Program Unrolling by Abstract Interpretation for Probabilistic Proofs

Patrick Cousot
New York University

Thursday 3rd June, 2021

1 Introduction

1.1 Context

DARPA's "Securing Information for Encrypted Verification and Evaluation" (SIEVE) wishes to enhance *zero-knowledge (ZK) proofs*, that are a protocol between a *prover* that creates a statement that it wants a *verifier* to accept, using knowledge that will remain hidden from the verifier. In the SIEVE-Pepper project, the statement is the result of the distant execution of a program. The program executions are assumed to be *bounded* (depending on the bounded size of the inputs or depending on the particular input x), so the program can be transformed into a finite Boolean-arithmetic circuit, the computation of which can be checked with probabilistic methods such that the prover can efficiently convince the verifier that

> "the program $\mathtt{P}$, when executed on this input x, produces that output y such that $y = \mathcal{S}^r[\![\mathtt{P}]\!](x)$ (with an arbitrary small probability of being unsound, where $\mathcal{S}^r[\![\mathtt{P}]\!]$ is the relational input-output semantics of the deterministic program $\mathtt{P}$)." [1] (1)

By the boundedness hypothesis, the certification technique involves a frontend which compiles the program $\mathtt{P}$ in the language P into a finite Boolean and arithmetic circuit $\mathcal{C}[\![\mathtt{P}]\!]$ (with the same semantics), by unrolling of iterations and of function and procedure calls; see [42, 39, 41].

[1] This prover/verifier terminology is somewhat confusing in the context of program verification and analysis.

1.2 Objective

The objective is to minimize the size of the generated circuit.

To be independent of the tool used to generate the circuit, we propose to unroll the source program into a more efficient source program (which therefore should yield a smaller circuit). Because the program is unrolled, its static analysis is much more precise than the one of the original program (no extrapolation by widening and interpolation by narrowing is necessary [7]). Then a transformer can optimize the unrolled program using the result of the static analysis in order to reduce the unrolled program size and the size of the data that it manipulates.

The advantage of unrolling is that the analysis and optimization of the unrolled program can be very aggressive. The inconvenient is of course the size of the unrolled program which may not be manageable by a classical analyser[2].

1.3 Content

We introduce an abstract interpretation-based methodology to formalize boundedness, semantic equivalence of the original and transformed program, and the simultaneous unrolling, static analysis, and transformation of the program by instantiation of a common abstract interpreter. We suggest a number of possible classes of optimizing transformations.

1.4 Future Work

Soundness proofs remain to be done (and maybe to be checked with a proof assistant such as Coq [24], as the style of [27]). The formal specification of the unroll, analyze, and transform algorithm in section 20 must be implemented for experimentation, using libraries of abstract domains (such as APRON [25], ELINA [37], or PPL [1, 3, 4] for numerical properties). The transformations remain to be studied in greater details, proved correct following [15], implemented, and experimented. These experimentations should determine which pairs of abstract domains and optimizations are effective for which classes of programs. Notice that the implementation can only be a lightweight academic prototype with limited scope. A professional-quality complex static analyzer like Astrée `https://www.absint.com/astree/index.htm` takes years in research and development by a dozen of researchers and engineers, not even counting the user interface.

[2] the size of the unrolled program is nevertheless smaller that the size of the generated Boolean and arithmetic circuit, which is itself a severe limitation of the present-day verifiable computation techniques

2 The Language

2.1 Syntax

We consider a small subset of the C programming language [28], as follows:

$\mathtt{x}, \mathtt{y}, \ldots$	$\in$	V				variables (V not empty)
$\mathtt{A}$	$\in$	A	$::=$	$\mathtt{1} \mid \mathtt{x} \mid \mathtt{A}_1 \mathtt{-} \mathtt{A}_2$		arithmetic expressions
$\mathtt{B}$	$\in$	B	$::=$	$\mathtt{A}_1 \mathtt{<} \mathtt{A}_2 \mid \mathtt{B}_1 \,\mathtt{nand}\, \mathtt{B}_2$		Boolean expressions
$\mathtt{E}$	$\in$	E	$::=$	$\mathtt{A} \mid \mathtt{B}$		expressions

$\mathtt{S}$	$::=$		statement $\mathtt{S} \in \mathsf{S}$
		`x = A ;`	assignment
	$\mid$	`;`	skip
	$\mid$	`if (B) S`	conditional
	$\mid$	`if (B) S else S`	
	$\mid$	`while (B) S`	iteration
	$\mid$	`break ;`	iteration break
	$\mid$	`{ Sl }`	compound statement
	$\mid$	`{| Sl |}`	breakable statement
$\mathtt{Sl}$	$::=$	$\mathtt{Sl}\ \ \mathtt{S} \mid \epsilon$	statement list $\mathtt{Sl} \in \mathsf{Sl}$, ϵ is the empty string
$\mathtt{P}$	$::=$	$\mathtt{Sl}$	program $\mathtt{P} \in \mathsf{P}$
Pc	$\triangleq$	$\mathsf{S} \cup \mathsf{Sl} \cup \mathsf{P}$	program component $\mathtt{S} \in \mathsf{Pc}$

The classical unrolling of the iteration is

$$\texttt{while (B) S} \;\equiv\; \texttt{if (B) \{ S while (B) S \}}$$

In C, the **`break ;`** statement should be included in a loop and branches out of the closest enclosing loop. So if the loop body S has a **`break ;`** statement, as in

$$\texttt{while (B) break ;} \;\equiv\; \texttt{if (B) \{ break ; while (B) break ; \}}$$

the outer **`break ;`** is no longer in a loop (or switch) statement, so that the transformed program is incorrect. To cope with this problem, we introduce breakable statements and we require that a **`break ;`** statement should be in a loop or breakable statement and that it branches out of the closest enclosing loop or breakable statement.

while (B) **break ;** $\equiv$ **if** (B) **{| break ; while** (B) **break ; |}**

Notice that **while** (B) **{|** ... **break ;** ... **|}**, the **break ;** breaks out of the breakable statement and so iterations go on. This is the effect of the `continue` statement in C.

With breakable statements, programs are no longer in the C language, but **break ;** statement out of breakable statements can be easily implemented in C by an unconditional branch.

```
if(0==0){ goto L; while(0==0)break; } L:;
```

However this transformation would loose the information that the unrolled program is still structured and would complexity the static analysis for no good reason. On the contrary, a very simple preprocessor can be written to rewrite the unrolled program in C using `goto`s.

2.2 Labelling

For discussing the semantics and correctness of programs it is necessary to introduce labelled program points.

$\ell, \ell_1, \ell', \ldots \in \mathsf{L}$ labelled program point (L denumerably infinite)

We postulate the following labelling:

at⟦S⟧	the program point at which execution of program component S starts;
after⟦S⟧	the program exit point after program component S, at which execution of S is supposed to normally terminate, if ever;
escape⟦S⟧	a Boolean indicating whether or not the program component S contains a **break ;** statement escaping out of that component S;
break-to⟦S⟧	the program point to which execution of the program component S goes when a **break ;** statement escapes out of that component S;
breaks-of⟦S⟧	the set of labels of all **break ;** statements that can escape out of component S;
in⟦S⟧	the set of program points inside program component S (including at⟦S⟧ but excluding after⟦S⟧ and break-to⟦S⟧);
labs⟦S⟧	the potentially reachable program points while executing S either in or after the statement (excluding reachability by a break) and
labx⟦S⟧	the potentially reachable program points while executing program component S at, in, or after the program component, or resulting from a break.

A formal definition is provided in [11, chapter 4]. The update to include **`break ;`** statements out of compound statements is simple:

Sl	::=	Sl′ S	break-to⟦Sl′⟧ ≜ break-to⟦S⟧ ≜ break-to⟦Sl⟧
S	::=	**if** (B) S_t	break-to⟦S_t⟧ ≜ break-to⟦S⟧
S	::=	**if** (B) S_t **else** S_f	break-to⟦S_t⟧ ≜ break-to⟦S_f⟧ ≜ break-to⟦S⟧
S	::=	**while** (B) S_b	break-to⟦S_b⟧ ≜ after⟦S⟧
S	::=	**{** Sl **}**	break-to⟦Sl⟧ ≜ break-to⟦S⟧
S	::=	**{|** Sl **|}**	break-to⟦Sl⟧ ≜ after⟦S⟧

The axiomatic definition of labels leaves open different possible interpretations. We explicitly decorate programs with labels, as in [11, sect. 4.2.3]. Notice that labels in [35] are the program remaining to be executed when execution reaches that program point and that this involve a one-unrolling of iterations (there are no breaks). These labels also satisfy our labelling requirements.

3 Abstract Domain

An abstract domain [11, sect. 4.2.3] is an algebra $\mathbb{D}$ defining the semantic domain $\mathbb{P}$ and basic operations on elements of the semantic domain (formalizing the effect of executing a program component), of the following type:

$$\mathbb{D} \triangleq \langle \mathbb{P},\ \mathsf{program},\ \mathsf{stmtlist},\ \mathsf{empty},\ \mathsf{assign},\ \mathsf{skip},\ \mathsf{if},\ \mathsf{ife},\ \mathsf{iter},\ \mathsf{break},\ \mathsf{compound},\ \mathsf{breakable}\rangle \tag{2}$$

The abstract domain $\mathbb{D}$ is *well defined* when $\mathbb{P}$ is a set and the abstract operations satisfy the following conditions:

$$\begin{aligned} \mathsf{program} &\in \mathsf{P} \to \mathbb{P} \to \mathbb{P}\ ^{3} \\ \mathsf{stmtlist} &\in \mathsf{Sl} \to \mathbb{P} \times \mathbb{P} \to \mathbb{P} \\ \mathsf{empty} &\in \mathsf{Sl} \to \mathbb{P} \\ \mathsf{assign}, \mathsf{skip}, \mathsf{break} &\in \mathsf{S} \to \mathbb{P} \\ \mathsf{if}, \mathsf{iter}, \mathsf{compound}, \mathsf{breakable} &\in \mathsf{S} \to \mathbb{P} \to \mathbb{P} \\ \mathsf{ife} &\in \mathsf{S} \to \mathbb{P} \times \mathbb{P} \to \mathbb{P} \end{aligned}$$

(A more precise specification would further restrict the type of admissible program components for each operation, for example, $\mathsf{empty} \in \{\ \epsilon\ \} \to \mathbb{P}$.)

[3] $A \to B$ defines the total maps from A to B, it is right associative since function application is left associative.

4 Abstract Interpreter

The abstract interpreter $\mathcal{S}(\!|\mathbb{D}|\!) \in \mathsf{Pc} \rightharpoonup \mathbb{P}$ is a partial function specifying the abstract semantics $\mathcal{S}(\!|\mathbb{D}|\!)[\![\mathtt{S}]\!]$ of a program component `S`. It is parameterized by an abstract domain $\mathbb{D}$ (2). We write $\mathcal{S}$ for $\mathcal{S}(\!|\mathbb{D}|\!)$ afterwards, when the abstract domain $\mathbb{D}$ is implicitly known from the context.

The abstract interpreter proceeds by structural induction on the program syntax, applying the operations of the abstract domain to each program subcomponent. The abstract interpreter is the common skeleton of all semantics, analyses, program transformations, and unrolling.

- *Abstract semantics of a program* `P ::= Sl`

$$\mathcal{S}[\![\mathtt{P}]\!] \triangleq \mathsf{program}[\![\mathtt{P}]\!](\mathcal{S}[\![\mathtt{Sl}]\!]) \tag{3}$$

- *Abstract semantics of a statement list* `Sl ::= Sl' S`

$$\mathcal{S}[\![\mathtt{Sl}]\!] \triangleq \mathsf{stmtlist}[\![\mathtt{Sl}]\!](\mathcal{S}[\![\mathtt{Sl}']\!], \mathcal{S}[\![\mathtt{S}]\!]) \tag{4}$$

- *Abstract semantics of an empty statement list* `Sl ::=` ϵ

$$\mathcal{S}[\![\mathtt{Sl}]\!] \triangleq \mathsf{empty}[\![\mathtt{Sl}]\!] \tag{5}$$

- *Abstract semantics of an assignment statement* `S ::= x = A ;`

$$\mathcal{S}[\![\mathtt{S}]\!] \triangleq \mathsf{assign}[\![\mathtt{S}]\!] \tag{6}$$

- *Abstract semantics of a skip statement* `S ::= ;`

$$\mathcal{S}[\![\mathtt{S}]\!] \triangleq \mathsf{skip}[\![\mathtt{S}]\!] \tag{7}$$

- *Abstract semantics of a conditional statement* `S ::= if (B)` $\mathtt{S}_t$

$$\mathcal{S}[\![\mathtt{S}]\!] \triangleq \mathsf{if}[\![\mathtt{S}]\!](\mathcal{S}[\![\mathtt{S}_t]\!]) \tag{8}$$

- *Abstract semantics of a conditional statement* `S ::= if (B)` $\mathtt{S}_t$ `else` $\mathtt{S}_f$

$$\mathcal{S}[\![\mathtt{S}]\!] \triangleq \mathsf{ife}[\![\mathtt{S}]\!](\mathcal{S}[\![\mathtt{S}_t]\!], \mathcal{S}[\![\mathtt{S}_f]\!]) \tag{9}$$

- *Abstract semantics of an iteration statement* `S ::= while (B)` $\mathtt{S}_b$

$$\mathcal{S}[\![\mathtt{S}]\!] \triangleq \mathsf{iter}[\![\mathtt{S}]\!](\mathcal{S}[\![\mathtt{S}_b]\!]) \tag{10}$$

- *Abstract semantics of a break statement* `S ::= break ;`

$$\mathcal{S}[\![\mathtt{S}]\!] \triangleq \mathsf{break}[\![\mathtt{S}]\!] \tag{11}$$

- *Abstract semantics of a compound statement* $\mathtt{S} ::= \mathtt{\{\ Sl\ \}}$

$$\mathcal{S}[\![\mathtt{S}]\!] \triangleq \mathsf{compound}[\![\mathtt{S}]\!](\mathcal{S}[\![\mathtt{Sl}]\!]) \tag{12}$$

- *Abstract semantics of a breakable statement* $\mathtt{S} ::= \mathtt{\{|\ Sl\ |\}}$

$$\mathcal{S}[\![\mathtt{S}]\!] \triangleq \mathsf{breakable}[\![\mathtt{S}]\!](\mathcal{S}[\![\mathtt{Sl}]\!]) \tag{13}$$

The advantage of this abstract interpreter parameterized by an abstract domain is that it can be reasoned upon by structural induction (that is induction on the program syntax) as opposed to reasoning on graphs with usual intermediate program representations. Moreover, it can be easily implemented with a functional programming language with modules and module functors.

5 Prefix Abstract Interpreter

Given a prelude R, that is a precondition when arriving at a program component $\mathtt{S}$, that is $\mathsf{at}[\![\mathtt{S}]\!]$, the prefix abstract semantics $\mathcal{S}^p[\![\mathtt{S}]\!]\ R$ of this program component $\mathtt{S}$ returns a continuation, specifying at each program point ℓ of $\mathtt{S}$, a description of the execution from $\mathsf{at}[\![\mathtt{S}]\!]$ when arriving at ℓ.

The prefix abstract interpreter is classically used as the concrete semantics for the abstract reachability analysis; see [11, chapter 42]

5.1 Prefix Abstract Domain

The prefix abstract domain $\mathbb{D}^p$ has type:

$$\mathbb{D}^p \triangleq \langle \mathbb{P}^p,\ \sqsubseteq,\ \bot,\ \sqcup,\ \mathsf{assign}^p,\ \mathsf{skip}^p,\ \mathsf{test}^p,\ \overline{\mathsf{test}}^p,\ \mathsf{break}^p \rangle \tag{14}$$

This prefix abstract domain $\mathbb{D}^p$ is *well defined* when $\langle \mathbb{P}^p,\ \sqsubseteq \rangle$ is a poset of properties with infimum $\bot$ and the partially defined least upper bound (lub) $\sqcup$.

$$\begin{aligned} \mathsf{empty}^p, \mathsf{skip}^p, \mathsf{break}^p &\in \mathsf{Pc} \to (\mathsf{L} \times \mathsf{L}) \to \mathbb{P}^p \nearrow \mathbb{P}^p \ [4] \\ \mathsf{assign}^p &\in \mathsf{Pc} \to (\mathsf{L} \times \mathsf{V} \times \mathsf{A} \times \mathsf{L}) \to \mathbb{P}^p \nearrow \mathbb{P}^p \\ \mathsf{test}^p, \overline{\mathsf{test}}^p &\in \mathsf{Pc} \to (\mathsf{L} \times \mathsf{B} \times \mathsf{L}) \to \mathbb{P}^p \nearrow \mathbb{P}^p \end{aligned}$$

The poset $\langle \mathbb{P}^p,\ \sqsubseteq,\ \bot,\ \sqcup \rangle$ is extended pointwise to $\langle \mathsf{L} \to \mathbb{P}^p,\ \dot{\sqsubseteq},\ \dot{\bot},\ \dot{\sqcup} \rangle$ and $\langle \mathbb{P}^p \nearrow \mathsf{L} \to \mathbb{P}^p,\ \ddot{\sqsubseteq},\ \ddot{\bot},\ \ddot{\sqcup} \rangle$. An additional requirement may be that

[4] $A \nearrow B$ defines the increasing/isotone maps when A and B are posets.

the lub $\sqcup$ is well defined both for pairs of elements of $\mathbb{P}^p$ and for $\sqsubseteq$-increasing chains. (15)

In that case, $\langle \mathbb{P}^p, \sqsubseteq \rangle$ is both a join-lattice and a complete partial order (CPO), and this extends to the preceding pointwise definitions.

5.2 Prefix Abstract Functor

The prefix abstract functor $\mathcal{D}^p$, maps a prefix abstract domain of type (14) into an abstract domain

$$\mathcal{D}^p(\!|\mathbb{D}^p|\!) \triangleq \langle \mathbb{P},\ \mathsf{program},\ \mathsf{stmtlist},\ \mathsf{empty},\ \mathsf{assign},\ \mathsf{skip},\ \mathsf{if},\ \mathsf{ife},\ \mathsf{iter},\ \mathsf{break},\ \mathsf{compound},\ \mathsf{breakable} \rangle \quad (16)$$

of type (2), defined, assuming (15), as follows:

- *Abstract prefix semantic domain* $\mathbb{P} \triangleq \mathbb{P}^p \stackrel{\nearrow}{\to} \mathsf{L} \to \mathbb{P}^p$ ($\langle \mathbb{P}, \ddot{\sqsubseteq} \rangle$ satisfies (2))
- *Abstract prefix semantics of a program* $\ell_0\ \mathtt{P}\ \ell_1 ::= \ell_0\ \mathtt{Sl}\ \ell_1$ (where $\ell_0 = \mathsf{at}[\![\mathtt{Sl}]\!] = \mathsf{at}[\![\mathtt{P}]\!]$ and $\ell_1 = \mathsf{after}[\![\mathtt{Sl}]\!] = \mathsf{after}[\![\mathtt{P}]\!]$))

$$\mathsf{program}[\![\mathtt{P}]\!]\ Sl\ R\,\ell \triangleq Sl(R)\ell \quad (17)$$

- *Abstract prefix semantics of a statement list* $\ell_0\ \mathtt{Sl}\ \ell_2 ::= \ell_0\ \mathtt{Sl}'\ \ell_1\ \mathtt{S}\ \ell_2$ (where $\ell_0 = \mathsf{at}[\![\mathtt{Sl}]\!] = \mathsf{at}[\![\mathtt{Sl}']\!]$, $\ell_1 = \mathsf{at}[\![\mathtt{S}]\!] = \mathsf{after}[\![\mathtt{Sl}']\!]$, and $\ell_2 = \mathsf{after}[\![\mathtt{S}]\!] = \mathsf{after}[\![\mathtt{Sl}]\!]$) [5]

$$\begin{array}{lll} \mathsf{stmtlist}[\![\mathtt{Sl}]\!](Sl', S)\ R\,\ell & \triangleq & (\ \ell \in \mathsf{labs}[\![\mathtt{Sl}']\!] \setminus \{\mathsf{at}[\![\mathtt{S}]\!]\}\ ?\ Sl'\ R\,\ell \\ & & \mid \ell \in \mathsf{labs}[\![\mathtt{S}]\!]\ ?\ S(Sl'\ R\,\mathsf{at}[\![\mathtt{S}]\!])\ \ell \\ & & :\ \bot\ ) \end{array} \quad (18)$$

- *Abstract prefix semantics of an empty statement list* $\ell_0 \mathtt{Sl} \ell_0 ::= \ell_0\ \epsilon\ \ell_0$ (where $\ell_0 = \mathsf{at}[\![\mathtt{Sl}]\!] = \mathsf{after}[\![\mathtt{Sl}]\!]$)

$$\mathsf{empty}[\![\mathtt{Sl}]\!]\ R\,\ell \triangleq (\ \ell = \mathsf{at}[\![\mathtt{Sl}]\!]\ ?\ \mathsf{empty}^p[\![\ell_0, \ell_0]\!]R\ :\ \bot\ ) \quad (19)$$

[5] $(\ldots ?\ \ldots \mid \ldots ?\ \ldots :\ \ldots)$ is the conditional expression (as in C)

- *Abstract prefix semantics of an assignment statement* $\ell_0 \mathtt{S} \ell_1 ::= \ell_0 \mathtt{x = A ;} \ell_1$ (where $\ell_0 = \mathsf{at}[\![\mathtt{S}]\!]$ and $\ell_1 = \mathsf{after}[\![\mathtt{S}]\!]$)

$$\begin{array}{lcl} \mathsf{assign}[\![\mathtt{S}]\!]\ R\,\ell & \triangleq & (\ell = \ell_0\ ?\ R \\ & & \mid \ell = \ell_1\ ?\ \mathsf{assign}^p[\![\ell_0, \mathtt{x}, \mathtt{A}, \ell_1]\!] R \\ & & :\ \bot\,) \end{array} \tag{20}$$

- *Abstract prefix semantics of a skip statement* $\ell_0 \mathtt{S} \ell_1 ::= \mathtt{;}$ (where $\ell_0 = \mathsf{at}[\![\mathtt{S}]\!]$ and $\ell_1 = \mathsf{after}[\![\mathtt{S}]\!]$)

$$\mathsf{skip}[\![\mathtt{S}]\!]\ R\,\ell \quad \triangleq \quad (\ell = \ell_0\ ?\ R \mid \ell = \ell_1\ ?\ \mathsf{skip}^p[\![\ell_0, \ell_1]\!] R : \bot\,) \tag{21}$$

- *Abstract prefix semantics of a conditional statement* $\ell_0 \mathtt{S} \ell_2 ::= \ell_0 \mathtt{if\ (B)}\ \ell_t \mathtt{S}_t \ell_2$ (where $\ell_0 = \mathsf{at}[\![\mathtt{S}]\!]$, $\ell_t = \mathsf{at}[\![\mathtt{S}_t]\!]$, and $\ell_2 = \mathsf{after}[\![\mathtt{S}]\!] = \mathsf{after}[\![\mathtt{S}_t]\!]$)

$$\begin{array}{lcl} \mathsf{if}[\![\mathtt{S}]\!](S)\ R\,\ell & \triangleq & (\ell = \ell_0\ ?\ R \\ & & \mid \ell \in \mathsf{in}[\![\mathtt{S}_t]\!]\ ?\ S\,(\mathsf{test}^p[\![\ell_0, \mathtt{B}, \ell_t]\!]\ R)\,\ell \\ & & \mid \ell = \ell_2\ ?\ S\,(\mathsf{test}^p[\![\ell_0, \mathtt{B}, \ell_t]\!]\ R)\,\ell \sqcup \overline{\mathsf{test}}^p[\![\ell_0, \mathtt{B}, \ell_2]\!]\ R \\ & & :\ \bot\,) \end{array} \tag{22}$$

- *Abstract prefix semantics of a conditional statement* $\ell_0 \mathtt{S} \ell_2 ::= \ell_0 \mathtt{if\ (B)}\ \ell_t \mathtt{S}_t\ \mathtt{else}\ \ell_f \mathtt{S}_f \ell_2$ (where $\ell_0 = \mathsf{at}[\![\mathtt{S}]\!]$, $\ell_t = \mathsf{at}[\![\mathtt{S}_t]\!]$, $\ell_f = \mathsf{at}[\![\mathtt{S}_f]\!]$, and $\ell_2 = \mathsf{after}[\![\mathtt{S}]\!] = \mathsf{after}[\![\mathtt{S}_t]\!] = \mathsf{after}[\![\mathtt{S}_f]\!]$)

$$\begin{array}{lcl} \mathsf{ife}[\![\mathtt{S}]\!](St, Sf)\ R\,\ell & \triangleq & (\ell = \ell_0\ ?\ R \\ & & \mid \ell \in \mathsf{in}[\![\mathtt{S}_t]\!]\ ?\ St\ (\mathsf{test}^p[\![\ell_0, \mathtt{B}, \ell_t]\!]\ R)\,\ell \\ & & \mid \ell \in \mathsf{in}[\![\mathtt{S}_f]\!]\ ?\ Sf\ (\overline{\mathsf{test}}^p[\![\ell_0, \mathtt{B}, \ell_f]\!]\ R)\,\ell \\ & & \mid \ell = \ell_2\ ?\ St\ (\mathsf{test}^p[\![\ell_0, \mathtt{B}, \ell_t]\!]\ R)\,\ell \sqcup Sf\ (\overline{\mathsf{test}}^p[\![\ell_0, \mathtt{B}, \ell_f]\!]\ R)\,\ell \\ & & :\ \bot\,) \end{array} \tag{23}$$

- *Abstract prefix semantics of an iteration statement* $\ell_0 \mathtt{S} \ell_2 ::= \mathtt{while}\ \ell_0\ \mathtt{(B)}\ \ell_1 \mathtt{S}_b\ \ell_2$ (where $\ell_0 = \mathsf{at}[\![\mathtt{S}]\!] = \mathsf{after}[\![\mathtt{S}_b]\!]$, $\ell_1 = \mathsf{at}[\![\mathtt{S}_b]\!]$, and $\ell_2 = \mathsf{after}[\![\mathtt{S}]\!]$) [6]

$$\begin{array}{lcl} \mathsf{iter}[\![\mathtt{S}]\!](Sb)\ R\,\ell & \triangleq & \mathsf{lfp}^{\dot{\sqsubseteq}}\,(\mathcal{F}^p[\![Sb]\!]\ R)\,\ell \\ \mathcal{F}^p[\![Sb]\!] & \in & \mathbb{P}^p \to ((\mathsf{L} \to \mathbb{P}^p) \to (\mathsf{L} \to \mathbb{P}^p)) \end{array} \tag{24}$$

[6] $\mathsf{lfp}^{\sqsubseteq}\, f$ denotes the $\sqsubseteq$-least fixpoint of an operator f on a poset $\langle \mathbb{P}, \sqsubseteq \rangle$, if any; for example [38].

$$
\begin{array}{l}
\mathcal{F}^p[\![Sb]\!]\ R\ X\ \ell\ \ = \\
\quad (\ \ell = \ell_0\ \mathbf{?}\ R \sqcup Sb\ (\mathsf{test}^p[\![\ell_0, \mathtt{B}, \ell_1]\!] X(\ell_0))\ \ell \\
\quad \mid \ell \in \mathsf{in}[\![\mathtt{S}_b]\!] \setminus \{\ell_0\}\ \mathbf{?}\ Sb\ (\mathsf{test}^p[\![\ell_0, \mathtt{B}, \ell_1]\!] X(\ell_0))\ \ell \\
\quad \mid \ell = \ell_2\ \mathbf{?}\ \overline{\mathsf{test}}^p[\![\ell_0, \mathtt{B}, \ell_2]\!] X(\ell_0) \sqcup \bigsqcup_{\ell' \in \mathsf{breaks\text{-}of}[\![\mathtt{S}_b]\!]} Sb\ (\mathsf{test}^p[\![\ell_0, \mathtt{B}, \ell_1]\!] X(\ell_0))\ \ell' \\
\quad :\ \bot\ )
\end{array}
$$

- *Abstract prefix semantics of a break statement* $\mathtt{S} ::= \ell_0$ **`break ;`** (where $\ell_0 = \mathsf{at}[\![\mathtt{S}]\!]$)

$$
\mathsf{break}[\![\mathtt{S}]\!]\ R\ \ell \ \triangleq\ (\ \ell = \ell_0\ \mathbf{?}\ \mathsf{break}^p[\![\ell_0, \mathsf{break\text{-}to}[\![\mathtt{S}]\!]]\!] R\ :\ \bot\ ) \tag{25}
$$

- *Abstract prefix semantics of a compound statement* $\mathtt{S} ::= \ell_0$**`{ Sl }`**ℓ_1 (where $\ell_0 = \mathsf{at}[\![\mathtt{Sl}]\!] = \mathsf{at}[\![\mathtt{S}]\!]$ and $\ell_0 = \mathsf{after}[\![\mathtt{S}]\!] = \mathsf{after}[\![\mathtt{Sl}]\!]$)

$$
\mathsf{compound}[\![\mathtt{S}]\!](Sl)\ R\ \ell \ \triangleq\ Sl(R)\ell \tag{26}
$$

- *Abstract prefix semantics of a breakable statement* $\mathtt{S} ::= \ell_0$**`{| Sl |}`**ℓ_1 (where $\ell_0 = \mathsf{at}[\![\mathtt{Sl}]\!] = \mathsf{at}[\![\mathtt{S}]\!]$ and $\ell_1 = \mathsf{after}[\![\mathtt{S}]\!] = \mathsf{after}[\![\mathtt{Sl}]\!]$)

$$
\begin{array}{rcl}
\mathsf{breakable}[\![\mathtt{S}]\!](Sl)\ R\ \ell & \triangleq & (\ \ell \in \mathsf{in}[\![\mathtt{S}]\!]\ \mathbf{?}\ Sl(R)\ell \\
& & \quad \mid \ell = \mathsf{after}[\![\mathtt{S}]\!]\ \mathbf{?}\ Sl(R)\ell \sqcup \bigsqcup_{\ell' \in \mathsf{breaks\text{-}of}[\![\mathtt{S}_b]\!]} Sl(R)\ell' \\
& & \quad :\ \bot\ )
\end{array} \tag{27}
$$

  where $\bigsqcup \emptyset = \bot$.

5.3 Prefix Abstract Interpreter

The prefix abstract interpreter $\mathcal{S}^p(\!|\mathbb{D}^p|\!) \in \mathsf{Pc} \to \mathbb{P}^p \xrightarrow{\nearrow} \mathsf{L} \to \mathbb{P}^p$ specifies the prefix abstract semantics $\mathcal{S}^p[\![\mathtt{S}]\!]$ of a program component $\mathtt{S} \in \mathsf{Pc}$. If any execution of $\mathtt{S}$ is started with precondition R satisfied, and later reaches program point ℓ of $\mathtt{S}$, then $\mathcal{S}^p[\![\mathtt{S}]\!]\ R\ \ell$ holds at that point. This prefix abstract interpreter is generic, meaning that it is parameterized by a prefix abstract domain $\mathbb{D}^p(14)$. It is the instance of the abstract interpreter $\mathcal{S}$ for the abstract domain $\mathcal{D}^p(\!|\mathbb{D}^p|\!)$:

$$
\mathcal{S}^p(\!|\mathbb{D}^p|\!) \ \triangleq\ \mathcal{S}(\!|\mathcal{D}^p(\!|\mathbb{D}^p|\!)|\!) \tag{28}
$$

It follows from this definition that

$$\forall \mathsf{S} \in \mathsf{Pc} \,.\, \forall \ell_{\in} \mathsf{L} \,.\, \ell \notin \mathsf{labs}[\![\mathsf{S}]\!] \;\; \Rightarrow \;\; \mathcal{S}^p[\![\mathsf{S}]\!]\, R\, \ell \triangleq \bot \tag{29}$$

meaning that no execution of a program component ever reaches a program point outside this program component and therefore there is no information at such exterior points (as denoted by $\bot$ meaning empty/void/…).

To use the prefix abstract interpreter $\mathcal{S}^p(\!|\mathbb{D}^p|\!)$, we have to provide an abstract domain $\mathbb{D}^p$ of type (14), as a parameter. This is what we do in the next section, to define the prefix trace semantics.

6 Prefix Trace Semantics

The prefix trace semantics $\mathcal{S}^{pt}$ is an instance of the prefix abstract interpreter $\mathcal{S}^p$ (itself an instance of the abstract interpreter $\mathcal{S}$, as shown by (28)).

Given a prelude R, that is execution traces arriving at a program component $\mathtt{S}$, that is $\mathsf{at}[\![\mathtt{S}]\!]$, the prefix trace semantics $\mathcal{S}^{pt}[\![\mathtt{S}]\!]\, R$ of this program component $\mathtt{S}$ returns a continuation, specifying at each program point ℓ of $\mathtt{S}$, a description of the execution traces from $\mathsf{at}[\![\mathtt{S}]\!]$ when arriving at ℓ. Traces are finite sequences of states separated by actions. The states are a pair of a program point and an environment assigning values to variables. An action records an elementary step in the program.

6.1 Variables

We let V to be the set of variables (of the language, program, or a parameter of the semantics; see [11, rem. 3.5]).

6.2 Values

We let $\nu \in \mathbb{V}$ to be the set of values (either integers, machine integers plus error, reals, floats, float intervals, or fixed points).[7]

6.3 Environments

We let $\mathbb{E} \triangleq \mathsf{V} \to \mathbb{V}$ to be the set of environments $\rho \in \mathbb{E}$, mapping variables $\mathtt{x} \in \mathsf{V}$ to their value $\rho(\mathtt{x}) \in \mathbb{V}$. The assignment of a value ν to a variable $\mathtt{x}$ in environment ρ is

[7] For non-integer values, arithmetic expressions can also be the constant `0.1`, which no exact representation in floats.

$$\begin{array}{lcl} \rho[\mathtt{x} \leftarrow \nu](\mathtt{x}) & \triangleq & \nu \\ \rho[\mathtt{x} \leftarrow \nu](\mathtt{y}) & \triangleq & \rho(\mathtt{y}) \quad \text{when} \quad \mathtt{y} \neq \mathtt{x} \end{array}$$

6.4 States

States are pairs $\sigma = \langle \ell,\ \rho \rangle \in \mathbb{S}$ of a program label ℓ recording the program point reached during a computation and an environment ρ recording the values of variables at that point in the computation.

6.5 Actions

Actions (or events) $\mathsf{a} \in (\mathsf{L} \times \mathsf{V} \times \mathsf{E} \times \mathsf{L}) \cup (\mathsf{L} \times \{\mathtt{;}\} \times \mathsf{L}) \cup (\mathsf{L} \times \mathsf{B} \times \mathsf{L}) \cup (\mathsf{L} \times \neg\mathsf{B} \times \mathsf{L}) \cup (\mathsf{L} \times \{\mathtt{break\ ;}\} \times \mathsf{L})$ record the execution of assignments, skips, true and false tests, and breaks.

6.6 Traces

Traces $\pi \in \mathbb{T}$ are nonempty finite sequences $\pi = \sigma_0 \xrightarrow{\mathsf{a}_0} \sigma_1 \xrightarrow{\mathsf{a}_1} \sigma_2 \ldots \sigma_{n-1} \xrightarrow{\mathsf{a}_{n-1}} \sigma_n$ of states separated by actions, recording an execution of length $n = |\pi| \geqslant 0$.

6.7 Prefix Trace Abstract Domain

The prefix trace abstract domain is

$$\mathbb{D}^{pt} \triangleq \langle \mathbb{P}^{pt},\ \dot{\subseteq},\ \dot{\emptyset},\ \dot{\cup},\ \mathsf{assign}^{pt},\ \mathsf{skip}^{pt},\ \mathsf{test}^{pt},\ \overline{\mathsf{test}}^{pt},\ \mathsf{break}^{pt} \rangle \tag{30}$$

of type (14), such that

$$\begin{array}{rcl} \mathbb{P}^{pt} & \triangleq & \wp(\mathbb{T}) \stackrel{\nearrow}{\to} \wp(\mathbb{T}) \\ \mathsf{assign}^{pt}[\![\ell_0, \mathtt{x}, \mathtt{A}, \ell_1]\!] R & \triangleq & \{\pi \xrightarrow{\mathsf{a}} \langle \ell_0',\ \rho \rangle \xrightarrow{\ell_0', \mathtt{x}, \mathtt{A}, \ell_1} \langle \ell_1,\ \rho[\mathtt{x} \leftarrow \mathcal{A}[\![\mathtt{A}]\!]\rho] \rangle \mid \\ & & \qquad \pi \xrightarrow{\mathsf{a}} \langle \ell_0',\ \rho \rangle \in R \wedge \ell_0' = \ell_0\} \\ \mathsf{skip}^{pt}[\![\ell_0, \ell_1]\!] R & \triangleq & \{\pi \xrightarrow{\mathsf{a}} \langle \ell_0',\ \rho \rangle \xrightarrow{\ell_0', \mathtt{;}, \ell_t} \langle \ell_t,\ \rho \rangle \mid \pi \xrightarrow{\mathsf{a}} \langle \ell_0',\ \rho \rangle \in R \wedge \ell_0' = \ell_0\} \\ \mathsf{test}^{pt}[\![\ell_0, \mathtt{B}, \ell_t]\!]\ R & \triangleq & \{\pi \xrightarrow{\mathsf{a}} \langle \ell_0',\ \rho \rangle \xrightarrow{\ell_0', \mathtt{B}, \ell_t} \langle \ell_t,\ \rho \rangle \mid \\ & & \qquad \pi \xrightarrow{\mathsf{a}} \langle \ell_0',\ \rho \rangle \in R \wedge \mathcal{B}[\![\mathtt{B}]\!]\rho \wedge \ell_0' = \ell_0\} \end{array}$$

$$\begin{array}{rcl}
\overline{\mathsf{test}}^{pt}[\![\ell_0, \mathtt{B}, \ell_t]\!]\ R & \triangleq & \{\pi \xrightarrow{\mathtt{a}} \langle \ell'_0,\ \rho\rangle \xrightarrow{\ell'_0, \neg \mathtt{B}, \ell_t} \langle \ell_t,\ \rho\rangle \mid \\
& & \qquad \pi \xrightarrow{\mathtt{a}} \langle \ell'_0,\ \rho\rangle \in R \wedge \neg \mathcal{B}[\![\mathtt{B}]\!]\rho \wedge \ell'_0 = \ell_0\} \\
\mathsf{break}^{pt}[\![\ell_0, \ell_1]\!]R & \triangleq & \{\pi \xrightarrow{\mathtt{a}} \langle \ell'_0,\ \rho\rangle \xrightarrow{\ell'_0, \mathtt{break\ ;}, \ell_t} \langle \ell_t,\ \rho\rangle \mid \\
& & \qquad \pi \xrightarrow{\mathtt{a}} \langle \ell'_0,\ \rho\rangle \in R \wedge \ell'_0 = \ell_0\}
\end{array}$$

where $\mathcal{A}[\![\mathtt{A}]\!]\rho$ is the value of arithmetic expression $\mathtt{A}$ in environment ρ and $\mathcal{B}[\![\mathtt{B}]\!]\rho$ the Boolean value of Boolean expression $\mathtt{B}$ in environment ρ; see [11, chapter 3] for integers and [11, chapter 32] for floats and interval arithmetics. Notice that $\langle \mathbb{P}^{pt}, \dot{\subseteq}\rangle$ is a complete lattice hence both a join-lattice and a CPO.

6.8 Prefix Trace Semantics

The finite prefix trace semantics $\mathcal{S}^{pt} \in \mathsf{Pc} \to \wp(\mathbb{T}) \xrightarrow{\nearrow} \mathsf{L} \to \wp(\mathbb{T})$ (as defined in [11, chapter 42]) is the instance of the prefix abstract interpreter $\mathcal{S}^p$ for the abstract domain

$$\mathcal{S}^{pt} \triangleq \mathcal{S}^p(\!|\mathbb{D}^{pt}|\!) = \mathcal{S}(\!|\mathcal{D}^p(\!|\mathbb{D}^{pt}|\!)|\!) \tag{31}$$

Notice that by composition, the finite prefix trace semantics $\mathcal{S}^{pt}$ is an instance of the abstract interpreter $\mathcal{S}$. The prefix trace semantics allows us to define precisely what we mean by *bounded execution*.

7 Boundedness Hypothesis

The boundedness hypothesis for a program component $\mathtt{S}$ states that there exists a bound $\beta[\![\mathtt{S}]\!] \in \mathbb{N}^+$ on the length of any prefix trace of $\mathtt{S}$ for all possible preludes [8]. Formally, define

$$\beta[\![\mathtt{S}]\!] \triangleq \max\{n-1 \mid \sigma_0 \xrightarrow{\mathtt{a}_0} \sigma_1 \ldots \sigma_{n-1} \xrightarrow{\mathtt{a}_{n-1}} \sigma_n \in \mathcal{S}^{pt}[\![\mathtt{S}]\!]\ R \wedge R \in \wp(\mathbb{T})\}$$

where $\max \mathbb{N}^+ = \infty$. The *boundedness hypothesis* is

$$\mathtt{S} \textit{ is bounded} \text{ if and only if } \beta[\![\mathtt{S}]\!] \in \mathbb{N}^+. \tag{32}$$

Static analysis can be used to determine this value, or bound it; see for example [8, 40, 13].

[8] A variant would require an hypothesis on input states.

8 Maximal Abstract Interpreter

A source code program transformation is correct whenever the transformed program is semantically equivalent to the original. The maximal abstract interpreter, which is an abstraction of the prefix abstract interpreter will help us define which particular formal semantics is considered when defining that equivalence.

The maximal abstract interpreter $\mathcal{S}^m \in \mathsf{Pc} \to \mathbb{P}^p \stackrel{\nearrow}{\to} \mathbb{P}^p \times \mathbb{P}^p$ is an instance of the abstract interpreter $\mathcal{S} \in \mathsf{Pc} \to \mathbb{P}$.

The maximal abstract interpreter $\mathcal{S}^m$ specifies the maximal abstract semantics $\langle T,\ B\rangle = \mathcal{S}^m[\![\mathtt{S}]\!]\ R$ of a program component `S` when execution of `S` starts with the precondition R being satisfied, reaches the exit program point $\mathsf{after}[\![\mathtt{S}]\!]$ of `S`, thus terminating execution. T describes normal termination (with a test which is false for an iteration) and B termination through a **`break ;`** for an iteration (and $\bot$ otherwise).

Potential nonterminating executions are all abstracted away (so this is partial correctness not total correctness for programs that may not terminate; partial and total correctness coincide under the boundedness hypothesis (32)).

8.1 Maximal Abstraction

The maximal abstraction is $\alpha^{p\to m} \in (\mathsf{Pc} \to \mathbb{P}^p \stackrel{\nearrow}{\to} \mathsf{L} \to \mathbb{P}^p) \stackrel{\nearrow}{\to} (\mathsf{Pc} \to \mathbb{P}^p \stackrel{\nearrow}{\to} \mathbb{P}^p \times \mathbb{P}^p)$ such that

$$\alpha^{p\to m}[\![\mathtt{S}]\!]\ \mathcal{S}\ R \ \triangleq\ \langle \mathcal{S}[\![\mathtt{S}]\!]\ R\, \mathsf{after}[\![\mathtt{S}]\!],\ \bigsqcup_{\ell \in \mathsf{breaks\text{-}of}[\![\mathtt{S}]\!]} \mathcal{S}[\![\mathtt{S}]\!]\ R\, \ell\rangle \tag{33}$$

This is a Galois connection ([11, ex. 11.21 and 11.20]). When applied to the prefix abstract interpreter $\mathcal{S}^p \in \mathsf{Pc} \to \mathbb{P}^p \stackrel{\nearrow}{\to} \mathsf{L} \to \mathbb{P}^p$, it yields the maximal abstract interpreter $\mathcal{S}^m \in \mathsf{Pc} \to \mathbb{P}^p \stackrel{\nearrow}{\to} \mathbb{P}^p$:

$$\mathcal{S}^m[\![\mathtt{S}]\!]\ R \ \triangleq\ \alpha^{p\to m}[\![\mathtt{S}]\!]\ (\mathcal{S}^p[\![\mathtt{S}]\!])\ R \tag{34}$$

Since the prefix abstract interpreter $\mathcal{S}^p$ is parameterized by the abstract domain $\mathbb{D}^p$ (14), the maximal abstract interpreter $\mathcal{S}^m$ is also parameterized by this same abstract domain $\mathbb{D}^p$.

8.2 Maximal Abstract Domain

The maximal abstract domain

$$\mathcal{D}^m(\!|\mathbb{D}^p|\!) \triangleq \langle \mathbb{P}^m, \mathsf{program}^m, \mathsf{stmtlist}^m, \mathsf{empty}^m, \mathsf{assign}^m, \mathsf{skip}^m, \mathsf{if}^m, \mathsf{ife}^m, \mathsf{iter}^m, \mathsf{break}^m, \mathsf{compound}^m, \mathsf{breakable}^m \rangle \quad (35)$$

is defined as a function $\mathcal{D}^m$ of $\mathbb{D}^p$, as follows:

- *Abstract maximal semantic domain* $\mathbb{P}^m \triangleq \mathbb{P}^p \stackrel{\nearrow}{\to} \mathbb{P}^p$
- *Abstract maximal semantics of a program* ℓ_0 `P` $\ell_1 ::= \ell_0$ `Sl` ℓ_1

$$\mathsf{program}^m[\![\mathtt{P}]\!]\ Sl\ R \triangleq \mathsf{let}\ \langle T,\ B\rangle = Sl(R)\ \mathsf{in}\ \langle \mathsf{program}^p[\![\ell_0, \ell_1]\!]\ (T),\ \bot\rangle \quad (36)$$

- *Abstract maximal semantics of a statement list* ℓ_0 `Sl` $\ell_2 ::= \ell_0$ `Sl'` ℓ_1 `S` ℓ_2

$$\begin{array}{lll} \mathsf{stmtlist}^m[\![\mathtt{Sl}]\!](Sl', S)\ R & \triangleq & \mathsf{let}\ \langle T',\ B'\rangle = Sl'(R)\ \mathsf{in} \\ & & \quad \mathsf{let}\ \langle T,\ B\rangle = S(T')\ \mathsf{in} \\ & & \quad\quad \langle T,\ B' \sqcup B\rangle \end{array} \quad (37)$$

- *Abstract maximal semantics of an empty statement list* ℓ_0`Sl`$\ell_0 ::= \ell_0\ \epsilon\ \ell_0$

$$\mathsf{empty}^m[\![\mathtt{Sl}]\!] \triangleq \langle \mathsf{empty}^p[\![\ell_0, \ell_0]\!] R,\ \bot\rangle \quad (38)$$

- *Abstract maximal semantics of an assignment statement* ℓ_0`S`$\ell_1 ::= \ell_0$`x = A ;`ℓ_1

$$\mathsf{assign}^m[\![\mathtt{S}]\!]\ R \triangleq \langle \mathsf{assign}^p[\![\ell_0, \mathtt{x}, \mathtt{A}, \ell_1]\!] R,\ \bot\rangle \quad (39)$$

- *Abstract maximal semantics of a skip statement* ℓ_0`S`$\ell_1 ::=$ `;`

$$\mathsf{skip}^m[\![\mathtt{S}]\!]\ R \triangleq \langle \mathsf{skip}^p[\![\ell_0, \ell_1]\!] R,\ \bot\rangle \quad (40)$$

- *Abstract maximal semantics of a conditional statement* ℓ_0`S`$\ell_2 ::= \ell_0$`if (B)` ℓ_t`S`$_t\ell_2$

$$\begin{array}{lll} \mathsf{if}^m[\![\mathtt{S}]\!](S)\ R & \triangleq & \mathsf{let}\ \langle T,\ B\rangle = S\left(\mathsf{test}^p[\![\ell_0, \mathtt{B}, \ell_t]\!]\ R\right)\ \mathsf{in} \\ & & \quad \langle T \sqcup \overline{\mathsf{test}}^p[\![\ell_0, \mathtt{B}, \ell_2]\!]\ R,\ B\rangle \end{array} \quad (41)$$

- *Abstract maximal semantics of a conditional statement* $\ell_0\mathtt{S}\ell_2 ::= \ell_0\mathtt{\textbf{if (B)}}\ \ell_t\mathtt{S}_t\ \mathtt{\textbf{else}}\ \ell_f\mathtt{S}_f\ell_2$

$$\begin{array}{rcl}\mathsf{ife}^m[\![\mathtt{S}]\!](St, Sf)\ R & \triangleq & \mathsf{let}\ \langle T_t,\ B_t\rangle = St\ (\mathsf{test}^p[\![\ell_0, \mathtt{B}, \ell_t]\!]\ R)\ \mathsf{in} \\ & & \quad \mathsf{let}\ \langle T_f,\ B_f\rangle = Sf\ (\overline{\mathsf{test}}^p[\![\ell_0, \mathtt{B}, \ell_f]\!]\ R)\ \mathsf{in} \\ & & \quad\quad \langle T_t \sqcup T_f,\ B_t \sqcup B_f\rangle \end{array} \tag{42}$$

- *Abstract maximal semantics of an iteration statement* $\ell_0\mathtt{S}\ell_2 ::= \mathtt{\textbf{while}}\ \ell_0\ \mathtt{\textbf{(B)}}\ \ell_1\mathtt{S}_b\ell_2$

$$\mathsf{iter}^m[\![\mathtt{S}]\!](\mathcal{S}[\![\mathtt{S}_b]\!])\ R \triangleq \mathsf{let}\ \langle T,\ B\rangle = \mathsf{lfp}^{\sqsubseteq_2}\ (\mathcal{F}^m[\![Sb]\!]\ R)\ \mathsf{in}\ \langle T \sqcup B,\ \bot\rangle \tag{43}$$

$$\mathcal{F}^m[\![Sb]\!] \in \mathbb{P}^p \to \mathbb{P}^p \times \mathbb{P}^p \to \mathbb{P}^p \times \mathbb{P}^p$$

$$\begin{array}{rcl}\mathcal{F}[\![Sb]\!]\ R\ \langle T,\ B\rangle & = & \mathsf{let}\ \langle T',\ B'\rangle = Sb\ (\mathsf{test}^p[\![\ell_0, \mathtt{B}, \ell_1]\!]T)\ \mathsf{in} \\ & & \quad \langle \overline{\mathsf{test}}^p[\![\ell_0, \mathtt{B}, \ell_2]\!]T',\ B \sqcup B'\rangle \end{array}$$

where the partial order $\langle \mathbb{P}^p \times \mathbb{P}^p,\ \sqsubseteq_2\rangle$ is componentwise and there is no break in an iteration that can break an outer loop.

- *Abstract maximal semantics of a break statement* $\mathtt{S} ::= \ell_0\ \mathtt{\textbf{break ;}}$ (where $\ell_0 = \mathsf{at}[\![\mathtt{S}]\!]$)

$$\mathsf{break}^m[\![\mathtt{S}]\!]\ R \triangleq \langle \bot,\ \mathsf{break}^p[\![\ell_0, \mathsf{break\text{-}to}[\![\mathtt{S}]\!]]\!]R\rangle \tag{44}$$

- *Abstract maximal semantics of a compound statement* $\mathtt{S} ::= \ell_0\mathtt{\textbf{\{}}\ [\ \mathtt{\textbf{\}}}m]\mathtt{Sl}\ell_1$

$$\begin{array}{rcl}\mathsf{compound}[\![\mathtt{S}]\!](Sl)R & \triangleq & \mathsf{let}\ \langle T,\ B\rangle = Sl(R)\ \mathsf{in} \\ & & \quad \langle \mathsf{compound}^p[\![\ell_0, \ell_1]\!](T),\ B\rangle \end{array} \tag{45}$$

- *Abstract maximal semantics of a breakable statement* $\mathtt{S} ::= \ell_0\mathtt{\textbf{\{|}}\ \mathtt{Sl}\ \mathtt{\textbf{|\}}}\ell_1$

$$\begin{array}{rcl}\mathsf{breakable}^m[\![\mathtt{S}]\!](Sl)R & \triangleq & \mathsf{let}\ \langle T,\ B\rangle = Sl(R)\ \mathsf{in} \\ & & \quad \langle \mathsf{breakable}^p[\![\ell_0, \ell_1]\!](T \sqcup B),\ \bot\rangle \end{array} \tag{46}$$

A breakable statement prevents internal breaks going outside this statement. They all go after the breakable statement.

8.3 Maximal Abstract Interpreter

The maximal abstract interpreter $\mathcal{S}^m$ is the instance of the abstract interpreter $\mathcal{S}$ for the abstract domain $\mathcal{D}^m(\!|\mathbb{D}^p|\!)$:

$$\mathcal{S}^m(\!|\mathbb{D}^p|\!) \quad \triangleq \quad \mathcal{S}(\!|\mathcal{D}^m(\!|\mathbb{D}^p|\!)|\!)$$

Using both structural and fixpoint induction, $\mathcal{S}^m$ yields the abstract version of Hoare logic of [11, chapter 26].

9 Maximal Trace Semantics

By the boundedness hypothesis there are no infinite traces and so the maximal trace semantics is the set of all maximal finite traces defined by the prefix trace semantics. It follows from (33) and (a proof by calculational design similar to) [11, chapter 20] that the maximal trace semantics $\mathcal{S}^{mt}$ is the instance of the maximal abstract interpreter $\mathcal{S}^m$ for the abstract domain $\mathbb{D}^{pt}$ (30), the operations of which have been defined in section 6.8.

$$\mathcal{S}^{mt}(\!|\mathbb{D}^{pt}|\!) \quad \triangleq \quad \mathcal{S}^m(\!|\mathbb{D}^{pt}|\!) \quad = \quad \mathcal{S}(\!|\mathcal{D}^m(\!|\mathbb{D}^{pt}|\!)|\!) \tag{47}$$

10 Relational Semantics

The relational semantics relates initial to final states of maximal computations.

10.1 Properties

Following [11, chapter 8], we represent properties by the set of elements which have this property. We are interested in relations between initial and final values of variables, that is properties in $\wp(\mathbb{E} \times \mathbb{E})$.

10.2 Relational Abstraction

The abstraction is

$$\begin{array}{rcl} \alpha^{mt\to r}(\langle T,\ B\rangle)R & \triangleq & \{\langle \rho,\ \rho_n\rangle \mid \langle \rho,\ \rho_0\rangle \in R \wedge \exists n \in \mathbb{N}\ .\ \langle \ell_0,\ \rho_0\rangle \xrightarrow{\mathsf{a}_0} \\ & & \ldots \xrightarrow{\mathsf{a}_{n-1}} \langle \ell_n,\ \rho_n\rangle \in T \cup B\} \end{array} \tag{48}$$

This in a Galois connection [11, ex. 11.8]. Typically, R is the identity $\mathbb{1}_{\mathbb{E}}$ on environments $\mathbb{E}$, in which case $\alpha^{mt\to r}(\mathcal{S})\mathbb{1}_{\mathbb{E}}$ is the program input-output relation for the trace semantics $\mathcal{S}$.

The relational semantics is

$$\mathcal{S}^r[\![\mathtt{S}]\!]R \quad \triangleq \quad \alpha^{mt\to r}(\mathcal{S}^{mt}(\!|\mathbb{D}^{pt}|\!)[\![\mathtt{S}]\!]R) \tag{49}$$

It relates initial and final values of variables on maximal finite executions (so non-termination is ignored).

10.3 Relational Semantic Domain

It follows from (48) and (a proof by calculational design similar to) [11, chapter 20] that the relational semantics $\mathcal{S}^r[\![\mathtt{S}]\!]$ is an instance of the maximal abstract interpreter for the following abstract domain:

$$\mathbb{D}^r \quad \triangleq \quad \langle \wp(\mathbb{E}\times\mathbb{E}),\ \subseteq,\ \emptyset,\ \cup,\ \mathsf{assign}^r,\ \mathsf{test}^r,\ \overline{\mathsf{test}}^r\rangle$$

such that

$$\begin{array}{rcl}
\mathsf{assign}^r[\![\mathtt{x},\mathtt{A}]\!]R & \triangleq & \{\langle\rho,\ \rho'[\mathtt{x}\leftarrow\mathcal{A}[\![\mathtt{A}]\!]\rho']\rangle \mid \langle\rho,\ \rho'\rangle\in R\}\\
\mathsf{test}^r[\![\mathtt{B}]\!]\ R & \triangleq & \{\langle\rho,\ \rho'\rangle \mid \langle\rho,\ \rho'\rangle\in R \wedge \mathcal{B}[\![\mathtt{B}]\!]\rho'\}\\
\overline{\mathsf{test}}^r[\![\mathtt{B}]\!]\ R & \triangleq & \{\langle\rho,\ \rho'\rangle \mid \langle\rho,\ \rho'\rangle\in R \wedge \neg\mathcal{B}[\![\mathtt{B}]\!]\rho'\}
\end{array}$$

extended to the following abstraction of the prefix trace domain $\mathbb{D}^{pt}$

$$\begin{array}{rcl}
\mathcal{D}^r(\!|\mathbb{D}^r|\!) & \triangleq & \langle \mathbb{P}^r,\ \dot{\subseteq}_2,\ \dot{\emptyset}_2,\ \dot{\cup}_2,\ \mathsf{program}^r,\ \mathsf{empty}^r,\ \mathsf{assign}^r,\ \mathsf{skip}^r,\ \mathsf{test}^r,\\
& & \overline{\mathsf{test}}^r,\ \mathsf{break}^r,\ \mathsf{compound}^r,\ \mathsf{breakable}^r\rangle
\end{array} \tag{50}$$

such that $\mathbb{P}^r \triangleq \wp(\mathbb{E}\times\mathbb{E}) \xrightarrow{\nearrow} \wp(\mathbb{E}\times\mathbb{E})$, $\langle \mathbb{P}^r,\ \dot{\subseteq}_2,\ \dot{\emptyset}_2,\ \dot{\cup}_2\rangle$ is the pointwise extension of the complete latice $\langle \wp(\mathbb{E}\times\mathbb{E}),\ \dot{\subseteq}_2,\ \dot{\emptyset}_2,\ \dot{\cup}_2\rangle$ ordered componentwise, and

$$\begin{array}{rcl}
\mathsf{program}^r[\![\ell_0,\ell_1]\!]R \ = \ \mathsf{empty}^r[\![\ell_0,\ell_1]\!]R \ = \ \mathsf{skip}^r[\![\ell_0,\ell_1]\!]R \ = \ \mathsf{break}^r[\![\ell_0,\ell_1]\!]R \ = & & \\
\mathsf{compound}^r[\![\ell_0,\ell_1]\!]R \ = \ \mathsf{breakable}^r[\![\ell_0,\ell_1]\!]R & \triangleq & R\\
\mathsf{assign}^r[\![\ell_0,\mathtt{x},\mathtt{A},\ell_1]\!]R & \triangleq & \mathsf{assign}^r[\![\mathtt{x},\mathtt{A}]\!]R\\
\mathsf{test}^r[\![\ell_0,\mathtt{B},\ell_t]\!]\ R & \triangleq & \mathsf{test}^r[\![\mathtt{B}]\!]\ R\\
\overline{\mathsf{test}}^r[\![\ell_0,\mathtt{B},\ell_t]\!]\ R & \triangleq & \overline{\mathsf{test}}^r[\![\mathtt{B}]\!]\ R
\end{array}$$

10.4 Relational Semantics

The relational semantics relates the final values of the variables to their initial values.

$$\mathcal{S}^r(\!|\mathbb{D}^r|\!) \triangleq \mathcal{S}^m(\!|\mathcal{D}^r(\!|\mathbb{D}^r|\!)|\!) = \mathcal{S}(\!|\mathcal{D}^m(\!|\mathcal{D}^r(\!|\mathbb{D}^r|\!)|\!)|\!)$$

11 Program Equivalence

The program transformations we consider must preserve the relational semantics, either for all inputs

$$(\mathtt{P} \equiv \mathtt{P}') \triangleq (\mathcal{S}^r[\![\mathtt{P}]\!]\mathbb{1}_{\mathbb{E}} = \mathcal{S}^r[\![\mathtt{P}']\!])\mathbb{1}_{\mathbb{E}} \tag{51}$$

or for some precondition $R \in \wp(\mathbb{E} \times \mathbb{E})$

$$(\mathtt{P} \equiv_R \mathtt{P}') \triangleq (\mathcal{S}^r[\![\mathtt{P}]\!]R = \mathcal{S}^r[\![\mathtt{P}']\!]R)$$

or for given input data $\rho \in \mathbb{E}$

$$(\mathtt{P} \equiv_\rho \mathtt{P}') \triangleq (\mathcal{S}^r[\![\mathtt{P}]\!]\{\langle\rho,\ \rho\rangle\} = \mathcal{S}^r[\![\mathtt{P}']\!]\{\langle\rho,\ \rho\rangle\})$$

In all cases this is an equivalence relation.

12 The Poset of Program Components

We can define a partial order on program components by requiring they are semantically equivalent and one is more efficient that the other:

$$\begin{aligned} \mathtt{P} \preccurlyeq \mathtt{P}' &\triangleq \mathtt{P} \equiv \mathtt{P}' \wedge \beta[\![\mathtt{P}]\!] \leqslant \beta[\![\mathtt{P}']\!] \\ \mathtt{P} \prec \mathtt{P}' &\triangleq \mathtt{P} \equiv \mathtt{P}' \wedge \beta[\![\mathtt{P}]\!] < \beta[\![\mathtt{P}']\!] \end{aligned}$$

$\langle \mathsf{Pc},\ \preccurlyeq \rangle$ is a poset (and $\langle \mathsf{Pc},\ \prec \rangle$ is a strict one). This poset has no infinite strictly decreasing chain, so any subset has a minimum. In general we reason on a given program $\mathtt{P}$ and we are interested in exploring the downset $\downarrow^{\preccurlyeq}[\![\mathtt{P}]\!]$ of equivalent and more efficient programs. $\downarrow^{\preccurlyeq}[\![\mathtt{P}]\!]$ is a finite total order hence both a join-lattice and a CPO.

13 Program Optimization

At this point, we have formalized our program optimization objective. Given a program P, transform it into an optimized program $\mathtt{P}' \in \min_{\preccurlyeq}\{\mathtt{P}'' \mid \mathtt{P}'' \preccurlyeq \mathtt{P}\}$ which is equivalent with a minimal cost that is the infimum of $\downarrow^{\preccurlyeq}[\![\mathtt{P}]\!]$. Because the problem is not decidable, we must resort to a sound but maybe not optimal solution $\mathtt{P}' \in \downarrow^{\preccurlyeq}[\![\mathtt{P}]\!]$. We compute $\mathtt{P}'$ by unrolling program P. Then we discuss how to optimize the unrolled program $\mathtt{P}'$, and finally how to unroll and optimize simultaneously.

Note that the above formalization covers program size optimization. It can be extended to include data size by defining the size of data in an environment. The objective is to minimize the pair of program and data sizes. The ideal measure would be the size of the generated circuit but it is difficult to relate it to these program and data sizes.

14 Full Unrolling

We formalize full program unrolling as an instance of the abstract interpreter. Note that we unroll a program into another program, but other transformations such as SSA [36], which is an abstract interpretation, could be equally considered.

14.1 Bounding Unrolling

We augment the language with an `error;` statement in case the execution of the unrolled program $\mathtt{P}^u$ for some input requires more than $\beta[\![\mathtt{P}]\!]$ steps, so that the unrolled program will be prematurely terminated by executing the `error;` statement (e.g. implemented as throwing an error).

S	::=		statement $\mathtt{S} \in \mathsf{S}^e$
		`x = A ;`	assignment
	|	`;`	skip
	|	`if (B) S`	conditional
	|	`if (B) S else S`	
	|	`while (B) S`	iteration
	|	`break ;`	iteration break
	|	`{ Sl }`	compound statement
	|	`{| Sl |}`	breakable statement
	|	`error;`	erroneous termination of an overthrown computation

$\texttt{Sl} \;::=\; \texttt{Sl}\ \texttt{S} \mid \epsilon$ statement list $\texttt{Sl} \in \mathsf{Sl}^e$, ϵ is the empty string

$\texttt{P}^e \;::=\; \texttt{Sl}$ program $\texttt{P}^e \in \mathsf{P}^e$

For each loop we assume that a maximal number $\beta[\![\texttt{while (B) S}]\!]$ of iterations is given, otherwise the loop is terminated in error. As stated in section 7, this can be over approximated by a static analysis.

14.2 Unrolling Abstract Domain

Program unrolling is the instance of the abstract interpreter $\mathcal{S}$ for the unrolling abstract domain

$$\begin{array}{rcl} \mathbb{D}^u & \triangleq & \langle \mathbb{P}^u,\ \mathsf{program}^u,\ \mathsf{stmtlist}^u,\ \mathsf{empty}^u,\ \mathsf{assign}^u,\ \mathsf{skip}^u,\ \mathsf{if}^u,\ \mathsf{ife}^u, \\ & & \mathsf{iter}^u,\ \mathsf{break}^u,\ \mathsf{compound}^u,\ \mathsf{breakable}^u \rangle \end{array} \tag{52}$$

defined as follows:

- *Unrolling semantics domain* $\mathbb{P}^u \triangleq \mathsf{P}^e \times \mathbb{Z}$, partially ordered by $\langle \texttt{P},\ n\rangle \sqsubseteq^u \langle \texttt{P}',\ m\rangle \triangleq \texttt{P} \equiv \texttt{P}' \wedge n \leqslant m$.

- *Unrolling semantics of a program* $\texttt{P} ::= \texttt{Sl}$

$$\begin{array}{rcl} \mathsf{program}^u[\![\texttt{P}]\!](Sl)\langle \texttt{P},\ n\rangle & \triangleq & \mathsf{let}\ \langle \texttt{Sl}',\ m\rangle = Sl\langle \texttt{Sl},\ n\rangle\ \mathsf{in} \\ & & (m < 0\ \texttt{?}\ \langle \texttt{Sl}'\ \texttt{error;},\ m\rangle\ \texttt{:}\ \langle \texttt{Sl}',\ m\rangle) \end{array} \tag{53}$$

- *Unrolling semantics of a statement list* $\texttt{Sl} ::= \texttt{Sl}'\ \texttt{S}$

$$\begin{array}{rcl} \mathsf{stmtlist}^u[\![\texttt{Sl}]\!](Sl', S)\langle \texttt{Sl},\ n\rangle & \triangleq & \mathsf{let}\ \langle \texttt{Sl}'',\ m\rangle = Sl'\langle \texttt{Sl}',\ n\rangle\ \mathsf{in} \\ & & \quad \mathsf{let}\ \langle \texttt{S}'',\ p\rangle = S\langle \texttt{S},\ m\rangle\ \mathsf{in} \\ & & \quad\ \langle \texttt{Sl}''\ \texttt{S}'',\ p\rangle \end{array} \tag{54}$$

- *Unrolling semantics of an empty statement list* $\texttt{Sl} ::= \epsilon$

$$\mathsf{empty}^u[\![\texttt{Sl}]\!]\langle \texttt{Sl},\ n\rangle \triangleq \langle \texttt{Sl},\ n\rangle \qquad \text{(no computation is involved)} \tag{55}$$

- *Unrolling semantics of an assignment statement* $\texttt{S} ::= \texttt{x = A ;}$

$$\mathsf{assign}^u[\![\texttt{S}]\!]\langle \texttt{S},\ n\rangle \triangleq \langle \texttt{S},\ n-1\rangle^{9} \tag{56}$$

[9] Instead of a cost of 1, we can assign reflecting the complexity of the arithmetic expression `A` so as to ensure that optimizations reduce the cost of evaluating this expression `A`.

- *Unrolling semantics of a skip statement* $\mathtt{S} ::= \mathtt{;}$

$$\mathsf{skip}^u[\![\mathtt{S}]\!]\langle \mathtt{S},\, n\rangle \triangleq \langle \mathtt{S},\, n-1\rangle \qquad (\text{a } \mathtt{NOP} \text{ is generated}) \tag{57}$$

- *Unrolling semantics of a conditional statement* $\mathtt{S} ::= \mathtt{if\ (B)\ } \mathtt{S}_t$

$$\begin{array}{rcl} \mathsf{if}^u[\![\mathtt{S}]\!](S_t)\langle \mathtt{S},\, n\rangle & \triangleq & \mathsf{let}\ \langle \mathtt{S}'_t,\, n_t\rangle = S_t\langle \mathtt{S}_t,\, n-1\rangle^{10}\ \mathsf{in} \\ & & \langle \mathtt{if\ (B)\ } \mathtt{S}'_t,\, n_t\rangle \end{array} \tag{58}$$

- *Unrolling semantics of a conditional statement* $\mathtt{S} ::= \mathtt{if\ (B)\ } \mathtt{S}_t \mathtt{\ else\ } \mathtt{S}_f$

$$\begin{array}{rcl} \mathsf{ife}^u[\![\mathtt{S}]\!](S_t, S_f)\langle \mathtt{S},\, n\rangle & \triangleq & \mathsf{let}\ \langle \mathtt{S}'_t,\, n_t\rangle = S_t\langle \mathtt{S}_t,\, n-1\rangle \\ & & \mathsf{and}\ \langle \mathtt{S}'_f,\, n_f\rangle = S_f\langle \mathtt{S}_t,\, n-1\rangle\ \mathsf{in} \\ & & \langle \mathtt{if\ (B)\ } \mathtt{S}'_t \mathtt{\ else\ } \mathtt{S}'_f,\, \max(n_t, n_f)\rangle \end{array} \tag{59}$$

- *Unrolling semantics of an iteration statement* $\mathtt{S} ::= \mathtt{while\ (B)\ } \mathtt{S}_b$

$$\begin{array}{rcl} \mathcal{F}^u[\![\mathtt{S}]\!](S_b)\langle \mathtt{S},\, k\rangle & \triangleq & (\,(k \leqslant 0)\ \mathtt{?}\ \langle \mathsf{error}\mathtt{;},\, 0\rangle \\ & & \mathtt{:}\ \mathsf{let}\ \langle \mathtt{S}'_b,\, m\rangle = S_b\langle \mathtt{S}_b,\, k\rangle\ \mathsf{in} \\ & & \quad \mathsf{let}\ \langle \mathtt{S}',\, p\rangle = \mathcal{F}^u[\![\mathtt{S}]\!](S_b)\langle \mathtt{S},\, m\rangle\ \mathsf{in} \\ & & \quad \langle \mathtt{if\ (B)\ \{|\ } \mathtt{S}'_b\ \mathtt{S}'\ \mathtt{|\}},\, p-1\rangle\,) \\ \mathsf{iter}^u[\![\mathtt{S}_b]\!](S_b)\langle \mathtt{S},\, n\rangle & \triangleq & \mathsf{let}\ \langle \mathtt{S}'',\, p\rangle = \mathcal{F}^u[\![\mathtt{S}]\!](S_b)\langle \mathtt{S},\, \beta[\![\mathtt{S}]\!]\rangle\ \mathsf{in} \\ & & \langle \mathtt{S}'',\, n - (\beta[\![\mathtt{S}]\!] - p)\rangle \end{array} \tag{60}$$

(where $\beta[\![\mathtt{S}]\!] \in \mathsf{Pc} \rightarrowtail \mathbb{N}$ specifies a sound bound (32) on the number of steps in the loop $\mathtt{S}$ which deduced from the global counter n)

- *Unrolling semantics of a break statement* $\mathtt{S} ::= \mathtt{break\ ;}$

$$\mathsf{break}[\![\mathtt{S}]\!]\langle \mathtt{S},\, n\rangle \triangleq \langle \mathtt{S},\, n-1\rangle \tag{61}$$

- *Unrolling semantics of a compound statement* $\mathtt{S} ::= \mathtt{\{\ Sl\ \}}$

$$\begin{array}{rcl} \mathsf{compound}[\![\mathtt{S}]\!](Sl)\langle \mathtt{S},\, n\rangle & \triangleq & \mathsf{let}\ \langle \mathtt{Sl}',\, m\rangle = Sl(\langle \mathtt{Sl},\, n\rangle)\ \mathsf{in} \\ & & \langle \mathtt{\{\ } \mathtt{Sl}'\ \mathtt{\}},\, m\rangle \end{array} \tag{62}$$

- *Unrolling semantics of a breakable statement* $\mathtt{S} ::= \mathtt{\{|\ Sl\ |\}}$

$$\begin{array}{rcl} \mathsf{breakable}[\![\mathtt{S}]\!](Sl)\langle \mathtt{S},\, n\rangle & \triangleq & \mathsf{let}\ \langle \mathtt{Sl}',\, m\rangle = Sl(\langle \mathtt{Sl},\, n\rangle)\ \mathsf{in} \\ & & \langle \mathtt{\{|\ } \mathtt{Sl}'\ \mathtt{|\}},\, m\rangle \end{array} \tag{63}$$

[10] Again the cost of 1 would better be refined to better reflect the complexity of evaluating B.

14.3 Unroller

The unrolling of a program component is

$$\mathcal{S}^u[\![\mathtt{P}]\!] \triangleq \mathcal{S}(\!|\mathbb{D}^u|\!)[\![\mathtt{P}]\!] \tag{64}$$

The unrolling of a program $\mathtt{P}$ with bound β on the program and loops is $\mathcal{S}^u[\![\mathtt{P}]\!](\langle \mathtt{P}, \beta[\![\mathtt{P}]\!]\rangle)$.

15 Semantic Equivalence of the Program and Its Full Unrolling

The guarantee provided by the unrolling is that the program and its unrolling have the same abstract prefix semantics:

$$\forall \mathtt{S} \in \mathsf{Pc} \,.\, \forall n \in \mathbb{N}^+ \,.\, (\langle \mathtt{S}^u,\, m\rangle = \mathcal{S}^u[\![\mathtt{S}]\!]\langle \mathtt{S},\, n\rangle \wedge m \geqslant 0) \Rightarrow \mathcal{S}^p(\!|\mathbb{D}^p|\!)[\![\mathtt{S}]\!] = \mathcal{S}^p(\!|\mathbb{D}^p|\!)[\![\mathtt{S}^u]\!]$$

The proof is by structural induction on the program component $\mathtt{S}$.

So, by instantiation, $\mathtt{S}$ and $\mathtt{S}^u$ have the same prefix trace semantics (31), and so, by abstraction (34) the same maximal abstract semantics and by instantiation (47), the same maximal trace semantics. It follows, by abstraction (49), that they have the same relational semantics and so, by definition (51) of equivalence, that they are equivalent $\mathtt{S}^u \equiv \mathtt{S}$. In conclusion the informal requirement (1) is satisfied.

16 Accuracy Estimation for Floating-point Computations

The equivalence of the program $\mathtt{P}$ and its unrolling $\mathtt{P}^u$ relies on the fact that they both use the same evaluations $\mathcal{A}$ of arithmetic expressions and $\mathcal{B}$ of Boolean expressions. This is a reasonable assumption for different machines and compilers, except for floating point computations.

The problem with float computations is that they are not exactly reproducible on different machines. For example 64 bits Intel processors have a float division performed on 80 bits and may produce results different from the IEEE 754 in double precision [33], not necessarily more precise and this can lead to catastrophic results. So the verifier should consider that the float results certified by the prover are valid up to some round-off error margin ϵ.

$$\begin{array}{l}\mathsf{let}\ \langle \rho,\, \rho'\rangle = \mathcal{S}^r[\![\mathtt{P}]\!](\langle \rho,\, \rho\rangle)\ \mathsf{and}\ \langle \rho,\, \rho''\rangle = \mathcal{S}^r[\![\mathcal{S}^u[\![\mathtt{P}]\!]]\!](\langle \rho,\, \rho\rangle)\ \mathsf{in}\\ \quad \forall \mathtt{x} \in \mathsf{V} \,.\, \rho'(\mathtt{x}) \in [\rho''(\mathtt{x}) - \epsilon, \rho''(\mathtt{x}) + \epsilon]\end{array}$$

16.1 Static Estimate of the Round-off Error

One possible solution to estimate this round-off error ϵ would be to determine the possible interval of float values using static analyzers such as Fluctuat [21, 22] which unroll the program and would perform better on an optimized unrolled program. This analysis is an instance of the static analysis framework considered in next section, and so can be formalized as an instance of the abstract interpreter.

16.2 Dynamic Estimate of the Round-off Error

A more precise estimation of accuracy could also be determined by dynamic interval analysis, also know as interval arithmetics in numerical analysis. Since it is an abstract interpretation (see [11, chapter 32]), it can be incorporated by a product to the program unrolling.

17 Static Program Analysis

Program analysis is an instance of an abstract interpreter for a reduced product [11, chapter 36] of abstract domains. For example, Astrée [19] has more than 50 abstract domains, that can be chosen on demand (e.g. filters for control/command programs, quadruples for spatial programs, etc) with an efficient approximation of the reduced product [18]. Abstract domain libraries provide reusable collections of abstract domains (like APRON [25] or ELINA [37] for numerical domains including intervals [14], octagons [32], zonotopes [23], and polyhedra [17]).

17.1 Abstract Domain of a Static Analysis

A static analysis is fully specified by an abstract domain

$$\mathbb{D}^a \triangleq \langle \mathbb{P}^a, \sqsubseteq^a, \bot^a, \sqcup^a, \mathsf{assign}^a, \mathsf{test}^a, \overline{\mathsf{test}}^a \rangle \tag{65}$$

This abstract domain is in general complex and decomposed into many subdomains composed, for example by a reduced product, which effect is to present the composition of these subdomains in the form (66).

The functor $\mathcal{D}^a$:

$$\begin{aligned} \mathcal{D}^a(\!|\mathbb{D}^a|\!) \triangleq \langle & \mathbb{P}^a, \sqsubseteq^a, \bot^a, \sqcup^a, \mathsf{program}^a, \mathsf{empty}^a, \mathsf{assign}^a, \mathsf{skip}^a, \mathsf{test}^a, \\ & \overline{\mathsf{test}}^a, \mathsf{break}^a, \mathsf{compound}^a, \mathsf{breakable}^a \rangle \end{aligned} \tag{66}$$

defined (similarly to $\mathcal{D}^r$ at (50)) as:

$$\begin{array}{rcl}
\mathsf{program}^a[\![\ell_0,\ell_1]\!]R & = & \mathsf{empty}^a[\![\ell_0,\ell_1]\!]R \;=\; \mathsf{skip}^a[\![\ell_0,\ell_1]\!]R \;=\; \mathsf{break}^a[\![\ell_0,\ell_1]\!]R \;= \\
\mathsf{compound}^a[\![\ell_0,\ell_1]\!]R & = & \mathsf{breakable}^a[\![\ell_0,\ell_1]\!]R \;\triangleq\; R \\
\mathsf{assign}^a[\![\ell_0,\mathtt{x},\mathtt{A},\ell_1]\!]R & \triangleq & \mathsf{assign}^a[\![\mathtt{x},\mathtt{A}]\!]R \\
\mathsf{test}^a[\![\ell_0,\mathtt{B},\ell_t]\!]\,R & \triangleq & \mathsf{test}^a[\![\mathtt{B}]\!]\,R \\
\overline{\mathsf{test}}^a[\![\ell_0,\mathtt{B},\ell_t]\!]\,R & \triangleq & \overline{\mathsf{test}}^a[\![\mathtt{B}]\!]\,R
\end{array}$$

provides an abstract domain of the same type as $\mathbb{D}^p$ in (14), and so, combined with the functor $\mathcal{D}^m$ yields a static analyzer for properties $\mathbb{P}^a$ by instantiation of the abstract interpreter.

17.2 Specification of a Static Analyzer

$$\mathcal{S}^a(\!|\mathbb{D}^a|\!) \;\triangleq\; \mathcal{S}^m(\!|\mathcal{D}^a(\!|\mathbb{D}^a|\!)|\!) \;=\; \mathcal{S}(\!|\mathcal{D}^m(\!|\mathcal{D}^a(\!|\mathbb{D}^a|\!)|\!)|\!)$$

17.3 Soundness of a Static Analysis

If the abstract domain defining the program semantics for a given abstract interpreter $\mathcal{S}^s$ is

$$\begin{array}{rcl}
\mathbb{D}^s & \triangleq & \langle \mathbb{P}^s,\ \sqsubseteq^s,\ \bot^s,\ \sqcup^s,\ \mathsf{assign}^s,\ \mathsf{test}^s,\ \overline{\mathsf{test}}^s \rangle \\
\mathcal{S}^s(\!|\mathbb{D}^s|\!) & \triangleq & \mathcal{S}^m(\!|\mathcal{D}^s(\!|\mathbb{D}^s|\!)|\!) \;=\; \mathcal{S}(\!|\mathcal{D}^m(\!|\mathcal{D}^s(\!|\mathbb{D}^s|\!)|\!)|\!)
\end{array} \tag{67}$$

and the abstract domain defining the program analysis for this same abstract interpreter $\mathcal{S}^a$ is (67) then, given an increasing concretization function:

$$\gamma \;\in\; \mathbb{P}^a \xrightarrow{\nearrow} \mathbb{P}^s,$$

the following pointwise *local soundness* conditions:

$$\begin{array}{c}
\mathsf{assign}^s[\![\mathtt{x = A\ ;}]\!] \circ \gamma \mathrel{\dot{\sqsubseteq}^s} \gamma \circ \mathsf{assign}^a[\![\mathtt{x = A\ ;}]\!] \\
\mathsf{test}^p[\![\mathtt{B}]\!] \circ \gamma \mathrel{\dot{\sqsubseteq}^s} \gamma \circ \mathsf{test}^p[\![\mathtt{B}]\!] \\
\overline{\mathsf{test}}^p[\![\mathtt{B}]\!] \circ \gamma \mathrel{\dot{\sqsubseteq}^s} \gamma \circ \overline{\mathsf{test}}^p[\![\mathtt{B}]\!]
\end{array}$$

ensure that the analysis is sound with respect to the semantics, that is:

$$\forall \mathtt{S} \in \mathsf{Pc}\ .\ \mathcal{S}^s(\!|\mathbb{D}^s|\!)[\![\mathtt{S}]\!] \mathrel{\dot{\sqsubseteq}^s} \gamma(\mathcal{S}^a(\!|\mathbb{D}^a|\!)[\![\mathtt{S}]\!])$$

(see the proof in [11, chapter 21]). It follows that any program component property proved in the abstract is valid, after concretization, in the concrete.

Classical static analysis aims at inferring invariants at each program point by abstraction of the prefix trace semantics, in which case $\mathcal{S}^s(\!|\mathbb{D}^s|\!)$ is the prefix trace semantics $\mathcal{S}^{pt}(\!|\mathbb{D}^{pt}|\!)$.

If any of these classical static analyzes if applied to the unrolled program, it will produce (under the boundedness hypothesis) refined results because no extrapolation (widening) / interpolation (narrowing) is necessary [11, chapter 34].

18 Direct Product of Unrolling and Analysis

The unrolling and static analysis are both instance of the abstract interpreter, respectively for abstract domain $\mathbb{D}^u$ and $\mathcal{D}^m(\!|\mathcal{D}^a(\!|\mathbb{D}^a|\!)|\!)$. Therefore, instead of first enrolling and then analyzing (and then transforming), we can perform both simultaneously thanks to a direct product $\mathbb{D}^u \times \mathcal{D}^m(\!|\mathcal{D}^a(\!|\mathbb{D}^a|\!)|\!)$ of the abstract domains [11, definition 36.1]. This does not brings in any gain in precision and performance, but is an essential step to be able to perform reductions [11, chapter 29].

19 Reduced Product by Program Transformation

The unrolling domain $\mathbb{D}^u$ cannot improve the static analysis domain $\mathcal{D}^m(\!|\mathcal{D}^a(\!|\mathbb{D}^a|\!)|\!)$ because the analysis is performed (simultaneously in the direct product) on the already unrolled prefix of the unrolled program.

However, the static analysis domain $\mathcal{D}^m(\!|\mathcal{D}^a(\!|\mathbb{D}^a|\!)|\!)$ can improve the unrolling domain $\mathbb{D}^u$ by performing a program transformation, as discussed in section 13. It follows that the direct product can be replaced by the reduced product $\mathbb{D}^u \otimes \mathcal{D}^m(\!|\mathcal{D}^a(\!|\mathbb{D}^a|\!)|\!)$ of the two domains [11, definition 36.7]. As shown by [11, theorem 36.23], the reduced product is obtained by applying a meaning-preserving reduction operator $\mathbb{r}$ to the elements of the direct product. However, because the optimal program transformation problem is undecidable, the reduced product is not effectively computable, and so neither is this reduction operator $\mathbb{r}$. The solution is to define a weaker reduction operator $\mathbb{t} \in (\mathbb{D}^u \times \mathcal{D}^m(\!|\mathcal{D}^a(\!|\mathbb{D}^a|\!)|\!)) \to (\mathbb{D}^u \times \mathcal{D}^m(\!|\mathcal{D}^a(\!|\mathbb{D}^a|\!)|\!))$ which, given a statement $\mathtt{S} \in \mathsf{S}$ and an abstract property $P \in \mathbb{P}^a$, returns an equivalent and more efficient transformed statement

$$\mathtt{S}' \quad = \quad \mathbb{t}(\mathtt{S}, P)$$

such that $\mathtt{S}' \preccurlyeq \mathtt{S}$ and preferably $\mathtt{S}' \prec \mathtt{S}$. Notice that in general, although $\mathtt{S}' \equiv \mathtt{S}$

the property $P' = \mathcal{S}^a(\!|\mathbb{D}^a|\!)[\![\mathtt{S}']\!]$ of the transformed program may be different from $P = \mathcal{S}^a(\!|\mathbb{D}^a|\!)[\![\mathtt{S}]\!]$. So the reduction operator can be defined as

$$\begin{array}{rcl} \mathbb{t}(\mathtt{S}, P) & = & \mathsf{let}\, \mathtt{S}' = \mathbb{t}(\mathtt{S}, P)\, \mathsf{in} \\ & & \quad \langle \mathtt{S}',\ P \sqcap^a \mathcal{S}^a(\!|\mathbb{D}^a|\!)[\![\mathtt{S}']\!] \rangle \end{array}^{11}$$

It follows that the pair $\langle \mathtt{S},\ P \rangle$ can be replaced by $\mathbb{t}(\mathtt{S}, P)$. Observe that if $\mathbb{t}$ is sound, increasing, and reductive then, by [11, theorem 29.2], the transformation can be improved by considering $\check{\mathbb{t}}(\mathtt{S}, P) \triangleq \mathsf{gfp}^{\sqsubseteq_2}_{\langle \mathtt{S},\ P \rangle} \mathbb{t}$ where $\sqsubseteq_2$ is the componentwise partial order on the product domain $\mathbb{D}^u \times \mathcal{D}^m(\!|\mathcal{D}^a(\!|\mathbb{D}^a|\!)|\!)$. Since the iterations to compute the greatest fixpoint might be costly, [11, corollary 29.3] shows that it is sound, but less precise, to stop at any iterate, including doing none. Applied at the program level, $\check{\mathbb{t}}$ essentially consists in iterating the abstract interpreter.

20 Unrolling, Analysis, and Transformation All Together

The static analysis and transformation can be done during the unrolling. We consider a static analysis which, given a precondition, returns the pair of a postcondition for normal termination and for termination by a **`break ;`** (hence over approximating the relational analysis of section 10). It is also possible to add an invariant based on a reachability analysis [11, chapter 47] for the prefix trace semantics of section 6.8 and moreover, to return a refined precondition, using a backward analysis [11, chapter 50] or an iterated combination of a forward and backward analysis [11, chapter 51].

The reductive transformation $\mathbb{t}[\![\mathtt{S}]\!]$ of a program component `S`, take as parameters its transformed components, the precondition and the pair of termination and break postcondition resulting from the analysis, and returns the optimize program component with the postconditions.

- *Unrolling, analysis, and transformation semantics domain* $\mathbb{P}^{uat} \triangleq (\mathsf{P}^e \times \mathbb{Z}) \to \mathbb{P}^a \to ((\mathsf{P}^e \times \mathbb{Z}) \times (\mathbb{P}^a \times \mathbb{P}^a))$, partially ordered componentwise.

- *Unrolling, analysis, and transformation semantics of a program* $\mathtt{P} ::= \mathtt{Sl}$

$$\begin{array}{rcl} \mathsf{program}^{uat}[\![\mathtt{P}]\!]\langle \mathtt{P},\ n \rangle R & \triangleq & \mathsf{let}\, \langle \langle \mathtt{Sl}',\ m \rangle,\ \langle T,\ B \rangle \rangle = Sl\langle \mathtt{Sl},\ n \rangle\, \mathsf{in} \\ & & \quad (m < 0\ \mathbf{?}\ \mathbb{t}[\![\mathtt{P}]\!](\langle \mathtt{Sl}'\ \mathsf{error;},\ m \rangle, R, \langle \bot,\ B \rangle) \\ & & \quad \mathbf{:}\ \mathbb{t}[\![\mathtt{P}]\!](\langle \mathtt{Sl}',\ m \rangle, R, \langle T,\ B \rangle)\,) \end{array} \quad (68)$$

[11] $\langle \mathtt{S}',\ P \rangle$ is also sound, although less precise.

- *Unrolling, analysis, and transformation semantics of a statement list* $\texttt{Sl} ::= \texttt{Sl}'\ \texttt{S}$

$$\begin{aligned}&\mathsf{stmtlist}^{uat}[\![\texttt{Sl}]\!](Sl', S)\langle \texttt{Sl},\ n\rangle R \ \triangleq \\ &\quad \mathsf{let}\ \langle\langle \texttt{Sl}'',\ m\rangle,\ \langle T',\ B'\rangle\rangle = Sl'\langle \texttt{Sl}',\ n-1\rangle R\ \mathsf{in} \\ &\qquad \mathsf{let}\ \langle\langle \texttt{S}'',\ p\rangle,\ \langle T'',\ B''\rangle\rangle = S\langle \texttt{S},\ m\rangle T'\ \mathsf{in} \\ &\qquad\quad \mathbb{t}[\![\texttt{Sl}]\!](\langle \texttt{Sl}''\ \texttt{S}'',\ p\rangle, R, \langle T'',\ B' \sqcup B''\rangle)\end{aligned} \tag{69}$$

- *Unrolling, analysis, and transformation semantics of an empty statement list* $\texttt{Sl} ::= \epsilon$

$$\mathsf{empty}^{uat}[\![\texttt{Sl}]\!]\langle \texttt{Sl},\ n\rangle R \ \triangleq\ \langle\langle \texttt{Sl},\ n\rangle,\ \langle R,\ \bot\rangle\rangle \tag{70}$$

- *Unrolling, analysis, and transformation semantics of an assignment statement* $\texttt{S} ::= \texttt{x = A ;}$

$$\mathsf{assign}^{uat}[\![\texttt{S}]\!]\langle \texttt{S},\ n\rangle R \ \triangleq\ \mathbb{t}[\![\texttt{S}]\!](\langle \texttt{S},\ n-1\rangle, R, \langle \mathsf{assign}^{a}[\![\texttt{x}, \texttt{A}]\!] R,\ \bot\rangle) \tag{71}$$

- *Unrolling, analysis, and transformation semantics of a skip statement* $\texttt{S} ::= \texttt{;}$

$$\mathsf{skip}^{uat}[\![\texttt{S}]\!]\langle \texttt{S},\ n\rangle R \ \triangleq\ \mathbb{t}[\![\texttt{S}]\!](\langle \texttt{S},\ n-1\rangle, R, \langle R,\ \bot\rangle) \tag{72}$$

- *Unrolling, analysis, and transformation semantics of a conditional statement* $\texttt{S} ::= \texttt{if (B)}\ \texttt{S}_t$

$$\begin{aligned}&\mathsf{if}^{uat}[\![\texttt{S}]\!](S_t)\langle \texttt{S},\ n\rangle R \ \triangleq\ \mathsf{let}\ \langle\langle \texttt{S}'_t,\ n_t\rangle,\ T,\ B\rangle = S_t\langle \texttt{S}_t,\ n\rangle(\mathsf{test}^{a}[\![\texttt{B}]\!]\ R)\ \mathsf{in} \\ &\qquad\qquad \mathbb{t}[\![\texttt{S}]\!](\langle \texttt{if (B)}\ \texttt{S}'_t,\ n_t\rangle, R, \langle T \sqcup \overline{\mathsf{test}}^{p}[\![\texttt{B}]\!]\ R,\ B\rangle)\end{aligned} \tag{73}$$

- *Unrolling, analysis, and transformation semantics of a conditional statement* $\texttt{S} ::= \texttt{if (B)}\ \texttt{S}_t\ \texttt{else}\ \texttt{S}_f$

$$\begin{aligned}&\mathsf{ife}^{uat}[\![\texttt{S}]\!](S_t, S_f)\langle \texttt{S},\ n\rangle R \ \triangleq \\ &\quad \mathsf{let}\ \langle\langle \texttt{S}'_t,\ n_t\rangle,\ T_t,\ B_t\rangle = S_t\langle \texttt{S}_t,\ n\rangle(\mathsf{test}^{a}[\![\texttt{B}]\!]\ R) \\ &\quad \mathsf{and}\ \langle\langle \texttt{S}'_f,\ n_f\rangle,\ T_f,\ B_f\rangle = S_f\langle \texttt{S}_f,\ n\rangle(\overline{\mathsf{test}}^{a}[\![\texttt{B}]\!]\ R)\ \mathsf{in} \\ &\qquad \mathbb{t}[\![\texttt{S}]\!](\langle \texttt{if (B)}\ \texttt{S}'_t\ \texttt{else}\ \texttt{S}'_f,\ \max(n_t, n_f)\rangle, R, \langle T_t \sqcup T_f,\ B_t \sqcup B_f\rangle)\end{aligned} \tag{74}$$

- *Unrolling, analysis, and transformation semantics of an iteration statement* $\texttt{S} ::= \texttt{while (B)}\ \texttt{S}_b$

$$\begin{aligned}&\mathcal{F}^{uat}[\![\texttt{S}]\!](S_b)\langle \texttt{S},\ k\rangle R \ \triangleq\ (\!|\ (k \leqslant 0)\ \texttt{?}\ \langle \mathsf{error}\texttt{;},\ 0\rangle \\ &\qquad\qquad \texttt{:}\ \mathsf{let}\ \langle\langle \texttt{S}'_b,\ m\rangle,\ \langle T_b,\ B_b\rangle\rangle = S_b\langle \texttt{S}_b,\ k\rangle R\ \mathsf{in} \\ &\qquad\qquad\quad \mathsf{let}\ \langle\langle \texttt{S}',\ p\rangle,\ \langle T',\ B'\rangle\rangle = \mathcal{F}^{uat}[\![\texttt{S}]\!](S_b)\langle \texttt{S},\ m\rangle T_b\ \mathsf{in} \\ &\qquad\qquad\qquad \mathbb{t}[\![\texttt{S}]\!](\langle \texttt{if (B) \{|}\ \texttt{S}'_b\ \texttt{S}'\ \texttt{|\}},\ p\rangle, R, \langle T',\ B_b \sqcup B'\rangle)\ |\!) \\ &\mathsf{iter}^{uat}[\![\texttt{S}_b]\!](S_b)\langle \texttt{S},\ n\rangle R \ \triangleq\ \mathsf{let}\ \langle\langle \texttt{S}'',\ p\rangle,\ \langle T,\ B\rangle\rangle = \mathcal{F}^{uat}[\![\texttt{S}]\!](S_b)\langle \texttt{S},\ \beta[\![\texttt{S}]\!]\rangle R\ \mathsf{in} \\ &\qquad\qquad \langle\langle \texttt{S}'',\ n - (\beta[\![\texttt{S}]\!] - p)\rangle,\ \langle T,\ B\rangle\rangle\end{aligned} \tag{75}$$

(where $\beta[\![\in]\!]\mathsf{Pc} \rightharpoonup \mathbb{N}$ specifies a sound bound (32) on the number of steps in the loop which deducted from the global counter n)

- *Unrolling, analysis, and transformation semantics of a break statement* $\mathtt{S} ::= \mathtt{break\ ;}$

$$\mathsf{break}[\![\mathtt{S}]\!]\langle \mathtt{S},\ n\rangle R \triangleq \mathbb{t}[\![\mathtt{S}]\!](\langle \mathtt{S},\ n-1\rangle, R, \langle \bot,\ R\rangle) \tag{76}$$

- *Unrolling, analysis, and transformation semantics of a compound statement* $\mathtt{S} ::= \mathtt{\{\ Sl\ \}}$

$$\begin{array}{l}\mathsf{compound}[\![\mathtt{S}]\!](Sl)\langle \mathtt{S},\ n\rangle R \triangleq \\ \quad \mathsf{let}\ \langle\langle \mathtt{Sl}',\ m\rangle,\ \langle T',\ B'\rangle\rangle = Sl(\langle \mathtt{Sl},\ n\rangle)R\ \mathsf{in} \\ \quad\quad \mathbb{t}[\![\mathtt{S}]\!](\langle \mathtt{\{\ Sl'\ \}},\ m\rangle, R, \langle T',\ B'\rangle)\end{array} \tag{77}$$

- *Unrolling, analysis, and transformation semantics of a breakable statement* $\mathtt{S} ::= \mathtt{\{|\ Sl\ |\}}$

$$\begin{array}{l}\mathsf{breakable}[\![\mathtt{S}]\!](Sl)\langle \mathtt{S},\ n\rangle R \triangleq \\ \quad \mathsf{let}\ \langle\langle \mathtt{Sl}',\ m\rangle,\ \langle T',\ B'\rangle\rangle = Sl(\langle \mathtt{Sl},\ n\rangle)R\ \mathsf{in} \\ \quad\quad \mathbb{t}[\![\mathtt{S}]\!](\langle \mathtt{\{|\ Sl'\ |\}},\ m\rangle, R, \langle T' \sqcup B',\ \bot\rangle)\end{array} \tag{78}$$

20.1 Streaming

Observe that, in full generality, the optimizing transformation $\mathbb{t}$ at (68) is global and performed on the whole unrolled program. This may be way too large to be managed efficiently. In that case, we could keep the optimizing transformation $\mathbb{t}$ at assignments (71), skips (72), tests (73), (74), and (75), and breaks (76) only. Then the unrolled program can be streamed out, thus considerably reducing memory consumption.

21 Program Optimizing Transformations and Corresponding Abstract Domains

The objective of the transformer reduction $\mathbb{t}[\![\mathtt{S}]\!](\langle \mathtt{S}',\ m\rangle, R, \langle T,\ B\rangle)$ is, knowing the transformed components $\mathtt{S}'$ of the program component $\mathtt{S}$, as well as the precondition R and pair $\langle T,\ B\rangle$ of postconditions on normal termination T and termination by a break B[12] must return a semantically equivalent transformed program component $\mathtt{S}''$ of $\mathtt{S}$ which is equivalent $\mathtt{S}'' \equiv \mathtt{S}$ and more efficient either in the size of the code (and ultimately of the corresponding circuit) and the necessary memory. Because

[12] and maybe other analyzes as already mentioned.

the problem is undecidable, the solution is necessarily an approximate but sound compromise.

It is extremely difficult to provide a transformer reduction independently of the application domain of the program. We review a few classical abstract domains and the corresponding transformations. They subsume partial evaluation, dead code elimination, and so forth [26].

21.1 Communication of the Transformer with Abstract Domains

In general the abstract domain $\mathbb{P}^a$ is the reduced product of many elementary abstract domains. In order to avoid modifying existing domains or the transformer each time an abstract domain is introduced or withdrawn, one can use a common interface, called a communication channel [18, 6], between abstract domains themselves and with the transformer; see [11, sction 36.4.6].

21.2 Reduction by Typing

In C, typing is very weak and static analysis is traditionally used to refine the C compiler typing algorithm. For example, in C, a boolean is an integer; 0 is false, different from 0 is true. An analysis could determine what integers are used as Booleans only and transform the program to $\{0, 1\}$ only (maybe with a pragma to indicate boolean type). Replacing an integer by a bit, will definitely reduce the size of the circuit.

21.3 Reduction by Determination of Side Effects

Although expressions in our subset of C have no side-effects, this is not the case in general. C compilers can evaluate expressions in any order, so different compilers may yield different results when expressions have different side effects. Moreover, for floats the order of evaluation may influence the accuracy of the result, which is similar to a side-effect.

A static analysis might checks for absence of side-effects in expressions and otherwise transform the source into a decomposed expressions with auxiliary variables enforcing an order of evaluation.

21.4 Elimination of Runtime Checks, Dead Code, and Variables

The elimination of runtime checks for integers, array indexes, floats, and so on essentially consists in bounding an arithmetic expression. Dead code follows from the constancy of Boolean expressions.

This can be solved, more or less precisely, by many abstract domains such as constancy, interval, congruences, zones, octagons, zonotopes, linear equalities, polyhedra, and so forth.

This information can also be used in optimizing the code in various ways. For example, linear equalities can be used for specialization (replacing variables by values) and to eliminate variables by substitution of their value in terms of the others. Another example is the encoding of variables with small values on less bits.

This information is also necessary when compiling code to an arithmetic circuit. For example $\max(a, b) = \frac{a+b}{2} + \frac{|a-b|}{2}$, except that $\max(a, b)$ does not overflow whereas $a + b$ and $|a - b|$ might (for example with $a =$ `INT_MIN` and $b = 0$ in 2's complement).

21.5 Code Simplification by Symbolic Execution

Symbolic execution is an abstract interpretation [9, section 3.4.5] which consists in providing symbolic names to the initial values and then calculating at each program point a path condition to reach that path and a symbolic value of the variables when reaching that program point. The traditional difficulties still unsolved over decades of research is the handling of loops and data structures such as array. The problem is solved by the unrolling (which also solve the array problem by considering each element as a simple variable). However, the implication problem remains. May formula simplifiers do exist and their effectiveness can only be determined by experimentation[13]

[13] See, for example, dCode https://www.dcode.fr/math-simplification, Fungrim https://fungrim.org, Julia https://discourse.julialang.org/t/how-to-simplify-symbolic-expression-using-calculusjl/32036, Maple https://www.maplesoft.com/support/help/Maple/view.aspx?path=simplify, Mathcad http://support.ptc.com/help/mathcad/en/index.html#page/PTC_Mathcad_Help/example_simplify_rewrite_expressions.html, Mathematica https://reference.wolfram.com/language/tutorial/AlgebraicCalculations.html, MATLAB https://www.mathworks.com/help/symbolic/formula-manipulation-and-simplification.html, Maude http://maude.cs.illinois.edu/w/index.php/The_Maude_System, Maxima https://maxima.sourceforge.io/docs/manual/maxima_46.html, Octave https://brandonrozek.com/blog/2017-03-09-simplifying-expressions-octave/, Racket https://docs.racket-lang.org/symalg/index.html, Sage https://doc.sagemath.org,

21.6 Code Elimination by Dependency Analysis

Dependency analysis [11, chapter 47] is a generalization of interference, dye, tracking, taint, binding time and other static analyses. It determines at each program point which variables depend on which inputs. Depend means that changing the inputs will change the observation of the variable in the computation from that program point on. In our case, the observation is defined by the relational analysis.

Assume only some of the results of a program are of interest. If the results of interest can be determined to depend on some $D \setminus I$ of the inputs I by a static analysis. All intermediate computations depending only on the other inputs in $I \setminus D$ can be eliminated. This is because if the results of interest were depending on these intermediate computations, they would, indirectly, depends on these other inputs $I \setminus D$, in contradiction with their independence, as soundly determined by the overapproximating analysis.

This forward slicing is different from classic slicing which proceeds backwards and does not take values of variables into account.

21.7 Code Improvement for Float Computations

Since float computations are in a finite field they do not have the same mathematical properties as real expressions. So equivalent expressions in the reals may have different results with floats. Given the float transliteration of a real expression in a program, it is interesting to replace it by the the equivalent one in the reals that is the most precise one (or close to the most precise one) in the floats. This can be supported by a static analysis; see for example [20]. [2] provides examples of floating-point computations occurring in programs that may be transformed in order to improve their numerical accuracy.

Such static analyzes can also be used for mixed computations where float computations that need not be precise are replaced by faster ones on less bits [5].

21.8 Strengthening User-Provided Invariants

Analysis is more difficult than verification [16]. The reason is that to prove program properties it is often necessary to strengthen them to be inductive [10], and the main objective of a static analyzer is to infer such inductive properties. Consider the following example from [6]:

---

simplt `https://rdrr.io/cran/simplr/`, SymPy `https://docs.sympy.org/`, and so on; see `https://en.wikipedia.org/wiki/Computer_algebra`.

```
int f(int x) {
   int low = x & 0xffff;
   int high = x >> 16;
   int rebuilt = low | (high << 16)
   return rebuilt
}
```

which is a complicated way to do nothing, that is `x` = `rebuilt`. SMT solvers with appropriate bit-string theories might be able to prove this postcondition. However, it is another story to discover it, as well as the necessary intermediate invariants, in particular with iteration or recursion. Astrée S infer this postcondition fully automatically without any user-provided information [6, section 18.1.2].

It follows that beyond the unroll/analyze/transform task, the abstract interpreter may be simultaneously used to generate inductive invariants needed to translate programs in high- level source languages (e.g., C) to logical constraint representations [34].

21.9 Termination Proofs

Although executions are assumed to be bounded, it is even better to prove it. This can be another task of the abstract interpreter. Because of the boundedness hypothesis, Knuth's method is sound and complete. It consist in adding a counter to loops (and recursion), initialized to zero, incremented at each iteration (recursive call), and proved to be bounded; see for example [12, section 2.2].

22 Future Work

22.1 Gadgets and Widgets

Gadgets and widgets replace a code statement by a more efficient check of this computation of the code specification rather than actually executing the statement. So it is not the mere replacement of a computation by another one, as we have considered in this work. We think that the present framework can be extended replace to a computation by a check of their result and to automate the insertion of gadgets and widgets, guided by an analysis, at least the simplest ones.

22.2 Replacement of Floats by Fixed Point Arithmetic

It is tempting to control accuracy to transform floating point computations into fixed point ones. The result will in general be different and static analysis might be useful to automatize the transformation. Static analysis of fixed point computations is a subject still undeveloped [30, 29].

22.3 Piecewise Partial Unrolling

Once unrolled, it might be that the program is much too large to be accepted by compilers. Usually, a compiler transforms programs into an intermediate representation, which is analyzed and transformed into the output code. If the output code/Boolean and arithmetic circuit can be streamed out, this is generally not the case of the input. In that case, the compiler will fail in combinatorial explosion because the unrolled input is much too large.

The solution that we envision is piecewise unrolling into a sequence of loop iterations where successive iterations correspond to different code optimizations. Consider for example the unrolling 1000 times of `for (i = 1; i<n; i=i+1) S` where an analysis has determined that `n` $\geqslant 500$ and `S` has no break and does not modify `i`. Then a partial enrolling could be

```
for (i = 1; i<500; i=i+1) S
for (i = 500; i<min(n,1000); i=i+1) S
if (n>1000) error;
```

Trace partitioning can be used to decide on such decompositions [31].

Acknowledgements

This material is based upon work supported by DARPA under Agreement No. HR00112020022. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government or DARPA.

References

[1] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. "The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems". In: *Sci. Comput. Program.* 72.1–2 (2008), pp. 3–21.

[2] Mikaël Barboteu, Nacera Djehaf, and Matthieu Martel. “Numerically accurate code synthesis for Gauss pivoting method to solve linear systems coming from mechanics”. In: *Comput. Math. Appl.* 77.11 (2019), pp. 2883–2893.

[3] Anna Becchi and Enea Zaffanella. “An Efficient Abstract Domain for Not Necessarily Closed Polyhedra”. In: *SAS*. Vol. 11002. Lecture Notes in Computer Science. Springer, 2018, pp. 146–165.

[4] Anna Becchi and Enea Zaffanella. “Revisiting Polyhedral Analysis for Hybrid Systems”. In: *SAS*. Vol. 11822. Lecture Notes in Computer Science. Springer, 2019, pp. 183–202.

[5] Dorra Ben Khalifa, Matthieu Martel, and Assalé Adjé. “POP: A Tuning Assistant for Mixed-Precision Floating-Point Computations”. In: *FTSCS*. Vol. 1165. Communications in Computer and Information Science. Springer, 2019, pp. 77–94.

[6] Marc Chevalier. “Proving the Security of Software–Intensive Embedded Systems by Abstract Interpretation. (Analyse de la sécurité de systèmes critiques embarqués à forte composante logicielle par interprétation abstraite)”. PhD thesis. Université PSL – Paris, Nov. 2020.

[7] Maria Christakis and Valentin Wüstholz. “Bounded Abstract Interpretation”. In: *SAS*. Vol. 9837. Lecture Notes in Computer Science. Springer, 2016, pp. 105–125.

[8] Nathanaël Courant and Caterina Urban. “Precise Widening Operators for Proving Termination by Abstract Interpretation”. In: *TACAS (1)*. Vol. 10205. Lecture Notes in Computer Science. 2017, pp. 136–152.

[9] Patrick Cousot. “Méthodes itératives de construction et d’approximation de points fixes d’opérateurs monotones sur un treillis, analyse sémantique de programmes”. Thèse d’État ès Sciences Mathématiques. Université de Grenoble Alpes, Mar. 21, 1978.

[10] Patrick Cousot. “On Fixpoint/Iteration/Variant Induction Principles for Proving Total Correctness of Programs with Denotational Semantics”. In: *LOPSTR*. Vol. 12042. Lecture Notes in Computer Science. Springer, 2019, pp. 3–18.

[11] Patrick Cousot. *Principles of Abstract Interpretation*. MIT Press, 2021.

[12] Patrick Cousot. “Proving Program Invariance and Termination by Parametric Abstraction, Lagrangian Relaxation and Semidefinite Programming”. In: *VMCAI*. Vol. 3385. Lecture Notes in Computer Science. Springer, 2005, pp. 1–24.

[13] Patrick Cousot and Radhia Cousot. “An abstract interpretation framework for termination”. In: *POPL*. ACM, 2012, pp. 245–258.

[14] Patrick Cousot and Radhia Cousot. “Static Determination of Dynamic Properties of Programs”. In: *Proceedings of the Second International Symposium on Programming*. Dunod, 1976, pp. 106–130.

[15] Patrick Cousot and Radhia Cousot. “Systematic design of program transformation frameworks by abstract interpretation”. In: *POPL*. ACM, 2002, pp. 178–190.

[16] Patrick Cousot, Roberto Giacobazzi, and Francesco Ranzato. “Program Analysis Is Harder Than Verification: A Computability Perspective”. In: *CAV (2)*. Vol. 10982. Lecture Notes in Computer Science. Springer, 2018, pp. 75–95.

[17] Patrick Cousot and Nicolas Halbwachs. “Automatic Discovery of Linear Restraints Among Variables of a Program”. In: *POPL*. ACM Press, 1978, pp. 84–96.

[18] Patrick Cousot et al. “Combination of Abstractions in the ASTRÉE Static Analyzer”. In: *ASIAN*. Vol. 4435. Lecture Notes in Computer Science. Springer, 2006, pp. 272–300.

[19] Patrick Cousot et al. “Why does Astrée scale up?” In: *Formal Methods Syst. Des.* 35.3 (2009), pp. 229–264.

[20] Nasrine Damouche, Matthieu Martel, and Alexandre Chapoutot. “Numerical program optimisation by automatic improvement of the accuracy of computations”. In: *Int. J. Intell. Eng. Informatics* 6.1/2 (2018), pp. 115–145.

[21] David Delmas et al. “Towards an Industrial Use of FLUCTUAT on Safety-Critical Avionics Software”. In: *FMICS*. Vol. 5825. Lecture Notes in Computer Science. Springer, 2009, pp. 53–69.

[22] Eric Goubault. “Static Analysis by Abstract Interpretation of Numerical Programs and Systems, and FLUCTUAT”. In: *SAS*. Vol. 7935. Lecture Notes in Computer Science. Springer, 2013, pp. 1–3.

[23] Éric Goubault, Sylvie Putot, and Franck Védrine. “Modular Static Analysis with Zonotopes”. In: *SAS*. Vol. 7460. Lecture Notes in Computer Science. Springer, 2012, pp. 24–40.

[24] Gérard P. Huet and Hugo Herbelin. “30 Years of Research and Development Around Coq”. In: *POPL*. ACM, 2014, pp. 249–250.

[25] Bertrand Jeannet and Antoine Miné. “Apron: A Library of Numerical Abstract Domains for Static Analysis”. In: *CAV*. Vol. 5643. Lecture Notes in Computer Science. Springer, 2009, pp. 661–667.

[26] Neil D. Jones. “An Introduction to Partial Evaluation”. In: *ACM Comput. Surv.* 28.3 (1996), pp. 480–503.

[27] Jacques–Henri Jourdan et al. “A Formally–Verified C Static Analyzer”. In: *POPL*. ACM, 2015, pp. 247–259.

[28] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language*. 2nd ed. Prentice–Hall, 1988.

[29] Asma Mansouri, Matthieu Martel, and Oana-Silvia Serea. “Fixed Point Computation by Exponentiating Linear Operators”. In: *CoDIT*. IEEE, 2019, pp. 1096–1102.

[30] Matthieu Martel. “RangeLab: A Static-Analyzer to Bound the Accuracy of Finite-Precision Computations”. In: *SYNASC*. IEEE Computer Society, 2011, pp. 118–122.

[31] Laurent Mauborgne and Xavier Rival. “Trace Partitioning in Abstract Interpretation Based Static Analyzers”. In: *ESOP*. Vol. 3444. Lecture Notes in Computer Science. Springer, 2005, pp. 5–20.

[32] Antoine Miné. “The Octagon Abstract Domain”. In: *Higher–Order and Symbolic Computation* 19.1 (2006), pp. 31–100.

[33] David Monniaux. “The pitfalls of verifying floating-point computations”. In: *ACM Trans. Program. Lang. Syst.* 30.3 (2008), 12:1–12:41.

[34] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. “CirC: Infrastructure for constraint compilers”. Preprint. 2020.

[35] Gordon D. Plotkin. “A Structural Approach to Operational Semantics”. In: *J. Log. Algebr. Program.* 1972–01 (2004), pp. 17–139.

[36] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. “Global Value Numbers and Redundant Computations”. In: *POPL*. ACM Press, 1988, pp. 12–27.

[37] Gagandeep Singh, Markus Püschel, and Martin T. Vechev. “A practical construction for decomposing numerical abstract domains”. In: *Proc. ACM Program. Lang.* 2.POPL (2018), 55:1–55:28.

[38] Alfred Tarski. “A Lattice Theoretical Fixpoint Theorem and Its Applications”. In: *Pacific J. of Math.* 5 (1955), pp. 285–310.

[39] Justin Thaler. *Proofs, Arguments, and Zero-Knowledge*. `http://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html`, Dec. 27, 2020.

[40] Caterina Urban and Antoine Miné. "Inference of ranking functions for proving temporal properties by abstract interpretation". In: *Comput. Lang. Syst. Struct.* 47 (2017), pp. 77–103.

[41] Riad S. Wahby et al. "Efficient RAM and control flow in verifiable outsourced computation". In: *NDSS*. The Internet Society, 2015.

[42] Michael Walfish and Andrew J. Blumberg. "Verifying computations without reexecuting them". In: *Commun. ACM* 58.2 (2015), pp. 74–84.