expression) before the function is called (the application). This language embodies the essential features of the Core language used in the Glasgow Haskell compiler.

For notational convenience, we use the abbreviation $\{x_i = e_i\}$ for the set of bindings x_1`=`$e_1, \ldots, x_n$`=`e_n. Similarly we write $[y_i \mapsto e_i]$ for the finite mapping $[y_1 \mapsto e_1, \ldots, y_n \mapsto e_n]$ and $e[e_i/x_i]$ for the substitution $e[e_1/x_1, \ldots, e_n/x_n]$. We also abbreviate $x_1 \cdots x_a$ with $\overline{x}_a$ and drop the $_{j=1}^{n}$ in the `case` alternatives. We use $\equiv$ for syntactic identity of expressions.

3.3 The Judgment Form

Our model is based on Launchbury's operational semantics for lazy graph reduction [Launchbury 1993; Sestoft 1997], augmented with a notion of cost attribution. We express judgments about the cost of evaluating an expression thus:

$$cc, \Gamma : e \Downarrow_\theta \Delta : z, cc_z$$

This should be read "In the context of the heap Γ and cost center cc, the expression e evaluates to the value z, producing a new heap Δ and cost center cc_z; the costs of this evaluation are described by the cost attribution θ." We use the following

Table I. Abstract Costs

Cost of	Denoted
Application	A
Case expression	C
Evaluating a thunk	V
Updating a thunk	U
Allocating a heap object	H
Entering an `scc`	E

vocabulary:

—*cost center* (cc): a label to which costs are attributed.

—*cost attribution* (θ): a finite mapping of cost centers to costs. A cost attribution records the total costs attributed to each cost center. Two cost attributions are combined using (an overloaded) $+$. We also use $-$ to remove costs from an attribution.

$$\begin{aligned}(\theta_1 +_{attr} \theta_2)(cc) &= \theta_1(cc) +_{cost} \theta_2(cc)\\ (\theta_1 -_{attr} \theta_2)(cc) &= \theta_1(cc) -_{cost} \theta_2(cc)\end{aligned}$$

—*heap* ($\Gamma, \Delta, \Theta, \Omega$): an annotated mapping from variable names to expressions. We use the notation $\Gamma[x \overset{cc}{\mapsto} e]$ to extend the heap Γ with a mapping from x to the expression e annotated with cost center cc.

In general, the cost center attached to a heap binding is the cost center which enclosed that binding. It serves two purposes: to ensure correct cost attribution when a *thunk* (unevaluated heap closure) is evaluated and to enable cost center attribution when a heap census is taken (Section 4.1).

The abstract costs are denoted by the symbols in Table I. One should not think of these costs as constants. The cost semantics specifies *which* cost center is attributed with the cost of (say) performing a heap allocation. The semantics makes no attempt to specify exactly *what* costs are attributed; it just attributes "H." The implementation attributes the *actual* execution costs (of time and space) to the cost center specified by the semantics; it does not count every A, C, and so on, at all.

The only exception to this is the "cost," E, of entering an `scc`. Since an `scc` is just an annotation, there is no real cost associated with executing it. Rather, our intention is to count the number of times each `scc` annotation is evaluated, in the same way that conventional profilers count the number of times each function is called.

The cost attribution of evaluating the whole program, θ_{MAIN}, is obtained from the judgment

$$\texttt{"MAIN"}, \Gamma_{init} : \texttt{main} \Downarrow_{\theta_{MAIN}} \Delta : z, cc_z.$$

The initial cost center is `"MAIN"`, to which all costs are attributed, except where an `scc` construct specifies otherwise. The initial heap, Γ_{init}, binds each top-level identifier to its right-hand side. What cost center should be associated with these bindings? A top-level binding defines either a *function* (if it has arguments) or a *constant applicative form* (if it does not). The costs of top-level functions should be subsumed by their caller, so we give their bindings in Γ_{init} the special pseudo-

cost-center "SUB" to indicate this fact. The "SUB" cost center is treated specially by the rules which follow. We discuss the treatment of constant applicative forms in Section 7.

The semantics is a "big-step" semantics, so it does not say anything about the cost attribution of nonterminating programs. This is more of a technical problem than a practical one. In practice, we arrange that if the program is interrupted then the profiling information accumulated to that point is dumped into the profile files before execution is finally terminated.

3.4 The Rules

The cost-augmented semantics is given in Figure 4. The following paragraphs discuss the rules.

The cost center on the left-hand side of the judgment is the "current cost center," to which the costs of evaluating the expression should be attributed. A useful invariant is that the current cost center is never "SUB". The last rule, *SCC*, is easy to understand: it simply makes the specified cost center, cc_{scc}, into the current cost center. In addition it counts the evaluation of the `scc` by attributing E to cc_{scc}.

It is less obvious why we need a cost center on the right-hand side, which we call the "returned cost center." The *Lam* and *Con* rules show where it comes from: in both cases the expression to be evaluated is in head normal form, so it is returned, along with the current cost center. (The other rules simply propagate it.) What use is made of the returned cost center? To see this we must look at the two rules which "consume" head normal forms, namely *App* (where a function is evaluated before applying it) and *Case* (where a data value is evaluated before taking it apart):

—In the *Case* rule the returned cost center cc_C is simply ignored. The appropriate alternative is chosen, and its right-hand side is evaluated in the context of the original cost center enclosing the `case`. That is as one would expect: the costs of evaluating the alternatives accrue to the cost center enclosing the `case` expression.

—In the *App* rule, the function e is evaluated, delivering (presumably) a λ-abstraction $\lambda y.e'$ and a returned cost center cc_λ. The body of the abstraction, e', is then evaluated in the context of the returned cost center cc_λ. In this way the costs of evaluating the body of the λ-abstraction accrue to the cost center enclosing the *declaration* of the λ-abstraction (see Section 4.3).

Lastly, we deal with the *Let* and *Var* rules, which concern the construction and evaluation of heap-allocated thunks. The *Let* rule extends the heap with bindings for newly allocated closures. The y_i are freshly chosen names, directly modeling heap addresses and are substituted for the corresponding x_i throughout. This substitution ensures that two instantiations of the same `let` expression do not interfere with each other by binding the same variable twice.

The current cost center is pinned on each binding created by the *Let* rule. The two *Var* rules shows how this cost center is used:

—When a variable is to be evaluated, and it is already bound to a value, the *Var*(*whnf*) rule says that the value is returned, with an unchanged heap, and the returned cost center is that pinned on the binding cc_z. However, if cc_z is

$$cc, \Gamma : \lambda y.e \Downarrow_{\{\}} \Gamma : \lambda y.e, cc \qquad \textit{Lam}$$

$$cc, \Gamma : C\ \overline{x}_a \Downarrow_{\{\}} \Gamma : C\ \overline{x}_a, cc \qquad \textit{Con}$$

$$\frac{cc, \Gamma : e \Downarrow_{\theta_1} \Delta : \lambda y.e', cc_\lambda \qquad cc_\lambda, \Delta : e'[x/y] \Downarrow_{\theta_2} \Theta : z, cc_z}{cc, \Gamma : e\ x \Downarrow_{\{cc \mapsto \mathrm{A}\} + \theta_1 + \theta_2} \Theta : z, cc_z} \qquad \textit{App}$$

$$cc, \Gamma[x \overset{cc_z}{\mapsto} z] : x \Downarrow_{\{cc \mapsto \mathrm{V}\}} \Gamma[x \overset{cc_z}{\mapsto} z] : z, \mathrm{S}\binom{cc}{cc_z} \qquad \textit{Var(whnf)}$$

$$\textbf{where } \mathrm{S}\binom{cc}{\texttt{"SUB"}} = cc \qquad \mathrm{S}\binom{cc}{cc_z} = cc_z$$

$$\frac{cc_e, \Gamma : e \Downarrow_\theta \Delta : z, cc_z}{cc, \Gamma[x \overset{cc_e}{\mapsto} e] : x \Downarrow_{\{cc \mapsto \mathrm{V}\} + \theta + \{cc_z \mapsto \mathrm{U}\}} \Delta[x \overset{cc_z}{\mapsto} z] : z, cc_z}\ e \not\equiv z \qquad \textit{Var(thunk)}$$

$$\frac{cc, \Gamma[y_i \overset{cc}{\mapsto} e_i[y_i/x_i]] : e[y_i/x_i] \Downarrow_\theta \Delta : z, cc_z}{cc, \Gamma : \texttt{let}\ \{x_i\ \texttt{=}\ e_i\}\ \texttt{in}\ e \Downarrow_{\{cc \mapsto n*\mathrm{H}\} + \theta} \Delta : z, cc_z}\ y_i \textbf{ fresh} \qquad \textit{Let}$$

$$\frac{cc, \Gamma : e \Downarrow_{\theta_1} \Delta : C_k\ \overline{x}_{a_k}, cc_C \qquad cc, \Delta : e_k[x_i/y_{ki}] \Downarrow_{\theta_2} \Theta : z, cc_z}{cc, \Gamma : \texttt{case}\ e\ \texttt{of}\ \{C_j\ \overline{y_j}_{a_j}\ \texttt{->}\ e_j\} \Downarrow_{\{cc \mapsto \mathrm{C}\} + \theta_1 + \theta_2} \Theta : z, cc_z} \qquad \textit{Case}$$

$$\frac{cc_{scc}, \Gamma : e \Downarrow_\theta \Delta : z, cc_z}{cc, \Gamma : \texttt{scc}\ cc_{scc}\ e \Downarrow_{\{cc_{scc} \mapsto \mathrm{E}\} + \theta} \Delta : z, cc_z} \qquad \textit{SCC}$$

Fig. 4. Formal cost semantics.

"SUB", the current cost center is returned instead. This ensures that the invariant mentioned above is maintained. It has the effect of subsuming the costs of top-level functions into their callers, regardless of the choice made in the *App* rule.

—The *Var*(*thunk*) rule deals with the situation when the variable is bound to an as yet unevaluated expression, or thunk. In this case, the expression to which the variable is bound is evaluated, in the context of the cost center pinned on the binding cc_e. The newly calculated value is recorded in the resulting heap, replacing the previous binding for the variable; and the costs of entering the thunk and updating the heap (V and U) are recorded in the cost attribution.

It is the *Var*(*thunk*) that implements the call-by-need behavior expected of nonstrict languages. When the thunk for x is evaluated the heap is modified to bind x to its newly calculated value. Subsequent attempts to evaluate x can

therefore use this value, rather than recomputing it as call-by-name would do.

Notice that the new binding, of the variable to its value, has the returned cost center cc_z pinned on it and that the cost of the update is attributed to cc_z. This is the second way in which the returned cost center is used. Finally observe that cc_e can never be `"SUB"`, since this cost center is only ever attached to top-level function values.

The crucial point is that these rules give us *a language in which to discuss cost attribution in a precise manner*. They provide a formal framework in which alternative design choices can be examined. For example, an alternative formulation of the *App* rule might evaluate the body of the λ-abstraction in the context of cc, the cost center enclosing the application. This particular alternative is discussed in Section 8.1.

Notice that the cost attribution is a conservative extension of the rules presented by Launchbury [1993]. The shape of a proof tree, and the value produced by such a proof, never depend on the current cost center, the heap annotations, or on θ, so the new cost-attribution mechanism does not change the results produced by a computation — an easy result, but an important one.

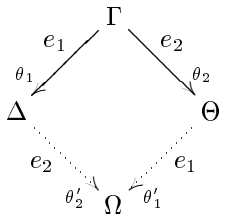
3.5 Cost Attribution and Evaluation Order

In Section 3.1 we claimed that the profiling results are *independent of the order of evaluation*. Given the formal cost semantics we can now state and prove that this is indeed the case (Theorem 3.5.1). A consequence of this is that the cost attribution is not dependent on the particular *order* of evaluation — only the *extent* of evaluation, i.e., if a closure is evaluated then it can be evaluated early without affecting the cost attribution (Corollary 3.5.2). This is important, as it means that strictness optimizations, which evaluate needed variables early, do not change the attribution of costs.

THEOREM 3.5.1 [2]

$$\begin{array}{lll} \text{If} & cc_1, \Gamma : e_1 \Downarrow_{\theta_1} \Delta : z_1, cc_{z_1} & \text{and} \quad cc_2, \Gamma : e_2 \Downarrow_{\theta_2} \Theta : z_2, cc_{z_2} \\ \text{Then} & cc_1, \Theta : e_1 \Downarrow_{\theta'_1} \Omega : z_1, cc_{z_1} & \text{and} \quad cc_2, \Delta : e_2 \Downarrow_{\theta'_2} \Omega : z_2, cc_{z_2} \\ & \text{and} \quad \theta_2 + \theta'_1 = \theta_1 + \theta'_2 & \end{array}$$

The theorem states that if two expressions, e_1 and e_2, can be evaluated in the context of a heap Γ and cost center cc, then they can be evaluated sequentially, in any order, giving the same results, final heap, and total cost attribution. That is, the following commuting diagram holds:



[2] In the statement of Theorem 3.5.1 (and Corollary 3.5.2) we have omitted details about renaming. These are addressed in Appendix B.

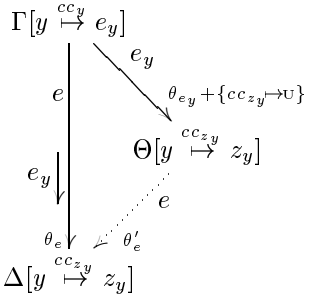
PROOF. See Appendix B. □

COROLLARY 3.5.2

If $cc, \Gamma[y \overset{cc_y}{\mapsto} e_y] : e \Downarrow_{\theta_e} \Delta[y \overset{cc_{z_y}}{\mapsto} z_y] : z, cc_z,\ e_y \not\equiv z_y$
and $cc_y, \Gamma : e_y \Downarrow_{\theta_{e_y}} \Theta : z'_y, cc'_{z_y}$

Then $z_y \equiv z'_y$ and $cc_{z_y} \equiv cc'_{z_y}$
and $cc, \Theta[y \overset{cc_{z_y}}{\mapsto} z_y] : e \Downarrow_{\theta'_e} \Delta[y \overset{cc_{z_y}}{\mapsto} z_y] : z, cc_z$
and $\theta_e = \theta_{e_y} + \{cc_{z_y} \mapsto \mathrm{U}\} + \theta'_e$

This states that if a closure $y \overset{cc_y}{\mapsto} e_y$ is evaluated during the evaluation of e and if it is possible to evaluate e_y immediately, then evaluating e in the context of the heap returned by the immediate evaluation of e_y with an updated binding for y gives the same result and total cost attribution. This is summarized by the following commuting diagram:



PROOF. Applying the *Var*(*thunk*) rule (using $\Downarrow_{\theta_{e_y}}$ as the premise) we have

$cc, \Gamma[y \overset{cc_y}{\mapsto} e_y] : y \Downarrow_{\theta_y} \Theta[y \overset{cc'_{z_y}}{\mapsto} z'_y] : z'_y, \mathrm{S}\binom{cc}{cc'_{z_y}}$
where $\theta_y = \{cc \mapsto \mathrm{V}\} + \theta_{e_y} + \{cc'_{z_y} \mapsto \mathrm{U}\}$.

Applying Theorem 3.5.1 to $\Downarrow_{\theta_e}$ and $\Downarrow_{\theta_y}$ we have

$cc, \Theta[y \overset{cc'_{z_y}}{\mapsto} z'_y] : e \Downarrow_{\theta'_e} \Omega : z, cc_z$ and $cc, \Delta[y \overset{cc_{z_y}}{\mapsto} z_y] : y \Downarrow_{\theta'_y} \Omega : z'_y, \mathrm{S}\binom{cc}{cc'_{z_y}}$
and $\theta_y + \theta'_e = \theta_e + \theta'_y$.

Since $\Downarrow_{\theta'_y}$ can be derived from the *Var*(*whnf*) rule, we have

$z_y \equiv z'_y$, $cc_{z_y} \equiv cc'_{z_y}$, $\Omega \equiv \Delta[y \overset{cc_{z_y}}{\mapsto} z_y]$ and $\theta'_y \equiv \{cc \mapsto \mathrm{V}\}$.

And finally, we have

$$\begin{aligned}\theta_e &= \theta_y + \theta'_e - \theta'_y = \{cc \mapsto \mathrm{V}\} + \theta_{e_y} + \{cc_{z_y} \mapsto \mathrm{U}\} + \theta'_e - \{cc \mapsto \mathrm{V}\} \\ &= \theta_{e_y} + \{cc_{z_y} \mapsto \mathrm{U}\} + \theta'_e. \quad \square\end{aligned}$$

Note that this proof only applies to the profiling results captured by the semantics, i.e., time and allocation. It does not guarantee that the transformation leaves the program's *residency*[3] unchanged. Indeed, changing the order of evaluation can result in significant changes in the heap requirements of a program.

[3]The residency of a program is the amount of reachable data, averaged over time; it gives a good measure of memory requirements and garbage collection cost and is only loosely related to the amount of allocation.

4. IMPLEMENTATION

We begin our description with an informal overview of the measurement technology (Section 4.1). Since there is quite a large gap between the (big-step) cost semantics of Section 3 and a concrete implementation, we present a small-step cost semantics and prove that it implements exactly the same cost attribution as the big-step semantics (Section 4.2). We also show that it implements the simple notion of "static scope" (Section 4.3). We conclude with some observations about the concrete implementation of the small-step semantics (Sections 4.4).

4.1 Implementation Overview

The profiling information presented in Section 2.3 is collected in the following way:

—At any moment the "current cost center" is stored in a known memory location `ccc`.[4] For example, when an `scc` annotation is evaluated, the annotating cost center is stored in `ccc`. A cost center is represented by a pointer to a static block of store which holds all the counters in which the costs attributed to that cost center are accumulated.

—A regular clock interrupt executes a service routine which increments a tick counter of the cost center currently stored in `ccc`. This enables the relative time of the different "parts" of the program to be determined and reported (the `%time` column in Figure 1). This statistical sampling works in much the same way as any standard Unix profiler; the more clock ticks that have elapsed, the more accurate the results. However, by attributing each sample to the current cost center (instead of the block of code currently being executed) time spent executing any library or system routines is attributed to the source code responsible for calling the routine.

—The compiler plants in-line code to increment an allocation counter held in the current cost center whenever a new heap object is allocated. In this way the profiler is able to record how much heap was allocated by each cost center (the `%alloc` column in Figure 1).

—When evaluation of (an instance of) an `scc` expression is begun, we increment the `scc` *entry count* of the *new* cost center and the `subcc` count of the *enclosing* cost center. The final counts are reported in the profile in the cost center profile (Figure 1).

—An extra field is added to the front of every heap object. This field is initialized from the current-cost-center register when the object is allocated. At regular intervals (multiples of a clock tick) a census of all live heap objects is taken. This allows the profiler to produce a *heap profile* which breaks down the contents of the heap by cost center, an example of which is given in Figure 2. Other heap censuses are also supported e.g., by constructor, type, module, etc. It is also possible to restrict a census to a particular subset of the heap e.g., profile the constructors produced by a particular cost center, or profile the cost centers which produced a particular constructor.

[4] The current cost center is stored in a memory location, rather then a machine register, so that it is available to the interrupt service routine.

Naturally, the space required for this additional field is discounted when gathering any profiling data.

—The cost center field in heap objects plays an additional role for thunks (suspensions). It records the cost center which was current when the thunk was allocated. When the thunk is subsequently evaluated, the current-cost-center register is loaded from the field in the thunk, thus neatly restoring its value at the moment the thunk was created. This is just what one would expect from the $Var(thunk)$ rule of Figure 4.

All of this is quite easily done; indeed, one of the beauties of the approach is that the runtime implementation is so straightforward. What takes a little more care is to make sure that every pass of our optimizing compiler respects cost attribution: that is, it must not move costs from one cost center to another. We return to this question in Section 5.

4.2 An Operational Semantics

There is a large gap between the formal cost semantics of Figure 4 and the implementation just described, because the cost semantics is "big-step," specifying the final result on the right-hand side of a judgment, while the implementation is necessarily "small-step," specifying intermediate steps in the computation.

Sestoft [1997] bridges this gap in his derivation of an abstract machine from Launchbury's natural semantics. We present a version of Sestoft's small-step state transition semantics, augmented with the manipulation of cost centers and a notion of cost attribution. This enables us to highlight a number of implementation-related design decisions, which are applicable to a number of different abstract machines (see Section 4.4).

The state transition rules, which include the manipulation of cost centers, are given in Figure 5. The state consists of a 5-tuple $(ccc, \Gamma, e, S, \theta)$ where ccc is the current cost center; Γ is the annotated heap; e is the expression currently being evaluated or the *control*; S is the stack (initially empty); and θ is the cost attribution of the evaluation to date.

The state transition rules correspond directly to rules in the abstract cost semantics. There is one state transition rule which initiates the evaluation associated with each subproof of each semantic rule. In addition the $Var(thunk)$ rule gives rise to a second state transition rule which updates the heap with the computed result. The correspondence between the abstract semantic rules and the state transition rules is summarized in Table II.

The key new component of the state transition rules is the stack. This is used to record any information which is required once the evaluation of a subproof has completed. There are three places where this is necessary:

—In the App rule the argument x is pushed onto the stack while the function expression e is evaluated (rule app_1). When the λ-abstraction is evaluated it retrieves this argument off the stack and evaluates its body (rule app_2).

—In the $Var(thunk)$ rule an update marker $\#x$ is pushed onto the stack while the thunk e is evaluated (rule var_2). When evaluation is complete the result value z (a λ-abstraction or constructor) encounters the update marker on the stack and updates the heap (rule var_3).

ccc	Heap	Control	Stack	θ	$\Longrightarrow$	ccc	Heap	Control	Stack	θ	Rule
cc	Γ	$e\ x$	S	θ	$\Longrightarrow$	cc	Γ	e	$x{:}S$	$\theta{+}\{cc \mapsto \text{A}\}$	app_1
cc_λ	Γ	$\lambda y.e$	$x{:}S$	θ	$\Longrightarrow$	cc_λ	Γ	$e[x/y]$	S	θ	app_2
cc	$\Gamma[x \overset{cc_z}{\mapsto} z]$	x	S	θ	$\Longrightarrow$	$\text{S}(^{cc}_{cc_z})$	$\Gamma[x \overset{cc_z}{\mapsto} z]$	z	S	$\theta{+}\{cc \mapsto \text{V}\}$	var_1
cc	$\Gamma[x \overset{cc_e}{\mapsto} e]$	x	S	θ	$\Longrightarrow$	cc_e	Γ	e	$\#x{:}S$	$\theta{+}\{cc \mapsto \text{V}\}$	var_2
cc_z	Γ	z	$\#x{:}S$	θ	$\Longrightarrow$	cc_z	$\Gamma[x \overset{cc_z}{\mapsto} z]$	z	S	$\theta{+}\{cc_z \mapsto \text{U}\}$	var_3
cc	Γ	$\texttt{let}\ \{x_i\texttt{=}e_i\}\ \texttt{in}\ e$	S	θ	$\Longrightarrow$	cc	$\Gamma[y_i \overset{cc}{\mapsto} \widehat{e_i}]$	$\widehat{e}$	S	$\theta{+}\{cc \mapsto n{*}\text{H}\}$	let
cc	Γ	$\texttt{case}\ e\ \texttt{of}\ alts$	S	θ	$\Longrightarrow$	cc	Γ	e	$(alts,cc){:}S$	$\theta{+}\{cc \mapsto \text{C}\}$	$case_1$
cc_C	Γ	$C_k\ \overline{x}_{a_k}$	$(alts,cc){:}S$	θ	$\Longrightarrow$	cc	Γ	$e_k[x_i/y_{ki}]$	S	θ	$case_2$
cc	Γ	$\texttt{scc}\ cc_{scc}\ e$	S	θ	$\Longrightarrow$	cc_{scc}	Γ	e	S	$\theta{+}\{cc_{scc} \mapsto \text{E}\}$	scc

In the var_1 rule $\text{S}(^{cc}_{cc_z})$ is as defined in Figure 4.
In the var_2 rule $e \not\equiv z$.
In the *let* rule the introduced variables, y_i, must be distinct and fresh. The notation $\widehat{e}$ means $e[y_i/x_i]$.
In the *case* rules *alts* stands for the list $\{C_j\ \overline{y_j}_{a_j}\ \texttt{->}\ e_j\}$ of alternatives.
In the $case_2$ rule e_k is the right-hand side of the kth alternative.

Fig. 5. State transition rules for lexical scoping.

Table II. Correspondence between Abstract and State Transition Semantics

Abstract Rule	State Transition Rule(s)
Lam	n.a.
App	app_1, app_2
$Var(whnf)$	var_1
$Var(thunk)$	var_2, var_3
Let	*let*
Con	n.a.
Case	$case_1$, $case_2$
SCC	*scc*

—In the *Case* rule the alternatives *alts* are saved on the stack while the case expression e is evaluated (rule $case_1$). When the evaluation of the constructor is complete the appropriate alternative is selected and evaluated (rule $case_2$). We also save the current cost center on the stack with the alternative, since it has to be restored when the alternative is evaluated.

The next step is to prove that the small-step semantics of Figure 5 is correct with respect to our earlier big-step cost semantics (Figure 4). This result is far from obvious. One merit of having a formal model of cost attribution is that it enables us to give both a programmer-oriented model and an execution-oriented model and to prove them equivalent.

THEOREM 4.2.1

$$(cc, \Gamma_{init}, e, [], \{\}) \Rightarrow^* (cc', \Delta, z, [], \theta) \text{ if and only if } cc, \Gamma_{init} : e \Downarrow_\theta \Delta : z, cc'$$

PROOF. See Appendix C. □

The proof is an extension of the correctness proof of Sestoft's original small-step state transition semantics [Sestoft 1997].

4.3 The Static Scope of Cost Centers

Our intuition about our profiling semantics is that all runtime costs are attributed to the cost center which statically enclosed the expression being evaluated. We now formalize this notion and prove that this is indeed the case, using the operational semantics.

Definition 4.3.1 We identify the *static scope of a cost center* using $\varphi^{cc}[e]$ which labels e with its statically enclosing cost center cc and labels all subexpressions of e with their statically enclosing cost center.

$$\begin{aligned}
\varphi^{cc}[\lambda x.e] &= (\lambda x.\varphi^{cc}[e])^{cc} \\
\varphi^{cc}[e\ x] &= (\varphi^{cc}[e]\ x)^{cc} \\
\varphi^{cc}[x] &= x^{cc} \\
\varphi^{cc}[\texttt{let}\ \{x_i = e_i\}\ \texttt{in}\ e] &= (\texttt{let}\ \{x_i = \varphi^{cc}[e_i]\}\ \texttt{in}\ \varphi^{cc}[e])^{cc} \\
\varphi^{cc}[C\ \overline{x}_a] &= (C\ \overline{x}_a)^{cc} \\
\varphi^{cc}[\texttt{case}\ e\ \texttt{of}\ \{C_j\ \overline{x_j}_{a_j}\ \texttt{->}\ e_j\}] &= (\texttt{case}\ \varphi^{cc}[e]\ \texttt{of}\ \{C_j\ \overline{x_j}_{a_j}\ \texttt{->}\ \varphi^{cc}[e_j]\})^{cc} \\
\varphi^{cc}[\texttt{scc}\ cc_{scc}\ e] &= (\texttt{scc}\ cc_{scc}\ \varphi^{cc_{scc}}[e])^{cc}
\end{aligned}$$

For example, labeling the expression

$$\texttt{let } f \texttt{ = scc "f" } \lambda x.\texttt{(+) } x \texttt{ ((*) } x\ x\texttt{) in } f\ (f\ v)$$

(once it has been desugared) with enclosing cost center `e` gives

$$\begin{array}{l}\varphi^{\texttt{e}}[\texttt{let } f \texttt{ = } \lambda x.\ \texttt{scc "f" let } a1 \texttt{ = (*) } x\ x \texttt{ in (+) } x\ a1\\ \quad\texttt{in let } a2 \texttt{ = } f\ v \texttt{ in } f\ a2]\\ \Longrightarrow^*\\ (\texttt{let } f \texttt{ = } (\lambda x.(\texttt{scc "f" (let } a1 \texttt{ = } (((*)^{\texttt{f}}\ x)^{\texttt{f}}\ x)^{\texttt{f}} \texttt{ in } (((+)^{\texttt{f}}\ x)^{\texttt{f}}\ a1)^{\texttt{f}}\ )^{\texttt{f}}\ \ )^{\texttt{e}}\ )^{\texttt{e}}\\ \quad\texttt{in } (\texttt{let } a2 \texttt{ = } (f^{\texttt{e}}\ v)^{\texttt{e}} \texttt{ in } (f^{\texttt{e}}\ two)^{\texttt{e}}\ )^{\texttt{e}}\ )^{\texttt{e}}\end{array}$$

Observe that the scope of the `scc` annotation is labeled with the annotating cost center `f`, not `e`, and that all nonatomic arguments are bound to labeled expressions. The arguments themselves, which must be atomic, are not labeled (but see Section 8.2).

Definition 4.3.2 For any labeled expression e^l, $lab(e^l) = l$. Observe that $lab(\varphi^{cc}[e]) \equiv cc$, since $\varphi^{cc}[e]$ always labels e with cc (see Definition 4.3.1).

Definition 4.3.3 For any labeled expression e^l, $U[e^l]$ recursively removes all the labels, i.e., $U[\varphi^{cc}[e]] \equiv e$.

Definition 4.3.4 A *reduction sequence over labeled expressions*

$$(cc, \Gamma, e, S, \theta) \Rightarrow^*_{lab} (cc', \Gamma', e', S', \theta')$$

is a reduction sequence where all control and heap-bound expressions have been labeled with their statically enclosing cost center. In the initial state

$$(cc_{init}, \Gamma^{lab}_{init}, e^{lab}_{init}, [\,], \{\})$$

$e^{lab}_{init} \equiv \varphi^{cc_{init}}[e_{init}]$ and the top-level subsumed functions, bound in Γ^{lab}_{init}, are labeled with the `"SUB"` cost center.[5] When a subsumed function is entered, a modified var_1 rule relabels the duplicated right-hand side with its subsuming cost center.

$$(cc, \Gamma[x \overset{cc_z}{\mapsto} z], x^l, S, \theta) \Rightarrow^{var_1}_{lab} (\mathrm{S}(^{cc}_{cc_z}), \Gamma[x \overset{cc_z}{\mapsto} z], \mathbf{labsub}(cc_z, cc, z), S, \theta + \{cc \mapsto \mathrm{v}\})$$

$$\begin{array}{rll}\text{where} & \mathbf{labsub}(\texttt{"SUB"}, cc, (\lambda y.e)^l) & = \varphi^{cc}[U[(\lambda y.e)^l]]\\ & \mathbf{labsub}(cc_z\ \ , cc, z\ \ \ \ \ \ ) & = z\end{array}$$

Since the control is a labeled expression, all dynamic heap bindings are created with labeled right-hand sides.

We are now ready to state the main result, namely that *an expression is always evaluated in the context of its statically enclosing cost center.*

THEOREM 4.3.5

$$(cc_{init}, \Gamma^{lab}_{init}, e^{lab}_{init}, [\,], \{\}) \Rightarrow^*_{lab} (cc, \Gamma, e, S, \theta) \ \Rightarrow\ cc \equiv lab(e) \text{ and } cc \not\equiv \texttt{"SUB"}$$

[5] We address the labeling of constant applicative forms in Γ^{lab}_{init} in Appendix D.

The theorem says that when an expression e is about to be evaluated the current cost center cc will always be the one that statically encloses e, namely $lab(e)$. Since the state transition rules only attribute costs to the current cost center (see Figure 5), it follows that costs are always attributed to the statically enclosing cost center.

The theorem also establishes the fact that no expression is ever evaluated in the context of the `"SUB"` cost center, i.e., all costs arising from subsumed top-level functions are actually subsumed (an invariant mentioned in Section 3.4).

PROOF. See Appendix D. □

Though it would be possible to express and prove Theorem 4.3.5 in the abstract semantics, it is more convenient to do this in the operational semantics, since this naturally captures the idea that the property holds for every intermediate state in the computation.

4.4 Implementing the Operational Semantics

The state transition semantics are easily mapped onto a number of different abstract machines, including the G-machine [Augustsson and Johnsson 1989], the STG-machine [Peyton Jones 1992], and the TIM [Fairbairn and Wray 1987], since these abstract machines are all based on the same form of push-enter stack-based model of execution which captures the behavior of our abstract state transition semantics. (The particular abstract machines differ in their organization of lower-level details such as the representation of environments and the update mechanisms employed.) We make no attempt to prove this step correct. However, since the state transition semantics of Figure 5 is small-step, we have much greater confidence that our implementation is faithful to the original semantics than if we had tried to implement the latter directly.

A flavor of the low-level execution details for our implementation, which is based on the STG-machine, was given in Section 4.1. A key operational property of the state transition semantics (Figure 5) is that *costs are only attributed to the current cost center*. This property justifies the implementation sketched in Section 4.1, in which the consumption of time and space is simply attributed to the current cost center.

5. TRANSFORMATION

In general, there is a tension between compiler optimizations and accurate profiling. On the one hand, optimizations tend to "mix together" originally separate parts of the program, making it harder for the programmer to interpret profiling measurements; but on the other hand, it is futile to profile only unoptimized code, since its behavior may be quite different to the fully optimized program.

To permit meaningful profiling, the optimizer must maintain some invariant concerning execution costs. What invariant should that be? Clearly it should *not* be required to maintain the cost of evaluating an expression, because the whole point of optimization is to reduce these costs! Fortunately, a weaker invariant suffices, namely that the optimizer should preserve the *attribution of cost*. To be more precise we can say that

No transformation should transfer cost from one cost center to another.

A transformation is, of course, free to *reduce* the cost attributed to any cost center. Only those transformations which transfer costs from one cost center to another must be avoided.

With this goal in mind we now show that

(1) *Local transformations within the static scope of a cost center do not move cost from one cost center to another* (Section 5.1).
(2) *The inlining of top-level functions does not move cost from one cost center to another* (Section 5.2).

It follows immediately that the optimization of a large program is largely unaffected by a modest number of `scc` annotations. Transformations are only hindered at `scc` boundaries, and the most important nonlocal transformation, inlining, is unaffected.

Some programs may have many `scc` annotations, however. We therefore also address the issue of performing transformations at the `scc` boundaries introduced by the programmer (Section 5.3), developing some additional `scc` transformations which preserve the attribution of costs and enable subsequent transformations to proceed. In practice, to ensure that the code we profile is as close as possible to the fully optimized code, we do perform certain `scc` transformations that transfer a *small fixed* cost between cost centers.

Our approach has a shortcoming that is occasionally important, concerning residency. The cost semantics carefully accounts for allocation, but it gives no information whatsoever about residency (that is, the amount of live data). We have found one program whose residency is substantially affected by the mere act of profiling. Some transformation that happens to improve residency is being prevented by the profiler, a very unfortunate "Heisenberg effect." Controlling residency is notoriously difficult, and we leave this as further work [Röjemo 1995].

5.1 Local Transformations

The Glasgow Haskell compiler makes use of many simple local transformations to optimize the code [Santos 1995]. Some examples are given in Figure 6.

In Section 4.3 we proved that an expression is always evaluated in the context of its statically enclosing cost center (Theorem 4.3.5). Thus, any local transformation whose effect is entirely within the scope of a cost center can proceed unhindered, since the transformation does not move cost from one cost center to another.

For example, consider an application of the *λ-of-let* transformation within the static scope of some cost center *cc*.

$$\lambda y.\texttt{let}\ x = e_x\ \texttt{in}\ e \ \Longrightarrow_{\lambda\text{-}of\text{-}let}\ \texttt{let}\ x = e_x\ \texttt{in}\ \lambda y.e \qquad y \notin FV(e_x)$$

Labeling the left-hand-side and right-hand-side expressions with *cc* we observe that all the subexpressions (the λ, the `let`, e_x, and e) are labeled with *cc* before and after the transformation, i.e., the transformation does not modify the labeling.

$$\begin{aligned} \text{lhs:}\ & (\lambda y.(\texttt{let}\ x = \varphi^{cc}[e_x]\ \texttt{in}\ \varphi^{cc}[e])^{cc}\ )^{cc} \\ \text{rhs:}\ & (\texttt{let}\ x = \varphi^{cc}[e_x]\ \texttt{in}\ (\lambda y.\varphi^{cc}[e])^{cc}\ )^{cc} \end{aligned}$$

Since we know that the costs of evaluating these subexpressions are attributed to the labeling cost center (Theorem 4.3.5), which has not changed, we conclude that the transformation does not move costs form one cost center to another. The

app-of-λ (β)	$(\lambda y.e)\ x$	$\Longrightarrow$	$e[x/y]$
app-of-let	$(\texttt{let}\ x = e_x\ \texttt{in}\ e)\ y$	$\Longrightarrow$	$\texttt{let}\ x = e_x\ \texttt{in}\ e\ y$
app-of-case	$(\texttt{case}\ e\ \texttt{of}\ \{alt_i \texttt{->}\ e_i\})\ y$	$\Longrightarrow$	$\texttt{case}\ e\ \texttt{of}\ \{alt_i \texttt{->}\ e_i\ y\}$
λ-of-let	$\lambda y.\texttt{let}\ x = e_x\ \texttt{in}\ e$ $\Longrightarrow\ \texttt{let}\ x = e_x\ \texttt{in}\ \lambda y.e$		$y \notin FV(e_x)$
let-to-case[6]	$\texttt{let}\ x = e_x\ \texttt{in}\ e$ $\Longrightarrow\ \texttt{case}\ e_x\ \texttt{of}\ x\ \texttt{->}\ e$		e strict in x, $x \notin FV(e_x)$
case-of-known	$\texttt{case}\ C_k\ \overline{x}_{a_k}\ \texttt{of}\ \{C_j\ \overline{y}_{a_j} \texttt{->}\ e_j\}$	$\Longrightarrow$	$e_k[x_i/y_{ki}]$
case-of-case[7]	$\texttt{case}\ (\texttt{case}\ e\ \texttt{of}\ \{alt_1 \texttt{->}\ e_1\})\ \texttt{of}\ \{alt_2 \texttt{->}\ e_2\}$ $\Longrightarrow\ \texttt{case}\ e\ \texttt{of}\ \{alt_1 \texttt{->}\ \texttt{case}\ e_1\ \texttt{of}\ \{alt_2 \texttt{->}\ e_2\}\}$		

Fig. 6. Some example local transformations.

transformation does, however, affect the costs attributed, since the `let` is no longer evaluated on every application of the λ-abstraction.

When transforming expressions in a subsumed scope (labeled with the `"SUB"` cost center) we do not know the actual cost center to which these costs will be attributed at runtime. However we do know that the entire function body will be subsumed (and relabeled) with a particular subsuming cost center, whenever the function is applied (see Section 4.3). Since all the `"SUB"` labels will be relabeled with the subsuming cost center, ensuring that the compile-time attribution of cost to the `"SUB"` cost center is preserved (by preserving the `"SUB"` labeling) will preserve the attribution of cost to subsuming cost center at runtime.

5.2 Inlining Top-Level Functions

A very important global transformation is the inlining of function definitions to avoid the overheads of the function call and to enable further optimization. For example we have

$$\begin{array}{ll} f = \lambda y.e & \Longrightarrow inline\ f \Longrightarrow \\ g = \ldots\ f\ \ldots & \end{array} \quad \begin{array}{l} f = \lambda y.e \\ g = \ldots\ (\lambda y.e)\ \ldots \end{array}$$

If we are to apply this transformation when profiling we must show that this transformation preserves the attribution of costs. Consider the evaluation of the occurrence of f in the body of g. This will occur in the context of some cost center, say cc, and some heap which contains a subsumed top-level binding for the function f, say $\Gamma[f \overset{\texttt{"SUB"}}{\mapsto} \lambda y.e]$.

$$cc, \Gamma[f \overset{\texttt{"SUB"}}{\mapsto} \lambda y.e] : f \Downarrow_{\{cc \mapsto \mathrm{V}\}} \Delta : \lambda y.e, cc$$

Since the top-level function is subsumed, the result is precisely the result obtained

[6] The *let-to-case* transformation uses a default `case` binding which is not part of the language given in Figure 3. It is quite straightforward to extend the language, semantics, and results with this construct.

[7] A more general *case-of-case* transformation is used when there are multiple alternatives in each `case` (see Santos [1995]).

when the inlining is evaluated in the context of the same cost center and heap.

$$cc, \Gamma[f \overset{\texttt{"SUB"}}{\mapsto} \lambda y.e] : \lambda y.e \Downarrow_{\{\}} \Delta : \lambda y.e, cc$$

The only effect of performing the inlining transformation is to eliminate the cost V. The rest of the cost attribution is unaffected by the transformation.

5.3 Transformations at `scc` Boundaries

The fact that the cost center boundaries are identified by an explicit language construct provides a natural barrier to any "bad" transformations. For example, in the expression

$$\lambda y.\texttt{scc}\ cc\ \texttt{let}\ x = e_x\ \texttt{in}\ e$$

the *λ-of-let* transformation is not applicable because the intervening `scc` inhibits the pattern match. This is a good thing, since it forces us to be explicit about what transformations occur at `scc` boundaries. The default behavior is that no transformations occur across these boundaries.

As well as inhibiting these "bad" transformations the `scc` construct also provides us with a language that can be used to express additional `scc` transformations that preserve the attribution of costs and enable subsequent transformations to proceed. We present some of the more important `scc` transformations below.

5.3.1 *scc-of-let*. We can safely float a `let` binding outside an `scc` annotation, provided that we annotate the subexpression which is moved into the scope of a different cost center with its original cost center.

$$\texttt{scc}\ cc\ \texttt{let}\ x = e_x\ \texttt{in}\ e \ \Longrightarrow\!\textit{scc-of-let}\!\Longrightarrow\ \texttt{let}\ x = \texttt{scc}_{sub}\ cc\ e_x\ \texttt{in}\ \texttt{scc}\ cc\ e$$

The $\texttt{scc}_{sub}$ annotation is identical to an `scc` annotation, except that it does not increment the cost center's entry count, i.e., no $\{cc_{scc} \mapsto \text{E}\}$ is attributed.

$$\frac{cc_{scc}, \Gamma : e \Downarrow_\theta \Delta : z, cc_z}{cc, \Gamma : \texttt{scc}_{sub}\ cc_{scc}\ e \Downarrow_\theta \Delta : z, cc_z} \qquad SCC(sub)$$

The entry count is only incremented when the original `scc` is evaluated.

Though this transformation may not seem particularly significant on its own, it may enable other important transformations, such as floating the `let` past an enclosing λ-abstraction, to proceed unhindered.

$$\begin{array}{l}
\lambda y.\texttt{scc}\ cc\ \texttt{let}\ x = e_x\ \texttt{in}\ e \\
\Longrightarrow\!\textit{scc-of-let}\!\Longrightarrow \\
\lambda y.\texttt{let}\ x = \texttt{scc}_{sub}\ cc\ e_x\ \texttt{in}\ \texttt{scc}\ cc\ e \\
\Longrightarrow\!\lambda\textit{-of-let}\!\Longrightarrow \quad y \notin FV(e_x) \\
\texttt{let}\ x = \texttt{scc}_{sub}\ cc\ e_x\ \texttt{in}\ \lambda y.\texttt{scc}\ cc\ e
\end{array}$$

The formal cost semantics provide us with a framework in which the effect of these alternative transformations — with respect to the attribution of costs — can be examined. For example, consider the proof trees for the left-hand-side and right-hand-side expressions of the *scc-of-let* transformation in the context of some

enclosing cost center *ecc* and some heap Γ.

$$\text{lhs:}\quad \frac{\dfrac{cc, \Gamma[x' \overset{cc}{\mapsto} \widehat{e}_x] : \widehat{e} \Downarrow_{\theta_e} \Delta : z, cc_z}{cc, \Gamma : \texttt{let } x = e_x \texttt{ in } e \Downarrow_{\{cc \mapsto \mathrm{H}\}+\theta_e} \Delta : z, cc_z}}{ecc, \Gamma : \texttt{scc } cc \texttt{ let } x = e_x \texttt{ in } e \Downarrow_{\{cc \mapsto \mathrm{E}\}+\{cc \mapsto \mathrm{H}\}+\theta_e} \Delta : z, cc_z}$$

$$\text{rhs:}\quad \frac{\dfrac{cc, \Gamma[x' \overset{ecc}{\mapsto} \texttt{scc}_{sub}\ cc\ \widehat{e}_x] : \widehat{e} \Downarrow_{\theta'_e} \Delta' : z, cc_z}{ecc, \Gamma[x' \overset{ecc}{\mapsto} \texttt{scc}_{sub}\ cc\ \widehat{e}_x] : \texttt{scc } cc\ \widehat{e} \Downarrow_{\{cc \mapsto \mathrm{E}\}+\theta'_e} \Delta' : z, cc_z}}{ecc, \Gamma : \texttt{let } x = \texttt{scc}_{sub}\ cc\ e_x \texttt{ in scc } cc\ e \Downarrow_{\{ecc \mapsto \mathrm{H}\}+\{cc \mapsto \mathrm{E}\}+\theta'_e} \Delta' : z, cc_z}$$

There are two distinct implications of this transformation:

—The cost of allocating the closure is moved from *cc* to the enclosing cost center *ecc*.

—The heap closure for x is annotated with *ecc* instead of *cc*. However this does not affect subsequent evaluation, since the $\texttt{scc}_{sub}$ annotation will update the current cost center with *cc* if x is ever entered.

If we were completely rigorous about not moving costs this transformation would not be acceptable, since the cost of the allocation has been moved. However, to be "faithful" to the fully optimized execution, we allow this transformation so that further transformations, such as the *λ-of-let* transformation above, are not prevented.

To minimize the impact of the transformation we introduce an auxiliary *Let* rule which is used when the heap binding is annotated with an $\texttt{scc}_{sub}$. It annotates the heap binding directly with the $\texttt{scc}_{sub}$ cost center when it is allocated, eliminating the $\texttt{scc}_{sub}$ annotation in the heap binding.

$$\frac{ecc, \Gamma[x' \overset{cc}{\mapsto} \widehat{e}_x] : \widehat{e} \Downarrow_{\theta_e} \Delta : z, cc_z}{ecc, \Gamma : \texttt{let } x = \texttt{scc}_{sub}\ cc\ e_x \texttt{ in } e \Downarrow_{\{ecc \mapsto \mathrm{H}\}+\theta_e} \Delta : z, cc_z} \qquad \mathit{Let}(\mathit{scc}_{sub})$$

Using this rule the only effect of the *scc-of-let* transformation is to attribute the time to allocate the closure for x to the enclosing cost center, since its space is now attributed to *cc*.

5.3.2 *app-of-scc*. This transformation moves an application inside an `scc` annotation.

$$(\texttt{scc } cc\ e)\ x \ \Longrightarrow \textit{app-of-scc} \Longrightarrow\ \texttt{scc } cc\ e\ x$$

Though this transformation moves the attribution of the small, fixed cost of the application, A, inside the `scc`, the cost of evaluating the body of the function being applied is still attributed to the cost center enclosing its declaration. We allow this transformation, in spite of the (small) change to the cost attribution, as the follow on effects of moving arguments inward can be quite significant. In particular, it may enable β-reduction or eliminate a partial application.

5.3.3 *let-of-scc*. This transformation compliments the *scc-of-let* transformation. It enables `let` bindings to be floated inside an `scc` annotation, provided the enclosing cost center (denoted *ecc*) is known at compile time.

$$\texttt{let } x = e_x \texttt{ in scc } cc\ e \ \Longrightarrow \textit{let-of-scc} \Longrightarrow\ \texttt{scc } cc \texttt{ let } x = \texttt{scc}_{sub}\ ecc\ e_x \texttt{ in } e$$

Unfortunately, if the enclosing cost center is not known at compile time this transformation cannot be performed.

The *let-of-scc* transformation facilities the inlining of local definitions inside an `scc` annotation. For example we have

$$
\begin{array}{l}
\texttt{let } f = \lambda y.e \texttt{ in scc } cc\ f\ x \\
\Longrightarrow \textit{let-of-scc} \Longrightarrow \\
\texttt{scc } cc \texttt{ let } f = \texttt{scc}_{sub}\ ecc\ \lambda y.e \texttt{ in } f\ x \\
\Longrightarrow \textit{inline } f \Longrightarrow \\
\texttt{scc } cc\ (\texttt{scc}_{sub}\ ecc\ \lambda y.e)\ x \\
\Longrightarrow \textit{app-of-scc} \Longrightarrow \\
\texttt{scc } cc\ \texttt{scc}_{sub}\ ecc\ (\lambda y.e)\ x \\
\Longrightarrow \beta \Longrightarrow \\
\texttt{scc } cc\ \texttt{scc}_{sub}\ ecc\ e[x/y]
\end{array}
$$

Note that, even though the `scc` cc annotation now appears trivial (all the costs are attributed to ecc), it cannot be eliminated, since it is still responsible for incrementing cc's entry count.

5.3.4 *Eliminating* $\texttt{scc}_{sub}$ *Annotations.* Multiple applications of the floating transformations above can introduce redundant $\texttt{scc}_{sub}$ annotations. For example, floating a `let` binding out of two `scc` annotations will result in nested $\texttt{scc}_{sub}$ annotations.

$$
\begin{array}{l}
\texttt{scc } cc1\ \texttt{scc } cc2\ \texttt{let } x = e_x \texttt{ in } e \\
\Longrightarrow \textit{scc-of-let} \Longrightarrow \\
\texttt{scc } cc1\ \texttt{let } x = \texttt{scc}_{sub}\ cc2\ e_x \texttt{ in scc } cc2\ e \\
\Longrightarrow \textit{scc-of-let} \Longrightarrow \\
\texttt{let } x = \texttt{scc}_{sub}\ cc1\ \texttt{scc}_{sub}\ cc2\ e_x \texttt{ in scc } cc1\ \texttt{scc } cc2\ e
\end{array}
$$

The redundant annotation, $\texttt{scc}_{sub}\ cc1$, is subsequently eliminated by applying the scc_{sub}-*of-*scc_{sub} transformation.

$$\texttt{scc}_{sub}\ cc1\ \texttt{scc}_{sub}\ cc2\ e \Longrightarrow scc_{sub}\textit{-of-}scc_{sub} \Longrightarrow \texttt{scc}_{sub}\ cc2\ e$$

It is also possible to eliminate an $\texttt{scc}_{sub}$ annotation which is moved back into the scope of its cost center using the scc_{sub}-*of-ecc* transformation.

$$\texttt{scc}_{sub}\ cc\ e \Longrightarrow scc_{sub}\textit{-of-ecc} \Longrightarrow e \qquad (cc \equiv ecc)$$

6. OVERHEADS

An important aspect of any profiler is the overhead it imposes — large overheads reduce the feasibility of profiling large applications, especially if they are interactive.

The overheads of our profiler can be broken down into a number of distinct components:

(1) The execution overhead of manipulating the current cost center, attaching cost centers to closures, saving cost centers on the stack, etc. These costs are independent of the `scc` annotations in the program.

(2) The overheads introduced by the particular `scc` annotations in the program. This includes the cost of executing the `scc` annotations, as well as an indirect effect arising from optimizations which were inhibited by the `scc` annotations (see Section 5).

(3) The servicing of timer interrupts.

(4) The additional garbage collection costs arising form the extra cost center word stored in each closure and a heap-profiling test whenever a closure is collected.

(5) The additional overheads associated with heap profiling. This involves taking a complete census of the heap at regular intervals.

We measured these overheads of the modified profiler of Section 7.2 on a sample of "real" programs from our `nofib` test suite [Partain 1993]. By "real" we mean that each is an application program written by someone other than ourselves to solve a particular problem. None was designed as a benchmark, and they range in size from a few hundred to a few thousand lines of Haskell.

Table III presents the basic execution overheads of our profiler. The first column gives the absolute number of SPARC instructions executed by each program, compiled with `ghc-0.26 -O`, i.e., without profiling but with optimization. In examining the basic execution overhead we are interested in determining the overheads of the code actually generated by the compiler — we exclude all instructions executed by the runtime system. The SPARC instruction counts were collected with SpixTools.

"`-prof` no `sccs`" reports the execution overhead when profiling with no `scc` annotations in the program. This corresponds to (1) above. The overhead ranges from 36% to 51% , with geometric mean 44%.

"`-prof -auto-all`" reports the execution overhead when profiling with *every* top-level definition annotated with an `scc`. This corresponds to (1+2) with a worst case for (2). It ranges from 39% to 75%, with mean 54%. "Total `scc` count" gives the total number of `scc` annotations evaluated during the run of the program.

We also measured what component of this `scc` overhead was due to the effect of the `scc` annotations on the optimization of the program ("Reduced Optimization"). This was done by comparing unprofiled execution with the execution of a program that was compiled and optimized with the `scc` annotations but then had *unprofiled* code generated. It is rather encouraging that the optimization effect is quite small — ranging from 0% to 6%.

Table IV presents the total space and time overheads (1-4) for profiling with `-auto-all` as seen by the user — i.e., including all runtime system costs.

"Size" gives the binary size of the executable image. The executable for profiled execution is typically about 60% larger. This is made up of the additional profiling code (both generated and runtime system), static data structures for each cost center, and static "description" strings used by the heap profiler.

"Resid" (residency) gives the average amount of live data during execution. The residency numbers were gathered by sampling the amount of live data at frequent intervals, using the garbage collector. The residency overhead of the profiler is about 33% which corresponds to adding a cost center word to an average closure size of three words. (This extra word is discounted when reporting heap profiling and collecting allocation costs.)

"Time" reports the *total* execution time, including the cost of garbage collection

Table III. Basic Execution Overheads of Cost Center Profiling

`nofib` Program	Unprofiled Non RTS Instrs (K)	`-prof` no `sccs` Overhead	`-prof` `-auto-all` Overhead	Total `scc` count (K)	Reduced Optimization
real/HMMS	1,879,669	42%	48%	9,185	−0%
real/anna	226,853	46%	59%	1,584	1%
real/bspt	19,172	49%	62%	36	3%
real/compress	817,767	46%	55%	6,918	0%
real/fulsom	852,191	43%	52%	4,608	0%
real/gamteb	205,772	36%	39%	80	2%
real/gg	25,583	41%	52%	34	6%
real/hidden	1,843,920	51%	75%	12,143	6%
real/hpg	125,486	41%	49%	90	4%
real/infer	134,523	46%	48%	118	1%
real/parser	79,974	44%	68%	611	4%
real/pic	18,289	39%	44%	2	3%
real/reptile	20,540	48%	49%	8	2%
Geometric Mean		44%	54%		2%

Table IV. Total Space and Time Overheads of `-auto-all` Profiling

`nofib` Program	Unprofiled			`-prof -auto-all`				`-Onot`
	Size (KB)	Resid (KB)	Time (User)	Time Profiling Size	Resid	`-pC` Time	`-hC` Time	`-pC` Time
real/HMMS	762	1,157	391.2s	61%	36%	46%	+167%	332%
real/anna	1,176	326	35.2s	90%	38%	76%	+66%	134%
real/bspt	664	313	3.4s	49%	19%	69%	+19%	138%
real/compress	264	157	136.6s	36%	30%	44%	+20%	88%
real/fulsom	624	1,155	177.3s	70%	27%	73%	+116%	149%
real/gamteb	448	299	68.7s	54%	34%	27%	+14%	107%
real/gg	592	344	5.0s	66%	29%	56%	+41%	99%
real/hidden	544	206	276.7s	56%	32%	85%	+23%	505%
real/hpg	584	494	36.2s	63%	37%	58%	+44%	132%
real/infer	360	925	20.7s	62%	32%	59%	+122%	104%
real/parser	545	497	13.2s	64%	34%	92%	+53%	188%
real/pic	408	246	3.8s	49%	25%	47%	+20%	202%
real/reptile	432	452	3.4s	74%	34%	55%	+37%	109%
Geometric Mean	569	505	90.1s	61%	31%	61%	+57%	176%

Table V. Comparison of Execution Profiling Overheads

Profiler	Language	Overhead
`gcc -p` (`prof`)	C	10%
`gcc -pg` (`gprof`)	C	20%
SML/NJ	SML	32%
`ghc -prof -auto-all`	Haskell	61%

and other runtime system routines,[8] on a Sun 4/60 with 28MB of memory (averaged over four runs). The mean overhead of 61% includes a 1-2% overhead incurred servicing timer interrupts.

Though the 61% total overhead is quite large, we believe that it is acceptable. There is a substantial benefit in performance, as well as faithfulness, from profiling optimized code. If we were instead to profile unoptimized code the mean overhead (when compared to normal optimized execution) rises to 176% (see the "`-Onot`" column in Table IV). Table V compares these overheads with the overheads of other execution profilers — it is not surprising that our overheads are significantly higher, since profiling in the presence of lazy evaluation requires additional bookkeeping.

Finally, the "`-hC`" column in Table IV gives the additional time overhead of heap profiling (5) when one heap census is taken every second. The heap-profiling overhead is directly related to the average residency, since the entire live heap must be traversed during each census. For a program with a typical residency of 500KB the additional heap-profiling overhead is about +55%. The heap-profiling overhead is easily controlled by specifying the interval between heap censuses.

7. CONSTANT APPLICATIVE FORMS

It would be nice to report that the profiler always does the "Right Thing" — that is, "what the programmer expected" — but our experience is otherwise: we found one important case when it does not. This experience has led us to complicate the profiler to make it appear simpler and more intuitive to the programmer.

Consider the top-level definition: `x :: Int`

`x = nfib 20` This sort of top-level definition, which has no arguments, is called a *constant applicative form* or *CAF*. When its value is first demanded, its right-hand side is evaluated, and the CAF is updated with the resulting value. Subsequent demands for its value incur no cost. Where, though, should the cost of the first evaluation be attributed? Since evaluation of `x` will be initiated by the *Var*(*thunk*) rule, `nfib 20` is evaluated in the context of the cost center (cc_e) attached to its heap binding. Since `x` is a top-level binding this must be specified in the initial heap, Γ_{init} (see Section 3.3).

We arrange for the cost of evaluating `x` to be attributed to a special `"CAF"` cost center by attaching it to the CAF binding in Γ_{init}.[9]

$$\begin{array}{lcll} \Gamma_{init} & = & \{\ x \overset{cc}{\mapsto} e :\ x = e \in prog \text{ and } cc = \mathbf{initcc}(e)\ \} & \\ \mathbf{initcc}(e) & = & \texttt{"SUB"} & \text{if } e \equiv \lambda y.e' \\ & = & \texttt{"CAF"} & \text{otherwise} \end{array}$$

[8] Note that, the non-gc runtime system costs do not incur any profiling overhead. If these fixed costs make up a significant proportion of execution the total time overhead reported may be less than the basic execution overhead reported in Table III (cf. gamteb). Though the profiler does attribute the costs of executing these routines to the cost center responsible for calling them, the relative costs are slightly undervalued because they do not incur any overhead.

[9] It does not make sense to attach the `"SUB"` cost center to CAF bindings, since we do not intend for the cost of evaluating a CAF to be subsumed. If we did arrange for the one-off cost of evaluating a CAF to be subsumed by the first cost center to demand its value (modifying the *Var*(*thunk*) rule) the cost attribution would not be independent of the evaluation order, and Theorem 3.5.1 would not hold.

In practice we often attach a specific cost center to each CAF, for example, `"CAF:x"`, so that the programmer can identify the costs associated with each CAF.

7.1 Unexpected "CAF" Cost Attribution

Unfortunately this treatment of CAFs gives some unexpected results. Consider the following two definitions:

```
and1,and2 :: [Bool] -> Bool
and1 xs = foldr (&&) True xs
and2    = foldr (&&) True
```

Any "red-blooded" functional programmer would expect these two definitions to be entirely equivalent: after all, the only difference is an η-reduction. However, they are given different cost centers in the initial environment, Γ_{init}, since `and1` is a top-level function while `and2` is a CAF:

$$\Gamma_{init} = \{\ \texttt{and1} \overset{\texttt{"SUB"}}{\mapsto} \lambda\texttt{xs.}\ \texttt{foldr (\&\&) True xs},\ \texttt{and2} \overset{\texttt{"CAF"}}{\mapsto} \texttt{foldr (\&\&) True},\ \ldots\ \}$$

This results in the costs of applying `and1` being subsumed by its call sites, while *the costs of applying* `and2` *are attributed to the cost center* `"CAF"`. A major change in cost attribution has resulted from an innocuous change in the program.

The following declaration highlights the problem:

```
y :: Int -> Int
y = if <expensive> then (\a->e1) else (\a->e2)
```

There are two sorts of cost associated with `y`: the one-off costs of deciding which of the two λ-abstractions is to be `y`'s value, and the repeated costs of applying that λ-abstraction. The cost semantics of Figure 4 attribute both these costs to the `"CAF"` cost center, whereas the programmer would probably expect the costs of applying `y` to be subsumed by its call site. However, the one-off cost of deciding which λ-abstraction is to be applied should still be attributed to the `"CAF"` cost center.

Attributing the cost of applying a λ-abstraction declared within the scope of a CAF to the `"CAF"` cost center turns out to be a real problem in practice. There are a number of common situations where this causes unexpected results:

—Our implementation of overloading in Haskell uses *dictionaries*, which are tuples of method functions. All the costs of executing these methods are attributed to the `"CAF"` cost center enclosing the dictionary definition. (The details are in Appendix E.)

—An important transformation is the floating of constant expressions outside a λ-abstraction. This may turn a subsumed function into a CAF. For example we could have

$$\begin{array}{l}\texttt{f = }\lambda\texttt{y.let z = }\mathit{exp}\texttt{ in g z y}\\ \Longrightarrow\!\mathit{\lambda\text{-of-let}}\!\Longrightarrow \quad y \notin FV(\mathit{exp})\\ \texttt{f = let z = }\mathit{exp}\texttt{ in }\lambda\texttt{y.g z y}\end{array}$$

Now the costs of applying `f` are attributed to the `"CAF"` cost center.

—Combinator-style programming results in a lot of CAFs which evaluate to λ-abstractions. An example taken from Hutton [1992] is

```
parse :: [Char] -> Script
parse = fst.hd.prog.strip.fst.hd.lexer.prelex
```

Though `parse` has a function type, the costs of applying the `parse` combinator are attributed to the `"CAF"` cost center (unless subsequent transformation η-expands the definition of `parse`).

In general, the sensitivity of the profiling results to the presence or absence of CAFs is very undesirable.

7.2 A Modified Cost Semantics

How, then, can we modify the formal cost semantics to deal with the unexpected attribution of costs to CAFs? Our solution is to extend the notion of subsumed cost to λ-*abstractions* with the `"CAF"` cost center. The one-off costs of evaluating a CAF are attributed to the `"CAF"` cost center, but the repeated costs of evaluating the body of a λ-abstraction which is attributed to a CAF are subsumed. Note that any λ-abstractions declared in the body of a function which is subsumed by a CAF will be attributed to the CAF and hence subsequently subsumed. This includes partial applications of top-level functions.

The modified cost semantics are given in Figure 7. In the *Var*(*whnf*) rule the subsuming predicate S is extended to return the current cost center for any λ-abstraction with a `"SUB"` or `"CAF"` cost center. Previously we only returned the current cost center if the closure had a `"SUB"` cost center (compare Figure 4).[10]

The *Var*(*thunk*) rule must also be modified to deal with an as yet unevaluated CAF closure. Since the one-off costs of evaluating a CAF should still be attributed to the `"CAF"` cost center we always evaluate the bound expression e in the context of the cost center cc_e. However, if the result of that evaluation z is a λ-abstraction with the cost center `"CAF"` we return the enclosing cost center cc instead. The closure is still updated with the result cost center cc_z so that subsequent references to the closure will be appropriately subsumed by the *Var*(*whnf*) rule. In this way the costs of repeatedly evaluating the body of a λ-abstraction, declared within the scope of a CAF, are subsumed by the application sites.

The corresponding modifications to the state transition rules are also given in Figure 7. Apart from the use of the extended predicate, $\mathrm{S}(^{cc}_{cc_z}\, z)$, the main difference is that when evaluation of a thunk is begun, the demanding cost center cc is saved on the stack along with the update marker $\#x$ (rule var_2). When evaluation of the thunk is complete, the demanding cost center is restored if the result has the cost center `"CAF"` (rule var_3). This mechanism for saving and restoring the current cost center is very similar to that used in the *case* rules in Figure 5, which save the current cost center before evaluating the scrutinee and restore it afterward.

7.2.1 *Implications.* A subtle implication of these modifications is that a runtime test is now required in the var_1 and var_3 rules to determine $\mathrm{S}(^{cc}_{cc_z}\, z)$.

In our implementation the var_1 rule is implemented by entering — that is, jumping to the code pointer of — the closure for x, relying on the code inside the closure to load the current cost center ccc as appropriate. There are two distinct cases:

Entering a λ-Abstraction. In the original semantics (Figure 5) the cost center to be loaded by a function closure could be statically determined: top-level (subsumed)

[10] In the original semantics z was not needed as an argument to the predicate S, as the `"SUB"` cost center was only ever attached to top-level λ-abstractions.

$$cc, \Gamma[x \overset{cc_z}{\mapsto} z] : x \Downarrow_{\{cc \mapsto \mathrm{V}\}} \Gamma[x \overset{cc_z}{\mapsto} z] : z, \mathrm{S}\binom{cc}{cc_z\ z} \qquad \mathit{Var(whnf)}$$

$$\frac{cc_e, \Gamma : e \Downarrow_\theta \Delta : z, cc_z}{cc, \Gamma[x \overset{cc_e}{\mapsto} e] : x \Downarrow_{\{cc \mapsto \mathrm{V}\} + \theta + \{cc_z \mapsto \mathrm{U}\}} \Delta[x \overset{cc_z}{\mapsto} z] : z, \mathrm{S}\binom{cc}{cc_z\ z}}\ e \not\equiv z \qquad \mathit{Var(thunk)}$$

$$\textbf{where}\ \begin{aligned} \mathrm{S}\binom{cc}{\texttt{"SUB"}\ \lambda y.e} &= cc \\ \mathrm{S}\binom{cc}{\texttt{"CAF"}\ \lambda y.e} &= cc \\ \mathrm{S}\binom{cc}{cc_z\ z} &= cc_z \end{aligned}$$

ccc	Heap	Control	Stack	θ	$\Longrightarrow$	ccc	Heap	Control	Stack	θ	Rule
cc	$\Gamma[x \overset{cc_z}{\mapsto} z]$	x	S	θ	$\Longrightarrow$	$\mathrm{S}\binom{cc}{cc_z\ z}$	$\Gamma[x \overset{cc_z}{\mapsto} z]$	z	S	$\theta + \{cc \mapsto \mathrm{V}\}$	var_1
cc	$\Gamma[x \overset{cc_e}{\mapsto} e]$	x	S	θ	$\Longrightarrow$	cc_e	Γ	e	$\binom{\#x}{cc} : S$	$\theta + \{cc \mapsto \mathrm{V}\}$	var_2
cc_z	Γ	z	$\binom{\#x}{cc} : S$	θ	$\Longrightarrow$	$\mathrm{S}\binom{cc}{cc_z\ z}$	$\Gamma[x \overset{cc_z}{\mapsto} z]$	z	S	$\theta + \{cc_z \mapsto \mathrm{U}\}$	var_3

In the var_1 and var_3 rules $\mathrm{S}\binom{cc}{cc_z\ z}$ is as defined above.
In the var_2 rule $e \not\equiv z$.

Fig. 7. Modified cost semantics and state transition rules.

functions leave *ccc* unchanged, while all local (nontop-level) functions load *ccc* with the cost center stored in the closure. With the modified var_1 rule, however, local functions now need to test the cost center attached to the closure, loading it into *ccc* only if it is not a `"CAF"` cost center. As an optimization, if an `scc` encloses a local function definition then the cost center attached to its closure is statically known, so the dynamic test is not needed.

Entering a Constructor. The code on entering a constructor unconditionally loads *ccc* with cc_z from the closure — see the definition of $\mathrm{S}(^{cc}_{cc_z}\, z)$ in Figure 7.

There are also two distinct cases for the var_3 rule:

Updating with a λ-Abstraction. The code must test the current cost center *ccc*, overwriting it with the cost center saved with the update marker, only if *ccc* is a `"CAF"` cost center. Since a single piece of code performs all λ-abstraction updates a dynamic test is always performed.

Updating with a Constructor. No test is required, since *ccc* is never modified.

Note that the runtime test only determines the choice of cost center at particular points in the computation — it has no effect on the value of the computation or the termination properties.

7.3 Discussion

Is the cure worse than the disease? We believe not: before we made this change we received many messages from confused users asking why a large fraction of their program's execution costs were attributed to `"CAF"`. Having made these modifications all the unexpected results produced by the profiler were found to be caused by unexpected properties of the program being profiled and not of the profiler itself.

8. VARIATIONS ON THE THEME

In this section we discuss three possible variants of our basic design.

8.1 Evaluation Scoping

Consider the expression

```
let f = scc "fun" (\x y -> x*y+1)
in scc "app" f 23 16
```

To which cost center should the cost of computing `(23*16+1)` be attributed? There are two alternatives:

Static Scoping. Attribute the cost to `"fun"`, the cost center which statically encloses (or subsumes) the λ`xy`-abstraction. (This is the choice we made in Section 3.1.)

Evaluation Scoping. Attribute the cost to `"app"`, the cost center active when the function `f` is applied.

In effect, static scoping attributes the cost of executing a function body to the cost center which encloses the *declaration site*, while evaluation scoping attributes the cost of executing the function body to the cost center of the function's *application site*. It turns out that the difference between static and evaluation scoping has a

precise and elegant manifestation in the semantics of Figure 4: to use evaluation scoping it suffices to change cc_λ to cc in the second assumption of the *App* rule.

The practical implications of this distinction are not immediately obvious. Consider the expression:

```
let f = scc "fun" exp
in (scc "app1" f a) + (scc "app2" f b)
```

Static scoping attributes the total cost associated with the declaration of `f` to the cost center `"fun"`. This includes both the cost of evaluating *exp* to a λ-abstraction and the cost of applying `f`. (The cost centers `"app1"` and `"app2"` are only attributed with the small cost of invoking the application.) In contrast, evaluation scoping provides a finer breakdown of costs, attributing the cost of evaluating *exp* to `"fun"` and the costs of applying `f` to the respective application sites `"app1"` and `"app2"`.

We initially implemented both profiling schemes and compared them in practice. Our experience is that static scoping usually measures "what the programmer expected," while evaluation scoping often does not. The main reason is that evaluation scoping is very sensitive to whether an `scc` construct is placed around the complete application of a function or not; for example, the costs attributed to `"f"` in `(scc "f" f) x` are very different to those attributed to `"f"` in `(scc "f" f x)`. Although this allows evaluation scoping to express distinctions which are inaccessible in static scoping, in practice such distinctions are more of a hindrance than a help. It also turns out that maintaining these extra distinctions imposes rather onerous additional restrictions on program transformation. As a result we abandoned evaluation scoping altogether.

8.2 Higher-Order Functions and Lexical Profiling

The cost semantics of Figure 4 attribute the costs of applying a subsumed function to the cost center enclosing its application site. However, for a subsumed function which is passed as a higher-order function argument the cost center enclosing the eventual application site may bear no relation to the cost center which enclosed the actual reference to the function in the source. For example, in

```
f x = expensive 2 x
h i = scc "h" (i 3)
g   = scc "g" h f
```

`f` is referred to in the scope of `"g"` but is applied in the scope of `"h"` after being substituted for `i`. The cost of evaluating the body of `f` accrues to `"h"`. A different behavior occurs if a local function or a partial application is passed to `h`. For example, in

```
f' w x = expensive w x
h  i   = scc "h" (i 3)
g'     = scc "g'" h (f' 2)
```

a local closure is allocated for the partial application `f' 2` which records the enclosing cost center `"g'"`. When this closure is entered in the body of `h` the cost center `"g'"` is loaded, and then `f'` is entered. The cost of evaluating the body of `f'` accrues to `"g'"`. The problem with the first example is that the argument `f` does

not carry any information which identifies the cost center of the scope in which it was referenced.

This problem is addressed by Clack et al. [1995]. They introduce the notion of *lexical profiling*, which specifies that the costs of a subsumed function be subsumed by the scope which made the original reference to the function. To keep track of the scope of the original reference to an actual parameter their implementation attaches profiling information to every field of every graph node in the graph, as well as the graph node itself.

It would be quite possible to extend our cost semantics to keep track of the cost center enclosing the original reference to an actual parameter when it is substituted for its formal parameter.[11] However, we have not developed this approach, since the corresponding modifications to closure layout and argument-passing conventions in a compiled, registerised implementation are substantial.

An alternative approach is suggested by the observation that most of the additional cost center information is redundant — the actual parameter is normally bound to a closure which has its original cost center attached. The only situation where this is not the case is when a top-level subsumed function is passed as an argument, since no closure is allocated. This can be remedied by allocating a dummy closure which records the original cost center, i.e.,

> *Whenever a top-level function is passed as an argument, introduce a local* `let` *binding for that argument:*
>
> $e\ f \implies \texttt{let}\ f' = f\ \texttt{in}\ e\ f'$ (f is a top-level function)

This *boxing transformation* is identical to the process of `let` binding nonatomic arguments when translating into the core language (Section 3.2).

The transformation ensures that the costs of evaluating top-level functions are attributed to the cost center enclosing the scope which made the original reference to the function, rather than the cost center enclosing the scope in which the function is eventually applied. In the example above, we would replace the application `(h f)` with the expression `(let f' = f in h f')`. The top-level function `f` is now passed in a locally allocated closure `f'` which is annotated with `"g"` — the cost center of the scope in which it was referenced. When `f'` is entered in `h` the cost center `"g"` is loaded, and then `f` is entered, as before. However, now the costs of evaluating the body of `f` accrue to `"g"`.

Note that these dummy `let` bindings appear to be simple renamings, which the compiler normally inlines freely. Care must be taken to ensure that they are treated as real `let` bindings which are only floated into the scope of another cost center or inlined in the scope of another cost center if they can be annotated with their original cost center (see Section 5.3). Finally, observe that this transformation is

[11] If the cost semantics (Figure 4) were extended to maintain the cost center of the original reference to an actual parameter Theorem 4.3.5 could be proved for a notion of *lexical scope*, defined by extending (1) the static scope labeling (Definition 4.3.1) to label *all* variables with their enclosing cost center and (2) a relabeling of subsumed functions (Definition 4.3.4) which labeled them with the cost center of their original reference. The correctness of the original semantics + boxing transformation could then be established by proving it equivalent to the extended semantics. This is left to future work.

not always necessary: if it can be determined at compile time that f is always applied in the same scope as its original reference then the transformation can be omitted.

There is a design choice to be made here. Adopting a notion of lexical scoping provides more consistent profiling results, but at the cost of additional overhead — both execution and implementation effort. In our experience, subsuming costs to their application site has not caused any problems in practice. Consequently, our current implementation does not perform this transformation.

8.3 Inheritance Profiling

Many profilers offer some way to aggregate the costs of executing a function with those of its caller. There are two distinct approaches:

(1) *Subsumption*: the costs of executing an unprofiled function are attributed to its caller [Appel et al. 1988; Clack et al. 1995].

(2) *Inheritance*: the costs of executing a profiled function are attributed to the function *and* to the stack of functions responsible for the call [Graham et al. 1983; Morgan and Jarvis 1995].

Our profiler supports subsumption, but not inheritance — it produces a flat profile, with costs only attributed to the immediately enclosing cost center. However, it is possible to extend the profiler with a mechanism for inheriting these costs to the stack of enclosing cost centers.

One possibility would be to use `gprof`-style statistical inheritance [Graham et al. 1983]. By attributing costs to a cost center pair, consisting of the two immediately enclosing cost centers, a postprocessor can inherit the costs up the enclosing-cost-center graph. Unfortunately, the accuracy of this scheme relies on the assumption that the average cost of evaluating the `scc` expression is independent of the enclosing cost center. If this is not the case an incorrect proportion of the costs are attributed to the different enclosing cost centers. This is particularly problematic in a nonstrict language where the amount of evaluation is dependent on the arguments passed *and* the demand placed on the result by the enclosing context.

An alternative approach has been taken by Morgan and Jarvis [1995] who have developed an extension to our implementation which accurately attributes costs to the *set* of enclosing cost centers during execution.[12] As well as maintaining the current cost center set, each closure must record the cost center set which was responsible for creating it. Much work has gone into minimizing the overheads of manipulating the cost center sets. The details can be found in Morgan and Jarvis [1995].

In terms of the formal framework presented here, it is quite straightforward to extend the semantics to maintain a current cost center set and attribute costs to cost center sets. The main implication for the compilation process as a whole is that the set of enclosing cost centers must now be maintained by any compiler transformations. This can be captured by labeling every expression with the set of enclosing cost centers and ensuring that this labeling is maintained by the compiler transformations. Under this regime the *let-of-scc* and scc_{sub}*-of-*scc_{sub} transformations

[12] A *set* of enclosing cost centers is used to solve problems caused by cyclic call graphs.

introduced in Section 5.3 are no longer valid.

8.4 Profiling Strict Languages

It is more straightforward to profile a strict language than a nonstrict one, since there are no unevaluated closures to worry about. The semantics in Figure 4 are easily "strictified" by the following: restricting the *Let* rule to only bind a set of mutually recursive function values; using a default `case` binding (see Section 5.1) to evaluate and bind the value of a strictly evaluated expression; and omitting the $Var(thunk)$ rule.

The result is a profiling scheme very similar to the SML/NJ profiler developed by Appel et al. [1988] (see Section 9.1).

9. RELATED WORK

Relatively few source-level execution profilers have been developed for high-level functional languages. Notable exceptions are

—the SML/NJ execution profiler (Section 9.1),

—the UCL Lexical profiler (Section 9.2), and

—the Runciman, Wakeling, Röjemo space profilers (Section 9.3).

Of these, only the UCL profiler profiles the execution time of a nonstrict language, but it only has an interpreted implementation.

9.1 The SML/NJ Profiler

Appel et al. [1988] develop a simple execution profiler for the New Jersey implementation of SML — a strict functional language. They report the call counts and execution time for a set of "profiled functions" which may include separately profiled local functions. They introduce the idea of a "current function," which is set whenever a profiled function is called and reset when execution returns from a call. Regular timer interrupts are attributed to the current function. Since unprofiled functions do not set the current function their costs are subsumed by the calling function. These ideas correspond directly with our notions of static scope, "current cost center," and subsumed top-level functions.

In their implementation only the profiled functions are compiled specially — all unprofiled functions use exactly the same code as normal execution. Unfortunately, this means that the unprofiled functions do not reset the current function when execution returns from a function call. If any of the functions called from an unprofiled function are profiled, the rest of the execution of the unprofiled function will be incorrectly attributed to the profiled function called. In contrast, we require that all code must be compiled for profiling, even if it is "unprofiled." In this way we ensure that the cost center is always restored when evaluation (invoked by a `case` expression) returns.

Their implementation uses side effects to manipulate the current function at the language level. The effect that these manipulations have on subsequent optimization is somewhat unclear, since it is dependent on the optimizer's treatment of the assignment operations which manipulate the current function. Our work could be viewed as a development of the SML profiler, extending it to nonstrict languages and providing a formal model to undergird it.

9.2 The UCL Lexical Profiler

Clack et al. [1995] have developed a time and space profiler for a nonstrict, interpreted graph reduction system. They report the call counts and the time and space usage for a set of "profiled functions." The costs of unprofiled functions are subsumed. Each of the profiled functions is assigned a unique "color." Their notion of "color" is similar to our "cost center" — both are attributed with the costs identified during execution.

Though there are encouraging similarities between the two profilers, there are also some significant differences.

—As discussed in Section 8.2 the UCL profiler adheres to the notion of *lexical scope*, attributing the costs of applying a higher-order argument to the color enclosing the original reference to the function. In our profiling scheme the cost of *top-level* higher-order arguments are subsumed by the cost center enclosing their eventual application site. However, we have proposed an extension to our scheme which implements lexical profiling.

—If a particular function is referenced (directly or indirectly) by two or more profiled functions, then it implicitly becomes a separately profiled function. They rely on the the attribution to color-pairs (see below) to distinguish the costs from different callers. In contrast, our profiler dynamically subsumes the cost of all unprofiled functions even if they are shared. We believe this to be a major strength of our profiling scheme, since it enables the costs of the logical "parts" of a program to be accurately subsumed, regardless of the sharing properties of the program.

—The UCL profiler attributes costs to color-pairs. This enables them to distinguish the costs of `map` called from `f` and `map` called from `g`. In fact, it provides enough information to perform `gprof`-style statistical inheritance postprocessing, though they have not actually implemented this. Currently we do not perform any inheritance, relying on unhindered subsumption instead (but see Section 8.3).

—In the UCL profiler an unshared CAF is subsumed by the scope which references it. Both the one-off costs and any repeated costs are attributed to the subsuming scope. However, a shared CAF is profiled separately. The one-off costs are attributed to the `ref1-caf` color-pair where `ref1` is the color of the first reference to demand the value of the CAF and where `caf` is the color of the CAF. Subsequent references to the CAF (or a λ-abstraction embedded within its result) cause the graph to be recolored with the appropriate `ref-caf` color-pair. In contrast, we always profile CAFs separately.[13] The one-off costs are attributed to the CAF cost center, and any repeated costs are subsumed (see Section 7).

Finally, their prototype implementation profiles *interpreted* graph reduction of a core functional language. They do not attempt to address the issues of program transformation or separate compilation. Since the prototype has only been applied to modest examples, it remains to be seen how their scheme scales to larger applications.

[13] If we know at compile time that a CAF is unshared there is no reason why it cannot be labeled with the `"SUB"` cost center and arrange for all its costs to be subsumed. A modified *Var*(*thunk*) rule can easily arrange for this.

9.3 Space Profilers

The **hbc/lml** heap profiler [Runciman and Wakeling 1993] was developed concurrently with our heap profiler and has very similar goals — indeed, we use its postprocessor to produce PostScript graphs. Aside from the absence of time profiling, the main difference from our work is that the **hbc/lml** profiler does not provide a mechanism for subsuming information about uninteresting functions up the call graph. For example, a profile may indicate that heap objects allocated by a certain function, say `map`, occupy a large amount of heap space. However there is no mechanism to determine which application(s) of `map` was (were) responsible for producing these cells [Kozato and Otto 1993].

More recently the **nhc** heap profiler [Röjemo and Runciman 1996; Runciman and Röjemo 1996] extends the **hbc/lml** heap-profiling ideas. It provides several additional heap profiles which attempt to identify the cause, rather than merely the presence, of an unexpected space leak.

10. CONCLUSIONS AND FURTHER WORK

A version of our profiler has been distributed with the Glasgow Haskell compiler since June, 1993, enabling other users to profile their Haskell applications. The largest applications profiled to date are the Glasgow Haskell Compiler (GHC) itself [Sansom 1994] and LOLITA[14] — a natural-language system developed by the Artificial Intelligence Research Group at the University of Durham [Morgan et al. 1994]. Each of these applications consist of over 30,000 lines of Haskell code, divided into more than 100 different source modules.

The experience gained actually using the profiler to profile real applications has provided invaluable feedback. It enabled us to compare the usability of alternative cost attribution semantics, and it drew our attention to the problem with the attribution of CAFs. It also prompted Morgan and Jarvis [1995] to develop an extension to our profiler which arranges for costs to be inherited (see Section 8.3).

The cost center profile has proved to be a very useful tool. As well as quickly identifying the execution hot spots in a program, it enables the performance of alternative implementations of a particular algorithmic component within a program to be easily compared. When space is identified as a bottleneck the heap profiles provide a very illuminating window on the space behavior. However, it often proves very difficult to identify the *cause* of an unexpected space leak — further work certainly needs to be done to aid this task [Runciman and Röjemo 1996].

A key contribution of our work has been the development of a formal model for cost attribution. This provides a precise framework in which to discuss design decisions and prove certain properties of the profiler without becoming enmeshed in the details of a particular implementation. Before we developed the formal model, we found alternative design choices almost impossible to understand and explain clearly, and the modified profiling scheme of Section 7.2 never occurred to us.

The formal cost semantics is deliberately slanted toward the programmer rather than the implementation. We also developed a small-step state transition semantics, which directly expresses the manipulation of cost centers performed by the

[14]LOLITA: Large-scale, object-based, linguistic interactor, translator and analyzer.

implementation, and proved it equivalent to the original cost semantics. This proof greatly increases our confidence in the correctness of our implementation.

Finally, there are several ways in which the work here could be developed:

—Lack of information about the enclosing cost center can hinder some of the `scc` transformations developed in Section 5.3. One possible solution to this is to dynamically bind the enclosing cost center to a first-class value which could then be used in subsequent `scc` annotations.

—As mentioned in Section 5, profiling occasionally disables optimizations that have a major effect on residency. When the user intends to profile the heap it is essential that such optimizations are not disabled. Though this requires further attention, we believe that developing the notion of the enclosing cost center (above) would go some way to ensuring this.

—In future we may deem it worthwhile to extend our profiler to implement *lexical scoping* using the higher-order boxing transformation described in Section 8.2.

—Apart from the subsumption of unprofiled costs, we have not implemented any form of inheritance, though it is straightforward in principle to do so (Section 8.3).

—We are also working on an extension to the profiler that supports the profiling of parallel programs.

APPENDIX

A. GLASGOW HASKELL

The Glasgow Haskell Compiler is freely available from a number of FTP sites. For more information please consult the GHC home page:

`http://www.dcs.glasgow.ac.uk/fp/software/ghc`

B. COST ATTRIBUTION AND EVALUATION ORDER

The original statement of Theorem 3.5.1 omitted details about renaming. The full theorem is stated and proved below for the modified semantics (Figure 4 with modifications in Figure 7). The proof is easily modified to show that the theorem also holds for the original semantics and the alternative evaluation scoping semantics (Section 8.1). First we need a couple of definitions.

Definition B.1 $\Delta' : e'$ *renames* $\Delta : e$ if there exists an injective mapping $R ::$ Variable $\rightarrow$ Variable such that

$$\forall x \in dom(\Delta) \cup FV(e), \quad \Delta'(R(x)) \equiv R(\Delta(x)) \text{ and } e' \equiv R(e)$$

where the occurrence of R on the right-hand side is the extension of the mapping R to a term substitution, and $\equiv$ denotes syntactic identity.

This just says that $\Delta' : e'$ is a copy of $\Delta : e$ (possibly with additional heap bindings), in which some heap-bound variables have been systematically replaced by new variables, in the expression e as well as in all expressions bound in Δ. $\Delta' : e'$ and $\Delta : e$ therefore have the same meaning, considered as computational results. This could be stated formally by proving that they map to the same meaning in a denotational semantics.

Definition B.2 The *size* of a derivation $|cc, \Gamma : e \Downarrow_\theta \Delta : z, cc_z|$ is the number of rule applications in the proof tree for $\Downarrow_\theta$. For convenience, we use the notation $|\Downarrow_\theta|$ where this unambiguously identifies a particular derivation.

THEOREM 3.5.1

If $cc_1, \Gamma : e_1 \Downarrow_{\theta_1} \Delta : z_1, cc_{z_1}$ and $cc_2, \Gamma : e_2 \Downarrow_{\theta_2} \Theta : z_2, cc_{z_2}$
Then there exists Θ', z_1', Δ', z_2' and Ω such that
$cc_1, \Theta' : e_1 \Downarrow_{\theta_1'} \Omega : z_1', cc_{z_1}$ and $cc_2, \Delta' : e_2 \Downarrow_{\theta_2'} \Omega : z_2', cc_{z_2}$
and $\theta_2 + \theta_1' = \theta_1 + \theta_2'$
where $\Delta' : z_1'$ renames $\Delta : z_1$ and $\Theta' : z_2'$ renames $\Theta : z_2$

PROOF. First we observe that there exist derivations $cc_1, \Gamma : e_1 \Downarrow_{\theta_1} \Delta' : z_1', cc_{z_1}$ and $cc_2, \Gamma : e_2 \Downarrow_{\theta_2} \Theta' : z_2', cc_{z_2}$ for which $(dom(\Delta') \setminus dom(\Gamma)) \cap (dom(\Theta') \setminus dom(\Gamma)) = \emptyset$ and for which $\Delta' : z_1'$ renames $\Delta : z_1$ and $\Theta' : z_2'$ renames $\Theta : z_2$.

This says that the newly allocated variables are distinct in the two derivations, but they give the same results.

The new derivations are constructed as copies of the original ones. First a renamed derivation $cc_1, \Gamma : e_1 \Downarrow^{\equiv}_{\theta_1} \Delta' : z_1', cc_{z_1}$ is constructed by choosing, in applications of the *Let* rule, fresh variables which occur in neither of the original derviations. Then a renamed derivation $cc_2, \Gamma : e_2 \Downarrow^{\equiv}_{\theta_2} \Theta' : z_2', cc_{z_2}$ is constructed by choosing, in applications of the *Let* rule, fresh variables which occur in neither of the original derivations nor in the newly constructed one for $cc_1, \Gamma : e_1 \Downarrow^{\equiv}_{\theta_1} \Delta' : z_1', cc_{z_1}$. This is possible because the supply of fresh variables is unlimited.

By construction, $\Delta' : z_1'$ renames $\Delta : z_1$, and $\Theta' : z_2'$ renames $\Theta : z_2$. Namely, the constructed derivations differ only in the choice of fresh variables in applications of the Let rule, and hence the results differ only in this respect also.

The proof now proceeds by induction on the total *size* of the renamed dervations. (For convenience we drop the $^{\equiv}$s and $'$s introduced by the renaming.)

Inductive Hypothesis:

If $cc_1, \Gamma : e_1 \Downarrow_{\theta_1} \Delta : z_1, cc_{z_1}$ and $cc_2, \Gamma : e_2 \Downarrow_{\theta_2} \Theta : z_2, cc_{z_2}$
and $(dom(\Delta) \setminus dom(\Gamma)) \cap (dom(\Theta) \setminus dom(\Gamma)) = \emptyset$
Then $cc_1, \Theta : e_1 \Downarrow_{\theta_1'} \Omega : z_1, cc_{z_1}$ and $cc_2, \Delta : e_2 \Downarrow_{\theta_2'} \Omega : z_2, cc_{z_2}$
and $\theta_2 + \theta_1' = \theta_1 + \theta_2'$ and $|\Downarrow_{\theta_1'}| \leq |\Downarrow_{\theta_1}|$ and $|\Downarrow_{\theta_2'}| \leq |\Downarrow_{\theta_2}|$.

The final two clauses establish that the sizes of the delayed derivations are no larger than the original derivations.

Assume the following: $cc_1, \Gamma : e_1 \Downarrow_{\theta_1} \Delta : z_1, cc_{z_1}$ and $cc_2, \Gamma : e_2 \Downarrow_{\theta_2} \Theta : z_2, cc_{z_2}$ and $(dom(\Delta) \setminus dom(\Gamma)) \cap (dom(\Theta) \setminus dom(\Gamma)) = \emptyset$. Consider the pair of rules used to derive $\Downarrow_{\theta_1}$ and $\Downarrow_{\theta_2}$.

Case: (*Lam*, *any*). The rule states that $cc_1, \Gamma : \lambda y.e \Downarrow_{\{\}} \Gamma : \lambda y.e, cc_1$. Now, $\Delta \equiv \Gamma$, $\theta_2 \equiv \theta_2'$, $\Theta \equiv \Omega$, $cc_1, \Theta : \lambda y.e \Downarrow_{\{\}} \Theta : \lambda y.e, cc_1$ (*Lam*), and $\theta_2 + \{\} = \{\} + \theta_2'$ and $|\Downarrow_{\theta_1'}| = 1 = |\Downarrow_{\theta_1}|$ and $|\Downarrow_{\theta_2'}| = |\Downarrow_{\theta_2}|$.

Case: (*Con*, *any*). The rule states that $cc_1, \Gamma : C\ \overline{x}_a \Downarrow_{\{\}} \Gamma : C\ \overline{x}_a, cc_1$. Now, $\Delta \equiv \Gamma$, $\theta_2 \equiv \theta_2'$, $\Theta \equiv \Omega$, $cc_1, \Theta : C\ \overline{x}_a \Downarrow_{\{\}} \Theta : C\ \overline{x}_a, cc_1$ (*Con*), and $\theta_2 + \{\} = \{\} + \theta_2'$ and $|\Downarrow_{\theta_1'}| = 1 = |\Downarrow_{\theta_1}|$ and $|\Downarrow_{\theta_2'}| = |\Downarrow_{\theta_2}|$.

Case: (*App*, *any*).

$$\frac{cc_1, \Gamma : e \Downarrow_{\theta_e} \Gamma_1 : \lambda y.e', cc_\lambda \qquad cc_\lambda, \Gamma_1 : e'[x/y] \Downarrow_{\theta_\lambda} \Delta : z_1, cc_{z_1}}{cc_1, \Gamma : e\ x \Downarrow_{\{cc_1 \mapsto \mathrm{A}\}+\theta_e+\theta_\lambda} \Delta : z_1, cc_{z_1}}$$

From the left premise ($\Downarrow_{\theta_e}$), the right assumption ($\Downarrow_{\theta_2}$), and the inductive hypothesis we have $cc_1, \Theta : e \Downarrow_{\theta'_e} \Omega_1 : \lambda y.e', cc_\lambda$ and $cc_2, \Gamma_1 : e_2 \Downarrow_{\theta'_2} \Omega_1 : z_2, cc_{z_2}$ and $\theta_2 + \theta'_e = \theta_e + \theta'_2$ and $|\Downarrow_{\theta'_e}| \leq |\Downarrow_{\theta_e}|$ and $|\Downarrow_{\theta'_2}| \leq |\Downarrow_{\theta_2}|$. Since $|\Downarrow_{\theta'_2}| \leq |\Downarrow_{\theta_2}|$, we can use the inductive hypothesis with the right premise ($\Downarrow_{\theta_\lambda}$) and $\Downarrow_{\theta'_2}$, giving $cc_\lambda, \Omega_1 : e'[x/y] \Downarrow_{\theta'_\lambda} \Omega_2 : z_1, cc_{z_1}$ and $cc_2, \Delta : e_2 \Downarrow_{\theta''_2} \Omega_2 : z_2, cc_{z_2}$ and $\theta'_2 + \theta'_\lambda = \theta_\lambda + \theta''_2$ and $|\Downarrow_{\theta'_\lambda}| \leq |\Downarrow_{\theta_\lambda}|$ and $|\Downarrow_{\theta''_2}| \leq |\Downarrow_{\theta'_2}|$. Applying the *App* rule we have $cc_1, \Theta : e\ x \Downarrow_{\{cc_1 \mapsto \mathrm{A}\}+\theta'_e+\theta'_\lambda} \Omega_2 : z_1, cc_{z_1}$ and $\theta_2 + \{cc_1 \mapsto \mathrm{A}\} + \theta'_e + \theta'_\lambda = \{cc_1 \mapsto \mathrm{A}\} + \theta_e + \theta'_2 + \theta'_\lambda = \{cc_1 \mapsto \mathrm{A}\} + \theta_e + \theta_\lambda + \theta''_2$ and $|\Downarrow_{\theta'_1}| = 1 + |\Downarrow_{\theta'_e}| + |\Downarrow_{\theta'_\lambda}| \leq 1 + |\Downarrow_{\theta_e}| + |\Downarrow_{\theta_\lambda}| = |\Downarrow_{\theta_1}|$.

Case: (*Let*, *any*).

$$\frac{cc_1, \Gamma[y_i \overset{cc_1}{\mapsto} \widehat{e_i}] : \widehat{e} \Downarrow_{\theta_e} \Delta : z_1, cc_{z_1}}{cc_1, \Gamma : \mathtt{let}\ \{x_i = e_i\}\ \mathtt{in}\ e \Downarrow_{\{cc_1 \mapsto n*\mathrm{H}\}+\theta_e} \Delta : z_1, cc_{z_1}}\ y_i \textbf{ fresh}$$

Since the fresh y_i in the renamed derivation are not bound in $\Downarrow_{\theta_2}$, we can construct an equivalent derivation, $\Downarrow^{\equiv}_{\theta_2}$, where the bindings for the y_i have been added to every heap in the derivation of $\Downarrow_{\theta_2}$. In all other respects $\Downarrow^{\equiv}_{\theta_2}$ is identical to $\Downarrow_{\theta_2}$.

$$cc_2, \Gamma[y_i \overset{cc_1}{\mapsto} \widehat{e_i}] : e_2 \Downarrow^{\equiv}_{\theta_2} \Theta[y_i \overset{cc_1}{\mapsto} \widehat{e_i}] : z_2, cc_{z_2}$$

From the premise ($\Downarrow_{\theta_e}$), the constructed derivation ($\Downarrow^{\equiv}_{\theta_2}$), and the inductive hypothesis we have $cc_1, \Theta[y_i \overset{cc_1}{\mapsto} \widehat{e_i}] : \widehat{e} \Downarrow_{\theta'_e} \Omega : z_1, cc_{z_1}$ and $cc_2, \Delta : e_2 \Downarrow_{\theta'_2} \Omega : z_2, cc_{z_2}$ and $\theta_2 + \theta'_e = \theta_e + \theta'_2$ and $|\Downarrow_{\theta'_e}| \leq |\Downarrow_{\theta_e}|$ and $|\Downarrow^{\equiv}_{\theta'_2}| \leq |\Downarrow^{\equiv}_{\theta_2}| = |\Downarrow_{\theta_2}|$. Applying the *Let* rule we have $cc_1, \Theta : \mathtt{let}\ \{x_i = e_i\}\ \mathtt{in}\ e \Downarrow_{\{cc_1 \mapsto n*\mathrm{H}\}+\theta'_e} \Omega : z_1, cc_{z_1}$ and $\theta_2 + \{cc_1 \mapsto n*\mathrm{H}\} + \theta'_e = \{cc_1 \mapsto n*\mathrm{H}\} + \theta_e + \theta'_2$ and $|\Downarrow_{\theta'_1}| = 1 + |\Downarrow_{\theta'_e}| \leq 1 + |\Downarrow_{\theta_e}| = |\Downarrow_{\theta_1}|$.

Case: (*Case*, *any*).

$$\frac{cc_1, \Gamma : e \Downarrow_{\theta_e} \Gamma_1 : C_k\ \overline{x}_{a_k}, cc_C \qquad cc_1, \Gamma_1 : e_k[x_i/y_{ki}] \Downarrow_{\theta_{e_k}} \Delta : z_1, cc_{z_1}}{cc_1, \Gamma : \mathtt{case}\ e\ \mathtt{of}\ alts \Downarrow_{\{cc_1 \mapsto \mathrm{C}\}+\theta_e+\theta_{e_k}} \Delta : z_1, cc_{z_1}}$$

From the left premise ($\Downarrow_{\theta_e}$), the right assumption ($\Downarrow_{\theta_2}$), and the inductive hypothesis we have $cc_1, \Theta : e \Downarrow_{\theta'_e} \Omega_1 : C_k\ \overline{x}_{a_k}, cc_C$ and $cc_2, \Gamma_1 : e_2 \Downarrow_{\theta'_2} \Omega_1 : z_2, cc_{z_2}$ and $\theta_2 + \theta'_e = \theta_e + \theta'_2$ and $|\Downarrow_{\theta'_e}| \leq |\Downarrow_{\theta_e}|$ and $|\Downarrow_{\theta'_2}| \leq |\Downarrow_{\theta_2}|$. Since $|\Downarrow_{\theta'_2}| \leq |\Downarrow_{\theta_2}|$, we can use the inductive hypothesis with the right premise ($\Downarrow_{\theta_{e_k}}$) and $\Downarrow_{\theta'_2}$, giving $cc_1, \Omega_1 : e_k[x_i/y_{ki}] \Downarrow_{\theta'_{e_k}} \Omega_2 : z_1, cc_{z_1}$ and $cc_2, \Delta : e_2 \Downarrow_{\theta''_2} \Omega_2 : z_2, cc_{z_2}$ and $\theta'_2 + \theta'_{e_k} = \theta_{e_k} + \theta''_2$ and $|\Downarrow_{\theta'_{e_k}}| \leq |\Downarrow_{\theta_{e_k}}|$ and $|\Downarrow_{\theta''_2}| \leq |\Downarrow_{\theta'_2}|$. Applying the *Case* rule we have $cc_1, \Theta : \mathtt{case}\ e\ \mathtt{of}\ alts \Downarrow_{\{cc_1 \mapsto \mathrm{C}\}+\theta'_e+\theta'_{e_k}} \Omega_2 : z_1, cc_{z_1}$ and $\theta_2 + \{cc_1 \mapsto \mathrm{C}\} + \theta'_e + \theta'_{e_k} = \{cc_1 \mapsto \mathrm{C}\} + \theta_e + \theta'_2 + \theta'_{e_k} = \{cc_1 \mapsto \mathrm{C}\} + \theta_e + \theta_{e_k} + \theta''_2$ and $|\Downarrow_{\theta'_1}| = 1 + |\Downarrow_{\theta'_e}| + |\Downarrow_{\theta'_{e_k}}| \leq 1 + |\Downarrow_{\theta_e}| + |\Downarrow_{\theta_{e_k}}| = |\Downarrow_{\theta_1}|$.

Case: (*SCC*, *any*). $$\frac{cc, \Gamma : e \Downarrow_{\theta_e} \Delta : z_1, cc_{z_1}}{cc_1, \Gamma : \mathtt{scc}\ cc\ e \Downarrow_{\{cc \mapsto \mathrm{E}\}+\theta_e} \Delta : z_1, cc_{z_1}}$$

From the premise ($\Downarrow_{\theta_e}$), the right assumption ($\Downarrow_{\theta_2}$), and the inductive hypothesis we have $cc, \Theta : e \Downarrow_{\theta'_e} \Omega : z_1, cc_{z_1}$ and $cc_2, \Delta : e_2 \Downarrow_{\theta'_2} \Omega : z_2, cc_{z_2}$ and $\theta_2 + \theta'_e =$

$\theta_e + \theta'_2$ and $|\Downarrow_{\theta'_e}| \leq |\Downarrow_{\theta_e}|$ and $|\Downarrow_{\theta'_2}| \leq |\Downarrow_{\theta_2}|$. Applying the *SCC* rule we have $cc_1, \Theta : \mathtt{scc}\ cc\ e \Downarrow_{\{cc \mapsto \mathrm{E}\}+\theta'_e} \Omega : z_1, cc_{z_1}$ and $\theta_2 + \{cc \mapsto \mathrm{E}\} + \theta'_e = \{cc \mapsto \mathrm{E}\} + \theta_e + \theta'_2$ and $|\Downarrow_{\theta'_1}| = 1 + |\Downarrow_{\theta'_e}| \leq 1 + |\Downarrow_{\theta_e}| = |\Downarrow_{\theta_1}|$.

Case: (*Var*(*whnf*), *any*). $cc_1, \Gamma'[x \overset{cc_z}{\mapsto} z] : x \Downarrow_{\{cc_1 \mapsto \mathrm{V}\}} \Gamma'[x \overset{cc_z}{\mapsto} z] : z, \mathrm{S}(\overset{cc_1}{cc_z}\, z)$. Now, $\Delta \equiv \Gamma \equiv \Gamma'[x \overset{cc_z}{\mapsto} z]$, $\theta_2 \equiv \theta'_2$, and $\Theta \equiv \Omega \equiv \Theta'[x \overset{cc_z}{\mapsto} z]$, since $\Downarrow_{\theta_2}$ must preserve the value binding for x. So we have $cc_1, \Theta'[x \overset{cc_z}{\mapsto} z] : x \Downarrow_{\{cc_1 \mapsto \mathrm{V}\}} \Theta'[x \overset{cc_z}{\mapsto} z] : z, \mathrm{S}(\overset{cc_1}{cc_z}\, z)$ by *Var*(*whnf*) and $\theta_2 + \{cc_1 \mapsto \mathrm{V}\} = \{cc_1 \mapsto \mathrm{V}\} + \theta'_2$ and $|\Downarrow_{\theta'_1}| = 1 = |\Downarrow_{\theta_1}|$ and $|\Downarrow_{\theta'_2}| = |\Downarrow_{\theta_2}|$.

Case: (*any*, *Lam*), (*any*, *App*), (*any*, *Let*), (*any*, *Con*), (*any*, *Case*) (*any*, *SCC*), and (*any*, *Var*(*whnf*)). We omit these cases, since they are essentially identical to the cases already given for $\Downarrow_{\theta_1}$.

Case: (*Var*(*thunk*), *Var*(*thunk*)). Either $e_1 \equiv x \equiv e_2$ or $e_1 \equiv x \not\equiv y \equiv e_2$.

Suppose $e_1 \equiv x \equiv e_2$, i.e.,

$$\frac{cc_x, \Gamma' : e_x \Downarrow_{\theta_{e_{x1}}} \Delta' : z_1, cc_{z_1}}{cc_1, \Gamma'[x \overset{cc_x}{\mapsto} e_x] : x \Downarrow_{\substack{\{cc_1 \mapsto \mathrm{V}\} + \theta_{e_{x1}} \\ +\{cc_{z_1} \mapsto \mathrm{U}\}}} \Delta'[x \overset{cc_{z_1}}{\mapsto} z_1] : z_1, \mathrm{S}(\overset{cc_1}{cc_{z_1}}\, z_1)} \; e_x \not\equiv z_1$$

$$\frac{cc_x, \Gamma' : e_x \Downarrow_{\theta_{e_{x2}}} \Theta' : z_2, cc_{z_2}}{cc_2, \Gamma'[x \overset{cc_x}{\mapsto} e_x] : x \Downarrow_{\substack{\{cc_2 \mapsto \mathrm{V}\} + \theta_{e_{x2}} \\ +\{cc_{z_2} \mapsto \mathrm{U}\}}} \Theta'[x \overset{cc_{z_2}}{\mapsto} z_2] : z_2, \mathrm{S}(\overset{cc_2}{cc_{z_2}}\, z_2)} \; e_x \not\equiv z_2$$

So $\theta_{e_{x1}} \equiv \theta_{e_{x2}}$, $\Delta' \equiv \Theta' \equiv \Omega'$, $z_1 \equiv z_2 \equiv z$, and $cc_{z_1} \equiv cc_{z_2} \equiv cc_z$. Now, $cc_1, \Theta'[x \overset{cc_z}{\mapsto} z] : x \Downarrow_{\{cc_1 \mapsto \mathrm{V}\}} \Omega'[x \overset{cc_z}{\mapsto} z] : z_1, \mathrm{S}(\overset{cc_1}{cc_{z_1}}\, z_1)$ by *Var*(*whnf*) and $cc_2, \Delta'[x \overset{cc_z}{\mapsto} z] : x \Downarrow_{\{cc_2 \mapsto \mathrm{V}\}} \Omega'[x \overset{cc_z}{\mapsto} z] : z_2, \mathrm{S}(\overset{cc_2}{cc_{z_2}}\, z_2)$ by *Var*(*whnf*) and $\{cc_2 \mapsto \mathrm{V}\} + \theta_{e_{x2}} + \{cc_{z_2} \mapsto \mathrm{U}\} + \{cc_1 \mapsto \mathrm{V}\} = \{cc_1 \mapsto \mathrm{V}\} + \theta_{e_{x1}} + \{cc_{z_1} \mapsto \mathrm{U}\} + \{cc_2 \mapsto \mathrm{V}\}$ and $|\Downarrow_{\theta'_1}| = 1 \leq 1 + |\Downarrow_{\theta_{e_{x1}}}| = |\Downarrow_{\theta_1}|$ and $|\Downarrow_{\theta'_2}| = 1 \leq 1 + |\Downarrow_{\theta_{e_{x2}}}| = |\Downarrow_{\theta_2}|$.

Suppose $e_1 \equiv x \not\equiv y \equiv e_2$, i.e.,

$$\frac{cc_x, \Gamma'[y \overset{cc_y}{\mapsto} e_y] : e_x \Downarrow_{\theta_{e_x}} \Delta' : z_1, cc_{z_1}}{cc_1, \Gamma'[y \overset{cc_y}{\mapsto} e_y, x \overset{cc_x}{\mapsto} e_x] : x \Downarrow_{\substack{\{cc_1 \mapsto \mathrm{V}\} + \theta_{e_x} \\ +\{cc_{z_1} \mapsto \mathrm{U}\}}} \Delta'[x \overset{cc_{z_1}}{\mapsto} z_1] : z_1, \mathrm{S}(\overset{cc_1}{cc_{z_1}}\, z_1)} \; e_x \not\equiv z_1$$

$$\frac{cc_y, \Gamma'[x \overset{cc_x}{\mapsto} e_x] : e_y \Downarrow_{\theta_{e_y}} \Theta' : z_2, cc_{z_2}}{cc_2, \Gamma'[y \overset{cc_y}{\mapsto} e_y, x \overset{cc_x}{\mapsto} e_x] : y \Downarrow_{\substack{\{cc_2 \mapsto \mathrm{V}\} + \theta_{e_y} \\ +\{cc_{z_2} \mapsto \mathrm{U}\}}} \Theta'[y \overset{cc_{z_2}}{\mapsto} z_2] : z_2, \mathrm{S}(\overset{cc_2}{cc_{z_2}}\, z_2)} \; e_y \not\equiv z_2$$

If the evaluation of x requires the value of y, and the evaluation of y requires the evaluation of x, we have a cyclic data dependency, and the assumption cannot hold.

Without loss of generality, assume the evaluation of y does not require the value of x, i.e., the final heap of $\Downarrow_{\theta_2}$ has an unevaluated binding for x ($\Theta' \equiv \Theta''[x \overset{cc_x}{\mapsto} e_x]$).

Since the binding for x is not required in the derivation of $\Downarrow_{\theta_2}$ we can construct an equivalent derivation, $\Downarrow^{\equiv}_{\theta_2}$, where the binding for x has been dropped from every heap in the derivation of $\Downarrow_{\theta_2}$. In all other respects $\Downarrow^{\equiv}_{\theta_2}$ is identical to $\Downarrow_{\theta_2}$.

$$cc_2, \Gamma'[y \overset{cc_y}{\mapsto} e_y] : y \Downarrow^{\equiv}_{\theta_2} \Theta''[y \overset{cc_{z_2}}{\mapsto} z_2] : z_2, \mathrm{S}(\overset{cc_2}{cc_{z_2}}\, z_2)$$

From the premise ($\Downarrow_{\theta e_x}$), the constructed derivation ($\Downarrow^{\equiv}_{\theta_2}$), and the inductive hypothesis we have $cc_x, \Theta''[y \overset{cc_{z_2}}{\mapsto} z_2] : e_x \Downarrow_{\theta'_{e_x}} \Omega' : z_1, cc_{z_1}$ and $cc_2, \Delta' : y \Downarrow_{\theta'_2} \Omega' : z_2, \mathrm{S}(^{cc_2}_{cc_{z_2}}\, z_2)$ and $\theta_2 + \theta'_{e_x} = \theta_{e_x} + \theta'_2$ and $|\Downarrow_{\theta'_{e_x}}| \leq |\Downarrow_{\theta e_x}|$ and $|\Downarrow_{\theta'_2}| \leq |\Downarrow^{\equiv}_{\theta_2}|$. Applying the *Var*(*thunk*) rule we have

$$cc_1, \Theta''[y \overset{cc_{z_2}}{\mapsto} z_2, x \overset{cc_x}{\mapsto} e_x] : x \Downarrow_{\substack{\{cc_1 \mapsto \mathrm{V}\} + \theta'_{e_x} \\ +\{cc_{z_1} \mapsto \mathrm{U}\}}} \Omega'[x \overset{cc_{z_1}}{\mapsto} z_1] : z_1, \mathrm{S}(^{cc_1}_{cc_{z_1}}\, z_1).$$

Since x is not bound in Δ' and since the derivation of $\Downarrow_{\theta'_2}$ does not choose x as a fresh variable (the choice of fresh variables is preserved from $\Downarrow_{\theta_2}$), we can construct an equivalent derivation, $\Downarrow^{\equiv}_{\theta'_2}$, where the binding for x has been added to every heap in the derivarion of $\Downarrow_{\theta'_2}$. In all other respects $\Downarrow^{\equiv}_{\theta'_2}$ is identical to $\Downarrow_{\theta'_2}$.

$$cc_2, \Delta'[x \overset{cc_{z_1}}{\mapsto} z_1] : y \Downarrow^{\equiv}_{\theta'_2} \Omega'[x \overset{cc_{z_1}}{\mapsto} z_1] : z_2, \mathrm{S}(^{cc_2}_{cc_{z_2}}\, z_2).$$

Finally, we have $\theta_2 + \{cc_1 \mapsto \mathrm{V}\} + \theta'_{e_x} + \{cc_{z_1} \mapsto \mathrm{U}\} = \{cc_1 \mapsto \mathrm{V}\} + \theta_{e_x} + \{cc_{z_1} \mapsto \mathrm{U}\} + \theta'_2$ and $|\Downarrow_{\theta'_1}| = 1 + |\Downarrow_{\theta'_{e_x}}| \leq 1 + |\Downarrow_{\theta_{e_x}}| = |\Downarrow_{\theta_1}|$ and $|\Downarrow_{\theta'_2}| = |\Downarrow^{\equiv}_{\theta'_2}| \leq |\Downarrow_{\theta_2}| = |\Downarrow^{\equiv}_{\theta_2}|$. □

C. CORRECTNESS OF THE OPERATIONAL SEMANTICS

THEOREM 4.2.1

$(cc, \Gamma_{init}, e, [], \{\}) \Rightarrow^* (cc', \Delta, z, [], \theta)$ if and only if $cc, \Gamma_{init} : e \Downarrow_\theta \Delta : z, cc'$

The proof below establishes the theorem for the modified semantics and state transition rules (Figures 4 and 5 with modifications in Figure 7). Its structure is similar to that of Sestoft's correctness proof for the unprofiled state transition system [Sestoft 1997]. First we prove the following lemmas.

LEMMA C.1

If $cc, \Gamma : e \Downarrow_\theta \Delta : e', cc'$
Then $(cc, \Gamma, e, S, \theta_0) \Rightarrow^* (cc', \Delta, e', S, \theta_1)$ and $\theta_1 = \theta_0 + \theta$.

PROOF. This is proved by induction on the derivation of $cc, \Gamma : e \Downarrow_\theta \Delta : e', cc'$.

Case: *Lam*. $cc, \Gamma : \lambda y.e \Downarrow_{\{\}} \Gamma : \lambda y.e, cc$. Clearly

$(cc, \Gamma, \lambda y.e, S, \theta_0) \Rightarrow^* (cc, \Gamma, \lambda y.e, S, \theta_0 + \{\})$ empty sequence

Case: *Con*. $cc, \Gamma : C\ \overline{x}_a \Downarrow_{\{\}} \Gamma : C\ \overline{x}_a, cc$. Clearly

$(cc, \Gamma, C\ \overline{x}_a, S, \theta_0) \Rightarrow^* (cc, \Gamma, C\ \overline{x}_a, S, \theta_0 + \{\})$ empty sequence

Case: *App*. $cc, \Gamma : e\ x \Downarrow_{\{cc \mapsto \mathrm{A}\} + \theta_1 + \theta_2} \Theta : z, cc_z$. Now

	$(cc, \Gamma, (e\ x), S, \theta_0)$	
$\Rightarrow$	$(cc, \Gamma, e, x{:}S, \theta_0 + \{cc \mapsto \mathrm{A}\})$	rule app_1
$\Rightarrow^*$	$(cc_\lambda, \Delta, \lambda y.e', x{:}S, \theta_0 + \{cc \mapsto \mathrm{A}\} + \theta_1)$	left premise and hyp.
$\Rightarrow$	$(cc_\lambda, \Delta, e'[x/y], S, \theta_0 + \{cc \mapsto \mathrm{A}\} + \theta_1)$	rule app_2
$\Rightarrow^*$	$(cc_z, \Theta, z, S, \theta_0 + \{cc \mapsto \mathrm{A}\} + \theta_1 + \theta_2)$	right premise and hyp.

Case: *Var*(*whnf*). $cc, \Gamma[x \overset{cc_z}{\mapsto} z] : x \Downarrow_{\{cc \mapsto \mathrm{V}\}} \Gamma[x \overset{cc_z}{\mapsto} z] : z, \mathrm{S}(^{cc}_{cc_z}\, z)$. Clearly

$$
\begin{array}{lll}
 & (cc, \Gamma[x \overset{cc_x}{\mapsto} z], x, S, \theta_0) & \\
\Rightarrow & (\mathrm{S}(^{cc}_{cc_z}\, z), \Gamma[x \overset{cc_x}{\mapsto} z], z, S, \theta_0 + \{cc \mapsto \mathrm{V}\}) & \text{rule } var_1
\end{array}
$$

Case: $Var(thunk)$. $cc, \Gamma[x \overset{cc_e}{\mapsto} e] : x \Downarrow_{\{cc \mapsto \mathrm{V}\}+\theta+\{cc_z \mapsto \mathrm{U}\}} \Delta[x \overset{cc_x}{\mapsto} z] : z, \mathrm{S}(^{cc}_{cc_z}\, z)$.

$$
\begin{array}{lll}
 & (cc, \Gamma[x \overset{cc_e}{\mapsto} e], x, S, \theta_0) & \\
\Rightarrow & (cc_e, \Gamma, e, (^{\#x}_{cc}) : S, \theta_0 + \{cc \mapsto \mathrm{V}\}) & \text{rule } var_2 \\
\Rightarrow^* & (cc_z, \Delta, z, (^{\#x}_{cc}) : S, \theta_0 + \{cc \mapsto \mathrm{V}\} + \theta) & \text{premise and hyp.} \\
\Rightarrow & (\mathrm{S}(^{cc}_{cc_z}\, z), \Delta[x \overset{cc_x}{\mapsto} x], z, S, \theta_0 + \{cc \mapsto \mathrm{V}\} + \theta + \{cc_z \mapsto \mathrm{U}\}) & \text{rule } var_3
\end{array}
$$

Case: *Let*. $cc, \Gamma : \texttt{let}\ \{x_i = e_i\}\ \texttt{in}\ e \Downarrow_{\{cc \mapsto n*\mathrm{H}\}+\theta} \Delta : z, cc_z$. Since the *let* rule can choose the same fresh y_i as were chosen by the *Let* rule we have

$$
\begin{array}{lll}
 & (cc, \Gamma, \texttt{let}\ \{x_i = e_i\}\ \texttt{in}\ e, S, \theta_0) & \\
\Rightarrow & (cc, \Gamma[y_i \overset{cc}{\mapsto} \widehat{e}_i], \widehat{e}, S, \theta_0 + \{cc \mapsto n*\mathrm{H}\}) & \text{rule } let \text{ (same } y_i \text{ as } Let) \\
\Rightarrow^* & (cc_z, \Delta, z, S, \theta_0 + \{cc \mapsto n*\mathrm{H}\} + \theta) & \text{premise and hyp.}
\end{array}
$$

Case: *Case*. $cc, \Gamma : \texttt{case}\ e\ \texttt{of}\ \{C_j\ \overline{y_{ja_j}}\ \texttt{->}\ e_j\} \Downarrow_{\{cc \mapsto \mathrm{C}\}+\theta_1+\theta_2} \Theta : z, cc_z$. Now

$$
\begin{array}{lll}
 & (cc, \Gamma, \texttt{case}\ e\ \texttt{of}\ alts, S, \theta_0) & \\
\Rightarrow & (cc, \Gamma, e, (alts, cc) : S, \theta_0 + \{cc \mapsto \mathrm{C}\}) & \text{rule } case_1 \\
\Rightarrow^* & (cc_C, \Delta, C_k\ \overline{x}_{a_k}, (alts, cc) : S, \theta_0 + \{cc \mapsto \mathrm{C}\} + \theta_1) & \text{left premise and hyp.} \\
\Rightarrow & (cc, \Delta, e_k[x_i/y_{ki}], S, \theta_0 + \{cc \mapsto \mathrm{C}\} + \theta_1) & \text{rule } case_2 \\
\Rightarrow^* & (cc_z, \Theta, z, S, \theta_0 + \{cc \mapsto \mathrm{C}\} + \theta_1 + \theta_2) & \text{right premise and hyp.}
\end{array}
$$

Case: *SCC*. $cc, \Gamma : \texttt{scc}\ cc_{scc}\ e \Downarrow_{\{cc_{scc} \mapsto \mathrm{E}\}+\theta} \Delta : z, cc_z$. Now

$$
\begin{array}{lll}
 & (cc, \Gamma, \texttt{scc}\ cc_{scc}\ e, S, \theta_0) & \\
\Rightarrow & (cc_{scc}, \Gamma, e, S, \theta_0 + \{cc_{scc} \mapsto \mathrm{E}\}) & \text{rule } scc \\
\Rightarrow^* & (cc_z, \Delta, z, S, \theta_0 + \{cc_{scc} \mapsto \mathrm{E}\} + \theta) & \text{premise and hyp.} \quad \square
\end{array}
$$

Definition C.2 A *balanced computation* is a computation $(cc, \Gamma, e, S, \theta) \Rightarrow^* (cc', \Gamma', e', S, \theta')$ in which the initial and final stacks are the same and in which all intermediate stacks are extensions of the initial stack.

Since the stack represents the context of a subderivation, a balanced computation captures the idea that the proof of each subderivation does not depend on the context in which that subderivation occurs, i.e., it does not consume any of its context.

Definition C.3 The *trace* of a computation is the sequence of transition rules used. A *balanced trace* is the trace of a balanced computation.

What are the possible forms of a balanced trace? Clearly the empty trace (having one state and no transitions) is balanced. Now consider nonempty traces, and assume an initial stack S. A nonempty balanced trace must begin with an app_1, var_1, var_2, *let*, $case_1$, or *scc* transition (since app_2, var_3, and $case_2$ would produce an intermediate state which is not an extension of S). We consider each initial transition in turn.

If the trace begins with app_1, which produces an intermediate stack of the form $x:S$, then eventually an app_2 transition must occur which restores the stack to S (no other transition can do this). Consider the first such transition; the subtrace between the initial app_1 transition and the occurrence must be balanced (with stacks which are extensions of $x:S$), and the subtrace after that occurrence is balanced (with stacks which are extensions of S). The trace must have the form $app_1\ bal\ app_2\ bal$ where bal stands for an arbitrary balanced trace.

A trace that begins with var_1 can only contain this single transition, since it produces an intermediate stack S and control z, which must be a λ-abstraction or a constructor. The only transitions which could follow are app_2, $case_2$, or var_3, all of which remove an element from the stack and contradict the balancedness of the trace.

A trace that begins with var_2 produces an intermediate stack with the form $\binom{\sharp x}{cc} : S$ which must be restored by a var_3 transition. As for the var_1 trace, the control after the var_3 transition must be a λ-abstraction or a constructor. Hence the occurrence of the var_3 transition must be the last element of the trace, which must have the form $var_2\ bal\ var_3$.

If the trace begins with let, then the subtrace after let must be balanced, so it has the form $let\ bal$.

A trace that begins with $case_1$ produces an intermediate stack with the form $(alts,cc):S$ which must be restored by a $case_2$ transition. Such a trace must have the form $case_1\ bal\ case_2\ bal$.

If the trace begins with scc, then the subtrace after scc must be balanced, so it has the form $scc\ bal$.

In summary, all balanced traces can be derived from the following grammar:

$$\begin{array}{rcl} bal & ::= & \epsilon \mid app_1\ bal\ app_2\ bal \mid var_1 \mid var_2\ bal\ var_3 \\ & & \mid let\ bal \mid case_1\ bal\ case_2\ bal \mid scc\ bal \end{array}$$

LEMMA C.4

If $(cc_0, \Gamma_0, e_0, S, \theta_0) \Rightarrow^* (cc_1, \Gamma_1, z, S, \theta_1)$ is a *balanced computation*
Then $cc_0, \Gamma_0 : e_0 \Downarrow_\theta \Gamma_1 : z, cc_1$ and $\theta_1 = \theta_0 + \theta$.

PROOF. This is proved by induction on the structure of balanced traces, following the grammar.

Case: ϵ. If $z \equiv \lambda y.e$ then it follows by rule *Lam* because we have $cc_0 = cc_1$, $\Gamma_0 = \Gamma_1$, and $e_0 \equiv z \equiv \lambda y.e$. Otherwise $z \equiv C\ \overline{x}_a$, and it follows by rule *Con*. In both cases $\theta = \{\}$, so $\theta_1 = \theta_0 + \theta$ since $\theta_0 = \theta_1$.

Case: $app_1\ bal\ app_2\ bal$. We must have $e_0 \equiv e\ x$. The state after app_1 must be $(cc_0, \Gamma_0, e, x:S, \theta_0 + \{cc_0 \mapsto \text{A}\})$, and the state before app_2 must be $(cc_\lambda, \Delta, \lambda y.e', x:S, \theta')$. Since the trace between these states is balanced we have $cc_0, \Gamma_0 : e \Downarrow_{\theta_{bal_1}} \Delta : \lambda y.e', cc_\lambda$ where $\theta' = \theta_0 + \{cc_0 \mapsto \text{A}\} + \theta_{bal_1}$ by the inductive hypothesis. The state after app_2 is $(cc_\lambda, \Delta, e'[x/y], S, \theta')$, and the trace of $(cc_\lambda, \Delta, e'[x/y], S, \theta') \Rightarrow^* (cc_1, \Gamma_1, z, S, \theta_1)$ is balanced; so we have $cc_\lambda, \Delta : e'[x/y] \Downarrow_{\theta_{bal_2}} \Gamma_1 : z, cc_1$ where $\theta_1 = \theta' + \theta_{bal_2}$ by the inductive hypothesis. Using the *App* rule we conclude that $cc_0, \Gamma_0 : e\ x \Downarrow_{\{cc_0 \mapsto \text{A}\} + \theta_{bal_1} + \theta_{bal_2}} \Gamma_1 : z, cc_1$ and observe that $\theta_1 = \theta' + \theta_{bal_2} = \theta_0 + \{cc_0 \mapsto \text{A}\} + \theta_{bal_1} + \theta_{bal_2}$ as required.

Case: var_1. We must have $e_0 \equiv x$ and $\Gamma_0 = \Gamma[x \overset{cc_z}{\mapsto} z]$. The state after var_1 must be $(\mathrm{S}\binom{cc_0}{cc_z} z), \Gamma[x \overset{cc_z}{\mapsto} z], z, S, \theta_0 + \{cc_0 \mapsto \mathrm{V}\})$. But the $Var(whnf)$ rule states that $cc_0, \Gamma[x \overset{cc_z}{\mapsto} z] : x \Downarrow_{\{cc \mapsto \mathrm{V}\}} \Gamma[x \overset{cc_z}{\mapsto} z] : z, \mathrm{S}\binom{cc_0}{cc_z} z)$ with $cc_1 = \mathrm{S}\binom{cc_0}{cc_z} z)$ and $\theta_1 = \theta_0 + \{cc_0 \mapsto \mathrm{V}\}$ as required.

Case: var_2 *bal* var_3. We must have $e_0 \equiv x$, $\Gamma_0 = \Gamma[x \overset{cc_e}{\mapsto} e]$, and $e \not\equiv z$. The state after var_2 must be $(cc_e, \Gamma, e, \binom{\#x}{cc_0} : S, \theta_0 + \{cc_0 \mapsto \mathrm{V}\})$; the state before var_3 must be $(cc_z, \Delta, z, \binom{\#x}{cc_0} : S, \theta')$; and the state after var_3 must be $(\mathrm{S}\binom{cc_0}{cc_z} z), \Delta[x \overset{cc_z}{\mapsto} z], z, S, \theta' + \{cc_z \mapsto \mathrm{U}\})$. Since the trace between var_2 and var_3 is balanced we have $cc_e, \Gamma : e \Downarrow_{\theta_{bal_1}} \Delta : z, cc_z$ where $\theta' = \theta_0 + \{cc_0 \mapsto \mathrm{V}\} + \theta_{bal_1}$ by the inductive hypothesis. Using the $Var(thunk)$ rule we conclude that $cc_0, \Gamma[x \overset{cc_e}{\mapsto} e] : x \Downarrow_{\{cc_0 \mapsto \mathrm{V}\} + \theta_{bal_1} + \{cc_z \mapsto \mathrm{U}\}} \Delta[x \overset{cc_z}{\mapsto} z] : z, \mathrm{S}\binom{cc_0}{cc_z} z)$ and observe that $\theta_1 = \theta' + \{cc_z \mapsto \mathrm{U}\} = \theta_0 + \{cc_0 \mapsto \mathrm{V}\} + \theta_{bal_1} + \{cc_z \mapsto \mathrm{U}\}$ as required.

Case: *let bal*. We must have $e_0 \equiv$ `let` $\{x_i$ `=` $e_i\}$ `in` e, and for the renaming $\widehat{e}$ some fresh y_i have been chosen. The state after *let* is $(cc_0, \Gamma_0[y_i \overset{cc_0}{\mapsto} \widehat{e_i}], \widehat{e}, S, \theta_0 + \{cc_0 \mapsto n * \mathrm{H}\})$, and since the trace after *let* is balanced we have $cc_0, \Gamma_0[y_i \overset{cc_0}{\mapsto} \widehat{e_i}] : \widehat{e} \Downarrow_{\theta_{bal_1}} \Gamma_1 : z, cc_1$ where $\theta_1 = \theta_0 + \{cc_0 \mapsto n * \mathrm{H}\} + \theta_{bal_1}$ by the inductive hypothesis. Using the *Let* rule, choosing the same fresh y_i, we conclude that $cc_0, \Gamma_0 :$ `let` $\{x_i$ `=` $e_i\}$ `in` $e \Downarrow_{\{cc_0 \mapsto n * \mathrm{H}\} + \theta_{bal_1}} \Gamma_1 : z, cc_1$ as required.

Case: $case_1$ *bal* $case_2$ *bal*. We must have $e_0 \equiv$ `case` e `of` $alts$. The state after $case_1$ must be $(cc_0, \Gamma_0, e, (alts, cc_0) : S, \theta_0 + \{cc_0 \mapsto \mathrm{C}\})$, and the state before $case_2$ must be $(cc_C, \Delta, C_k\ \overline{x}_{a_k}, (alts, cc_0) : S, \theta')$. Since the trace between these states is balanced we have $cc_0, \Gamma_0 : e \Downarrow_{\theta_{bal_1}} \Delta : C_k\ \overline{x}_{a_k}, cc_C$ where $\theta' = \theta_0 + \{cc_0 \mapsto \mathrm{C}\} + \theta_{bal_1}$ by the inductive hypothesis. The state after $case_2$ is $(cc_0, \Delta, e_k[x_i/y_{ki}], S, \theta')$, and the trace of $(cc_0, \Delta, e_k[x_i/y_{ki}], S, \theta') \Rightarrow^* (cc_1, \Gamma_1, z, S, \theta_1)$ is balanced; so we have $cc_0, \Delta : e_k[x_i/y_{ki}] \Downarrow_{\theta_{bal_2}} \Gamma_1 : z, cc_1$ where $\theta_1 = \theta' + \theta_{bal_2}$ by the inductive hypothesis. Using the *Case* rule we conclude that $cc_0, \Gamma_0 :$ `case` e `of` $alts \Downarrow_{\{cc_0 \mapsto \mathrm{C}\} + \theta_{bal_1} + \theta_{bal_2}}$ $\Gamma_1 : z, cc_1$ and observe that $\theta_1 = \theta' + \theta_{bal_2} = \theta_0 + \{cc_0 \mapsto \mathrm{C}\} + \theta_{bal_1} + \theta_{bal_2}$ as required.

Case: *scc bal*. We must have $e_0 \equiv$ `scc` cc_{scc} e. The state after *scc* is $(cc_{scc}, \Gamma_0, e, S, \theta_0 + \{cc_{scc} \mapsto \mathrm{E}\})$, and since the trace after *scc* is balanced we have $cc_{scc}, \Gamma_0 : e \Downarrow_{\theta_{bal_1}} \Gamma_1 : z, cc_1$ where $\theta_1 = \theta_0 + \{cc_{scc} \mapsto \mathrm{E}\} + \theta_{bal_1}$ by the inductive hypothesis. Using the *SCC* rule, we conclude that $cc_0, \Gamma_0 :$ `scc` cc_{scc} $e \Downarrow_{\{cc_{scc} \mapsto \mathrm{E}\} + \theta_{bal_1}} \Gamma_1 : z, cc_1$ as required. □

THEOREM 4.2.1

$(cc, \Gamma_{init}, e, [\,], \{\}) \Rightarrow^* (cc', \Delta, z, [\,], \theta)$ if and only if $cc, \Gamma_{init} : e \Downarrow_\theta \Delta : z, cc'$

PROOF.

If. This part follows from Lemma C.1.

Only if. Observe that the computation $(cc, \Gamma_{init}, e, [\,], \{\}) \Rightarrow^* (cc', \Delta, z, [\,], \theta)$ is necessarily balanced; then the result follows from Lemma C.4. □

The theorem says that the abstract machine terminates with a value z and cost attribution θ if and only if the natural semantics successfully derives this value and

cost attribution.

A second possibility is that the machine may terminate with a variable x in the control, indicating a "black hole," in which case the natural semantics permits no derivations at all. Finally, a third possibility is that the machine may embark on an infinite computation, corresponding to an infinite derivation tree in the natural semantics. Since both systems are deterministic (modulo the choice of fresh variables) these possibilities are mutually exclusive.

D. THE STATIC SCOPE OF COST CENTERS

Section 4.3 introduced Theorem 4.3.5 for the original semantics. Here the theorem is restated and proved for the modified semantics (Figure 5 with modifications in Figure 7). This requires a couple of extensions:

(1) The right-hand sides of all top-level bindings, including the CAF bindings introduced in Section 7, are labeled with the cost center annotating the heap binding.

$$\Gamma^{lab}_{init} \;=\; \{\; x \overset{cc}{\mapsto} \varphi^{cc}[e] :\;\; x \texttt{ = } e \in \mathit{prog} \text{ and } cc = \mathbf{initcc}(e) \;\}$$

(2) The modified var_1 and var_3 rules (Figure 7) are extended to relabel `"SUB"` and `"CAF"` values with the subsuming cost center. Note that a value may be relabeled more than once, since it may be subsumed by a `"CAF"` cost center and then re-subsumed by another cost center.

$$(cc, \Gamma[x \overset{cc_z}{\mapsto} z], x^l, S, \theta) \;\Rightarrow^{var_1}_{lab}\; (\mathrm{S}(^{cc}_{cc_z}\, z), \Gamma[x \overset{cc_z}{\mapsto} z], \mathbf{labsub}(cc_z, cc, z), S, \theta + \{cc \mapsto \mathrm{V}\})$$

$$(cc_z, \Gamma, z, (^{\#x}_{cc}) : S, \theta) \;\Rightarrow^{var_3}_{lab}\; (\mathrm{S}(^{cc}_{cc_z}\, z), \Gamma[x \overset{cc_z}{\mapsto} z], \mathbf{labsub}(cc_z, cc, z), S, \theta + \{cc_z \mapsto \mathrm{U}\})$$

where
$$\begin{aligned} \mathbf{labsub}(\texttt{"SUB"}, cc, \lambda y.e^{\,l}) &= \varphi^{cc}[U[\lambda y.e^{\,l}]] \\ \mathbf{labsub}(\texttt{"CAF"}, cc, \lambda y.e^{\,l}) &= \varphi^{cc}[U[\lambda y.e^{\,l}]] \\ \mathbf{labsub}(cc_z \quad, cc, z \quad) &= z \end{aligned}$$

THEOREM 4.3.5

$(cc_{init}, \Gamma^{lab}_{init}, e^{lab}_{init}, [], \{\}) \Rightarrow^{*}_{lab} (cc, \Gamma, e, S, \theta) \;\Rightarrow\; cc \equiv \mathit{lab}(e)$ and $cc \not\equiv$ `"SUB"`
where $\equiv$ denotes syntactic identity.

PROOF. This is proved by induction over the length of the trace of the labeled computation $\Rightarrow^{*}_{lab}$.

Inductive Hypothesis. $cc \not\equiv$ `"SUB"` and $e \equiv \varphi^{cc}[U[e]]$ and $\mathit{heapok}(\Gamma)$ and $\mathit{stackok}(S)$, where $\mathit{heapok}(\Gamma)$ requires that all the heap bindings in Γ are labeled with the attached cost center, unless they are subsumed, and $\mathit{stackok}(S)$ requires (1) that the cost center `"SUB"` does not appear on the stack and (2) any *alts* are correctly labeled.

$$\begin{aligned} \mathit{heapok}(\Gamma) &= \forall x, cc, e : x \overset{cc}{\mapsto} e \in \Gamma \Rightarrow e \equiv \varphi^{cc}[U[e]] \\ \mathit{stackok}(S) &= \forall cc, x, \mathit{alts} : (^{\#x}_{cc}) \in S \Rightarrow cc \not\equiv \texttt{"SUB"}, \text{ and} \\ &\qquad (\mathit{alts}, cc) \in S \Rightarrow cc \not\equiv \texttt{"SUB"} \text{ and } \forall e_j \in \mathit{alts} : e_j = \varphi^{cc}[U[e_j]] \end{aligned}$$

Base Case: $\Rightarrow^{0}_{lab}$. $cc \equiv cc_{init} \equiv$ `"MAIN"`, $\Gamma \equiv \Gamma^{lab}_{init}$, $e \equiv e^{lab}_{init} \equiv \varphi^{cc_{init}}[e_{init}]$, and $S \equiv []$. Clearly $cc \not\equiv$ `"SUB"`, $\mathit{stackok}([])$ and

$$\varphi^{cc}[U[e]] \equiv \varphi^{cc_{init}}[U[\varphi^{cc_{init}}[e_{init}]]] \equiv \varphi^{cc_{init}}[e_{init}] \equiv e.$$

From the definition of Γ^{lab}_{init} above we have $x \overset{cc}{\mapsto} e^{lab} \in \Gamma^{lab}_{init} \Rightarrow e^{lab} = \varphi^{cc}[e]$. Since $\varphi^{cc}[U[e^{lab}]] \equiv \varphi^{cc}[U[\varphi^{cc}[e]]] \equiv \varphi^{cc}[e] \equiv e^{lab}$, $\mathit{heapok}(\Gamma^{lab}_{init})$.

Inductive Case: $\Rightarrow^{k+1}_{lab}$. Consider the last transition in the trace of the labeled computation $\Rightarrow^{*}_{lab}$.

Case: app_1. $(cc, \Gamma, e\ x\ ^{l}, S, \theta) \Rightarrow^{app_1}_{lab} (cc, \Gamma, e, x{:}S, \theta{+}\{cc \mapsto \mathrm{A}\})$.

By the inductive hypothesis $cc \not\equiv$ "SUB", $\mathit{heapok}(\Gamma)$, $\mathit{stackok}(S)$, and $e\ x\ ^{l} \equiv \varphi^{cc}[U[e\ x\ ^{l}]]$. Clearly $\mathit{stackok}(x{:}S)$ and

$$\begin{aligned} & e\ x\ ^{l} \equiv \varphi^{cc}[U[e\ x\ ^{l}]] \ \equiv\ \varphi^{cc}[U[e]\ x] \ \equiv\ \varphi^{cc}[U[e]]\ x\ ^{cc} \\ \Rightarrow \quad & e \equiv \varphi^{cc}[U[e]]. \end{aligned}$$

Case: app_2. $(cc_\lambda, \Gamma, \lambda y.e\ ^{l}, x{:}S, \theta) \Rightarrow^{app_2}_{lab} (cc_\lambda, \Gamma, e[x/y], S, \theta)$.

By the inductive hypothesis $cc_\lambda \not\equiv$ "SUB", $\mathit{heapok}(\Gamma)$, $\mathit{stackok}(x{:}S)$, and $\lambda y.e\ ^{l} \equiv \varphi^{cc_\lambda}[U[\lambda y.e\ ^{l}]]$. Clearly $\mathit{stackok}(S)$ and

$$\begin{aligned} & \lambda y.e\ ^{l} \equiv \varphi^{cc_\lambda}[U[\lambda y.e\ ^{l}]] \ \equiv\ \varphi^{cc_\lambda}[\lambda y.U[e]] \ \equiv\ \lambda y.\varphi^{cc_\lambda}[U[e]]\ ^{cc_\lambda} \\ \Rightarrow \quad & e \equiv \varphi^{cc_\lambda}[U[e]]. \end{aligned}$$

Since the variable x is unlabeled, substituting x for y in e does not modify the labeling of e, i.e., $e[x/y] \equiv \varphi^{cc_\lambda}[U[e[x/y]]]$. Any labeled occurences of y in e will result in identically labeled occurences of x in $e[x/y]$.

Case: var_1. $(cc, \Gamma[x \overset{cc_z}{\mapsto} z], x^{l}, S, \theta)$
$\Rightarrow^{var_1}_{lab} (\mathrm{s}(^{cc}_{cc_z}\, z), \Gamma[x \overset{cc_z}{\mapsto} z], \mathbf{labsub}(cc_z, cc, z), S, \theta{+}\{cc \mapsto \mathrm{V}\})$.

By the inductive hypothesis $cc \not\equiv$ "SUB", $x^{l} \equiv \varphi^{cc}[U[x^{l}]]$, $\mathit{stackok}(S)$, and $\mathit{heapok}(\Gamma[x \overset{cc_z}{\mapsto} z]) \Rightarrow z \equiv \varphi^{cc_z}[U[z]]$. If $z \equiv \lambda y.e\ ^{l}$ and $cc_z \equiv$"SUB" or "CAF" we have $\mathrm{s}(^{cc}_{cc_z}\, z) = cc$ and $\mathbf{labsub}(cc_z, cc, z) = \varphi^{cc}[U[z]]$. Otherwise, $\mathrm{s}(^{cc}_{cc_z}\, z) = cc_z \not\equiv$ "SUB" and $\mathbf{labsub}(cc_z, cc, z) = z \equiv \varphi^{cc_z}[U[z]]$.

Case: var_2. $(cc, \Gamma[x \overset{cc_e}{\mapsto} e], x^{l}, S, \theta) \Rightarrow^{var_2}_{lab} (cc_e, \Gamma, e, \binom{\#x}{cc}{:}S, \theta{+}\{cc \mapsto \mathrm{V}\})$.

By the inductive hypothesis $cc \not\equiv$ "SUB", $\mathit{heapok}(\Gamma[x \overset{cc_e}{\mapsto} e])$, $\mathit{stackok}(S)$, and $x^{l} \equiv \varphi^{cc}[U[x^{l}]]$. Since $e \not\equiv \lambda y.e'$ we must have $cc_e \not\equiv$ "SUB" and $e \equiv \varphi^{cc_e}[U[e]]$. Since $cc \not\equiv$ "SUB" we have $\mathit{stackok}(\binom{\#x}{cc}{:}S)$. Finally it is okay to drop the heap binding, i.e., $\mathit{heapok}(\Gamma)$.

Case: var_3. $(cc_z, \Gamma, z, \binom{\#x}{cc}{:}S, \theta)$
$\Rightarrow^{var_3}_{lab} (\mathrm{s}(^{cc}_{cc_z}\, z), \Gamma[x \overset{cc_z}{\mapsto} z], \mathbf{labsub}(cc_z, cc, z), S, \theta{+}\{cc_z \mapsto \mathrm{U}\})$.

By the inductive hypothesis $cc_z \not\equiv$ "SUB", $\mathit{heapok}(\Gamma)$, $\mathit{stackok}(\binom{\#x}{cc} : S)$, and $z \equiv \varphi^{cc_z}[U[z]]$. Thus, the updated heap binding is okay, i.e., $\mathit{heapok}(\Gamma[x \overset{cc_z}{\mapsto} z])$. If $z \equiv \lambda y.e\ ^{l}$ and $cc_z \equiv$ "CAF" we have $\mathrm{s}(^{cc}_{cc_z}\, z) = cc$ and $\mathbf{labsub}(cc_z, cc, z) = \varphi^{cc}[U[z]]$ and $\mathit{stackok}(\binom{\#x}{cc} : S) \Rightarrow cc \not\equiv$ "SUB". Otherwise, $\mathrm{s}(^{cc}_{cc_z}\, z) = cc_z \not\equiv$ "SUB" and $\mathbf{labsub}(cc_z, cc, z) = z \equiv \varphi^{cc_z}[U[z]]$. Finally it is okay to pop $\binom{\#x}{cc}$ off the stack, i.e., $\mathit{stackok}(S)$.

Case: let. $(cc, \Gamma, \texttt{let}\ \{x_i = e_i\}\ \texttt{in}\ e\ ^{l}, S, \theta)$
$\Rightarrow^{let}_{lab} (cc, \Gamma[y_i \overset{cc}{\mapsto} \widehat{e}_i], \widehat{e}, S, \theta{+}\{cc \mapsto n{*}\mathrm{H}\})$.

By the inductive hypothesis $cc \not\equiv$ `"SUB"`, *heapok*(Γ), *stackok*(S), and `let` $\{x_i$ `=` $e_i\}$ `in` $e\ ^{l} \equiv \varphi^{cc}[U[$`let` $\{x_i$ `=` $e_i\}$ `in` $e\ ^{l}]]$. Now,

$$\begin{aligned}
\texttt{let}\ \{x_i \texttt{ = } e_i\}\ \texttt{in}\ e\ ^{l} &\equiv \varphi^{cc}[U[\texttt{let}\ \{x_i \texttt{ = } e_i\}\ \texttt{in}\ e\ ^{l}]] \\
&\equiv \varphi^{cc}[\texttt{let}\ \{x_i \texttt{ = } U[e_i]\}\ \texttt{in}\ U[e]] \\
&\equiv \texttt{let}\ \{x_i \texttt{ = } \varphi^{cc}[U[e_i]]\}\ \texttt{in}\ \varphi^{cc}[U[e]]\ ^{cc} \\
\Rightarrow\quad e &\equiv \varphi^{cc}[U[e]] \ \text{ and } \ \forall e_i :\ e_i \equiv \varphi^{cc}[U[e_i]].
\end{aligned}$$

Since the fresh y_i are unlabeled, substituting the y_i for the x_i will not modify the labeling, i.e., $\widehat{e} \equiv \varphi^{cc}[U[\widehat{e}]]$ and $\forall e_i :\ \widehat{e_i} \equiv \varphi^{cc}[U[\widehat{e_i}]]$. Thus, the new heap bindings are okay, i.e., *heapok*$(\Gamma[y_i \overset{cc}{\mapsto} \widehat{e_i}])$.

Case: $case_1$. $(cc, \Gamma,$ `case` e `of` $alts\ ^{l}, S, \theta)$
$\Rightarrow^{case_1}_{lab}$ $(cc, \Gamma, e, (alts,cc) : S, \theta + \{cc \mapsto \text{C}\})$.

By the inductive hypothesis $cc \not\equiv$ `"SUB"`, *heapok*(Γ), *stackok*(S), and `case` e `of` $alts\ ^{l} \equiv \varphi^{cc}[U[$`case` e `of` $alts\ ^{l}]]$. Now,

$$\begin{aligned}
\texttt{case}\ e\ \texttt{of}\ \{C_j\ \overline{x_{ja}}_{a_j}\ \texttt{->}\ e_j\}\ ^{l} &\equiv \varphi^{cc}[U[\texttt{case}\ e\ \texttt{of}\ \{C_j\ \overline{x_{ja}}_{a_j}\ \texttt{->}\ e_j\}\ ^{l}]] \\
&\equiv \varphi^{cc}[\texttt{case}\ U[e]\ \texttt{of}\ \{C_j\ \overline{x_{ja}}_{a_j}\ \texttt{->}\ U[e_j]\}] \\
&\equiv \texttt{case}\ \varphi^{cc}[U[e]]\ \texttt{of}\ \{C_j\ \overline{x_{ja}}_{a_j}\ \texttt{->}\ \varphi^{cc}[U[e_j]]\}\ ^{cc} \\
\Rightarrow\quad e &\equiv \varphi^{cc}[U[e]] \ \text{ and } \ \forall e_j :\ e_j \equiv \varphi^{cc}[U[e_j]].
\end{aligned}$$

Since $cc \not\equiv$ `"SUB"` and $\forall e_j \in alts : e_j \equiv \varphi^{cc}[U[e_j]]$ we have *stackok*$((alts,cc) : S)$.

Case: $case_2$. $(cc_C, \Gamma, C_k\ \overline{x}_{a_k}\ ^{l}, (alts,cc) : S, \theta) \Rightarrow^{case_2}_{lab} (cc, \Gamma, e_k[x_i/y_{ki}], S, \theta)$.

By the inductive hypothesis $cc_C \not\equiv$ `"SUB"`, *heapok*(Γ), *stackok*$((alts,cc) : S)$, and $C_k\ \overline{x}_{a_k}\ ^{l} \equiv \varphi^{cc}[U[C_k\ \overline{x}_{a_k}\ ^{l}]]$. Now, *stackok*$((alts,cc) : S) \Rightarrow cc \not\equiv$ `"SUB"` and $\forall e_j \in alts : e_j \equiv \varphi^{cc}[U[e_j]]$ and *stackok*(S). Since the x_i are unlabeled, substituting the x_i for the y_{ki} in e_k will not modify the labeling of e_k, i.e., $e_k[x_i/y_{ki}] \equiv \varphi^{cc}[U[e_k[x_i/y_{ki}]]]$.

Case: *scc*. $(cc, \Gamma,$ `scc` $cc_{scc}\ e\ ^{l}, S, \theta) \Rightarrow^{scc}_{lab} (cc_{scc}, \Gamma, e, S, \theta + \{cc_{scc} \mapsto \text{E}\})$.

By the inductive hypothesis $cc \not\equiv$ `"SUB"`, *heapok*(Γ), *stackok*(S), and `scc` $cc_{scc}\ e\ ^{l} \equiv \varphi^{cc}[U[$`scc` $cc_{scc}\ e\ ^{l}]]$. Since an explicit `scc` cannot annotate with the `"SUB"` cost center we have $cc_{scc} \not\equiv$ `"SUB"`. Now,

$$\begin{aligned}
\texttt{scc}\ cc_{scc}\ e\ ^{l} &\equiv \varphi^{cc}[U[\texttt{scc}\ cc_{scc}\ e\ ^{l}]] \equiv \varphi^{cc}[\texttt{scc}\ cc_{scc}\ U[e]] \\
&\equiv \texttt{scc}\ cc_{scc}\ \varphi^{cc_{scc}}[U[e]]\ ^{cc} \\
\Rightarrow\quad e &\equiv \varphi^{cc_{scc}}[U[e]].
\end{aligned}$$

Finally, we observe that $e \equiv \varphi^{cc}[U[e]] \Rightarrow lab(e) \equiv cc$ as required by the theorem. □

E. OVERLOADING, DICTIONARIES, AND PROFILING

Haskell has a systematic way of handling overloading through the use of type classes [Wadler and Blott 1989]. A type class is a set of types sharing some operations, called methods. A `class` declaration specifies what the common operations are. A type is declared to be in the class with an `instance` declaration. The instance declaration describes what the methods in the class do for that particular type, for example,

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  (/=) x y = if x == y then False else True     -- default method

instance Eq Char where
  (==) x y = eqChar x y
  (/=) x y = neChar x y

instance (Eq a) => Eq [a] where
  []     == []     = True
  (x:xs) == (y:ys) = x == y && xs == ys
  _      == _      = False

elem :: Eq a => a -> [a] -> Bool
elem v l = scc "elem" case l of []   -> False
                                x:xs -> v == x || elem v xs

seenStr :: [Char] -> [[Char]] -> Bool
seenStr str seen = scc "seenStr" elem str seen
```

The standard mechanism for implementing overloading has been to use method dictionaries [Hall et al. 1994; Wadler and Blott 1989], though various optimizations and alternative schemes have been proposed [Augustsson 1993; Jones 1992]. Each overloaded function is given an extra dictionary argument that contains the methods for the particular type at which the function is being applied. The dictionary is given as an argument to the generic method function, which extracts the particular method from the dictionary, which is then applied to the method arguments as before. The translation for the example above is[15]

```
(==) (m, _) = m
(\=) (_, n) = n
Eq.(==) dEq x y = error "no default method for =="
Eq.(/=) dEq x y = if (==) dEq x y then False else True

Char.(==) x y = eqChar x y
Char.(/=) x y = neChar x y
dict.Eq.Char = (Char.(==), Char.(/=))                    -- a CAF

List.(==) dEq []     []     = True
List.(==) dEq (x:xs) (y:ys) = (==) dEq x y && List.(==) dEq xs ys
List.(==) dEq _      _      = False
List.(/=) dEq xs     ys     = Eq.(/=) (dfun.Eq.List dEq) xs ys

dfun.Eq.List dEq = (List.(==) dEq, List.(/=) dEq)
dict.Eq.List.Char = dfun.Eq.List dict.Eq.Char            -- a CAF

elem dEq v l = scc "elem" case l of
                 []   -> False
                 x:xs -> (==) dEq v x || elem dEq v xs

seenStr str seen = elem dict.Eq.List.Char str seen
```

[15]For convenience we ignore the restrictions Haskell places on the characters in identifier names.

Given the original source code above, the programmer would expect the costs of comparing the strings to be subsumed by the cost center `"elem"`. However, under the original semantics these costs are actually attributed to the `"CAF"` cost center attached to the definitions of `dict.Eq.List.Char` and `dict.Eq.Char` (Section 7.1). This was the main motivation for introducing the modified semantics in Section 7.2, since users queried the unexpectedly large proportion of execution costs attributed to `"CAF"`. Under the modified scheme, the costs of constructing the dictionaries are attributed to the `"CAF"` cost center, but the cost of applying the methods are subsumed by the enclosing cost center, as required.

So that the cost of constructing dictionaries is reported separately we actually attach the cost center `"DICT"` to top-level dictionary declarations. This cost center is treated in exactly the same way as the `"CAF"` cost center by the $\mathrm{s}(^{cc}_{cc_z}\ z)$ predicate.

Unfortunately, not all dictionaries are declared at the top-level, since a particular dictionary may depend on a runtime dictionary argument. For example, searching for a singleton list in a list of lists might be defined as

```
elem_ll :: Eq a => a -> [[a]] -> Bool
elem_ll v ll = scc "elem_ll" elem [v] ll
```

This is translated to

```
elem_ll dEq v ll = scc "elem_ll" elem (dfunEqList dEq) [v] ll
```

which builds a dictionary for comparing lists of the element type and passes it to `elem`. In this situation the costs of comparing the strings are attributed to the cost center `"elem_ll"` because the methods are constructed by `dfunEqList` in the scope of `"elem_ll"`. To ensure that these dynamic dictionary methods are correctly subsumed by their actual call sites we place a special $\mathtt{scc}_{dict}$ annotation in the dictionary construction functions.

```
dfunEqList dEq = scc_dict "DICT" (EqList(==) dEq, EqList(/=) dEq)
```

The $\mathtt{scc}_{dict}$ annotation is identical to an `scc` annotation, except that it does not increment the `inner` count of the enclosing cost center. This means that the introduction of the $\mathtt{scc}_{dict}$ annotation does not affect the `inner` count of the enclosing cost center. The $\mathtt{scc}_{dict}$ annotation is different from the $\mathtt{scc}_{sub}$ annotation, introduced in Section 5.3, since it still increments the `scc` count of the new cost center — we are, after all, interested in how many dictionaries are constructed.

The end result of these modifications is that the costs of executing a particular method is always subsumed by the cost center enclosing the application site of the method selector, unless, of course, the particular method has an explicit `scc` annotation. This coincides with the programmer's expectations, since these applications are introduced where the overloaded method identifier occurs in the source.

ACKNOWLEDGMENTS

Thanks to the Glasgow Haskell team for providing the framework for making this work possible. Thanks also to Peter Sestoft, David Turner, Hans Loidl, Will Partain, and the anonymous referees for providing useful comments and suggestions on this article. Finally, we would like to acknowledge the support of the EPSRC and the Commonwealth Scholarship Commission.

REFERENCES

Appel, A. W., Duba, B. F., and MacQueen, D. B. 1988. Profiling in the presence of optimization and garbage collection. Tech. Rep. CS-TR-197-88, Dept. of Computer Science, Princeton Univ., Princeton, N.J.

Augustsson, L. 1993. Implementing Haskell overloading. In the *Conference on Functional Programming Languages and Computer Architecture*. ACM, New York, 65–73.

Augustsson, L. and Johnsson, T. 1989. The Chalmers Lazy-ML compiler. *Comput. J. 32*, 2, 127–141.

Bentley, J. L. 1982. *Writing Efficient Programs*. Prentice-Hall, Englewood Cliffs, N.J.

Clack, C., Clayman, S., and Parrott, D. 1995. Lexical profiling: Theory and practice. *J. Funct. Program. 5*, 2, 225–277.

Fairbairn, J. and Wray, S. 1987. TIM — A simple lazy abstract machine to execute supercombinators. In the *Proceedings of the IFIP Conference on Functional Programming Languages and Computer Architecture*, G. Kahn, Ed.,Lecture Notes in Computer Science, vol. 274. Springer-Verlag, Berlin, 34–45.

Graham, S. L., Kessler, P. B., and McKusick, M. K. 1983. An execution profiler for modular programs. *Softw. Pract. Exper. 13*, 8, 671–685.

Hall, C. V., Hammond, K., Peyton Jones, S. L., and Wadler, P. L. 1994. Type classes in Haskell. In the *European Symposium on Programming* (*ESOP'94*), D. Sannella, Ed.,Lecture Notes in Computer Science, vol. 788. Springer-Verlag, Berlin, 241–256.

Hudak, P., Peyton Jones, S. L., Wadler, P. L., Arvind, Boutel, B., Fairbairn, J., Fasel, J., Guzman, M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, R., Nikhil, R. S., Partain, W., and Peterson, J. 1992. Report on the functional programming language Haskell, Version 1.2. *ACM SIGPLAN Not. 27*, 5.

Hutton, G. 1992. Higher-order functions for parsing. *J. Funct. Program. 2*, 3, 323–343.

Ingalls, D. 1972. The execution profile as a measurement tool. In *Design and Optimization of Compilers*, R. Rustin, Ed. Prentice-Hall, Englewood Cliffs, N.J., 107–128.

Jones, M. P. 1992. Efficient implementation of type class overloading. Dept. of Computer Science, Oxford Univ., Oxford, U.K.

Knuth, D. E. 1971. An Empirical Study of FORTRAN Programs. *Softw. Pract. Exper. 1*, 105–133.

Kozato, Y. and Otto, G. P. 1993. Benchmarking real-life image processing programs in lazy functional languages. In the *Conference on Functional Programming Languages and Computer Architecture*. ACM, New York, 18–27.

Launchbury, J. 1993. A natural semantics for lazy evaluation. In the *20th ACM Symposium on the Principles of Programming Languages*. ACM, New York, 144–154.

Morgan, R. G. and Jarvis, S. A. 1995. Profiling large-scale lazy functional programs. In the *Proceedings of the High Performance Functional Computing Workshop*, A. P. Wim Bohm and J. T. Feo, Eds. Lawrence Livermore National Laboratory, Livermore, Calif., 222–234.

Morgan, R. G., Garigliano, R., Jarvis, S. A., and Parker, B. S. 1994. Maintenance and development of large scale lazy functional programs. In the *Dagstuhl Workshop on Functional Programming in the Real World*. Dagstuhl, Saarbrücken, Germany.

Partain, W. D. 1993. The `nofib` benchmark suite of Haskell programs. In *Functional Programming, Glasgow 1992*, J. Launchbury and P. M. Sansom, Eds.,Workshops in Computing. Springer-Verlag, Berlin, 195–202.

Peyton Jones, S. L. 1992. Implementing lazy functional languages on stock hardware: The Spineless Tagless G-machine. *J. Funct. Program. 2*, 2, 127–202.

Peyton Jones, S. L., Hall, C. V., Hammond, K., Partain, W. D., and Wadler, P. L. 1993. The Glasgow Haskell compiler: A technical overview. In the *Joint Framework for Information Technology* (*JFIT*) *Technical Conference Digest*. SERC, Swindon, U.K., 249–257.

RÖJEMO, N. 1995. Garbage collection and memory efficiency in lazy functional languages. Ph.D. thesis, Dept. of Computing Science, Chalmers Univ., Chalmers, Sweden.

RÖJEMO, N. AND RUNCIMAN, C. 1996. Lag, drag, void, and use: Heap profiling and space-efficient compilation revisited. In the *International Conference on Functional Programming*. ACM, New York, 34–41.

RUNCIMAN, C. AND RÖJEMO, N. 1996. New dimensions in heap profiling. *J. Funct. Program. 6*, 4, 587–620.

RUNCIMAN, C. AND WAKELING, D. 1993. Heap profiling of lazy functional programs. *J. Funct. Program. 3*, 2, 217–245.

SANSOM, P. M. 1994. Time profiling a lazy functional compiler. In *Functional Programming, Glasgow 1993*, K. Hammond and J. O'Donnell, Eds.,Workshops in Computing. Springer-Verlag, Berlin, 252–264.

SANTOS, A. 1995. Compilation by transformation in non-strict functional languages. Ph.D. thesis, Res. Rep. FP-1995-17, Dept. of Computing Science, Univ. of Glasgow, Scotland.

SESTOFT, P. 1997. Deriving a lazy abstract machine. *J. Funct. Program. 7*, 3. To be published.

WADLER, P. L. AND BLOTT, S. 1989. How to make ad-hoc polymorphism less ad hoc. In the *16th ACM Symposium on the Principles of Programming Languages*. ACM, New York.

Received February 1996; revised July 1996; accepted October 1996