Interprocedural Control Flow Analysis

Flemming Nielson and Hanne Riis Nielson

Department of Computer Science, University of Aarhus
Ny Munkegade, DK-8000 Aarhus C, Denmark
{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. 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. 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) into Control Flow Analysis (using Abstract Interpretation to induce the analyses from a collecting semantics).

1 Introduction

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 ([24,11,16] 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 [25] (à la call strings [23] or tokens [12]) 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 [8,9,18,19] whose least solution is then computed using graph-based ideas. However, the majority of papers on Control Flow Analysis (e.g. [24,25,11,16]) do not consider side-effects — a notable exception being [10].

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) [7]. It follows from Tarski's theorem [26] that the equations could equally well be presented as constraints: the least solution is the same.

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

S.D. Swierstra (Ed.): ESOP/ETAPS'99, LNCS 1576, pp. 20–39, 1999.

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 (see 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). Most papers on Data Flow Analysis (e.g. [23,13]) do not consider first-class procedures and therefore have no need for a component akin to Control Flow Analysis — a notable exception to this is [20].

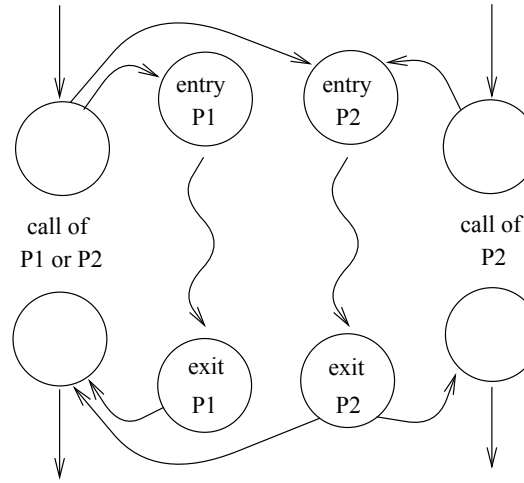

Fig. 1. Function call.

One approach deals with the interprocedural analysis by obtaining transfer functions for entire call statements [23,13] (and to some extent [3]). Alternatively, and as we shall do in this paper, one may dispense with formulating equations (or constraints) 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 [23,27].
- 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 [20,14,21]; an example is the use of so-called assumption sets that give information about the actual parameters.

Abstract Interpretation. In Abstract Interpretation [4], the systematic development of program analyses is likely to span a spectrum from *abstract* specifications (like [16] in the case of Control Flow Analysis), over *syntax-directed* specifications

(as in the present paper), to actual *implementations* in the form of constraints being generated and subsequently solved (as in [8,9,18,19,6]). 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. 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. To keep the specification compact we present the Control Flow Analysis in the form of a *succinct flow logic* [17]. 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 semantics. As in [10] we add components for tracking the side-effects occurring in the program and for explicitly propagating environments; for the side-effects this gives rise to a *flow-sensitive* analysis and for the environments we might coin the term *scope-sensitive*.

The analysis makes use of mementoes (for expressing context information in the manner of [5]) that are general enough that both call string based approaches (e.g. [23,25]) and dependent data approaches (in the manner of assumption-sets [20,14]) can be obtained by merely approximating the space of mementoes; this gives rise to a *context-sensitive* analysis. 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 [3,4,1].

Overview. Section 2 presents the syntax of a functional language with side-effects. Section 3 specifies the abstract domains and Section 4 the analysis itself. In Section 5 we then show how the classical developments mentioned above can be obtained as Abstract Interpretations. Finally, Section 6 concludes. — The full version of this paper is available as a technical report which establishes the correctness of the analysis and contains the proofs of the main results.

2 Syntax

We shall study a functional language with side-effects in the style of Standard ML [15]. It has variables $x, f \in \mathsf{Var}$, expressions e and constants c given by:

$$e ::= c \mid x \mid \mathtt{fn}_{\pi}\ x\ \mathtt{=>}\ e \mid \mathtt{fun}_{\pi}\ f\ x\ \mathtt{=>}\ e \mid (e_1\ e_2)^l \mid e_1\ \mathtt{;}\ e_2 \mid \mathtt{ref}_{\varpi}\ e$$
$$\mid\ \ \mathtt{!}e \mid e_1\ \mathtt{:=}\ e_2 \mid \mathtt{let}\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2 \mid \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$$

$$c ::= \mathtt{true} \mid \mathtt{false} \mid \mathtt{()} \mid \cdots$$

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

Table 1. Abstract domains.

Here $\texttt{fn}_\pi\ x\ \texttt{=>}\ e$ is a function that takes one argument and $\texttt{fun}_\pi\ f\ x\ \texttt{=>}\ e$ 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 occurrences of functions with a label $\pi \in \mathsf{Pnt_F}$ and all defining occurrences of references with a label $\varpi \in \mathsf{Pnt_R}$.

In Appendix A the semantics is specified as a big-step operational semantics with environments ρ and stores σ. The language has static scope rules and we give it a traditional call-by-value semantics using judgements of the form $\rho \vdash \langle e, \sigma_1 \rangle \to \langle \omega, \sigma_2 \rangle$.

3 Abstract Domains

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 [5]; in Section 5 we shall show how the set can be simplified into something more tractable.

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

Table 2. Caches.

Example 1. Consider the program "program" defined by:

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

The applications are performed in the order 3, 1 and 2. The mementoes of interest are going to be: $m_3 = ((3,\diamond), W_3, [\,], (x,\diamond)), m_1 = ((1,m_3), W_1, [\,], (z,\diamond)), m_2 = ((2,m_1), W_2, [\,], (z,\diamond))$ 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 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

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

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

- The *environment caches* $\mathcal{R}^d_F$ and $\mathcal{R}^c_F$: for each program point π, $\mathcal{R}^d_F(\pi)$ records the abstract environment at the definition point and $\mathcal{R}^c_F(\pi)$ records the same information but modified to each of the 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 $\mathcal{R}^d_F(\pi)(x)$ as (m_d, v) and in $\mathcal{R}^c_F(\pi)(x)$ as (m_c, v) where $m_d = \diamond$ in case of a top-level function and $m_c = ((l,\diamond), W, S, (\pi,\diamond))$ in case of a top-level application 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
$\mathcal{W}_F(\bullet\pi)$	$\{(m_3,(z,\diamond))\}$	$\emptyset$	$\{(m_1,(z,\diamond)),(m_2,(y,m_3))\}$
$\mathcal{W}_F(\pi\bullet)$	$\{(m_3,(y,m_3))\}$	$\emptyset$	$\{(m_1,(z,\diamond)),(m_2,(y,m_3))\}$
$\mathcal{S}_F(\bullet\pi)$	$[\,]$	$[\,]$	$[\,]$
$\mathcal{S}_F(\pi\bullet)$	$[\,]$	$[\,]$	$[\,]$
$\mathcal{R}^d_F(\pi)$	$[\,]$	$[\mathtt{x} \mapsto \{(m_3,(z,\diamond))\}]$	$[\,]$
$\mathcal{R}^c_F(\pi)$	$[\,]$	$[\,]$	$[\,]$
$\mathcal{M}_F(\pi)$	$\{m_3\}$	$\emptyset$	$\{m_1,m_2\}$

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 optimise the program. It can also be used as the basis for generating a set of constraints [17] whose least solution can be obtained using standard techniques (e.g. [2]).

Given a program e and the five caches $(\mathcal{R}^d_F, \mathcal{R}^c_F, \mathcal{M}_F, \mathcal{W}_F, \mathcal{S}_F)$ 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 $\perp$ 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) and then for the other constructs (Table 4). As in [16] 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 $\{(\diamond, (y, m_3))\}$. □

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 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

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

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

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

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

$$R, M \ \triangleright \ (e_1 \ e_2)^l : S_1 \to S_4 \ \& \ W$$
$$\quad \textsf{iff} \ R, M \ \triangleright \ e_1 : S_1 \to S_2 \ \& \ W_1 \ \wedge R, M \ \triangleright \ e_2 : S_2 \to S_3 \ \& \ W_2 \ \wedge$$
$$\quad\quad \forall \pi \in \{\pi \mid (m, (\pi, m_d)) \in W_1\} :$$
$$\quad\quad\quad \textsf{let} \ X = \overline{\textsf{new}}_\pi((l, M), W_2, S_3, W_1)$$
$$\quad\quad\quad\quad X_{dc} = \{(m_d, m_c) \mid (m_d, m_h, m_c) \in X\}$$
$$\quad\quad\quad\quad X_c \ = \{m_c \mid (m_d, m_h, m_c) \in X\}$$
$$\quad\quad\quad\quad X_{hc} = \{(m_h, m_c) \mid (m_d, m_h, m_c) \in X\}$$
$$\quad\quad\quad\quad X_{ch} = \{(m_c, m_h) \mid (m_d, m_h, m_c) \in X\}$$
$$\quad\quad\quad \textsf{in} \ \mathcal{R}^d_F(\pi)\lceil X_{dc} \rceil \sqsubseteq \mathcal{R}^c_F(\pi) \wedge X_c \subseteq \mathcal{M}_F(\pi) \wedge$$
$$\quad\quad\quad\quad W_2\lceil X_{hc} \rceil \subseteq \mathcal{W}_F(\bullet\pi) \wedge S_3\lceil X_{hc} \rceil \sqsubseteq \mathcal{S}_F(\bullet\pi) \wedge$$
$$\quad\quad\quad\quad \mathcal{W}_F(\pi\bullet)\lceil X_{ch} \rceil \subseteq W \wedge \mathcal{S}_F(\pi\bullet)\lceil X_{ch} \rceil \sqsubseteq S_4$$

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

Table 3. Analysis of the functional fragment.

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 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 $\mathcal{R}^d_F$ and then to analyse the body of the function in the context of the call which is specified by the caches $\mathcal{R}^c_F$, $\mathcal{M}_F$, $\mathcal{W}_F$ 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:

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

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

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

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.

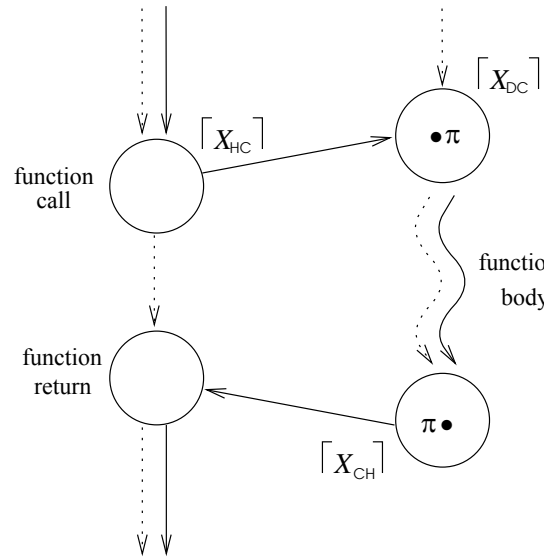

Fig. 2. Analysis of function call.

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 $\mathcal{R}_F^d(\pi)$ 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 $X_{dc} = \{(\diamond, m_1)\}$.) 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 $\lceil X_{dc} \rceil$ is used to modify the static environment of the definition point, the transfer function $\lceil X_{hc} \rceil$ is used to go from the application point to the function body and the transfer function $\lceil X_{ch} \rceil$ 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).

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 $\{(m_3, (z, \diamond))\}\lceil X_{hc} \rceil \subseteq \mathcal{W}_F(\bullet z)$ and $\mathcal{W}_F(z\bullet)\lceil X_{ch} \rceil \subseteq \{(m_3, (z, \diamond))\}$. Since $X = \{(\diamond, m_3, m_1)\}$ we have $X_{hc} = \{(m_3, m_1)\}$ 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, $X_{ch} = \{(m_1, m_3)\}$ 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. □

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

$$R, M \ \rhd \ e_1 \ ; \ e_2 : S_1 \to S_3 \ \& \ W_2$$
$$\quad \textsf{iff } R, M \ \rhd \ e_1 : S_1 \to S_2 \ \& \ W_1 \wedge R, M \ \rhd \ e_2 : S_2 \to S_3 \ \& \ W_2$$

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

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

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

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

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

Table 4. Analysis of the other constructs.

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 $\varphi^{[e,W]}_{\texttt{true}}(R)$ and $\varphi^{[e,W]}_{\texttt{false}}(R)$ whereas for the store we use the transfer functions $\phi^{[e,W]}_{\texttt{true}}(S_2)$ and $\phi^{[e,W]}_{\texttt{false}}(S_2)$. The result of both branches are possible for the whole construct.

As an example of the use of these transfer functions consider the expression `if` x `then` e_1 `else` e_2 where it will be natural to set

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

and similarly for $\varphi^{[x,W]}_{\texttt{false}}(R)$. Thus it will be possible to analyse each of the branches with precise information about x.

Little can be said in general about how to define such functions; to obtain a more concise statement of the theorems below we shall *assume* that the transfer functions $\phi^{\cdots}_{\cdots}$ and $\varphi^{\cdots}_{\cdots}$ of Table 4 are in fact the identities.

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

Table 5. Abstract domains for k-CFA.

5 Classical Approximations

***k*-CFA.** The idea behind k-CFA [11,24] 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

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

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.

To express the relationship between the two analyses 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 $\mu_0(m) = \varepsilon, \mu_{k+1}(\diamond) = \varepsilon, \mu_{k+1}((l, m), W, S, (\pi, m_d)) = l \hat{} \mu_k(m)$ 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 $\alpha_k^M(M) = \{\mu_k(m) \mid m \in M\}$ and $\gamma_k^M(M_k) = \{m \mid \mu_k(m) \in M_k\}$. Since α_k^M is surjective and defined in a pointwise manner there exists precisely one function such that

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

is a *Galois insertion* as studied in Abstract Interpretation [4]: 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

$$\widehat{\mathsf{Val}} \underset{\alpha_k^V}{\overset{\gamma_k^V}{\leftrightarrows}} \widehat{\mathsf{Val}}_k$$

we 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 taking $\eta_k(m_h, d) = (\mu_k(m_h), d)$, $\eta_k(m_h, (\pi, m_d)) = (\mu_k(m_h), (\pi, \mu_k(m_d)))$, and $\eta_k(m_h, (\varpi, m_d)) = (\mu_k(m_h), (\varpi, \mu_k(m_d)))$. Next define α_k^V and γ_k^V by $\alpha_k^V(W) = \{\eta_k(m, v) \mid (m, v) \in W\}$ and $\gamma_k^V(W_k) = \{(m, v) \mid \eta_k(m, v) \in W_k\}$. It is then straightforward to obtain a Galois insertion

$$\widehat{\mathsf{Env}} \underset{\alpha_k^E}{\overset{\gamma_k^E}{\leftrightarrows}} \widehat{\mathsf{Env}}_k$$

by setting $\alpha_k^E(R)(x) = \alpha_k^V(R(x))$ and $\gamma_k^E(R_k)(x) = \gamma_k^V(R_k(x))$. To obtain a Galois insertion

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

define $\alpha_k^S(S)(\varpi, m_{kd}) = \alpha_k^V(\bigcup\{S(\varpi, m_d) \mid \mu_k(m_d) = m_{kd}\})$ and $\gamma_k^S(S_k)(\varpi, m_d) = \gamma_k^V(S_k(\varpi, \mu_k(m_d)))$.

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

Theorem 1. *If* $(\mathcal{R}^d_{kF}, \mathcal{R}^c_{kF}, \mathcal{M}_{kF}, \mathcal{W}_{kF}, \mathcal{S}_{kF})$ *satisfies*

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

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

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

In the full version we establish the semantic correctness of the analysis of Section 4; it then follows 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 simple way by taking

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

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

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

where again this set may be slightly too large.

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

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

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

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

where l is the application point and π is the function called.

For functions being defined at top-level there is not likely 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.

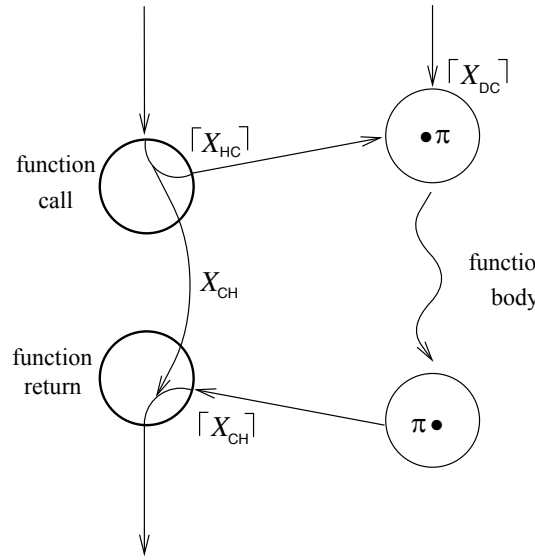

Fig. 3. Degenerate analysis of function call.

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 [27]; the scenario of [23] 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 1 still applies and therefore the semantic correctness result still carries over.

$$
\begin{array}{rl}
m_p \in \mathsf{Mem}_p & = \{\varepsilon\} \ \cup \ \mathcal{P}(\mathsf{Data} \cup \mathsf{Pnt_F} \cup \mathsf{Pnt_R}) \\
d \in \mathsf{Data} & = \cdots \qquad \text{(unspecified)} \\
(\pi, m_{pd}) \in \mathsf{Closure}_p & = \mathsf{Pnt_F} \times \mathsf{Mem}_p \\
(\varpi, m_{pd}) \in \mathsf{Cell}_p & = \mathsf{Pnt_R} \times \mathsf{Mem}_p \\
v_p \in \mathsf{Val}_{\mathsf{A}p} & = \mathsf{Data} \cup \mathsf{Closure}_p \cup \mathsf{Cell}_p \\
W_p \in \widehat{\mathsf{Val}}_p & = \mathcal{P}(\mathsf{Mem}_p \times \mathsf{Val}_{\mathsf{A}p}) \\
R_p \in \widehat{\mathsf{Env}}_p & = \mathsf{Var} \to \widehat{\mathsf{Val}}_p \\
S_p \in \widehat{\mathsf{Store}}_p & = \mathsf{Cell}_p \to \widehat{\mathsf{Val}}_p
\end{array}
$$

Table 6. Abstract domains for assumption sets.

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 [22] (that is somewhat less general).

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

$$\mathsf{new}_p : (\mathsf{Lab} \times \mathsf{Mem}_p) \times \widehat{\mathsf{Val}}_p \times \widehat{\mathsf{Store}}_p \times (\mathsf{Pnt_F} \times \mathsf{Mem}_p) \to \mathsf{Mem}_p$$

given by $\mathsf{new}_p((l, m_{ph}), W_p, S_p, (\pi, m_{pd})) = \{\mathsf{keep}_p(v_p) \mid (m_p, v_p) \in W_p\}$ where $\mathsf{keep}_p : \mathsf{Val}_{\mathsf{A}p} \to (\mathsf{Data} \cup \mathsf{Pnt_F} \cup \mathsf{Pnt_R})$ is given by $\mathsf{keep}_p(d) = d$, $\mathsf{keep}_p(\pi, m_{pd}) = \pi$, $\mathsf{keep}_p(\varpi, m_{pd}) = \varpi$.

Theoretical properties. We can now mimic the development performed above. The crucial point is the definition of a *surjective* mapping $\mu_p : \mathsf{Mem} \to \mathsf{Mem}_p$ showing how the precise mementoes of Table 1 are mapped into the approximative mementoes of Table 6. It is given by $\mu_p(\diamond) = \varepsilon$, and $\mu_p((l, m), W, S, (\pi, m_d))$ $= \{\mathsf{keep}'_p(v) \mid (m', v') \in W\}$. where $\mathsf{keep}'_p : \mathsf{Val}_{\mathsf{A}} \to (\mathsf{Data} \cup \mathsf{Pnt_F} \cup \mathsf{Pnt_R})$ is the obvious modification of keep_p to work on $\mathsf{Val}_{\mathsf{A}}$ rather than $\mathsf{Val}_{\mathsf{A}p}$. Based on μ_p we can now define Galois *insertions*

- (α^M_p, γ^M_p) between $\mathcal{P}(\mathsf{Mem})$ and $\mathcal{P}(\mathsf{Mem}_p)$
- (α^V_p, γ^V_p) between $\widehat{\mathsf{Val}}$ and $\widehat{\mathsf{Val}}_p$
- (α^E_p, γ^E_p) between $\widehat{\mathsf{Env}}$ and $\widehat{\mathsf{Env}}_p$
- (α^S_p, γ^S_p) between $\widehat{\mathsf{Store}}$ and $\widehat{\mathsf{Store}}_p$

very much as before and obtain the following analogue of Theorem 1:

Theorem 2. *If* $(\mathcal{R}^d_{pF}, \mathcal{R}^c_{pF}, \mathcal{M}_{pF}, \mathcal{W}_{pF}, \mathcal{S}_{pF})$ *satisfies*

$$R_p, M_p \ \rhd_p \ e : S_{p1} \to S_{p2} \ \& \ W_p$$

then $(\gamma^E_p \circ \mathcal{R}^d_{pF}, \gamma^E_p \circ \mathcal{R}^c_{pF}, \gamma^M_p \circ \mathcal{M}_{pF}, \gamma^V_p \circ \mathcal{W}_{pF}, \gamma^S_p \circ \mathcal{S}_{pF})$ *satisfies*

$$\gamma^E_p(R_p), \gamma^M_p(M_p) \ \rhd \ e : \gamma^S_p(S_{p1}) \to \gamma^S_p(S_{p2}) \ \& \ \gamma^V_p(W_p).$$

As before 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. At the same time we have used Abstract Interpretation to pass from the general mementoes of Section 3 to the more tractable mementoes of Section 5. In fact *all* our analyses are based on the specification of Tables 3 and 4.

Acknowledgement. This work has been supported in part by the DART project funded by the Danish Science Research Council.

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. Static determination of dynamic properties of recursive procedures. In E. J. Neuhold, editor, *Formal Description of Programming Concepts*. North Holland, 1978.
4. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. POPL '79*, pages 269–282, 1979.
5. 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.
6. 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.
7. M. S. Hecht. *Flow Analysis of Computer Programs*. North Holland, 1977.

8. N. Heintze. Set-based analysis of ML programs. In *Proc. LFP '94*, pages 306–317, 1994.
9. N. Heintze and J. Jaffar. An engine for logic program analysis. In *Proc. LICS '92*, pages 318–328, 1992.
10. S. Jagannathan and S. Weeks. Analyzing Stores and References in a Parallel Symbolic Language. In *Proc. LFP '94*, pages 294–305, 1994.
11. S. Jagannathan and S. Weeks. A unified treatment of flow analysis in higher-order languages. In *Proc. POPL '95*. ACM Press, 1995.
12. 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.
13. 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.
14. W. Landi and B. G. Ryder. Pointer-Induced Aliasing: A Problem Classification. In *Proc. POPL '91*, pages 93–103. ACM Press, 1991.
15. R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML.* MIT Press, 1990.
16. F. Nielson and H. R. Nielson. Infinitary Control Flow Analysis: a Collecting Semantics for Closure Analysis. In *Proc. POPL '97*. ACM Press, 1997.
17. 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.
18. 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.
19. J. Palsberg. Closure analysis in constraint form. *ACM TOPLAS*, 17 (1):47–62, 1995.
20. 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.
21. E. Ruf. Context-insensitive alias analysis reconsidered. In *Proc. PLDI '95*, pages 13–22. ACM Press, 1995.
22. 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.
23. 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.
24. O. Shivers. Control flow analysis in Scheme. In *Proc. PLDI '88*, volume 7 (1) of *ACM SIGPLAN Notices*, pages 164–174. ACM Press, 1988.
25. O. Shivers. The semantics of Scheme control-flow analysis. In *Proc. PEPM '91*, volume 26 (9) of *ACM SIGPLAN Notices*. ACM Press, 1991.
26. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309, 1955.
27. 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

Table 7. Operational semantics.

A Semantics

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:

$$\begin{aligned} \iota &\in \mathsf{Loc} = \cdots && \text{(unspecified)} \\ \omega &\in \mathsf{Val} \\ \omega &::= c \mid \mathsf{close}\ (\mathtt{fn}_\pi\ x\ \mathtt{=>}\ e)\ \mathsf{in}\ \rho \mid \mathsf{close}\ (\mathtt{fun}_\pi\ f\ x\ \mathtt{=>}\ e)\ \mathsf{in}\ \rho \mid \iota \end{aligned}$$

$$
\begin{array}{l}
R, M \ \rhd\ \mathtt{ref}_{\varpi}\ e : S_1 \to S_3\ \&\ W' \\
\quad \text{iff } R, M \ \rhd\ e : S_1 \to S_2\ \&\ W \wedge \{(m, (\varpi, m)) \mid m \in M\} \subseteq W' \wedge \\
\qquad \forall m \in M : S_2 \oplus ((\varpi, m), W) \sqsubseteq S_3
\end{array}
$$

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

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

Table 8. Dealing with reference counts.

$$
\begin{array}{rcl}
e & ::= & \cdots \mid \iota \\
\rho & \in & \mathsf{Env} = \mathsf{Var} \to_{\mathrm{fin}} \mathsf{Val} \\
\sigma & \in & \mathsf{Store} = \mathsf{Loc} \to_{\mathrm{fin}} \mathsf{Val}
\end{array}
$$

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; the clauses themselves should be fairly straightforward. (We should also note that the choice of big-step operational semantics is not crucial for the development.)

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 [27] we could change the definition of $\widehat{\mathsf{Store}}$ (in Table 1) to have

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

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).

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

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

We also need a new operation for removing a reference:

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

where

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

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