J. Functional Programming 5 (4): 501–547, October 1995 © 1995 Cambridge University Press

Semantics directed program execution monitoring[†]

AMIR KISHON and PAUL HUDAK

Department of Computer Science, Yale University,
New Haven, CT 06520, USA
e-mail: {kishon,hudak}@cs.yale.edu

Abstract

Monitoring semantics is a formal model of program execution which captures 'monitoring activity' as found in profilers, tracers, debuggers, etc. Beyond its theoretical interest, this formalism provides a new methodology for implementing a large family of source-level monitoring activities for sequential deterministic programming languages. In this article we explore the use of monitoring semantics in the specification and implementation of a variety of monitors: profilers, tracers, collecting interpreters, and, most importantly, interactive source-level debuggers. Although we consider such monitors only for (both strict and non-strict) functional languages, the methodology extends easily to imperative languages, since it begins with a continuation semantics specification.

In addition, using standard partial evaluation techniques as an optimization strategy, we show that the methodology forms a practical basis for building real monitors. Our system can be optimized at two levels of specialization: specializing the interpreter with respect to a monitor specification automatically yields an instrumented interpreter; further specializing this instrumented interpreter with respect to a source program yields an instrumented program, i.e. one in which the extra code to perform monitoring has been automatically embedded into the program.

Capsule Review

Monitoring activities are those performed by profilers, tracers, etc. In this paper the authors describe a formalism for combining several monitoring semantics in a correct way. The importance here being that the monitoring activities do not interfere with each other or with the standard semantics. The authors present several examples of monitoring semantics for both a strict and lazy functional language and combine these with the corresponding semantics, especially if a tracer for a lazy language does not change the order of evaluation. The authors show how partial evaluation can be used to automatically transform a standard interpreter and a monitor specification into an instrumented interpreter that performs the monitoring activity. They also show partial evaluation which can then be used to generate instrumented programs by specializing these instrumented interpreters with respect to subject programs. The results obtained in this way are very impressive and compare well with those of handwritten programs.

[†] This research was supported by ARPA under ONR contracts N00014-90-C-0024 and N00014-91-J-4043.

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

1 Introduction

We define a *program execution monitor*, or simply 'monitor', as any software tool that monitors some aspect of the dynamics of program execution. Examples include debuggers, profilers, tracers, collecting interpreters, etc. We are interested in the general problem of formally specifying and implementing monitors, although in this paper we limit the scope to monitors for *sequential, deterministic* programming languages (or a language's sequential, deterministic interpretation).

Despite the importance of monitors in any software development environment, there has been little work on either formal or general treatments of the problem. Current systems are primarily based on two approaches:

- **Code transformation.** The source program is transformed (instrumented) to include monitoring instructions (for representative research see (Dybvig *et al.*, 1988; O'Donnell and Hall, 1988; Tolmach and Appel, 1990)).
- **Execution transformation.** The process that executes the source code (e.g. an interpreter) is modified (instrumented) to incorporate monitoring activities (see (O'Donnell and Hall, 1988; Safra and Shapiro, 1989; Shapiro, 1982; Sterling and Shapiro, 1986; Toyn and Runciman, 1986) for some exemplary work).

These strategies have their advantages, but as general methodologies many limitations arise. In particular:

1. **Informality.** Approaches based on a language's formal semantics (whether denotational or otherwise) are rare (some preliminary work in the area can be found (Berry, 1991; Bertot, 1988; Toyn and Runciman, 1986)). Thus many of the current techniques have little formal semantic justification.
2. **Unsoundness.** Some approaches use unsound transformations which may interact adversely with normal execution or with other monitoring activities. For example, O'Donnell and Hall's (1988) instrumentation of functional programs interferes with non-strict evaluation order, and the debugging data is not necessarily faithful to the standard semantics.
3. **Hand-crafting.** Many approaches amount to a hand-crafting of debugging tools. As a result, common elements of monitors' designs are often overlooked, and the solutions do not provide a basis for a general framework.
4. **Non-compositionality.** The composition of monitoring tools is usually neglected. For example, by instrumenting the source code to perform one kind of monitoring, another monitoring activity that relies on the source code will 'see' the extra code.

Motivated by the need for a more formal treatment of program execution monitoring, we have developed a specification technique that we call *monitoring semantics* and an implementation technique based on *partial evaluation* that together provide a sound and effective methodology for building program execution monitors (Kishon, 1992; Kishon, Hudak and Consel, 1988).

The soundness of our methodology begins with the observation that a language's *continuation semantics* specifies a linear ordering on program execution, and thus

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

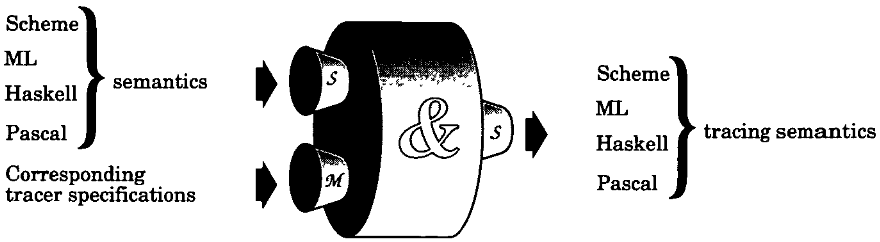
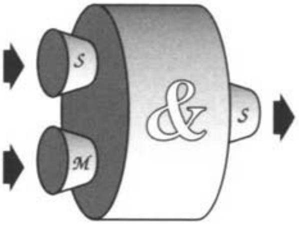


Fig. 1. Combining tracer specifications with different standard semantics

can be used as the basis for ordering monitoring activity (just as it is used to guide code gene ;d compilers'). Using the framework of continuation sen behaviour of a large family of monitors. The resulting m hen be automatically combined with a language's stan(*composite semantics* that captures both standard behavi es. This composite semantics, instead of interpreting a p :ment α in a domain $\mathbf{Ans}_{std}$ of standard 'final answers', i with type $\mathbf{Ans}_{mon} = \mathbf{MS} \rightarrow (\mathbf{Ans}_{std} \times \mathbf{MS})$, where **MS** is a domain of 'monitor states'. Given $\sigma_0 : \mathbf{MS}$ as an initial (presumably empty) monitor state, then:



$$f\ \sigma_0 \ \Rightarrow\ \langle \alpha', \sigma_f \rangle$$

where σ_f is the resulting monitoring information and α' is the result of the standard evaluation. We systematically construct the composite semantics in such a way that all instantiations of the semantics (i.e. all possible monitors defined using our approach) have the property that $\alpha' = \alpha$.

In this paper we describe our overall methodology and its application, highlighting the following attributes:

1. **Applicability.** A monitoring semantics can be specified for any language for which a continuation semantics is available (e.g. Scheme (Clinger and Rees, 1991), Pascal (Tennent, 1977), Prolog (Allison, 1986), etc.).
2. **Expressiveness.** The methodology is able to capture a large family of sequential monitoring activities (e.g. profiling, tracing, interactive debugging, etc.).
3. **Modularity.** The monitor specification is written relatively independently from the standard semantics, yet inherits the standard semantics for actual computation. This means that a single tracer specification, for example, can be combined with the the standard semantics of several different source languages, as shown in Figure 1. Similarly, one can construct different debugging tools for a language by combining its standard semantics with different monitor specifications, as shown in Figure 2. (The '&' operator is the key element in our framework that combines an interpreter and monitor specification.)
4. **Soundness.** We show that *any* monitor constructed within our framework *cannot* alter the standard semantics, even though it relies on the standard semantics.
5. **User Interaction.** Interactive debuggers are obviously an important capability, and despite the implied I/O-dependencies, this capability is easily captured in our framework.

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

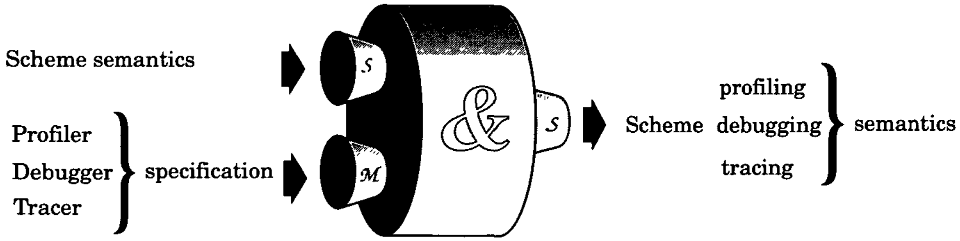
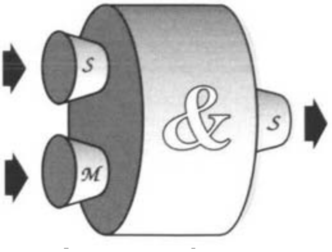


Fig. 2. Combining different monitor specifications with standard semantics

6. **Practicality.** To use our methodology to build practical monitors, we rely on automatic ier *et al.*, 1988; Jones *et al.*, 1987) as an optimizati
   (a) **Instrun** ng an interpreter with respect to a fixed set of omatically yields an interpreter instru- mentec
   (b) **Instrur** the instrumented interpreter (from the previoı urce program produces an instrumented program; i.e. a program with extra code to perform the monitoring actions.



With this technique we have built monitors whose execution speed compares reasonably well to the execution speeds of conventional interpreters and compiled programs instrumented for debugging.

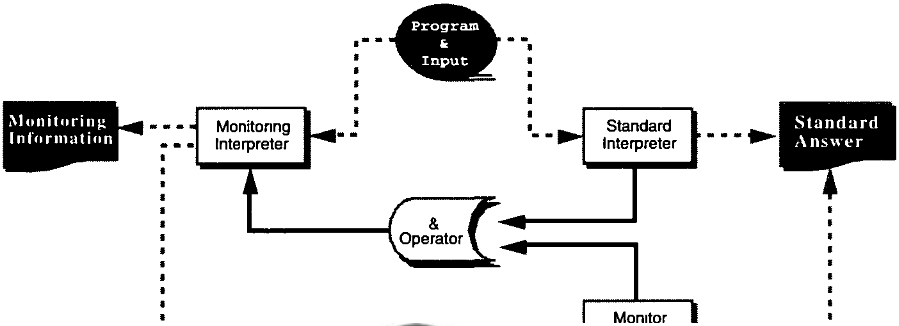
1.1 Overview

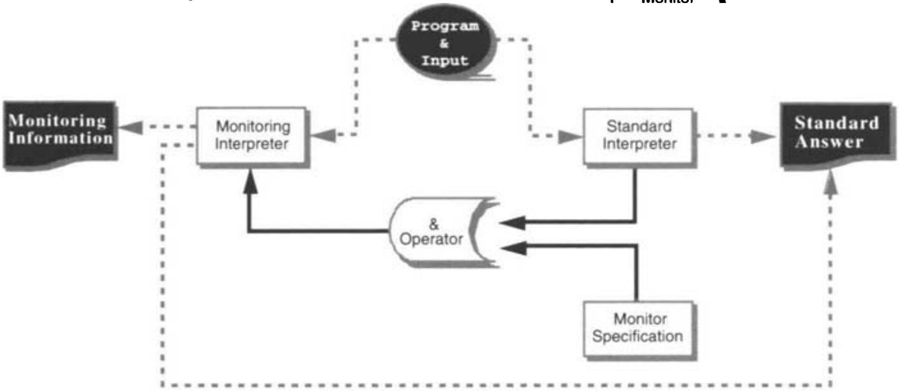
Figure 3 illustrates the relationships between the various components of our system. From the standard interpreters for strict and non-strict functional languages described in Section 2, and the monitor specifications described in Section 3, we describe in Section 4 how to automatically derive a monitoring interpreter by combining both specifications (using the '&' operator). Then in Section 5 we present several more monitor specifications for the strict and and non-strict standard interpreters, followed by a discussion of monitor composition in Section 6. Section 7 describes how to specify interactive monitors, using as an example a generic interactive source-level debugger for both the strict and non-strict languages. Finally, in Sections 8 and 9 we describe optimizations based on partial evaluation, and provide some detailed benchmarks of the resulting monitors.

Rather than use denotational semantics notation, all of the semantics specified in this paper are written in Haskell,[†] which resembles closely the meta-language of denotational semantics. But this is more than just a stylistic choice: all of the specifications given can be executed, and indeed our goal is to create real monitors, not just mathematical specifications. Readers not familiar with Haskell are referred elsewhere (Hudak and Fasel, 1992; Hudak *et al.*, 1992).

[†] As an aid to the eye we use 'λ' instead of Haskell's '\' for lambda abstractions.

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press





ard
the
and

2.1 Kernel language

Our source language will be a sugared lambda calculus typical of the 'kernel language' of many functional languages. Its abstract syntax is given by:

$$\begin{array}{lll} e ::= & k & (\textit{constant}) \\ & \mid x & (\textit{variable}) \\ & \mid e_1\ e_2 & (\textit{application}) \\ & \mid \texttt{lambda}\ x\ .\ e & (\textit{abstraction}) \\ & \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 & (\textit{conditional}) \\ & \mid e_1\ op\ e_2 & (\textit{binop}) \\ & \mid \texttt{letrec}\ f = \texttt{lambda}\ x\ .\ e_1\ \texttt{in}\ e_2 & (\textit{letrec}) \\ & \mid \{\mu\} : e & (\textit{labeled expressions}) \end{array}$$

or, as it would be defined as a Haskell datatype:

```
module KernelSyntax where
data Exp = Con Int                -- Constant
         | Var Id                 -- Variable
         | App Exp Exp            -- Application
         | Abs Id Exp             -- Abstraction
         | Cnd Exp Exp Exp        -- Conditional
         | Bop Id Exp Exp         -- Binary Application
         | Rec Id Exp Exp         -- Letrec
         | Lxp Label Exp          -- Labeled Expression
```

Note the category of 'labeled expressions', whose purpose is solely for monitoring; the purpose of these labels will be discussed later. In the remainder of this section

```
module Lazy where      -- see KernelSyntax module for syntax specification

--- SEMANTIC ALGEBRAS ---

-- Denotable values: integers, booleans and functions
data D = Num Int | Bol Bool | Fun (StoreVal -> Store -> Kont -> Ans)
-- Storeable values: values, thunks and undefined store values
data StoreVal = Val D | Thunk (Store -> Kont -> Ans) | StoreValUndef

type Env = Id -> Loc                                  -- Environments
envInit  = (λid -> error "undef id") :: Env
type Store = StoreType StoreVal    -- Stores
storeInit  = (storeEmpty StoreValUndef) :: Store
type Kont = (D,Store) -> Ans                          -- Continuations
kontInit  = (λ(v,store) -> toAns (v,store)) :: Kont

--- VALUATION FUNCTIONAL ---

eval :: Exp -> (Env,Store) -> Kont -> Ans
eval = fix lazyEvalf

lazyEvalf :: Functional (Exp -> (Env,Store) -> Kont -> Ans)
lazyEvalf eval = λexp (env,s) k -> case exp of
   (Con v)        -> k (Num v,s)
   (Var id)       -> case (storeLook s loc) of (Val v)   -> k (v,s)
                                               (Thunk t) -> t s k'
                     where k' (v,s) = k (v,storeUpd s loc (Val v))
                           loc = (env id)
   (Abs id e1)    -> k (Fun f,s)
                     where f v s = eval e1 (env',s')
                                   where (loc,s') = storeAlloc s
                                         s'      = storeUpd s' loc v
                                         env'     = envUpd env id loc
   (App e1 e2)    -> eval e1 (env,s) (λ(Fun f,s') ->
                          f (Thunk (λs' -> eval e2 (env,s'))) s' k)
   (Cnd e1 e2 e3) -> eval e1 (env,s) (λ(Bol v,s') ->
                          eval (if v then e2 else e3) (env,s') k
   (Bop id e1 e2) -> eval e1 (env,s) (λ(v',s') ->
                          eval e2 (env,s') (λ(v',s') ->
                               k (applyBop id v' v',s')))
   (Rec id e1 e2) -> eval e2 (env',s') k
                     where (loc,s') = storeAlloc s
                           env'     = envUpd env id loc
                           thunk    = Thunk (λs k -> eval e1 (env',s) k)
                           s'      = storeUpd s' loc thunk

applyBop :: Id -> D -> D -> D    -- similar to applyBop in module Eager
```

Fig. 4. Lazy interpreter.

we present two standard interpreters for this language, one using lazy evaluation, the other eager.[‡]

2.2 Lazy interpreter

As discussed in Section 1, we require that the standard semantics be given in a *continuation-passing style* (often written 'CPS'). At first this may seem odd for a purely functional language, but in fact for a non-strict language it is quite useful, since it allows us to explicate the manipulation of thunks in a sequential interpretation. More precisely, we use the environment to map identifiers to locations, the store to map locations to either a closure (thunk) or a computed value (if the thunk has already been evaluated), and continuations to capture the evaluation order. In this way we are able to reflect the fact that lazy evaluation results in each expression being evaluated 'at most once'.

The lazy interpreter's specification is shown in Figure 4 (some of the operations used are defined in the 'standard algebras' section in Appendix A). There are two unconventional features in this interpreter: First is the explicit use of the *fixpoint of functionals* to specify the valuation functions (see `lazyEvalf`); this will allow us later to derive enhanced valuation functions that 'inherit' the behaviour of the standard semantics (we use the term *functional* to denote a function which we intend to take the fixpoint of to capture the intended semantics of interest).

The second unconventional feature is the omission of the definition of the answer domain `Ans` and its constructor `toAns`. This was done intentionally, since we would like to parameterize the interpreter with respect to its final answer. For example, a very simple answer algebra for this language might be given by:

```
module StdLazyAnswer where

type Ans = Int

toAns :: (D,Store) -> Ans
toAns (Num n,store) = n
```

which will interpret the results as integers. However, for the purpose of monitoring, we wish to interpret a program as a function `f :: MonState -> (stdAns,MonState)`, such that given `monState :: MonState` as an initial monitor state, then:

```
(stdAns,monState') = f monState
```

where `monState'::MonState` is the resulting monitoring information and `stdAns` is the standard interpretation. To do this we redefine the answer algebra as follows

[‡] We use the terms 'lazy' and 'eager' here rather than 'strict' and 'non-strict' because we are capturing an operational property, not just a mathematical one.

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

(the actual definition of the monitor state datatype MonState will be defined later in the monitor specification):

```
module MonLazyAnswer where

type Ans = MonState -> (Int, MonState)

toAns :: (D,Store) -> Ans
toAns (Num n,store) = λmonState -> (n,monState)
```

An 'interpreter package' and the standard driver. To help modularize the design, we now package the components of the interpreter specification within the following datatype:

```
data InterpreterType parser evalf semArgs kont =
     Interpreter parser evalf semArgs kont
```

Thus, for the lazy interpreter we define:

```
lazy :: InterpreterType
           (String->Exp)                              -- Parser
           (Functional (Exp->(Env,Store)->Kont->Ans)) -- Valuation fnal
           (Env,Store)                                -- Semantic args
           Kont                                       -- Continuation
lazy = Interpreter expParse lazyEvalf (envInit,storeInit) kontInit
```

where expParse is a parsing function whose definition we omit.

Finally, we provide a 'driver' for the standard interpretation:

```
stdExecute (Interpreter parse evalf semArgs kont) prog =
   (fix evalf) (parse prog) semArgs kont

fix :: (a -> a) -> a
fix f = f (fix f)
```

which is polymorphic in interpreters. Thus given an expression involving factorial (in which we use a hypothetical source-level syntax):

```
fact3 = "letrec mul = lambda x y . x * y in
         in letrec fac = lambda n acc .
                            if n=0 then acc else fac (n-1) (mul n acc)
            in fac 3 1"
```

we can evaluate it using our lazy interpreter as follows:

```
Run> stdExecute lazy fact3
6
```

On the other hand, if we use the enhanced answer algebra (module MonLazyAnswer) and apply the result to an initial monitoring state monState = (), we get:

```
Run> stdExecute lazy fact3 ()
(6,())
```

Since the actual interpretation is still not monitored (we did not enhance the interpreter), the initial monitor state (in this case a null value) remains unchanged. We will of course alter this behaviour shortly.

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

```
module Eager where     -- see KernelSyntax module for syntax specification

--- SEMANTIC ALGEBRAS ---

-- Denotable values: integers,booleans,functions and undefined values
data D = Num  Int | Bol Bool | Fun (D -> Kont -> Ans) | DUndef

type Env = Id -> D                                   -- Environments
envInit id = DUndef
type Kont = D -> Ans                                 -- Continuations
kontInit = λv -> toAns v

--- VALUATION FUNCTIONAL ---

eval :: Exp -> Env -> Kont -> Ans
eval = fix eagerEvalf

eagerEvalf :: Functional (Exp -> Env -> Kont -> Ans)
eagerEvalf eval = λexp env k -> case exp of
      (Con v)        -> k (Num v)
      (Var id)       -> k (env id)
      (Abs id e1)    -> k (Fun (λv -> eval e1 (envUpd env id v)))
      (Cnd e1 e2 e3) -> eval e1 env (λ(Bol v) ->
                             eval (if v then e2 else e3) env k
      (Bop id e1 e2) -> eval e1 env (λv1 ->
                             eval e2 env (λv2 ->
                                  k (applyBop id v1 v2)))
      (App e1 e2)    -> eval e1 env (λ(Fun f) -> eval e2 env (λv -> f v k))
      (Rec id (Abs arg body) e2) ->
                        eval e2 env' k
                        where env'      = envUpd env id (Fun closure)
                              closure v = eval body (envUpd env' arg v)

applyBop :: Id -> D -> D -> D
applyBop id (Num v1) (Num v2) = case id of "+" -> Num (v1 + v2)
                                           "-" -> Num (v1 - v2)
                                           "*" -> Num (v1 * v2)
                                           "=" -> Bol (v1 == v2)

--- STRICT INTERPRETER ---

eager =  Interpreter expParse eagerEvalf envInit kontInit
```

Fig. 5. Eager interpreter (valuation function).

2.3 Eager interpreter

In the same fashion we can interpret our language as strict by defining an 'eager' interpreter, which we can also package into an interpreter datatype that we will call eager. Again, we use a fairly straightforward and conventional continuation semantics specification for our interpreter, as shown in Figure 5.

As with the lazy interpreter we can define different answer algebras to map the final answer to arbitrary domains. For example a standard answer algebra to map the results to integers is given by:

```
module StdEagerAnswer where

type Ans = Int

toAns :: D -> Ans
toAns (Num n) = n
```

whereas the monitoring answer algebra that we will be using is:

```
module MonEagerAnswer where

type Ans = MonState -> (Int, MonState)

toAns :: D -> Ans
toAns (Num n) = λmonState -> (n, monState)
```

The standard driver can now make use of `eager` to evaluate `fact3` as before:

```
Run> stdExecute eager fact3
6
```

3 Second step: monitor specifications

The derivation of a monitoring interpreter can be viewed as combining two interpretations: the standard interpretation and a non-standard monitor interpretation. In this section we discuss the specification of the latter, then in the next section we show how to combine them together.

3.1 A monitor

Like the standard interpreter, the monitor specification has its own syntax, algebras, and 'semantic functions'.

Monitor syntax

To invoke a particular monitoring activity at a specific program point we will *annotate* the source code. In the simplest form these annotations might simply be labels through which the system may uniquely reference any program point; in more complex situations, they may involve 'directives' to control the monitoring process. We use the 'labeled expressions' given earlier in the syntax of our languages to realize these annotations.

As an example, let us introduce two labels `A` and `B`, and annotate a factorial program with them (written again in our hypothetical source language):

```
fac(n) = if (n = 0) then {A}:1 else {B}:(n * fac(n-1))
```

We can now design a monitor to increment a corresponding counter whenever an expression annotated with `{A}` or `{B}` is evaluated; i.e. a simple profiler. The full specification of this monitor is presented in Section 3.2.

```
module Profile where

--- SYNTAX ---

data Label = LProfile FunName

annotate :: Exp -> Exp  -- omitted, annotate every declared function

--- ALGEBRAS ---

-- Profiler state: association list of counters
type MonState = AssocType FunName Counter
type Counter  = Int

--- MONITORING FUNCTIONS ---

pre :: Label -> ast -> semArgs -> MonState -> MonState
pre (LProfile funName) exp semArgs profileEnv =
   assocPut funName (v+1) profileEnv
   where v = if (assocExist funName profileEnv)
             then (assocGet funName profileEnv) else 0

post :: Label -> ast -> semArgs -> kontArgs -> MonState -> MonState
post label exp semArgs kontArgs profileEnv = profileEnv
```

Fig. 6. A profiler monitor specification.

Monitor algebras

Like the standard interpreter, a monitor utilizes a set of algebras (i.e. datatypes and operations) to support its activity. For example, a profiler uses an environment which maps function names to their corresponding counters. We will refer to the parameters needed to support the incremental monitoring activity as the *monitor state.* That is, the monitor state captures information of interest to a specific monitor, and will be incrementally updated during the monitoring process.

Monitoring functions

The monitoring semantic functions actually perform the monitoring activity. A key aspect of our framework is how we construct these: At program points that we wish to monitor, we probe the standard evaluation process just *before* and just *after* evaluation. Thus, a monitor specification defines a *pair* of functions: what we call the *pre-* and *post-monitoring functions.* Both functions receive the current monitor state and evaluation context (i.e. the expression, annotation, and semantic arguments), and they both yield an updated monitor state. But since the post-monitoring function is invoked *after* evaluation, it takes as an additional argument the intermediate result normally passed to the continuation.

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

The need for both a pre- and a post-monitoring function simply reflects the needs of specific monitors; later examples will validate this need.

3.2 My first monitor: a profiler specification

We *profile* a program by associating a counter with each function definition in the source program, and incrementing a function's counter whenever its body is about to be evaluated. To achieve this in our framework we simply annotate each function body with its own function name.

Figure 6 shows the specification of our profiler. The syntax specification defines the annotation syntax by defining the `Label` datatype. The `annotate` function, however, which maps unannotated programs into annotated ones, is left undefined; it is our way, here and throughout the rest of the paper, of ignoring the details of which expressions get annotated and how (for example, in a graphical environment a simple mouse click may induce the annotation). The monitor state is simply an association list (see Appendix A) that maps function names to counter values. Last is the specification of the monitoring functions: the pre-monitoring function increments the appropriate counter upon seeing a profiled expression, and the post-monitoring function does nothing.

The behaviour of the standard interpreters combined with this profiler is given in the next section.

A monitor package

Like standard interpreters, monitor components are packaged within a datatype:

```
data MonitorType ann ast semArgs intermediateRes monState =
  Monitor (ast->ast)                                  -- Annotate function
          (ann->ast->semArgs->monState->monState)     -- Pre function
          (ann->ast->semArgs->
                 intermediateRes->monState->monState) -- Post function
          monState                                    -- Monitor state
```

Thus, for the above profiler we define:

```
profiler :: MonitorType Label Exp semArgs kontArgs MonState
profiler = Monitor annotate pre post assocEmpty
```

4 Third Step: Combining a monitor with an interpreter

So far we have managed to speak fairly abstractly about monitors; for example, we have given the specification of a profiler independently of the standard semantics specification. But now we must show how such specifications can be combined with the standard semantics. The binary operator (&) is used for this purpose, and its definition is given in Figure 7. Note that it constructs an enhanced interpreter out of the monitor and standard interpreter datatypes by simply combining their syntactic and semantic components, as follows.

First, the enhanced parser is constructed by composing the standard parser with

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

```
module Combine where

--(&) :: (MonitorType..) -> (InterpreterType..) -> (InterpreterType..)
monitor & interpreter =
  Interpreter (annotate . parse)           -- combined parser
              (combineEval evalf pre post) -- combined val fun
              (semArgs,monState)           -- combined semantic args
              kont                         -- continuation
  where (Monitor annotate pre post monState)   = monitor
        (Interpreter parse evalf semArgs kont) = interpreter

combineEval evalFunctional preFun postFun =
 λnewEval ->
  λexp semArgs kont ->
   case exp of
    (Lxp label exp') ->
      (newEval exp' semArgs (postMonitor kont)) . preMonitor
      where preMonitor       = preFun label exp' semArgs
            postMonitor kont = λkArgs ->
                kont kArgs . postFun label exp' semArgs kArgs
    otherwise -> evalFunctional newEval exp semArgs kont
```

Fig. 7. A combine operator for an interpreter and a monitor specification

the monitor's annotate function. Then, the more difficult part, an enhanced valuation functional is synthesized by combining the standard valuation functional with the pre- and post-monitoring functions (see `combineEval`). This is the most interesting aspect of the design, and it relies intrinsically on the manipulation of functionals rather than their fixpoints. Note first that the new derived functional has the same behaviour as the standard functional (`evalFunctional`) for all expressions except those tagged with monitor annotations. Second, recall the intent that the standard interpreter communicate its dynamic context to the monitor before and after the valuation of an annotated expression. This is accomplished by composing the pre- and post-monitoring functions (`preFun` and `postFun`) with the standard functional in such a way that the standard value is passed along unchanged, whereas the monitor state is threaded and (possibly) updated by the monitoring functions. Finally, the valuation function is the fixpoint of the newly derived functional, and thus the new behaviour is exhibited at all levels of recursion, i.e. for all subexpressions of the original program.

As stated earlier, the resulting composite interpretation is a function mapping an initial monitor state to a pair: the original answer and a final monitor state. To accommodate this new behaviour we redefine the standard driver:

```
execute (Interpreter parse evalf (semArgs,monState) kontInit) prog =
        (fix evalf) (parse prog) semArgs kontInit monState
```

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

4.1 *A working profiler*

We are now ready to combine the profiler discussed in Section 3.2 with the standard interpreters and apply it to a program. Consider:

```
fact3 = "letrec mul = lambda x y . {LProfile mul}:x * y
         in letrec fac = lambda n acc .
              {LProfile fac}: if n=0 then acc
                                    else fac (n-1) (mul n acc)
            in fac 3 1"
```

For clarity, the annotations generated by the `annotate` function are explicitly included here in the source-level syntax, distinguished by curly brackets.

Let us now test our profiler on the above program:

```
Run> execute (profiler & eager) fact3
(6, [(fac,4), (mul,3)])

Run> execute (profiler & lazy) fact3
(6, [(fac,4), (mul,3)])
```

In this example the profiling results for both interpreters are the same. However, this is not always the case; for example, if we change the consequent branch in `fac` to `1` rather than `acc` (a plausible error):

```
badFact3 = "letrec mul = lambda x y . x * y
            in letrec fac = lambda n acc . if n=0 then 1
                                                else fac (n-1) (mul n acc)
               in fac 3 1"
```

then the lazy profiler result differs from the eager one because `acc` is never used:

```
Run> execute (profiler & eager) badFact3
(1, [(fac,4), (mul,3)])

Run> execute (profiler & lazy) badFact3
(1, [(fac,4)])
```

This profiler is simple enough to suit both the eager and lazy interpreters. However, not all monitor specifications can be combined with either interpreter; some will be dependent on the specific characteristics of each interpreter (for example, the eager interpreter stores identifier values in an environment while the lazy interpreter uses a store). Examples of this appear in the next section.

5 Monitors, monitors and more monitors

In this section we give several more examples of useful monitor specifications for both the lazy and eager interpreters, including:

- A tracer.
- A collecting monitor (*à la* collecting interpretations).
- A demon (event monitor).

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

```
module EagerTrace where

-- SYNTAX --

-- Tracer label: function and formal arguments names
data Label = LTrace FunName [ArgName]
annotate :: Exp -> Exp  -- omitted

-- ALGEBRAS --

-- Tracer state: list of trace messages and a trace depth counter.
type MonState   = ([TraceMsg],TraceDepth)
data TraceMsg   = Receive TraceDepth FunName [D]
                | Return  TraceDepth FunName D
type TraceDepth = Int

-- MONITORING FUNCTIONS --

pre :: Label -> exp -> Env -> MonState -> MonState
pre (LTrace funName args) exp env (traceMsgs,depth) =
  (traceMsgs ++ [Receive depth funName (map env args)], depth+1)

post :: Label -> exp -> Env -> D -> MonState -> MonState
post (LTrace funName args) exp env result (traceMsgs,depth) =
     (traceMsgs ++ [Return depth' funName result], depth')
     where depth' = depth - 1

-- A TRACER FOR EAGER --

eagerTracer :: MonitorType Label Exp Env D MonState
eagerTracer =  Monitor annotate pre post ([],0)
```

Fig. 8. A tracer for eager.

5.1 A tracer

A *tracer* is designed to report, for every traced function:

- The dynamic values of the formal parameters at each function call.
- The resulting value at each function return.

5.1.1 An eager tracer

The specification of a tracer for eager is presented in Figure 8. It is designed to collect the tracing information before and after evaluating any function body. As for the profiler, we therefore annotate each function body with a tracer annotation, but in addition to the name of the function, this syntax also includes the formal parameter names. The tracer state consists of a history of tracing data and a trace depth counter. Each piece of tracing data captures the trace depth level, the function name, and either the values of the actual arguments or the returned value.

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

Combining this tracer with the eager standard interpreter yields the expected tracing behaviour, as shown in Figure 10 for the evaluation of fact3, annotated as follows:

```
fact3 = "letrec mul = lambda x y . {LTrace mul(x,y)}:x * y
         in letrec fac = lambda n acc .
              {LTrace fac(x,y)}: if n=0 then acc
                                  else fac (n-1) (mul n acc)
            in fac 3 1"
```

The results presented in Figure 10 are 'pretty-printed', but for simplicity we have omitted the code that accomplishes this.

5.1.2 *A lazy tracer*

The definition of a lazy tracer is similar to that of the eager tracer. The only difference relates to the fact that function arguments are possibly unevaluated at call time. This is at odds with the conventional tracing strategy of printing function arguments' dynamic values at call time. We are faced with two options: either use the conventional strategy and get only the partial information about argument values that exists at call time *or* wait until the execution ends and then look up the arguments' values in the final store. The latter technique provides more information since arguments eventually used are by then already evaluated. On the other hand, accumulating such trace information constitutes a potentially large space leak. Nevertheless, we opt for this strategy because of its increased utility to the programmer.

Our lazy tracer is shown in Figure 9. Since we do not wish to force the evaluation of formal parameters at call time (thus changing evaluation order), we only determine and save their store locations. After the entire program execution is complete, we look at the saved locations in the store to determine their values. This post-interpretation processing is achieved by annotating the whole program with a special tracer label {LTrace top()}.

Figure 11 shows the evaluation of fact3 using the lazy tracer, with the output again pretty-printed; it should be contrasted with the eager results in Figure 10. The differences reflect the different evaluation orders between the lazy and strict interpreters.

Another useful feature of our lazy tracer is the way it handles unevaluated arguments. For example, consider the following program and its trace:

```
silly = "letrec baz = lambda x . x + 1 in
         letrec foo = lambda x y . baz x
         in foo 3 2"


Run> execute (lazyTracer & lazy) silly
(4, [foo receives [3,<thunk>]
     | baz receives [3]
     | baz returns 4
     foo returns 4])
```

```
module LazyTrace where

-- SYNTAX {omitted, same as in EagerTrace} --

-- ALGEBRAS --
-- Tracer state: list of trace messages and a trace depth counter.
type MonState   = ([TraceMsg],TraceDepth)
data TraceMsg   = Receive TraceDepth FunName [ValString]
                | Return  TraceDepth FunName ValString
type TraceDepth = Int

-- MONITORING FUNCTIONS --
pre :: Label -> exp -> (Env,Store) -> MonState -> MonState
pre (LTrace funName args) exp (env,store) (traceMsgs,depth) =
 case funName of
  "top"     -> (traceMsgs,depth)
  otherwise -> (traceMsgs ++ [rcvMsg], depth+1)
               where rcvMsg = Receive depth funName (map (show . env) args)

post::Label -> exp -> (Env,Store) -> (D,Store) -> MonState -> MonState
post (LTrace funName _) exp (_,_) (result,store') (traceMsgs,depth) =
 case funName of
  "top"     -> (map (postLookup store') traceMsgs, depth)
  otherwise -> (traceMsgs ++ [rtnMsg], depth')
               where rtnMsg = Return depth' funName (show result)
                     depth' = depth - 1

postLookup :: Store -> TraceMsg -> TraceMsg
postLookup store traceMsg =
 case traceMsg of
  (Receive depth funName locs) ->
        Receive depth funName (map getVal locs)
        where getVal loc = case (storeLook store (read loc)) of
                                    (Val d)   -> (show d)
                                    (Thunk t) -> "<thunk>"
  otherwise -> traceMsg

-- A TRACER FOR LAZY --
lazyTracer :: MonitorType Label Exp (Env,Store) (D,Store) MonState
lazyTracer =  Monitor annotate pre post ([],0)
```

Fig. 9. A tracer for `lazy`.

Notice that unevaluated arguments (e.g. the second argument of `foo`) remain unevaluated and are displayed as `<thunk>`s. In this way the tracer output reflects correctly the behaviour of lazy evaluation. Although the trace does not provide information about the time at which arguments are evaluated, this could be achieved by, for example, adding a counter that kept track of 'reduction steps'.

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

```
Run> execute (eagerTracer & eager) fact3
 (6, [fac receives [3,1]
      | mul receives [3,1]
      | mul returns 3
      | fac receives [2,3]
      | | mul returns [2,3]
      | | mul returns 6
      | | fac receives [1,6]
      | | | mul receives [1,6]
      | | | mul returns 6
      | | | fac receives [0,6]
      | | | fac returns 6
      | | fac returns 6
      | fac returns 6
      fac returns 6 ])
```

Fig. 10. Tracing eager evaluation of fact3.

```
Run> execute (lazyTracer & lazy) fact3
 (6, [fac receives [3,1]
      | fac receives [2,3]
      | | fac receives [1,6]
      | | | fac receives [0,6]
      | | | | mul receives [1,6]
      | | | | | mul receives [2,3]
      | | | | | | mul receives [3,1]
      | | | | | | mul returns 3
      | | | | | mul returns 6
      | | | | mul returns 6
      | | | fac returns 6
      | | fac returns 6
      | fac returns 6
      fac returns 6 ])
```

Fig. 11. Tracing lazy evaluation of fact3.

5.1.3 Non-termination

What happens when a program fails to terminate? In the case of our lazy tracer, the final store is unattainable and no trace will be produced. This problem can be solved by using the tracing approach based on partial information discussed earlier. In particular, we note that this problem is not inherent in our framework, even though it seems that the final result contains both the standard answer and the monitor state. One can inspect portions of the monitor state even though the program may not have terminated; as long as the specific monitor data does not rely on the final state or answer, there is no problem. In practice, we rely on the non-strict semantics of Haskell to make this possible, and we later exploit this attribute in the construction of interactive monitors (see Section 7).

5.1.4 Tracing and profiling any expression

The asture reader will have noted that both the profiler and tracer are general enough to profile and trace *any* expression, not just function bodies. This can be done by attaching a label to the expression to be monitored; in the case of the tracer, the 'formal parameters' then correspond to free variables in the expression, thus allowing one to display any desired portion of the lexical environment. For example:

```
letrec mul = lambda x y. if (x == 0)
                         then {LTrace mulTrue(x,y)}:0
                         else {LTrace mulFalse(x,y)}:(y+(mul (x-1) y))
in mul 2 3
```

The eager tracer returns:

```
Run> execute (eagerTracer & eager) mult
(6, [mulFalse receives [2,3]
     | mulFalse receives [1,3]
     | | mulTrue receives [0,3]
     | | mulTrue returns 0
     | mulFalse returns 3
     mulFalse returns 6)
```

5.2 A Collecting Monitor

A *collecting interpretation of expressions* (or what we will call a 'collecting monitor') is an interpretation of a program that answers questions of the sort: 'What are all possible values to which an expression might evaluate during program execution?' (Hudak and Young, 1988). A collecting monitor for eager is shown in Figure 12. The collecting monitor for lazy shares almost exactly the same code, the difference being only in the post function which, for the lazy interpreter, receives a pair (i.e. the result and an updated store) as intermediate results:

```
post :: Label -> exp -> semArgs -> (D,Store) -> MonState -> MonState
post (LCollect ide) exp semArgs (result,store') collectEnv =
  assocPut ide (setInsert result vs) collectEnv
  where vs = if   (assocExist ide collectEnv)
             then (assocGet ide collectEnv)
             else setEmpty
```

For a test run, consider again badFact3, but this time with the following annotations:

```
letrec mul = lambda x y . x * y in
letrec fac = lambda n acc . if {LCollect test}:(n=0) then 1
                            else fac (n-1) (mul {LCollect n}:n acc)
in fac 3 1"
```

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

```
module EagerCollect where

-- SYNTAX --
-- Label Syntax: expression tags
data Label = LCollect IdeName
annotate  :: Exp -> Exp -- omitted

-- ALGEBRAS --
-- Collecting monitor state: association list of tags to dynamic values
type MonState = AssocType IdeName (SetType D)

-- MONITORING FUNCTIONS --
pre :: Label -> exp -> semArgs -> MonState -> MonState
pre _ _ _ state = state

post :: Label -> exp -> semArgs -> D -> MonState -> MonState
post (LCollect ide) exp semArgs result collectEnv =
  assocPut ide (setInsert result vs) collectEnv
  where vs = if   (assocExist ide collectEnv)
             then (assocGet ide collectEnv)
             else setEmpty

-- A COLLECTING MONITOR --
eagerCollector = Monitor annotate pre post assocEmpty
```

Fig. 12. A collecting monitor for eager.

The results of both collecting monitors in action are compared below:

```
Run> execute (eagerCollector & eager) badFact3
(1, [(test,[True,False]), (n,[1,2,3])])

Run> execute (lazyCollector & lazy) badFact3
(1, [(test,[True,False])])
```

Note again how the reduction strategy affects the monitoring result.

5.3 Event monitoring via demons

Often a programmer wants to invoke monitoring actions only if a specific execution event (such as an identifier assuming a negative value) occurs. A simple mechanism called a *demon* is proposed in (Delisle *et al.*, 1984) for such a purpose, where an interactive environment for Pascal called Magpie is described. We show in this section how to specify demons within our framework. Whereas Magpie allows monitoring events associated with a particular identifier, we show how to specify demons for essentially any semantic event.

To specify a demon in our framework, one first annotates all program points where an event might occur. Then one specifies the semantic conditions under which an action is to be triggered. Finally, the actions themselves are specified in the

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

```
module ForceFind where

-- SYNTAX --
data Label = LDemonInfo FunName -- Function name
           | LDemonTag          -- Force-finder tag
annotate   :: Exp -> Exp        -- omitted

-- ALGEBRAS --
-- Force finder state: current function name,
--                     and function name where force occurred
type MonState = (FunName,FunName)

-- MONITORING FUNCTIONS --
pre :: Label -> exp -> (Env,Store) -> MonState -> MonState
pre (LDemonInfo fn') semArgs (env,store) (fn,where) = (fn',where)
pre (LDemonTag)      semArgs (env,store) (fn,where) = (fn,fn)

post::Label -> exp -> (Env,Store) -> (D,Store) -> MonState -> MonState
post _ _ _  _ monState = monState

-- A "DEMON" --
lazyDemon :: MonitorType Label Exp (Env,Store) (D,Store) MonState
lazyDemon  = Monitor annotate pre post ("<void>","<no force>")
```

Fig. 13. A force-finder demon for `lazy`.

monitoring functions. As an example, Figure 13 shows the specification of a demon that determines at what point an expression is ‘forced’ under lazy evaluation. The technique is simple: we use two labels, one to inform the monitor of the currently executing function name, and the other to identify the monitored expression. We know that the pre-monitoring function will be called *only* when this expression is forced, therefore we just update this information when such a force occurs.

As an example, consider the following annotated program:

```
silly = "letrec baz = lambda x . {LDemonInfo baz}:(x + 1) in
         letrec foo = lambda x y . {LDemonInfo foo}:(baz x)
         in foo {LDemonTag}:(2+1) (1+1)"
```

The demon will return `baz` indicating that `(2+1)` was forced at `baz`. However, if we annotate the second argument to `foo` then the result would be `<no force>`.

Clearly demons may be defined for intercepting a wide spectrum of semantic events, and one could argue that this capability should be given to the user. This could be accomplished either by exposing our entire framework to the user, or by exposing a form tailored to this demon-specification process. In practice, this capability is quite useful in discovering ‘hard-to-find’ bugs.

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

5.4 A comment on modularity

How much does the monitor implementor need to know about the standard interpreter implementation? And conversely, how much does the standard interpreter implementor need to know about the monitor specification? The answer to both questions is: very little. Both implementors need only exchange the interface functionality of their respective programs; i.e. the dynamic information that will be passed between the standard interpreter and the monitor (e.g. environment, store, monitor state, etc.). Although Figure 1 in the introduction implies that the same monitor can be used with different languages, in fact this is not always true. But we feel that the changes are usually so small that it is virtually true: most importantly, the degree of code reuse is very high.

6 Composing Monitors

Recall that the combining operator (&) takes an interpreter and a monitor and returns a monitored interpreter. By treating the result as a new interpreter, we should be able to repeat the same procedure and compose a second monitor to the system. To support this, however, we must adjust our design slightly, as follows:

1. **Disjoint labels.** Each monitor must have a unique set of monitor labels distinguishable from other monitors' labels.
2. **Multiple monitor states.** Instead of propagating a single monitor state we will be passing a collection of monitor states, one for each monitor in the system.

Figure 14 presents a combine operator which supports multiple monitors. Note that each monitor now has a unique name and thus its label and state can be easily identified. The monitor states are stored in an association list that maps monitor names to their state. The combine operator will look up the appropriate monitor state in this environment and update it using the corresponding monitoring function (see updMonEnv.)

Using the new operator we can now compose monitors freely:

```
Run> execute (eagerTracer & profiler & eager) fact3
(6, [(profile, [(fac,4), (mul,3)])
     (trace, fac receives [3,1]
             | mul receives [3,1]
             | mul returns 3
             | fac receives [2,3]
             and so on...
```

This composition may be repeated an arbitrary number of times. Since a monitor can only modify its own arguments, we maintain a high degree of safety such that new monitors cannot change the behaviours of other monitors.

7 Interactive monitoring

No doubt, source-level interactive debugging is one of the most important components of a program development environment. Yet rarely are formal specifications of

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

```
module MultiCombine where

-- (&)::(MonitorType..) -> (InterpreterType..) -> (InterpreterType..)
monitor & interpreter =
 Interpreter (annotate . parse)                    -- combined parser
             (combineEval evalf monName pre post) -- combined val fun
             (semArgs,monEnv')                     -- combined semantic args
             kont                                  -- continuation
 where (Monitor monName annotate pre post monState) = monitor
       (Interpreter parse evalf (semArgs,monEnv) kont) = interpreter
       monEnv' = assocPut monName monState monEnv

combineEval evalFunctional monName preFun postFun =
 λnewEval ->
  λexp semArgs kont ->
   case exp of
    -- Note that labels now include the monitor name
    (Lxp monName' label exp') | monName' == monName ) ->
     (newEval exp' semArgs (postMonitor kont)) . preMonitor
     where preMonitor       = updMonEnv monName (preFun label exp' semArgs)
        postMonitor kont = λkArgs ->
          kont kArgs . updMonEnv monName (postFun label exp' semArgs kArgs)
    otherwise -> evalFunctional newEval exp semArgs kont

updMonEnv :: MonName -> (MonState -> MonState) -> MonEnv -> MonEnv
updMonEnv monName transtate monEnv =
  monEnv'
  where monState     = assocGet monName monEnv
        monState'    = transtate monState
        monEnv'      = assocPut monName monState' monEnv
```

Fig. 14. A modular combine operator.

such debuggers given. This is partly because debuggers tend to evolve into large software projects in which the engineering details overshadow the conceptual content. Fortunately, using monitoring semantics the high-level specification of an effective debugger can be easily presented.

The approach that we take is based on the 'stream model' of purely functional I/O as used by languages such as Haskell (Hudak *et al.*, 1992). In this model a program communicates to the outside world via *streams of messages*: a program issues a stream of *requests* to the operating system and in return receives a stream of *responses*. Thus a program is a function with type Dialogue, given by:

```
type Dialogue = [Response] -> [Request]
```

where [Response] is an ordered list of *responses* and [Request] is an ordered list of *requests*; the *n*th response is the operating system's reply to the *n*th request.

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

7.1 Interactive answer algebra

To adopt this model for our use, we first change the answer algebras to reflect the new I/O behaviour:

```
module IOEagerAnswer where

type Ans = MonState -> InChan -> (InChan,OutChan)

toAns :: D -> Ans
toAns (Num v) = λmonState inChan ->
                    (inChan,"the result is:" ++ (show v) ++ "\n")
```

and:

```
module IOLazyAnswer where

type Ans = MonState -> InChan -> (InChan,OutChan)

toAns :: (D,Store) -> Ans
toAns (Num v,store) = λmonState inChan ->
                          (inChan,"the result is:" ++ (show v) ++ "\n")
```

Note that the new 'final answer' is still a function, but it takes as an additional argument an input stream, and it returns a pair: the remainder of the input stream and the output stream (the standard answer is converted into a string and incorporated in the output stream).

We also define a top-level function to communicate with the operating system. This function receives an interpreter and a program and returns a Dialogue. The first request in the dialogue is a request for an input stream; in response the operating system returns (Str userInput), the desired stream. The rest of the dialogue is a list of output requests (i.e. printouts) by the system.

```
-- main :: (InterpreterType..) -> ProgText -> Dialogue
main interpreter prog =
 λrsps -> ReadChan stdin : map (AppendChan stdout) (linesln out)
          where (in,out) = execute interpreter prog userInput
                (Str userInput) = head rsps
                -- linesln chops the output stream to a list of lines
                linesln out = map (λline -> (line ++ "\n")) (lines out)
```

Using the factorial of Section 2.2, we can perform an initial test run:

```
Run> main eager fact3
the result is: 6

Run> main lazy fact3
the result is: 6
```

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

7.2 Interactive monitor

Similarly, we change the functionality of a monitor to correspond to the new I/O behaviour: the interactive monitoring functions will receive an input stream as an additional argument and return a triple – the rest of the input stream, an output stream (monitor messages), and the updated monitor state.

```
data IOMonitorType ann ast semArgs intermediateRes monState =
 IOMonitor (ast->ast)                             -- Annotate function
           (ann->ast->semArgs->monState
             ->InChan->(InChan,OutChan,monState)) -- Pre function
           (ann->ast->semArgs->intermediateRes->monState
             ->InChan->(InChan,OutChan,monState)) -- Post function
           monState                               -- Monitor state

type InChan  = String  -- Input Channel
type OutChan = String  -- Output Channel
```

7.3 Combining an interactive monitor with an interpreter

As in Section 4, we define a combine operator for interactive monitor and interpreter specifications. However, unlike before we extend the behaviour of the new monitoring interpreter to include 'single stepping', i.e. breaking at every single interpretive step, which is necessary in certain debugging contexts. This is implemented by calling the monitoring functions before and after *every* interpretive step (i.e. every continuation) with a special label LBreak. Except for this single stepping and the propagation of I/O streams, the resulting interactive combine operator shown in Figure 15 is similar to the one given in Section 4.

7.4 An interactive source-level debugger

In Appendix B we present the actual high-level specification of a source-level debugger for the languages presented in Section 2. Note that:

- Like all monitors, the debugger has its own syntax, algebras and monitoring functions.
- The design is for a toy debugger, but it has most of the essential features of real debuggers: break-points, single command stepping, recursive evaluator and debugger, and several other informative commands.
- Only minor changes (7 lines of code!) are necessary to transform the eager debugger into a truly lazy one. The lazy debugger is of course consistent with lazy evaluation. In particular, delayed expressions (i.e. thunks) are not forced, yet can be 'peeked into' using the recursive debugger; when the recursive session is complete, all subexpressions return to their original thunk status.

Interactive monitors composition As a final comment, we note that interactive monitors can be composed similarly to other monitors. By composing two interactive monitors they share the same input and output streams.

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

```
module IOCombine where

ioMonitor & interpreter =
     Interpreter (annotate . parse) (combineEval evalf name pre post)
                 (semArgs,monState) kont
     where (IOMonitor annotate pre post monState) = ioMonitor
           (Interpreter parse evalf semArgs kont) = interpreter

combineEval evalFunctional preFun postFun =
 λnewEval ->
  λexp semArgs k ->
   case exp of
    (Lxp label exp') -> eval <=> preMonitor
      where preMonitor    = preFun label exp' semArgs
            eval          = newEval exp' semArgs (postMonitor k)
            postMonitor k =
              λkArgs -> k kArgs <=> postFun label exp' semArgs kArgs
    otherwise -> eval <=> preMonitor
      where preMonitor = preFun LBreak exp semArgs
            eval       = evalFunctional newEval exp semArgs (postMonitor k)
            postMonitor k =
              λkArgs -> k kArgs <=> postFun LBreak exp semArgs kArgs

(<=>) ::    (monState -> inChan -> (inChan, OutChan))
         -> (monState -> inChan -> (inChan, OutChan, monState))
         -> (monState -> inChan -> (inChan, OutChan))
evalActivity <=> monitorActivity =
 λmonState input -> (input', output' ++ output')
   where (input',output',monState') = monitorActivity monState input
         (input',output')         = evalActivity monState' input'
```

Fig. 15. Combining an interactive monitor with an interpreter.

8 Optimization via partial evaluation

Partial evaluation is a program transformation technique for specializing a program with respect to some known part of its input. The resulting *residual program* has the property that when applied to the remaining part of the input, it will yield the desired result. Since many problems can be shown to be ‘specializations’ of a more general problem, this provides the basis for a simple, automatic, formal-methods approach to program development.

The most widely studied application of partial evaluation is *semantics-directed compilation*, and our use can be seen as a variation of that idea. As was reported by Lee (1989), many semantics-directed methodologies generate systems with performance characteristics that are several orders of magnitude worse than those exhibited by handwritten techniques. Partial evaluation can alleviate this problem. The work of Safra and Shapiro (1989) can be seen as most closely related to ours.

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

8.1 Introduction

When reasoning about partial evaluation, it is important to distinguish between a *program* (a syntactic object) and the *function* that the program denotes (a mathematical object). Notationally we name programs using large capitals, as in P or *Int*, and the functions they denote using small capitals, as in p or *int*. We move freely between programs and their denotations, knowing that the mapping is implicitly defined by the language's formal semantics, but we also assume that there exists an evaluator *eval* such that:

$$eval(P,D) = p(d)$$

where D is the program's input data.

A *partial evaluator* is a program PE that, given a program P and one input argument V, computes a *residual program* P_V which, when applied to a second argument W, returns the desired result. We can write this as:

$$\begin{aligned}&eval(P_V, W) \equiv eval(P,(V,W))\\ &\text{where } P_V = eval(1,(P,V))\end{aligned}$$

although extensionally it is perhaps clearer to write as:

$$p_v(w) \equiv p(v,w) \qquad \text{where } p_v = 1(p,v)$$

We say that P_V is a *specialized* version of P with respect to the argument V.

At the time of this research, the best available partial evaluators were for pure Lisp (or related dialects), and thus for our experiments we chose *Schism*, a partial evaluator for pure Scheme. The version of Schism that we used was written in T, a close dialect of Scheme. The Haskell specifications were translated into Scheme, partially evaluated, and then benchmarked under the Orbit implementation of Scheme/T (Kranz, 1988; Kranz *et al.*, 1986). However, for continuity of exposition, we have translated the residual Scheme programs back into Haskell for inclusion in this paper.

8.2 Partial evaluation of monitoring semantics

One can view our overall system as a 'meta-level interpreter' **ML-Interpreter** which takes as arguments a standard interpreter, a monitor specification, a program, and the program's inputs, and returns a standard value together with monitoring data. Thus **ML-Interpreter** has functionality:

$$\textbf{ML-Interpreter} \;:\; \textbf{Interpreter} \times \textbf{Monitor} \times \textbf{Program} \times \textbf{Input} \;\rightarrow\; (\textbf{Answer}, \textbf{MonInfo})$$

Application of **ML-Interpreter** can be optimized using partial evaluation at two levels of specialization, as described in the following two subsections and illustrated in Figure 16.

8.2.1 Level 1 Specialization: instrumented interpreter

Specializing the meta-level interpreter with respect to its first two arguments, the standard interpreter and the monitor specification, automatically yields an *instru-*

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

System Functionality:

$$\underbrace{\text{Interpreter} \times \text{Monitor}}_{\mathcal{PE}} \times \text{Program} \times \text{Input} \rightarrow (\text{Answer,MonInfo})$$

Specializing the interpreter w.r.t. monitor.

$$\underbrace{\underline{\textit{Instrumented-Interpreter}} \times \text{Program}}_{\mathcal{PE}} \times \text{Input} \rightarrow (\text{Answer, MonInfo})$$

Specializing the instrumented interpreter w.r.t. a program [Safra & Shapiro] .

$$\underline{\textit{Instrumented-Program}} \times \text{Input} \rightarrow (\text{Answer, MonInfo})$$

Fig. 16. Partial evaluation optimization levels.

mented interpreter, i.e. an interpreter instrumented with monitoring actions:

$$\underbrace{\textit{Instrumented--interpreter}}_{PE(\textbf{ML-Interpreter},(\textbf{Interpreter},\textbf{Monitor}))} \times \textbf{Prog} \times \textbf{Input} \rightarrow (\textbf{Ans} \times \textbf{MonInfo})$$

This step removes the interpretive overhead of the meta-level interpretation.

As an example, the Haskell code in Figure 17 is the residual program resulting from the above specialization process, where **Interpreter** = `eager`, **Monitor** = `eagerTracer`,[§] and **ML-Interpreter** is a straightforward interpreter for the language in which `eager` and `eagerTracer` are written (in our case Scheme). Note how partial evaluation has interlaced the tracer functionality into the interpreter and evaluated its static components. The resulting specialized interpreter has the same behaviour as the standard interpreter for all expressions except those labelled with monitor annotations.

8.2.2 Level II Specialization: instrumented program

Specializing the *instrumented interpreter* of the previous section with respect to a *source program* removes the second level of interpretive overhead (that associated with monitoring), yielding now an *instrumented program* in which the extra code to perform monitoring actions has been automatically embedded into the program:

$$\underbrace{\textit{Instrumented--program}}_{PE(\textbf{Instrumented-interpreter},\textbf{Program})} \times \textbf{Input} \rightarrow (\textbf{Ans} \times \textbf{MonInfo})$$

For some examples, we specialize the instrumented eager-tracer from the last section with respect to a factorial program and a power-of-two program, as shown

[§] For simplicity we have eliminated the bookkeeping of the trace depth from the specification.

```
eagerEvalf :: Functional (Exp -> Env -> Kont -> Ans)
eagerEvalf eval =
 λexp env k ->
   case exp of
    (Con v)        -> k (Num v)
    (Var id)       -> k (env id)
    (Abs id e1)    -> k (Fun (λv -> eval e1 (envUpd env id v)))
    (Cnd e1 e2 e3) -> eval e1 env (λ(Bol v) -> if v
                                                  then (eval e2 env k)
                                                  else (eval e3 env k))
    (Bop id e1 e2) -> eval e1 env
                         (λ v1 -> eval e2 env
                                    (λ v2 -> (k (applyBop id v1 v2))))
    (App e1 e2)    -> eval e1 env
                         (λ(Fun f) -> eval e2 env (λv -> f v k))
    (Let id e1 e2) -> eval e1 env (λv -> eval e2 (envUpd env id v) k)
    (Rec id (Abs arg body) e2) ->
      eval e2 env' k
      where env' = envUpd env id (Fun closure)
                   where closure v = eval body (envUpd env' arg v)
    {- all equations above are the same as in the standard interpreter -}
    (Lxp (fn,args) e1) ->
       λtraceMsgs ->
          eval e1
               env
               (λv traceMsgs' -> k v (traceMsgs'++[Return fn v]))
               (traceMsgs++[Receive fn (map env args)])
```

Fig. 17. Instrumented eager for tracing.

in Figures 18 and 19, respectively. Each of these figures shows the original program, a hand-crafted program instrumented according to O'Donnell and Hall's (1988) methodology, and the instrumented program produced by our methodology.

The first thing to note in these figures is that our instrumented programs are in continuation-passing style (CPS) while the corresponding handwritten programs are in direct style. That ours is in CPS should come as no surprise, since partial evaluation 'compiles' the source program according to the whims of the interpreter, which in our case is itself in CPS (Figure 17). Note also that O'Donnell and Hall's program propagates the monitoring information by enhancing a function's arguments and result with debugging information (which they call *shadow variables*), whereas our instrumented programs use higher-order functions and continuations to achieve the same effect. Higher-order functions are probably more expensive than O'Donnell and Hall's shadow variables, but their technique propagates the monitoring information globally, whereas we use higher-order functions to update the monitor information only when a monitoring activity is required.

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

```
fac n = if (n == 0)
        then 1
        else n * (fac (n-1))
```

(a) Factorial standard function.

```
fac n =
  fac' n []
  where
    fac' n debin =
       if   (n == 0)
       then (1, debin++[(Receive "fac" [1]),(Return "fac" 1)])
       else (result, fdeb++[Return "fac" result])
            where
              result      = n * fres
              (fres,fdeb) = fac' (n-1) (debin ++ [Receive "fac" [n]])
```

(b) Traced factorial (O'Donnell and Hall).

```
fac n =
  fac' (λv traceMsgs -> (v,traceMsgs)) n n []
  where
    fac' k n x =
     λtraceMsgs ->
           ((if (x == 0)
             then kPost 1
             else fac' (λv -> kPost (x*v)) n (x-1))
            (traceMsgs ++ [Receive "fac" [x]]))
     where kPost v traceMsgs = k v (traceMsgs ++ [Return "fac" v])
```

(c) Monitoring semantics instrumented code.

Fig. 18. Comparison of instrumented factorial programs.

9 Performance measurements

In this section we evaluate the performance of our implementation and compare it to the performances of other implementation techniques. We also measure the optimizations gained by specializing the system with respect to specific programs and monitors.

9.1 Methodology

All experiments were run on a SUN Sparc station model 4/60fgx with a 20 MHz clock, 16 Megabytes of main memory, a hard disk with an average seek time of 22 milliseconds, and running SunOS Unix Release 4.1.1.

From our Haskell specifications we derived Scheme implementations of the meta-level interpreter (Section 8.2), the standard interpreter (Figure 17), a set of monitor specifications (namely a profiler, a tracer and a debugger), and a suite of benchmark programs (described in the next section).

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

For each of the monitor specifications, we generated an instrumented monitor as described in Section 8.2.1. Then for each instrumented monitor and for each benchmark program we generated an instrumented program as described in Section 8.2.2. All resulting programs were compiled and run as Scheme programs. In our benchmark programs, we tried to minimize runtime overhead by using type-specific arithmetic and keeping storage allocation to a minimum. For the same reason, all our measurements exclude the time for garbage collection. In each test run the benchmark was iterated such that it would execute for a fixed processor time (50–100 s). The number of iterations was typically in the hundreds and above. The reported results are averages over all iterations.

9.1.1 Benchmark programs

The programs comprising the benchmark suite are:

- `fac`: Integer factorial. The standard recursive factorial program to calculate the nth factorial ($n = 12$).
- `power2`: Integer power of 2, i.e. 2^n. A log n recursive program to compute 2^n ($n = 28$).
- `deriv`: Symbolic derivation. A straightforward recursive program for the symbolic derivation of polynomials. The polynomials are represented symbolically as a list and the result is a list representing the derivative of the input. The expression used in this experiment is: $3x^2 + ax + 2x + 5$.
- `qsort`: The standard recursive quicksort routine using lists. We measured the optimization for small lists because of the overhead of running the unoptimized system for larger lists.
- `nsqrt`: Floating-point square root using Newton's method ($n = 3.0$, margin of error $\epsilon = 1e{-}6$). This program is the only floating-point program in the suite, and has the usual overheads associated with floating-point computations.

9.2 Benchmark results

In this section we present performance measurements for the following quantities:

- Partial evaluation time.
- Partial evaluation speedup.
- Comparison of instrumented interpreters.
- Comparison of instrumented programs.

9.2.1 Execution time for partial evaluation

Is it computationally expensive to instrument interpreters and programs using partial evaluation? For the level I specialization in Section 8.2.1, specializing the meta-level interpreter with respect to the eager interpreter and eager tracer took 17 seconds; similar results were obtained for the other instrumented interpreters. Since we do not expect the user to re-specialize her interpreter frequently, and because, as we

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

```
power2 n =
   if (n == 0) then 1
               else if isEven n
                    then sqr (power2 (n ‘div‘ 2))
                    else 2 * power2 (n - 1)
   where isEven n = (2 * (n ‘div‘ 2)) == n
         sqr x = x * x
```

(a) Power of 2 standard function.

```
power2 n =
  if (n==0) then (1,1)
            else if (isEven n)
                 then (sqr fres, fdeb+1)
                      where (fres,fdeb) = power2 (n ‘div‘ 2)
                 else (2*fres, fdeb+1)
                      where (fres,fdeb) = power2 (n-1)
  where isEven n = (2 * (n ‘div‘ 2)) == n
        sqr x = x * x
```

(b) Profiled power of 2 (O'Donnell and Hall).

```
power2 n =
   power2' (λv fdeb -> (v,fdeb)) data data 0
   where
    power2' kont fdeb n =
     λfdeb ->
      ((if (n == 0)
        then kont 1
        else if ((2 * (n ‘dev‘ 2)) == n)
             then power2' (λv (kont (v*v))) data (n ‘div‘ 2)
             else power2' (λv -> kont (2*v)) data (n-1))
       (fdeb+1))
```

(c) Monitoring semantics instrumented code.

Fig. 19. Comparison of instrumented power-of-2 programs.

will see later, this optimization entails speedups of up to 50 times, the overhead of this task seems reasonable.

For the level II specialization in Section 8.2.2, the following listing shows partial evaluation times (broken down into binding-time analysis (BTA) and specialization times) for various benchmark programs. As can be seen, it takes 7–16 seconds to instrument the benchmark programs (size ranging from three lines for `fac` to about 30 lines for `deriv`). This overhead should be weighed against the resulting increase in performance, which in some cases is a 70-fold decrease in execution time (see Section 9.2.2).

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

Program	Activity	BTA (sec.)	Specialization (sec.)
$PE(\langle$*instrumented interpreter*$\rangle$, `fac`)	*tracing*	*4.45*	*2.24*
$PE(\langle$*instrumented interpreter*$\rangle$, `power2`)	*profiling*	*4.11*	*3.63*
$PE(\langle$*instrumented interpreter*$\rangle$, `deriv`)	*profiling*	*4.11*	*11.96*
$PE(\langle$*instrumented interpreter*$\rangle$, `qsort`)	*tracing*	*4.45*	*3.87*
$PE(\langle$*instrumented interpreter*$\rangle$, `nsqrt`)	*tracing*	*4.45*	*3.85*

9.2.2 *Partial evaluation speedup*

The following table compares the speedups gained by partial evaluation for the benchmark programs:

Program	Unoptimized system (ms)	Instrumented interpreter (ms)	Instrumented program (ms)	Total speedup
`fac`	*478.42*	*11.20 (×43)*	*0.69 (×16)*	*×693*
`power2`	*568.17*	*14.17 (×40)*	*0.34 (×42)*	*×1671*
`deriv`	*2642.00*	*61.53 (×40)*	*0.88 (×70)*	*×2797*
`qsort`	*1554.50*	*36.82 (×42)*	*2.34 (×16)*	*×664*
`nsqrt`	*494.00*	*12.08 (×41)*	*1.16 (×10)*	*×425*

The listing shows the execution times of the benchmark programs in the unoptimized system, the instrumented interpreter level, and the instrumented program level. Each optimization removes one level of interpretation which results in the speedup shown in parentheses. Every interpretation level contributes a slowdown of about 15–70 times. By removing these levels of interpretation using partial evaluation, the speedup gained is up to three orders of magnitude (the largest speedup being 2797). These results dramatically reveal the advantage of partial evaluation.

9.2.3 *Comparison of instrumented interpreters*

How well do our instrumented interpreters compare with other interpreters? Figure 20 compares the performance of our interpreter for the execution of the benchmark programs with:

- **Standard interpreter.** A conventional hand-written Scheme interpreter (Abelson and Sussman, 1985), written in Scheme.
- **Hand-crafted monitored interpreter.** An instrumented version of the above interpreter which propagates monitoring information through function parameters (similar to O'Donnell and Hall's method).

Notice that our automatically generated instrumented interpreter is 20–80% slower than the standard interpreter and 10–50% slower than a hand-crafted monitoring

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

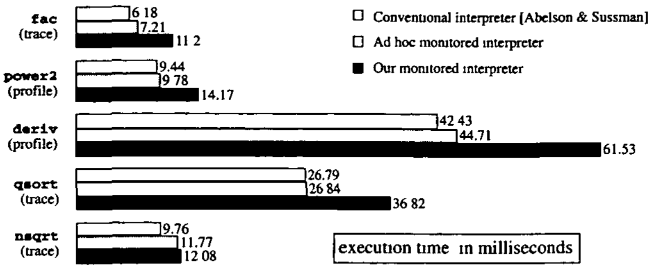
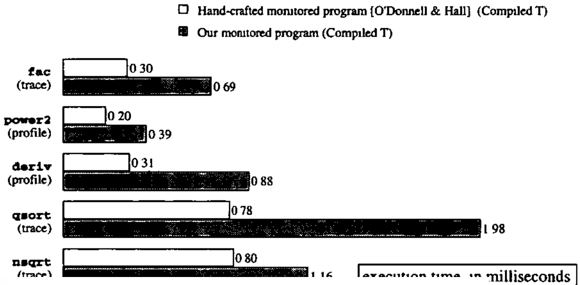
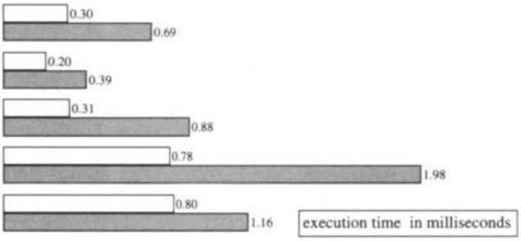


Fig. 20. Interpreters compared.





d.

interpreter. Tl ainly by the additional monitoring ac ıd the CPS form of our monitoring interpreter (*vis-à-vis* the hand-crafted monitoring interpreter and the standard interpreter which are specified in direct style). The overhead of CPS is explored further in Section 9.2.4.

9.2.4 Comparison of instrumented programs

In this section we compare the performance of our automatically instrumented programs with respect to handwritten instrumented programs.

Monitoring semantics versus O'Donnell and Hall. Figure 21 compares our instrumented benchmark programs with corresponding hand-crafted instrumented programs written using O'Donnell and Hall's technique (for actual code of some of the instrumented programs see Section 8.2.2). Our instrumented programs run about 1.5–2.8 times slower than their corresponding handwritten programs. This is probably a result of the inherent CPS style of our code.

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

Overhead of CPS. To assess the potential inefficiency of programs written in CPS, the following compares some of our instrumented programs with the corresponding non-monitored programs written in Scheme in both direct and CPS style.

Program	Direct (ms)	CPS (ms)	Ours monitored (ms)
fac	*0.12*	*0.20*	*0.69*
qsort	*0.22*	*0.56*	*1.98*
nsqrt	*0.58*	*0.72*	*1.16*

Our instrumented programs are about 2–9 times slower than the standard Scheme program written in direct style, and 1.6–3.5 times slower than the corresponding CPS programs. However, this is an 'unfair' comparison since our programs are also performing the monitoring activity. Nevertheless, these numbers can provide a feel for the performance expected from our fully optimized system.

We also point out that the Orbit compiler performs a CPS transformation as part of its normal compilation process, and thus our programs essentially undergo a double CPS conversion! Note from the table that the CPS programs are about 1.3–2.5 slower than their direct style counterparts, a clear indication of the penalty of the double conversion. As we discuss later in Section 10.2.1, an obvious direction to explore is to interface our output directly with the intermediate form of Orbit (or similar CPS-based compiler).

Monitoring semantics versus conventional systems. To assess our system's performance compared to conventional facilities, we compared our tracer with the one used in the Orbit/T system, which performs tracing by redefining a function so that it prints tracing information at call and return time. Since the Orbit/T tracer only works on interpreted code, for fairness we ran our programs interpreted as well. The following listing presents the results:

Program	Orbit/T traced (ms)	Ours traced (ms)
fac	*39.70*	*25.81*
qsort	*126.40*	*59.6*
nsqrt	*541.30*	*441.86*

Note that our instrumented programs perform 1.2–2.0 times better than the Orbit/T technique. This is probably because in the Orbit/T system, every function call goes first through a function which prints the tracing messages and then calls the original function, whereas partial evaluation has 'in-lined' our capability into the function itself. The Orbit/T technique is also limited since it is based on redefinition of top level functions, whereas our technique allows tracing local definitions. Finally,

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

our approach allows compilation of the instrumented programs, thus providing an additional order of magnitude speedup.

10 Related and future work

The work presented in this paper, to the best of our knowledge, is one of the first attempts to provide a *general* and *formal* framework for monitoring. However, several ideas in programming language research has influenced or compares closely with our work:

- O'Donnell and Hall's (1988; 1985) work is notable, and is closely related to ours in its advocation of program instrumentation for debugging purposes. Our work improves on theirs in that monitoring semantics provides a higher degree of modularity and abstraction, monitors are derived automatically rather than hand-crafted, monitors are compositional and proven not to interfere with the standard semantics, and we have explored a broader range of monitors.
- Shapiro's work on the use of enhanced meta-circular interpreters for debugging Prolog programs (Shapiro, 1982; Sterling and Shapiro, 1986) was the first to note the power of meta-circular interpreters for debugging purposes.
- Safra and Shapiro's (1989) work, which advocates the use of partial evaluation to instrument programs with monitoring activities. Our work has much in common with theirs, but many of the details are different because of the differences between logic and functional programming. Our work extends theirs conceptually in at least one important way: the use of partial evaluation to generate instrumented interpreters, not just instrumented programs.
- Various efforts at using functionals to specify semantics, in particular semantics involving notions of *inheritance* (Cook and Palsberg, 1989; Hudak and Young, 1988; Reddy, 1988). To the best of our knowledge, the monitoring semantics framework is unique in providing an extensive exploration of this technique as a method to enhance semantics.

There are several avenues of future research stemming from the results in this paper, as outlined below.

10.1 'Destructive' monitoring

We can classify monitoring activities into two major categories: *neutral* and *non-neutral* activities. Neutral activities only measure or determine the values of certain parameters associated with the dynamic state of the evaluation. On the other hand non-neutral monitoring activities aim at changing standard program behaviour to obtain debugging information (e.g. modifying run time values of certain arguments).

Though such capability obviously sacrifices the correctness property, in practice non-neutral monitoring activities are widely used. The current design of our system only allows monitoring functions to modify their own monitor state. If we would like monitors to change program behaviour we must provide the monitoring functions with more freedom.

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

Changing runtime values For example, to enable the change of identifier values we could allow monitoring functions to modify the environment. Thus, instead of returning just their updated state, monitoring functions would also return an updated environment. Such non-neutral monitoring functions would have functionality:

$$\begin{aligned} \mathcal{M}^{\mathcal{E}}_{pre} &: \mathbf{Ann} \to \mathbf{Exp} \to \mathbf{Env} \to \mathbf{MS} \to (\mathbf{MS} \times \mathbf{Env}) \\ \mathcal{M}^{\mathcal{E}}_{post} &: \mathbf{Ann} \to \mathbf{Exp} \to \mathbf{Env} \to \mathbf{D} \to \mathbf{MS} \to (\mathbf{MS} \times \mathbf{Env}) \end{aligned}$$

Continuations and reverse execution Another very interesting direction is integrating the monitor more closely with the continuation passing style, so that the monitoring functions would receive the continuation as an argument:

$$\begin{aligned} \mathcal{M}^{\mathcal{E}}_{pre} &: \mathbf{Ann} \to \mathbf{Exp} \to \mathbf{Env} \to \mathbf{MS} \to \mathbf{Kont} \to \mathbf{Ans}_{mon} \\ \mathcal{M}^{\mathcal{E}}_{post} &: \mathbf{Ann} \to \mathbf{Exp} \to \mathbf{Env} \to \mathbf{D} \to \mathbf{MS} \to \mathbf{Kont} \to \mathbf{Ans}_{mon} \end{aligned}$$

The continuation could then be stored and invoked later, for example. This would allow the user to 'travel back in time' to previous execution states. Tolmach and Appel (1990) have been using such a technique to provide 'reverse-execution' for ML.

Communicating monitors Currently monitors can only read and update their own state. However, it may be desirable for monitors to communicate and share information. This can be easily implemented by allowing each monitoring function to read and update other monitors' states. Note that if we only provide a read capability, our correctness result still holds. However, if monitors can update other monitors' states we can no longer ensure non-interference amongst monitors.

10.2 Implementing monitors for other languages

Can our methodology be scaled up from the kernel functional languages treated here to meet the demands of 'real' programming languages? As stated earlier, the framework is applicable to any language which admits continuation semantics specification. This includes a large family of programming languages: imperative, logic, functional, etc. In Kishon (1992) the specification of a monitoring system for a simple Pascal-like language is given. The implementation strategy repeats itself almost identically.

An interesting question is what happens in imperative languages in case of jumps, non-local exits, or even call-with-current-continuation constructs. Recall that the post monitoring function is embedded in the continuation. However, in jump-like commands (e.g. goto's, callcc, etc.) the continuation is discarded for a new continuation. As a result, post-monitoring activity is lost together with its continuation. Is this a feature or a bug? We believe that this behaviour is faithful to the actual intent; by jumping we do not conclude the evaluation and therefore it is expected that the post monitoring function would not be called. However, this could be changed by identifying the program points where a jump may occur and calling the post-monitoring function just *before* the jump.

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

10.2.1 'Industrial-strength' Iimplementation

Every theory longs for its 'industrial-strength implementation'. In most cases such longings remain unanswered. We believe that the methodology presented in this paper can effectively be used as a basis for such an implementation. Still, to provide reasonable industrial-strength performance we need to improve our system in the following ways:

Tight integration with CPS-based compiler In order to exploit CPS rather than be penalized by it (Section 9.2.4), it is desirable for the system to be integrated more tightly with the compiler, since a certain degree of 'compilation' has already been done. This would be most easily done with a CPS-based compiler, such as Orbit or Standard ML of New Jersey (Appel and Jim, 1989).

Better partial evaluation technology Our system depends heavily on the performance of the partial evaluator. Schism performed admirably for this purpose, but at the time of our experiments it was not quite 'industrial strength'. Since then, Schism has been improved in many directions: better residual code, better BTA analysis, and improved user interface. As partial evaluation technology improves so is the performance and effectiveness of our methodology. But in addition, we would prefer a partial evaluator for *Haskell* to take advantage of Haskell's similarity to denotational semantics notation.

10.3 User-defined monitors

We believe that our methodology is well-structured enough to permit user-defined monitors; our correctness results provide a degree of safety and modularity not achievable before. We envision a programming environment which provides a toolbox of common monitors to which the user may add his or her own monitors and compose them safely with existing ones. Ultimately this environment could be populated with different language modules (e.g. Pascal, C, ML, Haskell, etc.), and with proper engineering it will have good performance.

10.4 Strict versus non-strict debugging

There seems to be some consensus that monitoring approaches for languages based on different orders of evaluation are inherently different. However, our monitor specifications for the strict and non-strict interpreters share a large amount of code. Indeed, it is quite surprising that the lazy debugger, for example, differs only in seven lines of code from its eager counterpart. A closer look at the code reveals that most differences are related to how values are retrieved from the environment. This suggests that by a proper modular design, debugging environments for multiple languages can be made to share most of their code.

On the other hand, it could be argued that our sequential interpretation of lazy evaluation is artificial, and too operational. More research is required to explore other solutions to this problem.

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

Acknowledgements

We wish to thank Charles Consel for his involvement in earlier phases of this research, in particular his assistance in the use of Schism. Funding for this research was received from the Advanced Research Projects Agency under ONR contracts N00014-90-C-0024 and N00014-91-J-4043.

A Haskell support code

In this appendix we present some common datatypes and utility functions used by the Haskell specifications presented earlier.

General datatypes

```
    type Functional a = a -> a
    type ProgramText  = String
    type Id           = String
    type FunName      = String
    type ArgName      = String
    type ValString    = String
    type IdeName      = String
```

Standard algebras

```
-- STACKS --

type StackType a = [a]

stkEmpty :: StackType a
stkEmpty = []

stkPush :: a -> StackType a -> StackType a
stkPush x s = x:s

stkPop :: StackType a -> StackType a
stkPop (_:s) = s

stkTop :: StackType a -> a
stkTop (x:_) = x

-- SETS --

type SetType a = [a]

setEmpty :: SetType a
setEmpty = []

setMember :: (Eq a) => a -> SetType a -> Bool
setMember x []     = False
setMember x (y:xs) = (x == y) || setMember x xs

setInsert :: (Eq a) => a -> SetType a -> SetType a
```

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

```
setInsert x xs = x:[ y | y <- xs, x /= y ]

setDelete :: (Eq a) => a -> SetType a -> SetType a
setDelete x s = filter ((/=) x) s

-- ASSOCIATION LISTS --

type AssocType a b = [(a,b)]

assocEmpty :: AssocType a b
assocEmpty = []

assocPut :: (Eq a) => a -> b -> AssocType a b -> AssocType a b
assocPut id x []                             = [(id,x)]
assocPut id x ((id',y):rest)  | id==id'      = (id,x):rest
                              | otherwise    = (id',y):
                                               (assocPut id x rest)

assocGet :: (Eq a) => a -> AssocType a b -> b
assocGet id ((id',x):rest) | id==id'   = x
                           | otherwise = assocGet id rest

assocExist :: (Eq a) => a -> AssocType a b -> Bool
assocExist id []             = False
assocExist id ((id',x):rest) = (id == id') || (assocExist id rest)
-- ENVIRONMENTS --

type EnvType key value = key -> value

envUpd :: (Eq key) => EnvType key val -> key -> val -> EnvType key val
envUpd env id val = env' where
                    env' id' | id == id' = val
                             | otherwise = env id'

envEmpty :: val -> EnvType key val
envEmpty undefVal = λkey -> undefVal


-- STORES --

type StoreType storeableVal = (Loc, EnvType Loc storeableVal)
type Loc = Int

storeEmpty :: undefVal ->  StoreType undefVal
storeEmpty undefVal = (1,envEmpty undefVal)

storeLook :: StoreType a -> Loc -> a
storeLook (l,env) loc = env loc

storeUpd :: StoreType a -> Loc -> a -> StoreType a
storeUpd (l,env) loc val = (l,envUpd env loc val)

storeAlloc :: StoreType a -> (Loc,StoreType a)
storeAlloc (l,env) = (l,(l+1,env))
```

B An interactive source-level debugger

In Section 7 we described how to synthesize an interactive monitor. In this section we specify a particular source-level interactive debugger for the languages presented in Section 2.

Debugger annotations

The debugger annotations are similar to the tracer annotations; every function body is annotated with a label – `LDebug` – which provides the debugger with information about the function: the function name, its formal arguments and the the names of the local definitions. Another debugger label which was already discussed in Section 7.3 is `LBreak`. This label is reported at every interpretive step thus allowing the debugger to single step through the execution.

Debugger algebras

The debugger maintains a *stack*, called the *frame stack*, which records the frame of each function call; each such frame records the function name, its formal arguments identifiers and the names of local definitions. This stack is updated before every function call and after its return. In addition, the debugger maintains a set of breakpoints represented by function names. The user can add or delete functions from this set. All these arguments and more are captured within the monitor state.

Monitoring functions

The monitoring functions perform the simple chore of updating the frame-stack whenever a function call/return occurs. In addition, the pre-monitoring function enters the interactive session with the user whenever a break is desired.

B.1 Source-level debugger for strict functional language

Figures 22—23 contain the full specification of the debugger for `eager`, which responds to the following commands:

- **run:** begin/continue with the execution of the program.
- **step:** perform a single interpretive step.
- **list:** display the next expression to be evaluated.
- **stop** $\langle fn \rangle$: stop execution when function $\langle fn \rangle$ is called.
- **unstop** $\langle fn \rangle$: clear breakpoint at function $\langle fn \rangle$.
- **show:** display the formal and local arguments of the current function.
- **eval** $\langle exp \rangle$: evaluate $\langle exp \rangle$ with current dynamic environment.
- **debug** $\langle exp \rangle$: debug $\langle exp \rangle$ with current dynamic environment (*recursive debugging*).
- **where:** display the current function call chain.

Other commands like watch points, tracing, stopping at line numbers etc. can be added to this debugger using the same methodology.

```
module EagerDebug where

--- SYNTAX ---
-- Debugger Label Syntax: a call frame and a break
data Label = LDebug FunName [ArgName] [ArgName] | LBreak
annotate :: Exp -> Exp                    -- omitted

--- ALGEBRAS ---
type Frame    = (FunName,[ArgName],[ArgName])
type IOState  = (InChan,OutChan,MonState) -- Input x Output x MonState
type SemArgs  = Env                       -- semantic arguments
type KontArgs = D                         -- continuation arguments

-- Debugger State: stop-at set, call-frame stack, and single-step flag
type MonState = (SetType Id, StackType Frame, Bool)
updStp stp ( _ ,stk,brk) = (stp,stk,brk)  -- State updaters
updStk stk (stp, _ ,brk) = (stp,stk,brk)
updBrk brk (stp,stk, _ ) = (stp,stk,brk)

writeUsr :: String -> OutChan
writeUsr msg = msg ++ "\n"

readUsr :: InChan -> (String,InChan)
readUsr inChan = (usrInput, inChan') where (usrInput,inChan') = lex inChan

getCmd :: Exp -> SemArgs -> InChan -> MonState -> IOState
getCmd exp semArgs inp dstate = (inp3, out1++out2++out3, dstate3)
 where out1 = writeUsr "command?"
       (cmd,inp1) = readUsr inp
       (resume,(inp2,out2,dstate2)) = processCmd cmd exp semArgs inp1 dstate
       (inp3,out3,dstate3) = if resume then (inp2,[],dstate2)
                                       else getCmd exp semArgs inp2 dstate2

processCmd :: String->Exp->SemArgs->InChan->MonState->(Bool,IOState)
processCmd cmd exp semArgs inp dstate@(stp,stk,brk) = case cmd of
   "run"     -> (True, (inp,[],dstate))
   "step"    -> (True, (inp,[],updBrk True dstate))
   "list"    -> (False, (inp, writeUsr (show (delabel exp)), dstate))
   "stop"    -> (False,(inp1,[],updStp (setInsert fn stp) dstate))
                where (fn,inp1) = readUsr inp
   "unstop"  -> (False, (inp1,[],updStp (setDelete fn stp) dstate))
                where (fn,inp1) = readUsr inp
   "show"    -> (False,(inp, out1++out2, dstate))
                where (fn,fvars,lvars) = stkTop stk
                            out1 = writeVars "formal " fvars semArgs
                            out2 = writeVars "local  " lvars semArgs
   "eval"    -> (False,recDebug semArgs inp dstate False)
   "debug"   -> (False,recDebug semArgs inp dstate True)
   "where"   -> (False, (inp, writeUsr (show fns), dstate))
                where fns = [ fn | (fn,_,_) <- stk ]
   otherwise -> (False, (inp,writeUsr "<undef command>", dstate))
```

Fig. 22. Source-level debugger for eager (part 1).

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

```
recDebug :: SemArgs -> InChan -> MonState -> Bool -> IOState
recDebug currentSemArgs inp dstate@(stp,stk,brk) isDebug = (inp2,out3,dstate)
 where (Interpreter parse evalf semArgs kont) = interpreter
       interpreter'  = Interpreter parse evalf currentSemArgs kont
       debugger'     = IOMonitor "debug" annotate pre post
                                   (setEmpty,stkEmpty,isDebug)
       [(exp, inp1)] = reads inp
       (inp2,out2)   = execute (debugger' & interpreter') (delabel exp) inp1
       out3          = if isDebug
                         then writeUsr ">> Enter Recursive Debug" ++
                              out2 ++ writeUsr ">> Exit Recursive Debug"
                         else out2

interpreter = eager

writeVars :: String -> [Id] -> SemArgs -> OutChan
writeVars prefix ids semArgs =
 case ids of []        -> []
             (id:rest) -> writeUsr out ++ writeVars prefix rest semArgs
                 where out = prefix ++ id ++ " = " ++ getIdVal semArgs id

getIdVal :: SemArgs -> Id -> String
getIdVal env id = case (env id) of  (Num n)  ->  show n
                                    (Bol b)  ->  show b
                                    (Fun f)  ->  "<fun>"
                                    (DUndef) -> "<undef>"

writeTrace :: SemArgs -> (StackType Frame) -> OutChan
writeTrace semArgs stk = (writeUsr ("Stop in " ++ fn)) ++ out
                   where (fn,fargs,_) = stkTop stk
                         out = writeVars "Formal argument " fargs semArgs

-- MONITORING FUNCTIONS --

pre :: Label -> Exp -> SemArgs -> MonState -> InChan -> IOState
pre label exp semArgs dstate@(stp,stk,brk) inp = case label of
 (LBreak) -> if brk then getCmd exp semArgs inp (updBrk False dstate)
                    else (inp,[],dstate)
 (LDebug fn fvars lvars) ->
   if setMember fn stp then (inp,writeTrace semArgs stk',updBrk True dstate')
                        else (inp,[],dstate')
    where stk'    = stkPush (fn,fvars,lvars) stk
          dstate' = updStk stk' dstate

post :: Label->Exp->SemArgs->KontArgs->MonState->InChan->IOState
post label exp semArgs result dstate@(stp,stk,brk) inp = case label of
   (LBreak) -> (inp,[],dstate)
   (LDebug fn fvars lvars) -> (inp,[],updStk (stkPop stk) dstate)

-- PUTTING IT ALL TOGETHER --

eagerDebugger :: IOMonitorType Label Exp Env D MonState
eagerDebugger = IOMonitor annotate pre post (setEmpty,stkEmpty,True)
```

Fig. 23. Source-level debugger for eager (part 2).

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

```
Run> main (eagerDebugger & eager) simpleFact3
 command? stop fac
 command? run
   Stop in fac
   Formal argument n = 3
 command? show
   formal n = 3
   local  r = <undef>
 command? run
   Stop in fac
   Formal argument n = 2
 command? where
   [fac,fac]
 command? step
 command? list
   (n==0)
 command? eval n
   the result is:2
 command? debug (fac 0)
 >> Enter Recursive Debug
 command? stop fac
 command? run
   Stop in fac
   Formal argument n = 0
 command? step
 command? list
   (n==0)
 command? step
 command? step
 command? step
 command? list
   1
 command? step
   the result is:1
 >> Exit Recursive Debug
 command? unstop fac
 command? run
   the result is: 6
```

Fig. 24. Debugging simpleFact3 with eagerDebugger.

The session shown in Figure 24 was produced by debugging the execution of simpleFact3, whose annotated abstract syntax is given by:

```
letrec fac = lambda n . {LDebug fac(n)(r)}:
               if n=0 then 1 else let r = fac (n-1) in n * r
in fac 3
```

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

```
Main> main (lazyDebugger & lazy) silly
 command? list
   letrec baz = lambda x . x + 1
   letrec foo = lambda x y . baz x
   in foo 3 2
 command? stop baz
 command? run
   Stop in baz
   Formal argument x = <thunk>    {x is a thunk}
 command? eval x
   the result is: 3               {x is forced in recursive eval}
 command? show
   formal x = <thunk>             {x remains unevaluated!}
 command? list
   x + 1
 command? step
 command? step
 command? step
 command? step
 command? show
   formal x = 3                   {eventually x is forced}
 command? run
 the result is:4
```

Fig. 25. Debugging `silly` with `lazyDebugger`.

B.2 Source-level debugger for a Non-strict functional language

In the previous section we presented the specification of a debugger for the eager interpreter. Interestingly, only minor changes (seven lines of code) are necessary to transform the eager debugger into a fully lazy one. As in previous monitor examples, the changes are mostly related to the way identifier values are looked up in the environment. Specifically, the following changes are required:

- Redefine `SemArgs` and `KontArgs`. We change the semantic domains types to reflect the lazy interpreter domains:

  ```
  type SemArgs  = (Env, Store)
  type KontArgs = (D, Store)
  ```

- Change interpreter used in `recDebug`.

  ```
  interpreter = lazy
  ```

- Change the way identifier values are lookup in the semantic domains. `getIdVal` now is passed the environment and the store as the semantic arguments and instead of a simple environment lookup we need to go through the store to get the actual value.

```
getIdVal :: SemArgs -> Id -> String
getIdVal (env,store) id = case (storeLook store (env id)) of
                               (Val d)         -> show d
                               (Thunk t)       -> "<thunk>"
                               (StoreValUndef) -> "<undef>"
```

- Finally, assign the new debugger to lazyDebugger:

```
lazyDebugger = IOMonitor annotate pre post (setEmpty,stkEmpty,True)
```

Obviously, the lazy debugger has similar functionality as the eager debugger, yet preserves lazy evaluation. To demonstrate this debugger in action, we present in Figure 24 a session that was produced by debugging silly (Section 5.1.2).

In addition to its laziness and consistency this debugger has some other novel features. In particular, like the tracer for lazy, unevaluated values are represented as <thunk>s, and the user is presented with a faithful rendition of the sequential lazy evaluation order. Nevertheless, one can force values in recursive debugging without affecting the value at the current evaluation state – this provides an elegant way to 'peek' into thunks without disturbing the actual lazy evaluation order (for example, see how x is forced in the silly debugging session).

References

Abelson, H., Sussman, G. J. and Sussman, J. (1985) *Structure and Interpretation of Computer Programs.* MIT Press.

Allison, L. (1986) *A Practical Introduction to Denotational Semantics.* Cambridge University Press.

Appel, A. W. and Jim, T. (1989) Continuation-passing, closure-passing style. In: *ACM Symposium on Principles of Programming Languages*, pp. 193–302, January.

Berry, D. (1991) *Generating Program Animators from Programming Language Semantics.* PhD thesis, University of Edinburgh, June.

Bertotm Y. (1988) Occurrences in debugger specifications. In: *Proceedings ACM Conference on Programming Languages Design and Implementation.* ACM, June.

Bjørner, D., Ershov, A. P. and Jones, N. D. (1988) *Partial Evaluation and Mixed Computation.* North-Holland.

Clinger, W. and Rees, J. (1991) Revised^4 report on the algorithmic language scheme. Technical Report MIT AI MEMO, MIT, Cambridge, MA, November.

Cook, W. and Palsberg, J. (1989) A denotational semantics of inheritance and its correctness. In: *OOPSLA 1989. SIGPLAN Notices*, **24**(10), October.

Delisle, N. M., Menicosy, D. E. and Schwarts, M. D. (1984) Viewing a programming environment as a single tool. In: *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April. (*SIGPLAN Notices*, **19**(5), May.)

Dybvig, K. R., Friedman, D. P. and Haynes, C. T. (1988) Expansion-passing style: A general macro mechanism. In: *Lisp and Symbolic Computation, 1*, pp. 53–75. Kluwer Academic.

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press

Hall, C. V. and O'Donnell, J. T. (1985) Debugging in a side effect free programming environment. In: *Proceedings SIGPLAN Symposium on Programming Languages and Programming Environments*, June.

Hudak, P. and Fasel, J. (1992) A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992.

Hudak, P., Peyton Jones, S. and Wadler, P. (eds.) (1992) Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *SIGPLAN Notices*, **27**(5), May.

Hudak, P. and Young, J. (1988) A collecting interpretation of expressions (without powerdomains). In: *Proceedings of the ACM Symposium of Principles of Programming Languages.* ACM.

Jones, N. D., Sestoft, P. and Sondergaard, H. (1987) *Mix: a self-applicable partial evaluator for experiments in compiler generation.* Technical Report DIKU Report 87/08, University of Copenhagen, Denmark.

Kishon, A. (1992) *Theory and Practice of Semantics-directed Program Execution Monitoring.* PhD thesis, Yale University, May. (Also Yale Research Report YALEU/DCS/RR-905.)

Kishon, A., Hudak, P. and Consel, C. (1988) Monitoring semantics: A formal framework for specifying, implementing and reasoning about execution monitors. In: *Proceedings of the ACM Conference on Programming Languages Design and Implementation.* ACM, June.

Kranz, D. A. (1988) *ORBIT: An Optimizing Compiler for Scheme.* PhD thesis, Yale University.

Kranz, D. A., Kelsey, R., Rees, J. A., Hudak, P., Philbin, J. and Adams, N. I. (1986) Orbit: An optimizing compiler for Scheme. In: *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pp. 219–233, June.

Lee, P. (1989) *Realistic Compiler Generation.* MIT Press.

O'Donnell, J. T. and Hall, C. V. (1988) Debugging in applicative languages. In: *Lisp and Symbolic Computation, 1.* Kluwer Academic.

Reddy, U. (1988) Objects as closures: Abstract semantics of object oriented languages. In: *ACM Conference on Lisp and Functional Programming.*

Safra, S. and Shapiro, E. (1989) Meta interpreters for real. In: *Concurrent Prolog, collected papers, Vol. 2.* MIT Press.

Shapiro, E. (1982) *Algorithmic Program Debugging.* MIT Press.

Sterling, L. and Shapiro, E. (1986) *The Art of Prolog, Advanced Programming Techniques.* MIT Press.

Tennent, R. G. (1977) *A denotational definition of the programming language Pascal.* Technical Report Technical Report 77–47, Department of Computing Sciences, Queen's University, Ontario.

Tolmach, A. P. and Appel, A. W. (1990) Debugging Standard ML without reverse engineering. In: *Proceedings ACM Conference on Lisp and functional programming*, June.

Toyn, I. and Runciman, C. (1986) Adapting combinator and secd machines to display snapshots of functional computations. *New Generation Computing*, **4**:339—363.

https://doi.org/10.1017/S0956796800001465 Published online by Cambridge University Press