Formal Aspects of Computing (1992) 4: 727–754

Formal Aspects of Computing

A Security Flow Control Algorithm and Its Denotational Semantics Correctness Proof

Masaaki Mizuno and David Schmidt

Computing and Information Sciences Department, Kansas State University, Manhattan, USA

Keywords: Abstract interpretation; Denotational semantics; Information flow control; Security

Abstract. We derive a security flow control algorithm for message-based, modular systems and prove the algorithm correct. The development is noteworthy because it is completely rigorous: the flow control algorithm is derived as an abstract interpretation of the denotational semantics of the programming language for the modular system, and the correctness proof is a proof by logical relations of the congruence between the denotational semantics and its abstract interpretation.

Effectiveness is also addressed: we give conditions under which an abstract interpretation can be computed as a traditional iterative data flow analysis, and we prove that our security flow control algorithm satisfies the conditions. We also show that symbolic expressions (that is, data flow values that contain unknowns) can be used in a convergent, iterative analysis. An important consequence of the latter result is that the security flow control algorithm can analyse individual modules in a system for well formedness and later can link the analyses to obtain an analysis of the entire system.

1. Introduction

Flow of information must be regulated in message-based, modular systems that deal with classified information. For example, let {*unclassified, classified, secret, topsecret*} be a set of security classifications, linearly ordered from left to right (e.g., *classified* $\sqsubseteq$ *secret*). Readers of messages are given security clearances, e.g., a reader with a *secret* clearance may read *secret* (or *classified* or *unclassified*) messages, but

Correspondence and offprint requests to: D. Schmidt, Computing and Information Sciences Department, Kansas State University, Manhattan, KS 66506, USA.

not *topsecret* ones. It is essential that the security of a system of readers, writers, and messages is validated by some form of *flow control algorithm*. The correctness of the flow control algorithm itself is, of course, crucial.

The pioneering worker in the area of flow control algorithms was Denning ([Den75], [Den76], [DeD77]). She developed a compile-time algorithm for certifying the secure execution of a program where the security class of each message, variable and file remained constant throughout the lifetime of the program [DeD77]. Since the security class of every program variable and formal parameter must be explicitly specified, separate versions of functionally equivalent procedures are required to handle different security classes of actual parameters.

Another noteworthy effort was made by Andrews and Reitman, who developed a compile-time certification technique based on Hoare's logic [AnR80]. Their technique allows the security class of a variable to change during execution of the program, but application to modular systems is difficult, because the verification of a procedure invocation requires verification of the called procedure.

Proofs of correctness of the above methods were intuitive or informal.

The method we study in this paper was developed by Mizuno ([Miz87], [MiO87], [Miz89]). Mizuno's method analyses modular systems, where readers and writers are modelled by procedures, and messages are parameters. It has these features:

- The security classes of a procedure's local variables and parameters can remain constant or can change during execution. Procedures may also use global variables whose values persist after procedure termination. The security classes of global variables must be constant.
- Procedures and global variables are grouped into modules. A compile-time algorithm verifies the security of an individual procedure in a module and outputs a "summary data structure" that describes the module's behaviour. A link-time algorithm certifies a system of modules by combining the summary data structures and validating their consistency.

In this paper, we present a rigorous, formal, correctness proof of Mizuno's method. It is, to our knowledge, the first such correctness proof for a security flow control algorithm. We begin with a *denotational semantics* ([Sch88], [Sto77]) of the programming language one uses to code the modules, and we show that the compile-time analysis algorithm is an *abstract interpretation* ([CoC77], [Nie83]) of the denotational semantics. A *congruence proof by logical relations* ([Nie89], [Plo80]) establishes the correctness of the compile-time algorithm. This allows the language's semantics to be *staged* ([JøS86], [MoW88]) into a compile-time analysis semantics and a run-time semantics, where security classifications need not be monitored in the latter.

Analysis of a system of modules is formalised by generalising the abstract interpretation to operate upon symbolic expressions (*polynomials* [Gra79]) that represent references to procedures in external modules. We prove that the compile-time analysis can compute upon such symbolic expressions, and we show how the results can be linked into a correct analysis of an entire system.

We also address effectiveness. We state sufficient criteria for implementing an abstract interpretation as a traditional, iterative, data flow analysis, and we show that Mizuno's compile-time analysis fits the criteria. We also prove that the iterative analysis can be used with polynomials. Hence, the formal derivation of Mizuno's method, as an abstract interpretation, matches the pragmatic implementation of it, as a pair of iterative, compile-time, link-time algorithms.

In the sections that follow, we review the information flow policy enforced by Mizuno's method, we describe the compile-time and link-time algorithms, and we formally derive the algorithms and prove them correct.

2. Information Flow Policy

It is helpful to think of security classifications as "data types" and information flow as "type propagation". Mizuno's method enforces Denning's information flow policy [Den76]. An *information flow* from variable X to Y occurs when information in X is transferred to Y. Information flow indicates that information in Y can be used to deduce information in X. An assignment, e.g., Y := W + X causes information flow from W and X to Y. If W held a *secret* value and X held a *topsecret* one, then Y receives *topsecret* information (since *topsecret* = *secret* ⊔ *topsecret*).

Flows are *explicit* or *implicit*. An explicit flow from a variable A to X occurs when a statement assigns information from A to X, as in the assignment statement above. An implicit flow from A to X occurs when the execution of a statement that assigns some information to X is conditioned upon a test that references A. For example:

if A > 0 **then** X := Y **else** X := Z

causes an explicit flow from Y to X when A > 0 and from Z to X when A ⩽ 0. The statement also causes an implicit flow from A to X, regardless of the value of A. Implicit flow is significant – in the above example, if Y has value 0 and Z has value 1, then after execution of the if-statement, information about the value of A can be deduced from the value of X. Thus, X should have a security classification at least as strong as A's.

The underlying theory of information flow is based on a pointed, finite height, sup-semilattice structure $(SC, \sqsubseteq, \bot, \sqcup)$, where:

- *SC* is a set of security classes.
- $\sqsubseteq$ is a partial ordering on *SC* such that there are no infinitely ascending chains ([Coc77], [Nie83]).
- $\bot$ is the least element of *SC*.
- $\sqcup$ is the join ("least upper bound") operation on *SC*.

All information values in a system are tagged with security values from *SC*. A program variable may be either *statically* or *dynamically* bound to a security class. A statically bound variable is assigned a fixed security class, s, at the time of its definition. All values assigned to it must have a security class s' such that $s' \sqsubseteq s$. The security class of a dynamically bound variable changes with the class of the value assigned to it.

If X is a variable, then its security class is denoted by X (that is, in *italic*). The system's information flow policy may be stated as follows: if Y is a statically bound variable, then an information flow from X to Y is secure iff $X \sqsubseteq Y$ holds. If Y is a dynamically bound variable, Y becomes X and no security violation occurs.

3. Overview of the Algorithms

It is helpful to think of the security control algorithms as data flow analysis algorithms for type inference. There are a compile-time algorithm and a link-time algorithm. The compile-time algorithm certifies the security of each procedure independently of other procedures, generating a *summary data structure* of symbolic equations that encode the security classes of the procedure's in–out-parameters. The link-time algorithm completes the certification of a system by combining the summary data structures and calculating interprocedural information flows. The strategy is easily adapted to a module-based system, where a summary data structure is generated for each module.

Let a procedure declaration have the form

procedure PROC (**in** X:T; **out** X′:T′) = D; C **end**

where the **in** parameter is "call by value", the **out** parameter is "call by result", D is the set of local declarations, and C is the procedure's body. The T and T′ state whether the formal parameters have static or dynamic security classifications.

The compile-time algorithm, which is an iterative data flow analysis [ASU86], traces the information flow through the control paths of procedure PROC, verifying that every statically bound variable V receives values whose security classes are $\sqsubseteq V$.

External procedure calls cause problems. For example, let PROC be

procedure PROC (**in** A:**dynamic**; **out** B:**dynamic**) =
  var X:**static** *classified*; **var** Y:**dynamic**; X := 0; **call** P(**in** X, **out** Y);
  B := 3 + Y + A; X := B **end**

The assignment X := 0 is safe (assuming that numerals have class *unclassified*), but the assignment B := 3 + Y + A causes problems: the class of B is unknown until the class of in-parameter A and the analysis of external procedure P are known. So, the compile-time algorithm uses the expression $\mathit{unclassified} \sqcup P.Y \sqcup A$ to denote B, where $P.Y$ denotes the class of the out-parameter from the call to P. Further, the symbolic equation $P.X = \mathit{classified}$ is saved in the summary data structure for PROC; $P.X$ denotes the class of the in-parameter to the call to P. When the data structures for PROC and P are linked, the value of $P.Y$ can be calculated from P's summary data structure and $P.X = \mathit{classified}$. The validation of the assignment X := B must be postponed until link-time, so the compile-time algorithm adds the symbolic (in)equation $\mathit{unclassified} \sqcup P.Y \sqcup A \sqsubseteq \mathit{classified}$ to PROC's summary structure. Finally, the equation $B = \mathit{unclassified} \sqcup P.Y \sqcup A$, which defines the class of the output parameter of PROC, is generated and added to PROC's summary structure.

In the above example, P.X and B are *output variables*, and A and P.Y are *input* variables of PROC. In general, input variables of a procedure PROC are

1. Actual **in** parameters of PROC
2. Formal **out** parameters returned from procedures that are called by PROC
3. Global variables read by PROC (recall that global variables must be statically bound)

Possible output variables of PROC are

4. Actual **out** parameters of PROC
5. Formal **in** parameters to procedures that are called by PROC
6. Global variables written by PROC

If input variables (1) and (2) are dynamically bound, their security classes cannot be determined until link-time. During compilation, the classes of these variables are represented by symbolic expressions, like the ones seen in the example above. The compile-time algorithm also generates a symbolic equation for each output variable (4) and (5). If the class of value assigned to a statically bound variable cannot be determined at compile-time, a symbolic inequation is generated (cf. the above example).

The link-time algorithm collects the symbolic equations for all procedures and calculates, for each variable corresponding to a parameter, the run-time security class of information flowing to the variable. This is done by solving the set of symbolic equations with the usual iterative least fixed point calculation ([ASU86, CoC77]). The results are used to evaluate the symbolic inequations that correspond to the unresolved classes of static variables. If all inequations hold true, the system is certified secure and allowed to execute.

3.1. The Compile-Time Algorithm

We overview the compile-time algorithm, emphasising its treatment of implicit information flow, and show an example. Full details of the compile-time algorithm are given in [Miz87] and [MiO87]. Consider

if A > 0 **then** X := B **else** X := C

The algorithm deduces that X must be $B \sqcup C \sqcup A$, since there is implicit flow from A and explicit flows from B and C to X. For a while-loop, the number of iterations of the loop's body is unknown at compile-time, so the compile-time algorithm accounts for the "worst case" information flow. Consider

A := X; **while** C > 0 **do call** P(**in** A, **out** B); **call** Q(**in** B, **out** A) **od**

On the first iteration, the class of **in** parameter A to P is $X \sqcup C$, due to explicit flow from X and implicit flow from C. In subsequent iterations, A receives information from the out-parameter from Q. Thus, $P.A = X \sqcup C \sqcup Q.A$.

There is also *implicit interprocedural information flow*. In the above example, there is implicit flow from C into every global variable used in P's code, and into those global variables used by procedures called by P, and so on. The compile-time algorithm accounts for this flow by constructing a symbolic equation $P.implicit = C \sqcup implicit$, where $P.implicit$ defines the implicit interprocedural flow into P, and *implicit* denotes the implicit interprocedural flow incoming to the procedure calling P. When an execution starts with the "main" procedure, *implicit* for "main" is $\perp$, the least security class.

Figure 1 presents an example. Procedure MAIN calls F, and F and G call each other. We show the actions of the compile-time algorithm on F. The security class of variable C in line (a) is $\perp \sqcup implicit$, i.e., *implicit*, since there is an explicit flow from 2 (whose class is $\perp$), and there is implicit interprocedural flow from the caller of F. In line (b), input parameter C has security classification *implicit* (from line (a)), and it also receives implicit flow from A as well as implicit interprocedural flow; the following equation is constructed:

$G.C = implicit \sqcup A \sqcup implicit$, that is, $G.C = A \sqcup implicit$

The algorithm also generates an equation for the implicit interprocedural flow into G:

$G.implicit = A \sqcup implicit$

```
program MAIN
  var A, B: dynamic; SV1: static topsecret;
  SV1 := 4; A := SV1;
  call F(in A, out B);
  SV1 := B+20
end MAIN
module M
  global var SV2: static secret;
  procedure F(in A: dynamic; out B: dynamic);
    var C: dynamic;
    C := 2;                    (a)
    if A > 0
      then
        call G(in C, out B);   (b)
        SV2 := B;              (c)
      else B := 10             (d)
  end F
end module M
module N
  procedure G(in X: dynamic; out Y: dynamic);
    var SV3; static confidential;
    if X < 100
      then call F(in X, out Y); SV3 := Y .
      else Y := 3
  end G
end module N
```

Fig. 1. A modular system

In line (c), static variable SV2 is affected by an explicit flow from B (whose class is $G.B$), and implicit flow from A, and implicit interprocedural flow. The resulting inequation for SV2 is

$$G.B \sqcup A \sqcup implicit \sqsubseteq secret$$

Output variable B is assigned values in lines (b) and (d); there is implicit flow from A as well as implicit interprocedural flow. The following equation is generated:

$$B = G.B \sqcup A \sqcup implicit$$

The summary data structures for F, MAIN, and G are shown in Fig. 2.

3.2. The Link-Time Algorithm

The link-time algorithm makes the correspondence between formal and actual parameters and solves the equation sets. Its full description is in [Miz89]; here we summarise. Consider a procedure P. Its summary data structure contains symbolic equations for its output variables and inequations for its static variables. Say that P has in-parameter A, and say that external procedures Q and R call P with actual in-parameters B and C, respectively. To account for worst case information flow, the algorithm constructs an equation that binds A to all its actual parameters: $A =$

Symbolic equations for MAIN:
$F.B \sqsubseteq topsecret$ (for SV1)
$F.implicit = \bot$
$F.A = topsecret$
Symbolic equations for F:
$G.B \sqcup A \sqcup implicit \sqsubseteq secret$ (for SV2)
$G.implicit = A \sqcup implicit$
$G.C = A \sqcup implicit$
$B = G.B \sqcup A \sqcup implicit$
Symbolic equations for G:
$X \sqcup F.Y \sqcup implicit \sqsubseteq confidential$ (for SV3)
$F.implicit = X \sqcup implicit$
$F.X = X \sqcup implicit$
$Y = X \sqcup F.Y \sqcup implicit$

Fig. 2. Summary data structures

$P.B \sqcup P.C$, where $P.B$ is defined by an equation in Q's summary structure, and $P.C$ is defined by an equation in R's summary structure. (Hereon, we subscript the names to clarify their sources, e.g.: $A_P = P.B_Q \sqcup P.C_R$.) Since Q and R call P, an equation for the implicit flow into P must also be constructed: $implicit_P = P.implicit_Q \sqcup P.implicit_R$.

We can best understand the method by linking the equation sets in Fig. 2. Since MAIN calls F, we define

$$F.B_{MAIN} = B_F$$

to give the value of the variable B returned by F. The completed set of equations for MAIN is

$$F.implicit_{MAIN} = \bot$$
$$F.A_{MAIN} = topsecret$$
$$F.B_{MAIN} = B_F$$

(The inequation $F.B \sqsubseteq topsecret$ is saved for later.)

The equation set for F is augmented by equations for input parameter A, the output parameter, B, from G, and for implicit flow from the callers of F. The completed equation set is

$$G.implicit_F = A_F \sqcup implicit_F$$
$$G.C_F = A_F \sqcup implicit_F$$
$$B_F = G.B_F \sqcup A_F \sqcup implicit_F$$
$$A_F = F.A_{MAIN} \sqcup F.X_G$$
$$G.B_F = Y_G$$
$$implicit_F = F.implicit_{MAIN} \sqcup F.implicit_G$$

The equations for A_F and $implicit_F$ reference values defined by the equation sets for MAIN and G, since both call F.

Finally, the completed equation set for G is

$$F.implicit_G = X_G \sqcup implicit_G$$
$$F.X_G = X_G \sqcup implicit_G$$
$$Y_G = X_G \sqcup F.Y_G \sqcup implicit_G$$

$F.implicit_{MAIN} = \bot$
$F.A_{MAIN} = topsecret$
$F.B_{MAIN} = topsecret$

$B_F = topsecret$
$G.implicit_F = topsecret$
$G.C_F = topsecret$
$A_F = topsecret$
$G.B_F = topsecret$
$implicit_F = topsecret$

$Y_G = topsecret$
$F.implicit_G = topsecret$
$F.X_G = topsecret$
$X_G = topsecret$
$F.Y_G = topsecret$
$implicit_G = topsecret$

Fig. 3. Solutions to fixed point calculations

$X_G = G.C_F$
$F.Y_G = B_F$
$implicit_G = G.implicit_F$

Now we solve all equations simultaneously with the usual iterative least fixed point calculation ([ASU86], [CoC77]). The results are shown in Fig. 3. Based on the solutions, we substitute into the inequations:

For SV1: $F.B_{MAIN} \sqsubseteq topsecret \equiv topsecret \sqsubseteq topsecret \equiv true$

For SV2: $A_F \sqcup G.B_F \sqcup implicit_F \sqsubseteq secret \equiv topsecret \sqcup topsecret \sqcup topsecret \sqsubseteq secret \equiv false$

For SV3: $X_G \sqcup F.Y_G \sqcup implicit_G \sqsubseteq confidential \equiv topsecret \sqcup topsecret \sqcup topsecret \sqsubseteq confidential \equiv false$

and the system is found to be potentially insecure in its treatment of SV2 and SV3.

4. Derivation and Proof of Correctness

The derivation of the compile-time and link-time algorithms is nontrivial, so we present it in stages. We begin with a while-loop language with dynamic variables and then extend the language with:

1. Static variables
2. Input parameters
3. Procedures and calls to external procedures
4. Linkages of procedures, recursion and global variables

At each stage, we begin with a standard ("full") denotational semantics of the language. We define an abstract interpretation [CoC77] of the standard semantics and prove it safe with respect to the standard semantics. The abstract interpretation is a formal description of the flow control algorithm, and we show how it defines the iterative algorithm described in the previous section. We assume familiarity with elementary denotational semantics ([Sch88], [Sto77]) and denotational semantics-based abstract interpretation ([BHA86], [Don82], [Nie83], [Nie85], [Nie89]).

4.1. The While-Loop Language

We start with a while-loop language that has only dynamic variables, that is, the

$\mathbf{C}$: Command $\to$ *Store* $\to$ *Poststore*

$\mathbf{C}[\![\mathrm{I} := \mathrm{E}]\!] = \mathit{update}\ \mathrm{I}\ \mathbf{E}[\![\mathrm{E}]\!]$

$\mathbf{C}[\![\mathrm{C}_1;\mathrm{C}_2]\!] = \mathbf{C}[\![\mathrm{C}_2]\!]\ \mathit{comp}\ \mathbf{C}[\![\mathrm{C}_1]\!]$

$\mathbf{C}[\![\textbf{if}\ \mathrm{E}\ \textbf{then}\ \mathrm{C}_1\ \textbf{else}\ \mathrm{C}_2]\!] = \mathit{cond}\ (\mathbf{I}[\![\mathrm{C}_1]\!] \cup \mathbf{I}[\![\mathrm{C}_2]\!])\ \mathbf{E}[\![\mathrm{E}]\!]\ \mathbf{C}[\![\mathrm{C}_1]\!]\ \mathbf{C}[\![\mathrm{C}_2]\!]$

$\mathbf{C}[\![\textbf{while}\ \mathrm{E}\ \textbf{do}\ \mathrm{C}]\!] = \mathit{fix}(\lambda f.\mathit{cond}\ \mathbf{I}[\![\mathrm{C}]\!]\ \mathbf{E}[\![\mathrm{E}]\!]\ (f\ \mathit{comp}\ \mathbf{C}[\![\mathrm{C}]\!])\ \mathit{skip})$

$\mathbf{E}$: Expression $\to$ *Store* $\to$ *Expressible*

$\mathbf{E}[\![\mathrm{K}]\!] = \mathit{put}\ \mathrm{K}$ (*Note:* K represents constants, e.g., numerals.)

$\mathbf{E}[\![\mathrm{I}]\!] = \mathit{access}\ \mathrm{I}$

$\mathbf{E}[\![\mathrm{op}\ \mathrm{E}_1\ \mathrm{E}_2]\!] = \mathit{do}\ \mathrm{op}\ \mathbf{E}[\![\mathrm{E}_1]\!]\ \mathbf{E}[\![\mathrm{E}_2]\!]$

$\mathbf{I}$: Command $\to$ $\mathbb{P}$(Identifier)

$\mathbf{I}[\![\mathrm{I} := \mathrm{E}]\!] = \{\mathrm{I}\}$

$\mathbf{I}[\![\mathrm{C}_1;\mathrm{C}_2]\!] = \mathbf{I}[\![\mathrm{C}_1]\!] \cup \mathbf{I}[\![\mathrm{C}_2]\!]$

$\mathbf{I}[\![\textbf{if}\ \mathrm{E}\ \textbf{then}\ \mathrm{C}_1\ \textbf{else}\ \mathrm{C}_2]\!] = \mathbf{I}[\![\mathrm{C}_1]\!] \cup \mathbf{I}[\![\mathrm{C}_2]\!]$

$\mathbf{I}[\![\textbf{while}\ \mathrm{E}\ \textbf{do}\ \mathrm{C}]\!] = \mathbf{I}[\![\mathrm{C}]\!]$

where:

update: Identifier $\to$ (*Store* $\to$ *Expressible*) $\to$ *Store* $\to$ *Poststore*

comp: (*Store* $\to$ *Poststore*) $\to$ (*Store* $\to$ *Poststore*) $\to$ *Store* $\to$ *Poststore*

cond: $\mathbb{P}$(Identifier) $\to$ (*Store* $\to$ *Expressible*) $\to$ (*Store* $\to$ *Poststore*) $\to$ (*Store* $\to$ *Poststore*) $\to$ *Store* $\to$ *Poststore*

skip: *Store* $\to$ *Poststore*

put: *Value* $\to$ *Store* $\to$ *Expressible*

access: Identifier $\to$ *Store* $\to$ *Expressible*

do: (*Expressible* $\times$ *Expressible* $\to$ *Expressible*) $\to$ (*Store* $\to$ *Expressible*) $\to$ (*Store* $\to$ *Expressible*) $\to$ *Store* $\to$ *Expressible*

are defined in the figures that follow.

(*Note:* $\mathbb{P}$(Identifier) is the set of all subsets of Identifier, discretely ordered.)

Fig. 4. Core semantics

security classifications of variables vary as the program executes. Figure 4 gives the "core semantics" of the programming language we study. We use a "factorised semantics" in the style of [JoM86], [Nie83] and [Nie85]. The core semantics states that a command is a mapping from an input store to an output store, called a "poststore". The standard interpretation of the operators in the core semantics is given in Fig. 5. The core semantics composed with the standard interpretation is called the "standard semantics" or "full semantics", and we write $\mathbf{C}_{full}$ to denote it. $\mathbf{C}_{full}$ shows that the values in storage cells are pairs of the form (t, v), where $t \in Sec$ and $v \in Value$. $\mathbf{C}_{full}$ defines an interpreter for the language; indeed, values v can be tagged with security classes s at run-time, but this is inefficient.

The interpretations in Fig. 5 are more or less obvious; only *cond* needs explanation. It formalises the implicit flow of the security classification of the test expression into the arms of the conditional. For expression denotation b and command denotations c_1 and c_2, $(cond\ S\, b\, c_1\, c_2\, s)$ evaluates the test $(b\, s)$, augments the identifiers in S by the security classification of the test and selects c_1 or c_2.

We now define two additional interpretations: the *abstract* ("compile-time") semantics, seen in Fig. 6, and the *execution* ("run-time") semantics, given in Fig. 7. The abstract interpretation composed with the core semantics is denoted $\mathbf{C}_{abs}$, and the execution interpretation composed with the core semantics is denoted $\mathbf{C}_{exec}$.

The $\mathbf{C}_{full}$ semantics has been "staged" into a compile-time semantics $\mathbf{C}_{abs}$ and a run-time semantics $\mathbf{C}_{exec}$ ([JøS86], [MoW88]). In particular, $\mathbf{C}_{abs}$ maps an input

$s \in Store = (\text{Identifier} \to Storable)$ (assume the Identifier set is finite)
$p \in Poststore = Store_\perp$ (*Note:* $\perp$ represents nontermination.)
$Storable = Sec \times Value$
$Expressible = Storable$
$t \in Sec =$ a finite height, pointed, sup-semilattice of security classifications
$v \in Value =$ primitive values, e.g., integers, booleans
$update\ i\,f = \lambda s.[i \mapsto (f s)]s$
$c_2\ comp\ c_1 = \lambda s.\text{let } s' = (c_1 s) \text{ in } (c_2 s')$
$cond\ S\,b\,c_1\,c_2 = \lambda s.\text{let } (t,v) = (b\,s) \text{ in}$
$\quad \text{let } s' = (\lambda i.\text{let } (t',v') = (s\,i) \text{ in } i \in S \to ((t \sqcup t'), v') \,[\!]\, (t',v'))$
$\quad\quad \text{in } v \to (c_1 s') \,[\!]\, (c_2 s')$
$skip = \lambda s.s$
$put\ k = \lambda s.(\perp, k)$ (*Note:* $\perp \in Sec$ represents the minimal security classification.)
$access\ i = \lambda s.(s\,i)$
$do\ op\,f\,g = \lambda s.\text{let } (t_1,v_1) = (f s) \text{ in let } (t_2,v_2) = (g\,s) \text{ in } (t_1 \sqcup t_2, op\ v_1\,v_2)$
(*Note:* "let $s =$ " is *strict* on $\perp \in Poststore$ arguments: "let $s = \perp$ in e" equals $\perp$.)

Fig. 5. Full interpretation

$a \in Store = \text{Identifier} \to Storable$
$Poststore = Store$
$Storable = Sec$
$Expressible = Storable$
$Sec =$ as in Fig. 2
$update\ i\,f = \lambda a.[i \mapsto (f a)]a$
$c_2\ comp\ c_1 = \lambda a.c_2(c_1\,a)$
$cond\ S\,b\,c_1\,c_2 = \lambda a.\text{let } a' = (\lambda i.i \in S \to (b\,a) \sqcup (a\,i) \,[\!]\, (a\,i)) \text{ in } (c_1\,a') \sqcup (c_2\,a')$
$skip = \lambda a.a$
$put\ k = \lambda a.\perp$
$access\ i = \lambda a.(a\,i)$
$do\ op\,f\,g = \lambda a.(f a) \sqcup (g\,a)$

Fig. 6. Abstract interpretation

$s \in Store = (\text{Identifier} \to Storable)_\perp$
$Poststore = Store_\perp$
$Storable = Value$
$Expressible = Storable$
$Value =$ as in Fig. 2
$update\ i\,f = \lambda s.\ [i \mapsto (f s)]s$
$c_2\ comp\ c_1 = \lambda s.\ \text{let } s' = (c_1\,s) \text{ in } (c_2 s')$
$cond\ b\,c_1\,c_2 = \lambda s.\ (b\,s) \to (c_1\,s) \,[\!]\, (c_2\,s)$
$skip = \lambda s.s$
$put\ k = \lambda s.k$
$access\ i = \lambda s.\ (s\,i)$
$do\ op\,f\,g = \lambda s.op\,(f s)\,(g\,s)$
(*Note:* "let $s =$ " is strict on $\perp$ arguments.)

Fig. 7. Execution interpretation

store of variables and their initial security classifications into an output poststore of variables and their final security classifications. We first prove that the abstract and execution semantics, working together, safely approximate the full semantics. For clarity, a domain D in interpretation i will be denoted D_i (e.g., $Store_{abs}$).

Proposition 1. $\mathbf{C}_{exec}$ is a "projection" of $\mathbf{C}_{full}$ in the sense that for all $C \in$ Command, for all $s \in Store_{full}$, $\mathbf{C}_{full}[\![C]\!]s = \bot = \mathbf{C}_{exec}[\![C]\!](second\ s)$ or else $second\ (\mathbf{C}_{full}[\![C]\!]s) = \mathbf{C}_{exec}[\![C]\!](second\ s))$, where $second: Store_{full} \to Store_{exec}$ is defined as $second = \lambda s.\lambda i.(s\, i)\downarrow 2$.

Proof. The proof is by induction on the structure of Command, proving $\mathbf{C}_{full}[\![C]\!]\ proj_{Store \to Poststore}\ \mathbf{C}_{exec}[\![C]\!]$, for the logical relation ([Nie89], [Plo80]) $proj$ between the domains of $\mathbf{C}_{full}$ and $\mathbf{C}_{exec}$:

$v\ proj_{Storable}\ v'$ iff $v \downarrow 2\ proj_{Value}\ v'$
$v\ proj_{Expressible}\ v'$ iff $v\ proj_{Storable}\ v'$
$v\ proj_{Value}\ v'$ iff $v = v'$
$s\ proj_{Store}\ s'$ iff for all $i \in$ Identifier, $(s, i)\ proj_{Storable}(s'\ i)$
$p\ proj_{Poststore}\ p'$ iff $(p = \bot = p')$ or $p\ proj_{Store}\ p'$
$f\ proj_{D_1 \to D_2} f'$ iff for all $d \in D_{1full}$, $d' \in D_{1exec}$, $d\ proj_{D_1}\ d'$ implies $(f\,d)\ proj_{D_2}$ $(f'\,d')$

The key to the proof is showing, for each operator $f: D \to D'$ named in Fig. 4, that $f_{full}\ proj_{D \to D'}\ f_{abs}$. □

Proposition 1 states that the execution semantics calculates the same output store as the full semantics.

Proposition 2. $\mathbf{C}_{abs}$ is a "conservative projection" of $\mathbf{C}_{full}$ in the sense that for all $C \in$ Command, for all $s \in Store_{full}$, $\mathbf{C}_{full}[\![C]\!]s = \bot$ or else $first(\mathbf{C}_{full}[\![C]\!]s) \sqsubseteq \mathbf{C}_{abs}[\![C]\!]$ $(first\ s))$, where $first: Store_{full} \to Store_{abs}$ is defined as $first = \lambda s.\lambda i.\ (s\, i)\downarrow 1$.

Proof. The proof is by induction on the structure of Command, proving $\mathbf{C}_{full}[\![C]\!]$ $cons_{Store \to Store}\ \mathbf{C}_{abs}[\![C]\!]$ for the logical relation $cons$ between the domains of $\mathbf{C}_{full}$ and $\mathbf{C}_{abs}$:

$v\ cons_{Storable}\ v'$ iff $v \downarrow 1\ cons_{Sec}\ v'$
$v\ cons_{Expressible}\ v'$ iff $v\ cons_{Storable}\ v'$
$t\ cons_{Sec}\ t'$ iff $t \sqsubseteq t'$
$s\ cons_{Store}\ a$ iff for all $i \in$ Identifier, $(s\, i)\ cons_{Storable}\,(a\, i))$
$p\ cons_{Poststore}\ p'$ iff $(p = \bot)$ or $p\ cons_{Store}\ p'$
$f\ cons_{D_1 \to D_2} f'$ iff for all $d \in D_{1full}$, $d' \in D_{1exec}$, $d\ cons_{D_1}\ d'$ implies $(f\,d)$ $cons_{D_2}\ (f'\,d')$ □

Proposition 2 states that the abstract semantics approximates the full semantics in its calculation of security classifications for the variables. Hence, a separate, *safe*, compile-time analysis of security classifications can be undertaken. For example, if it is critical that the security classes of some output variables be less than some security classification $t \in Sec$, then the abstract analysis will give a safe answer.

Shortly, we will study the effective implementation of the abstract semantics.

4.2. Static Variables

Next, we extend the three interpretations to include variables with fixed security classes (that is, *static variables*). Analysis of information flow into static variables is

$s \in Store = (\text{Identifier} \to Storable)$
$p \in Poststore = Store_{\perp}^{\top}$ (*Note:* $\top$ denotes "error".)
$Storable = Static + Dynamic$ (*Note:* the " + " denotes disjoint union.)
$Static = Dynamic = Expressible$
$Expressible = Sec \times Value$
$t \in Sec =$ as in Fig. 5
$v \in Value =$ as in Fig. 5
$update\ i\ f = \lambda s.\text{cases } (s\,i) \text{ of}$
$\quad \text{is}Static(t,v) \to (((f s)\downarrow 1 \sqsubseteq t \to [i \mapsto \text{in}Static(t,(f s)\downarrow 2)]s \,\|\, \top)$
$\quad \| \text{ is}Dynamic(t,v) \to [i \mapsto \text{in}Dynamic(f s)]s \text{ end}$
$c_2\ comp\ c_1 =$ as in Fig. 5
$cond\ b\,c_1\,c_2 = \lambda s.\text{let } (t,v) = (b\,s) \text{ in let } s' = newcontext\ S\ t\,s \text{ in } v \to (c_1\,s') \,\|\, (c_2\,s')$
$\quad$ where $newcontext\ \{\,\}\ t\,s = s$
$\qquad newcontext\ \{i\} \cup S\ t\ s = \text{cases } (s\,i) \text{ of}$
$\qquad\quad \text{is}Static(t',v') \to (t \sqsubseteq t' \to newcontext\ S\ t\ s \,\|\, \top)$
$\qquad\quad \| \text{ is}Dynamic(t',v') \to newcontext\ S\,t\ [i \mapsto \text{in}Dynamic(t \sqcup t',v')]s \text{ end}$
$skip =$ as in Fig. 5
$put\ k =$ as in Fig. 5
$access\ i = \lambda s.\text{cases } (s\,i)\downarrow 1 \text{ of is}Static(t,v) \to (t,v) \,\|\, \text{is}Dynamic(t,v) \to (t,v) \text{ end}$
$do\ op\,f\,g =$ as in Fig. 5
(*Note:* "let $s =$ " is *fully strict*: it is strict, and "let $s = \top$ *in* e" equals $\top$.)

Fig. 8. Full semantics with static variables

a crucial job of the security flow control algorithm; the potential flow of a value of security class t into a static variable of class t', where $t \not\sqsubseteq t'$, must be reported as a potential security violation.

We again use the core semantics of Fig. 4. Assume there are some global or default declarations that fix all of a program's variables to be static or dynamic. This information is placed in the program's initial store via "in*Static*" and "in*Dynamic*" type tags. The full semantics interpretation is presented in Fig. 8. The semantics is defined so that a security error in a program causes a denotation of $\top$ (read as "error"), even if the program would ultimately loop. (Since we do not recover from errors, $\top$ proves adequate [Sto77].)

There is no change to the execution semantics, since it describes the run-time values of variables, which are independent of the security classifications. Figure 9 gives the new abstract semantics. The new version of $\mathbf{C}_{abs}$ maps an input store of variables and their initial security classes to an output poststore of variables and their final security classes, *if* no violation of a static variable can occur. If there is a potential insecure assignment to a static variable, the output is $\top$. (*Note:* the $\perp$ value is added to $Poststore_{abs}$ to force it to be a pointed cpo. An easy induction proof shows that it is unused: for all C$\in$Command, $a \in Store_{abs}$, $\mathbf{C}_{abs}[\![\mathrm{C}]\!]a \neq \perp$. The intuitive reason is that the abstract interpretation of a **while**-loop uses *cond*, which joins the input store, which is non-$\perp$, to the meanings of the iterations of the loop body.)

Proposition 3. $\mathbf{C}_{abs}$ is a "conservative projection" of $\mathbf{C}_{full}$ in the sense that for all $C \in$Command, for all $s \in Store_{full}$, $\mathbf{C}_{full}[\![\mathrm{C}]\!]s = \perp$ or else $(\mathbf{C}_{full}[\![\mathrm{C}]\!]s = \top = \mathbf{C}_{abs}[\![\mathrm{C}]\!](first\ s))$ or else $(first(\mathbf{C}_{full}[\![\mathrm{C}]\!]s) \sqsubseteq \mathbf{C}_{abs}[\![\mathrm{C}]\!](first\ s))$, where $first: Store_{full} \to Store_{abs}$ is defined as

$$first = \lambda s.\lambda i.\text{cases } (s\,i) \text{ of is}Static(t,v) \to \text{in}Static(t)$$
$$\|\text{is}Dynamic(t,v) \to \text{in}Dynamic(t) \text{ end.}$$

$a \in Store = (\text{Identifier} \to Storable)$
$Poststore = Store_{\perp}^{\top}$
$Storable = Static + Dynamic$
$Static = Dynamic = Expressible$
$Expressible = Sec$
$t \in Sec =$ as in Fig. 5
$update\ i f = \lambda a.\text{cases}\ (a\, i)\ \text{of is}Static(t) \to ((f\, a) \sqsubseteq t \to a \,[\!]\, \top)$
$\quad [\!]\ \text{is}Dynamic(t) \to [i \mapsto \text{in}Dynamic(f\, a)]a\ \text{end}$
$c_2\ comp\ c_1 = \lambda a.\text{let}\ a' = (c_1\, a)\ \text{in}\ (c_2\, a')$
$cond\ S\, b\, c_1\, c_2 = \lambda a.\text{let}\ a' = newcontext\ S\ (b\, a)\ a\ \text{in}\ (c_1\, a') \sqcup (c_2\, a')$
where $newcontext\ \{\}\ t\, a = a$
$newcontext\ (\{i\} :: S)\ t\, a = \text{cases}\ (a\, i)\ \text{of}$
$\text{is}Static(t') \to (t \sqsubseteq t' \to newcontext\ S\, t\, a \,[\!]\, \top)$
$[\!]\ \text{is}Dynamic(t') \to newcontext\ S\, t\ [i \mapsto \text{in}Dynamic(t \sqcup t')]a\ \text{end}$
$skip =$ as in Fig. 6
$put\ k =$ as in Fig. 6
$access\ i = \lambda a.\text{cases}\ (a\, i)\ \text{of is}Static(t) \to t \,[\!]\, \text{is}Dynamic(t) \to t\ \text{end}$
$do\ op\, f\, g =$ as in Fig. 6
(*Note:* "let $a =$" is fully strict.)

Fig. 9. Abstract interpretation with static variables

Proof. The proof is by induction on the structure of Command, proving a logical relation *cons* between the domains of $\mathbf{C}_{full}$ and $\mathbf{C}_{abs}$ that is similar to the one in Proposition 2, except for

$x\ cons_{Storable}\, x'$ iff ($x = \text{in}Static(t, v)$ and $x' = \text{in}Static(t)$)
or ($x = \text{in}Dynamic(t, v)$ and $x' = \text{in}Dynamic(t')$ and $t\ cons_{Sec}\, t'$)
$s\ cons_{Store}\, a$ iff for all $i \in$ Identifier, $(s\, i)\ cons_{Storable}\ (a\, i)$
$p\ cons_{Poststore}\, p'$ iff $p = \perp$ or $p' = \top$ or $p\ cons_{Store}\ p'$

Note that $\top$ is an isolated element in both $Store_{full}$ and $Store_{abs}$, hence it is easy to verify that *cons* is an inclusive predicate ([Sch88], [Sto77]). □

Proposition 4. $\mathbf{C}_{exec}$ is a "liberal projection" of $\mathbf{C}_{full}$ in the sense that for all $C \in$ Command, for all $s \in Store_{full}$, $\mathbf{C}_{full}[\![C]\!]s = \perp = \mathbf{C}_{exec}[\![C]\!](second\, s)$ or else ($\mathbf{C}_{full}[\![C]\!]s \neq \top$ implies $second(\mathbf{C}_{full}[\![C]\!]s) = \mathbf{C}_{exec}[\![C]\!](second\, s)$)), where $second\colon Store_{full} \to Store_{exec}$ is defined as

$$second = \lambda s.\lambda i.\text{cases}\ (s\, i)\ \text{of is}Static(t, v) \to v \,[\!]\, \text{is}Dynamic(t, v) \to v\ \text{end}.$$

Proof. The proof is by induction on the structure of Command, proving a logical relation between the domains of $\mathbf{C}_{full}$ and $\mathbf{C}_{exec}$. The relation is the same as the one used in the proof of Proposition 2, with these exceptions:

$x\ proj_{Storable}\, x'$ iff ($x = \text{in}Static(t, v)$ and $v\ proj_{Value}\, x'$) or ($x = \text{in}Dynamic(t, v)$ and $v\ proj_{Value}\, x'$)
$s\ proj_{Store}\, s'$ iff for all $i \in$ Identifier, $(s\, i)\ proj_{Storable}\ (s'\, i)$
$p\ proj_{Poststore}\, p'$ iff $p = \top$ or $p = \perp = p'$ or $p\ proj_{Store}\, p'$ □

Proposition 4 implies that the execution semantics cannot be trusted on its own. But the following corollary is immediate from Propositions 3 and 4:

Corollary 1. $\mathbf{C}_{abs}[\![C]\!](first\, s) \neq \top$ implies ($\mathbf{C}_{full}[\![C]\!]s = \perp = \mathbf{C}_{exec}[\![C]\!](second\, s)$ or else $second(\mathbf{C}_{full}[\![C]\!]s) = \mathbf{C}_{exec}[\![C]\!](second\, s)$).

That is, the abstract semantics can be used as a compile-time check to ensure the correctness of the execution semantics with respect to the full semantics.

4.3. Distributivity of the Abstract Semantics

An important property of an abstract semantics is *distributivity*. For sup-semilattices A and B, a function $f: A \to B$ is *distributive* iff for all $a, b \in A, f(a \sqcup b) = (fa) \sqcup (fb)$. Distributivity is important in theory, because it ensures that the least fixed point (that is, iterative [ASU86]) data flow analysis method gives the same result as the "meet over all paths" data flow analysis method ([KaU77], [Nie83]). For our work, it is important in practice, because it allows us to transform the abstract semantics into an effective, iterative data flow analysis algorithm. Indeed, we will transform the abstract semantics into the iterative data flow algorithm described in Section 3. We begin with this easy-to-prove proposition:

Proposition 5. All expressions built with the operations in Fig. 6 are distributive in their *Store* arguments.

The proof of distributivity for the abstract semantics with static variables hinges upon a simple property about abstract stores:

Definition 1. For $a, b \in Store_{abs}$, a and b are *variable consistent* if for all $i \in$ Identifier, $((ai) = \text{in}Static(t)$ *iff* $(b\,i) = \text{in}Static(t))$ and $((a\,i) = \text{in}Dynamic(t')$ *iff* $(b\,i) = \text{in}Dynamic(t''))$.

That is, stores a and b are variable consistent if they have the same static variables with the same security classifications and they have the same dynamic variables.

Proposition 6. For all expressions f built from operations in Fig. 9, for all $a \in Store_{abs}$, a is variable consistent with (fa).

Proposition 7. For all expressions $f: Store \to Store$ built from operations in Fig. 9, for all $a, b \in Store$, if a and b are variable consistent, then $(fa) \sqcup (fb) = f(a \sqcup b)$.

Proof. Similar to Proposition 5. One case is

- *update if*: $(update\ if a) \sqcup (update\ if b) =$ (cases $(a\,i)$ of $\cdots$ end) $\sqcup$ (cases $(b\,i)$ of $\cdots$ end).

Due to Proposition 6, there are but two cases on the pair $((a\,i), (b\,i)$ to consider:

1. is$Dynamic(t)$, is$Dynamic(t')$: as in the proof of Proposition 5.
2. is$Static(t)$, is$Static(t)$: we get $((fa \sqsubseteq t_0 \to a \,[\!]\, \top) \sqcup ((fb) \sqsubseteq t_0 \to b \,[\!]\, \top)$, where $t_0 = (a\,i) = (b\,i)$. There are four possible outcomes of the predicates $(fa) \sqsubseteq t_0$, $(fb) \sqsubseteq t_0$:
   - true, true: then $a \sqcup b = ((fa) \sqcup (fb) \sqsubseteq t_0 \to a \sqcup b \,[\!]\, \top$. Since f is distributive, we are finished.
   - true, false: then $\top = ((fa) \sqcup (fb) \sqsubseteq t_0 \to a \sqcup b \,[\!]\, \top)$.
   - other cases: like the one just seen. □

4.4. Effective Calculation of the Abstract Semantics

For an abstract store $a_0 \in Store_{abs}$ and a program C, we wish to calculate $\mathbf{C}_{abs}[\![\mathrm{C}]\!]a_0$

effectively. This calculation is supposedly a matter of simple rewriting, but the *fix* operator in $\mathbf{C}_{abs}[\![\textbf{while}\ \mathrm{E}\ \textbf{do}\ C]\!] = fix\,F$, where $F = (\lambda f.\,cond\,\mathrm{E}[\![\mathrm{E}]\!](f\,comp\cdots)(\cdots))$, presents the possibility of an infinite rewriting sequence.

Recall that, for a functional $F{:}A \to A$, $fix\,F = \bigsqcup_{i\geqslant 0} F^i \bot_A$, where $F^i = F \circ F \circ \cdots \circ F$, F composed i times. If A has no infinitely ascending chains, there exists some $k \geqslant 0$ such that $fix\,F = \bigsqcup_{0\leqslant i\leqslant k} F^i \bot = F^k \bot$. Further, there exists a least such k, and it is the first $k \geqslant 0$ such that $F^{k+1}\bot = F^k \bot$. This fact is the heart of all iterative data flow analysis algorithms.

But for $F{:}(A\to A)\to(A\to A)$, calculating $(fix\,F)a_0$ presents some difficulties, which can be understood from these facts:

1. $(fix\,F)a_0 = \bigsqcup_{i\geqslant 0}(F^i \bot\, a_0)$.
2. The set $\{(F^i \bot\, a_0) \mid i \geqslant 0\}$ forms a chain, and when A has no infinitely ascending chains, there exists some $k \geqslant 0$ such that $F^k \bot\, a_0 = (fix\,F)a_0$.
3. But $(F^j \bot\, a_0) = (F^{j+1} \bot\, a_0)$, for some $j \geqslant 0$, does not guarantee that $(F^{j+1}\bot\, a_0) = (fix\,F)a_0$.

For example, for $\mathbf{C}_{exec}[\![\mathrm{X} := 2;\ \text{while}\ \mathrm{X} > 0\ \text{do}\ \mathrm{X} := \mathrm{X}-1]\!]s_0 = (fix\,F)s_0$, the calculations are

$$F_0 s_0 = \bot,\ F_1 s_0 = \bot,\ F_2 s_0 = \bot,\ F_3 s_0 = [X \mapsto 0]s_0 = (fix\,F)s_0$$

so we have no reliable method for checking convergence.

Fortunately, we *can* detect convergence of $(fix\,F)a_0$ for those functionals $F{:}(A\to A)\to(A\to A)$ with the format

$$F = \lambda f.\lambda a.f(h\,a) \sqcup (g\,a)$$

provided that A has no infinitely ascending chains and $g, h{:}A\to A$ are distributive. The definition of $\mathbf{C}_{abs}[\![\textbf{while}\ \mathrm{E}\ \textbf{do}\ \mathrm{C}]\!]$ based on Fig. 6 has this structure, where $g = kontext$ and $h = \mathbf{C}[\![\mathrm{C}]\!] \circ kontext$, where $kontext = (\lambda a'.\lambda i.\, i \in S \to (\mathrm{E}[\![\mathrm{E}]\!]a') \sqcup (a'\,i)\ [\!]\ (a'i))$. (The version in Fig. 9 can be similarly expressed.) Think of h as the "loop body" and g as the "termination step" of the **while**-loop. Then it is clear that F specifies an iterative analysis for which convergence is detectable.

Before we prove the above claims, we require two lemmas. First, for $h{:}A\to A$, let $h^0 = id_A$, and $h^{i+1} = h \circ h^i$.

Lemma 1. For $F{:}(A\to A)\to(A\to A)$, $F = \lambda f.\lambda a.f(h\,a) \sqcup (g\,a)$, for all $j \geqslant 0$, $F^{j+1} = \bigsqcup_{0\leqslant i\leqslant j} g \circ h^i$.

Corollary 2. $fix\,F = \bigsqcup_{i\geqslant 0} g\circ h^i$.

Lemma 2. If $h{:}A\to A$ is distributive, then if there exists a $k \geqslant 0$ such that $\bigsqcup_{0\leqslant i\leqslant k+1}(h^i a) = \bigsqcup_{0\leqslant i\leqslant k}(h^i a)$, then for all $m \geqslant k$, $\bigsqcup_{0\leqslant i\leqslant k}(h^i a) = \bigsqcup_{0\leqslant i\leqslant m}(h^i a)$.

Proof. Since the antecedent is equivalent to $(h^{k+1} a) \sqsubseteq \bigsqcup_{0\leqslant i\leqslant k}(h^i a)$, we assume the latter and show for all $m \geqslant k$, $(h^m a) \sqsubseteq \bigsqcup_{0\leqslant i\leqslant k}(h^i a)$, which is equivalent to the succedent. The proof is by induction on j, where $m = k + j$. □

Theorem 1. If $F = \lambda f.\lambda a.(f(h\,a)) \sqcup (g\,a)$, where $g, h{:}A\to A$ are distributive and A is a pointed, sup-semilattice with no infinitely ascending chains, then for all $a \in A$, $(fix\,F)a = g(\bigsqcup_{0\leqslant i\leqslant k}(h^i a))$ where k is any natural number such that $\bigsqcup_{0\leqslant i\leqslant k}(h^i a) = \bigsqcup_{0\leqslant i\leqslant k+1}(h^i a)$.

Proof. Since A has no infinitely ascending chains, there exists an $m \geqslant 0$ such that $(fix\,F)a = \bigsqcup_{0\leqslant i\leqslant m+1}(F^i \bot\, a) = (F^{m+1}\bot\, a)$. By Lemma 1, $(F^{m+1}\bot\, a) = (\bigsqcup_{1\leqslant i\leqslant m} g\circ h^i)a = (\bigsqcup_{0\leqslant i\leqslant m}(g(h^i a)) = g(\bigsqcup_{0\leqslant i\leqslant m}(h^i a))$, by distributivity of g. Next, we build the chain $\{a,(a \sqcup (h\,a)), (a \sqcup (h\,a) \sqcup (h^2 a)),\cdots\}$; there exists some $k \geqslant 0$ such that $\bigsqcup_{0\leqslant i\leqslant k}(h^i a) = \bigsqcup_{0\leqslant i\leqslant k+1}(h^i a)$, that is, $(h_{k+1}\,a) \sqsubseteq \bigsqcup_{0\leqslant i\leqslant k}(h^i a)$. If $k \leqslant m$, then by Lemma 2, $\bigsqcup_{0\leqslant i\leqslant m}(h^i a) = \bigsqcup_{0\leqslant i\leqslant k}(h^i a)$. If $m < k$, then $g(\bigsqcup_{0\leqslant i\leqslant m}(h^i a)) = F^{m+1} \perp a = F^{k+1} \perp a = (\bigsqcup_{0\leqslant i\leqslant k} g \circ h^i)a = \bigsqcup_{0\leqslant i\leqslant k}(g\,(h^i a)) = g(\bigsqcup_{0\leqslant i\leqslant k}(h^i a))$. □

Theorem 1 tells us that we effectively calculate $(fix\,F)a$ as follows:

1. For $k = 0, 1, \cdots$, calculate $(h^{k+1} a) = h(h^k a)$ and $\bigsqcup_{0\leqslant i\leqslant k+1}(h^i a) = (h^{k+1} a) \sqcup (\bigsqcup_{0\leqslant i\leqslant k} h^i a)$ until some $j \geqslant 0$ is found such that $\bigsqcup_{0\leqslant i\leqslant j}(h^i a) = \bigsqcup_{0\leqslant i\leqslant j+1}(h^i a)$. Call this value a_0.
2. Calculate $(g\,a_0)$.

This is the iterative data flow analysis algorithm found in [ASU86] and used in Section 3: h represents the "loop body", and g represents the "termination step". In [NiN92], it is shown that the value of j is linear with respect to the length of the program analysed.

4.5. Parameters

So far, we have defined, proved safe, and implemented abstract interpretations for completed programs. We now study programs parametrized on unknowns. This prepares us for the introduction of procedures and separately compiled modules.

We begin with the abstract semantics based on Fig. 6, that is, with dynamic variables only. Say that the input abstract store used by a program maps some identifier I to an unknown value, α. The unknown α is a "placeholder", as in elementary algebra. Since the substitution and simplification laws of the semantics are algebraic, we can proceed as described in the earlier sections to calculate the output security information for the program, which will be a poststore that maps identifiers to symbolic expressions (*polynomials* [Gra79]) containing α. But we must ensure that we can still effectively detect convergence as described by Theorem 1. Our plan is to show that the polynomial $\mathbf{C}_{abs}[\![\mathrm{C}]\!]a_0$, where $a_0 \in Store_{abs}$ contains an unknown α, can always be simplified into the *canonical form*: $[i \mapsto e_i]_{i\in\mathrm{Identifier}}$ where e_i has the form v_i or the form $v_i \sqcup \alpha$, where $v_i \in Sec$. Hereafter, we abbreviate the canonical form to $[i \mapsto v_i[\sqcup \alpha]]_{i\in\mathrm{Identifier}}$. (The italicised brackets denote optional information.) An example poststore polynomial is $[\mathrm{A} \mapsto \mathit{classified} \sqcup \alpha][\mathrm{B} \mapsto \mathit{top\text{-}secret}][\mathrm{C} \mapsto \perp \sqcup \alpha]$.

Proposition 8. If the polynomial $a \in Store$ has canonical form, that is, $[i \mapsto v_i[\sqcup \alpha]]_{i\in\mathrm{Identifier}}$, then for all $\mathrm{C} \in$ Command, the polynomial $\mathbf{C}_{abs}[\![\mathrm{C}]\!]a$ can be rewritten into canonical form.

Proof. We first note, for all $\mathrm{E} \in$ Expression, that $\mathbf{E}_{abs}[\![\mathrm{E}]\!]a$ can be rewritten into the form $v_i[\sqcup \alpha]$, when a has canonical form. The proof is by induction on the structure of E. The main result is proved by induction on the structure of Command. The interesting case is

- **while** E **do** C: From Theorem 1, we have, for some $k \geqslant 0$, $\mathbf{C}_{abs}[\![\textbf{while}\ \mathrm{E}\ \textbf{do}\ C]\!]a = kontext(\bigsqcup_{0\leqslant i\leqslant k} e_i)$, where $e_0 = a$ and $e_{i+1} = \mathbf{C}[\![\mathrm{C}]\!](kontext\; e_i)$, where $kontext$ is defined in Section 4.4. By the induction hypothesis on C and the result for $\mathbf{E}_{abs}[\![\mathrm{E}]\!]a$, we have that all e_i have form $[i \mapsto v_i[\sqcup \alpha]]_{i\in\mathrm{Identifier}}$. Hence so does $\bigsqcup_{0\leqslant i\leqslant k} e_i$.

  Can we detect convergence, i.e., can we effectively determine k? If we cannot, there must be a sequence of nonconverging $Store$-typed polynomials, implying there is a sequence of nonconverging polynomials $x_0, x_1, \cdots, x_i, \cdots$

representing values in *Sec* (since the domain Identifier is finite). Each x_j has form $v_j[\sqcup \alpha]$, for $v_j \in Sec$. Due to Theorem 1, $x_{j+1} = \bigsqcup_{0 \leqslant i \leqslant j+1}(e_i x) = ((\bigsqcup_{0 \leqslant i \leqslant j} e_i) \sqcup e_{j+1})x = (\bigsqcup_{0 \leqslant i \leqslant j}(e_i x)) \sqcup (e_{j+1} x)$. Hence, if α is in x_j, it must also appear in x_m, for all $m \geqslant j$. That is, the sequence of polynomials forms a "chain" in an appropriate partial ordering. But such a chain must converge, since *Sec* has the finite chain property. □

Proposition 8 complements and supersedes the results of the previous section. Not only do we know that convergence – even in the presence of an unknown, α – must arise, we also know that an algebraic-style rewriting into canonical form detects it.

We now prove a similar result for the abstract semantics with static variables, i.e., for Fig. 9. The static variables introduce "inequations" on the poststore of the form $e \sqsubseteq t$, where $t \in Sec$ and e is a symbolic expression. Let V be the static variables in the program. We will show that poststore polynomials have the canonical form

$$\{v_x[\sqcup \alpha] \sqsubseteq t_x\}_{x \in V'} \rightarrow ([x \mapsto \mathrm{in}Static(t_x)]_{x \in V}$$
$$[i \mapsto \mathrm{in}Dynamic(v_i[\sqcup \alpha])]_{i \in \mathrm{Identifier}-V}) \,[\!]\, \top$$

where $V' \subseteq V$ and $v_i, v_x, t_x \in Sec$. Call the above form a *guarded form*. Likewise, call

$$[x \mapsto \mathrm{in}Static(t_x)]_{x \in V}[i \mapsto \mathrm{in}Dynamic(v_i[\sqcup \alpha])]_{i \in \mathrm{Identifier}-V}$$

an *unguarded form*. Here is an example of a poststore polynomial in guarded form for a program with static variables X of class *topsecret*, Y of class *secret*, and a dynamic variable Z:

$$\{secret \sqcup \alpha \sqsubseteq topsecret,\ unclassified \sqsubseteq secret\} \rightarrow ([\mathrm{X} \mapsto \mathrm{in}Static(topsecret)]$$
$$[\mathrm{Y} \mapsto \mathrm{in}Static(secret)]\,[\mathrm{Z} \mapsto \mathrm{in}Dynamic(secret)])\,[\!]\, \top$$

The guarded form tells us, if the two inequations hold true, the output poststore will have a *topsecret* value for X and *secret* values for Y and Z. The algorithm in Section 3 constructs inequations like the two shown here.

Lemma 3. If expressions p_1 and p_2 have guarded form, that is, $p_i = C_i \rightarrow a_i \,[\!]\, \top$, $i \in 1, 2$, then $p_1 \sqcup p_2 = C_1 \cup C_2 \rightarrow a_1 \sqcup a_2 \,[\!]\, \top$.

Now we show that the analysis of a command with an input store in unguarded form must produce an output poststore in guarded form:

Theorem 2. If polynomial $a \in Store_{abs}$ has unguarded form, then $\mathbf{C}_{abs}[\![\mathrm{C}]\!]a$ can be rewritten into guarded form, for all $C \in$ Command.

Proof. Appendix 1. □

Theorem 2 generalises to any finite number of unknown values.

In Section 3.1, we saw that the result of analysing a procedure was written as a summary data structure (cf. Fig. 2). A poststore polynomial in guarded form encodes such a data structure. For poststore polynomial:

$$\{v_x \sqsubseteq t_x\}_{x \in V} \rightarrow [x \mapsto \mathrm{in}Static(t_x)]_{x \in V}[i \mapsto \mathrm{in}Dynamic(v_i)]_{i \in \mathrm{Identifier}-V} \,[\!]\, \top$$

the corresponding summary data structure is

$v_1 \sqsubseteq t_1$ (for x_i)
...
$v_n \sqsubseteq t_n$ (for x_n) for $x_1, \cdots, x_n \in V$
$i_1 = v_1$
...
$i_m = v_m$ for $i_1, \cdots, i_m \in$ Identifier $- V$

4.6. Procedures

A program that uses a store with k unknown values can be thought of as a procedure parametrised on k parameters. In this section, we formalise this idea and derive the analysis of a parametrised procedure that invokes other external procedures.

The syntax of procedures is

$$
\begin{aligned}
&\mathrm{P} ::= \textbf{proc } \mathrm{F}(\textbf{in } \mathrm{D}_1;\ \textbf{out } \mathrm{D}_2) = \mathrm{D}^*;\ \mathrm{C}\\
&\mathrm{D} ::= \mathrm{I}\,\textbf{:static } \mathrm{T} \mid \mathrm{I}\,\textbf{:dynamic}\\
&\mathrm{T} ::= \mathit{Sec}\\
&\mathrm{C} ::= \cdots \mid \textbf{call } \mathrm{F}(\textbf{in } \mathrm{E},\ \textbf{out } \mathrm{I})
\end{aligned}
$$

That is, a procedure has an input parameter, an output parameter and a list of local declarations. (We limit the parameters to two for simplicity. We also assume that the input and output formal parameters have distinct names.) For the moment, we will not allow a procedure to reference globally declared variables. Hence, the procedure is merely a function of its input parameter; the output from a procedure is the value bound to its output parameter. Procedures are declared globally and can be referenced by other procedures.

Figure 10 gives the core semantics of procedures, and Fig. 11 gives the standard, abstract and execution interpretations. The semantics of procedure declaration goes as follows: a procedure is a mapping from its input parameter, an expressible

P: Procedure $\to$ *Expressible* $\to$ *Proc-result*

$\mathbf{P}[\![\textbf{proc } \mathrm{F}(\textbf{in } \mathrm{D}_1;\textbf{out } \mathrm{D}_2) = \mathrm{D}^*;\mathrm{C}]\!] = \lambda\alpha \in \mathit{Expressible}.$
  $((\mathit{return}\ \mathbf{V}[\![\mathrm{D}_2]\!]) \circ \mathbf{C}[\![\mathrm{C}]\!])$
    $\mathit{comp}\ ((\mathit{update}\ \mathbf{V}[\![\mathrm{D}_1]\!]\ (\mathit{const}\,\alpha)) \circ \mathbf{D}[\![\mathrm{D}^*]\!] \circ \mathbf{D}[\![\mathrm{D}_2]\!] \circ \mathbf{D}[\![\mathrm{D}_1]\!]))\ \mathit{empty}$

(*Note:* "$\circ$" is ordinary function composition.)

D: Declaration $\to$ *Store* $\to$ *Store*

$\mathbf{D}[\![\mathrm{I}\textbf{:static } \mathrm{T}]\!] = \mathit{allocate\text{-}static}\ \mathrm{I}\ \mathbf{T}[\![\mathrm{T}]\!]$

$\mathbf{D}[\![\mathrm{I}\textbf{:dynamic}]\!] = \mathit{allocate\text{-}dynamic}\ \mathrm{I}\ \bot$ (*Note:* $\bot \in \mathit{Sec}$ is the minimum security classification.)

$\mathbf{C}[\![\textbf{call } \mathrm{F}(\textbf{in } \mathrm{E}, \textbf{out } \mathrm{I})]\!] = \mathit{call}\,f\ \mathbf{E}[\![\mathrm{E}]\!]\ \mathrm{I}$
  where f: *Expressible* $\to$ *Proc-result* is the denotation of F

$\mathbf{I}[\![\textbf{call } \mathrm{F}(\textbf{in } \mathrm{E}, \textbf{out } \mathrm{I})]\!] = \{\mathrm{I}\}$

T: Type $\to$ *Sec*

$\mathbf{T}[\![\mathrm{T}]\!] = \mathrm{T}$

V: Declaration $\to$ Identifier

$\mathbf{V}[\![\mathrm{I}\textbf{:static } \mathrm{T}]\!] = \mathrm{I}$

$\mathbf{V}[\![\mathrm{I}\ \textbf{dynamic}]\!] = \mathrm{I}$

where
  empty: *Store*
  const: *Expressible* $\to$ *Store* $\to$ *Expressible*
  allocate-static: *Identifier* $\to$ *Sec* $\to$ *Store* $\to$ *Store*
  allocate-dynamic: Identifier $\to$ *Sec* $\to$ *Store* $\to$ *Store*
  call: (*Expressible* $\to$ *Proc-result*) $\to$ (*Store* $\to$ *Expressible*) $\to$ Identifier $\to$ *Store* $\to$ *Poststore*
  return: Identifier $\to$ *Poststore* $\to$ *Proc-result*
are defined in the interpretations.

Fig. 10. Core semantics of procedures

Full interpretation:
Proc-result $= Expressible_{\perp}^{\top}$
empty = [] (that is, a mapping over an empty set of identifiers)
const v $= \lambda a.v$
allocate-static i v $= \lambda a.\ [i \mapsto \text{in}Static(v, ?)]a$
allocate-dynamic i v $= \lambda a.\ [i \mapsto \text{in}Dynamic(v, ?)]a$
 where *?* is some initial value
call f g i $= \lambda a.update\ i\ (f \circ g)\ a$
 Note: $(f \circ g)a = \perp$ implies $update\ i\ (f \circ g)\ a = \perp$
 $(f \circ g)a = \top$ implies $update\ i\ (f \circ g)\ a = \top$
return i $= \lambda p.$let $a = p$ in $access\ i\ a$

Execution interpretation:
Proc-result $= Expressible_{\perp}$
 all operations coded as above, except for
allocate-static i v $= \lambda a \perp\ [i \mapsto ?]a$
allocate-dynamic i v $= \lambda a.\ [i \mapsto ?]a$

Abstract interpretation:
Proc-result $= Expressible_{\perp}^{\top}$
 all operations coded as above, except for
allocate-static i v $= \lambda a.\ [i \mapsto \text{in}Static(v)]a$
allocate-dynamic i v $= \lambda a.\ [i \mapsto \text{in}Dynamic(v)]a$
(*Note:* "let $a =$" is fully strict for $\perp$ and $\top$ arguments.)

Fig. 11. Interpretation of procedures

value, to its result. When called, the procedure starts with a fresh (*empty*) store, which immediately gets cells for the input parameter D_1, output parameter D_2, and local declarations D*. The *update* operation binds actual parameter α to D_1; then body C executes. On termination, the value in cell D_2 is returned as the result.

The semantics of procedure call matches the above: the *call* operation uses the store to calculate the value of actual parameter E and invokes the denotation of the procedure with the actual parameter. The result returned by the called procedure is bound to the output variable I.

One fundamental difficulty arises: we cannot effectively check the convergence of a while-loop in the presence of a call to an unknown external procedure. For example, for C_{abs}[**while true do call** F(**in** X, **out** X)]a_0, say that variable X is bound to v_0 in a_0 and say that the denotation of external procedure F is represented by the unknown, *f*. Then, the sequence of polynomials that denote X's value in the loop are

$$v_0$$
$$v_0 \sqcup (f v_0)$$
$$v_0 \sqcup (f v_0) \sqcup (f(v_0 \sqcup (f v_0))) = v_0 \sqcup (f v_0) \sqcup (f(f v_0)), \text{ since } f \text{ is distributive}$$
$$v_0 \sqcup (f v_0) \sqcup (f(f v_0)) \sqcup (f(f(f v_0)))$$
$$\ldots$$

and the presence of polynomials does not stabilise.

Of course, we can use some artificially high upper bound of iteration (say, the number of identifiers in the procedure times the height of the semilattice *Sec*) and quit generating expressions at that point, since the underlying semantic values must converge by then, but this is impractical. We solve the problem with an alternative abstract semantics for procedure call that approximates the semantics in Figs 10 and 11.

First, let each call of a procedure F be uniquely indexed. We write F_i for a call. For each occurrence of an F_i, we allocate a (dynamic) "dummy variable" $f_i.in$ in the store and make the semantics of procedure call:

$$\mathbf{C}[\![\mathbf{call}\ F_i(\mathbf{in}\ E,\ \mathbf{out}\ I)]\!] = (call\ f\ (access\ f_i.in)\ I)\ comp\ (update\ f_i.in\ \mathbf{E}[\![E]\!])$$

That is, the value of the actual parameter E is copied into the dummy variable $f_i.in$, whose value is immediately given to the called procedure f.

Call the semantics with the dummy variables C^+, and call a program C with annotated procedure calls C^+. It is easy to prove:

Proposition 9. For all $C \in$ Command, $a \in Store$, $\mathbf{C}[\![C]\!]a = \top = \mathbf{C}^+[\![C^+]\!]a^+$ or else $\mathbf{C}[\![C]\!]a = (\mathbf{C}^+[\![C^+]\!]a^+|_{\text{Identifier}})$, where a^+ is a with cells for the dummy identifiers.

(Recall that $f|_{D'}$ represents function $f: D \to E$ restricted to domain $D' \sqsubseteq D$.) Hereon, let $\mathbf{C}^+_{abs}$ represent the abstract interpretation of C with annotated procedure calls and dummy variables.

We now define an abstract interpretation that uses a family of unknowns (*not* dummy variables!), $f_i.\mu$, one for each procedure call F_i in a program. Let $\mathbf{C}'_{abs}$ be the abstract semantics $\mathbf{C}^+_{abs}$ but with the following semantics of annotated procedure call:

$$\mathbf{C}'_{abs}[\![\mathbf{call}\ F_i(\mathbf{in}\ E,\ \mathbf{out}\ I)]\!] = (update\ I\ (const\ f_i.\mu))\ comp\ (update\ f_i.in\ (\mathbf{E}[\![E]\!] \sqcup (access\ f_i.in)))$$

The intuition is, rather than reason about external procedure F_i, we use the unknown $f_i.\mu$ to stand for its output. The semantics of procedure call now states: variable $f_i.in$ remembers the value of actual parameter E *plus* the values of *all* the actual parameters from previous calls to F_i. (At link-time, this information will be used to calculate an output from F_i.) Next, the unknown $f_i.\mu$ is bound to the output variable I. (At link-time, $f_i.\mu$ will be instantiated to a value representing the outputs from all the calls to F_i.) This strategy matches the one used in Section 3; a call F_i(**in** A, **out** B) causes the algorithm to generate variables $F_i.A$ and $F_i.B$, which hold the security classifications of the input and output parameters to the call of F_i, respectively. Of course, $F_i.A$ is just $f_i.in$, and $F_i.B$ is just $f_i.\mu$.

The unknown $f_i.\mu$ is introduced so that we can manipulate expressions with just first order unknowns: sequences of the form $v_0, v_0 \sqcup f(v_0), \cdots$, noted earlier, never arise, and we can effectively detect convergence. Say that $a \in Store$ is an expression with no unknowns, so it denotes a unique value in *Store*. Then, $\mathbf{C}^+_{abs}[\![C]\!]a$ denotes a unique value in *Poststore*. In contrast, $\mathbf{C}'_{abs}[\![C]\!]a$ is a poststore polynomial, containing occurrences of the $f_i.\mu$ unknowns. We will define a mapping *recover*, which maps a store polynomial to a unique store value, and we will prove the following theorem:

Theorem 3. For all $C \in$ Command, $a \in Store$, $\mathbf{C}^+_{abs}[\![C]\!]a \sqsubseteq recover(\mathbf{C}'_{abs}[\![C]\!]a)$.

The job of *recover* is to replace the $f_i.\mu$ unknowns by their "true" values. The definition is

$$recover\ a' = [f((calc\ a') \downarrow i)/f_i.\mu]_{i \in I}\, a'.$$

where

$$calc\ a' = fix(\lambda(\tau_i)_{i \in I}.(access f_i.in\ [(f\tau_i)/f_i.\mu]_{i \in I}\, a')_{i \in I})$$

(Note: $f: Expressible \to Proc\text{-}result$ is the denotation of F_i.) This deserves explanation. Our intuitions tell us that *recover* should replace each $f_i.\mu$ unknown by $f(access\, f_i.in\ a')$, since f denotes F_i, and $f_i.in$ holds the argument for f. But $(access\ f_i.in\ a'))$ is itself a polynomial and may well contain occurrences of very $f_i.\mu$ that we are trying to replace! The situation must be resolved by a fixed point calculation, so *calc* a' computes the values of the $f_i.ins$.

Theorem 3 follows from the proof of the following claim: for all C ∈ Command,

$$C^{+}_{abs}[\![C]\!]\ rec_{Store \to Poststore}\ C'_{abs}[\![C]\!]$$

where logical relation *rec* is defined to be

$t\ rec_{Sec}\ t'$ iff $t \sqsubseteq t'$.
$a\ rec_{Store}\ a'$ iff $a \sqsubseteq (recover\ a')$
$p\ rec_{Poststore}\ p'$ iff $p\ rec_{Store}\ p'$
$f\ rec_{D_1 \to D_2}\ f'$ iff for all $a \in D_1^{+}$, $a' \in D_1'$, $a\ rec_{D_1}\ a'$ implies $(f a)\ rec_{D_2}\ (f' a')$

The proof of Theorem 3 is given in full in Appendix 2.

Since the $f_i.\mu$ unknowns are no different from the α unknowns used to represent input parameters, we can perform an effective, convergent analysis of a parametrised procedure that calls external procedures and can obtain the usual output poststore polynomial. The values of the $f_i.\mu$ unknowns are resolved when the link-time algorithm is applied.

4.7 Linking Analysed Procedures

We now consider how to combine the analyses of the independently analysed procedures into an analysis of a complete system.

When we analyse a procedure independently, the analyses produces a polynomial of form $(\lambda\alpha \in Expressible.return\ i\ a')$, where a' has the guarded form: $(C_\alpha \to s'_\alpha \,[\!]\, \top)$. (See Theorem 2 and its postscript: C_α encodes the inequations for static variables, and s'_α encodes the value equations.) Both C_α and s'_α may contain occurrences of unknowns $f_i.\mu$s and α. These unknowns are instantiated when the procedure is linked to the procedures that it calls and calls it, respectively.

Here is an example. Say that we link an analysed procedure G to an external procedure F. Let F: *Expressible* → *Proc-result* represent F's denotation, let $a' = (C_\alpha \to s'_\alpha \,[\!]\, \top)$, and let G's polynomial have form: $(\lambda\alpha.return\ Z\ a')$. We link G to F by calculating $(\lambda\alpha.return\ Z\ recover\ a')$, where occurrences of f in the definition of *recover* are replaced by F. The result is a polynomial having as its only unknown, α.

Consider the form of $recover\ a' = [F(calc\ a' \downarrow i)/f_i.\mu]_{i \in I}\, a'$. First, the information in $(calc\ a')$ can be expressed as a set of equations:

$$\{\tau_i = (access\ f_i.in\ a')\}_{i \in I} \cup \{f_i.\mu = F(\tau_i)\}_{i \in I}$$

The equations are solved in the usual way; the solution must converge, even in the presence of the unknown, α, by Theorem 1. This gives us

$$\{\tau_i = u_i\}_{i \in I} \cup \{f_i.\mu = v_i\}_{i \in I}$$

for some values u_i and v_i. So, $recover\ a' = [v_i/f_i.\mu]_{i \in I}\, a'$. Since a' has guarded form, so does $(recover\ a') = ([v_i/f_i.\mu]_{i \in I}\, C_\alpha \to [v_i/f_i.\mu]_{i \in I}\, s'_\alpha \,[\!]\, \top)$. Thus, the linkage of G to the called procedure F is

$$\lambda\alpha.return\ Z\ ([v_i/f_i.\mu]_{i \in I}\, C_\alpha \to (([v_i/f_i.\mu]_{i \in I}\, s'_\alpha)\ Z) \,[\!]\, \top)$$

which makes clear that the constraints $[v_i/f_i.\mu]_{i\in I}\, C\alpha$ must be validated whenever G is called.

But what of the constraints for the called procedure F? The above calculation of the values for the τ_i's and $f_i.\mu$'s glossed over their presence. We know that F has the form

$$F = (\lambda\alpha'.return\ \mathrm{Y}\ (C'_{\alpha'} \to s''_{\alpha'} \,[\!]\, \top)).$$

Stated more precisely, the equation for the τ_i's and $f_i.\mu$'s have the form

$$\{\tau_i = (C_\alpha \to (s'_\alpha f_i.in) \,[\!]\, \top)\}_{i\in I} \cup \{f_i.\mu = (\lambda\alpha'.return\ \mathrm{Y}\ (C'_{\alpha'} \to s''_{\alpha'} \,[\!]\, \top))\tau_i\}_{i\in I}$$

that is, for the $f_i.\mu$'s: $\{f_i.\mu = (C'_{\tau_i} \to (s''_{\tau_i}\mathrm{Y}) \,[\!]\, \top)\}_{i\in I}$. Although a convergent solution for the equation set exists, manipulation of the constraints sets is tedious. There is a simpler approach; the above equation set has the same solutions as

$$\begin{aligned}&\text{let } eqns = fix(\{\tau_i = (s'_\alpha f_i.in)\}_{i\in I} \cup \{f_i.\mu = (s''_{\tau_i}\mathrm{Y})\}_{i\in I})\\ &\quad \text{in } \{C_\alpha\} \cup \{C'_{\tau_i}\}_{i\in I} \to eqns \,[\!]\, \top\end{aligned}$$

The reason is, if some C_α or C'_{τ_i} is false at some stage k of the least fixed point calculation, then since the constraints have form $v_i[\ \sqcup\ \alpha] \sqsubseteq t$, then the least fixed point solution will make the constraints false, too. The converse holds due to the finite chain property for *Sec*. (Note that all operations – including substitution and tupling – are strict on $\top$.)

The linking method described in Section 3 uses this latter method: the equations are solved first; the constraints are checked second. One difference is that the algorithm in Section 3 joints together all calls to a procedure F, giving a safe but less precise analysis than the one described here.

4.8. Recursive Procedures

A procedure that invokes itself can be analysed just like any other: the recursion is resolved at link-time, when the analysed procedure is linked to itself. A family of procedures that mutually invoke one another are handled similarly, where we require that the family of procedures be linked as a group.

Here is an example; say that procedure F invokes itself. Let a'_α represent the analysed body of F, having the form $(C_\alpha \to s'_\alpha \,[\!]\, \top)$. If we use the equational representation of F's linked definition, we have

$$\begin{aligned}&f = \lambda\alpha.f.\mu,\\ &\quad \text{where: } f.\mu = access\ \mathrm{I}\ a'_\alpha\\ &\quad \{\tau_i = access\ f_i.in\ a'_\alpha\}_{i\in I}\\ &\quad \{f_i.\mu = f(\tau_i)\}_{i\in I}\end{aligned}$$

Notice that f is both the name of the denotation of F and the name of the recursive (external) procedure.

The recursive reference to f can be resolved with the usual least fixed point calculation: $\bigsqcup_{i\geq 0} F^i(\lambda\alpha.\bot)$, where $F = (\lambda f.\lambda\alpha.f.\mu)$, and convergence is guaranteed by Theorem 3, but this is impractical. In Appendix 3, we derive a safe, but less precise, first-order analysis – the analysis in Section 3.

$\mathbf{P}$: Procedure $\rightarrow$ *Expressible* $\rightarrow$ *Store* $\rightarrow$ *Proc-result*
$\mathbf{P}[\![\textbf{proc } \mathrm{F}(\textbf{in } \mathrm{D}_1; \textbf{out } \mathrm{D}_2) = \mathrm{D}^*; \mathrm{C}]\!] = \lambda\alpha \in \mathit{Expressible}.\lambda s \in \mathit{Store}.$
$((\mathit{return}'\ \mathrm{V}[\![\mathrm{D}_2]\!]\ (\mathit{size\text{-}of}\ s)) \circ \mathrm{C}[\![\mathrm{C}]\!])$
$\mathit{comp}\ ((\mathit{update}\ \mathrm{V}[\![\mathrm{D}_1]\!]\ (\mathit{const}\ \alpha)) \circ \mathrm{D}[\![\mathrm{D}^*]\!] \circ \mathrm{D}[\![\mathrm{D}_2]\!] \circ \mathrm{D}[\![\mathrm{D}_1]\!])\ s$
where $(\mathit{size\text{-}of}\ s) \in \mathit{Store\text{-}size}$ denotes the number of cells in store s
$\mathrm{C}[\![\textbf{call } \mathrm{F}(\textbf{in } \mathrm{E}, \textbf{out } \mathrm{I})]\!] = \mathit{call}'\ f\ \mathrm{E}[\![\mathrm{E}]\!]\ \mathrm{I}$
where f: *Expressible* $\rightarrow$ *Store* $\rightarrow$ *Proc-result* is the denotation of F
$\mathrm{I}[\![\textbf{call } \mathrm{F}(\textbf{in } \mathrm{E}, \textbf{out } \mathrm{I})]\!] = \{\mathrm{I}\} \cup (\mathit{global\text{-}variables\text{-}used\text{-}by}\ \mathrm{F})$
where:
call': $(\mathit{Expressible} \rightarrow \mathit{Store} \rightarrow \mathit{Proc\text{-}result}) \rightarrow (\mathit{Store} \rightarrow \mathit{Expressible}) \rightarrow$ Identifier $\rightarrow \mathit{Store} \rightarrow \mathit{Store}$
return': Identifier $\rightarrow \mathit{Store\text{-}size} \rightarrow (\mathit{store} \rightarrow \mathit{Poststore}) \rightarrow \mathit{Proc\text{-}result}$

Full interpretation:
$\mathit{Proc\text{-}result} = (\mathit{Expressible} \times \mathit{Store})^{\top}_{\bot}$
$\mathit{call}'\ f\ g\ i = \lambda s.\mathrm{let}\ (v, s') = f(g\ s)s\ \mathrm{in}\ \mathit{update}\ i\ (\mathit{const}\ v)\ s'$
$\mathit{return}'\ i\ m = \lambda p.\mathrm{let}\ s = p\ \mathrm{in}\ (\mathit{access}\ i\ s, \mathit{pop\text{-}to}\ m\ s)$
where $\mathit{pop\text{-}to}$: *Store-size* $\rightarrow$ *Store* $\rightarrow$ *Store*
$(\mathit{pop\text{-}to}\ m\ s)$ outputs store s truncated to m cells

Execution interpretation:
$\mathit{Proc\text{-}result} = (\mathit{Expressible} \times \mathit{Store})_{\bot}$
operations as above

Abstract interpretation:
$\mathit{Proc\text{-}result} = (\mathit{Expressible} \times \mathit{Store})^{\top}_{\bot}$
operations as above

Fig. 12. Global variables and procedure calls

4.9. Global Variables

So far, we have assumed that all variables used by procedures are local. Hence, procedures are "pure functions" from their input parameters to their output parameters; implicit interprocedural flow, described in Section 3, had no effect.

This changes when we add global variables to the language. Global variables exist between invocations of procedures; they are crucial to module- and object-based systems; and they model input–output files. We limit the complications caused by global variables by requiring that they be declared with static security classes. This permits independent analysis of procedures that share global variables and allows the analysis to extend to concurrent systems.

We assume that global variables are predeclared in the store. The semantics of procedure call changes in that a called procedure receives as its input an actual parameter and the store (containing the global variables), and the procedure produces as its output the value of its output parameter and the updated store (containing the global variables). The new core semantics and its interpretations are shown in Fig. 12. When a procedure is called, it augments the store it is given with cells for its local declarations. The cells for local declarations are "popped" from the store on procedure exit. (Inequations for static variables – local or global – are not "popped".) The congruences of the abstract and execution semantics to the full semantics are straightforward to prove.

The key clause in Fig. 12 is the one for **I[callF(in E, out I)]**: all the global variables used by the called procedure F (and the procedures that F calls) must be known to compute the correct interprocedural implicit flows. But this definition is impractical for separate analysis of individual procedures. An obvious implementational solution is: when analysing a procedure P, if P calls Q, generate an equation to remember the implicit flow value that affects the global variables used by Q. The equation has form: $Q.implicit = \cdots$. When procedure Q is linked to P, the value $Q.implicit$ is joined to the inequations for each of the global variables used by Q. Since all global variables must have static security classes, it is straightforward to show that the implementation calculates the same inequations as does the abstract interpretation. The algorithm in Section 3 uses the name *implicit* to denote the places in Q's inequations where the value of $Q.implicit$ should be inserted, and the link-time algorithm sets $implicit = Q.implicit$.

5. Conclusion

The previous development is complete for a sequential programming language, but it does not consider concurrency and system failures. Informal reasoning [Miz87] suggests that the compile-time and link-time algorithms can also verify concurrent systems, but proofs have not been completed. System failures are troublesome, and further work is needed to adapt the algorithms.

Acknowledgements

This research was partially supported by NSF Grant CCR-8822378.

Anindya Banerjee suggested a significant improvement to the core semantics and the proof of Proposition 2, and the referees' comments helped many aspects of the work. Flemming Nielson's papers and correspondence have proved invaluable.

References

[ASU86] Aho, A., Sethi, R. and Ullman, J.: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[AnR80] Andrews, G. R. and Reitman, R. P.: An Axiomatic Approach to Information Flow in Programs. *ACM Transactions on Programming Languages and Systems*, **2** (1), 56–76 (1980).

[BHA86] Burn, G., Hankin, C. and Abramsky, S.: Strictness Analysis for Higher-Order Functions. *Science of Computer Programming*, **7**, 249–278 (1986).

[CoC77] Cousot, P. and Cousot, R.: Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs. *Proc. 4th ACM Principles of Programming Languages*, 1977, pp. 238–252.

[Den75] Denning, D. E.: Secure Information Flow in Computer Systems. PhD Dissertation, Purdue University, 1975.

[Den76] Denning, D. E.: A Lattice Model of Secure Information Flow. *Communications of the ACM*, **19** (5), 236–243 (1976).

[DeD77] Denning, D. E. and Denning, P. J.: Certification of Programs for Secure Information Flow. *Communications of the ACM*, **20** (7), 504–512 (1977).

[Don82] Donzeau-Gouge, V.: Denotational Definition of Properties of Program Computations. In: *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones (eds), Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 343–379.

[Gra79] Graetzer, G.: *Universal Algebra*, 2nd edn. Springer, Berlin, 1979.

[JoM86] Jones, N. and Mycroft, A.: Data Flow Analysis of Applicative Programs Using Minimal Function Graphs, *Proc. 13th ACM Symp. on Principles of Programming Languages*, St. Petersburg, FL, 1986.

[JøS86] Jørring, U. and Scherlis, W. L.: Compilers and Staging Transformations. *Proc. 13th ACM Symp. on Principles of Programming Languages*, St. Petersburg, FL, 1986, pp. 86–96.

[KaU77] Kam, J. and Ullman, J.: Monotone Data Flow Analysis Networks. *Acta Informatica*, **7**, 305–317 (1977).

[MiO87] Mizuno, M. and Oldehoeft, A. E.: Information Flow Control in a Distributed Object-Oriented System with Statically Bound Object Variables. *Proc. 10th National Computer Security Conf.*, 1987, pp. 56–67.

[Miz87] Mizuno, M.: Highly Structured Software for Network Systems and Protection. PhD dissertation, Computer Science Dept., Iowa State University, Ames, Iowa, 1987.

[Miz89] Mizuno, M.: A Least Fixed Point Approach to Inter-Procedural Information Flow Control. *Proc. 12th National Computer Security Conf.*, 1989, pp. 558–570.

[MoW88] Montenyohl, M. and Wand, M.: Correct Flow Analysis in Continuation Semantics, *Proc. 15th Annual ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages*, San Diego, California, January 1988, pp. 204–218.

[Nie83] Nielson, F.: A Denotational Framework for Data Flow Analysis. *Acta Informatica*, **18**, 265–288 (1983).

[Nie85] Nielson, F.: Program Transformations in a Denotational Setting. *ACM Transactions on Programming Languages and Systems*, **7**, 359–379 (1985).

[Nie89] Nielson, F.: Two Level Semantics and Abstract Interpretation. *Theoretical Computer Science*, **69** (2), 117–242 (1989).

[NiN92] Nielson, H. R. and Neilson, F.: Bounded Fixed Point Iteration. *Proc. 19th ACM Symp. on Principles of Programming Languages*, January 1992.

[Plo80] Plotkin, G.: Lambda-Definability in the Full Type Hierarchy. In: *To: H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, J. P. Seldin and J. R. Hindley (eds), Academic Press, New York, 1980.

[Sch88] Schmidt, D. A.: *Denotational Semantics*. W. C. Brown, Dubuque, Iowa, 1988.

[Sto77] Stoy, J.: *Denotational Semantics*. MIT Press, Cambridge, MA, 1977.

Appendix 1. Proof of Theorem 2

First, for all E ∈ Expression, if a has unguarded form, then E⟦E⟧a has form $v_i[\ \sqcup \alpha]$; the proof is an easy induction. Now we consider the cases for C:

- I := E: *update i* E⟦E⟧a = cases $(a\,i)$ of is*Static*$(t) \to$ (E⟦E⟧$a \sqsubseteq t \to a\ [\,]\ \top$) is*Dynamic*$(t) \to [i \mapsto$ in*Dynamic*(E⟦E⟧a)] a end.

The polynomial must simplify to either of the following:

1. (E⟦E⟧$a \sqsubseteq t \to a\ [\,]\ \top) = (v[\ \sqcup \alpha] \sqsubseteq t \to s\ [\,]\ \top)$, which is a guarded form.
2. $[i \mapsto$ in*Dynamic*$(v[\ \sqcup \alpha])]s$, which is a guarded form ($C = \varnothing$, that is, *true*).

- C$_1$;C$_2$: (C⟦C$_2$⟧ *comp* C⟦C$_1$⟧ a) = let a' = C⟦C$_1$⟧a in C⟦C$_2$⟧a'. By the inductive hypothesis, C⟦C$_1$⟧a has guarded form: $(C \to a'\ [\,]\ \top)$. Since "let" is "$\top$-strict", the denotation is $(C \to$ C⟦C$_2$⟧$a'\ [\,]\ \top)$. By the inductive hypothesis, C⟦C$_2$⟧a' has guarded form as well, so we have $(C \to (C' \to a''\ [\,]\ \top)\ [\,]\ \top) = (C \cup C' \to a''\ [\,]\ \top)$, where $C \cup C'$ represents the merging of the two constraints sets into one, that is, the merging of $e_1 \sqsubseteq t_0$ with $e_2 \sqsubseteq t_0$ is: $e_1 \sqcup e_2 \sqsubseteq t_0$.
- **if** E **then** C$_1$ **else** C$_2$: Follows from the definition of *cond*, Lemma 3 and the inductive hypothesis.
- **while** E **do** C: From Theorem 1, we have C⟦**while** E **do** C⟧a = ($\lambda a.$*newcontext* I⟦C⟧ (E⟦E⟧a) a)$(\bigsqcup_{0 \leqslant i \leqslant k} e_i)$, where $e_0 = a$, and $e_{i+1} = (\lambda a.$let a' = *newcontext* I⟦C⟧ (E⟦E⟧a) a in C⟦C⟧$a')e_i$. All e_i rewrite to guarded form (cf. the previous two cases), hence so does $\bigsqcup_{0 \leqslant i \leqslant k} e_i$, by Lemma 3.

As in Proposition 8, we must verify that convergence is detectable. We must find a k, for reasons similar to those in the proof of Proposition 8: if there is not convergence, then there is a sequence of nonconverging expressions C_i or s_i in the sequence of guarded poststore expressions. Hence, there exists a nonconverging sequence x_i of *Sec* expressions. By Theorem 1, the sequence x_i is a "chain" in the sense explained in the proof of Proposition 8. The result follows. □

Appendix 2. Proof of Theorem 3

The relation $p\, rec_{Poststore}\, p'$ boils down to $p \sqsubseteq (recover\, p')$, which we use hereon. Without loss of generality, when assuming $a\, rec_{Store}\, a'$, also assume that a and a' are variable consistent (cf. Definition 1). In the following, we use $[\cdots e_1 \cdots]e_2$ as an abbreviation for $[f(calc\, e_1)\downarrow i/f_i.\mu]_{i\in I}\, e_2$. The proof is an induction on C, proving $C^+_{abs}[\![C]\!]\, rec_{Poststore}\, C'_{abs}[\![C]\!]$:

- $I := E$: $C^+_{abs}[\![I := E]\!]a = update\ I\ E[\![E]\!]a$, and $recover\ (C'_{abs}[\![I := E]\!]a') = (recover\ a'') = [\cdots a'' \cdots]a''$, where $a'' = update\ I\ E[\![E]\!]\, a'$.

  Now, $(calc\, a'') = fix(\lambda(\tau_i)_{i\in I}.\,(access\, f_i.in\ ([f\tau_i)/f_i.\mu]_{i\in I}\,(update\ I\ E[\![E]\!]\, a')))_{i\in I}) = fix(\lambda(\tau_i)_{i\in I}.(access\ f_i.in\ ([(f\tau_i)/f_i.\mu]_{i\in I} a'))_{i\in I})$, since $I \neq f_i.in$ for any $i \in I$, $= (calc\, a')$.

  So, we have $recover(C'_{abs}[\![I := E]\!]a') = [\cdots a' \cdots]a'' = (update\ I\ E[\![E]\!]\ [\cdots a' \cdots]a') = (update\ I\ E[\![E]\!]\ (recover\ a'))$.

  Since $a\, rec_{Store}\, a'$, we have $a \sqsubseteq recover\, a'$, and the monotonicity of $E[\![E]\!]$ gives the result.

- $C_1;C_2$: the result is immediate from the definition of *comp*.

- **call** F_k(**in** E, **out** I): $C^+_{abs}[\![\mathbf{call}\ F_k(\mathbf{in}\ E, \mathbf{out}\ I)]\!]a = call\, f\, (access\, f_k.in\ I)\, a_1 = update\ I\ (f \circ (access\, f_k.in))a_1 = update\ I\ (f \circ E[\![E]\!])\, a_1$, because no $f_i.in$ appear in E, where $a_1 = update\ f_k.in\ E[\![E]\!]a$. And, $recover(C'_{abs}[\![\mathbf{call}\ F_k(\mathbf{in}\ E,\ \mathbf{out}\ I)]\!]a') = [\cdots a'_1 \cdots]a'_1$, where $a'_1 = update\ I\ (const\, f_k.\mu)\ (update\, f_k.in\ (E[\![E]\!] \sqcup (access\, f_k.in)))\ a'$.

  Clearly, for all $j \neq k$, $(calc\ a'_1)\downarrow j = (calc\, a')\downarrow j$; and $(calc\, a'_1)\downarrow k = (calc\ (update\ f_k.in\ (E[\![E]\!] \sqcup (access\, f_k.in))a')\downarrow k$, since $I \neq f_k.in$, for all $i \in I$. This equals $(E[\![E]\!]\ [\cdots a' \cdots]a' \sqcup ([\cdots a' \cdots]a'\, f_k.in)$, by unfolding the definition of *calc* and indexing by k. So, $(calc\ a'_1\downarrow k) = E[\![E]\!](recover\ a') \sqcup (access\, f_k.in\ (recover\ a'))$.

  From the above, we have that: $recover(C'_{abs}[\![\mathbf{call}\ F_k(\mathbf{in}\ E, \mathbf{out}\ I)]\!]a') = [\cdots a'_1 \cdots]a'_1 = (update\ I\ (const\ (f(E[\![E]\!](recover\ a') \sqcup (access\, f_k.in\ (recover\ a'))))) \ (update\, f_k.in\ (E[\![E]\!] \sqcup (access\, f_k.in))a'')$, where $a'' = [\cdots a'_1 \cdots]a'$, that is

$$a'' = [\cdots a' \cdots]_{i\in I-\{k\}}[f(E[\![E]\!]\,(recover\ a') \sqcup (access\, f_k.in\ (recover\ a')))/f_k.\mu]a'$$

  Clearly, $(recover\ a') \sqsubseteq a''$. Hence, $a \sqsubseteq a''$, by $a\, rec_{Store}\, a'$. Hence, $a_1 \sqsubseteq (update\ f_k.in\ (E[\![E]\!] \sqcup (access\ f_k.in))\ a''$. (Call this value a'''.) Since $(f(access\ f_k.in\ a_1)) = (f(E[\![E]\!]a)) \sqsubseteq (f(E[\![E]\!](recover\ a') \sqcup (access\ f_k.in\ (recover\ a'))))$, we have that $(update\ I\ (f \circ (access\, f_k.in))a_1) \sqsubseteq (update\ I\ (const\ (f(E[\![E]\!](recover\ a') \sqcup (access\, f_k.in\ (recover\ a')))))$, which gives the result.

- **if** E **then** C_1 **else** C_2: $C^+_{abs}[\![\mathbf{if}\ E\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2]\!]a = \text{let } a_0 = (newcontext\ (I[\![C_1]\!] \cup I[\![C_2]\!]\ (E[\![E]\!]a)\ a) \text{ in } C[\![C_1]\!]a_0 \cup C[\![C_2]\!]a_0$. C'_{abs} is defined similarly. We can assume $a\, rec_{Store}\, a'$. The result follows from a proof that $newcontext\ I[\![C_1]\!] \cup I[\![C_2]\!]\ (E[\![E]\!]a)$

$a\ rec_{Poststore}$ *newcontext* $I[\![C_1]\!] \cup I[\![C_2]\!]$ $(E[\![E]\!]a')\ a'$, and the proof is by induction on $I[\![C_1]\!] \cup I[\![C_2]\!]$. There are two cases:

1. $\{\ \}$: immediate.
2. $\{i\}::S$: There are two subcases:

(a) (a, i) is in*Static*(t_0): Then, $(a'\ i)$ is in*Static*(t_0), by variable consistency. Let $a_1 = (E[\![E]\!]a' \sqsubseteq t_0 \rightarrow newcontext\ S\ (E[\![E]\!]a')\ a' \,[\!]\, \top)$; this means that *recover*(cases $(a'\ i)$ of $\cdots) = recover(a_1) = E[\![E]\!]\ [\cdots a_1 \cdots]a' \sqsubseteq t_0 \rightarrow [\cdots a_1 \cdots](newcontext\ S\ (E[\![E]\!]a') \,[\!]\, \top$.

With a fixed point induction, we can show $calc\, a' \sqsubseteq calc\, a_1$, since the proof boils down to showing, for an arbitrary $f_i.\mu$, that $a' \sqsubseteq a_1$. Thus, $a \sqsubseteq [\cdots a' \cdots]a' \sqsubseteq [\cdots a_1 \cdots]a'$. This information plays a key role in this analysis of cases:

(i) $E[\![E]\!]a \sqsubseteq t_0$ is false: then $E[\![E]\!]\ [\cdots a_1 \cdots]a' \sqsubseteq t_0$ is also false, and we have $\top \sqsubseteq \top$.

(ii) $E[\![E]\!]a \sqsubseteq t_0$ is true: if $E[\![E]\!]\ [\cdots a_1 \cdots]a' \sqsubseteq t_0$ is false, then $(newcontext\ S\ (E[\![E]\!]a)\ a) \sqsubseteq \top$. If $E[\![E]\!]\ [\cdots a_1 \cdots]a' \sqsubseteq t_0$ is true, then we must verify that $(newcontext\ S\ (E[\![E]\!]a)\ a) \sqsubseteq [\cdots a_1 \cdots]\,(newcontext\ S\ (E[\![E]\!]a')\ a')$. By hypothesis on S, we have $newcontext\ S\ (E[\![E]\!]a)\ a \sqsubseteq recover(newcontext\ S\ (E[\![E]\!]a')\ a') = [\cdots a' \cdots]\,(newcontext\ S\ (E[\![E]\!]a')\ a')$. Since $calc\, a' \sqsubseteq calc\, a_1$, we have that the latter is $\sqsubseteq [\cdots a_1 \cdots]\,(newcontext\ S\ (E[\![E]\!]a')\ a')$.

(b) $(a\ i)$ is in*Dynamic*(t_0): Then, $(a'\ i)$ is in*Dynamic*(t_1), by variable consistency. If we show $[i \mapsto \text{in}Dynamic(E[\![E]\!]a \sqcup t_0)]a\ rec_{Store}\,[i \mapsto \text{in}Dynamic(E[\![E]\!]a' \sqcup t_1)]a'$, then by the inductive hypothesis on S, we have the result.

Let $a_1 = [i \mapsto \text{in}Dynamic(E[\![E]\!]a' \sqcup t_1)]a'$. Then, $recover(a_1) = [i \mapsto \text{in}Dynamic(E[\![E]\!]\,[\cdots a_1 \cdots]a' \sqcup [\cdots a_1 \cdots]t_1)]\,[\cdots a_1 \cdots]a'$. It is straightforward to prove $calc\, a' \sqsubseteq calc\, a_1$. So, we need only show $(a\ i) = t_0 \sqsubseteq [\cdots a_1 \cdots]t_1 = [\cdots a_1 \cdots](a'\ i)$. We have that $(a\ i) \sqsubseteq (recover\ a')i = ([\cdots a' \cdots]a')i \sqsubseteq ([\cdots a_1 \cdots]a')i$, since $calc\, a' \sqsubseteq calc\, a_1$.

• **while E do C**: $C^{+}_{abs}[\![\text{while E do C}]\!] = \bigsqcup_{i \geqslant 0} g_i$, $C'_{abs}[\![\text{while E do C}]\!] = \bigsqcup_{i \geqslant 0} g'_i$. Using a development like that in the previous case, we can show, for all $i \geqslant 0$, that $g_i\, rec_{Store \rightarrow Store}\, g'_i$. Since the relation *rec* is inclusive, the result follows. □

Appendix 3. Derivation of Recursive Procedure Analysis

The intuition in the derivation that follows is that the recursive calls and returns are converted into "go-tos". First, we compress all recursive calls into one call by renaming all occurrences of the $f_i.\mu$ unknowns to $f'.\mu$. Let

$$\begin{aligned} &f_1 = \lambda\alpha.f.\mu \\ &\text{where } f.\mu = access\ I\ [f'.\mu/f_i.\mu]_{i\in I}\, a'_\alpha \\ &\{\tau_i = access\ f_i.in\ [f'.\mu/f_i.\mu]_{i\in I}\, a'_\alpha\}_{i\in I} \\ &f'.\mu = f(\textstyle\bigsqcup_{i\in I}\tau_i) \end{aligned}$$

Clearly, $f \sqsubseteq f_1$. Next, we weaken f_1 by replacing all occurrences of α by $\alpha \sqcup (\bigsqcup_{i\in I}\tau_i)$ and $f(\bigsqcup_{i\in I}\tau_i)$ by $f(\alpha \sqcup (\bigsqcup_{i\in I}\tau_i))$. We obtain

$$\begin{aligned} &f_2 = \lambda\alpha.f.\mu \\ &\text{where } f.\mu = access\ I\ [f'.\mu/f_i.\mu]_{i\in I}\, a'_{\alpha \sqcup (\bigsqcup_{i\in I}\tau_i)} \\ &\{\tau_i = access\ f_i.in\ [f'.\mu/f'_i.\mu]_{i\in I}\, a'_{\alpha \sqcup (\bigsqcup_{i\in I}\tau_i)}\}_{i\in I} \\ &f'.\mu = f(\textstyle\bigsqcup_{i\in I}\tau_i) \end{aligned}$$

We have $f_1 \sqsubseteq f_2$. Next, we weaken the call $f(\bigsqcup_{i\in I}\tau_i)$, giving

$$
\begin{aligned}
&f_3 = \lambda\alpha.f.\mu\\
&\text{where } f.\mu = \mathit{access}\ \mathrm{I}\ [f'.\mu/f_i.\mu]_{i\in I}\, a'_{\alpha \sqcup (\bigsqcup_{i\in I}\tau_i)}\\
&\{\tau_i = \mathit{access}\, f_i.\mathit{in}\ [f'.\mu/f_i.\mu]_{i\in I}\, a'_{\alpha \sqcup (\bigsqcup_{i\in I}\tau_i)}\}_{i\in I}\\
&f'.\mu = f_3(\alpha \sqcup (\bigsqcup_{i\in I}\tau_i))
\end{aligned}
$$

We have $f \sqsubseteq f_2 \sqsubseteq f_3$. Once we show that $f_3(\alpha) = f_3(\alpha \sqcup (\bigsqcup_{i\in I}\tau_i))$, we note that $f'.\mu = f_3(\alpha) = f.\mu$, and we obtain the first-order version of f_3:

$$
\begin{aligned}
&f_3 = \lambda\alpha.f.\mu\\
&\text{where } f.\mu = \mathit{access}\ \mathrm{I}\ [f'.\mu/f_i.\mu]_{i\in I}\, a'_{\alpha \sqcup (\bigsqcup_{i\in I}\tau_i)}\\
&\{\tau_i = \mathit{access}\, f_i.\mathit{in}\ [f'.\mu/f_i.\mu]_{i\in I}\, a'_{\alpha \sqcup (\bigsqcup_{i\in I}\tau_i)}\}_{i\in I}\\
&f'.\mu = f.\mu
\end{aligned}
$$

This is the form of analysis of recursive procedures used in Section 3. We now show why $f_3(\alpha) = f_3(\alpha \sqcup (\bigsqcup_{i\in I}\tau_i))$. First, $f_3(\alpha \sqcup (\bigsqcup_{i\in I}\tau_i)) = f.\mu$ where

$$
\begin{aligned}
&f.\mu = \mathit{access}\ \mathrm{I}\ [f'.\mu/f_i.\mu]_{i\in I}\, a'_{\alpha \sqcup (\bigsqcup_{i\in I}\tau_i) \sqcup (\bigsqcup_{i\in I}\tau'_i)}\\
&\{\tau'_i = \mathit{access}\, f_i.\mathit{in}\ [f'.\mu/f_i.\mu]_{i\in I}\, a'_{\alpha \sqcup (\bigsqcup_{i\in I}\tau_i) \sqcup (\bigsqcup_{i\in I}\tau'_i)}\}_{i\in I}\\
&f'.\mu = f_3(\alpha \sqcup (\bigsqcup_{i\in I}\tau_i) \sqcup \bigsqcup_{i\in I}\tau'_i)
\end{aligned}
$$

Regardless of the value of $\bigsqcup_{i\in I}\tau'_i$, we have $\alpha \sqcup (\bigsqcup_{i\in I}\tau_i) \sqsubseteq \alpha \sqcup (\bigsqcup_{i\in I}\tau_i) \sqcup (\bigsqcup_{i\in I}\tau'_i)$. Since the τ_i's are

$$
\{\tau_i = \mathit{access}\, f_i.\mathit{in}\ [f_3(\alpha \sqcup (\bigsqcup_{i\in I}\tau_i))/f_i.\mu]_{i\in I}\, a'_{\alpha \sqcup (\bigsqcup_{i\in I}\tau_i)}\}_{i\in I}
$$

it must be, for all $i \in I$, that $\tau_i \sqsubseteq \tau'_i$. Hence, $\bigsqcup_{i\in I}\tau_i \sqsubseteq \bigsqcup_{i\in I}\tau'_i$, implying that $\alpha \sqcup (\bigsqcup_{i\in I}\tau_i) \sqcup \bigsqcup_{i\in I}\tau'_i = \alpha \sqcup \bigsqcup_{i\in I}\tau'_i$. Hence, $f_3(\alpha \sqcup (\bigsqcup_{i\in I}\tau_i)) = f.\mu$, where

$$
\begin{aligned}
&f.\mu = \mathit{access}\ \mathrm{I}\ [f'.\mu/f_i.\mu]_{i\in I}\, a'_\alpha \sqcup (\bigsqcup_{i\in I}\tau'_i)\\
&\{\tau'_i = \mathit{access}\, f_i.\mathit{in}\ [f'.\mu/f_i.\mu]_{i\in I}\, a'_\alpha \sqcup (\bigsqcup_{i\in I}\tau'_i)\}_{i\in I}\\
&f'.\mu = f_3(\alpha \sqcup \bigsqcup_{i\in I}\tau'_i)
\end{aligned}
$$

which is just $f_3(\alpha)$.

Received May 1991
Accepted in revised form in January 1992 by D. Bjorner