Interprocedural Control Flow Analysis

(Extended Version)

Flemming Nielson and Hanne Riis Nielson

Department of Computer Science, University of Aarhus,
Ny Munkegade, DK-8000 Aarhus C, Denmark.

Electronic mail: {fn,hrn}@daimi.au.dk
Web address: http://www.daimi.au.dk/~{fn,hrn}

Abstract. Control Flow Analysis is a widely used approach for analysing functional and object oriented programs and recently it has also successfully been used to analyse more challenging notions of computation involving concurrency. However, once the applications become more demanding also the analysis needs to be more precise in its ability to deal with mutable state (or side-effects) and to perform polyvariant (or context-sensitive) analysis. Several insights in Data Flow Analysis and Abstract Interpretation show how to do so for imperative programs but the techniques have not had much impact on Control Flow Analysis because of the "less abstract" way in which the techniques are normally expressed. In this paper we show how to incorporate a number of key insights from Data Flow Analysis (involving such advanced interprocedural techniques as call strings and assumption sets) using Abstract Interpretation to induce the analyses from a general collecting semantics.

Keywords. Control Flow Analysis; Interprocedural Data Flow Analysis; Abstract Interpretation.

Note. A shortened version is to appear in the Proceedings of ESOP'99 to be published in Springer Lecture Notes in Computer Science.

1 Introduction

Consider a functional language with assignments in the manner of Standard ML where both functions and references are first-class entities. This language is sufficiently complex that precise Control Flow Analysis information is needed and at the same time the language is sufficiently close to advanced imperative languages that established techniques from Data Flow Analysis and Abstract Interpretation should be applicable.

Control Flow Analysis. The primary aim of Control Flow Analysis is to determine the set of functions that can be called at each application (e.g. `x e` where `x` is a formal parameter to some function) and has been studied quite extensively ([28, 13, 20] to cite just a few). In terms of paths through the program, one tries to *avoid* working with a complete flow graph where all call sites are linked to all function entries and where all function exits are linked to all return sites. Often this is accomplished by means of contours [29] (à la call strings [27] or tokens [14]) so as to improve the precision of the information obtained. One way to specify the analysis is to show how to generate a set of constraints [10, 11, 22, 23] whose least solution is then computed using graph-based ideas. However, the majority of papers on Control Flow Analysis (e.g. [28, 29, 13, 20]) do not consider side-effects — an exception to this is [12] that considers a functional language with shared variable communication between concurrent processes.

Data Flow Analysis. The *intra*procedural fragment of Data Flow Analysis ignores procedure calls and usually formulates a number of data flow equations whose least solution is desired (or sometimes the greatest when a dual ordering is used) [9, 15, 16]. It follows from Tarski's theorem [30] that the equations could equally well be presented as constraints: the least solution is the same (also see [6]).

The *inter*procedural fragment of Data Flow Analysis takes procedure calls into account and aims at treating calls and returns more precisely than mere goto's: if a call site gives rise to analysing a procedure with a certain piece of information, then the resulting piece of information holding at the procedure exit should ideally only be propagated back to the return site corresponding to the actual call site. This is illustrated in Figure 1. In other words, the *intra*procedural view is that all paths through a program are valid (and this set of paths is a regular language), whereas the *inter*procedural view is that only those paths will be valid where procedure entries and exits match in the manner of parentheses (and this set of paths is a proper context free language). Looking at the literature, the majority of papers on Data Flow Analysis (e.g. [27, 17]) do not consider first-class procedures and therefore have no need for a component akin to Control Flow Analysis — an exception to this is [24] that studies an object-oriented language

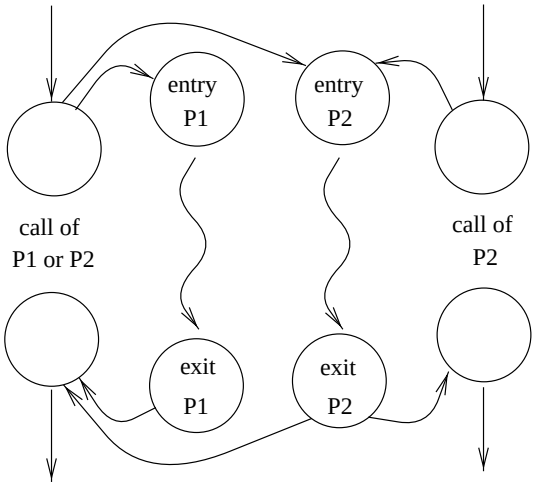

Figure 1: Function call.

with virtual method calls. In the sequel we shall provide a rough classification of the approaches used in the literature (although hybrids exists, e.g. [32]).

- One approach deals with the interprocedural analysis by obtaining transfer functions for entire call statements (obtained by constructing and then solving an equation system that relates transfer functions in a manner dependent upon the program to be analysed) after which the standard techniques for intraprocedural analysis suffice (for example for solving equation systems by iterative techniques or using interval analysis) [27, 17] (and to some extent [4]).

Alternatively one may dispense with formulating equations (or contraints) as the function level and extend the space of properties to include explicit context information.

- A widely used approach modifies the space of properties to include information about the pending procedure calls so as to allow the correct propagation of information at procedure exits even when taking a mainly intraprocedural approach; this is often formulated by means of call strings [27, 31].
- A somewhat orthogonal approach modifies the space of properties to include information that is dependent on the information that was valid at the last procedure entry [24, 18, 25]; an example is the use of so-called assumption sets that give information about the actual parameters.

In this paper we shall focus on the last two groups of techniques.

Abstract Interpretation. In Abstract Interpretation [5], the systematic development of program analyses is likely to span a spectrum from *abstract* specifications (like [20] in the case of Control Flow Analysis), over *syntax-directed* specifications[1] (as in the present paper), to actual *implementations* in the form of constraints being generated and subsequently solved (as in [10, 11, 22, 23, 8]). The main advantage of this approach is that semantic issues can be ignored in later stages once they have been dealt with in earlier stages, and another advantage is that in all stages the set of solutions will be known always to have a least solution that will continue to be a solution also to the specifications of earlier stages. The first stage, often called the collecting semantics, is intended to cover a superset of the semantic considerations that are deemed of potential relevance for the analysis at hand. The purpose of each subsequent stage is to incorporate additional implementation oriented detail so as to obtain an analysis that satisfies the given demands on efficiency with respect to the use of time and space.

Aims. This paper presents an approach to program analysis that allows the simultaneous formulation of techniques for Control and Data Flow Analysis while taking the overall path recommended by Abstract Interpretation. Many of the ingredients presented have been studied before in other contexts; what is unique about the present paper is the way in which the interplay between these techniques is expressed. Indeed, we take care to express the development in a form that is close to formulations of Control Flow Analysis to be found in the literature so as to make the development more accessible.

For this to succeed it is essential to keep the specification compact, as irrelevant details otherwise obscures the main insights, and to this end we present the Control Flow Analysis in the form of a *succinct flow logic*; we refer to [21] for how to implement such specifications. Throughout the development we maintain a clear separation between *environment*-like data and *store*-like data so that the analysis more clearly corresponds to the (formal or informal) semantics of the language analysed. As in [12] we add components for tracking the side-effects occurring in the program and for explicitly propagating environments (thereby extending [20]); for the side-effects this gives rise to a *flow-sensitive* analysis and for the environments we might coin the term *scope-sensitive*. (As we shall see, the techniques for scope-sensitivity are also useful for analysing languages without side-effects in that they allow to incorporate the results of tests into the analysis information passed to the branches of the test.)

Another important ingredient is the judicious choice of a space of mementoes (for expressing context information) that is general enough that both call string

[1] For *syntax-directed* specifications the coinductive method identified in [20] *coincides* with the inductive method.

based approaches (e.g. [27, 29]) and dependent data approaches (in the manner of assumption-sets [24, 18]) can be obtained by merely approximating the space of mementoes; this gives rise to a *context-sensitive* analysis. The most general choice of mementoes considered here was inspired by [7]. The mementoes themselves are approximated using a surjective function and this approach facilitates describing the approximations between the various solution spaces using Galois connections as studied in the framework of Abstract Interpretation [4, 5]. Compared with previous approaches based on Abstract Interpretation (e.g. [4, 1, 7]) we retain the flavour of the "high-level" syntax and semantics of the programming language, because we do not resort[2] to a collecting semantics operating on "low-level" traces of activation records.

Overview. In Section 2 we present the syntax and big-step operational semantics of a functional language with side-effects. In Section 3 we specify the abstract domains of our analysis using a general and rather expressive set of mementoes and in Section 4 we give the details of the analysis. In Section 5 we then show how the classical developments mentioned above can be obtained as Abstract Interpretations of the general choice of mementoes. Finally, Section 6 contains the concluding remarks.

Appendix A establishes the correctness of the analysis, Appendix B presents a generalisation of the analysis to deal with reference counts (so as to be able to kill as well as generate new information), and Appendix C contains the proofs of the main results.

2 Syntax and Semantics

We shall study a functional language with side-effects in the style of Standard ML [19]:

$$
\begin{array}{rcll}
x, f & \in & \mathsf{Var} = \cdots & \text{(unspecified)} \\
e & ::= & c \mid x \mid \mathtt{fn}_{\pi}\ x\ \mathtt{=>}\ e \mid \mathtt{fun}_{\pi}\ f\ x\ \mathtt{=>}\ e \mid (e_1\ e_2)^{l} \mid e_1\ \mathtt{;}\ e_2 \mid \mathtt{ref}_{\varpi}\ e \mid\ !e \\
 & \mid & e_1\ \mathtt{:=}\ e_2 \mid \mathtt{let}\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2 \mid \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 & \\
c & ::= & \mathtt{true} \mid \mathtt{false} \mid \mathtt{()} \mid \cdots &
\end{array}
$$

Here $\mathtt{fn}_{\pi}\ x\ \mathtt{=>}\ e$ is a function that takes one argument and $\mathtt{fun}_{\pi}\ f\ x\ \mathtt{=>}\ e$ is a recursive function (named f) that also takes one argument. We have labelled all syntactic occurrences of function applications with a label $l \in \mathsf{Lab}$, all defining

[2]There are considerable differences in the field as to what constitutes a "high" level of abstraction and what constitutes a "low" level of abstraction; hence our use of "low-level" should not be understood in a negative sense.

$$
\begin{array}{rcll}
m & \in & \mathsf{Mem} & = \{\diamond\} \cup (\mathsf{Lab} \times \mathsf{Mem}) \times \widehat{\mathsf{Val}} \times \widehat{\mathsf{Store}} \times (\mathsf{Pnt_F} \times \mathsf{Mem}) \\
d & \in & \mathsf{Data} & = \cdots \qquad \text{(unspecified)} \\
(\pi, m_d) & \in & \mathsf{Closure} & = \mathsf{Pnt_F} \times \mathsf{Mem} \\
(\varpi, m_d) & \in & \mathsf{Cell} & = \mathsf{Pnt_R} \times \mathsf{Mem} \\
v & \in & \mathsf{Val_A} & = \mathsf{Data} \cup \mathsf{Closure} \cup \mathsf{Cell} \\
W & \in & \widehat{\mathsf{Val}} & = \mathcal{P}(\mathsf{Mem} \times \mathsf{Val_A}) \\
R & \in & \widehat{\mathsf{Env}} & = \mathsf{Var} \to \widehat{\mathsf{Val}} \\
S & \in & \widehat{\mathsf{Store}} & = \mathsf{Cell} \to \widehat{\mathsf{Val}}
\end{array}
$$

Table 1: Abstract domains.

occurrences of functions with a label $\pi \in \mathsf{Pnt_F}$ and all defining occurrences of references with a label $\varpi \in \mathsf{Pnt_R}$. These labels will play a crucial role in the specification of the analysis; it is not required that the labels are distinct but the analysis will be more precise if they are.

The semantics is specified as a big-step operational semantics with environments $\rho \in \mathsf{Env}$ and stores $\sigma \in \mathsf{Store}$. The language has static scope rules and we give it a traditional call-by-value semantics. The semantic domains are:

$$
\begin{array}{rcl}
\iota & \in & \mathsf{Loc} = \cdots \qquad \text{(unspecified)} \\
\omega & \in & \mathsf{Val} \\
\omega & ::= & c \mid \mathsf{close}\ (\mathtt{fn}_\pi\ x\ \mathtt{=>}\ e)\ \mathsf{in}\ \rho \mid \mathsf{close}\ (\mathtt{fun}_\pi\ f\ x\ \mathtt{=>}\ e)\ \mathsf{in}\ \rho \mid \iota \\
e & ::= & \cdots \mid \iota \\
\rho & \in & \mathsf{Env} = \mathsf{Var} \to_{\mathrm{fin}} \mathsf{Val} \\
\sigma & \in & \mathsf{Store} = \mathsf{Loc} \to_{\mathrm{fin}} \mathsf{Val}
\end{array}
$$

The set Loc of locations for references is left unspecified. The judgements of the semantics have the form

$$\rho \vdash \langle e, \sigma_1 \rangle \to \langle \omega, \sigma_2 \rangle$$

and are specified in Table 7 of the Appendix; the clauses themselves should be fairly straightforward. (We should also note that the choice of big-step operational semantics is not crucial for the subsequent development.)

3 Abstract Domains

The abstract domains of the analysis are shown in Table 1 and are explained below.

Mementoes. The analysis will gain its precision from the so-called *mementoes* (or contours or tokens). A memento $m \in \mathsf{Mem}$ represents an approximation of the *context* of a program point: it will either be $\diamond$ representing the initial context where no function calls have taken place or it will have the form

$$((l, m_h), W, S, (\pi, m_d))$$

representing the context in which a function is called. The idea is that

- (l, m_h) describes the application point; l is the label of the function application and m_h is the memento at the application point,
- W is an approximation of the actual parameter at the application point,
- S is an approximation of the store at the application point, and
- (π, m_d) describes the function that is called; π is the label of the function definition and m_d is the memento at the definition point of the function.

Note that this is well-defined (in the manner of context-free grammars): composite mementoes are constructed from simpler mementoes and in the end from the initial memento $\diamond$. This definition of mementoes is akin to the contexts considered in [7]; in Section 5 we shall show how the set can be simplified into something more tractable.

Example 1. As a running example we shall consider the program "program" defined by:

$$((\texttt{fn}_x\ \texttt{x => ((x x)}^1\ (\texttt{fn}_y\ \texttt{y => x}))^2)\ (\texttt{fn}_z\ \texttt{z => z}))^3$$

The applications are performed in the order 3, 1 and 2. The mementoes of interest are going to be

$$\begin{aligned} m_3 &= ((3, \diamond), W_3, [\,], (x, \diamond)) \\ m_1 &= ((1, m_3), W_1, [\,], (z, \diamond)) \\ m_2 &= ((2, m_1), W_2, [\,], (z, \diamond)) \end{aligned}$$

where W_1, W_2 and W_3 will be specified in Example 2 and $[\,]$ indicates that the store is empty. □

Abstract values. We operate on three kinds of abstract values: data, function closures and reference cells. Function closures and reference cells are represented as pairs consisting of the label (π and ϖ, respectively) of the definition point and the memento m_d at the definition point; this will allow us to distinguish between the various instances of the closures and reference cells. The abstract values will

$$
\begin{array}{rcll}
\mathcal{R}_F^{\mathrm{D}}, \mathcal{R}_F^{\mathrm{C}} & \in & \widehat{\mathsf{RCache_F}} & = \mathsf{Pnt_F} \to \widehat{\mathsf{Env}} \\
\mathcal{M}_F & \in & \widehat{\mathsf{MCache_F}} & = \mathsf{Pnt_F} \to \mathcal{P}(\mathsf{Mem}) \\
\mathcal{W}_F & \in & \widehat{\mathsf{WCache_F}} & = (\bullet\mathsf{Pnt_F} \cup \mathsf{Pnt_F}\bullet) \to \widehat{\mathsf{Val}} \\
\mathcal{S}_F & \in & \widehat{\mathsf{SCache_F}} & = (\bullet\mathsf{Pnt_F} \cup \mathsf{Pnt_F}\bullet) \to \widehat{\mathsf{Store}}
\end{array}
$$

Table 2: Caches.

always come together with the memento (i.e. the context) in which they live so the analysis will operate over sets of pairs of mementoes and abstract values. The set $\widehat{\mathsf{Val}}$ obtained in this way is equipped with the subset ordering (denoted $\subseteq$). The sets $\widehat{\mathsf{Env}}$ and $\widehat{\mathsf{Store}}$ of abstract environments and abstract stores, respectively, are now obtained in an obvious way and ordered by the pointwise extension of the subset ordering (denoted $\sqsubseteq$).

Example 2. Continuing Example 1 we have

$$W_3 = \{(\diamond, (z, \diamond))\} \quad W_1 = \{(m_3, (z, \diamond))\} \quad W_2 = \{(m_1, (y, m_3))\}$$

since the function z is defined at the top-level ($\diamond$) and y is defined inside the application 3. □

Caches. The analysis will operate on five caches associating information with functions; their functionality is shown in Table 2 and their use will be explained below.

The caches $\mathcal{R}_F^{\mathrm{D}}$, $\mathcal{R}_F^{\mathrm{C}}$ and $\mathcal{M}_F$ associate information with the labels π of function *definitions*:

- The *environment caches* $\mathcal{R}_F^{\mathrm{D}}$ and $\mathcal{R}_F^{\mathrm{C}}$: for each program point π, $\mathcal{R}_F^{\mathrm{D}}(\pi)$ records the abstract environment at the definition point (i.e. as applies to the free variables) and $\mathcal{R}_F^{\mathrm{C}}(\pi)$ records the same information but modified to each of the (presumably different) contexts in which the function body might be executed. As an example, the same value v of a variable x used in a function labelled π may turn up in $\mathcal{R}_F^{\mathrm{D}}(\pi)(x)$ as (m_d, v) and in $\mathcal{R}_F^{\mathrm{C}}(\pi)(x)$ as (m_c, v) where $m_d = \diamond$ in case of the function being declared at top-level and $m_c = ((l, \diamond), W, S, (\pi, \diamond))$ in case of the call being performed at the top-level application labelled l.
- The *memento cache* $\mathcal{M}_F$: for each program point π, $\mathcal{M}_F(\pi)$ records the set of contexts in which the function body might be executed; so $\mathcal{M}_F(\pi) = \emptyset$ means that the function is never executed.

The caches $\mathcal{W}_F$ and $\mathcal{S}_F$ associate information with function *calls*. For a function with label $\pi \in \mathsf{Pnt_F}$ we shall use $\bullet\pi$ ($\in \bullet\mathsf{Pnt_F}$) to denote the point just before

entering the body of the function, and we shall use $\pi\bullet$ ($\in \mathsf{Pnt_F}\bullet$) to denote the point just after leaving the body of the function. The idea now is as follows:

- The *value cache* $\mathcal{W}_F$: for each entry point $\bullet\pi$, $\mathcal{W}_F(\bullet\pi)$ records the abstract value describing the possible actual parameters, and for each exit point $\pi\bullet$, $\mathcal{W}_F(\pi\bullet)$ records the abstract value describing the possible results of the call.
- The *store cache* $\mathcal{S}_F$: for each entry point $\bullet\pi$, $\mathcal{S}_F(\bullet\pi)$ records the abstract store describing the possible stores at function entry, and for each exit point $\pi\bullet$, $\mathcal{S}_F(\pi\bullet)$ records the abstract store describing the possible stores at function exit.

Example 3. For the example program we may take the following caches:

π	x	y	z
$\mathcal{W}_F(\bullet\pi)$	$\{(m_3,(z,\diamond))\}$	$\emptyset$	$\{(m_1,(z,\diamond)),(m_2,(y,m_3))\}$
$\mathcal{W}_F(\pi\bullet)$	$\{(m_3,(y,m_3))\}$	$\emptyset$	$\{(m_1,(z,\diamond)),(m_2,(y,m_3))\}$
$\mathcal{S}_F(\bullet\pi)$	$[\,]$	$[\,]$	$[\,]$
$\mathcal{S}_F(\pi\bullet)$	$[\,]$	$[\,]$	$[\,]$
$\mathcal{R}^{\mathrm{D}}_F(\pi)$	$[\,]$	$[\mathtt{x}\mapsto\{(m_3,(z,\diamond))\}]$	$[\,]$
$\mathcal{R}^{\mathrm{C}}_F(\pi)$	$[\,]$	$[\,]$	$[\,]$
$\mathcal{M}_F(\pi)$	$\{m_3\}$	$\emptyset$	$\{m_1,m_2\}$

4 Syntax-directed Analysis

The specification developed in this section is a recipe for *checking* that a proposed solution is indeed acceptable. This is useful when changing libraries of support code or when installing software in new environments: one merely needs to check that the new libraries or environments satisfy the solution used to optimize the program. It can also be used as the basis for generating a set of constraints [21] whose least solution can be obtained using standard techniques (e.g. [2]).

Given a program e and the five caches $(\mathcal{R}^{\mathrm{D}}_F, \mathcal{R}^{\mathrm{C}}_F, \mathcal{M}_F, \mathcal{W}_F, \mathcal{S}_F)$ the purpose of the analysis is to check whether or not the caches are acceptable solutions to the Data and Control Flow Analysis. The first step is to find (or guess) the following auxiliary information:

- an abstract environment $R \in \widehat{\mathsf{Env}}$ describing the free variables in e (and typically it is $\bot$ if there are no free variables in the program),
- a set of mementoes $M \in \mathcal{P}(\mathsf{Mem})$ describing the possible contexts in which e can be evaluated (and typically it is $\{\diamond\}$),

- an initial abstract store $S_1 \in \widehat{\mathsf{Store}}$ describing the mutable store before evaluation of e begins (and typically it is $\top$ if the store is not initialised before use),
- a final abstract store $S_2 \in \widehat{\mathsf{Store}}$ describing the mutable store after evaluation of e completes (and possibly it is $\top$), and
- an abstract value $W \in \widehat{\mathsf{Val}}$ describing the value that e can evaluate to (and it also possibly is $\top$).

The second step is to check whether or not the formula

$$R, M \ \rhd \ e : S_1 \to S_2 \ \& \ W$$

is satisfied with respect to the caches supplied. This means that when e is executed in an environment described by R, in a context described by M, and upon a state described by S_1 the following happens: if e terminates successfully then the resulting state is described by S_2 and the resulting value by W.

We shall first specify the analysis for the functional fragment of the language (Table 3 in Section 4.1) and then for the other constructs (Table 4 in Section 4.2). As in [20] any free variable on the right-hand side of the clauses should be regarded as existentially quantified; in principle this means that their values need to be guessed, but in practice the best (or least) guess mostly follows from the subformulae.

Example 4. Given the caches of Example 3, we shall check the formula:

$$[\,], \{\diamond\} \rhd \mathsf{program} : [\,] \to [\,] \ \& \ \{(\diamond, (y, m_3))\}$$

So the initial environment is empty, the initial context is $\diamond$, the program does not manipulate the store, and the final value is described by $\{(\diamond, (y, m_3))\}$. □

4.1 The functional fragment

For all five constructs in the functional fragment of the language the handling of the store is straightforward since it is threaded in the same way as in the semantics.

For constants and variables it is fairly straightforward to determine the abstract value for the construct; in the case of variables we obtain it from the environment and in the other case we construct it from the set M of mementoes of interest.

For function definitions no changes need take place in the store so the abstract store is simply threaded as in the previous cases. The abstract value representing

$$R, M \ \rhd \ c : S_1 \to S_2 \ \& \ W$$
$$\quad \textsf{iff } S_1 \sqsubseteq S_2 \wedge \{(m, d_c) \mid m \in M\} \subseteq W$$

$$R, M \ \rhd \ x : S_1 \to S_2 \ \& \ W$$
$$\quad \textsf{iff } S_1 \sqsubseteq S_2 \wedge R(x) \subseteq W$$

$$R, M \ \rhd \ \texttt{fn}_\pi \ x \ \texttt{=>} \ e : S_1 \to S_2 \ \& \ W$$
$$\quad \textsf{iff } S_1 \sqsubseteq S_2 \wedge \{(m, (\pi, m)) \mid m \in M\} \subseteq W \wedge R \sqsubseteq \mathcal{R}^{\text{D}}_F(\pi) \wedge$$
$$\qquad \mathcal{R}^{\text{C}}_F(\pi)[x \mapsto \mathcal{W}_F(\bullet\pi)], \mathcal{M}_F(\pi) \ \rhd \ e : \mathcal{S}_F(\bullet\pi) \to \mathcal{S}_F(\pi\bullet) \ \& \ \mathcal{W}_F(\pi\bullet)$$

$$R, M \ \rhd \ \texttt{fun}_\pi \ f \ x \ \texttt{=>} \ e : S_1 \to S_2 \ \& \ W$$
$$\quad \textsf{iff } S_1 \sqsubseteq S_2 \wedge \{(m, (\pi, m)) \mid m \in M\} \subseteq W \wedge$$
$$\qquad R[f \mapsto \{(m, (\pi, m)) \mid m \in M\}] \sqsubseteq \mathcal{R}^{\text{D}}_F(\pi) \wedge$$
$$\qquad \mathcal{R}^{\text{C}}_F(\pi)[x \mapsto \mathcal{W}_F(\bullet\pi)], \mathcal{M}_F(\pi) \ \rhd \ e : \mathcal{S}_F(\bullet\pi) \to \mathcal{S}_F(\pi\bullet) \ \& \ \mathcal{W}_F(\pi\bullet)$$

$$R, M \ \rhd \ (e_1 \ e_2)^l : S_1 \to S_4 \ \& \ W$$
$$\quad \textsf{iff } R, M \ \rhd \ e_1 : S_1 \to S_2 \ \& \ W_1 \ \wedge R, M \ \rhd \ e_2 : S_2 \to S_3 \ \& \ W_2 \ \wedge$$
$$\qquad \forall \pi \in \{\pi \mid (m, (\pi, m_d)) \in W_1\} :$$
$$\qquad\quad \textsf{let } X = \overline{\textsf{new}}_\pi((l, M), W_2, S_3, W_1)$$
$$\qquad\qquad X_{\text{DC}} = \{(m_d, m_c) \mid (m_d, m_h, m_c) \in X\}$$
$$\qquad\qquad X_{\text{C}} \ = \{m_c \mid (m_d, m_h, m_c) \in X\}$$
$$\qquad\qquad X_{\text{HC}} = \{(m_h, m_c) \mid (m_d, m_h, m_c) \in X\}$$
$$\qquad\qquad X_{\text{CH}} = \{(m_c, m_h) \mid (m_d, m_h, m_c) \in X\}$$
$$\qquad\quad \textsf{in } \ \mathcal{R}^{\text{D}}_F(\pi)\lceil X_{\text{DC}}\rceil \sqsubseteq \mathcal{R}^{\text{C}}_F(\pi) \wedge X_{\text{C}} \subseteq \mathcal{M}_F(\pi) \wedge$$
$$\qquad\qquad W_2\lceil X_{\text{HC}}\rceil \subseteq \mathcal{W}_F(\bullet\pi) \wedge S_3\lceil X_{\text{HC}}\rceil \sqsubseteq \mathcal{S}_F(\bullet\pi) \wedge$$
$$\qquad\qquad \mathcal{W}_F(\pi\bullet)\lceil X_{\text{CH}}\rceil \subseteq W \wedge \mathcal{S}_F(\pi\bullet)\lceil X_{\text{CH}}\rceil \sqsubseteq S_4$$

$$\overline{\textsf{new}}_\pi((l, M), W, S, W') =$$
$$\quad \{(m_d, m_h, m_c) \mid (m_h, (\pi, m_d)) \in W', \ m_h \in M,$$
$$\qquad\qquad m_c = \textsf{new}((l, m_h), W, S, (\pi, m_d))\}$$

Table 3: Analysis of the functional fragment.

the function definition contains a nested pair (a triple) for each memento m in the set M of mementoes according to which the function definition can be reached: in a nested pair $(m_1, (\pi, m_2))$ the memento m_1 represents the current context and the pair (π, m_2) represents the value produced (and demanding that $m_1 = m_2$ corresponds to performing a precise relational analysis rather than a less precise independent attribute analysis). Finally, the body of the function is analysed in the relevant abstract environment, memento set, initial abstract state, final abstract state and final abstract value; this information is obtained from the

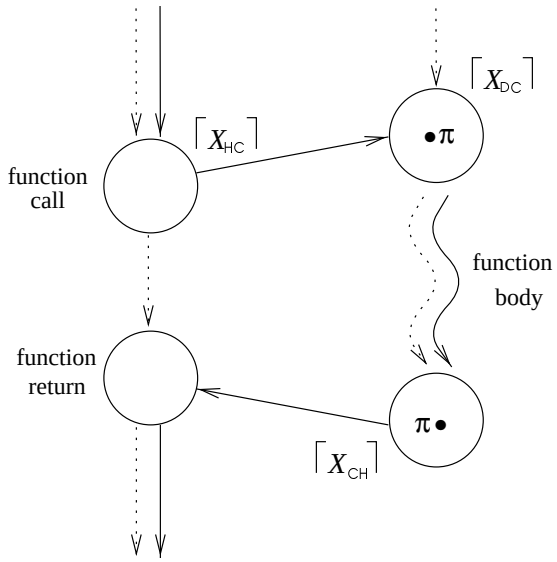

Figure 2: Analysis of function call.

caches that are in turn updated at the corresponding call points. More precisely, the idea is to record the abstract environment at the definition point in the cache $\mathcal{R}_F^{\mathrm{D}}$ and then to analyse the body of the function in the context of the call which is specified by the caches $\mathcal{R}_F^{\mathrm{C}}$, $\mathcal{M}_F$, $\mathcal{W}_F$ and $\mathcal{S}_F$ as explained in Section 3. The clause for recursive functions is similar.

Example 5. To check the formula of Example 4 we need among other things to check:

$$[\,], \{\diamond\} \rhd \mathtt{fn}_z\ \mathtt{z}\ \mathtt{=>}\ \mathtt{z} : [\,] \to [\,]\ \&\ \{(\diamond, (z, \diamond))\}$$

This follows from the clause for function definition because $[\,] \sqsubseteq [\,]$ and the clause for variables gives:

$$[\mathtt{z} \mapsto \{(m_1, (z, \diamond)), (m_2, (y, m_3))\}], \{m_1, m_2\} \rhd \mathtt{z} : [\,] \to [\,]\ \&\ \{(m_1, (z, \diamond)), (m_2, (y, m_3))\}$$

Note that although the function z is *called twice*, it is only *analysed once*. □

In the clause for the function application $(e_1\ e_2)^l$ we first analyse the operator and the operand while threading the store. Then we use W_1 to determine which functions can be called and for each such function π we proceed in the following way.

First we determine the mementoes to be used for analysing the body of the function π. More precisely we calculate a set X of triples (m_d, m_h, m_c) consisting of a definition memento m_d describing the point where the function π was defined,

a current memento m_h describing the call point, and a memento m_c describing the entry point to the procedure body. (For the call $(\mathtt{x}\ \mathtt{x})^1$ in Example 1 we would have $X = \{(\diamond, m_3, m_1)\}$ and $\pi = z$.) For this we use the operation $\overline{\mathsf{new}}_\pi$ whose definition (see Table 3) uses the function

$$\mathsf{new} : (\mathsf{Lab} \times \mathsf{Mem}) \times \widehat{\mathsf{Val}} \times \widehat{\mathsf{Store}} \times (\mathsf{Pnt}_\mathsf{F} \times \mathsf{Mem}) \to \mathsf{Mem}$$

for converting its argument to a memento. With Mem defined as in Table 1 this will be the identity function but for simpler choices of Mem it will discard some of the information supplied by its argument.

The sets X_{DC}, X_{C}, X_{HC}, and X_{CH} are "projections" of X. The body of the function π will be analysed in the set of mementoes obtained as X_{C} and therefore X_{C} is recorded in the cache $\mathcal{M}_F$ for use in the clause defining the function. Because the function body is analysed in this set of mementoes we need to modify the mementoes components of all the relevant abstract values. For this we use the operation

$$W\lceil Y \rceil = \{(m_2, v) \mid (m_1, v) \in W, (m_1, m_2) \in Y\}$$

defined on $W \subseteq \widehat{\mathsf{Val}}$ and $Y \subseteq \mathsf{Mem} \times \mathsf{Mem}$. This operation is lifted to abstract environments and abstract stores in a pointwise manner.

Coming back to the clause for application in Table 3, the abstract environment $\mathcal{R}^{\text{D}}_F(\pi)$ is relative to the mementoes of the definition point for the function and thus has to be modified so as to be relative to the mementoes of the called function body and the set X_{DC} facilitates performing this transformation. (For the call $(\mathtt{x}\ \mathtt{x})^1$ in Example 1 we would have that $X_{\text{DC}} = \{(\diamond, m_1)\}$.) In this way we ensure that we have static scoping of the free variables of the function. The actual parameter W_2 is relative to the mementoes of the application point and has to be modified so as to be relative to the mementoes of the called function body and the set X_{HC} facilitates performing this transformation; a similar modification is needed for the abstract store at the entry point. We also need to link the results of the analysis of the function body back to the application point and here the relevant transformation is facilitated by the set X_{CH}.

The clause for application is illustrated in Figure 2. On the left-hand side we have the application point with explicit nodes for the call and the return. The dotted lines represent the abstract environment and the relevant set of mementoes whereas the solid lines represent the values (actual parameter and result) and the store. The transfer function $\lceil X_{\text{DC}} \rceil$ is used to modify the static environment of the definition point, the transfer function $\lceil X_{\text{HC}} \rceil$ is used to go from the application point to the function body and the transfer function $\lceil X_{\text{CH}} \rceil$ is used to go back from the function body to the application point. Note that the figure clearly indicates the different paths taken by environment-like information and store-like information – something that is not always clear from similar figures appearing in the literature (see Section 5.2).

Example 6. Checking the formula of Example 4 also involves checking:

$$[\mathtt{x} \mapsto \{(m_3, (z, \diamond))\}], \{m_3\} \rhd (\mathtt{x}\ \mathtt{x})^1 : [\,] \to [\,] \ \&\ \{(m_3, (z, \diamond))\}$$

For this, the clause for application demands that we check

$$[\mathtt{x} \mapsto \{(m_3, (z, \diamond))\}], \{m_3\} \rhd \mathtt{x} : [\,] \to [\,] \ \&\ \{(m_3, (z, \diamond))\}$$

which follows directly from the clause for variables.

Only the function z can be called so we have to check the many conditions only for this function. We shall concentrate on checking that $\{(m_3, (z, \diamond))\}\lceil X_{\text{HC}}\rceil \subseteq \mathcal{W}_F(\bullet z)$ and $\mathcal{W}_F(z\bullet)\lceil X_{\text{CH}}\rceil \subseteq \{(m_3, (z, \diamond))\}$. Since $X = \{(\diamond, m_3, m_1)\}$ we have $X_{\text{HC}} = \{(m_3, m_1)\}$ and the effect of the transformation will be to remove all pairs that do not have m_3 as the first component and to replace the first components of the remaining pairs with m_1; using Example 3 it is immediate to verify that the condition actually holds. Similarly, $X_{\text{CH}} = \{(m_1, m_3)\}$ so in this case the transformation will remove pairs that do not have m_1 as the first component (i.e. pairs that do not correspond to the current call point) and replace the first components of the remaining pairs with m_3; again it is immediate to verify that the condition holds. □

4.2 Other constructs

The clauses for the other constructs of the language are shown in Table 4. The clauses reflect that the abstract environment and the set of mementoes are passed to the subexpressions in a syntax-directed way and that the store is threaded through the constructs. The analysis is fairly simple-minded in that it does not try to predict when a reference (ϖ, m_d) in the analysis only represents one location in the semantics and hence the analysis does not contain any kill-components (but see Appendix B).

For the `let`-construct we perform the expected threading of the abstract environment and the abstract store. For the conditional we first analyse the condition. Based on the outcome we then modify the environment and the store to reflect the (abstract) value of the test. For the environment we use the transfer functions $\varphi^{[e,W]}_{\mathtt{true}}(R)$ and $\varphi^{[e,W]}_{\mathtt{false}}(R)$ whereas for the store we use the transfer functions $\phi^{[e,W]}_{\mathtt{true}}(S_2)$ and $\phi^{[e,W]}_{\mathtt{false}}(S_2)$. The result of both branches are possible for the whole construct.

As an example of the use of these transfer functions consider the expression

$$\mathtt{if}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$$

where it will be natural to set

$$\varphi^{[x,W]}_{\mathtt{true}}(R) = R[x \mapsto W \cap \{(m, d_{\mathtt{true}}) \mid m \in \mathsf{Mem}\}]$$
$$\varphi^{[x,W]}_{\mathtt{false}}(R) = R[x \mapsto W \cap \{(m, d_{\mathtt{false}}) \mid m \in \mathsf{Mem}\}]$$

$R, M \ \triangleright \ e_1 \ ; \ e_2 : S_1 \to S_3 \ \& \ W_2$
  iff $R, M \ \triangleright \ e_1 : S_1 \to S_2 \ \& \ W_1 \wedge R, M \ \triangleright \ e_2 : S_2 \to S_3 \ \& \ W_2$

$R, M \ \triangleright \ \texttt{ref}_{\varpi} \ e : S_1 \to S_3 \ \& \ W'$
  iff $R, M \ \triangleright \ e : S_1 \to S_2 \ \& \ W \wedge \{(m, (\varpi, m)) \mid m \in M\} \subseteq W' \wedge$
    $S_2 \sqsubseteq S_3 \wedge \forall m \in M : W \subseteq S_3(\varpi, m)$

$R, M \ \triangleright \ !e : S_1 \to S_2 \ \& \ W'$
  iff $R, M \ \triangleright \ e : S_1 \to S_2 \ \& \ W \wedge \forall (m, (\varpi, m_d)) \in W : S_2(\varpi, m_d) \subseteq W'$

$R, M \ \triangleright \ e_1 \ \texttt{:=} \ e_2 : S_1 \to S_4 \ \& \ W$
  iff $R, M \ \triangleright \ e_1 : S_1 \to S_2 \ \& \ W_1 \wedge R, M \ \triangleright \ e_2 : S_2 \to S_3 \ \& \ W_2 \wedge$
    $\{(m, d_{()}) \mid m \in M\} \subseteq W \wedge S_3 \sqsubseteq S_4 \wedge$
    $\forall (m, (\varpi, m_d)) \in W_1 : W_2 \subseteq S_4(\varpi, m_d)$

$R, M \ \triangleright \ \texttt{let} \ x \ \texttt{=} \ e_1 \ \texttt{in} \ e_2 : S_1 \to S_3 \ \& \ W_2$
  iff $R, M \ \triangleright \ e_1 : S_1 \to S_2 \ \& \ W_1 \wedge R[x \mapsto W_1], M \ \triangleright \ e_2 : S_2 \to S_3 \ \& \ W_2$

$R, M \ \triangleright \ \texttt{if} \ e \ \texttt{then} \ e_1 \ \texttt{else} \ e_2 : S_1 \to S_5 \ \& \ W'$
  iff $R, M \ \triangleright \ e : S_1 \to S_2 \ \& \ W \wedge$
    let $R_1 = \varphi^{[e,W]}_{\texttt{true}}(R); \quad R_2 = \varphi^{[e,W]}_{\texttt{false}}(R); \quad S_3 = \phi^{[e,W]}_{\texttt{true}}(S_2); \quad S_4 = \phi^{[e,W]}_{\texttt{false}}(S_2)$
    in $R_1, M \ \triangleright \ e_1 : S_3 \to S_5 \ \& \ W' \ \wedge R_2, M \ \triangleright \ e_2 : S_4 \to S_5 \ \& \ W'$

Table 4: Analysis of the other constructs.

Thus even though $R(x)$ might be $\{(m_h, d_{\texttt{true}}), (m_h, d_{\texttt{false}})\}$ it will be possible to analyse each of the branches with precise information about x as was already hinted at in the Introduction. Little can be said in general about how to define such functions: it depends on the exact details of the expression upon which the test is made.

To avoid cluttering the presentation of the analysis we have only inserted transfer functions where they are essential for our main story. There are certainly a number of additional places where non-identity transfers could be useful for certain applications. One example of this is the clause for `let` where we could use transfer functions to allow recording the fact that the bound variable is first "allocated" and later "de-allocated".

To obtain a more concise statement of the theorems below we shall mostly assume that the transfer functions $\phi^{\cdots}_{\cdots}$ and $\varphi^{\cdots}_{\cdots}$ of Table 4 are in fact the identities. In Appendix A we establish the semantic correctness of the analysis presented here.

$$
\begin{array}{rcll}
m_k & \in & \mathsf{Mem}_k & = & \mathsf{Lab}^{\leq k} \\
d & \in & \mathsf{Data} & = & \cdots \qquad \text{(unspecified)} \\
(\pi, m_{kd}) & \in & \mathsf{Closure}_k & = & \mathsf{Pnt}_\mathsf{F} \times \mathsf{Mem}_k \\
(\varpi, m_{kd}) & \in & \mathsf{Cell}_k & = & \mathsf{Pnt}_\mathsf{R} \times \mathsf{Mem}_k \\
v_k & \in & \mathsf{Val}_{\mathsf{A}k} & = & \mathsf{Data} \cup \mathsf{Closure}_k \cup \mathsf{Cell}_k \\
W_k & \in & \widehat{\mathsf{Val}}_k & = & \mathcal{P}(\mathsf{Mem}_k \times \mathsf{Val}_{\mathsf{A}k}) \\
R_k & \in & \widehat{\mathsf{Env}}_k & = & \mathsf{Var} \to \widehat{\mathsf{Val}}_k \\
S_k & \in & \widehat{\mathsf{Store}}_k & = & \mathsf{Cell}_k \to \widehat{\mathsf{Val}}_k
\end{array}
$$

Table 5: Abstract domains for k-CFA.

5 Classical Approximations

The definition of mementoes used in Table 1 is much too unwieldy to be of practical interest. In this section we shall study some alternative choices.

5.1 k-CFA

The idea behind k-CFA [13, 28] is to restrict the mementoes to keep track of the last k call sites only. This leads to the abstract domains of Table 5 that are intended to replace Table 1. Naturally, the analysis of Tables 2, 3, and 4 must be modified to use the new abstract domains; also the function $\overline{\mathsf{new}}_\pi$ must be modified to make use of the function

$$\mathsf{new}_k : (\mathsf{Lab} \times \mathsf{Mem}_k) \times \widehat{\mathsf{Val}}_k \times \widehat{\mathsf{Store}}_k \times (\mathsf{Pnt}_\mathsf{F} \times \mathsf{Mem}_k) \to \mathsf{Mem}_k$$

defined by

$$\mathsf{new}_k((l, m_{kh}), W_k, S_k, (\pi, m_{kd})) = \mathsf{take}_k(l \hat{} m_{kh})$$

where "$\hat{}$" denotes prefixing and take_k returns the first k elements of its argument. This completes the definition of the analysis.

Theoretical properties. One of the strong points of our approach is that we can use the framework of Abstract Interpretation to describe *how* the more tractable choices of mementoes arise from the general definition. Previously such explanations have been performed at "lower levels" of syntactic and semantic abstraction, and in our view this has hampered the wide-spread use of these techniques. However, the reader unaccustomed to Abstract Interpretation might prefer to skip this explanation on a first reading.

To express the relationship between the two analyses we define a *surjective* mapping $\mu_k : \mathsf{Mem} \to \mathsf{Mem}_k$ showing how the precise mementoes of Table 1 are

truncated into the approximative mementoes of Table 5. It is defined by

$$\begin{aligned}\mu_0(m) &= \varepsilon\\ \mu_{k+1}(\diamond) &= \varepsilon\\ \mu_{k+1}((l,m),W,S,(\pi,m_d)) &= l\hat{}\mu_k(m)\end{aligned}$$

where ε denotes the empty sequence. It gives rise to the functions $\alpha_k^M : \mathcal{P}(\mathsf{Mem}) \to \mathcal{P}(\mathsf{Mem}_k)$ and $\gamma_k^M : \mathcal{P}(\mathsf{Mem}_k) \to \mathcal{P}(\mathsf{Mem})$ defined by

$$\alpha_k^M(M) = \{\mu_k(m) \mid m \in M\} \qquad \gamma_k^M(M_k) = \{m \mid \mu_k(m) \in M_k\}$$

Since α_k^M is surjective and defined in a pointwise manner there exists precisely one function such that

$$\mathcal{P}(\mathsf{Mem}) \underset{\alpha_k^M}{\overset{\gamma_k^M}{\leftarrow\!\!\rightarrow}} \mathcal{P}(\mathsf{Mem}_k)$$

is a *Galois insertion* as studied in Abstract Interpretation [3, 5]: this means that α_k^M and γ_k^M are both monotone and that $\gamma_k^M(\alpha_k^M(M)) \supseteq M$ and $\alpha_k^M(\gamma_k^M(M_k)) = M_k$ for all $M \subseteq \mathsf{Mem}$ and $M_k \subseteq \mathsf{Mem}_k$. One may check that γ_k^M is as displayed above.

To obtain a Galois insertion

$$\widehat{\mathsf{Val}} \underset{\alpha_k^V}{\overset{\gamma_k^V}{\leftarrow\!\!\rightarrow}} \widehat{\mathsf{Val}}_k$$

we proceed in two steps. First define a surjective mapping $\eta_k : \mathsf{Mem} \times \mathsf{Val}_\mathsf{A} \to \mathsf{Mem}_k \times \mathsf{Val}_{\mathsf{A}k}$ by

$$\begin{aligned}\eta_k(m_h,d) &= (\mu_k(m_h),d)\\ \eta_k(m_h,(\pi,m_d)) &= (\mu_k(m_h),(\pi,\mu_k(m_d)))\\ \eta_k(m_h,(\varpi,m_d)) &= (\mu_k(m_h),(\varpi,\mu_k(m_d)))\end{aligned}$$

Next define α_k^V and γ_k^V by

$$\alpha_k^V(W) = \{\eta_k(m,v) \mid (m,v) \in W\} \qquad \gamma_k^V(W_k) = \{(m,v) \mid \eta_k(m,v) \in W_k\}$$

It is then straightforward to obtain a Galois insertion

$$\widehat{\mathsf{Env}} \underset{\alpha_k^E}{\overset{\gamma_k^E}{\leftarrow\!\!\rightarrow}} \widehat{\mathsf{Env}}_k$$

by setting

$$\alpha_k^E(R)(x) = \alpha_k^V(R(x)) \qquad \gamma_k^E(R_k)(x) = \gamma_k^V(R_k(x))$$

To obtain a Galois insertion

$$\widehat{\mathsf{Store}} \underset{\alpha_k^S}{\overset{\gamma_k^S}{\leftrightarrows}} \widehat{\mathsf{Store}}_k$$

we define

$$\begin{aligned}\alpha_k^S(S)(\varpi, m_{kd}) &= \alpha_k^V(\bigcup\{S(\varpi, m_d) \mid \mu_k(m_d) = m_{kd}\}) \\ \gamma_k^S(S_k)(\varpi, m_d) &= \gamma_k^V(S_k(\varpi, \mu_k(m_d)))\end{aligned}$$

We now have the machinery needed to state the relationship between the present k-CFA analysis (denoted $\rhd_k$) and the more general analysis of Section 4 (denoted $\rhd$):

Theorem 5.1 *If* $(\mathcal{R}^{\mathrm{D}}_{kF}, \mathcal{R}^{\mathrm{C}}_{kF}, \mathcal{M}_{kF}, \mathcal{W}_{kF}, \mathcal{S}_{kF})$ *satisfies*

$$R_k, M_k \ \rhd_k \ e : S_{k1} \to S_{k2} \ \& \ W_k$$

then $(\gamma_k^E \circ \mathcal{R}^{\mathrm{D}}_{kF}, \gamma_k^E \circ \mathcal{R}^{\mathrm{C}}_{kF}, \gamma_k^M \circ \mathcal{M}_{kF}, \gamma_k^V \circ \mathcal{W}_{kF}, \gamma_k^S \circ \mathcal{S}_{kF})$ *satisfies*

$$\gamma_k^E(R_k), \gamma_k^M(M_k) \ \rhd \ e : \gamma_k^S(S_{k1}) \to \gamma_k^S(S_{k2}) \ \& \ \gamma_k^V(W_k)$$

(provided all transfer functions $\phi^{\cdots}_{\cdots}$ *and* $\varphi^{\cdots}_{\cdots}$ *are the identities).*

The proof is by structural induction on e. In the case of application we establish three auxiliary facts: the relationship between new_k and new, the relationship between $\overline{\mathsf{new}}_{k\pi}$ and $\overline{\mathsf{new}}_{\pi}$, and the relationship between $\gamma(\cdot)\lceil\cdot\rceil$ and $\gamma(\cdot\lceil\cdot\rceil)$. (The statement and proof of the theorem for general transfer functions $\phi^{\cdots}_{\cdots}$ and $\varphi^{\cdots}_{\cdots}$ is somewhat more complex.)

In Appendix A we establish the semantic correctness of the analysis of Section 4; it is then a consequence of the above theorem that semantic correctness holds for k-CFA as well.

Call strings of length k

The clause for application involves a number of transfers using the set X relating definition mementoes, current mementoes and mementoes of the called function body. In the case of a k-CFA like approach it may be useful to simplify these transfers.

The transfer using X_{HC} can be implemented in a particularly simple way by taking

$$X_{\mathrm{HC}} = \{(m_h, \mathsf{take}_k(l\hat{}m_h)) \mid m_h \in M\}$$

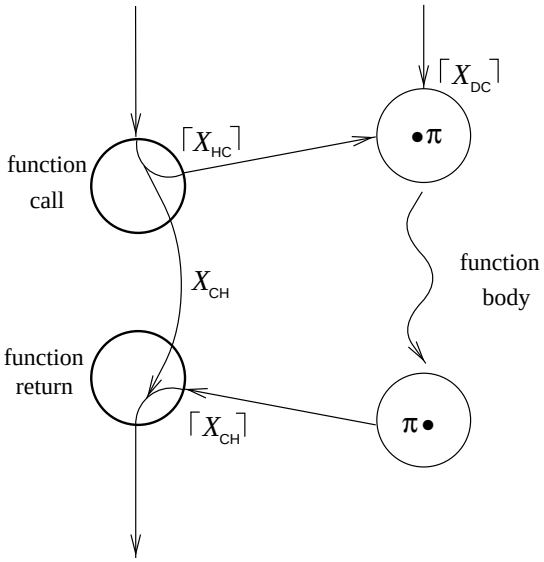

Figure 3: Degenerate analysis of function call.

where l is the label of the application point. This set may be slightly too large because it is no longer allowed to depend on the actual function called (the π) and because there may be $m_h \in M_k$ for which no $(m_h, (\pi, m_d))$ is ever an element of W_1. However, this is just a minor imprecision aimed at facilitating a more efficient implementation. In a similar way, one may take

$$X_{\text{C}} = \{\mathsf{take}_k(l\hat{}m_h) \mid m_h \in M\}$$

where again this set may be slightly too large.

The transfers using X_{CH} can also be somewhat simplified by taking

$$\begin{aligned} X_{\text{CH}} &= \{(\mathsf{take}_k(l\hat{}m_h), m_h) \mid m_h \in M\} \\ &= \{(m_c, \mathsf{drop}_1(m_c)) \mid \mathsf{drop}_1(m_c) \in M\} \\ &\quad \cup \{(m_c, \mathsf{drop}_1(m_c)\hat{}l') \mid \mathsf{drop}_1(m_c)\hat{}l' \in M\} \end{aligned}$$

where drop_1 drops the first element of its argument (yielding ε if the argument does not have at least two elements). Again this set may be slightly too large.

The transfer using X_{DC} can be rewritten as

$$X_{\text{DC}} = \{(m_d, \mathsf{take}_k(l\hat{}m_h)) \mid m_h \in M, (m_h, (\pi, m_d)) \in W_1\}$$

where again l is the label of the application point and π indicates the actual function being called. For functions being defined at top-level there is not likely

$$
\begin{array}{rcll}
m_{\mathrm{P}} & \in & \mathsf{Mem}_{\mathrm{P}} & = \{\varepsilon\} \cup \mathcal{P}(\mathsf{Data} \cup \mathsf{Pnt}_{\mathsf{F}} \cup \mathsf{Pnt}_{\mathsf{R}}) \\
d & \in & \mathsf{Data} & = \cdots \quad \text{(unspecified)} \\
(\pi, m_{\mathrm{P}d}) & \in & \mathsf{Closure}_{\mathrm{P}} & = \mathsf{Pnt}_{\mathsf{F}} \times \mathsf{Mem}_{\mathrm{P}} \\
(\varpi, m_{\mathrm{P}d}) & \in & \mathsf{Cell}_{\mathrm{P}} & = \mathsf{Pnt}_{\mathsf{R}} \times \mathsf{Mem}_{\mathrm{P}} \\
v_{\mathrm{P}} & \in & \mathsf{Val}_{\mathrm{AP}} & = \mathsf{Data} \cup \mathsf{Closure}_{\mathrm{P}} \cup \mathsf{Cell}_{\mathrm{P}} \\
W_{\mathrm{P}} & \in & \widehat{\mathsf{Val}}_{\mathrm{P}} & = \mathcal{P}(\mathsf{Mem}_{\mathrm{P}} \times \mathsf{Val}_{\mathrm{AP}}) \\
R_{\mathrm{P}} & \in & \widehat{\mathsf{Env}}_{\mathrm{P}} & = \mathsf{Var} \to \widehat{\mathsf{Val}}_{\mathrm{P}} \\
S_{\mathrm{P}} & \in & \widehat{\mathsf{Store}}_{\mathrm{P}} & = \mathsf{Cell}_{\mathrm{P}} \to \widehat{\mathsf{Val}}_{\mathrm{P}}
\end{array}
$$

Table 6: Abstract domains for assumption sets.

to be too much information that need to be transformed using X_{DC}; however, simplifying X_{DC} to be independent of π is likely to be grossly imprecise.

Performing these modifications to the clause for application there is no longer any need for an explicit call of $\overline{\mathsf{new}}_{\pi}$. The resulting analysis is similar in spirit to the call string based analysis of [31]; the scenario of [27] is simpler because the language considered there does not allow local data. Since we have changed the definition of the sets X_{DC}, X_{C}, X_{HC} and X_{CH} to something that is no less than before, it follows that an analogue of Theorem 5.1 still applies and therefore the semantic correctness result still carries over.

It is interesting to note that if the distinction between environment and store is not clearly maintained then Figure 2 degenerates to the form of Figure 3; this is closely related to the scenario in [26] (that is somewhat less general).

5.2 Assumption sets

The idea behind this analysis is to restrict the mementoes to keep track of the parameter of the last function call only; such information is often called assumption sets. This leads to the abstract domains of Table 6 that are intended to replace Table 1. Naturally, the analysis of Tables 2, 3, and 4 must be modified to use the new abstract domains; also the function $\overline{\mathsf{new}}_{\pi}$ must be modified to make use of the function

$$\mathsf{new}_{\mathrm{P}} : (\mathsf{Lab} \times \mathsf{Mem}_{\mathrm{P}}) \times \widehat{\mathsf{Val}}_{\mathrm{P}} \times \widehat{\mathsf{Store}}_{\mathrm{P}} \times (\mathsf{Pnt}_{\mathsf{F}} \times \mathsf{Mem}_{\mathrm{P}}) \to \mathsf{Mem}_{\mathrm{P}}$$

defined by

$$\mathsf{new}_{\mathrm{P}}((l, m_{\mathrm{P}h}), W_{\mathrm{P}}, S_{\mathrm{P}}, (\pi, m_{\mathrm{P}d})) = \{\mathsf{keep}_{\mathrm{P}}(v_{\mathrm{P}}) \mid (m_{\mathrm{P}}, v_{\mathrm{P}}) \in W_{\mathrm{P}}\}$$

where $\mathsf{keep}_{\mathrm{P}} : \mathsf{Val}_{\mathrm{AP}} \to (\mathsf{Data} \cup \mathsf{Pnt}_{\mathsf{F}} \cup \mathsf{Pnt}_{\mathsf{R}})$ is defined as follows

$$\mathsf{keep}_{\mathrm{P}}(d) \quad = \quad d$$

$$\begin{aligned}\mathsf{keep}_{\mathrm{P}}(\pi, m_{\mathrm{P}d}) &= \pi \\ \mathsf{keep}_{\mathrm{P}}(\varpi, m_{\mathrm{P}d}) &= \varpi\end{aligned}$$

This completes the definition of the analysis.

Theoretical properties. We can now mimic the development performed in Section 5.1; once again the reader unaccustomed to Abstract Interpretation might prefer to skip this explanation on a first reading.

The crucial point is the definition of a *surjective* mapping $\mu_{\mathrm{P}} : \mathsf{Mem} \to \mathsf{Mem}_{\mathrm{P}}$ showing how the precise mementoes of Table 1 are mapped into the approximative mementoes of Table 6. It is defined by

$$\begin{aligned}\mu_{\mathrm{P}}(\diamond) &= \varepsilon \\ \mu_{\mathrm{P}}((l, m), W, S, (\pi, m_d)) &= \{\mathsf{keep}'_{\mathrm{P}}(v) \mid (m', v') \in W\}\end{aligned}$$

where $\mathsf{keep}'_{\mathrm{P}} : \mathsf{Val}_{\mathsf{A}} \to (\mathsf{Data} \cup \mathsf{Pnt}_{\mathsf{F}} \cup \mathsf{Pnt}_{\mathsf{R}})$ is the obvious modification of $\mathsf{keep}_{\mathrm{P}}$ to work on $\mathsf{Val}_{\mathsf{A}}$ rather than $\mathsf{Val}_{\mathsf{AP}}$. Based on μ_{P} we can now define Galois *insertions*

- $(\alpha^M_{\mathrm{P}}, \gamma^M_{\mathrm{P}})$ between $\mathcal{P}(\mathsf{Mem})$ and $\mathcal{P}(\mathsf{Mem}_{\mathrm{P}})$
- $(\alpha^V_{\mathrm{P}}, \gamma^V_{\mathrm{P}})$ between $\widehat{\mathsf{Val}}$ and $\widehat{\mathsf{Val}}_{\mathrm{P}}$
- $(\alpha^E_{\mathrm{P}}, \gamma^E_{\mathrm{P}})$ between $\widehat{\mathsf{Env}}$ and $\widehat{\mathsf{Env}}_{\mathrm{P}}$
- $(\alpha^S_{\mathrm{P}}, \gamma^S_{\mathrm{P}})$ between $\widehat{\mathsf{Store}}$ and $\widehat{\mathsf{Store}}_{\mathrm{P}}$

very much as in Section 5.1 and we then obtain the following analogue of Theorem 5.1:

Theorem 5.2 *If* $(\mathcal{R}^{\mathrm{D}}_{\mathrm{P}F}, \mathcal{R}^{\mathrm{C}}_{\mathrm{P}F}, \mathcal{M}_{\mathrm{P}F}, \mathcal{W}_{\mathrm{P}F}, \mathcal{S}_{\mathrm{P}F})$ *satisfies*

$$R_{\mathrm{P}}, M_{\mathrm{P}} \ \rhd_{\mathrm{P}} \ e : S_{\mathrm{P}1} \to S_{\mathrm{P}2} \ \& \ W_{\mathrm{P}}$$

then $(\gamma^E_{\mathrm{P}} \circ \mathcal{R}^{\mathrm{D}}_{\mathrm{P}F}, \gamma^E_{\mathrm{P}} \circ \mathcal{R}^{\mathrm{C}}_{\mathrm{P}F}, \gamma^M_{\mathrm{P}} \circ \mathcal{M}_{\mathrm{P}F}, \gamma^V_{\mathrm{P}} \circ \mathcal{W}_{\mathrm{P}F}, \gamma^S_{\mathrm{P}} \circ \mathcal{S}_{\mathrm{P}F})$ *satisfies*

$$\gamma^E_{\mathrm{P}}(R_{\mathrm{P}}), \gamma^M_{\mathrm{P}}(M_{\mathrm{P}}) \ \rhd \ e : \gamma^S_{\mathrm{P}}(S_{\mathrm{P}1}) \to \gamma^S_{\mathrm{P}}(S_{\mathrm{P}2}) \ \& \ \gamma^V_{\mathrm{P}}(W_{\mathrm{P}})$$

(provided all transfer functions $\phi^{\cdots}_{\cdots}$ *and* $\varphi^{\cdots}_{\cdots}$ *are the identities).*

(The statement and proof of the theorem for general transfer functions $\phi^{\cdots}_{\cdots}$ and $\varphi^{\cdots}_{\cdots}$ is somewhat more complex.) As in Section 5.1 it is a consequence of the above theorem that semantic correctness holds for the assumption set analysis as well.

6 Conclusion

We have shown how to express interprocedural and context-sensitive Data Flow Analysis in a syntax-directed framework that is reminiscent of Control Flow Analysis; thereby we have not only extended the ability of Data Flow Analysis to deal with higher-order functions but we also have extended the ability of Control Flow Analysis to deal with mutable data structures.

We used Abstract Interpretation to pass from the general mementoes of Section 3 to the more tractable mementoes of Section 5. While there are related studies in the literature, we believe that our development is more faithful to the "level of abstraction" given by the "high-level" syntax and semantics of modern programming languages. In particular, our development was not forced to start from a collecting semantics operating upon "low-level" stacks of activation records: *all* our analyses[3] are based on the specification of Tables 3 and 4. We strongly believe that this aspect of our development is likely to be crucial for Abstract Interpretation to become more widely used.

Acknowledgement. This work has been supported in part by the DART project funded by the Danish Science Research Council. The ideas presented here have benefited from conversations with Patrick Cousot, Laurie Hendren, Suresh Jagannathan, Alan Mycroft, Tom Reps, Barbara Ryder, Helmut Seidl, and Mitch Wand.

References

[1] F. Bourdoncle. Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity. In *Proc. PLILP '90*, volume 456 of *Lecture Notes in Computer Science*, pages 307–323. Springer, 1990.

[2] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. Formal Methods in Programming and Their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141. Springer, 1993.

[3] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. POPL '77*, pages 238–252. ACM Press, 1977.

[4] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E. J. Neuhold, editor, *Formal Description of Programming Concepts*. North Holland, 1978.

[3] It remains an open problem how to integrate Abstract Interpretation with the abstract approach of [20].

[5] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. POPL '79*, pages 269–282, 1979.

[6] P. Cousot and R. Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game theoretic form (invited paper). In *Proc. CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 293–308. Springer, 1995.

[7] A. Deutsch. On Determining Lifetime and Aliasing of Dynamically Allocated Data in Higher Order Functional Specifications. In *Proc. POPL '90*, pages 157–169. ACM Press, 1990.

[8] K. L. S. Gasser, F. Nielson, and H. R. Nielson. Systematic realisation of control flow analyses for CML. In *Proc. ICFP '97*, pages 38–51. ACM Press, 1997.

[9] M. S. Hecht. *Flow Analysis of Computer Programs.* North Holland, 1977.

[10] N. Heintze. Set-based analysis of ML programs. In *Proc. LFP '94*, pages 306–317, 1994.

[11] N. Heintze and J. Jaffar. An engine for logic program analysis. In *Proc. LICS '92*, pages 318–328, 1992.

[12] S. Jagannathan and S. Weeks. Analyzing Stores and References in a Parallel Symbolic Language. In *Proc. LFP '94*, pages 294–305, 1994.

[13] S. Jagannathan and S. Weeks. A unified treatment of flow analysis in higher-order languages. In *Proc. POPL '95.* ACM Press, 1995.

[14] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proc. POPL '82*, pages 66–74. ACM Press, 1982.

[15] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[16] G. Kildall. A Unified Approach to Global Program Optimization. In *Proc. POPL '73*, pages 194–206. ACM Press, 1973.

[17] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Proc. CC '92*, volume 641 of *Lecture Notes in Computer Science*, pages 125–140. Springer, 1992.

[18] W. Landi and B. G. Ryder. Pointer-Induced Aliasing: A Problem Classification. In *Proc. POPL '91*, pages 93–103. ACM Press, 1991.

[19] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML.* MIT Press, 1990.

[20] F. Nielson and H. R. Nielson. Infinitary Control Flow Analysis: a Collecting Semantics for Closure Analysis. In *Proc. POPL '97.* ACM Press, 1997.

[21] H. R. Nielson and F. Nielson. Flow logics for constraint based analysis. In *Proc. CC '98*, volume 1383 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 1998.

[22] J. Palsberg. Global program analysis in constraint form. In *Proc. CAAP '94*, volume 787 of *Lecture Notes in Computer Science*, pages 255–265. Springer, 1994.

[23] J. Palsberg. Closure analysis in constraint form. *ACM TOPLAS*, 17 (1):47–62, 1995.

[24] H. D. Pande and B. G. Ryder. Data-flow-based virtual function resolution. In *Proc. SAS '96*, volume 1145 of *Lecture Notes in Computer Science*, pages 238–254. Springer, 1996.

[25] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proc. PLDI '95*, pages 13–22. ACM Press, 1995.

[26] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *Proc. TAPSOFT '95*, volume 915 of *Lecture Notes in Computer Science*, pages 651–665, 1995.

[27] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis*. Prentice Hall International, 1981.

[28] O. Shivers. Control flow analysis in Scheme. In *Proc. PLDI '88*, volume 7 (1) of *ACM SIGPLAN Notices*, pages 164–174. ACM Press, 1988.

[29] O. Shivers. The semantics of Scheme control-flow analysis. In *Proc. PEPM '91*, volume 26 (9) of *ACM SIGPLAN Notices*. ACM Press, 1991.

[30] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309, 1955.

[31] J. Vitek, R. N. Horspool, and J. S. Uhl. Compile-Time Analysis of Object-Oriented Programs. In *Proc. CC '92*, volume 641 of *Lecture Notes in Computer Science*, pages 236–250. Springer, 1992.

[32] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proc. PLDI '95*, pages 1–12. ACM Press, 1995.

A Semantic Correctness

To formulate the correctness result we need a way to associate the locations of the semantics to the cells of the analysis. We shall therefore introduce the concept of a *store model*:

$$\Sigma : \mathsf{Loc} \hookrightarrow \mathsf{Pnt_R} \times \mathsf{Mem}$$

The idea is that $\Sigma(\iota) = (\varpi, m_d)$ means that the location ι is created at the program point ϖ in the context described by the memento m_d. We shall impose an ordering on store models:

$$\Sigma_1 \leq \Sigma_2$$

if and only if $dom(\Sigma_1) \subseteq dom(\Sigma_2)$ and $\forall \iota \in dom(\Sigma_1) : \Sigma_1(\iota) = \Sigma_2(\iota)$.

We also need a way of associating a definition memento with the closures of the semantics. So we introduce the concept of a *value model*:

$$\Omega \in \mathcal{P}(\mathsf{Val} \times \mathsf{Mem})$$

That $(\mathsf{close}\ (\mathtt{fn}_\pi\ x\ \mathtt{=>}\ e)\ \mathsf{in}\ \rho, m_d) \in \Omega$ means that $\mathsf{close}\ (\mathtt{fn}_\pi\ x\ \mathtt{=>}\ e)\ \mathsf{in}\ \rho$ is created in the context described by the memento m_d. A similar comment holds for the closures of recursive functions. The relation will never contain pairs where the first component is a constant or a location. Obviously value models can be ordered by subset inclusion.

In order to formulate the correctness result we shall need a couple of predicates of the form "$\in_M^{\Sigma,\Omega}$"; they are defined relative to $(\mathcal{R}_F^{\mathrm{D}}, \mathcal{R}_F^{\mathrm{C}}, \mathcal{M}_F, \mathcal{W}_F, \mathcal{S}_F)$ (just as the inference system is). We shall need one of these predicates for values, one for environments and one for stores. The predicates are defined co-inductively due to the presence of recursive functions.

For values we define

$$\omega \in_m^{\Sigma,\Omega} W$$

to mean that either

- $\omega = c$ and then $(m, d_c) \in W$.
- $\omega = \mathsf{close}\ (\mathtt{fn}_\pi\ x\ \mathtt{=>}\ e)\ \mathsf{in}\ \rho$ and then there exists $(\omega, m_d) \in \Omega$ such that
  - $(m, (\pi, m_d)) \in W$,
  - $\rho \in_{m_d}^{\Sigma,\Omega} \mathcal{R}_F^{\mathrm{D}}(\pi)$, and
  - $\mathcal{R}_F^{\mathrm{C}}(\pi)[x \mapsto \mathcal{W}_F(\bullet\pi)], \mathcal{M}_F(\pi)\ \rhd\ e : \mathcal{S}_F(\bullet\pi) \to \mathcal{S}_F(\pi\bullet)\ \&\ \mathcal{W}_F(\pi\bullet)$
- $\omega = \mathsf{close}\ (\mathtt{fun}_\pi\ f\ x\ \mathtt{=>}\ e)\ \mathsf{in}\ \rho$ and then there exists $(\omega, m_d) \in \Omega$ such that
  - $(m, (\pi, m_d)) \in W$,

- $(\rho[f \mapsto \omega]) \in^{\Sigma,\Omega}_{m_d} \mathcal{R}^{\mathrm{D}}_F(\pi)$, and
- $\mathcal{R}^{\mathrm{C}}_F(\pi)[x \mapsto \mathcal{W}_F(\bullet\pi)], \mathcal{M}_F(\pi) \ \rhd \ e : \mathcal{S}_F(\bullet\pi) \to \mathcal{S}_F(\pi\bullet) \ \& \ \mathcal{W}_F(\pi\bullet)$

- $\omega = \iota$ and then $(m, \Sigma(\iota)) \in W$.

For environments we define

$$\rho \in^{\Sigma,\Omega}_m R \text{ iff } \forall x : \rho(x) \in^{\Sigma,\Omega}_m R(x)$$

and for stores we define

$$\sigma \in^{\Sigma,\Omega}_m S \text{ iff } \forall \iota : \sigma(\iota) \in^{\Sigma,\Omega}_m S(\Sigma(\iota))$$

Theorem A.1 *Assume that*

$$\rho \vdash \langle e, \sigma_1 \rangle \to \langle \omega, \sigma_2 \rangle$$

and that $(\mathcal{R}^{\mathrm{D}}_F, \mathcal{R}^{\mathrm{C}}_F, \mathcal{M}_F, \mathcal{W}_F, \mathcal{S}_F)$ *satisfies the formula*

$$R, M \ \rhd \ e : S_1 \to S_2 \ \& \ W$$

For all $m \in M$ *and all* Σ *and* Ω*: If*

$$\rho \in^{\Sigma,\Omega}_m R \text{ and } \sigma_1 \in^{\Sigma,\Omega}_m S_1$$

then there exists $\Sigma' \geq \Sigma$ *and* $\Omega' \supseteq \Omega$ *such that*

$$\sigma_2 \in^{\Sigma',\Omega'}_m S_2 \text{ and } \omega \in^{\Sigma',\Omega'}_m W$$

(provided all transfer functions $\phi^{\cdots}_{\cdots}$ *and* $\varphi^{\cdots}_{\cdots}$ *are the identities).*

The proof is by induction on the inference in the semantics and using auxiliary facts expressing that for "$\in^{\Sigma,\Omega}_m$" it is possible to enlarge W, R, and S, to enlarge Σ and Ω, and to modify the mementoes using the operation $\cdot\lceil\cdot\rceil$. In the case of recursive function definition we rely on coinduction. (The statement and proof of the theorem for general transfer functions $\phi^{\cdots}_{\cdots}$ and $\varphi^{\cdots}_{\cdots}$ is somewhat more complex.)

B Reference Counts

An obvious extension of the work performed here is to incorporate an abstract notion of reference count for dynamically created cells. In the manner of [31] we could change the definition of $\widehat{\mathsf{Store}}$ (in Table 1) to have

$$\begin{array}{rcl} S \in \widehat{\mathsf{Store}} & = & \mathsf{Cell} \to (\widehat{\mathsf{Val}} \times \widehat{\mathsf{Pop}}) \\ p \in \widehat{\mathsf{Pop}} & = & \{\mathsf{O}, \mathsf{I}, \mathsf{M}\} \end{array}$$

$$\rho \vdash \langle c, \sigma\rangle \to \langle c, \sigma\rangle$$

$$\rho \vdash \langle x, \sigma\rangle \to \langle \omega, \sigma\rangle \text{ if } \omega = \rho(x)$$

$$\rho \vdash \langle \mathtt{fn}_\pi\ x\ \mathtt{=>}\ e, \sigma\rangle \to \langle \mathsf{close}\ (\mathtt{fn}_\pi\ x\ \mathtt{=>}\ e)\ \mathsf{in}\ \rho, \sigma\rangle$$

$$\rho \vdash \langle \mathtt{fun}_\pi\ f\ x\ \mathtt{=>}\ e, \sigma\rangle \to \langle \mathsf{close}\ (\mathtt{fun}_\pi\ f\ x\ \mathtt{=>}\ e)\ \mathsf{in}\ \rho, \sigma\rangle$$

$$\frac{\rho \vdash \langle e_1, \sigma_1\rangle \to \langle \mathsf{close}\ (\mathtt{fn}_\pi\ x\ \mathtt{=>}\ e)\ \mathsf{in}\ \rho', \sigma_2\rangle, \quad \rho \vdash \langle e_2, \sigma_2\rangle \to \langle \omega_2, \sigma_3\rangle, \quad \rho'[x \mapsto \omega_2] \vdash \langle e, \sigma_3\rangle \to \langle \omega, \sigma_4\rangle}{\rho \vdash \langle (e_1\ e_2)^l, \sigma_1\rangle \to \langle \omega, \sigma_4\rangle}$$

$$\frac{\rho \vdash \langle e_1, \sigma_1\rangle \to \langle \mathsf{close}\ (\mathtt{fun}_\pi\ f\ x\ \mathtt{=>}\ e)\ \mathsf{in}\ \rho', \sigma_2\rangle, \quad \rho \vdash \langle e_2, \sigma_2\rangle \to \langle \omega_2, \sigma_3\rangle, \quad \rho'[f \mapsto \mathsf{close}\ (\mathtt{fun}_\pi\ f\ x\ \mathtt{=>}\ e)\ \mathsf{in}\ \rho'][x \mapsto \omega_2] \vdash \langle e, \sigma_3\rangle \to \langle \omega, \sigma_4\rangle}{\rho \vdash \langle (e_1\ e_2)^l, \sigma_1\rangle \to \langle \omega, \sigma_4\rangle}$$

$$\frac{\rho \vdash \langle e_1, \sigma_1\rangle \to \langle \omega_1, \sigma_2\rangle, \quad \rho \vdash \langle e_2, \sigma_2\rangle \to \langle \omega_2, \sigma_3\rangle}{\rho \vdash \langle e_1\ \mathtt{;}\ e_2, \sigma_1\rangle \to \langle \omega_2, \sigma_3\rangle}$$

$$\frac{\rho \vdash \langle e, \sigma_1\rangle \to \langle \omega, \sigma_2\rangle}{\rho \vdash \langle \mathtt{ref}_\varpi\ e, \sigma_1\rangle \to \langle \iota, \sigma_2[\iota \mapsto \omega]\rangle} \text{ where } \iota \text{ is the first unused location}$$

$$\frac{\rho \vdash \langle e, \sigma_1\rangle \to \langle \iota, \sigma_2\rangle}{\rho \vdash \langle\, !\, e, \sigma_1\rangle \to \langle \omega, \sigma_2\rangle} \text{ where } \omega = \sigma_2(\iota)$$

$$\frac{\rho \vdash \langle e_1, \sigma_1\rangle \to \langle \iota, \sigma_2\rangle, \quad \rho \vdash \langle e_2, \sigma_2\rangle \to \langle \omega, \sigma_3\rangle}{\rho \vdash \langle e_1\ \mathtt{:=}\ e_2, \sigma_1\rangle \to \langle \mathtt{()}, \sigma_3[\iota \mapsto \omega]\rangle}$$

$$\frac{\rho \vdash \langle e_1, \sigma_1\rangle \to \langle \omega_1, \sigma_2\rangle, \quad \rho[x \mapsto \omega_1] \vdash \langle e_2, \sigma_2\rangle \to \langle \omega_2, \sigma_3\rangle}{\rho \vdash \langle \mathtt{let}\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2, \sigma_1\rangle \to \langle \omega_2, \sigma_3\rangle}$$

$$\frac{\rho \vdash \langle e, \sigma_1\rangle \to \langle \mathtt{true}, \sigma_2\rangle, \quad \rho \vdash \langle e_1, \sigma_2\rangle \to \langle \omega, \sigma_3\rangle}{\rho \vdash \langle \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2, \sigma_1\rangle \to \langle \omega, \sigma_3\rangle}$$

$$\frac{\rho \vdash \langle e, \sigma_1\rangle \to \langle \mathtt{false}, \sigma_2\rangle, \quad \rho \vdash \langle e_2, \sigma_2\rangle \to \langle \omega, \sigma_3\rangle}{\rho \vdash \langle \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2, \sigma_1\rangle \to \langle \omega, \sigma_3\rangle}$$

Table 7: Operational semantics.

Here the new $\widehat{\mathsf{Pop}}$ component denotes how many concrete locations may simultaneously be described by the abstract reference cell: O means zero, I means at most one, and M means arbitrarily many (including zero and one).

$R, M \ \rhd\ \texttt{ref}_{\varpi}\ e : S_1 \to S_3\ \&\ W'$
  iff $R, M \ \rhd\ e : S_1 \to S_2\ \&\ W \wedge \{(m, (\varpi, m)) \mid m \in M\} \subseteq W' \wedge$
    $\forall m \in M : S_2 \oplus ((\varpi, m), W) \sqsubseteq S_3$

$R, M \ \rhd\ !e : S_1 \to S_2\ \&\ W'$
  iff $R, M \ \rhd\ e : S_1 \to S_2\ \&\ W \wedge \forall (m, (\varpi, m_d)) \in W : S_2(\varpi, m_d) \sqsubseteq (W', \mathsf{M})$

$R, M \ \rhd\ e_1\ \texttt{:=}\ e_2 : S_1 \to S_4\ \&\ W$
  iff $R, M \ \rhd\ e_1 : S_1 \to S_2\ \&\ W_1 \wedge R, M \ \rhd\ e_2 : S_2 \to S_3\ \&\ W_2 \wedge$
    $\{(m, d_{()}) \mid m \in M\} \subseteq W \wedge$
    $\forall (m, (\varpi, m_d)) \in W_1 : (S_3 \ominus (\varpi, m_d)) \oplus ((\varpi, m_d), W_2) \sqsubseteq S_4$

Table 8: Dealing with reference counts.

This makes it possible for the analysis sometimes to overwrite (as opposed to always augment) the value of a cell that is created or assigned. For this we need a new operation for adding a reference:

$$S \oplus ((\varpi, m), W) = S[(\varpi, m) \mapsto (W'', p'')]$$

where

$$\begin{array}{rcl} (W', p') & = & S(\varpi, m) \\ (W'', p'') & = & \begin{cases} (W \cup W', \mathsf{M}) & \text{if } p' \neq \mathsf{O} \\ (W, \mathsf{I}) & \text{if } p' = \mathsf{O} \end{cases} \end{array}$$

We also need a new operation for removing a reference:

$$S \ominus (\varpi, m) = S[(\varpi, m) \mapsto (W'', p'')]$$

where

$$\begin{array}{rcl} (W', p') & = & S(\varpi, m) \\ (W'', p'') & = & \begin{cases} (W', p') & \text{if } p' = \mathsf{M} \\ (\emptyset, \mathsf{O}) & \text{if } p' \neq \mathsf{M} \end{cases} \end{array}$$

The necessary modifications to the analysis are shown in Table 8.

C Proofs of Theorems

Proof of Theorem 5.1. The proof proceeds by structural induction on e. In all cases the definition of Tables 3 and 4 (for $\rhd_k$) is used to "unfold" the assumption. Next the induction hypothesis and simple calculations are used to obtain similar unfolded statements for the conclusion. Finally the definition of Tables 3 and 4 (for $\rhd$) is used to "fold" the statements so as to obtain the conclusion.

The proof exploits that

- μ_k is *surjective*, and
- all (α_k, γ_k) are Galois *insertions*.

In the case of conditional the assumptions about $\phi^{\cdots}_{\cdots}$ and $\varphi^{\cdots}_{\cdots}$ simplify what needs to be proved; for a more general statement one would need explicit assumptions about the relations between the two sets of transfer functions.

In the case of application the following fact is used:

Fact C.1 $\mu_k(\mathsf{new}((l, m_h), W, S, (\pi, m_d))) =$
$\mathsf{new}_k((l, \mu_k(m_h)), \alpha_k^V(W), \alpha_k^S(S), (\pi, \mu_k(m_d)))$.

Writing $\mu_k[X] = \{(\mu_k(x_1), \cdots, \mu_k(x_n)) \mid (x_1, \cdots, x_n) \in X\}$ this allows us to establish:

Fact C.2 *Setting*

$$\begin{array}{rcl} X_k & = & \overline{\mathsf{new}}_{\pi k}((l, M_k), W_k, S_k, W_k') \\ X & = & \overline{\mathsf{new}}_{\pi}((l, \gamma_k^M(M_k)), \gamma_k^V(W_k), \gamma_k^S(S_k), \gamma_k^V(W_k')) \end{array}$$

we have that $\mu_k[X] \subseteq X_k$.

Fact C.3 *If* $\mu_k[X'] \subseteq X_k'$ *then* $\gamma_k^V(W_k)\lceil X' \rceil \subseteq \gamma_k^V(W_k \lceil X_k' \rceil)$.

Proof of Theorem 5.2. We construct the Galois insertions as follows. First the function μ_P is extended in a pointwise manner to obtain a Galois insertion

$$\mathcal{P}(\mathsf{Mem}) \underset{\alpha_{\mathrm{P}}^M}{\overset{\gamma_P^M}{\leftrightarrows}} \mathcal{P}(\mathsf{Mem}_{\mathrm{P}})$$

where

$$\begin{array}{rcl} \alpha_{\mathrm{P}}^M(M) & = & \{\mu_{\mathrm{P}}(m) \mid m \in M\} \\ \gamma_{\mathrm{P}}^M(M_{\mathrm{P}}) & = & \{m \mid \mu_{\mathrm{P}}(m) \in M_{\mathrm{P}}\} \end{array}$$

To obtain a Galois insertion

$$\widehat{\mathsf{Val}} \underset{\alpha_{\mathrm{P}}^V}{\overset{\gamma_{\mathrm{P}}^V}{\leftrightarrows}} \widehat{\mathsf{Val}}_{\mathrm{P}}$$

we once more proceed in two steps. First define a surjective mapping

$$\eta_{\mathrm{P}} : \mathsf{Mem} \times \mathsf{Val}_{\mathsf{A}} \to \mathsf{Mem}_{\mathrm{P}} \times \mathsf{Val}_{\mathsf{AP}}$$

by

$$\begin{array}{rcl}\eta_{\mathrm{P}}(m_h, d) & = & (\mu_{\mathrm{P}}(m_h), d)\\ \eta_{\mathrm{P}}(m_h, (\pi, m_d)) & = & (\mu_{\mathrm{P}}(m_h), (\pi, \mu_{\mathrm{P}}(m_d)))\\ \eta_{\mathrm{P}}(m_h, (\varpi, m_d)) & = & (\mu_{\mathrm{P}}(m_h), (\varpi, \mu_{\mathrm{P}}(m_d)))\end{array}$$

Next define α^V_{P} and γ^V_{P} by

$$\begin{array}{rcl}\alpha^V_{\mathrm{P}}(W) & = & \{\eta_{\mathrm{P}}(m, v) \mid (m, v) \in W\}\\ \gamma^V_{\mathrm{P}}(W_{\mathrm{P}}) & = & \{(m, v) \mid \eta_{\mathrm{P}}(m, v) \in W_{\mathrm{P}}\}\end{array}$$

It is then straightforward to obtain a Galois insertion

$$\widehat{\mathsf{Env}} \underset{\alpha^E_{\mathrm{P}}}{\overset{\gamma^E_{\mathrm{P}}}{\leftrightarrows}} \widehat{\mathsf{Env}_{\mathrm{P}}}$$

by setting

$$\begin{array}{rcl}\alpha^E_{\mathrm{P}}(R)(x) & = & \alpha^V_{\mathrm{P}}(R(x))\\ \gamma^E_{\mathrm{P}}(R_{\mathrm{P}})(x) & = & \gamma^V_{\mathrm{P}}(R_{\mathrm{P}}(x))\end{array}$$

To obtain a Galois insertion

$$\widehat{\mathsf{Store}} \underset{\alpha^S_{\mathrm{P}}}{\overset{\gamma^S_{\mathrm{P}}}{\leftrightarrows}} \widehat{\mathsf{Store}_{\mathrm{P}}}$$

we define

$$\begin{array}{rcl}\alpha^S_{\mathrm{P}}(S)(\varpi, m_{Pd}) & = & \alpha^V_{\mathrm{P}}(\cup\{S(\varpi, m_d) \mid \mu_{\mathrm{P}}(m_d) = m_{Pd}\})\\ \gamma^S_{\mathrm{P}}(S_{\mathrm{P}})(\varpi, m_d) & = & \gamma^V_{\mathrm{P}}(S_{\mathrm{P}}(\varpi, \mu_{\mathrm{P}}(m_d)))\end{array}$$

The proof of the theorem then proceeds as for Theorem 5.1; the crucial step is that the following fact holds:

Fact C.4 $\mu_{\mathrm{P}}(\mathsf{new}((l, m_h), W, S, (\pi, m_d))) =$
$\mathsf{new}_{\mathrm{P}}((l, \mu_{\mathrm{P}}(m_h)), \alpha^V_{\mathrm{P}}(W), \alpha^S_{\mathrm{P}}(S), (\pi, \mu_{\mathrm{P}}(m_d)))$.

Proof of Theorem A.1. The proof is by induction on the inference in the semantics. In all cases Tables 7, 3 and 4 are used to "unfold" the assumptions. Next the three facts below are used to establish the prerequisites for the application of the induction hypothesis. Finally the required result is obtained by simple calculations using the three facts below.

The case of recursive function definition is slightly more complicated than the other cases because of the coinductive nature of the definition of "$\in^{\Sigma,\Omega}_m$".

Fact C.5 *The abstract element can be enlarged:*

- *if* $\omega \in_m^{\Sigma,\Omega} W$ *and* $W \subseteq W'$ *then* $\omega \in_m^{\Sigma,\Omega} W'$
- *if* $\rho \in_m^{\Sigma,\Omega} R$ *and* $R \sqsubseteq R'$ *then* $\rho \in_m^{\Sigma,\Omega} R'$
- *if* $\sigma \in_m^{\Sigma,\Omega} S$ *and* $S \sqsubseteq S'$ *then* $\sigma \in_m^{\Sigma,\Omega} S'$

Fact C.6 *The store model and the value model can be enlarged: Assume* $\Sigma \leq \Sigma'$ *and* $\Omega \subseteq \Omega'$*. Then*

- *if* $\omega \in_m^{\Sigma,\Omega} W$ *then* $\omega \in_m^{\Sigma',\Omega'} W$
- *if* $\rho \in_m^{\Sigma,\Omega} R$ *then* $\rho \in_m^{\Sigma',\Omega'} R$
- *if* $\sigma \in_m^{\Sigma,\Omega} S$ *then* $\sigma \in_m^{\Sigma',\Omega'} S$

Fact C.7 *The memento can be changed: Assume* $(m, m') \in Y$*. Then*

- *if* $\omega \in_m^{\Sigma,\Omega} W$ *then* $\omega \in_{m'}^{\Sigma,\Omega} W\lceil Y \rceil$
- *if* $\rho \in_m^{\Sigma,\Omega} R$ *then* $\rho \in_{m'}^{\Sigma,\Omega} R\lceil Y \rceil$
- *if* $\sigma \in_m^{\Sigma,\Omega} S$ *then* $\sigma \in_{m'}^{\Sigma,\Omega} S\lceil Y \rceil$