Monitoring Semantics: A Formal Framework for Specifying, Implementing, and Reasoning about Execution Monitors

Amir Kishon Paul Hudak Charles Consel

Research Report YALEU/DCS/RR-850
March, 1991

This research was supported by DARPA grant N00014-88-K-0573
and a DARPA sub-contract from Software Options, Inc.

Monitoring Semantics: A Formal Framework for Specifying, Implementing, and Reasoning about Execution Monitors [*]

Amir Kishon Paul Hudak Charles Consel

Yale University
Department of Computer Science
P.O. Box 2158 Yale Station
New Haven, CT 06520
{kishon,hudak,consel}@cs.yale.edu

Abstract

We introduce *monitoring semantics*, a non-standard model of program execution that captures "monitoring activity" as found in debuggers, profilers, tracers, etc. A monitoring semantics is a conservative extension of a language's standard denotational semantics, and is parameterized with respect to specifications of monitoring activity.

Beyond its theoretical interest, monitoring semantics forms a practical basis for building effective monitors, with standard partial evaluation techniques being used as a key optimization strategy. In particular, specializing a monitoring semantics with respect to a source program amounts to removing the interpretive overhead associated with the static aspects of monitoring, yielding an *instrumented* program in which the extra code to perform monitoring actions has been automatically "embedded" into the program.

A monitoring semantics can be *automatically* derived for any language for which a continuation semantics specification has been provided. This is achieved by using *functionals* to embed non-standard behavior at all levels of recursion. We illustrate the approach for a higher-order functional language, showing examples of several different monitor specifications, including benchmarks of their implementations.

[*]This is an extended version of the paper to appear in the Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation, June 1991

1 Introduction

A *program execution monitor* is a system that monitors the dynamic run-time behavior of a program. Examples include debuggers, profilers, tracers and demons. Monitors are an obviously important element in any software development environment. However, as was mentioned in a 1981 survey of the field, "program execution monitoring has been neglected as a research topic; the available literature contains mainly case studies, without an adequate discussion of the fundamental concepts, goals and limitations" [PN81]. The work described in this paper grew out of a perceived need for a more formal treatment of program execution monitoring, with the hope of developing a formal basis for such monitors that could easily be related to the standard semantics of a source language, and that could form a practical basis for building effective monitors for use in the real world.

This paper introduces *monitoring semantics*, a formal basis for program execution monitors. Monitoring semantics provides a generic semantic model for monitors with the following characteristics:

1. A monitoring semantics can be *automatically* derived from any denotational semantics specification expressed in continuation style. This is achieved by using *functionals* to embed non-standard behavior at all levels of recursion such that their fixpoints yield the desired result.

2. The derived monitoring semantics is parameterized with monitor specifications which, once instantiated, are able to capture any sequential, deterministic monitoring activity.

3. Any monitoring semantics derived in this way is *consistent* with the original standard semantics—i.e. it is not possible for a monitor to change program behavior.

4. The method is *compositional*—monitors may be built on top of other monitors to produce composite monitoring behaviors.

5. Using standard partial evaluation techniques, it is possible to build *practical* monitors using our methodology; i.e. monitors whose execution speed matches that of conventional interpreters.

Points 1 and 2 indicate the degree of *generality* of our methodology—it can be used for any programming language for which a continuation semantics is available, and it is able to capture a very broad class of program monitoring activities. Points 3 and 4 highlight the *safety* and *modularity* of our approach, respectively, thus easing the processes of specifying and reasoning about monitors. Finally, point 5 indicates the *practicality* of our approach, and indeed we have working implementations of several monitors built using our technique.

A philosophical consequence of our methodology is the raising of "monitoring semantics" to the same first-class status as "standard semantics"—a consequence that seems eminently desirable, given the intimate relationship between the programmer and the "non-standard" behaviors that typify the program development process (it is only the final product whose "black-box" behavior corresponds most closely with the standard semantics). Perhaps more importantly, we believe that our methodology will enable programmers to write their *own* monitors in an effective, straightforward way (without fear of changing program behavior), rather than be confined to the limitations and rigidity of most software development environments. As an example of our methodology in action we will introduce the monitoring semantics of a higher-order language and provide the specification of several common monitoring activities for that language (a tracer, a profiler, a demon and a collecting monitor).

Current Methodologies Despite the importance of monitors in any software development environment, the lack of formal and general treatment of the problem is evident in current research. Current approaches for constructing monitoring systems are mainly based on instrumenting either the source

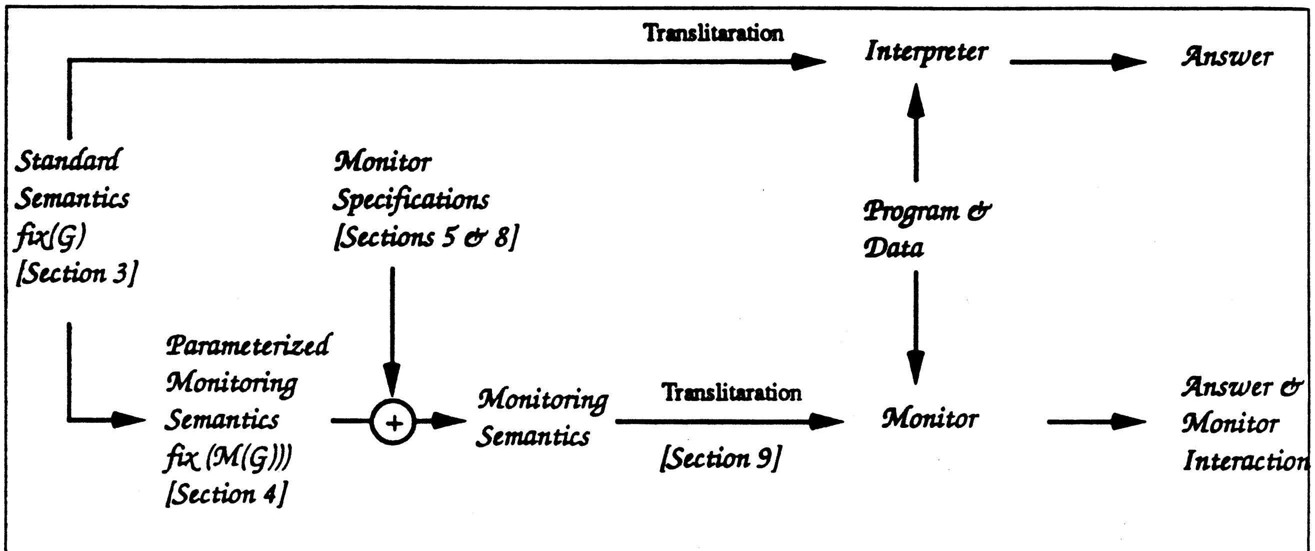
Translitaration
Interpreter
Answer
Standard Semantics fix(G) [Section 3]
Monitor Specifications [Sections 5 & 8]
Program & Data
Parameterized Monitoring Semantics fix(M(G))) [Section 4]
Monitoring Semantics
Translitaration
[Section 9]
Monitor
Answer & Monitor Interactio

Figure 1: Semantic relations diagram

code [DFH88, HO85, TA90] or an interpreter (or abstract machine) [OH88, SS89, Sha82] to include monitoring behavior. As general methodologies many of these approaches have limitations: they may interact adversely with the normal execution of the program (or with other monitoring activities), they have little formal semantic justification, and they usually rely on hand-crafted techniques. Our approach attempts to resolve these problems by providing a general framework for specifying, implementing and reasoning about monitors. Our work compares most closely with O'Donnell and Hall's work [HO85, OH88], which advocates enhancement of program results with debugging data, and with Safra and Shapiro's work [SS89], which advocates the use of partial evaluation to instrument interpreters.

Outline This paper is organized as shown in Figure 1, which also illustrates well the relationships between the various semantic entities (e.g. standard semantics, monitoring semantics) and concrete entities (e.g. interpreters, monitors) that our methodology comprises. In Section 2 we present an overview of our approach. From the standard semantics specification $\mathcal{G}$ described in Section 3, we describe in Section 4 how to automatically derive a parameterized monitoring semantics $\mathcal{M}(\mathcal{G})$. Then in Section 5 we describe how to *instantiate* such a semantics with particular monitor specifications, thus yielding a complete monitor. Sections 6 and 7 discuss the compositionality and soundness of monitoring semantics, respectively, and Section 8 presents several examples. Finally, Section 9 addresses implementation issues.

2 An Overview of Semantic Monitoring

In a nutshell, our approach to monitoring program execution can be intuitively described as follows:

A language's continuation semantics specifies a linear ordering on program execution, and thus can be used as the basis for ordering monitoring activity (just as it is used to guide code generation in "semantics-directed compilers"). But instead of interpreting a program's meaning as an element α in a domain **Ans** of "final answers," we will interpret it as a function f with type $\overline{\textbf{Ans}} = \textbf{MS} \rightarrow (\textbf{Ans} \times \textbf{MS})$, where **MS** is a domain of "monitor states". Given $\sigma : \textbf{MS}$ as an initial (presumably empty) monitor state, then $f\ \sigma = \langle \alpha', \sigma' \rangle$, where σ' is the resulting monitoring information. We construct such an interpretation as a *parameterized* semantics in such a way that all instantiations of the semantics (i.e. all possible monitors defined using our approach) have the property that $\alpha' = \alpha$.

The technique used in deriving the monitoring semantics for a language is somewhat unique in itself, relying not just on standard use of higher-order functions, but on the use of *functionals* to embed non-standard behavior at all levels of recursion such that their fixpoints yield the desired result.

Finally, by treating the resulting specification as a *program* (via transliteration into a functional language), we have essentially constructed a (non-standard) interpreter. Although inefficient if executed directly, partial evaluation can be used to enhance efficiency on a sound basis, just as is done in work on semantics-directed compilation.

Notation The reader is assumed to be familiar with the concepts and terminology of continuation semantics as found, for example, in Gordon [Gor79], Schmidt [Sch86], and Stoy [Sto77]. We will use **boldface** for domains, and the notations $\mathbf{X} \rightarrow \mathbf{A}*_i \rightarrow \mathbf{Y}$ and $f\ x\ a*_i\ y$ to denote $\mathbf{X} \rightarrow \mathbf{A}_{i_1} \rightarrow \ldots \rightarrow \mathbf{A}_{i_n} \rightarrow \mathbf{Y}$ and $f\ x\ a_{i_1} \ldots a_{i_n}\ y$ for some n respectively. As an aid to the eye, continuations are enclosed in braces $\{\}$. Square brackets are used for environment update, as in $\rho[e \mapsto x]$; angle brackets are used for tupling, as in $\langle e_1, e_2, e_3 \rangle$; and domain projection and injection are noted by $(x|\mathbf{D})$ and $(x \text{ in } \mathbf{D})$ respectively.

3 Continuation Semantics Framework

As mentioned earlier, we assume that a continuation semantics is used to express the standard semantics. This does not result in any significant loss of expressiveness, since a continuation semantics is able to capture the behavior of any deterministic sequential language. Languages that don't require the extra expressiveness (such as purely functional languages) can still be expressed in a continuation style.

Definition 3.1 (Standard Continuation Semantics) *A* standard continuation semantics specification *is a triple* Den = ⟨Syn, Alg, Val⟩ *where:*

- Syn *denotes the* abstract syntax *definition of the language in the form of BNF rules specifying syntactic domains* $\mathbf{S}_i$.
- Alg *defines the set of* semantic algebras, *which are the semantic domains* $\mathbf{A}_i$ *plus operations over them. In particular* **Ans** *denotes the domain of answers.*
- Val *denotes the collection of* valuation functionals *(one for each syntax domain). Each valuation functional* $\mathcal{G}_i$ *has functionality:*

$$\begin{array}{lrcl} \mathcal{G}_i : \mathbf{T}_i \rightarrow \mathbf{T}_i & & & \\ \text{where} & \mathbf{T}_i & = & \mathbf{S}_i \rightarrow \mathbf{A}*_i \rightarrow \mathbf{Kont}_i \rightarrow \mathbf{Ans} \\ & \mathbf{Kont}_i & = & \mathbf{A}*_i{}' \rightarrow \mathbf{Ans} \end{array}$$

The fixpoints $\mathcal{V}_i : \mathbf{T}_i$ *of the functionals* $\mathcal{G}_i$ *(i.e.* $\mathcal{V}_i = fix\ \mathcal{G}_i$*) are the* valuation functions *that map syntactic terms to their meanings.*

Abstract Syntax (Syn):

k	$\in$	**Con**	Constants
x, f	$\in$	**Ide**	Identifiers, either bound variables or function names
e	$\in$	**Exp**	Expressions

$e ::=$	k	Constant
	$\mid x$	Identifier
	$\mid$ `lambda` x . e	Lambda abstraction
	$\mid$ `if` e_1 `then` e_2 `else` e_3	Conditional
	$\mid e_1\ e_2$	Application
	$\mid$ `letrec` f = `lambda` x . e_1 `in` e_2	Letrec

Semantic Algebras (Alg):

b	$\in$	$\mathbf{Bas} = \mathbf{Int} + \mathbf{Bool} + \ldots$	Basic values
		$\mathbf{Fun} = \mathbf{V} \rightarrow \mathbf{Kont} \rightarrow \mathbf{Ans}$	Function values
v	$\in$	$\mathbf{V} = \mathbf{Bas} + \mathbf{Fun}$	Denotable values
a	$\in$	$\mathbf{Ans}$	Answers
ρ	$\in$	$\mathbf{Env} = \mathbf{Ide} \rightarrow \mathbf{V}$	Environments
κ	$\in$	$\mathbf{Kont} = \mathbf{V} \rightarrow \mathbf{Ans}$	Expression continuations

Valuation Functionals (Val):

$\mathcal{G}_\lambda :$	$\mathbf{T}_\lambda \rightarrow \mathbf{T}_\lambda$	Expression valuation functional
	where $\mathbf{T}_\lambda = \mathbf{Exp} \rightarrow \mathbf{Env} \rightarrow \mathbf{Kont} \rightarrow \mathbf{Ans}$	Gives meaning to expressions
$\mathcal{K} :$	$\mathbf{Con} \rightarrow \mathbf{V}$	Gives meaning to constants

$\mathcal{G}_\lambda\ \mathcal{E}\ e\ \rho\ \kappa =$

case e of

$[\![k]\!]$	:	$\kappa\ (\mathcal{K}\ [\![k]\!])$
$[\![x]\!]$	:	$\kappa\ (\rho\ x)$
$[\![$`lambda` x . $e]\!]$	:	$\kappa\ ((\lambda v.\ \mathcal{E}[\![e]\!]\ \rho[x \mapsto v])$ in $\mathbf{Fun})$
$[\![$`if` e_1 `then` e_2 `else` $e_3]\!]$	:	$\mathcal{E}[\![e_1]\!]\ \rho\ \{\lambda v.\ v\vert\mathbf{Bool} \rightarrow \mathcal{E}[\![e_2]\!]\ \rho\ \kappa,\ \mathcal{E}[\![e_3]\!]\ \rho\ \kappa\}$
$[\![e_1\ e_2]\!]$	:	$\mathcal{E}[\![e_2]\!]\ \rho\ \{\lambda v_2.\ \mathcal{E}[\![e_1]\!]\ \rho\ \{\lambda v_1.\ (v_1\vert\mathbf{Fun})\ v_2\ \kappa\}\}$
$[\![$`letrec` f = `lambda` x . e_1 `in` $e_2]\!]$	:	$\mathcal{E}[\![e_2]\!]\ \rho'\ \kappa$
		whererec $\rho' = \rho[f \mapsto (\lambda v.\ \mathcal{E}[\![e_1]\!]\ \rho'[x \mapsto v])$ in $\mathbf{Fun}]$

Figure 2: Standard Continuation semantics for $\mathcal{L}_\lambda$

The explicit identification of the functionals will allow us to derive new valuation functions that "inherit" the behavior of the old. The sequence $\mathbf{A}*_i$ of arguments to $\mathcal{V}_i$ constitutes the normal semantic context necessary to perform an evaluation (e.g. environment, state, etc.), and the sequence $\mathbf{A}*_i'$ of arguments to $\mathbf{Kont}_i$ denotes intermediate results.

As an example of this semantic framework, Figure 2 shows the continuation semantics for a higher-order functional language consisting of conventional constructs such as applications, abstractions, constants, identifiers, conditionals and letrec clauses. This language, which we will call $\mathcal{L}_\lambda$, will be used throughout the paper to illustrate the discussion; its simplicity will allow us to focus on the essence of our approach.

3.1 Parameterizing standard continuation semantics with respect to its final answer

Since a continuation represents a program's complete computation upon some intermediate results, the continuation may contain some "final processing" that produces the final answer. This allows one to map a program's final answer to arbitrary value spaces. For example, consider the answer of a $\mathcal{L}_\lambda$ program with an initial continuation:

$$\kappa_{init} = \{\lambda v.\ FinalProcessing(v)\}$$

This would map $\mathcal{L}_\lambda$'s final answer to whatever is specified by *FinalProcessing*. As is mentioned in [Sch86] this generalization makes continuations especially suitable for handling unusual answers. Later, we will make an extensive use of this property in deriving the monitoring semantics; but first let us formalize this notion.

Definition 3.2 (Answer Algebra) *Let* Den *be a Standard Continuation Semantics with semantic domains* $\mathbf{A}_i$*, an algebra:* Ans = $[\mathbf{Ans}; \{\phi_1, \ldots, \phi_n]$ *is an* Answer Algebra *for* Den *iff*

- *It defines the final answer domain* **Ans** *of* Den.
- *It defines all the operations* ϕ_i *mapping values in* $\mathbf{A}_i$ *into a final answer in* **Ans**. *That is,*

$$\phi_i :: \mathbf{A}*_i \rightarrow \mathbf{Ans} \qquad i = 1 \ldots n$$

By factoring out the answer algebra from the specification, we can essentially parameterize the continuation semantics with respect to its final answer domain.

Definition 3.3 (Parameterized Continuation Semantics) *Let* Den *be a Standard Continuation Semantics (Definition 3.1),* Den *is said to be* parameterized with the answer domain *iff its answer domain and operations on this domain are defined by an algebra (i.e. an answer algebra) which is assumed to be given as a parameter to the specification.*

The explicit identification of the answer algebra will later allow us to redefine the semantics' final answer by changing this algebra. For example, consider the $\mathcal{L}_\lambda$'s specification shown in Figure 2. Since $\mathcal{L}_\lambda$'s final answer is solely produced by its initial continuation the only answer algebra operation needed is a mapping from denotable values to final answers. Such mapping, denoted ϕ, is assumed to be provided by an answer algebra which is given as a parameter to the specification. An initial continuation defined by:

$$\kappa_{init} = \{\lambda v.\ (\phi\ v)\}$$

will provide the desired parameterization of $\mathcal{L}_\lambda$. We can now instantiate ϕ to map $\mathcal{L}_\lambda$'s final answer to the desired domain. Note that in the case of the standard semantics, a typical answer algebra for $\mathcal{L}_\lambda$ is given by:

$$\begin{aligned}\mathrm{Ans}_{std}^{\mathcal{L}_\lambda} &= [\mathbf{Bas}; \{\phi : \mathbf{V} \rightarrow \mathbf{Bas}\}]\\ &\quad \text{where } \phi\ v = v|\mathbf{Bas}\end{aligned}$$

However, we can make the $\mathcal{L}_\lambda$'s final answer more interesting by providing a more elaborate answer algebra, for example:

$$\begin{aligned}\mathrm{Ans}_{str}^{\mathcal{L}_\lambda} = &\ [\mathbf{String}; \{\phi : \mathbf{V} \rightarrow \mathbf{String}\}]\\ &\text{where } \phi : \mathbf{V} \rightarrow \mathbf{String}\\ &\qquad \phi\ v = \texttt{"The result is:"} +\!\!+\ \mathit{toStr}(v)\\ &\qquad\qquad \text{where } \mathit{toStr} : \mathbf{V} \rightarrow \mathbf{String}\end{aligned}$$

This algebra maps the results of $\mathcal{L}_\lambda$ to character strings. Similarly, in the next section we will redefine the answer algebra to map the semantics' final answer to the monitoring answer domain.

4 Parameterized Monitoring Semantics

We now wish to enhance the continuation semantics in such a way as to yield a *parameterized monitoring semantics* which, when instantiated with the appropriate functionality (a monitor specification), will yield a complete monitor. The necessary changes to the continuation semantics are described in the following paragraphs.

4.1 Monitoring Annotations

We make only one assumption concerning the syntax of the programming language: that every syntactic category may be "tagged" with auxiliary monitoring information. Technically we could avoid this assumption by noting that every program point may be uniquely identified by tracing its location from the root of the program's syntax tree, and thus one could provide, as a separate specification, a mapping of these program points to the auxiliary information. However, we find our approach to be simpler and clearer.

The annotations themselves may contain arbitrary information for use by the monitor. In their simplest form they might act as 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 imagine that in practice the annotations would not be added explicitly by the user, but rather would be supplied by a suitably engineered programming environment. For example, a user may invoke a command to trace calls to the function `f`, and the system would then virtually (or perhaps literally) add the appropriate annotation to the definition of `f`. The examples in Section 8 were in fact generated in this way. Such a mechanism eases the burden on the user and reduces the possibility of editing errors.

In any case, for technical convenience we assume the presence of annotations. Informally, for each syntactic domain $\mathbf{S}_i$, let $\mu_i \in \mathbf{Ann}_i$ be the set of annotations allowed for that domain, and define the *annotated syntactic domain* $\overline{\mathbf{S}}_i$ by:

$$\bar{s}_i \in \overline{\mathbf{S}}_i ::= \text{(original BNF clauses)} \mid \{\mu_i\}{:}\bar{s}_i$$

As an example, the enhanced syntax for expressions in $\mathcal{L}_\lambda$ with $\mu \in \mathbf{Ann}$ is simply:

$$\bar{e} \in \overline{\mathbf{Exp}} ::= k \mid x \mid \mathtt{lambda}\ x\ .\ \bar{e} \mid \ldots \mid \{\mu\}{:}\bar{e}$$

It is noteworthy that this enhancement can be expressed formally in a way analogous to the enhancement to the valuation functionals (to be described in the next section). To do so, we use a non-conventional method to prescribe the annotated syntax—"*syntactic functionals.*" Using the $\mathcal{L}_\lambda$ syntax to illustrate this method, let $\mathcal{H}$ be:

$$\begin{aligned}\mathcal{H}(\mathbf{Exp}) = \; & \mathbf{Con} \\ & \cup\, \mathbf{Ide} \\ & \cup\, \{[\![\texttt{lambda}\ x\ .\ e]\!] \mid x \in \mathbf{Ide},\ e \in \mathbf{Exp}\} \\ & \cup\, \{[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else} e_3]\!] \mid e_1, e_2, e_3 \in \mathbf{Exp}\} \\ & \cup\, \{[\![e_1\ e_2]\!] \mid e_1, e_2 \in \mathbf{Exp}\} \\ & \cup\, \{[\![\texttt{letrec}\ f = \texttt{lambda}\ x\ .\ e_1\ \texttt{in}\ e_2]\!] \mid x, f \in \mathbf{Ide},\ e_1, e_2 \in \mathbf{Exp}\}\end{aligned}$$

We interpret $(fix\ \mathcal{H})$ as denoting the set satisfying $\mathbf{Exp} = \mathcal{H}(\mathbf{Exp})$. The new grammar can then be defined by $(fix\ \overline{\mathcal{H}})$ where:

$$\overline{\mathcal{H}}(\overline{\mathbf{Exp}}) = \mathcal{H}(\overline{\mathbf{Exp}}) \ \cup\ \{\{\mu\}{:}\bar{e} \mid \mu \in \mathbf{Ann},\ \bar{e} \in \overline{\mathbf{Exp}}\}$$

This technique will be useful in incrementally enhancing the grammar in the presence of cascaded monitors.

4.2 Monitoring State

One of the assumed parameters in the new semantics will be the *monitoring state* **MS**, which captures information of interest to the monitoring process, and which will be incrementally modified by the monitoring functions described below.

4.3 Monitoring Functions

At program points that we wish to monitor, we "probe" the evaluation process just before, and just after, evaluation. The flow of control necessary to achieve this is already captured by the continuations, and thus the monitoring semantics to be defined will essentially compose pre- and post-processing functions before and after each monitored continuation.

More specifically, for each valuation function $\mathcal{V}_i : \mathbf{S}_i \to \mathbf{A}*_i \to \mathbf{Kont}_i \to \mathbf{Ans}$, a pair of *monitoring functions* $\mathcal{M}^{\mathcal{V}_i}$ is defined consisting of two functions $\mathcal{M}^{\mathcal{V}_i}_{pre}$ and $\mathcal{M}^{\mathcal{V}_i}_{post}$ with functionalities:

$$\begin{aligned}\mathcal{M}^{\mathcal{V}_i}_{pre} &: \mathbf{Ann}_i \to \mathbf{S}_i \to \mathbf{A}*_i \to \mathbf{MS} \to \mathbf{MS} \\ \mathcal{M}^{\mathcal{V}_i}_{post} &: \mathbf{Ann}_i \to \mathbf{S}_i \to \mathbf{A}*_i \to \mathbf{A}*_i{}' \to \mathbf{MS} \to \mathbf{MS}\end{aligned}$$

The *pre-monitoring function* $\mathcal{M}^{\mathcal{V}_i}_{pre}$ gathers information *before* the evaluation of a program point, and thus takes as arguments the syntax, the semantic arguments $\mathbf{A}*_i$, and the current monitoring state; it yields a new monitoring state. The *post-monitoring function* $\mathcal{M}^{\mathcal{V}_i}_{post}$ is invoked *after* the valuation, and thus takes as an additional argument the "intermediate results" $\mathbf{A}*_i{}'$ normally passed to the continuation. This behavior characterizes the interaction between the standard behavior and the monitoring behavior, and we argue that it captures the generic characteristics of any sequential monitoring behavior.

4.4 Monitoring Answer Algebra

Recall that the meaning of a program in monitoring semantics is a function of type $\mathbf{MS} \to (\mathbf{Ans} \times \mathbf{MS})$ (Section 2), where **MS** is the monitor state domain and **Ans** is the original final answer domain. We define a new answer algebra which will map the semantics' final answer to the desired domain.

Definition 4.1 (Monitoring Answer Algebra)
Let $\text{Ans}_{std} = [\textbf{Ans}; \{\phi_1, \ldots, \phi_n\}]$ *be the standard continuation semantics' answer algebra and let* θ *be an answer transformer defined by:*

$$\theta : \textbf{Ans} \rightarrow \overline{\textbf{Ans}}$$
$$\theta\ \alpha = \lambda\sigma.\ \langle \alpha, \sigma \rangle$$

Then a monitoring semantics answer algebra *is derived as follows:*

$$\text{Ans}_{mon} = [\overline{\textbf{Ans}} = \textbf{MS} \rightarrow (\textbf{Ans} \times \textbf{MS}); \{\bar{\phi}_1, \ldots, \bar{\phi}_n\}]$$
$$\text{where}\quad \bar{\phi}_i = \theta \circ \phi_i$$

Finally, the resulting *parameterized monitoring semantics* is presented in Definition 4.2. In other words, as stated earlier, the meaning of a program will be a function mapping an initial monitor state into a pair: the original answer and a final monitoring state.[2]

To give some intuition behind parameterized monitoring semantics, Figure 3 shows the monitoring semantics for $\mathcal{L}_\lambda$. In particular, consider the valuation functional $\mathcal{G}_\lambda$ of $\mathcal{L}_\lambda$ (Figure 2), whose derived valuation functional $\overline{\mathcal{G}_\lambda}$ is shown in Figure 3. Note first that the functional $\overline{\mathcal{G}_\lambda}$ has essentially the same behavior as $\mathcal{G}_\lambda$ for all expressions except labeled ones. It is the use of the *functionals* describing the valuation functions that permits this extension to be carried out in a modular fashion. Since the valuation function is the fixpoint of the newly derived functional, the new behavior is exhibited at all levels of recursion, and thus for all subexpressions of the original program.

This technique has also been used to specify a collecting interpretation [HY88] and to give formal semantics to inheritance in object-oriented languages [CP89, Red88]. The analogy to inheritance is very interesting since the monitoring semantics $\overline{\mathcal{G}_\lambda}$ can be viewed as "inheriting" the behavior of the standard semantics $\mathcal{G}_\lambda$ to produce a more complex behavior.

Recall that the valuation process communicates its dynamic state to the monitor before and after valuation of a labeled expression. This communication is carried out by the *updPre* and *updPost* auxiliary functions, which apply the corresponding pre- and post-monitoring functions (i.e. $\mathcal{M}^{\mathcal{E}}_{pre}$ and $\mathcal{M}^{\mathcal{E}}_{post}$) to the dynamic information needed to produce an updated monitor state. Note how the updated monitor state is propagated throughout the evaluation process (just as would be the store in an imperative language). As described earlier, an initial continuation defined by

$$\kappa_{init} = \{\lambda v.\ (\bar{\phi}\ v)\}$$

will return the desired result.

5 Denotational Specification of a Monitor

In the last section the structure of parameterized monitoring semantics was formally defined. This section describes how to *instantiate* a parameterized monitoring semantics with a monitor specification, thus yielding a complete monitor. The format that we use is very similar to that of a conventional

[1]These three components will be packaged into a *monitor* in Definition 5.1.
[2]It is worth pointing out that there is a relationship between this transformation and *monads* as reported in [Mog89, Wad90].

Definition 4.2 (Parameterized Monitoring Semantics) *Let* Den *be a continuation semantics (see Definition 3.1) parameterized with the answer domain (see Definition 3.3),*
A monitoring semantics *for* Den *parameterized with:*[1]

- *annotation syntax* $\mu_i \in \mathbf{Ann}_i$ *(Section 4.1).*
- *domain* $\sigma \in \mathbf{MS}$ *of monitor states (Section 4.2)*
- *a monitoring function pair* $\mathcal{M}^{\mathcal{V}_i}$ *for each valuation function* $\mathcal{V}_i$ *in* Den *(Section 4.3).*

is defined as follows:

- **Abstract Syntax***: For each syntactic category* $\mathbf{S}_i$ *define an annotated syntax* $\overline{\mathbf{S}}_i$ *(see Section 4.1).*
- **Semantic Algebras***: Define a monitoring answer algebra to map* Den*'s final answer to domain* $\overline{\mathbf{Ans}} = \mathbf{MS} \rightarrow (\mathbf{Ans} \times \mathbf{MS})$ *(see Definition 4.1). All other algebras are the same as in* Den*.*
- **Valuation Functionals***: For each valuation functional* $\mathcal{G}_i$ *in* Den*, define* $\overline{\mathcal{G}}_i$ *as follows:*

$$
\begin{array}{l}
\overline{\mathcal{G}}_i : \overline{\mathbf{T}}_i \rightarrow \overline{\mathbf{T}}_i \\
\qquad \text{where } \overline{\mathbf{T}_i} \;=\; \overline{\mathbf{S}}_i \rightarrow \mathbf{A*}_i \rightarrow \mathbf{Kont}_i \rightarrow \overline{\mathbf{Ans}} \\
\overline{\mathcal{G}}_i\ \overline{\mathcal{V}}_i\ \bar{s}_i\ a*_i\ \kappa_i = \\
\quad \text{case } \bar{s}_i \text{ of} \\
\quad\quad [\![\{\mu_i\}{:}\bar{s}'_i]\!] : \;(\overline{\mathcal{V}}_i[\![\bar{s}'_i]\!]\ a*_i\ \kappa_{post}) \circ \mathit{updPre} \\
\quad\quad\quad \text{where} \\
\quad\quad\quad\quad \mathit{updPre}\ : \mathbf{MS} \rightarrow \mathbf{MS} \\
\quad\quad\quad\quad \mathit{updPre} = \mathcal{M}^{\mathcal{V}_i}_{pre}\ [\![\mu_i]\!]\ [\![\bar{s}'_i]\!]\ a*_i \\
\quad\quad\quad\quad \kappa_{post} = \ \{\lambda\iota*_i.\ (\kappa_i\ \iota*_i) \circ \mathit{updPost}\} \\
\quad\quad\quad\quad\quad \text{where} \\
\quad\quad\quad\quad\quad\quad \mathit{updPost} : \mathbf{MS} \rightarrow \mathbf{MS} \\
\quad\quad\quad\quad\quad\quad \mathit{updPost} = \mathcal{M}^{\mathcal{V}_i}_{post}[\![\mu_i]\!]\ [\![\bar{s}'_i]\!]\ a*_i\ \iota*_i \\
\quad\quad \text{else} : \quad \mathcal{G}_i\ \overline{\mathcal{V}}_i\ \bar{s}_i\ a*_i\ \kappa_i
\end{array}
$$

Abstract Syntax (Syn):

$\bar{e}$	$\in$	$\overline{\textbf{Exp}}$	annotated expressions
μ	$\in$	$\textbf{Ann}$	monitor annotations

$$\overline{\mathcal{H}}(\overline{\textbf{Exp}}) = \mathcal{H}(\overline{\textbf{Exp}}) \cup \{\{\mu\}{:}\bar{e} \mid \mu \in \textbf{Ann},\ \bar{e} \in \overline{\textbf{Exp}}\} \qquad (\mathcal{H} \text{ is defined in Section 4.1})$$

Semantic Algebras (Alg):

b	$\in$	$\textbf{Bas} = \textbf{Int} + \textbf{Bool} + \ldots$	Basic values
		$\underline{\textbf{Fun}} = \textbf{V} \to \textbf{Kont} \to \overline{\textbf{Ans}}$	Function values
v	$\in$	$\underline{\textbf{V} = \textbf{Bas} + \textbf{Fun}}$	Denotable values
a	$\in$	$\overline{\textbf{Ans}} = \textbf{MS} \to (\textbf{Ans} \times \textbf{MS})$	Monitoring Answers
ρ	$\in$	$\underline{\textbf{Env} = \textbf{Ide} \to \textbf{V}}$	Environments
κ	$\in$	$\textbf{Kont} = \textbf{V} \to \overline{\textbf{Ans}}$	Expression continuations

Valuation Functionals (Val):

$\overline{\mathcal{G}_\lambda} : \textbf{T}_\lambda \to \textbf{T}_\lambda$

where $\textbf{T}_\lambda = \overline{\textbf{Exp}} \to \textbf{Env} \to \textbf{Kont} \to \overline{\textbf{Ans}}$

$\overline{\mathcal{G}_\lambda}\ \overline{\mathcal{E}}\ \bar{e}\ \rho\ \kappa =$ case $\bar{e}$ of

$[\![\{\mu\}{:}\bar{e}']\!]$: $(\overline{\mathcal{E}}\ [\![\bar{e}']\!]\ \rho\ \kappa_{post}) \circ \mathit{updPre}$

where $\mathit{updPre} : \textbf{MS} \to \textbf{MS}$

$\mathit{updPre} = \mathcal{M}^{\mathcal{E}}_{pre}\ [\![\mu]\!]\ [\![\bar{e}']\!]\ \rho$

$\kappa_{post} = \{\lambda v.\ (\kappa\ v) \circ \mathit{updPost}\}$

where $\mathit{updPost} : \textbf{MS} \to \textbf{MS}$

$\mathit{updPost} = \mathcal{M}^{\mathcal{E}}_{post}\ [\![\mu]\!]\ [\![\bar{e}']\!]\ \rho\ v$

else : $\mathcal{G}_\lambda\ \overline{\mathcal{E}}\ \bar{e}\ \rho\ \kappa$ $\quad$ ($\mathcal{G}_\lambda$ is defined in Figure 2)

Figure 3: Parameterized Monitoring Semantics for $\mathcal{L}_\lambda$

language specification—it consists of three parts: the *monitor syntax*, the *monitor algebras*, and the *monitoring functions*.

Definition 5.1 (Monitor Specification) *A* monitor specification *for a standard continuation semantics* Den *is a triple* Mon = ⟨MSyn, MAlg, MFun⟩, *where:*

- MSyn *denotes the* monitor syntax *which defines the syntactic domain of monitor annotations.*
- MAlg *is the set of* monitor algebras *which includes the monitor state domain.*
- MFun *denotes the set of pairs of* monitoring functions, *one pair for each valuation function in* Den.

Monitor syntax (MSyn):
$i \in$ **Ann**
$i ::= \mathtt{A} \mid \mathtt{B}$

Monitor Algebras (MAlg):
I. Monitor state
$\sigma \in \mathbf{MS} = (\mathbb{N} \times \mathbb{N})$
Operations

$inc\_A : \mathbf{MS} \rightarrow \mathbf{MS}$
$inc\_A\ \langle n_1, n_2 \rangle = \langle n_1 + 1, n_2 \rangle$

$inc\_B : \mathbf{MS} \rightarrow \mathbf{MS}$
$inc\_B\ \langle n_1, n_2 \rangle = \langle n_1, n_2 + 1 \rangle$

$init\_state : \mathbf{MS}$
$init\_state = \langle 0, 0 \rangle$

Monitoring functions (MFun):
$\mathcal{M}^{\mathcal{E}}_{pre} : \mathbf{Ann} \rightarrow \mathbf{Exp} \rightarrow \mathbf{Env} \rightarrow \mathbf{MS} \rightarrow \mathbf{MS}$
$\mathcal{M}^{\mathcal{E}}_{pre}\ [\![\mathtt{A}]\!]\ [\![e]\!]\ \rho\ \sigma = inc\_A\ \sigma$
$\mathcal{M}^{\mathcal{E}}_{pre}\ [\![\mathtt{B}]\!]\ [\![e]\!]\ \rho\ \sigma = inc\_B\ \sigma$

$\mathcal{M}^{\mathcal{E}}_{post} : \mathbf{Ann} \rightarrow \mathbf{Exp} \rightarrow \mathbf{Env} \rightarrow \mathbf{V} \rightarrow \mathbf{MS} \rightarrow \mathbf{MS}$
$\mathcal{M}^{\mathcal{E}}_{post}\ [\![i]\!]\ [\![e]\!]\ \rho\ v\ \sigma = \sigma$

Figure 4: Specification of a simple profiler

The similarity between the monitor semantics and a standard language semantics is not surprising. Indeed, monitor semantics consists of a language (monitor syntax) to specify monitoring operations, monitor domains as value spaces in monitoring semantics, and monitoring functions to map a language abstract syntax annotated with monitor syntax to its "monitoring meaning" drawn from semantic and monitor domains.

As an example of a complete monitor specification, Figure 4 shows the specification of a profiler which performs the simple chore of counting the number of times an expression with either annotation "A" or "B" is evaluated. The first component of the specification is the annotation syntax, defined in the same way as the abstract syntax of any programming language—i.e. using conventional BNF notation. The state algebra is shown next, consisting of a pair of counters together with operations that increment those counters. Last is the specification of the monitoring functions which perform the obvious tasks: the pre-monitoring function increments the appropriate counter, and the post-monitoring function does nothing.

The following program illustrates the use of the profiler:

```
letrec fac = lambda x . if (x = 0)
                        then {A}:1
                        else {B}:(x * fac(x - 1))
  in fac 5
```

Each branch of the conditional has been labeled with a different monitoring annotation. The profiling information gathered by monitoring this program with the above monitor would be $\sigma = \langle 1, 5 \rangle$.

6 Monitors May be Composed

With the simple constraint that the annotation syntaxes are disjoint, monitors may be composed in such a way that they are guaranteed not to interfere with each other. The method is simple: construct the first monitor from the original continuation semantics as described earlier, then treat the result as *a new continuation semantics*, and repeat the same procedure in constructing the second monitor. The new answer domain would then be:

$$\overline{\overline{\mathbf{Ans}}} = \overline{\mathbf{Ans} \times \mathbf{MS}_1} = \mathbf{MS}_2 \rightarrow ((\mathbf{Ans} \times \mathbf{MS}_1) \times \mathbf{MS}_2)$$

This process may be repeated an arbitrary number of times. Furthermore, this technique could be used for building extensible programming development environment (a collection of monitors, that is) in an effective, safe and straightforward way. Another interesting consequence of cascading monitors is that a monitor could monitor the behavior of the monitors before it in the cascade by providing its monitoring functions with the monitor states.

As an example of this technique, Figure 5 shows the result of starting with the parameterized monitoring semantics in Figure 3, treating it as a standard continuation semantics, and deriving a new parameterized monitoring semantics. The new monitor is assumed to have monitor annotations μ_2 and monitoring functions: $\mathcal{M}_{2_{pre}}^{\mathcal{E}}$ and $\mathcal{M}_{2_{post}}^{\mathcal{E}}$.

7 Soundness

This section discusses the consistency of the monitoring semantics with the original standard semantics. It is shown that monitoring semantics is a meaning preserving enhancement to the standard semantics. For the simplicity of the presentation we will consider a standard semantics that has only one valuation functional.

We begin by defining some properties of the standard continuation semantics which we will take as assumptions in the main theorem.

Since our approach is based on continuation semantics it is important to precisely articulate the characteristics of continuation semantics which are relevant to our framework. Obviously, the valuation functionals should be specified as a standard continuation semantics as defined earlier. In particular, all calls to functions which return a final answer (e.g. valuation functions, continuations) must be "tail-recursive": values are only passed forward. In other words, we require that final answers must not be used as an argument to other calls. As can be seen, this implies that each call to a valuation function or a continuation is expected to return the final answer of the semantics (no further evaluation is expected after the call). Indeed, these characteristics correspond to Reynolds' definition of continuations [Rey72], in particular to his notion of *serious* functions. Note that we consider the monitoring functions to be *trivial* (this is the reason why they are not expressed in continuation style).

The following definition ensures that the standard semantics is oblivious to monitor annotations.

Abstract Syntax (Syn):

$\bar{\bar{e}}$	$\in$	$\overline{\overline{\textbf{Exp}}}$	doubly annotated expressions
μ_2	$\in$	$\textbf{Ann}$	monitor annotations of second monitor

$$\overline{\overline{\mathcal{H}}}(\overline{\overline{\textbf{Exp}}}) = \overline{\mathcal{H}}(\overline{\overline{\textbf{Exp}}}) \cup \{\{\mu_2\}{:}\bar{\bar{e}} \mid \mu_2 \in \textbf{Ann}_2,\ \bar{\bar{e}} \in \overline{\overline{\textbf{Exp}}}\} \qquad (\overline{\mathcal{H}} \text{ is defined in Figure 3})$$

Semantic Algebras (Alg):

b	$\in$	$\textbf{Bas} = \textbf{Int} + \textbf{Bool} + \ldots$	Basic values
		$\textbf{Fun} = \textbf{V} \to \textbf{Kont} \to \textbf{MS}_1 \to \overline{\overline{\textbf{Ans}}}$	Function values
v	$\in$	$\textbf{V} = \textbf{Bas} + \textbf{Fun}$	Denotable values
a	$\in$	$\overline{\overline{\textbf{Ans}}} = \textbf{MS}_2 \to ((\textbf{Ans} \times \textbf{MS}_1) \times \textbf{MS}_2)$	Monitoring Answers
ρ	$\in$	$\textbf{Env} = \textbf{Ide} \to \textbf{V}$	Environments
κ	$\in$	$\textbf{Kont} = \textbf{V} \to \textbf{MS}_1 \to \overline{\overline{\textbf{Ans}}}$	Expression continuations

Valuation Functionals (Val):

$$\overline{\overline{\mathcal{G}_\lambda}} : \textbf{T}_\lambda \to \textbf{T}_\lambda \text{ where } \textbf{T}_\lambda = \overline{\overline{\textbf{Exp}}} \to \textbf{Env} \to \textbf{Kont} \to \textbf{MS}_1 \to \overline{\overline{\textbf{Ans}}}$$

$$\begin{array}{lll}
\overline{\overline{\mathcal{G}_\lambda}}\ \overline{\overline{\mathcal{E}}}\ \bar{\bar{e}}\ \rho\ \kappa\ \sigma_1 = & \text{case } \bar{\bar{e}} \text{ of} & \\
\quad [\![\{\mu_2\}{:}\bar{\bar{e}}']\!] & : & (\overline{\overline{\mathcal{E}}}\ [\![\bar{\bar{e}}']\!]\ \rho\ \kappa_{post}\ \sigma_1) \circ updPre \\
 & & \text{where } \ updPre\ : \textbf{MS}_2 \to \textbf{MS}_2 \\
 & & \qquad updPre = \mathcal{M}2^{\mathcal{E}}_{pre}\ [\![\mu_2]\!]\ [\![\bar{\bar{e}}']\!]\ \rho \\
 & & \qquad \kappa_{post} = \ \{\lambda v\ \sigma_1'.\ (\kappa\ v\ \sigma_1') \circ updPost\} \\
 & & \qquad\qquad \text{where } updPost : \textbf{MS}_2 \to \textbf{MS}_2 \\
 & & \qquad\qquad\qquad updPost = \mathcal{M}2^{\mathcal{E}}_{post}\ [\![\mu_2]\!]\ [\![\bar{\bar{e}}']\!]\ \rho\ v \\
\quad \text{else} & : & \overline{\mathcal{G}_\lambda}\ \overline{\overline{\mathcal{E}}}\ \bar{\bar{e}}\ \rho\ \kappa\ \sigma_1 \qquad (\overline{\mathcal{G}_\lambda} \text{ is defined in Figure 3})
\end{array}$$

Figure 5: Derived monitoring semantics for $\overline{\mathcal{L}_\lambda}$

Definition 7.1 *Let* Den *be a denotational specification with valuation functional* $\mathcal{G}$ *and syntax* S. *Given any program* $s \in$ S *and* $\bar{s} \in \overline{\text{S}}$, *such that* $\bar{s}$ *is* s *augmented with monitor annotations, then* $\mathcal{G}$ *is said to be* oblivious to monitor annotations *iff:*

$$\begin{array}{lll}
(fix\ \mathcal{G})\ [\![s]\!] = (fix\ \mathcal{G}_{obl})\ [\![\bar{s}]\!] & & \\
\text{where } \mathcal{G}_{obl}\ \mathcal{V}\ \bar{s} = \ \text{case } \bar{s} \text{ of} & & \\
\qquad [\![\{\mu\}{:}\bar{s}']\!] & : & \mathcal{V}\ [\![\bar{s}']\!] \\
\qquad else & : & \mathcal{G}\ \mathcal{V}\ \bar{s}
\end{array}$$

In other words, if $\mathcal{G}$ is oblivious to monitor annotations then it is guaranteed to disregard monitor annotations provided that it is enhanced according to the above construction of $\mathcal{G}_{obl}$.

Assumption 7.2 *A denotational specification* Den *is said to be* well-specified *with respect to monitoring semantics iff:*

1. Den *is specified in continuation style (as described above).*
2. Den *is parameterized with the final answer (Definition 3.3).*
3. Den*'s valuation functionals are oblivious to monitor annotations (Definition 7.1).*

Note that Assumption 7.2 does not result in any significant loss of generality since it closely corresponds to the conventional style of specifying continuation semantics (see Figure 2 as an example). We now state and prove several lemmas which lead to our main result.

Notation we write "v / Ans" to denote that v is parameterized with an answer algebra Ans. Also, we use Ans_{std} and the corresponding Ans_{mon} (Definition 4.1) to denote the standard and monitoring answer algebras respectively. Finally, we define an inverse function for the answer transformer θ (see Definition 4.1):

$$\theta^{-1}\ \bar{\alpha} = (\bar{\alpha}\ \sigma)\downarrow_1 \quad (\sigma \textit{ is arbitrary})$$

such that $(\theta^{-1} \circ \theta) = Id$

Recall that a well specified semantics is parameterized with respect to the final answer. We make the following observation:

Lemma 7.3 *Let* Den *be a well-specified semantics with valuation functional $\mathcal{G}$ and syntax* S. *Then for all $s \in$ S, $a_j \in$ A$_j$ (i.e. the semantic context: environment, store etc.), $\kappa \in$* **Kont***:*

$$(fix\ \mathcal{G})\ [\![s]\!]\ a{*}\ \kappa\ /\ \text{Ans}_{std} \;=\; \theta^{-1}((fix\ \mathcal{G})\ [\![s]\!]\ a{*}\ \kappa)\ /\ \text{Ans}_{mon}$$

Proof: Let $\alpha \in$ **Ans** be the result of $((fix\ \mathcal{G})\ [\![s]\!]\ a{*}\ \kappa\ /\ \text{Ans}_{std})$ then according to Definition 4.1 of the monitoring answer algebra the result of $((fix\ \mathcal{G})\ [\![s]\!]\ a{*}\ \kappa\ /\ \text{Ans}_{mon})$ is $(\theta\ \alpha) \in \overline{\textbf{Ans}}$. Consequently

$$\begin{array}{lcl} (fix\ \mathcal{G})\ [\![s]\!]\ a{*}\ \kappa\ /\ \text{Ans}_{std} & = & \theta^{-1}((fix\ \mathcal{G})\ [\![s]\!]\ a{*}\ \kappa)\ /\ \text{Ans}_{mon} \\ \alpha\ /\ \text{Ans}_{std} & = & (\theta^{-1}\ (\theta\ \alpha))\ /\ \text{Ans}_{mon} \\ & = & \alpha\ /\ \text{Ans}_{mon} \end{array}$$

At this point we are still have to prove that the last line is true; that is, to show to what extent standard values are equal to monitored values. To do so, let us first notice that in the equation of Lemma 7.3, the left hand side is parameterized with respect to answer algebra different from the right hand side. Consequently, if α is an element of a domain which depends on the answer domain then the results of both sides of the equation are of different domains. To give some intuition behind this let us consider **Ans = Bas+Fun** the standard answer domain for $\mathcal{L}_\lambda$. A domain like **Fun = V $\rightarrow$ Kont $\rightarrow$ Ans** does change when the answer algebra of the standard semantics is enriched in the monitoring semantics, that is, **Fun = V $\rightarrow$ Kont $\rightarrow$ $\overline{\textbf{Ans}}$**. This can be easily verified by the domain equations in Figures 2 and Figures 3. However, a domain like the basic values domain **Bas** is the same domain in both the standard and the monitoring semantics. Indeed, all domains that are not defined in terms of the answer domain do not change as a result of replacing the answer algebra. For simplicity we will consider only an answer domain which is not recursive (e.g. first order values)[3]. □

The next lemma proves the correspondence between the standard semantics and the monitoring semantics both parameterized with the monitoring answer algebra. But first, we define a logical relation and introduce a lemma to facilitate the proof. We define a relation to equate two monitoring semantics answers $\bar{\alpha}_i : \overline{\mathbf{Ans}}$

Definition 7.4

$$\bar{\alpha}_1 \;\mathcal{R}\; \bar{\alpha}_2 \quad\Longrightarrow\quad \forall \sigma_1 \in \mathbf{MS}\; \sigma_2 \in \mathbf{MS}.\;\; (\bar{\alpha}_1\; \sigma_1){\downarrow}_1 = (\bar{\alpha}_2\; \sigma_2){\downarrow}_1$$

The relation above is invariant to a certain kind of operations. The invariance is captured by the following lemma.

Lemma 7.5 *Let v be a monitor state transformer, i.e., $v : \mathbf{MS} \to \mathbf{MS}$. Then*

$$\bar{\alpha}_1 \;\mathcal{R}\; \bar{\alpha}_2 \Longleftrightarrow \bar{\alpha}_1 \;\mathcal{R}\; (\bar{\alpha}_2 \circ v)$$

Proof:

$$\begin{array}{rcll}
\bar{\alpha}_1 & \mathcal{R} & \bar{\alpha}_2 & \\
(\bar{\alpha}_1\; \sigma_1){\downarrow}_1 & = & (\bar{\alpha}_2\; \sigma_2){\downarrow}_1 & \textit{(by Definition 7.4)} \\
(\bar{\alpha}_1\; \sigma_1){\downarrow}_1 & = & (\bar{\alpha}_2\; (v\; \sigma_2)){\downarrow}_1 & \textit{(σ is universally quantified in Definition 7.4)} \\
(\bar{\alpha}_1\; \sigma_1){\downarrow}_1 & = & ((\bar{\alpha}_2\; \circ\; v)\; \sigma_2){\downarrow}_1 & \textit{(by definition of composition)} \\
\bar{\alpha}_1 & \mathcal{R} & (\bar{\alpha}_2\; \circ v) &
\end{array}$$

In other words, the $\mathcal{R}$ relation is invariant to composition of monitor state updates. □

Lemma 7.6 *Let* Den *be a well-specified semantics with valuation functional $\mathcal{G}$ and syntax* S *and let the derived monitoring semantics for* Den *be with valuation functional $\overline{\mathcal{G}}$ and annotated syntax* $\overline{\text{S}}$. *Given any $s \in$ S and $\bar{s} \in \overline{\text{S}}$, such that $\bar{s}$ is s augmented with the appropriate monitor annotations, then for all $a_j \in \mathbf{A}_j$, $\kappa \in \mathbf{Kont}$ and $\sigma \in \mathbf{MS}$:*

$$((\mathit{fix}\; \mathcal{G})\; [\![s]\!]\; a{*}\; \kappa\; /\; \text{Ans}_{mon})\; \mathcal{R}\; ((\mathit{fix}\; \overline{\mathcal{G}})\; [\![\bar{s}]\!]\; a{*}\; \kappa\; /\; \text{Ans}_{mon})$$

Proof: Since Den is well-specified then according to Assumption 7.1

$$(\mathit{fix}\; \mathcal{G})\; [\![s]\!] = (\mathit{fix}\; \mathcal{G}_{obl})\; [\![\bar{s}]\!]$$

Consequently, we can use $\mathcal{G}_{obl}$ instead of $\mathcal{G}$ for the proof. The proof is a fixpoint induction over $\overline{\mathcal{G}}$ and $\mathcal{G}_{obl}$ functionals. (Note that we will not mention the parameterization by the answer algebras in the derivation since both sides are parameterized with the same answer algebra.)
Induction Basis Let $\mathcal{G}_{obl} = \perp$ and $\overline{\mathcal{G}} = \perp$ then $(\perp)\; \mathcal{R}\; (\perp)$ trivially follows.
Induction Assumption Let $(\mathcal{V} = \mathit{fix}\; \mathcal{G}_{obl})$ and $(\overline{\mathcal{V}} = \mathit{fix}\; \overline{\mathcal{G}})$. Then,

$$(\mathcal{V}\; [\![\bar{s}]\!]\; a{*}\; \kappa)\; \mathcal{R}\; (\overline{\mathcal{V}}\; [\![\bar{s}]\!]\; a{*}\; \kappa)$$

Induction Step (by structural induction over the syntax)
Annotated syntax. For the annotated syntax we have:

[3]The proof can be generalized to cover a recursive answer domain by using a recursive congruence relation rather than equality.

$$\begin{array}{rcll}
(\mathcal{G}_{obl}\ \mathcal{V}\ [\![\{\mu\}{:}\bar{s}]\!]\ a{*}\ \kappa) & \mathcal{R} & (\overline{\mathcal{G}}\ \overline{\mathcal{V}}\ [\![\{\mu\}{:}\bar{s}]\!]\ a{*}\ \kappa) & \\
(\mathcal{V}[\![\bar{s}]\!]\ a{*}\ \kappa) & \mathcal{R} & (\overline{\mathcal{V}}[\![\bar{s}]\!]\ a{*}\ \{\lambda\ \iota{*}.\ (\kappa\ \iota{*}) \circ \mathit{updPost}\}) \circ \mathit{updPre} & \textit{(by definition of } \overline{\mathcal{G}} \textit{ and } \mathcal{G}_{obl}\textit{)} \\
(\mathcal{V}[\![\bar{s}]\!]\ a{*}\ \kappa) & \mathcal{R} & (\overline{\mathcal{V}}[\![\bar{s}]\!]\ a{*}\ \{\lambda\ \iota{*}.\ (\kappa\ \iota{*})\}) & \textit{(by Lemma 7.5)} \\
(\mathcal{V}[\![\bar{s}]\!]\ a{*}\ \kappa) & \mathcal{R} & (\overline{\mathcal{V}}[\![\bar{s}]\!]\ a{*}\ \kappa) & \textit{(}\eta\textit{-reduction)}
\end{array}$$

The reader can verify that *updPre* and *updPost* are monitor state transformers (see Definition 4.2). Furthermore, since both functionals are specified in continuation style, we know that every call to either the valuation function or a continuation returns the final answer (no further evaluation is expected). Therefore, by composing the *updPre* or *updPost* with the continuation or the valuation function we essentially compose a monitor state update to the final answer. By Lemma 7.5 we know that the composition of such updates are invariant to the relation, therefore we can safely discard these updates. Finally, we can use the induction assumption to establish the relation.

Non annotated syntax. Notice that the valuation equations of $\overline{\mathcal{G}}$ and $\mathcal{G}_{obl}$ for non-annotated syntax are derived from the same functional $\mathcal{G}$:

$$\begin{array}{rcll}
(\mathcal{G}_{obl}\ \mathcal{V}\ [\![\bar{s}]\!]\ a{*}\ \kappa) & \mathcal{R} & (\overline{\mathcal{G}}\ \overline{\mathcal{V}}\ [\![\bar{s}]\!]\ a{*}\ \kappa) & \\
(\mathcal{G}\ \mathcal{V}\ [\![\bar{s}]\!]\ a{*}\ \kappa) & \mathcal{R} & (\mathcal{G}\ \overline{\mathcal{V}}\ [\![\bar{s}]\!]\ a{*}\ \kappa) & \textit{(by definition of } \overline{\mathcal{G}} \textit{ and } \mathcal{G}_{obl}\textit{)}
\end{array}$$

The only difference between the valuation equations of $\overline{\mathcal{G}}$ and the valuation equations of $\mathcal{G}_{obl}$ is that $\mathcal{V}$ is used in $\mathcal{G}_{obl}$ equations and $\overline{\mathcal{V}}$ is used in $\overline{\mathcal{G}}$ equations. We consider two cases: atomic syntactic terms (e.g. constants, identifiers) and compound syntactic constructs (e.g. conditional, application).

Atomic terms. The basis cases are the valuation equations for syntactic terms without sub-components. Since there are no sub-components then there are no recursive calls to the valuation functions. Consequently, whether we provide $\mathcal{G}$ with $\overline{\mathcal{V}}$ or $\mathcal{V}$ is irrelevant. Thus:

$$(\mathcal{G}\ \mathcal{V}\ [\![\bar{s}]\!]\ a{*}\ \kappa) \quad = \quad (\mathcal{G}\ \overline{\mathcal{V}}\ [\![\bar{s}]\!]\ a{*}\ \kappa)$$

Note that we assume that basis cases do exist. This is a valid assumption since we are not interested in non-terminating valuation functionals.

Compound terms. In this case the equations are for syntactic terms with sub-components Consequently, we expect these semantic equations to include recursive calls to the valuation functional $\overline{\mathcal{V}}$ and $\mathcal{V}$. The subtle point to notice is that using the inductive assumption we can replace all $\overline{\mathcal{V}}$ occurrences on the right hand side with $\mathcal{V}$ (or vice versa) making both sides equal.

We have exhausted all cases thus Lemma 7.6 is true. □

Finally, we can now state the soundness theorem:

Theorem 7.7 (Soundness) *Let* Den *be a well-specified semantics with valuation functional* $\mathcal{G}$ *and syntax* S. *Also, let the derived monitoring semantics for* Den *have a valuation functional* $\overline{\mathcal{G}}$ *and annotated syntax* $\overline{\text{S}}$. *Given any* $s \in \text{S}$ *and* $\bar{s} \in \overline{\text{S}}$, *such that* $\bar{s}$ *is* s *augmented with the appropriate monitor annotations, then for all values* $a_j \in \text{A}_j$ *(i.e. the semantic context: environment, store etc.),* $\kappa \in \mathbf{Kont}$ *and* $\sigma \in \mathbf{MS}$*:*

$$(\mathit{fix}\ \mathcal{G})\ [\![s]\!]\ a{*}\ \kappa\ /\ \text{Ans}_{std} \quad = \quad ((\mathit{fix}\ \overline{\mathcal{G}})\ [\![\bar{s}]\!]\ a{*}\ \kappa\ \sigma){\downarrow_1}\ /\ \text{Ans}_{mon}$$

Proof: According to Lemma 7.3:

$$(\mathit{fix}\ \mathcal{G})\ [\![s]\!]\ a{*}\ \kappa\ /\ \text{Ans}_{std} \quad = \quad \theta^{-1}((\mathit{fix}\ \mathcal{G})\ [\![s]\!]\ a{*}\ \kappa)\ /\ \text{Ans}_{mon}$$

Also, according to Lemma 7.6:

$$\theta^{-1}((fix\ \mathcal{G})\ [\![s]\!]\ a*\ \kappa)\ /\ \mathrm{Ans}_{mon} = \theta^{-1}((fix\ \overline{\mathcal{G}})\ [\![\bar{s}]\!]\ a*\ \kappa)\ /\ \mathrm{Ans}_{mon}$$

Then, by transitivity:

$$(fix\ \mathcal{G})\ [\![s]\!]\ a*\ \kappa\ /\ \mathrm{Ans}_{std} \quad = \quad \theta^{-1}((fix\ \overline{\mathcal{G}})\ [\![\bar{s}]\!]\ a*\ \kappa)\ /\ \mathrm{Ans}_{mon}$$

□

8 Examples: a Tracer, a Profiler, a Demon and a collecting monitor for $\mathcal{L}_\lambda$

In this section we study four monitor specifications for $\mathcal{L}_\lambda$—a tracer, a profiler, a demon and a collecting monitor. The specifications follow the format of a monitor specification discussed in Section 5. In addition, we have implemented all the examples in Haskell, and provide running output to demonstrate the results.

It is important to note that this framework can also support interactive monitors (e.g. symbolic debuggers, steppers) by providing an input as well as an output stream to and from the monitor. The implementation of interactive tools is discussed in [Kis91].

Profiler. The specification of a *program execution profiler* is shown in Figure 6. The profiler counts the number of times that all named functions are called. An environment domain is introduced that maps a function name to its corresponding counter value: $\rho_c \in \mathbf{CEnv} = \mathbf{Ide} \rightarrow \mathbb{N}$. The operations for this domain include usual environment operations: environment lookup, denoted by $\rho_c(f)$; environment update, denoted by $\rho_c[f \mapsto n]$; and initial environment—*initEnv*. Also, *incCtr* increments the corresponding counter for a given function name (or initializes it to 1 if the function was never used.) The only algebra needed for the profiler is the counter environment algebra (notice that it can also serve as the result of the profiler). This suggests that the state of this monitor is the domain **CEnv**.

We profile by incrementing the counter associated with a function whenever the function *body* is evaluated. We therefore annotate each function body with its function name. The annotation will trigger the profiler semantics whenever the function body is evaluated and the annotation syntax will provide the profiler with the name of the function. All named functions should be labeled this way.

For an example of the implemented profiler in action, consider the annotated program (assuming -,* and = are primitives):

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

The profiler semantics would provide the following information in the counter environment:

$$[\texttt{fac} \mapsto 4,\ \texttt{mul} \mapsto 3]$$

Tracer. The specification of a *program execution tracer* is shown in Figure 7. Note that the tracer state consists of an output channel and a trace level indicator. We treat the output channel as an abstract datatype with operations *addStream* to add a new string to a given stream, and *initStream* which provides the initial stream. The trace level indicator is simply an integer. The tracer is designed

Monitor Syntax (MSyn):

$f \in$ **Ide** Function name (defined in Fig. 2)

Monitor Algebras:

I. Monitor State: Counter Environment
Domain **MS** = **CEnv**

II. Counter Environment
Domain $\rho_c \in$ **CEnv** = **Ide** $\rightarrow \mathbb{N}$
Operations :

$incCtr :$ **Ide** $\rightarrow$ **CEnv** $\rightarrow$ **CEnv**

$incCtr\ [\![f]\!]\ \rho_c \ = \ \rho_c[f \mapsto n]$
where $n = [\![f]\!] \in dom(\rho_c) \ \rightarrow \ \rho_c(f) + 1,\ 1$

$initEnv :$ **CEnv**

$initEnv \ = \lambda x.\ \bot_{Nat}$

Monitoring functions:

$\mathcal{M}^{\mathcal{E}}_{pre} :$ **Ide** $\rightarrow$ **Exp** $\rightarrow$ **Env** $\rightarrow$ **MS** $\rightarrow$ **MS**

$\mathcal{M}^{\mathcal{E}}_{pre}[\![f]\!]\ [\![e]\!]\ \rho\ \rho_c = incCtr\ [\![f]\!]\ \rho_c$

$\mathcal{M}^{\mathcal{E}}_{post} :$ **Ide** $\rightarrow$ **Exp** $\rightarrow$ **Env** $\rightarrow$ **V** $\rightarrow$ **MS** $\rightarrow$ **MS**

$\mathcal{M}^{\mathcal{E}}_{post}[\![f]\!]\ [\![e]\!]\ \rho\ v\ \rho_c = \rho_c$

Figure 6: Profiler specification

to print tracing information before and after evaluating any function body annotated with an element from the syntax domain **Fh** (this provides the tracer with the name of the function and its arguments). As an example of the implemented tracer in action, consider the annotated program:

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

The tracer provides the following information on the output channel:

```
[FAC receives (3)]
|    [FAC receives (2)]
|    |    [FAC receives (1)]
|    |    |    [FAC receives (0)]
|    |    |    [FAC returns 1]
|    |    |    [MUL receives (1 1)]
|    |    |    [MUL returns 1]
|    |    [FAC returns 1]
|    |    [MUL receives (2 1)]
|    |    [MUL returns 2]
|    [FAC returns 2]
|    [MUL receives (3 2)]
|    [MUL returns 6]
[FAC returns 6]
```

Monitor syntax:

$fh \in \mathbf{Fh}$ Function header, where $fh ::= f(x_1, \ldots, x_n)$

$x, f \in \mathbf{Ide}$ (defined by the language abstract syntax)

Monitor Algebras:

I. State

Domain $\sigma \in \mathbf{MS} = \mathbf{OutChan} \times \mathbb{N}$

Operations

$initState : \mathbf{MS}$

$initState = \langle initStream, 0 \rangle$

II. Output channel

Domain $o \in \mathbf{OutChan} = \mathbf{Stream}$

Operations

$printChan : \mathbf{String} \rightarrow \mathbb{N} \rightarrow \mathbf{OutChan} \rightarrow \mathbf{OutChan}$

$printChan\ x\ n\ o\ = addStream\ x\ (indent\ n\ o)$

$indent : \mathbb{N} \rightarrow \mathbf{OutChan} \rightarrow \mathbf{OutChan}$

$indent\ n\ o\ =\ spaces\ n\ (addStream\ \text{"}\mathtt{L_F}\text{"}\ o)$

where $spaces\ 0\ o = o$

$spaces\ n\ o = spaces\ (n-1)\ (addStream\ \text{"}\mathtt{|}\ \ \text{"}\ o)$

III. Streams

Domain $s \in \mathbf{Stream}$

Operations

$addStream : \mathbf{String} \rightarrow \mathbf{Stream} \rightarrow \mathbf{Stream}$

$initStream : \mathbf{Stream}$

Monitoring functions:

$\mathcal{M}^{\mathcal{E}}_{pre} : \mathbf{Fh} \rightarrow \mathbf{Exp} \rightarrow \mathbf{Env} \rightarrow \mathbf{MS} \rightarrow \mathbf{MS}$

$\mathcal{M}^{\mathcal{E}}_{pre}[\![f(x_1, \ldots, x_n)]\!]\ [\![e]\!]\ \rho\ \langle o, n \rangle =$

$\langle printChan\ (\text{"}\mathtt{[}\text{"} +\!\!+ [\![f]\!] +\!\!+ \text{"}\mathtt{receives\ \ (}\text{"} +\!\!+ ToStr(\rho(x_1)) +\!\!+ \ldots +\!\!+ ToStr(\rho(x_n)) +\!\!+ \text{"}\mathtt{)]}\text{"})\ n\ o,\ n+1 \rangle$

$\mathcal{M}^{\mathcal{E}}_{post} : \mathbf{Fh} \rightarrow \mathbf{Exp} \rightarrow \mathbf{Env} \rightarrow \mathbf{V} \rightarrow \mathbf{MS} \rightarrow \mathbf{MS}$

$\mathcal{M}^{\mathcal{E}}_{post}[\![f(x_1, \ldots, x_n)]\!]\ [\![e]\!]\ \rho\ v\ \langle o, n \rangle =$

$\langle printChan\ (\text{"}\mathtt{[}\text{"} +\!\!+ [\![f]\!] +\!\!+ \text{"}\mathtt{returns}\text{"} +\!\!+ ToStr(v) +\!\!+ \text{"}\mathtt{]}\text{"})\ (n-1)\ o,\ n-1 \rangle$

Figure 7: Fancy tracer specification

Event Monitoring—Demons Often a programmer wants to invoke monitoring actions if a specific execution event (such as an assignment to a variable) occurs. A simple mechanism called a *demon* is proposed in [DMS84] for event monitoring. This section discusses specifying demons in monitoring semantics.

Magpie [DMS84], an interactive environment for Pascal, supports an event monitoring mechanism that monitors events associated with a particular identifier. Our approach improves on Magpie in that it provides a mechanism to specify demons for *any* semantic event. In particular, the specification of a demon in monitoring semantics follows quite naturally: first we need to identify all the program points where the event might occur and label their corresponding syntactic term with a demon annotation. Second, knowing that the monitor will be called at these points, we need to specify the criteria (based on the semantic context provided) for the events to trigger an action. Finally, we need to specify

Monitor Syntax (MSyn):

$p \in$ **Ide** Program Point Label

Monitor Algebras:

I. Name set

Domain $\sigma \in$ **MS** = {**Ide**}

Operations (usual set operations)

Monitoring functions:

$\mathcal{M}^{\mathcal{E}}_{pre} : \textbf{Ide} \rightarrow \textbf{Exp} \rightarrow \textbf{Env} \rightarrow \textbf{MS} \rightarrow \textbf{MS}$

$\mathcal{M}^{\mathcal{E}}_{pre}[\![p]\!]\ [\![e]\!]\ \rho\ \sigma = \sigma$

$\mathcal{M}^{\mathcal{E}}_{post} : \textbf{Ide} \rightarrow \textbf{Exp} \rightarrow \textbf{Env} \rightarrow \textbf{V} \rightarrow \textbf{MS} \rightarrow \textbf{MS}$

$\mathcal{M}^{\mathcal{E}}_{post}[\![p]\!]\ [\![e]\!]\ \rho\ v\ \sigma = \quad sorted?\ v \ \rightarrow\ \sigma, \{[\![p]\!]\} \cup \sigma$

where

```
  sorted? (x : xs) =
      case xs of
          (y : ys)  :  (x ≤ y) & (sorted? xs)
          Nil       :  True
  sorted? Nil = True
```

Figure 8: Simple demon

the actions to be taken when a demon event occurs. These three steps correspond directly to the specification of a monitor.

Figure 8 presents the specification of a demon that checks for unsorted lists. For simplicity the monitor will only return the name of the functions where unsorted lists were encountered. The monitor and the example that follows are self explanatory. For the following program

```
letrec inclist = lambda l. lambda acc. if (l=[]) then acc else inclist (tl l) ((hd l)+1):acc in
   letrec l1 = {l1}:(inclist [1,10,100] []) in
      letrec l2 = {l2}:(inclist l1 []) in
         letrec l3 = {l3}:(inclist l2 [])
            in l3
```

The demon returns the following information in its state:

$$\sigma = \{l1, l3\}$$

Collecting Monitor A collecting interpretation of expressions is an interpretation of a program that allows one to answer questions of the sort: "What are all possible values to which an expression might evaluate during program execution?"[HY88]. A collecting monitor *à la* collecting interpretation is defined in Figure 9. Each expression that we want to monitor is tagged with an identifier. The post monitoring function updates the environment which keeps the set of values for each tagged expression. For the following program:

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

Monitor Syntax (MSyn):

$x \in$ **Ide** *Name tags*

Monitor Algebras:

I. State: Interpretations Environment

Domain $\sigma \in \mathbf{MS} = \mathbf{Ide} \rightarrow \{\mathbf{V}\}$

Operations (*usual*)

Monitoring functions:

$$\mathcal{M}^{\mathcal{E}}_{pre} : \mathbf{Ide} \rightarrow \mathbf{Exp} \rightarrow \mathbf{Env} \rightarrow \mathbf{MS} \rightarrow \mathbf{MS}$$
$$\mathcal{M}^{\mathcal{E}}_{pre}[\![x]\!]\,[\![e]\!]\,\rho\,\sigma = \sigma$$
$$\mathcal{M}^{\mathcal{E}}_{post} : \mathbf{Ide} \rightarrow \mathbf{Exp} \rightarrow \mathbf{Env} \rightarrow \mathbf{V} \rightarrow \mathbf{MS} \rightarrow \mathbf{MS}$$
$$\mathcal{M}^{\mathcal{E}}_{post}[\![x]\!]\,[\![e]\!]\,\rho\,v\,\sigma = \sigma[x \mapsto \sigma(x) \cup \{v\}]$$

Figure 9: Collecting Interpretation Monitor

The collecting monitor provides the following information in its final state:

$$[\texttt{test} \mapsto \{True, False\},\ \texttt{n} \mapsto \{1,2,3\}]$$

9 Implementation Issues

9.1 Partial Evaluation

A functional implementation of the monitored semantics can be obtained by straightforward transliteration into a functional program. The resulting program is essentially an enhanced definitional interpreter which is passed a set of monitor specifications (representing the monitoring activities) and yields a value together with monitoring information. Thus, it has functionality:

$$\overline{\mathcal{P}} : \mathbf{Mon}^{*} \times \mathbf{Prog} \times \mathbf{Input}^{*} \rightarrow (\mathbf{Ans} \times \mathbf{MS})$$

This monitoring process can clearly be optimized using partial evaluation [BEJ88] by three levels of specialization:

1. Specializing the *program representing* $\overline{\mathcal{P}}$ with respect to a fixed set of *monitor specifications* would automatically yield a *concrete monitor*; i.e. an interpreter instrumented with monitoring actions.

2. Specializing the *monitor itself* (from the previous step) with respect to a *source program* would produce an *instrumented program* [SS89]; i.e. a program including extra code to perform the monitoring actions.

3. Specializing the instrumented program (from the previous step) with respect to some partial input would produce a specialized program.

These levels of optimizations are illustrated in Figure 10.

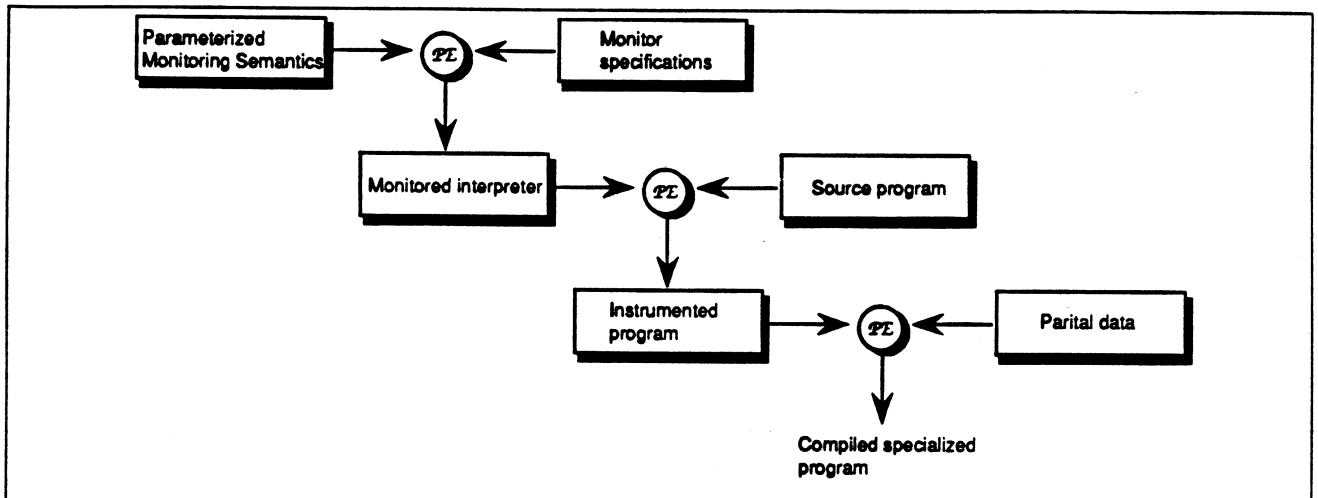

Figure 10: Opportunities for Partial Evaluation in *monitoring semantics*

Note that these instrumentations are achieved uniformly using partial evaluation. This should be contrasted with the usual *ad hoc* techniques of instrumentation.

As a preliminary evaluation of our approach we have implemented a standard interpreter for $\mathcal{L}_\lambda$ as well as the parameterized monitored interpreter whose semantics is given in Section 4. We have optimized that interpreter according to the levels of specialization suggested above and compared the results to the conventional approaches for monitoring described in Section 1.

At the first level of specialization, we have instrumented the parameterized interpreter with the tracer specification described in Section 8. The specialization yields an instrumented interpreter similar to the first conventional approach; i.e. *monitoring by instrumenting an interpreter.* For a given program, our tracer is about 11% slower than the standard interpreter—this slowdown is caused by the extra tracing activity. However (in contrast with the conventional approach) using partial evaluation we can go beyond interpretation by specializing the monitored interpreter with respect to a source program (the second level of specialization). This yields a program instrumented with tracing operations which is 85% faster than the monitored interpreter and 83% faster than the standard interpreter.

Interestingly, the second specialization yields instrumented code similar to the code expected from the second conventional approach: *monitoring by program instrumentation.* However as mentioned above, the instrumentation is achieved uniformly using partial evaluation rather than by using *ad hoc* code instrumentation.

These preliminary results suggest that, like a standard semantics, a monitor semantics possesses both static and dynamic computations. The static computations depend on the program text, including monitoring annotations; and the dynamic computations manipulate run-time values. Thus, the degree of optimization, obtained by partial evaluation, will depend on how much static computation is defined by the monitor. For example, the tracer of Section 8 has static environment lookup but dynamic stream operations.

Figure 11 compares the performances of the $\mathcal{L}_\lambda$ standard interpreter and its tracer for a simple test program. Notice that the monitor performance approaches the standard interpreter performance (the x axis) as the monitoring activity decreases (i.e. as the number of requested trace printouts decreases). This suggests that essentially the *only* overhead in using the monitored interpreter is the extra computation performed by the monitoring activity. Consequently, the monitored interpreter performance graph in Figure 11 corresponds to the linear complexity of the tracer dynamic behavior.

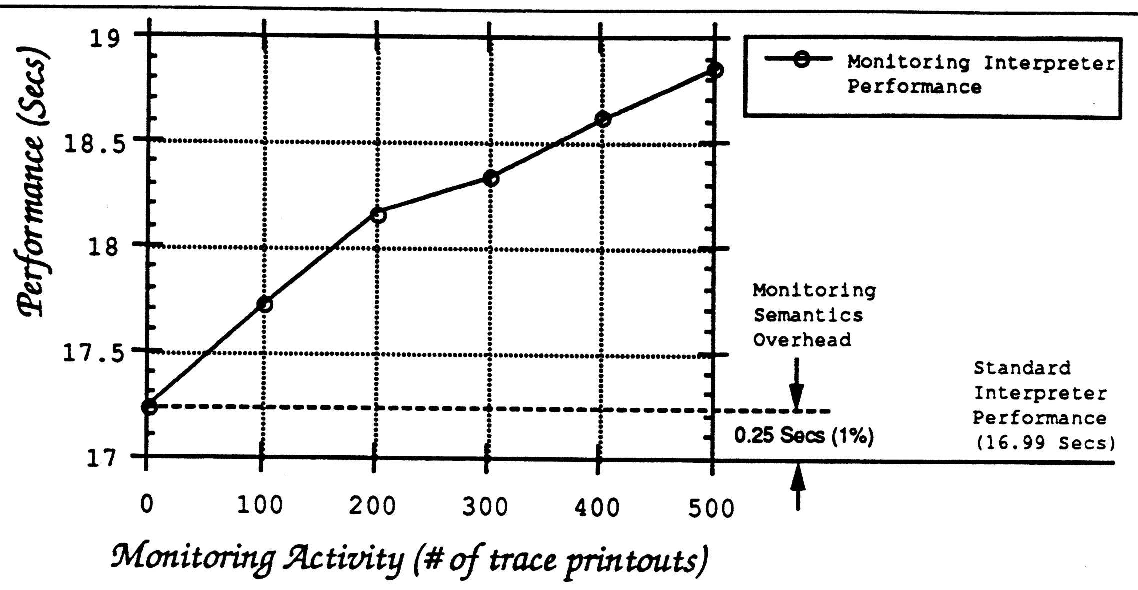

Figure 11: Performance graph

These experiments were achieved using Schism[Con89, Con90], a partial evaluator for pure Scheme.

9.2 Haskell implementation

We have implemented the framework presented in this paper. The implementation provides a generic programming environment which allows automatic integration of monitoring tools with several language modules (lazy, strict and imperative languages). The system is written in Haskell [HW$^+$90]. Currently the environment has a toolbox of predefined monitor specifications which includes: an interactive debugger *à la* dbx, a stepper, a tracer, a profiler, a collecting monitor and other specific monitors for each language. Furthermore, the user can compose several monitors from the existing toolbox with one of the existing language modules. For example: to profile and use a symbolic debugger for a strict interpretation of prog, the user simply types:

```
evaluate (profile & debug & strict) prog
```

where & is a composition operator defined for monitors. Similarly, the user can compose his own defined tools with the existing languages and monitors; Haskell's static type system ensures that new specifications of monitors are well-defined (this can be easily verified by inspecting the type of the monitor). Moreover, since a monitor can only modify its own state, we maintain a high-degree of safety such that new monitors can not change the behaviors of other elements in the environment. For details of this implementation, the reader is referred to [Kis91].

10 Conclusion

We have introduced a semantic framework for run-time monitors. This framework is general in that a monitoring semantics can be automatically obtained from any denotational continuation semantics. Furthermore, a monitoring semantics can capture any sequential, deterministic monitoring activity. Finally, monitoring semantics is compositional and preserves the standard semantics.

On the practical side, just as denotational semantics of programming languages can be viewed as interpreters, monitoring semantics can be seen as executable specifications of monitors. As such, not only does monitoring semantics provide a formal framework to reason about monitoring activities, but it also enables practical implementation of monitors. Using partial evaluation, instrumented interpreters as well as instrumented programs can be derived from a monitored semantics on a uniform basis.

Finally, as was mentioned in Section 9.2, the framework presented in this paper has actually been implemented in Haskell [HW+90]. The implementation provides a programming environment that allows automatic integration of several language modules with an extendable toolbox of monitors [Kis91].

Acknowledgments We would like to thank Olivier Danvy, Siau-Cheng Khoo, Dan Rabin, Raman S. Sundaresh and the entire Yale Haskell group for their helpful comments.

References

[BEJ88] D. Bjørner, A. P. Ershov, and N. D. Jones. *Partial Evaluation and Mixed Computation.* North-Holland, 1988.

[Con89] C. Consel. *The Schism Manual.* Yale University, New Haven, USA, 1989.

[Con90] C. Consel. Binding time analysis for higher order untyped functional languages. *ACM Conference on Lisp and Functional Programming*, pages 264–272, 1990.

[CP89] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. *OOPSLA 1989, SIGPLAN Notices 24:10*, 89.

[DFH88] R. Kent Dybvig, D. P. Friedman, and C. T. Haynes. Expansion-passing style: A general macro mechanism. In *Lisp and Symbolic Computation, 1*, pages 53–75. Kluwer Academic Publishers, Netherlands, 1988.

[DMS84] N. M. Delisle, D. E. Menicosy, and M. D. Schwarts. Viewing a programming environment as a single tool. In *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 1984. ACM SIGPLAN Notices, Vol 19, No. 5, May 1984.

[Gor79] M. J. C. Gordon. *The Denotational Description of Programming Languages.* Springer, New York, 1979.

[HO85] C. V. Hall and J. T. O'Donnell. Debugging in a side effect free programming environment. In *Proc. 1985 SIGPLAN Symposium on Programming Languages and Programming environments*, June 1985.

[HW+90] P. Hudak, P. Wadler, et al. Report on the programming language haskell - a non-strict, purely functional language. Tech. Report YALEU/DCS/RR-777, Yale University, April 1990.

[HY88] P. Hudak and J. Young. A collecting interpretation of expressions (without powerdomains). In *Proceedings of the 1988 ACM Symposium of Principles of Programming Languages.* ACM, 1988.

[Kis91] A. Kishon. *Monitoring Semantics: Theory and Practice of Semantics-directed Execution Monitoring.* PhD thesis, Yale University, 1991. (Forthcoming).

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings 1989 IEEE Symposium on Logic in Computer Science.* IEEE, 1989.

[OH88] J. T. O'Donnell and C. V. Hall. Debugging in applicative languages. In *Lisp and Symbolic Computation, 1.* Kluwer Academic Publishers, Netherlands, 1988.

[PN81] B. Plattner and J. Nievergelt. Monitoring program execution: A survey. *IEEE Computer*, pages 76–93, November 1981.

[Red88] U. Reddy. Objects as closures: Abstract semantics of object oriented languages. *ACM Conference on Lisp and Functional Programming*, 1988.

[Rey72] J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Procs. ACM National Conference*, pages 717–740, 1972.

[Sch86] David A. Schmidt. *Denotational Semantics.* Wm. C. Brown Publishers, Dubuque, Iowa, 1986.

[Sha82] E. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1982.

[SS89] S. Safra and E. Shapiro. Meta interpreters for real. In *Concurrent Prolog, collected papers*, volume 2. MIT Press, 1989.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, Massachusetts, 1977.

[TA90] A. P. Tolmach and A. W. Appel. Debugging Standard ML without reverse engineering. In *Proc. 1990 ACM Conference on Lisp and functional programming*, June 1990.

[Wad90] Philip L. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, 1990.