Watchpoint Semantics: A Tool for Compositional and Focussed Static Analyses

Fausto Spoto

Dipartimento Scientifico e Tecnologico
Strada Le Grazie, 15, Ca' Vignal, 37134 Verona, Italy
spoto@sci.univr.it

Abstract We abstract a denotational trace semantics for an imperative language into a compositional and focussed *watchpoint semantics*. Every abstraction of its computational domain induces an abstract, still compositional and focussed watchpoint semantics. We describe its implementation and instantiation with a domain of signs. It shows that its space and time costs are proportional to the number of watchpoints and that abstract compilation reduces those costs significantly.

1 Introduction

A *compositional* analysis of a complex statement is defined in terms of that of its components. Then the analysis of a procedure depends only on that of the procedures it calls and the analysis of a huge program can be easily kept up-to-date. This is important if a local change is applied to the program, during debugging or as a consequence of program transformation. A *focussed* or *demand-driven* analysis is *directed* to a given set of program points, called *watchpoints*, and has a cost (in space and time) proportional to their number. This is important if only those points are relevant. For instance, zero information is typically useful only before a division. Class information for object-oriented programs is typically useful only before a method call. During debugging the programmer wants to analyse a program in very few points, with a cost proportional to their number.

Our first contribution here is the definition of a compositional and focussed *watchpoint semantics*, as an abstract interpretation (AI) [6] specified by the watchpoints of interest of a more concrete trace semantics. An optimality result (Equation (4)) states that no precision is lost by this abstraction w.r.t. the information at the watchpoints. The *computational domain* is identified as a data structure with some operations. The second contribution states that every AI of the computational domain induces an abstraction of the watchpoint semantics. This reduces the problem of static analysis to that of the development of abstract domains. The third contribution is the description of our implementation of the watchpoint semantics instantiated for sign analysis. It shows that the space and time costs of the analysis are proportional to the number of watchpoints. The final contribution is to show that abstract compilation [11] leads to a significant improvement in the time and space costs of the analysis.

1.1 Related Works

Traditionally, compositionality is synonym with denotational semantics. But the usual denotational semantics [16] provides an input/output characterisation of procedures which is too *abstract* to observe their internal behaviour. This has been recognised in [5], where Cousot models information at internal program points through a more concrete, denotational *trace semantics*. Here, a procedure is denoted by a map from an initial *state* to a *trace* of states representing its execution from the initial state. Our trace semantics is a instance of the *maximal trace semantics* of [5]. Even [4] observes that traces contain more information than a traditional input/output denotation w.r.t. software pipelining, loop-invariant removal and data alias detection. That framework is however based on an operational definition. In [15] an operational trace semantics is defined and abstracted through AI into an *abstract trace semantics*. The abstraction of a trace is a (regular) tree, because of the non-deterministic nature of the abstract semantics. Information at program points (what they call the *collecting semantics*) is extracted from those trees *after* the fixpoint of the semantics is reached. Thus, their analysis is *not* focussed, since the whole trace semantics (the abstract trees) must be computed and then projected on program points. This is justified by the fact that they are interested in properties of traces, like those considered in the context of model-checking. However, for properties of states, like in sign, interval, class and security context analysis of variables (and, more generally, in all those analyses called *first-order* in [12], page 210), trees are not needed and can be safely abstracted into sets of states.

Abstract denotational semantics need a representation of the abstract input/output behaviour of a procedure. Since abstract inputs can partially overlap, the *meaning* or, in abstract interpretation words, the concretisation of an abstract denotation is not so easily devisable. In Section 6 we show a solution when the abstract domain elements can be written in terms of *union-irreducible* elements. For the general case, we rely on the *functional partinioning* technique defined in [1].

Focussed or demand-driven frameworks for analysis have been developed in the past. In [2] backward propagation of assertions was applied to the debugging of a high-order imperative language. In [8,9,13] backward dataflow analysis from a given query is defined and shown more efficient than an *exhaustive*, unfocussed analysis. The analyses in [8,9] are provably as precise as the corresponding unfocussed versions for *distributive finite dataflow problems*, while our optimality result (Equation (4)) holds for every abstract domain. Queries can be checked to hold in a given program point, but cannot be computed by the analysis. It is not shown how those analyses scale w.r.t. the number of queries. No notion of computational domain is defined, which makes harder the definition of new abstract analyses. Abstract compilation cannot be applied because those analyses are not compositional. In [1] it is studied a very general and abstract way of looking at the problem of localised analyses in a given set of program points. However, its actual application to a real programming language is not tackled there.

Abstract compilation (AC) was born and applied only in the context of the analysis of logic programs [3,11]. It is an optimised computation of the fixpoint of an abstract semantics where at the i-th iteration part of the analysis is *compiled* when it becomes clear that it will not change at the $(i + 1)$-th iteration.

Our work has been heavily influenced by the systematic construction of semantics for logic programming from the observable property of interest [10]. There, semantics for resultants, call patterns and computed answers of logic programs are derived through AI of the very concrete semantics of SLD-derivations. In particular, the call pattern semantics collects the information in some program points only.

1.2 Plan of the Paper

Section 2 introduces some preliminary notations. Section 3 defines the simple imperative language used in the paper. Section 4 defines the concrete trace semantics which is then abstracted into our watchpoint semantics of Section 5 and its collecting version. Section 6 shows how every abstract interpretation of the concrete computational domain induces an abstract watchpoint semantics. Section 7 describes our implementation of the watchpoint semantics instantiated for sign analysis. Section 8 concludes. Proofs are omitted.

2 Preliminaries

A *sequence* of elements from a set S is denoted by $\mathsf{seq}(S)$. The cardinality of a set S is denoted by $\#S$. A definition like $S = \langle a, b \rangle$, with a and b meta-variables, silently defines the selectors $s.a$ and $s.b$ for $s \in S$. For instance, Definition 8 defines $t.w$ and $t.s$ for $t \in \mathcal{T}_\tau^w$. An element $x \in X$ will often stand for the singleton $\{x\} \subseteq X$.

The domain (codomain) of a function f is $\mathsf{dom}(f)$ $(\mathsf{cd}(f))$. A total (partial) function is denoted by $\mapsto$ $(\rightarrow)$. We denote by $[v_1 \mapsto t_1, \ldots, v_n \mapsto t_n]$ a function f whose domain is $\{v_1, \ldots, v_n\}$ and such that $f(v_i) = t_i$ for $i = 1, \ldots, n$. Its update is $f[d_1/w_1, \ldots, d_m/w_m]$, where the domain can be potentially enlarged. By $f|_s$ $(f|_{-s})$ we denote the restriction of f to $s \subseteq \mathsf{dom}(f)$ (to $\mathsf{dom}(f) \setminus s$).

A pair $(C, \leq)$ is a *poset* if $\leq$ is reflexive, transitive and antisymmetric on C. A poset is a *complete lattice* when *least upper bounds* (lub) and *greatest lower bounds* (glb) always exist, a *complete partial order* (CPO) when lubs exist for the non-empty chains (totally ordered subsets). A CPO is *pointed* when it has a bottom element. A map is *additive* when it preserves all lubs. If $f(x) = x$ then x is a *fixpoint* of f. If a least fixpoint exists, it is denoted by $\mathsf{lfp}(f)$.

Let $(C, \leq)$ and $(A, \preceq)$ be two posets (the concrete and the abstract domain). A *Galois connection* [6,7] is a pair of monotonic maps $\alpha : C \mapsto A$ and $\gamma : A \mapsto C$ such that $\gamma\alpha$ is extensive and $\alpha\gamma$ is reductive. It is a *Galois insertion* when $\alpha\gamma$ is the identity map, i.e., when the abstract domain does not contain *useless* elements. This is equivalent to α being onto, or γ one-to-one. The *abstraction* α

and the *concretisation* γ determine each other. If C and A are complete lattices and α is additive, it is the abstraction map of a Galois connection.

An abstract operator $\hat{f} : A^n \to A$ is *correct* w.r.t. $f : C^n \to C$ if $\alpha f \gamma \preceq \hat{f}$. For each operator f, there exists an *optimal* (most precise) correct abstract operator $\hat{f}$ defined as $\hat{f} = \alpha f \gamma$. If $\hat{f}\alpha = \alpha f$, we say that $\hat{f}$ is α-*optimal* w.r.t. f, i.e., $\hat{f}$ computes the same abstract information as f.

In AI, the *semantics* of a program is the fixpoint of some $f : C \mapsto C$, where C is the computational domain [5]. Its *collecting version* [6] works over *properties* of C, i.e., over $\wp(C)$ and is the fixpoint of the powerset extension of f. If f is defined through suboperations, their powerset extensions *and* $\cup$ (which merges the semantics of the branches of a conditional) induce the extension of f.

3 A Simple Language

Our language is left expandable at the price of some redundancy in the definitions. Integers are its only basic type, with two operations ($=$ and $+$). Booleans are implemented as integers ($false$ is represented by a negative integer and *true* by any other integer). We do not have procedures but only functions. Those limitations are just meant to simplify the presentation. Our framework can be extended to cope with those missing items.

Definition 1. *Let Id be a set of* identifiers *and $\mathcal{F} \subseteq Id$ a finite set of* function symbols*.* Expressions *$\mathcal{E}$ and* commands *$\mathcal{C}$ are defined by the grammar*

$$e ::= i \mid v \mid f(v_1, \ldots, v_n) \mid e = e \mid e + e$$
$$c ::= (v := e) \mid c;c \mid \texttt{let } v{:}t \texttt{ in } c \mid \texttt{if } e \texttt{ then } c \texttt{ else } c \mid \texttt{while } e \texttt{ do } c$$

with $Type = \{int\}$, $Int = \mathbb{Z}$, $t \in Type$, $i \in Int$, $f \in \mathcal{F}$ and $v, v_1, \ldots, v_n \in Id$.

A typing gives types to a finite set of variables. The map $Pars$ binds every function to the typing (its signature) and the list of its parameters. This list provides the order of the variables in the definition of the function. The function *Code* binds a function symbol to its (syntactically correct and type-checked) code. Local variables are always introduced by a `let` construct.

Definition 2. *We define $Typing = \{\tau : Id \to Type \mid \mathsf{dom}(\tau)$ is finite$\}$, $Code = (\mathcal{F} \mapsto \mathcal{C})$ and $Pars = (\mathcal{F} \mapsto \mathsf{seq}(Id) \times Typing)$. If $p \in Pars$ and $f \in \mathcal{F}$, then $p(f) = \langle s, \tau \rangle$, where $f \in s$ and $\mathsf{dom}(\tau) = s$. The variable f holds the return value of the function, like in Pascal.*

A program is specified by $\mathcal{F}$ and two elements in *Code* and $Pars$. In the following, we assume that we have a program $P = \langle \mathcal{F}, c, p \rangle$.

Expressions have a type in a typing. In our case, $\mathsf{type}_\tau(e) = int$ for $e \in \mathcal{E}$.

Example 1. Figure 10(a) gives a representation of the program for computing the n-th Fibonacci number (lines introduced by % are comments and will be discussed later). Note that the name `fib` of the (only) function in the program is used to hold its result value. Moreover, it is described by the $Pars$ map p of the program.

$$\mathsf{nop}_\tau : \Sigma_\tau \mapsto \Sigma_\tau$$
$$\mathsf{get_int}^i_\tau : \Sigma_\tau \mapsto \Sigma_{\tau[int/res]} \quad \text{with } res \notin \mathsf{dom}(\tau),\ i \in Int$$
$$\mathsf{get_var}^v_\tau : \Sigma_\tau \mapsto \Sigma_{\tau[\tau(v)/res]} \quad \text{with } v \in \mathsf{dom}(\tau),\ res \notin \mathsf{dom}(\tau)$$
$$\mathsf{put_var}^v_\tau : \Sigma_\tau \mapsto \Sigma_{\tau|_{-res}} \quad \text{with } res \in \mathsf{dom}(\tau),\ v \in \mathsf{dom}(\tau),\ v \neq res,\ \tau(v) = \tau(res)$$
$$=_\tau, +_\tau : \Sigma_\tau \mapsto (\Sigma_\tau \mapsto \Sigma_\tau) \quad \text{with } res \in \mathsf{dom}(\tau),\ \tau(res) = int$$
$$\mathsf{scope}^{f,v_1,\ldots,v_{\#p(f).s-1}}_\tau : \Sigma_\tau \mapsto \Sigma_{p(f).\tau|_{-f}}$$
$$\text{with } p(f).s \setminus f = \langle \iota_1, \ldots, \iota_n \rangle \text{ and } \{v_1, \ldots, v_{\#p(f).s-1}\} \subseteq \mathsf{dom}(\tau)$$
$$\mathsf{unscope}^f_\tau : \Sigma_\tau \mapsto (\Sigma_{p(f).\tau|_f} \to \Sigma_{\tau[p(f).\tau(f)/res]}) \quad \text{with } res \notin \mathsf{dom}(\tau)$$
$$\mathsf{restrict}^{vs}_\tau : \Sigma_\tau \mapsto \Sigma_{\tau|_{-vs}} \quad \text{with } vs \subseteq \mathsf{dom}(\tau)$$
$$\mathsf{expand}^{v:t}_\tau : \Sigma_\tau \mapsto \Sigma_{\tau[t/v]} \quad \text{with } v \notin \mathsf{dom}(\tau),\ t \in Type$$
$$\mathsf{is_true}_\tau, \mathsf{is_false} \subseteq \wp(\Sigma_\tau) \quad \text{with } \tau(res) = int\ .$$

$$\mathsf{nop}_\tau(\sigma) = \sigma \qquad \mathsf{get_int}^i_\tau(\sigma) = \sigma[i/res]$$
$$\mathsf{get_var}^v_\tau(\sigma) = \sigma[\sigma(v)/res] \qquad \mathsf{put_var}^v_\tau(\sigma) = \sigma[\sigma(res)/v]|_{-res}$$
$$+_\tau(\sigma_1)(\sigma_2) = \sigma_2[\sigma_1(res) + \sigma_2(res)/res] \qquad =_\tau(\sigma_1)(\sigma_2) = \begin{cases} \sigma_2[1/res] & \text{if } \sigma_1(res) = \sigma_2(res) \\ \sigma_2[-1/res] & \text{if } \sigma_1(res) \neq \sigma_2(res) \end{cases}$$
$$\mathsf{scope}^{f,v_1,\ldots,v_n}_\tau(\sigma) = [\iota_1 \mapsto \sigma(v_1), \ldots, \iota_n \mapsto \sigma(v_n)] \quad \text{where } \langle \iota_1, \ldots, \iota_n \rangle = p(f).s \setminus f$$
$$\mathsf{unscope}^f_\tau(\sigma_1)(\sigma_2) = \sigma_1[\sigma_2(f)/res] \qquad \mathsf{restrict}^{vs}_\tau(\sigma) = \sigma|_{-vs} \qquad \mathsf{expand}^{v:t}_\tau(\sigma) = \sigma[\mathsf{init}(t)/v]$$
$$\mathsf{is_true}_\tau(\sigma) \text{ if and only if } \sigma(res) \geq 0 \qquad \mathsf{is_false}_\tau(\sigma) \text{ if and only if } \sigma(res) < 0$$
$$\mathsf{init}(int) = 0\ .$$

Figure 1. The signature and implementation of the operations over the states.

4 Trace Semantics

The computational domain of states described here is used below to define a trace semantics for our language. Each of its abstractions will induce an abstraction of that semantics (Section 6), as usual in AI (see for instance [14]). More complex notions of states could be used here, maybe dealing with locations and memory.

Definition 3. *Let $Value = Int$ and $\Sigma = \cup_{\tau \in Typing} \Sigma_\tau$ where, for $\tau \in Typing$, states $\sigma \in \Sigma_\tau$ map variables to values consistent with their declared type, i.e.,*

$$\Sigma_\tau = \left\{ \sigma \,\middle|\, \begin{array}{l} \sigma \in \mathsf{dom}(\tau) \mapsto Value \text{ and} \\ \text{for every } v \in \mathsf{dom}(\tau) \text{ if } \tau(v) = int \text{ then } \sigma(v) \in Int \end{array} \right\}.$$

States are endowed with the operations shown in Figure 1.

In the operations of Figure 1, the variable *res* holds intermediate results. The nop operation does nothing. The $\mathsf{get_int}$ ($\mathsf{get_var}^v$) operation loads an integer (the value of v) in *res*. The $\mathsf{put_var}^v$ operation copies the value of *res* in v. There is no result, then *res* is removed. For every binary operation like $=$ and $+$, there is an operation on states. The operations scope and $\mathsf{unscope}$ are used before and after a call to a function f, respectively. The former creates a new state in which

f can execute. Its typing $p(f).\tau|_{-f}$ describes the input parameters (the variable f is not among them). The latter copies in the variable res of the state before the call, i.e., its first argument, the result of f, i.e., the variable f of its second argument. The operation expand ($\mathsf{restrict}$) adds (removes) variables from a state. The $\mathsf{is_true}$ ($\mathsf{is_false}$) predicate checks whether res contains $true$ ($false$).

Since res plays a major role, we introduce the following abbreviations.

Definition 4. *For $\tau \in Typing$, $\sigma \in \Sigma_\tau$ and $e \in \mathcal{E}$, let $\tau^e = \tau[\mathsf{type}_\tau(e)/res]$ and $\llcorner \sigma \lrcorner_\tau = \mathsf{restrict}_\tau^{res}(\sigma)$ (τ will be always omitted).*

We define now an instance of the *maximal trace semantics* of [5].

Definition 5. *A* trace *$t \in \mathcal{T}$ is a non-empty sequence in Σ. A* convergent *trace $\sigma_1 \to \cdots \to \sigma_n$ represents a terminated computation, a* finite divergent *trace $\sigma_1 \to \cdots \to \tilde{\sigma}_n$ a yet non-terminated computation and an* infinite divergent *trace $\sigma_1 \to \cdots \to \sigma_n \to \cdots$ a divergent computation. Arrows are given* labels *$l \in Label$, like in $\to^l$, meaning that the interpreter was then in a watchpoint labelled with l (see Section 5). We assume $\to$ is given a hidden mark $_ \notin Label$.*

The first state of $t \in \mathcal{T}$ is $\mathsf{fst}(t)$. The predicate $\mathsf{div}(t)$ means that t is divergent. If $\neg\,\mathsf{div}(t)$, the last state of t is $\mathsf{lst}(t)$. For $l \in Label$ and $\sigma \in \Sigma$, we let $\sigma \in_l t$ mean that σ occurs in t before an arrow $\to^l$.

The $\sqsubseteq$ ordering on traces (extension of finite divergent traces) is the minimal relation such that $t_1 \sqsubseteq t_2$ if $t_1 = t_2$ or (t_1 is finite divergent and $t_2 = \underline{t_1} \to^l t'$ for some $t' \in \mathcal{T}$ and $l \in Label \cup \{_\}$), where $\underline{t_1}$ is t_1 deprived of the tilde sign.

Expressions and commands are *denoted* by a map from an initial state to a trace t. In the first case, if $\neg\,\mathsf{div}(t)$ then $\mathsf{lst}(t)(res)$ is the value of the expression.

Proposition 1. *Given $\tau, \tau' \in Typing$, we expand the $\sqsubseteq$ ordering on traces to*

$$C_{\tau,\tau'} = \{c \in \Sigma_\tau \mapsto \mathcal{T}_{\tau'} \mid \textit{for every } \sigma \in \Sigma_\tau \textit{ we have } \mathsf{fst}(c(\sigma)) = \sigma\}\ . \tag{1}$$

The pair $\langle C_{\tau,\tau'}, \sqsubseteq\rangle$ is a pointed CPO whose bottom is $\perp_{C_{\tau,\tau'}} = \lambda\sigma \in \Sigma_\tau.\tilde{\sigma}$.

Interpretations denote every $f \in \mathcal{F}$ with an element of $C_{p(f).\tau|_{-f},p(f).\tau|_f}$. Indeed, its input variables are $p(f).s \setminus f$ and its output variable is named f.

Example 2. The program of Figure 10(a) is denoted by an interpretation which denotes `fib` with an element of $C_{[\mathtt{n}\mapsto int],[\mathtt{fib}\mapsto int]}$.

Definition 6. *The* interpretations *$\mathcal{I}$ are maps $I : \mathcal{F} \mapsto (\Sigma \to \mathcal{T})$ such that $I(f) \in C_{p(f).\tau|_{-f},p(f).\tau|_f}$ for $f \in \mathcal{F}$. The $\sqsubseteq$ ordering is point-wise extended to $\mathcal{I}$.*

Proposition 2. *The semantic operations on denotations of Figure 2 (the subscripts will be usually omitted) are monotonic w.r.t. $\sqsubseteq$.*

The operation $[op]$ applies an operation op from Figure 1. The operation ? joins the denotation E of an expression with that of one of two commands, depending on $\mathsf{is_true}$ and $\mathsf{is_false}$ on the final states of E. Since commands do not receive a partial result in res, we restrict those states through $\llcorner\,\lrcorner$. The operations $\otimes$

$$
\begin{aligned}
[op] &: C_{\tau,\tau'}, \text{ with } \tau,\tau' \in \mathit{Typing},\ op : \Sigma_\tau \to \Sigma_{\tau'}\\
?_\tau &: C_{\tau,\tau[int/res]} \times C^2_{\tau,\tau} \mapsto C_{\tau,\tau}, \text{ with } \tau \in \mathit{Typing},\ res \notin \mathsf{dom}(\tau)\\
\otimes_{\tau,\tau',\tau''} &: (C_{\tau,\tau'} \times C_{\tau',\tau''}) \mapsto C_{\tau,\tau''}, \text{ with } \tau,\tau',\tau'' \in \mathit{Typing}\\
\otimes_{bop,\tau} &: C_{\tau,\tau[t_1/res]} \times C_{\tau,\tau[t_2/res]} \mapsto C_{\tau,\tau'}\\
&\quad \text{with } \tau,\tau' \in \mathit{Typing},\ res \notin \mathsf{dom}(\tau),\ t_1,t_2 \in \mathit{Type},\\
&\quad \text{and } bop_{\tau[t_1/res]} : \Sigma_{\tau[t_1/res]} \mapsto (\Sigma_{\tau[t_2/res]} \to \Sigma_{\tau'})\\
\bowtie_\tau (f(v_1,\ldots,v_{\#p(f)-1})) &: \mathcal{I} \mapsto C_{\tau,\tau}, \text{ with } \tau \in \mathit{Typing},\ \{v_1,\ldots,v_{\#p(f)-1}\} \subseteq \mathsf{dom}(\tau)
\end{aligned}
$$

$$
[op]_{\tau,\tau'}(\sigma) = \begin{cases} \sigma \to op(\sigma) & \text{if } op(\sigma) \text{ is defined}\\ \sigma \to \tilde{\sigma} & \text{otherwise} \end{cases}
$$

$$
?_\tau(E,S_1,S_2)(\sigma) = \begin{cases} E(\sigma) \qquad \text{if } \mathsf{div}(E(\sigma))\\ E(\sigma) \to S_1(\llcorner \mathsf{lst}(E(\sigma)) \lrcorner)\\ \quad \text{if } \neg\,\mathsf{div}(E(\sigma)) \text{ and } \mathsf{is_true}_{\tau[int/res]}(\mathsf{lst}(E(\sigma)))\\ E(\sigma) \to S_2(\llcorner \mathsf{lst}(E(\sigma)) \lrcorner)\\ \quad \text{if } \neg\,\mathsf{div}(E(\sigma)) \text{ and } \mathsf{is_false}_{\tau[int/res]}(\mathsf{lst}(E(\sigma))) \end{cases}
$$

$$
(S_1 \otimes_{\tau,\tau',\tau''} S_2)(\sigma) = \begin{cases} S_1(\sigma) & \text{if } \mathsf{div}(S_1(\sigma))\\ S_1(\sigma) \to S_2(\mathsf{lst}(S_1(\sigma))) & \text{otherwise.} \end{cases}
$$

$$
(S_1 \otimes_{bop,\tau} S_2)(\sigma) = \begin{cases} S_1(\sigma) \qquad \text{if } \mathsf{div}(S_1(\sigma))\\ S_1(\sigma) \to S_2(\llcorner l_1 \lrcorner) \qquad \text{if } \neg\,\mathsf{div}(S_1(\sigma)) \text{ and } \mathsf{div}(S_2(\llcorner l_1 \lrcorner))\\ S_1(\sigma) \to S_2(\llcorner l_1 \lrcorner) \to \tilde{l}_2\\ \quad \text{if } \neg\,\mathsf{div}(S_1(\sigma)),\ \neg\,\mathsf{div}(S_2(\llcorner l_1 \lrcorner))\\ \quad \text{and } bop_{\tau[t_1/res]}(l_1)(l_2) \text{ is undefined}\\ S_1(\sigma) \to S_2(\llcorner l_1 \lrcorner) \to bop_{\tau[t_1/res]}(l_1)(l_2) \qquad \text{otherwise.} \end{cases}
$$

$$
\text{where } l_1 = \mathsf{lst}(S_1(\sigma)) \text{ and } l_2 = \mathsf{lst}(S_2(\llcorner l_1 \lrcorner))
$$

$$
\bowtie_\tau (f(v_1,\ldots,v_n))(I)(\sigma) = \begin{cases} \sigma \to i & \text{if } \mathsf{div}(i)\\ \sigma \to i \to \mathsf{unscope}^f_\tau(\sigma)(\mathsf{lst}(i)) & \text{otherwise} \end{cases}
$$

$$
\text{where } i = I(f)(\mathsf{scope}^{f,v_1,\ldots,v_n}_\tau(\sigma)).
$$

Figure 2. The signature and the implementation of the semantic operations.

and $\otimes_{bop}$ join two denotations S_1 and S_2. Divergent traces in S_1 are not joined, since they represent an incomplete computation. Moreover, $\otimes_{bop}$ applies a binary operation *bop* to the final states of S_1 and S_2 ($\llcorner\lrcorner$ removes res from the final states of S_1). The operation $\bowtie$ calls a function by using an interpretation.

Example 3. Assume that τ is such that Σ_τ contains exactly three distinct states σ_1, σ_2 and σ_3. Consider $S_1, S_2 \in C_{\tau,\tau}$ such that

$$
\begin{aligned}
S_1(\sigma_1) &= \sigma_1 \to \sigma_2 \to^{l_1} \sigma_3 & S_2(\sigma_1) &= \sigma_1 \to \tilde{\sigma}_2\\
S_1(\sigma_2) &= \sigma_2 \to^{l_2} \tilde{\sigma}_1 & S_2(\sigma_2) &= \sigma_2 \to^{l_3} \sigma_3 \to^{l_2} \tilde{\sigma}_1\\
S_1(\sigma_3) &= \sigma_3 \to^{l_2} \sigma_1 \to^{l_3} \sigma_3 & S_2(\sigma_3) &= \sigma_3 \to^{l_1} \tilde{\sigma}_1\ .
\end{aligned}
$$

Let $S = S_1 \otimes S_2$. We have

$$
\begin{aligned}
S(\sigma_1) &= \sigma_1 \to \sigma_2 \to^{l_1} \sigma_3 \to \sigma_3 \to^{l_1} \tilde{\sigma}_1 & S(\sigma_2) &= \sigma_2 \to^{l_2} \tilde{\sigma}_1\\
S(\sigma_3) &= \sigma_3 \to^{l_2} \sigma_1 \to^{l_3} \sigma_3 \to \sigma_3 \to^{l_1} \tilde{\sigma}_1\ .
\end{aligned}
$$

$$
\begin{array}{ll}
\mathcal{E}_\tau[\![i]\!]I = [\mathsf{get_int}^i_\tau] & \mathcal{E}_\tau[\![e_1 = e_2]\!]I = \mathcal{E}_\tau[\![e_1]\!]I \otimes_= \mathcal{E}_\tau[\![e_2]\!]I \\
\mathcal{E}_\tau[\![v]\!]I = [\mathsf{get_var}^v_\tau] & \mathcal{E}_\tau[\![e_1 + e_2]\!]I = \mathcal{E}_\tau[\![e_1]\!]I \otimes_+ \mathcal{E}_\tau[\![e_2]\!]I \\
& \mathcal{E}_\tau[\![f(v_1,\ldots,v_n)]\!]I = \bowtie_\tau (f(v_1,\ldots,v_n))(I) \ .
\end{array}
$$

$$
\begin{array}{c}
\mathcal{C}_\tau[\![v := e]\!]I = \mathcal{E}_\tau[\![e]\!]I \otimes [\mathsf{put_var}^v_{\tau^e}] \qquad \mathcal{C}_\tau[\![c_1; c_2]\!]I = \mathcal{C}_\tau[\![c_1]\!]I \otimes \mathcal{C}_\tau[\![c_2]\!]I \\
\mathcal{C}_\tau[\![\mathtt{let}\ v{:}t\ \mathtt{in}\ c]\!]I = [\mathsf{expand}^{v:t}_\tau] \otimes \mathcal{C}_{\tau[t/v]}[\![c]\!]I \otimes [\mathsf{restrict}^v_{\tau[t/v]}] \\
\mathcal{C}_\tau[\![\mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2]\!]I = ?(\mathcal{E}_\tau[\![e]\!]I, \mathcal{C}_\tau[\![c_1]\!]I, \mathcal{C}_\tau[\![c_2]\!]I) \\
\mathcal{C}_\tau[\![\mathtt{while}\ e\ \mathtt{do}\ c]\!]I = \mathsf{lfp}_{C_{\tau,\tau}}\ \lambda fix.?(\mathcal{E}_\tau[\![e]\!]I, \mathcal{C}_\tau[\![c]\!]I \otimes fix, [\mathsf{nop}_\tau]) \ .
\end{array}
$$

Figure 3. The rules of our denotational trace semantics.

By using the above operations, we build a denotational semantics for our language. The map $\mathcal{E}_\tau[\![e]\!] : I \mapsto C_{\tau,\tau^e}$ is shown in Figure 3 (for τ^e, see Definition 4). The basic cases of the denotation of an expression are immediate. For cases like e_1 *bop* e_2, the denotations of the two expressions are joined through $\otimes_{bop}$. For function call, we use $\bowtie$. The map $\mathcal{C}_\tau[\![]\!] : \mathcal{C} \times \mathcal{I} \mapsto C_{\tau,\tau}$ is shown in Figure 3. The denotation of an assignment applies $\mathsf{put_var}$ to the final environments of the denotation of the right hand side. The introduction of a local variable v evaluates the code in a state expanded with v. Conditionals are modelled through the ? operation. A `while` command is denoted by a least fixpoint over a conditional [16]. It is well-defined since both $\mathcal{C}[\![]\!]$ and $\mathcal{E}[\![]\!]$ are monotonic (Proposition 2), and because of Proposition 1. It is the least upper bound of an ascending transfinite chain which starts from $\bot_{C_{\tau,\tau}}$.

The semantics of a program is a least fixpoint defined through $\mathcal{C}[\![]\!]$ [16]. Namely, for every $f \in \mathcal{F}$ we initialise (expand) the variable f, we compute the denotation of its code and we remove all the variables except f.

Definition 7. *By Props. 1 and 2, the semantics of* $P = \langle \mathcal{F}, c, p \rangle$ *is defined as* $\mathcal{S}_P = \bigsqcup_o I^o_P$ *where, letting* $\tau = p(f).\tau$*,* $f \in \mathcal{F}$*,* i *finite ordinal and* l *limit ordinal,*

$$
I^0_P(f) = \bot_{C_{\tau|_{-f},\tau|_f}},\ I^{i+1}_P(f) = [\mathsf{expand}^{f:\tau(f)}_{\tau|_{-f}}] \otimes \mathcal{C}_\tau[\![c(f)]\!]I^i_P \otimes [\mathsf{restrict}^{\mathsf{dom}(\tau)\setminus f}_\tau],\ I^l_P(f) = \bigsqcup\nolimits_{m<l} I^m_P(f) \ .
$$

5 Watchpoint Semantics

We specify a program point of interest (a *watchpoint*) through the command $\mathtt{watchpoint}(l)$, with $l \in \mathit{Label}$. We expand the rules in Figure 3 with

$$
\mathcal{C}_\tau[\![\mathtt{watchpoint}(l)]\!]I = [\mathit{watch}]_{\tau,l} \ .
$$

For $\tau \in \mathit{Typing}$ and $l \in \mathit{Label}$, $[\mathit{watch}]_{\tau,l} \in C_{\tau,\tau}$ creates a $\rightarrow^l$ transition, i.e.,

$$
[\mathit{watch}]_{\tau,l}(\sigma) = \sigma \rightarrow^l \sigma \ . \tag{2}
$$

Note that the typing τ_l in a watchpoint l is statically known.

Assume we are not interested in the states before an unnamed transition, but only in those before $\rightarrow^l$ with $l \in Label$, which can be selected through a map

$$w : \mathcal{T} \mapsto (Label \mapsto \wp(\Sigma))$$

(for its explicit definition, see Definition 9) pointwise extended to denotations (Eq. (1)) and interpretations (Def. 6). Instead of computing $w(\mathcal{S}_P)$ (Def. 7), we want to *push* w *inside* the semantics, i.e., compute the abstract *watchpoint semantics induced* by the abstraction w.

5.1 Why a New Semantics

Given $t \in \mathcal{T}$, by definition $w(t)$ is more *abstract* than t, and requires less space (memory) to be stored. Let $\varsigma(x)$ be the space needed to store x. Since we are particularly interested in the case when the program to be analysed is huge and the number of watchpoints is relatively small, we can assume that

$$\varsigma(w(t)) \ll \varsigma(t) \tag{3}$$

for $t \in \mathcal{T}$. Assume we want to compute $w(t)$ where t is a trace for the command $c_1; c_2$, i.e., the concatenation of a trace t_1 generated by c_1 and a trace t_2 generated by c_2 from $\mathsf{lst}(t_1)$. We can compute t_1, abstract it in $w(t_1)$, compute t_2, abstract it in $w(t_2)$ and *merge* $w(t_1)$ and $w(t_2)$ into $w(t)$. For this we need *at most* $m_1 = \max\{\varsigma(t_1)+\varsigma(w(t_1)), \varsigma(w(t_1))+\varsigma(w(t_2))+\varsigma(t_2)\}$ space (we never hold both t_1 and t_2 in memory at the same time). Instead, if we compute t and *then* $w(t)$, we need *at least* $m_2 = \varsigma(t) + \varsigma(w(t)) = \varsigma(t_1) + \varsigma(t_2) + \varsigma(w(t_1)) + \varsigma(w(t_2))$ space. Since $m_1 \leq m_2$, *pushing* w *inside* the semantics induces a lighter calculation.

This claim does not work for the time of the analysis, since a state depends on its predecessors in a trace. Hence all states must be considered during the computation of the semantics, not just those before a $\rightarrow^l$ transition. But the watchpoint semantics reduces the cost in time of the analysis for other reasons.

1. A more abstract fixpoint computation might require fewer iterations, and hence less space and time. Section 7 shows that this is very often the case.
2. Consider `while` e `do` c, where c contains some watchpoints, c is denoted by d and e by d'. If we *unfold* d after d' until the fixpoint (Fig. 3), we then need to *scan* a trace looking for the $\rightarrow^l$ transitions. If, instead, we had a denotation $w(d)$ such that $w(d)(\sigma) = w(d(\sigma))$, we could just merge, during the fixpoint calculation, the states for the same watchpoints, without scanning any trace.
3. Dealing with smaller data structures (as shown before) leads in general to faster analyses. From Equation (3), this could mean sometimes that virtual memory is not needed by the analyser, i.e., swapping is avoided.
4. Analyses based on a trace semantics use *widening* to avoid dealing with infinite traces. For instance, [4] and [15] use regular trees, which add complexity to the analyser. A watchpoint semantics does not need such a widening.

Consider how the analysis scales with the number of watchpoints. Of course, fewer watchpoints means lighter data structures, i.e., less space requirements.

W.r.t. time, fewer watchpoints means faster analysis for points 1 and 3 above. Finally since, for every watchpoint l, we need to compute the union (join) of the states before a $\rightarrow^l$ transition, it even means fewer joins, i.e., a faster analysis.

These considerations have been experimentally verified in Section 7.

5.2 The Semantics

We define here in detail the watchpoint semantics. To observe the states in the watchpoints, we can abstract the traces in sets of states, one for each watchpoint. But this abstraction induces too coarse optimal abstract operations, since the $\otimes$ operation (Figure 2) joins the traces through their last state. Thus, for better precision, we abstract the traces in *watchpoint traces*, i.e., a set of states for every watchpoint, collected into an element of $\mathcal{W}^w$, and a set for the final states.

Definition 8. *Let*

$$\mathcal{W}^w = \{w \in Label \mapsto \wp(\Sigma) \mid given\ l \in Label\ we\ have\ w(l) \in \wp(\Sigma_{\tau_l})\}\ .$$

A $w \in \mathcal{W}^w$ is finite *if $w(l)$ is finite for every $l \in Label$. The set $\mathcal{W}^w$ is a complete lattice ordered w.r.t. the pointwise extension of $\subseteq$. Lub and glb are (pointwise) $\cup$ and $\cap$, its bottom is $\bot = \lambda l \in Label.\emptyset$.*

The set of watchpoint traces *is $\mathcal{T}^w = \cup_{\tau \in Typing} \mathcal{T}^w_\tau$ where, for $\tau \in Typing$,*

$$\mathcal{T}^w_\tau = \{\langle w, s\rangle \mid w \in \mathcal{W}^w,\ \ s \in \Sigma_\tau \cup \{\sim\}\ and\ if\ s \neq \sim\ then\ w\ is\ finite\}\ .$$

They are ordered as $\langle w, s\rangle \sqsubseteq^w \langle w, s\rangle$ and $\langle w, \sim\rangle \sqsubseteq^w \langle w', s\rangle$ if and only if $w \subseteq w'$.

Elements of $\mathcal{W}^w$ are extentionally represented as $[l_1 \mapsto \Sigma_1, \ldots, l_n \mapsto \Sigma_n]$, meaning that the label l_i is mapped to the set of states Σ_i for $i = 1, \ldots, n$. If a label is not contained in that enumeration, it is assumed that it is mapped to $\emptyset$.

Example 4. We have

$$\langle [l_1 \mapsto \{\sigma_3\}, l_3 \mapsto \{\sigma_2\}], \sim\rangle \sqsubseteq^w \langle [l_1 \mapsto \{\sigma_1, \sigma_3\}, l_2 \mapsto \{\sigma_2\},] l_3 \mapsto \{\sigma_2, \sigma_3\}], \sigma_1\rangle\ .$$

A watchpoint trace $\langle w, s\rangle$ with $s \neq \sim$ represents all convergent traces which end with s and contain exactly the watchpoints in w. If $s = \sim$, instead, it represents all divergent traces which contain exactly the watchpoints in w.

Definition 9. *Given $t \in \mathcal{T}$, we define $w(t) \in \mathcal{W}^w$ and $\alpha^w : \mathcal{T} \mapsto \mathcal{T}^w$ as*

$$w(t)(l) = \{\sigma \mid \sigma \in_l t\} \qquad \textit{for every } l \in Label.$$

$$\alpha^w(t) = \begin{cases} \langle w(t), \sim\rangle & \textit{if } \mathsf{div}(t) \\ \langle w(t), \mathsf{lst}(t)\rangle & \textit{otherwise.} \end{cases}$$

Example 5. Let $Label = \{l_1, l_2\}$. Then

$$\alpha^w(\sigma_1 \rightarrow \sigma_2 \rightarrow^{l_1} \sigma_3 \rightarrow \sigma_4 \rightarrow^{l_2} \sigma_5 \rightarrow^{l_1} \sigma_6) = \langle [l_1 \mapsto \{\sigma_2, \sigma_5\}, l_2 \mapsto \{\sigma_4\}], \sigma_6\rangle$$
$$\alpha^w(\sigma_1 \rightarrow \sigma_2 \rightarrow^{l_1} \sigma_3 \rightarrow \sigma_4 \rightarrow^{l_2} \sigma_5 \rightarrow^{l_1} \tilde{\sigma}_6) = \langle [l_1 \mapsto \{\sigma_2, \sigma_5\}, l_2 \mapsto \{\sigma_4\}], \sim\rangle\ .$$

$$[op]^w(\sigma) = \begin{cases} \langle \bot, op(\sigma) \rangle & \text{if } op(\sigma) \text{ is defined} \\ \langle \bot, \sim \rangle & \text{otherwise,} \end{cases} \qquad [watch]_l^w(\sigma) = \langle \bot[\{\sigma\}/l], \sigma \rangle$$

$$?^w(E, S_1, S_2)(\sigma) = \begin{cases} E(\sigma) \qquad \text{if } E(\sigma).s = \sim \\ \langle E(\sigma).w \cup S_1(\llcorner E(\sigma).s \lrcorner).w, S_1(\llcorner E(\sigma).s \lrcorner).s \rangle \\ \quad \text{if } E(\sigma).s \neq \sim \text{ and } \mathsf{is_true}_{\tau[int/res]}(E(\sigma).s) \\ \langle E(\sigma).w \cup S_2(\llcorner E(\sigma).s \lrcorner).w, S_2(\llcorner E(\sigma).s \lrcorner).s \rangle \\ \quad \text{if } E(\sigma).s \neq \sim \text{ and } \mathsf{is_false}_{\tau[int/res]}(E(\sigma).s) \end{cases}$$

$$(S_1 \otimes^w S_2)(\sigma) = \begin{cases} S_1(\sigma) & \text{if } S_1(\sigma).s = \sim \\ \langle S_1(\sigma).w \cup S_2(S_1(\sigma).s).w, S_2(S_1(\sigma).s).s \rangle & \text{otherwise.} \end{cases}$$

$$(S_1 \otimes_{bop}^w S_2)(\sigma) = \begin{cases} S_1(\sigma) \qquad \text{if } S_1(\sigma).s = \sim \\ \langle S_1(\sigma).w \cup S_2(\llcorner S_1(\sigma).s \lrcorner).w, \sim \rangle \\ \quad \text{if } S_1(\sigma).s \neq \sim \text{ and } (S_2(\llcorner S_1(\sigma).s \lrcorner).s = \sim \text{ or } b \text{ is undefined}) \\ \langle S_1(\sigma).w \cup S_2(\llcorner S_1(\sigma).s \lrcorner).w, b \rangle \qquad \text{otherwise,} \end{cases}$$

$$\text{with } b = bop_{\tau[t_1/res]}(S_1(\sigma).s)(S_2(\llcorner S_1(\sigma).s \lrcorner).s)$$

$$\bowtie^w (f(v_1, \ldots, v_n))(I)(\sigma) = \begin{cases} \langle i.w, \sim \rangle & \text{if } i.s = \sim \\ \langle i.w, \mathsf{unscope}^f(\sigma)(i.s) \rangle & \text{otherwise,} \end{cases} \quad i = I(f)(\mathsf{scope}^{f,v_1,\ldots,v_n}(\sigma)) \ .$$

Figure 4. The operations on watchpoint traces.

We define now the abstract counterpart of the set $C_{\tau,\tau'}$ of Equation (1).

Proposition 3. *Let $\tau, \tau' \in Typing$ and $W_{\tau,\tau'} = \Sigma_\tau \mapsto \mathcal{T}^w_{\tau'}$. The $\sqsubseteq^w$ order (α^w) is pointwise extended to $W_{\tau,\tau'}$ ($C_{\tau,\tau'}$). The pair $\langle W_{\tau,\tau'}, \sqsubseteq^w \rangle$ is a pointed CPO with bottom $\lambda\sigma \in \Sigma_\tau.\langle \bot, \sim \rangle$, and α^w is well-defined, onto, strict and additive.*

Proposition 4. *The operations in Fig. 4, whose signatures are the α^w-abstraction of those in Fig. 2, are monotonic and α^w-optimal w.r.t. those in Fig. 2.*

Example 6. Consider the concrete denotations of Example 3. Let $S_1^w = \alpha^w(S_1)$, $S_2^w = \alpha^w(S_2)$ and $S^w = \alpha^w(S)$. We have

$$\begin{array}{ll} S_1^w(\sigma_1) = \langle [l_1 \mapsto \{\sigma_2\}], \sigma_3 \rangle & S_2^w(\sigma_1) = \langle \bot, \sim \rangle \\ S_1^w(\sigma_2) = \langle [l_2 \mapsto \{\sigma_2\}], \sim \rangle & S_2^w(\sigma_2) = \langle [l_2 \mapsto \{\sigma_3\}, l_3 \mapsto \{\sigma_2\}], \sim \rangle \\ S_1^w(\sigma_3) = \langle [l_2 \mapsto \{\sigma_3\}, l_3 \mapsto \{\sigma_1\}], \sigma_3 \rangle & S_2^w(\sigma_3) = \langle [l_1 \mapsto \{\sigma_3\}], \sim \rangle \ . \end{array}$$

Moreover, we have that S^w is

$$\begin{array}{ll} S^w(\sigma_1) = \langle [l_1 \mapsto \{\sigma_2, \sigma_3\}], \sim \rangle & S^w(\sigma_2) = \langle [l_2 \mapsto \{\sigma_2\}], \sim \rangle \\ S^w(\sigma_3) = \langle [l_1 \mapsto \{\sigma_3\}, l_2 \mapsto \{\sigma_3\}, l_3 \mapsto \{\sigma_1\}], \sim \rangle \ , & \end{array}$$

which is *exatcly* $S_1^w \otimes^w S_2^w$.

Like in Section 4, we define a *watchpoint semantics* $\mathcal{S}_P^w$. By Propositions 3 and 4, it computes the same information about watchpoints as our trace semantics, i.e.,

$$\alpha^w(\mathcal{S}_P) = \mathcal{S}_P^w \ . \tag{4}$$

$$
\begin{aligned}
[op]^{co}(\eta) &= \langle \bot, op(\eta) \rangle \qquad [watch]^{co}_l(\eta) = \langle \bot[\eta/l], \eta \rangle \\
?^{co}(E, S_1, S_2)(\eta) &= \langle E(\eta).w \cup S_1(\llcorner \eta_t \lrcorner).w \cup S_2(\llcorner \eta_f \lrcorner).w, S_1(\llcorner \eta_t \lrcorner).\eta \cup S_2(\llcorner \eta_f \lrcorner).\eta \rangle \\
&\qquad \text{where } \eta_t = \mathsf{is_true}_{\tau[int/res]}(E(\eta).\eta) \text{ and } \eta_f = \mathsf{is_false}_{\tau[int/res]}(E(\eta).\eta) \\
(S_1 \otimes^{co} S_2)(\eta) &= \langle S_1(\eta).w \cup S_2(S_1(\eta).\eta).w, S_2(S_1(\eta).\eta).\eta \rangle \\
(S_1 \otimes^{co}_{bop} S_2)(\eta) &= \langle S_1(\eta).w \cup S_2(\llcorner S_1(\eta).\eta \lrcorner).w, bop_{\tau[t_1/res]}(S_1(\eta).\eta)(S_2(\llcorner S_1(\eta).\eta \lrcorner).\eta) \rangle \\
\bowtie^{co} (f(v_1, \ldots, v_n))(I)(\eta) &= \langle i.w, \mathsf{unscope}^f(\eta)(i.\eta) \rangle, \quad \text{where } i = I(f)(\mathsf{scope}^{f,v_1,\ldots,v_n}(\eta)).
\end{aligned}
$$

Figure 5. The operations on collecting watchpoint traces.

The *collecting* or *static* semantics [6] $\mathcal{S}_P^{\wp(w)}$ is the powerset lifting of $\mathcal{S}_P^w$. It works over $\wp(W_{\tau,\tau'})$, i.e., it models properties of watchpoint denotations. Since we are interested in properties of states, we define below a semantics $\mathcal{S}_P^{co}$ which works over watchpoint traces of sets of states. It is an AI of $\mathcal{S}_P^{\wp(w)}$ and will be called *collecting* though, strictly speaking, the real collecting semantics is $\mathcal{S}_P^{\wp(w)}$.

Definition 10. *The set of* collecting watchpoint traces $\mathcal{T}^{co} = \cup_{\tau \in Typing} \mathcal{T}^{co}_\tau$ *where, letting* $\tau \in Typing$ *and* $\mathcal{W}^{co} = \mathcal{W}^w$ *(Definition 8)* $\mathcal{T}^{co}_\tau = \{\langle w, \eta \rangle \mid w \in \mathcal{W}^{co}$ *and* $\eta \in \wp(\Sigma_\tau)\}$*, is ordered as* $\langle w_1, \eta_1 \rangle \sqsubseteq^{co} \langle w_2, \eta_2 \rangle$ *iff* $w_1 \subseteq w_2$ *and* $\eta_1 \subseteq \eta_2$*.*

Denotations are identified by their values on singleton sets. Denotations with more than one argument will be useful at the end of this section. This is formalised below.

Proposition 5. *Let* $n \geq 1$ *and* $\tau_1, \ldots, \tau_n, \tau' \in Typing$*. Let* $CO_{\tau_1,\ldots,\tau_n,\tau'}$ *be*

$$
\left\{ \begin{array}{c} co \in \wp(\Sigma_{\tau_1}) \mapsto \cdots \\ \cdots \mapsto \wp(\Sigma_{\tau_n}) \mapsto \mathcal{T}^{co}_{\tau'} \end{array} \;\middle|\; \begin{array}{c} co(\eta_1)\cdots(\eta_n) = \langle \bigcup_{\sigma_1 \in \eta_1, \ldots, \sigma_n \in \eta_n} co(\{\sigma_1\})\cdots(\{\sigma_n\}).w, \\ \bigcup_{\sigma_1 \in \eta_1, \ldots, \sigma_n \in \eta_n} co(\{\sigma_1\})\cdots(\{\sigma_n\}).\eta \rangle \end{array} \right\} . \quad (5)
$$

The $\sqsubseteq^{co}$ *order of Definition 10 is pointwise extended to denotations* CO*. The pair* $\langle CO_{\tau_1,\ldots,\tau_n,\tau'}, \sqsubseteq^{co} \rangle$ *is a complete lattice with bottom* $\lambda\eta_1 \cdots \lambda\eta_n.\langle \bot, \emptyset \rangle$*.*

A collecting watchpoint trace represents a set of watchpoint traces. This abstraction induces optimal abstract counterparts of the operations in Figure 4.

Proposition 6. *Let* $\alpha^{co} : \wp(\mathcal{T}^w) \mapsto \mathcal{T}^{co}$ *be* $\alpha^{co}(S) = \langle \cup_{t \in S} t.w, \{t.s \mid t \in S$ *and* $t.s \neq \sim\}\rangle$*. Its extension* $\alpha^{co} : \wp(W_{\tau,\tau'}) \mapsto CO_{\tau,\tau'}$*, for* $\tau, \tau' \in Typing$*, given by* $(\alpha^{co}(W))(\eta) = \alpha^{co}(\{w(\sigma) \mid w \in W$ *and* $\sigma \in \eta\})$ *for* $\eta \in \wp(\Sigma_\tau)$*, is well-defined, onto, strict and additive (hence, the abstraction of a Galois insertion).*

Proposition 7. *The operations in Figure 5 are monotonic and* α^{co}*-optimal w.r.t. the pointwise extension of those in Figure 4.*

We define a *collecting watchpoint semantics* $\mathcal{S}_P^{co}$. We have $\mathcal{S}_P^{co} = \alpha^{co}(\mathcal{S}_P^{\wp(w)})$.

The operations in Figure 5 use objects in $\wp(\Sigma)$ (like $S_1(\eta).\eta$ in $\otimes^{co}$), $\mathcal{W}^{co}$ (like $S_1(\eta).w$ in $\otimes^{co}$) and CO (like S_1 in $\otimes^{co}$). To simplify the abstraction of Section 6, we *compile* them in terms of smaller operations over CO only, given in Figure 6. The compilation is shown in Figure 7.

Proposition 8. *The operations in Figure 5 and those in Figure 7 are the same.*

$$[op] : CO_{\tau_1,\cdots,\tau_n,\tau'}, \text{ if } op : \Sigma_{\tau_1} \mapsto \cdots \Sigma_{\tau_n} \mapsto \Sigma_{\tau'} \qquad [watch]_l : CO_{\tau,\tau}$$
$$.w : CO_{\tau,\tau'} \mapsto CO_{\tau,\tau'}$$
$$\circ : CO_{\tau_1,\ldots,\tau_n,\tau'} \times CO_{\tau,\tau_1} \cdots \times CO_{\tau,\tau_n} \mapsto CO_{\tau,\tau'} \qquad \cup : CO^2_{\tau,\tau'} \mapsto CO_{\tau,\tau'}$$

$$[op] = \lambda\eta_1 \ldots \lambda\eta_n.\langle \bot, op(\eta_1,\cdots,\eta_n)\rangle \qquad [watch]_l = \lambda\eta.\langle \bot[\eta/l], \eta\rangle$$
$$T.w = \lambda\eta.\langle T(\eta).w, \emptyset\rangle \qquad T \circ (T_1,\ldots,T_n) = \lambda\eta.T(T_1(\eta).\eta)\cdots(T_n(\eta).\eta)$$
$$T_1 \cup T_2 = \lambda\eta.\left\langle \begin{array}{c} T_1(\eta).w \cup T_2(\eta).w, \\ T_1(\eta).\eta \cup T_2(\eta).\eta \end{array} \right\rangle .$$

Figure 6. A minimal set of operations over CO.

$$\llcorner T \lrcorner = [\mathsf{restrict}^{res}] \circ T , \qquad [op]^{co} = [op] , \qquad [watch]^{co}_l = [watch]_l$$
$$?^{co}(E, S_1, S_2) = E.w \cup (S_1 \circ \llcorner T_t \lrcorner) \cup (S_2 \circ \llcorner T_f \lrcorner)$$
$$\text{where } T_t = [\mathsf{is_true}_{\tau[int/res]}] \circ E \text{ and } T_f = [\mathsf{is_false}_{\tau[int/res]}] \circ E$$
$$S_1 \otimes^{co} S_2 = S_1.w \cup (S_2 \circ S_1)$$
$$S_1 \otimes^{co}_{bop} S_2 = S_1.w \cup (S_2 \circ \llcorner S_1 \lrcorner).w \cup [bop_{\tau[t_1/res]}] \circ (S_1, S_2 \circ \llcorner S_1 \lrcorner)$$
$$\bowtie^{co} (f(v_1,\ldots,v_n))(I) = T_i.w \cup [\mathsf{unscope}^f] \circ ([\mathsf{nop}], T_i) \qquad \text{where } T_i = I(f) \circ [\mathsf{scope}^{f,v_1,\ldots,v_n}].$$

Figure 7. The operations in Figure 5 in terms of those in Figure 6.

6 From Abstract Domains to Abstract Semantics

We show here how every abstraction of the domain of states (Definition 3) induces an abstraction of the denotations CO (Equation (5)) and of their operations (Figure 6) and hence of the collecting watchpoint semantics of last section. This reduces the definition of a static analysis to the definition of abstract states.

Every abstract denotational semantics works over abstract denotations which are maps from abstract inputs to abstract outputs. In our watchpoint semantics, the abstract outputs are actually abstract traces. The problem here is how to define the concretisation of such abstract denotations. If a concrete state belongs to two abstract inputs, how should it behave in the concretisation? We do not consider this problem in details here, since it has already been studied in a more general setting. Consider for instance the *functional partitioning* technique in [1]. Instead, we assume here that in the lattice of abstract states there exists a set of *union-irreducible* states in terms of which all other abstract states can be expressed. This condition holds for the case of sign analysis shown in Section 7.

For $\tau \in Typing$, let $\langle D_\tau, \sqsubseteq\rangle$ be a complete lattice and α^{D_τ} and γ^{D_τ} the abstraction and concretisation maps of a Galois insertion from $\langle \wp(\Sigma_\tau), \subseteq\rangle$ to $\langle D_\tau, \sqsubseteq\rangle$ (typings will be often omitted).

Definition 11. *Let* $\mathcal{W}^a = \{w \in Label \mapsto D \mid \textit{for } l \in Label \textit{ we have } w(l) \in D_{\tau_l}\}$. *The set of* abstract watchpoint traces *is* $\mathcal{T}^a = \cup_{\tau \in Typing} \mathcal{T}^a_\tau$ *where*

$$\mathcal{T}^a_\tau = \{\langle w, d\rangle \mid w \in \mathcal{W}^a \textit{ and } d \in D_\tau\} \qquad \textit{for every } \tau \in Typing,$$

$$[op]^a = \lambda d_1 \ldots \lambda d_n.\langle \bot, \alpha^D(op(\gamma^D(d_1), \ldots, \gamma^D(d_n)))\rangle \qquad [watch]^a_l = \lambda d.\langle \bot[d/l], d\rangle$$

$$T.w^a = \lambda d.\langle T(d).w, \emptyset\rangle$$

$$T \circ^a (T_1, \ldots, T_n) = \lambda d. \sqcup^a_{d' \in S} T(T_1(d')) \cdots (T_n(d')) \ , \qquad \text{with } S \in \rho(d)$$

$$T_1 \cup^a T_2 = \lambda d. \left\langle \begin{array}{c} T_1(d).w \cup^D T_2(d).w, \\ T_1(d).d \cup^D T_2(d).d \end{array} \right\rangle \qquad \cup^D \text{ is the best approximation of } \cup \text{ over } D.$$

Figure 8. The generic abstract counterparts of the operations of Figure 6.

ordered as $\langle w_1, d_1\rangle \sqsubseteq^a \langle w_2, d_2\rangle$ *if and only if* $w_1 \sqsubseteq^a w_2$ *(pointwise) and* $d_1 \sqsubseteq^a d_2$. *The map* α^D *is expanded to* $\mathcal{T}^{co}_\tau$ *as* $\alpha^D(\langle w, \eta\rangle) = \langle \lambda l.\alpha^{D_{\tau_l}}(w(l)), \alpha^{D_\tau}(\eta)\rangle$.

Definition 12. *Let* $\tau \in Typing$. *The* union-reductions *of* $d \in D_\tau$ *are*

$$\rho(d) = \{S \in \wp(D_\tau) \mid \gamma^D(d) = \cup_{d' \in S}\gamma^D(d') \text{ and } \#\rho(d') = 1 \text{ for every } d' \in S\} \ .$$

If $\#\rho(d) = 1$ *(i.e.,* $\rho(d) = \{\{d\}\}$*) we say that* d *is* union-irreducible. *If every* $d \in D_\tau$ *is such that* $\rho(d) \neq \emptyset$, *we say that* D_τ *is* union-reducible.

Proposition 9. *Assume that* D_τ *is union-reducible for every* $\tau \in Typing$. *Given* $n \geq 1$ *and* $\tau_1, \ldots, \tau_n, \tau' \in Typing$, *let* $A_{\tau_1,\ldots,\tau_n,\tau'}$ *be*

$$\left\{ a \in D_{\tau_1} \mapsto \cdots D_{\tau_n} \mapsto D_{\tau'} \;\middle|\; \begin{array}{l} a \text{ is monotonic and given } 1 \leq i \leq n \text{ and } S \in \rho(d_i) \\ a(d_1)\cdots(d_i)\cdots(d_n) = \bigsqcup^{D_{\tau'}}_{d' \in S} a(d_1)\cdots(d')\cdots(d_n) \end{array} \right\}, \quad (6)$$

i.e., denotations in $A_{\tau_1,\ldots,\tau_n,\tau'}$ *are identified by the union-irreducible elements. Let* $\alpha^a : CO_{\tau_1,\ldots,\tau_n,\tau'} \mapsto A_{\tau_1,\ldots,\tau_n,\tau'}$ *be* $\alpha^a(co) = \alpha^D co\gamma^D$. *The set* $A_{\tau_1,\ldots,\tau_n,\tau'}$ *is a complete lattice with bottom* $\lambda d_1 \ldots \lambda d_n.\langle \bot, \bot_{D_{\tau'}}\rangle$ *and* α^a *is well-defined, onto, strict and additive (hence, the abstraction map of a Galois insertion).*

Proposition 10. *Assume that* D_τ *is union-reducible for every* $\tau \in Typing$. *The operations in Figure 8 are the best approximations over* A *of those in Figure 6 (note that* $\circ^a$ *is not the composition of functions).*

In conclusion, given a union-reducible abstraction D_τ of $\wp(\Sigma_\tau)$ for every $\tau \in Typing$, and the best approximations over D of the powerset extension of the operations of Figure 1 (used in $[op]^a$) and of $\cup$, we obtain an abstract watchpoint semantics, correct w.r.t. the collecting watchpoint semantics of Subsection 5.2. As said before, similar results can be obtained in the more general case of non-union-reducible lattices by using the functional partitioning technique of [1].

7 Implementation

We describe here our implementation in Prolog of the watchpoint semantics of Sections 5 and 6 instantiated with sign analysis. It can be downloaded from `http://www.sci.univr.it/~spoto/watch.tar.gz`. We have chosen Prolog for fast prototyping, and sign analysis because it is a well-known, simple analysis.

$$\mathsf{nop}^s_\tau(\varsigma) = \varsigma \qquad (\mathsf{get_int}^i_\tau)^s(\varsigma) = \begin{cases} \varsigma[+/res] & \text{if } i \geq 0 \\ \varsigma[-/res] & \text{if } i < 0 \end{cases}$$

$$(\mathsf{get_var}^v_\tau)^s(\varsigma) = \varsigma[\varsigma(v)/res] \qquad (\mathsf{put_var}^v_\tau)^s(\varsigma) = \varsigma[\varsigma(res)/v]|_{-res}$$

$$=^s_\tau(\varsigma_1)(\varsigma_2) = \begin{cases} \varsigma_2[-/res] & \text{if } \varsigma_1(res) = - \\ & \text{and } \varsigma_2(res) = + \\ & \text{or vice versa} \\ \varsigma_2[u/res] & \text{otherwise} \end{cases} \quad +^s_\tau(\varsigma_1)(\varsigma_2) = \begin{cases} \varsigma_2[+/res] & \text{if } \varsigma_1(res) = \varsigma_2(res) = + \\ \varsigma_2[-/res] & \text{if } \varsigma_1(res) = \varsigma_2(res) = - \\ \varsigma_2[u/res] & \text{otherwise} \end{cases}$$

$$(\mathsf{scope}^{f,v_1,\ldots,v_n}_\tau)^s(\varsigma) = [\iota_1 \mapsto \varsigma(v_1), \ldots, \iota_n \mapsto \varsigma(v_n)] \quad \text{where } \langle \iota_1, \ldots, \iota_n \rangle = p(f).s \setminus f$$

$$(\mathsf{unscope}^f_\tau)^s(\varsigma_1)(\varsigma_2) = \varsigma_1[\varsigma_2(f)/res] \qquad (\mathsf{restrict}^{vs}_\tau)^s(\varsigma) = \varsigma|_{-vs} \qquad (\mathsf{expand}^{v:t}_\tau)^s(\varsigma) = \varsigma[+/v]$$

$$\mathsf{is_true}^s_\tau(\varsigma) = \begin{cases} \mathsf{empty} & \text{if } \varsigma(res) = - \\ \varsigma[+/res] & \text{otherwise} \end{cases} \qquad \mathsf{is_false}^s_\tau(\varsigma) = \begin{cases} \mathsf{empty} & \text{if } \varsigma(res) = + \\ \varsigma[-/res] & \text{otherwise} \end{cases}$$

$$\cup^s_\tau(\mathsf{empty})(x) = \cup^s_\tau(x)(\mathsf{empty}) = x \quad \cup^s_\tau(\varsigma_1)(\varsigma_2) = \lambda v \in \mathsf{dom}(\tau).\begin{cases} \varsigma_1(v) & \text{if } \varsigma_1(v) = \varsigma_2(v) \\ u & \text{otherwise.} \end{cases}$$

Figure 9. The abstract operations over the domain of signs.

The module `analyser.pl` implements the fixpoint calculation (Figure 3 and Definition 7) by using the semantic operations (Figures 7 and 8) implemented in the module `semantic.pl`. The module `typing.pl` manipulates typings. The module `domain.pl` implements the abstract counterparts of the operations of Figure 1. Only this module depends from the domain of analysis.

Our domain for sign analysis is similar to that in [6].

Definition 13. *For every $\tau \in Typing$, let $S_\tau = \{\mathsf{empty}\} \cup \{\varsigma : \mathsf{dom}(\tau) \mapsto \{+,-,u\}\}$. The abstraction map $\alpha : \wp(\Sigma_\tau) \mapsto S_\tau$ is such that, for $X \neq \emptyset$ and $v \in \mathsf{dom}(\tau)$,*

$$\alpha(\emptyset) = \mathsf{empty} \qquad \alpha(X)(v) = \begin{cases} + & \text{if } \sigma(v) \geq 0 \text{ for every } \sigma \in X \\ - & \text{if } \sigma(v) < 0 \text{ for every } \sigma \in X \\ u & \text{otherwise.} \end{cases}$$

Let $\leq$ be reflexive and let $+ \leq u$ and $- \leq u$. The set S_τ is ordered as $\mathsf{empty} \sqsubseteq^s s$ for every $s \in S_\tau$ and $\varsigma_1 \sqsubseteq^s \varsigma_2$ if and only if $\varsigma_1(v) \leq \varsigma_2(v)$ for every $v \in \mathsf{dom}(\tau)$. The optimal counterparts over S of the powerset extension of the operations in Fig. 1 are (all but $\cup^s$) strict on empty. Otherwise, they are given in Figure 9.

Given $\tau \in Typing$, the union-irreducible elements of S_τ are empty and those $\varsigma \in S_\tau$ such that $\varsigma(v) \neq u$ for every $v \in \mathsf{dom}(\tau)$. If $\varsigma(v) = u$ for some $v \in \mathsf{dom}(\tau)$, instead, the concretisation of ς can be shown to be the union of the concretisations of $\varsigma[+/v]$ and $\varsigma[-/v]$. Therefore, we have

$$\rho(\mathsf{empty}) = \{\mathsf{empty}\} \qquad \rho(\varsigma) = \left\{ \varsigma' \,\middle|\, \begin{array}{l} \text{for all } v \in \mathsf{dom}(\tau) \text{ we have } \varsigma'(v) \neq u \\ \text{and if } \varsigma(v) \neq u \text{ then } \varsigma(v) = \varsigma'(v) \end{array} \right\} .$$

By the results of Section 6, the abstract denotations are maps whose domain is made of empty and of all ς which never bind a variable to u. The values for the other elements of S_τ are induced.

```
F = {fib}

p(fib) = ⟨⟨fib, n⟩, [fib ↦ int, n ↦ int]⟩

c(fib) = if (n =< 1) then
           %watchpoint(p1);
           fib := 1
         else
           %watchpoint(p2);
           let n1 : int in let n2 : int in
             %watchpoint(p3);
             n1 := n − 1;
             %watchpoint(p4);
             n2 := n − 2;
             %watchpoint(p5);
             fib := fib(n1) + fib(n2);
           %watchpoint(p6)
```

(a)

```
| ? − interpret.
Analysing [fib] : iteration 1
Analysing [fib] : iteration 2
fixpoint reached

Procedure : fib

Input : empty Output : empty
Watchpoints : p3 : empty

Input : [+] Output : [+]
Watchpoints : p3 : [+, +, +, +]

Input : [−] Output : [+]
Watchpoints : p3 : empty
```

(b)

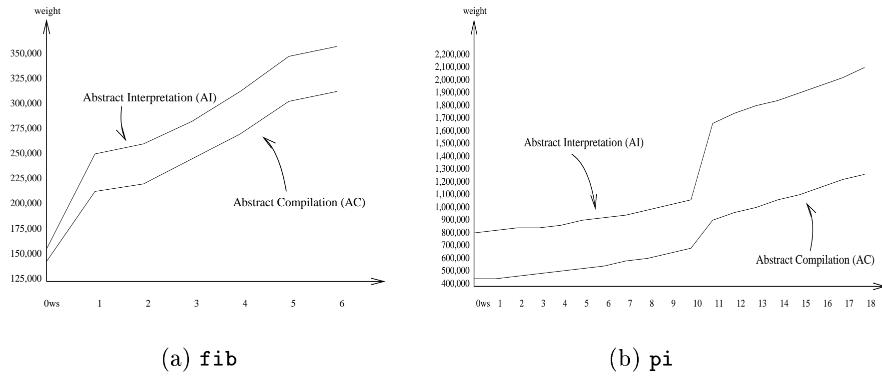
Figure 10. The Fibonacci procedure and one of its possible analyses.

Elements of S_τ are implemented as the term `empty` or lists of `+`, `-` and `u`, ordered alphabetically w.r.t. the names in $\mathsf{dom}(\tau)$. For instance, if $\tau = [a \mapsto int, c \mapsto int, b \mapsto int]$ then $\varsigma = [a \mapsto +, c \mapsto -, b \mapsto u]$ is implemented as `[+,u,-]`. We are aware of cleverer implementations, but in this paper we focus on the semantics.

The input of the analyser is a Prolog term which represents the abstract syntax of a program. Figure 10(a) shows a program for the n-th Fibonacci number, with six possible watchpoints. The file `fib.pl` contains its abstract syntax. We download it with `[fib].` and we analyse it with `interpret.` Figure 10(b) shows the result when only watchpoint p_3 is not commented. The input of `fib` is the value of `n`, its output is the value of the variable `fib` at its end. As you can see, if we start with an empty set of states we never reach watchpoint p_3. If we start with a state where `n` is positive, the output is positive and we reach watchpoint p_3 with a state where `fib`, `n`, `n1` and `n2` are positive. Indeed the initial value of a variable is 0 and in the `else` branch we have $n > 1$. Finally, if we start with a state where `n` is negative, the output is positive and watchpoint p_3 is never reached. Indeed, if $n < 0$ the `then` branch is executed. If we start with an unknown value for `n` we would obtain the least upper bound of the last two cases.

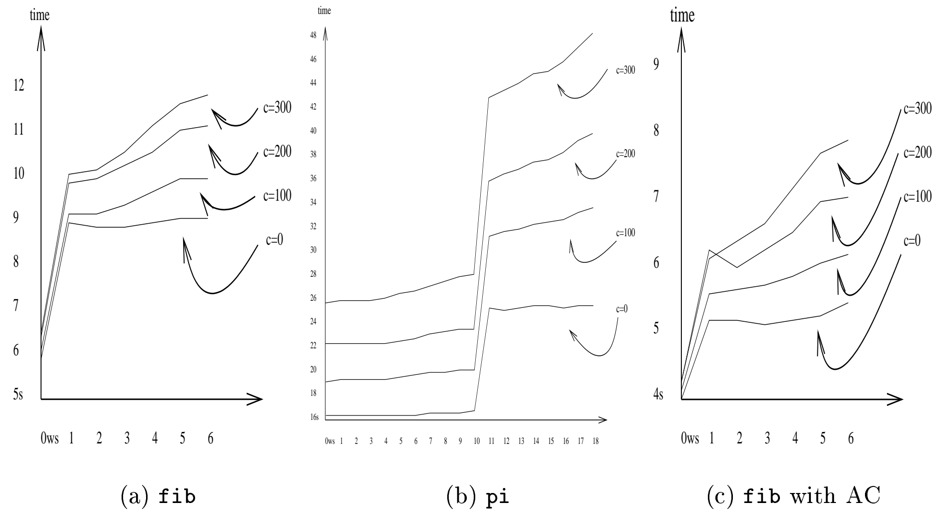
7.1 The Costs in Space and in Time of the Analysis

To estimate the space used by our analyser independently from its implementation, we count the number of Prolog atoms contained in the denotations it computes (*weight*). Fig. 11 gives the weight for the analysis of `fib` (Fig. 10(a)) and `pi` (a Monte Carlo algorithm computing π), as a function of the number of active watchpoints. For now, consider only the lines marked with *Abstract Interpretation*. Horizontally, an integer like 3 means that only watchpoints p_1, p_2 and p_3 were active. As you can see, the weight grows with the number of active watchpoints. When passing from 0 watchpoints to 1 watchpoint in Fig. 11(a)



(a) `fib` (b) `pi`

Figure 11. The cost in space of the analysis w.r.t. the number of active watchpoints.



(a) `fib` (b) `pi` (c) `fib` with AC

Figure 12. The cost in time of the analysis w.r.t. the number of active watchpoints.

and from 10 watchpoints to 11 in Fig. 11(b) one more iteration is needed to reach the fixpoint. Thus less watchpoints *does* mean less iterations (Subs. 5.1).

We expect the time of the analysis to grow with the number of watchpoints, proportionally with the cost of the abstract join (Subs. 5.1). Fig. 12 confirms this. The constant c is a fictitious cost added to the computation of the join. Note that domains more realistic than signs usually feature complex joins. Note again the jump when one more iteration is needed for the fixpoint calculation. The benchmark `nested` shows the same behaviour of Fig. 12. But if a benchmark does not contain recursive predicates nor conditionals nor iterative constructs, then the time for its analysis is independent from the number of watchpoints, like for `arith`, whose abstract execution tree is actually a finite trace.

Note that [4], [5] and [15] do not provide a link to an implementation.

Benchmark	Watchpoints	AI/AC	Iterations	Time (seconds)	Weight (atoms)
fib	$\emptyset$	AI	2	5.92	160795
fib	$\emptyset$	AC	2	4.03	142299
fib	$\{p_1, \ldots, p_6\}$	AI	3	9.19	349631
fib	$\{p_1, \ldots, p_6\}$	AC	3	5.28	305363
pi	$\emptyset$	AI	2	16.61	833419
pi	$\emptyset$	AC	2	8.91	463294
pi	$\{p_1, \ldots, p_{18}\}$	AI	3	25.93	2107643
pi	$\{p_1, \ldots, p_{18}\}$	AC	3	10.58	1287127
arith	$\emptyset$	AI	1	303.12	7049327
arith	$\emptyset$	AC	1	308.42	7049327
nested	$\emptyset$	AI	3	661.43	14253268
nested	$\emptyset$	AC	3	369.99	8419626

Figure 13. A comparison of abstract interpretation with abstract compilation.

7.2 Abstract Compilation

In Figure 10(a) we note that the denotation of the `then` branch is independent from the partial denotation computed for `fib`. Thus, it does not need to be computed at every iteration, like, instead, that of the `else` branch, which contains two calls to `fib`. However, its first part, till the watchpoint p_5, does not contain recursive calls, and can be safely analysed only once. Those optimisations are examples of abstract compilation (AC). Our analyser uses AC by invoking the goal `compile`. The result is like that in Figure 10(b), with smaller space and time costs, as Figures 11 and 12(c) show for weight (space) and time, respectively. Moreover, Figure 12(c) shows that the time still depends from the number of watchpoints and the cost of the join. Finally, Figure 13 shows that AC leads very often to major improvements, but is of no help with the *flat* benchmark `arith`.

8 Conclusions

We have shown that, if we are interested in the analysis of a program in a small set of *watchpoints*, it is worth abstracting a trace semantics in a lighter, compositional and still as precise *watchpoint semantics*. We have shown through an implementation that it is *focussed*, i.e., its complexity grows with the number of watchpoints, and that abstract compilation improves significantly the fixpoint calculation.

The analysis process is defined as a fixpoint computation. For better efficiency, if a set of call patterns is known for some functions, this computation can be done *on demand*, simulating a top-down analysis. This means that the abstract denotations are enriched at fixpoint computation time whenever the behaviour of a function for a new input is needed.

Our results apply to the modular analysis of large programs and to the analysis inside smart cards, where memory requirements must be kept small.

References

1. F. Bourdoncle. Abstract Interpretation by Dynamic Partitioning. *Journal of Functional Programming*, 2(4):407–423, 1992.

2. F. Bourdoncle. Abstract Debugging of High-Order Imperative Languages. In *Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation*, pages 46–55, Albuquerque, New Mexico, June 1993. ACM Press.
3. M. Codish and B. Demoen. Deriving Polymorphic Type Dependencies for Logic Programs Using Multiple Incarnations of Prop. In *Proc. of the first International Symposium on Static Analysis*, volume 864 of *Lecture Notes in Computer Science*, pages 281–296. Springer-Verlag, 1994.
4. C. Colby and P. Lee. Trace-based Program Analysis. In *Proc. of POPL'96*, pages 195–207, St. Petersburg, FLA, USA, January 1996. ACM Press.
5. P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. In S. Brookes and M. Mislove, editors, *Proc. of the 13th Conf. on Mathematical Foundations of Programming Semantics (MFPS XIII)*, volume 6 of *Electronic Notes on Theoretical Computer Science*, Pittsburgh, PA, USA, March 1997. Elsevier Science Publishers. Available at `http://www.elsevier.nl/locate/entcs/volume6.html`.
6. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252, 1977.
7. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *6th ACM Symp. on Principles of Programming Languages*, pages 269–282, 1979.
8. E. Duesterwald, R. Gupta, and M. Soffa. Demand-Driven Computation of Interprocedural Data Flow. In *Proceedings of POPL'95*, pages 37–48, San Francisco, CA, January 1995.
9. E. Duesterwald, R. Gupta, and M. Soffa. A Practical Framework for Demand-Driven Interprocedural Data Flow Analysis. *TOPLAS*, 19(6):992–1030, 1997.
10. M. Gabbrielli, G. Levi, and M. C. Meo. Resultants Semantics for PROLOG. *Journal of Logic and Computation*, 6(4):491–521, 1995.
11. M. Hermenegildo, W. Warren, and S.K. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(2 & 3):349–366, 1992.
12. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.
13. T. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of POPL'95*, pages 49–61, San Francisco, CA, January 1995.
14. M. Sagiv, T. Reps, and R. Wilhelm. Solving Shape-Analysis Problems in Languages with Destructive Updates. In *Proc. of POPL'96*, pages 16–31, St. Petersburg, FLA, USA, January 1996. ACM Press.
15. D. A. Schmidt. Trace-based Abstract Interpretation of Operational Semantics. *Journal of LISP and Symbolic Computation*, 10(3):237–271, 1998.
16. G. Winskel. *The Formal Semantics of Programming Languages.* The MIT Press, 1993.