Program Unrolling by Abstract Interpretation for Probabilistic Proofs

By

Daniel Noah Feldan

A thesis submitted in partial fulfillment

Of the requirements for the degree of

Master of Science

Courant Institute of Mathematical Sciences

New York University

September, 2022

[illegible]

Patrick Cousot

Abstract

Zero-knowledge proofs are cryptographic protocols that enable one party to prove the validity of a statement to another party while revealing no additional information beyond the statement's truth. These protocols have a myriad of applications, especially within the realm of cloud computing. Zero-knowledge protocols can be used to probabilistically verify cloud-computed program by first converting an input program into a Boolean circuit, then using this circuit in a zero-knowledge proof system to show the correctness of the computed output. This work focuses on enhancing zero-knowledge proofs for practical implementation, as many of the current protocols are currently very computationally expensive.

The primary contribution of this thesis is a formalization of a program transformation technique that combines abstract interpretation and program unrolling to analyze, transform and optimize a program before transforming it into a Boolean circuit. By analyzing a program with an abstract interpreter while simultaneously unrolling it, we achieve a significantly more precise static analysis without the need for traditional widening and narrowing operations. This approach enables aggressive optimization of the unrolled program, reducing both the cost of transforming the program into a circuit, and the resulting circuit's size.

Contents

1 Introduction

1.1 Context

With the advent of cloud computing, remote computation has become more and more popular. Service providers (SPs) can now give users the opportunity to rent clusters of machines for the purpose of computing programs normally too large to process with their own devices. This has opened up the possibility of researchers analyzing and solving problems which normally would be prohibitively expensive. However, this new age of remote computation comes with major pitfalls. It is very difficult to trust that computation done by a SP is correct. Because they utilize large amounts of interconnected devices, each of those devices is a liability and can cause some error to occur, either because of configurations, corruption of data, hardware issues, malicious operations, and more [20]. Furthermore, since the computation done through a SP is done in a black box fashion, any issues that do occur are very difficult to trace.

There have been many proposed solutions to this problem – such as repeating the computation, auditing solutions, and ensuring the SP is using trusted hardware – however each proposed solution has major pitfalls. A new area of research attempts to remove these pitfalls through what is called proof-based verifiable computation. In this model, the SP does not only return the results to the requested program, but they also return a proof affirming that everything was done correctly. Furthermore, as long as a user can frame their computation as a mathematical assertion, then the proof can be constructed as a protocol similar to a Probabilistic Proof System from the field of cryptography. In this protocol a prover, in our case the SP, creates a statement that it wants a verifier, the user, to accept, using knowledge that will remain hidden from the verifier. Once this protocol is established, the field of cryptography gives us the tools to verify this proof very efficiently.

This type of Probabilistic Proof System protocol, specifically where the prover wants to be able to keep knowledge hidden from the verifier, is known as a zero-knowledge (ZK) proof. However, currently the process of creating a ZK protocol is not efficient enough to be practically used. Therefore, DARPA's "Securing Information for Encrypted Verification and Evaluation" (SIEVE) wishes to enhance zero-knowledge proofs. In this case, the input 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, which is needed so that the computation can be checked with probabilistic methods. Formally, we want the prover to 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 front-end 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 [20, 17, 17, 19].

1.2 Objective

Our 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 [2]). A transformer can then 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 then discuss the usage and implementation of several different abstract domains to be used for static analysis. Finally we suggest a number of possible classes of optimizing transformations.

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

[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.

1.4 Future Work

Soundness proofs remain to be done (and maybe to be checked with a proof assistant such as Coq [9], as the style of [12]). Furthermore, the implementation created only works with integers. Future work could allow machine integers or floats. The transformations remain to be studied in greater details, proved correct following [6], implemented, and experimented. These experiments 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 Language

2.1 Syntax

We consider a small subset of the C programming language [13], as follows:

$\mathtt{x}, \mathtt{y}, \mathtt{z}[\mathtt{i}] \ldots$	$\in$	V			variables (V not empty)
$\mathtt{A}$	$\in$	A	$::=$	$\mathtt{1} \mid \mathtt{x} \mid \mathtt{x}[\mathtt{A}_1] \mid \mathtt{A}_1\ \mathtt{op}\ \mathtt{A}_2$	arithmetic expressions
		$\mathtt{op}$	$::=$	$\mathtt{+} \mid \mathtt{-} \mid \mathtt{*} \mid \mathtt{/}$	binary operators
$\mathtt{B}$	$\in$	B	$::=$		Boolean expressions
				$\mathtt{A}$[3]	
			$\mid$	$\mathtt{A}_1 \mathtt{<} \mathtt{A}_2$	
			$\mid$	$\mathtt{A}_1 \mathtt{<=} \mathtt{A}_2$	
			$\mid$	$\mathtt{A}_1 \mathtt{>} \mathtt{A}_2$	
			$\mid$	$\mathtt{A}_1 \mathtt{>=} \mathtt{A}_2$	
			$\mid$	$\mathtt{A}_1 \mathtt{==} \mathtt{A}_2$	
			$\mid$	$\mathtt{A}_1 \mathtt{!=} \mathtt{A}_2$	
			$\mid$	$\mathtt{B}_1\ \mathtt{nand}\ \mathtt{B}_2$	
$\mathtt{E}$	$\in$	E	$::=$	$\mathtt{A} \mid \mathtt{B}$	expressions

[3]In the C language, boolean values are represented by integers – 0 representing false and all non-zero integers representing true.

S	::=		statement $\mathtt{S} \in \mathsf{S}$
		int x ;	variable initialization
	|	x = A ;	variable assignment
	|	;	skip
	|	**if** (B) S	conditional
	|	**if** (B) S **else** S	
	|	**while** (B) S	iteration
	|	**while [n]** (B) S	bounded iteration
	|	**break ;**	iteration break
	|	**{** Sl **}**	compound statement
	|	**{|** Sl **|}**	breakable statement
Sl	::=	Sl S | ϵ	statement list $\mathtt{Sl} \in \mathsf{Sl}$, ϵ is the empty string
P	::=	Sl	program $\mathtt{P} \in \mathsf{P}$
Pc	$\triangleq$	$\mathsf{S} \cup \mathsf{Sl} \cup \mathsf{P}$	program component $\mathtt{S} \in \mathsf{Pc}$

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⟧);

$\mathsf{labs}[\![\mathtt{S}]\!]$	the potentially reachable program points while executing `S` either in or after the statement (excluding reachability by a break) and
$\mathsf{labx}[\![\mathtt{S}]\!]$	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 [4, chapter 4]. We change and add some definitions in order to extend the definition to the updated syntax, as well as have the labels better reflect an unrolled program.
All rules ($\mathsf{at}[\![S]\!]$, $\mathsf{after}[\![S]\!]$, ect...) for variable and array initialization are the same as assignments.
Furthermore, to include break statements out of compound statements we use the following rules:

$\mathtt{Sl}$	::=	$\mathtt{Sl}'\ \mathtt{S}$	$\mathsf{break\text{-}to}[\![\mathtt{Sl}']\!] \triangleq \mathsf{break\text{-}to}[\![\mathtt{S}]\!] \triangleq \mathsf{break\text{-}to}[\![\mathtt{Sl}]\!]$
$\mathtt{S}$	::=	$\mathtt{\textbf{if}\ (B)\ S}_t$	$\mathsf{break\text{-}to}[\![\mathtt{S}_t]\!] \triangleq \mathsf{break\text{-}to}[\![\mathtt{S}]\!]$
$\mathtt{S}$	::=	$\mathtt{\textbf{if}\ (B)\ S}_t\ \mathtt{\textbf{else}\ S}_f$	$\mathsf{break\text{-}to}[\![\mathtt{S}_t]\!] \triangleq \mathsf{break\text{-}to}[\![\mathtt{S}_f]\!] \triangleq \mathsf{break\text{-}to}[\![\mathtt{S}]\!]$
$\mathtt{S}$	::=	$\mathtt{\textbf{while}\ (B)\ S}_b$	$\mathsf{break\text{-}to}[\![\mathtt{S}_b]\!] \triangleq \mathsf{after}[\![\mathtt{S}]\!]$
$\mathtt{S}$	::=	$\mathtt{\{\ Sl\ \}}$	$\mathsf{break\text{-}to}[\![\mathtt{Sl}]\!] \triangleq \mathsf{break\text{-}to}[\![\mathtt{S}]\!]$
$\mathtt{S}$	::=	$\mathtt{\{\mid\ Sl\ \mid\}}$	$\mathsf{break\text{-}to}[\![\mathtt{Sl}]\!] \triangleq \mathsf{after}[\![\mathtt{S}]\!]$

Finally we change the labels themselves. **Original Code**

```
([],0) int x;
([],1) int y;
([],2) x = 0;
([],3) y = 0;
([],4) while(x<2)
       {
([],5)    x = x+1;
([],6)    while(y=0)
([],7)       y = 1;
([],8)    y = 0;
       }
```

Unrolled Code

```
([],0)     int x;
([],1)     int y;
([],2)     x = 0;
([],3)     y = 0;
([],4)     if(x<2)
           {
([1],5)       x = x+1;
([1,1],6)     if(y=0)
([1,1],7)        y = 1;
([1],8)       y=0;
           }
([],4)     if(x<2)
           {
([2],5)       x = x+1;
([1,2],6)     if(y=0)
([1,2],7)        y = 1;
([2],8)       y=0;
           }
```

To be more formal, we define a label ℓ as a tuple such that $\ell.l$ is an integer list and $\ell.n$ is an integer. The first time a **`while ()`** statement is unrolled into an **`if ()`** statement, we append $\mathsf{at}[\![$**`if ()`**$]\!].l$ with a 1. Then every future time we unroll the same **`while ()`** statement, we increment the head of the list by 1. We can see from the example code above that since we have a nested loop, there are labels that have 2 elements, the head corresponding to the inner loop and the tail corresponding to the outer loop. Furthermore, at the second iteration of the unrolling, the element corresponding to the outer loop becomes 2.
Regarding $\ell.n$, we provide the following definition:

`Sl`	::=	$^{\ell_1}\mathtt{Sl}'\ ^{\ell_2}\ \mathtt{S}^{\ell_3}$	$\ell_2.l = \ell_3.l;\ \ell_3.n = \ell_2.n + 1$
`S`	::=	$^{\ell_1}\mathbf{if}\ \mathtt{(B)}\ ^{\ell_2}\mathtt{S}_t$	$\ell_1.l = \ell_2.l;\ \ell_2.n = \ell_1.n + 1$
`S`	::=	$^{\ell_1}\mathbf{if}\ \mathtt{(B)}\ ^{\ell_2}\mathtt{S}_t\ \mathbf{else}\ ^{\ell_3}\mathtt{S}_f$	$\ell_1.l = \ell_2.;= \ell_3.l;\ \ell_2.n = \ell_1.n + 1;\ \ell_3.n = \mathsf{after}[\![S_t]\!].n + 1$
`S`	::=	$^{\ell_1}\mathbf{while}\ \mathtt{(B)}\ ^{\ell_2}\mathtt{S}_t$	$\ell_1.l = \ell_2.l;\ \ell_2.n = \ell_1.n + 1$

For all other statements not specified, labels work as defined in [4, chapter 4]

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 Values

The C language has a concrete semantics that contains concrete values which it uses when executing a program. However, in order to statically analyze a program, we instead use an abstract semantics with abstract values. For the purposes of this paper we will not be discussing the concrete semantics of C, and will only discuss the abstract semantics.

3.1 Variables

We let V to be the set of variables (of the language, program, or a parameter of the semantics; see [4]). Furthermore, we delineate two types of variables for this project. Simple variables and arrays. Simple variables would be handled like the variables in [4]. Arrays on the other hand, are more complicated and therefore which will be discussed in further detail below.

3.2 Values

We let $\nu \in \mathbb{V}$ to be the set of abstract values. These abstract values can include integers, a symbolic value, top (unknown, denoted by $\top$), or bot (unreachable, denoted by $\bot$).

3.3 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 abstract values $\rho(\mathtt{x}) \in \mathbb{V}$. The assignment of a value ν to a variable $\mathtt{x}$ in environment ρ is

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

3.4 Evaluation

We denote $\mathcal{A}[\![\mathtt{A}]\!]\rho$ as the abstract value of arithmetic expression $\mathtt{A}$ in environment ρ and $\mathcal{B}[\![\mathtt{B}]\!]\rho$ the abstract Boolean value of Boolean expression $\mathtt{B}$ in environment ρ; see [4, chapter 3]. For the purpose of this project, $\mathcal{A}[\![\mathtt{A}]\!]\rho$ can be one of three values: $\top$ (unknown), an integer (exactly known), $\bot$ (Not able to be evaluated).
To evaluate Boolean expressions, we use ternary logic. Due to the limitations of static verification it is impossible to always precisely evaluate Boolean expressions. However, due to abstractions, we may be able to see if a Boolean expression will always be true (T), sometimes be true (TF) or always be false (F). This information will be very important when exploring different optimizations that can be done to a program.

3.5 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.

3.6 Actions

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

3.7 Initialization

Initially in an environment ρ, $\forall \mathtt{x} \in \mathsf{V}, \rho(\mathtt{x}) = \bot$. Once a variable is initialized using an **int** x ; statement, that variable is assigned an initial value. We use three different possible initial values, which we use as parameters that can change:

1. A numeric initial values. Usually this is set to zero. This is equivalent to the statement **int** x = C ; for some integer constant C.

2. A symbolic value. We denote this initial value with "_0". For example given a variable x, it would be assigned an initial value x_0 (We reserve '_' as a special character in our syntax for this purpose). These variables cannot be assigned to any value, and are just used to delineate a symbolic initial value for variables. This is a relational value, as the values of variables are now functions of initial symbolic values.

3. $\top$. This means that the variables can have any initial value, but the exact value is unknown. It also implies that if two different variables initialized to $\top$, they may not have the same initial concrete value. Furthermore, if a single variable is initialized multiple times, it could theoretically have a different initial concrete value at each initialization.

For simplicity, we will assume in this paper that all variables are assumed to be initialized to a symbolic value unless explicitly stated otherwise.

3.8 Arrays

Arrays are more difficult to handle than simple variables, because we allows arrays to be indexed through any valid arithmetic expression. This means, that given an array x[i], where $\mathcal{A}[\![i]\!]\rho \triangleq \top$, we do not know which index of the array we are referring to. This can cause many issues, especially if we attempt to assign a value to an unknown index to an array. To circumvent this issue, we employ array expansion, or the assignment of a simple variable to each array index. This tactic is commonly used to handle arrays during unrolling.
Practically we implement array expansion by doing the following: Initially an environment ρ does not contain any reference to any array variables. When an array a[i] is initialized in an environment ρ, we let $n \triangleq \mathcal{A}[\![i]\!]\rho$. If n is an integer, we add n+1 variables to the V such that $\{a., a.0, a.1, \ldots a.(n-1)\} \in \mathsf{V}$. Just like "_", we reserve "." as a special character in our syntax. The first variable a. is a sentinel variable that stores the length of the array in our environment. Each of the other variables

$a.0, a.1, \dots a.(n-1)$ correspond to a single index in the array. When n is $\top$, we only add the single sentinel variable to the V.
Once an array has been initialized, given an expression $A = \mathtt{a[i]}$ in environment ρ, and $n \triangleq \mathcal{A}[\![i]\!]\rho$:

$$\begin{aligned}
\mathcal{A}[\![A]\!] \triangleq \bot & \quad \text{when } n \triangleq \bot \text{ or } \mathsf{test}[\![n < 0]\!] \text{ or } \mathsf{test}[\![n \geq \rho(a.)]\!] \\
\mathcal{A}[\![A]\!] \triangleq \top & \quad \text{when } n \triangleq \top \text{ or } \rho(a.) = \top \\
\mathcal{A}[\![A]\!] \triangleq \rho(a.n) & \quad \text{otherwise}
\end{aligned}$$

We note that when the upper bound of the array is $\top$, then the value of any reference to that array will be either $\top$ or $\bot$.
As stated above, a big issue is when there is an array assignment to an index that evaluates to $\top$. In this case, we simply change the value of all the elements of the array to $\top$, as now the contents of the array cannot be precisely known.

4 Abstract Domain

An abstract domain [4, 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:

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

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

$$\begin{aligned}
\mathsf{program} &\in \mathsf{P} \to \mathbb{P} \to \mathbb{P}\ ^{4} \\
\mathsf{stmtlist} &\in \mathsf{Sl} \to \mathbb{P} \times \mathbb{P} \to \mathbb{P} \\
\mathsf{empty} &\in \mathsf{Sl} \to \mathbb{P} \\
\mathsf{init}, \mathsf{assign}, \mathsf{skip}, \mathsf{break} &\in \mathsf{S} \to \mathbb{P} \\
\mathsf{if}, \mathsf{iter}, \mathsf{compound}, \mathsf{breakable} &\in \mathsf{S} \to \mathbb{P} \to \mathbb{P} \\
\mathsf{ife} &\in \mathsf{S} \to \mathbb{P} \times \mathbb{P} \to \mathbb{P}
\end{aligned}$$

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

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

5 Abstract Interpreter

The abstract interpreter $\mathcal{S}(\!|\mathbb{D}|\!) \in \mathsf{Pc} \nrightarrow \mathbb{P}$ 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* $\mathtt{P} ::= \mathtt{Sl}$
$$\mathcal{S}[\![\mathtt{P}]\!] \triangleq \mathsf{program}[\![\mathtt{P}]\!](\mathcal{S}[\![\mathtt{Sl}]\!]) \tag{3}$$
- *Abstract semantics of a statement list* $\mathtt{Sl} ::= \mathtt{Sl}'\ \mathtt{S}$
$$\mathcal{S}[\![\mathtt{Sl}]\!] \triangleq \mathsf{stmtlist}[\![\mathtt{Sl}]\!](\mathcal{S}[\![\mathtt{Sl}']\!], \mathcal{S}[\![\mathtt{S}]\!]) \tag{4}$$
- *Abstract semantics of an empty statement list* $\mathtt{Sl} ::= \epsilon$
$$\mathcal{S}[\![\mathtt{Sl}]\!] \triangleq \mathsf{empty}[\![\mathtt{Sl}]\!] \tag{5}$$
- *Abstract semantics of an initialization statement* $\mathtt{S} ::= \mathbf{int}\ \mathtt{x}\ \mathbf{;}$
$$\mathcal{S}[\![\mathtt{S}]\!] \triangleq \mathsf{init}[\![\mathtt{S}]\!] \tag{6}$$
- *Abstract semantics of an assignment statement* $\mathtt{S} ::= \mathtt{x} = \mathtt{A}\ \mathbf{;}$
$$\mathcal{S}[\![\mathtt{S}]\!] \triangleq \mathsf{assign}[\![\mathtt{S}]\!] \tag{7}$$
- *Abstract semantics of a skip statement* $\mathtt{S} ::= \mathbf{;}$
$$\mathcal{S}[\![\mathtt{S}]\!] \triangleq \mathsf{skip}[\![\mathtt{S}]\!] \tag{8}$$
- *Abstract semantics of a conditional statement* $\mathtt{S} ::= \mathbf{if}\ (\mathtt{B})\ \mathtt{S}_t$
$$\mathcal{S}[\![\mathtt{S}]\!] \triangleq \mathsf{if}[\![\mathtt{S}]\!](\mathcal{S}[\![\mathtt{S}_t]\!]) \tag{9}$$
- *Abstract semantics of a conditional statement* $\mathtt{S} ::= \mathbf{if}\ (\mathtt{B})\ \mathtt{S}_t\ \mathbf{else}\ \mathtt{S}_f$
$$\mathcal{S}[\![\mathtt{S}]\!] \triangleq \mathsf{ife}[\![\mathtt{S}]\!](\mathcal{S}[\![\mathtt{S}_t]\!], \mathcal{S}[\![\mathtt{S}_f]\!]) \tag{10}$$
- *Abstract semantics of an iteration statement* $\mathtt{S} ::= \mathbf{while}\ (\mathtt{B})\ \mathtt{S}_b$
$$\mathcal{S}[\![\mathtt{S}]\!] \triangleq \mathsf{iter}[\![\mathtt{S}]\!](\mathcal{S}[\![\mathtt{S}_b]\!]) \tag{11}$$

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

$$\mathcal{S}[\![\mathtt{S}]\!] \;\triangleq\; \mathsf{break}[\![\mathtt{S}]\!] \tag{12}$$

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

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

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

$$\mathcal{S}[\![\mathtt{S}]\!] \;\triangleq\; \mathsf{breakable}[\![\mathtt{S}]\!](\mathcal{S}[\![\mathtt{Sl}]\!]) \tag{14}$$

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.

6 Prefix Abstract Interpreter

Given a prelude R, that is a precondition when arriving at a program component S or just $\mathsf{at}[\![\mathtt{S}]\!]$, the prefix abstract semantics $\mathcal{S}^p[\![\mathtt{S}]\!]\,R$ of this program component S returns a continuation, specifying at each program point ℓ of 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 [4, chapter 42]

6.1 Prefix Abstract Domain

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

$$\mathbb{D}^p \;\triangleq\; \langle \mathbb{P}^p,\ \sqsubseteq,\ \bot,\ \sqcup,\ \mathsf{init}^p,\ \mathsf{assign}^p,\ \mathsf{skip}^p,\ \mathsf{test}^p,\ \overline{\mathsf{test}}^p,\ \mathsf{break}^p\rangle \tag{15}$$

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) $\sqcup$.

$$\begin{array}{rcl}
\mathsf{empty}^p, \mathsf{skip}^p, \mathsf{break}^p & \in & \mathsf{Pc} \to (\mathsf{L}\times\mathsf{L}) \to \mathbb{P}^p \overset{\nearrow}{\to} \mathbb{P}^p\ {}^{5}\\
\mathsf{init}^p & \in & \mathsf{Pc} \to (\mathsf{L}\times\mathsf{V}\times\mathsf{L}) \to \mathbb{P}^p \overset{\nearrow}{\to} \mathbb{P}^p\\
\mathsf{assign}^p & \in & \mathsf{Pc} \to (\mathsf{L}\times\mathsf{V}\times\mathsf{A}\times\mathsf{L}) \to \mathbb{P}^p \overset{\nearrow}{\to} \mathbb{P}^p\\
\mathsf{test}^p, \overline{\mathsf{test}}^p & \in & \mathsf{Pc} \to (\mathsf{L}\times\mathsf{B}\times\mathsf{L}) \to \mathbb{P}^p \overset{\nearrow}{\to} \mathbb{P}^p
\end{array}$$

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

the lub $\sqcup$ is well defined both for pairs of elements of $\mathbb{P}^p$ and for $\sqsubseteq$-increasing chains. (16)

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.

6.2 Prefix Abstract Functor

The prefix abstract functor $\mathcal{D}^p$, maps a prefix abstract domain of type (15) into an abstract domain

$$\mathcal{D}^p(\!|\mathbb{D}^p|\!) \triangleq \langle \mathbb{P}, \mathsf{program}, \mathsf{stmtlist}, \mathsf{empty}, \mathsf{init}, \mathsf{assign}, \mathsf{skip}, \mathsf{if}, \mathsf{ife}, \mathsf{iter}, \mathsf{break}, \mathsf{compound}, \mathsf{breakable} \rangle \tag{17}$$

of type (2), defined, assuming (16), as follows:

- *Abstract prefix semantic domain* $\mathbb{P} \triangleq \mathbb{P}^p \overset{\nearrow}{\rightarrow} \mathsf{L} \rightarrow \mathbb{P}^p$ ($\langle \mathbb{P}, \ddot{\sqsubseteq} \rangle$ satisfies (2))
- *Abstract prefix semantics of a program* ℓ_0 `P` ℓ_1 ::= ℓ_0 `Sl` ℓ_1 (where $\ell_0 = \mathsf{at}[\![\mathtt{Sl}]\!] = \mathsf{at}[\![\mathtt{P}]\!]$ and $\ell_1 = \mathsf{after}[\![\mathtt{Sl}]\!] = \mathsf{after}[\![\mathtt{P}]\!]$))

$$\mathsf{program}[\![\mathtt{P}]\!]\ Sl\ R\ \ell \triangleq Sl(R)\ell \tag{18}$$

- *Abstract prefix semantics of a statement list* ℓ_0 `Sl` ℓ_2 ::= ℓ_0 `Sl′` ℓ_1 `S` ℓ_2 (where $\ell_0 = \mathsf{at}[\![\mathtt{Sl}]\!] = \mathsf{at}[\![\mathtt{Sl'}]\!]$, $\ell_1 = \mathsf{at}[\![\mathtt{S}]\!] = \mathsf{after}[\![\mathtt{Sl'}]\!]$, and $\ell_2 = \mathsf{after}[\![\mathtt{S}]\!] = \mathsf{after}[\![\mathtt{Sl}]\!]$) [6]

$$\begin{aligned}\mathsf{stmtlist}[\![\mathtt{Sl}]\!](Sl', S)\ R\ \ell \triangleq\ & (\ \ell \in \mathsf{labs}[\![\mathtt{Sl'}]\!] \setminus \{\mathsf{at}[\![\mathtt{S}]\!]\}\ \mathbf{?}\ Sl'\ R\ \ell \\ & \mid \ell \in \mathsf{labs}[\![\mathtt{S}]\!]\ \mathbf{?}\ S(Sl'\ R\ \mathsf{at}[\![\mathtt{S}]\!])\ \ell \\ & \mathbf{:}\ \bot\ )\end{aligned} \tag{19}$$

- *Abstract prefix semantics of an empty statement list* ℓ_0`Sl`ℓ_0 ::= $\ell_0\ \epsilon\ \ell_0$ (where $\ell_0 = \mathsf{at}[\![\mathtt{Sl}]\!] = \mathsf{after}[\![\mathtt{Sl}]\!]$)

$$\mathsf{empty}[\![\mathtt{Sl}]\!]\ R\ \ell \triangleq (\ \ell = \mathsf{at}[\![\mathtt{Sl}]\!]\ \mathbf{?}\ \mathsf{empty}^p[\![\ell_0, \ell_0]\!]R\ \mathbf{:}\ \bot\ ) \tag{20}$$

[5] $A \overset{\nearrow}{\rightarrow} B$ defines the increasing/isotone maps when A and B are posets.
[6] $(\ldots\ \mathbf{?}\ \ldots \mid \ldots\ \mathbf{?}\ \ldots\ \mathbf{:}\ \ldots)$ is the conditional expression (as in C)

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

$$\mathsf{init}[\![\mathtt{S}]\!]\, R\, \ell \triangleq \begin{array}{l}(\ \ell = \ell_0\ \mathbf{?}\ R \\ \mid \ell = \ell_1\ \mathbf{?}\ \mathsf{init}^p[\![\ell_0, \mathtt{x}, \ell_1]\!]R \\ \mathbf{:}\ \bot\ )\end{array} \tag{21}$$

- *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 $\ell_1 = \mathsf{after}[\![\mathtt{S}]\!]$)

$$\mathsf{assign}[\![\mathtt{S}]\!]\, R\, \ell \triangleq \begin{array}{l}(\ \ell = \ell_0\ \mathbf{?}\ R \\ \mid \ell = \ell_1\ \mathbf{?}\ \mathsf{assign}^p[\![\ell_0, \mathtt{x}, \mathtt{A}, \ell_1]\!]R \\ \mathbf{:}\ \bot\ )\end{array} \tag{22}$$

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

$$\mathsf{skip}[\![\mathtt{S}]\!]\, R\, \ell \triangleq (\ \ell = \ell_0\ \mathbf{?}\ R \mid \ell = \ell_1\ \mathbf{?}\ \mathsf{skip}^p[\![\ell_0, \ell_1]\!]R\ \mathbf{:}\ \bot\ ) \tag{23}$$

- *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 $\ell_2 = \mathsf{after}[\![\mathtt{S}]\!] = \mathsf{after}[\![\mathtt{S}_t]\!]$)

$$\mathsf{if}[\![\mathtt{S}]\!](S)\, R\, \ell \triangleq \begin{array}{l}(\ \ell = \ell_0\ \mathbf{?}\ R \\ \mid \ell \in \mathsf{in}[\![\mathtt{S}_t]\!]\ \mathbf{?}\ S\,(\mathsf{test}^p[\![\ell_0, \mathtt{B}, \ell_t]\!]\ R)\ \ell \\ \mid \ell = \ell_2\ \mathbf{?}\ S\,(\mathsf{test}^p[\![\ell_0, \mathtt{B}, \ell_t]\!]\ R)\ \ell \sqcup \overline{\mathsf{test}}^p[\![\ell_0, \mathtt{B}, \ell_2]\!]\ R \\ \mathbf{:}\ \bot\ )\end{array} \tag{24}$$

- *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 $\ell_2 = \mathsf{after}[\![\mathtt{S}]\!] = \mathsf{after}[\![\mathtt{S}_t]\!] = \mathsf{after}[\![\mathtt{S}_f]\!]$)

$$\mathsf{ife}[\![\mathtt{S}]\!](St, Sf)\, R\, \ell \triangleq \begin{array}{l}(\ \ell = \ell_0\ \mathbf{?}\ R \\ \mid \ell \in \mathsf{in}[\![\mathtt{S}_t]\!]\ \mathbf{?}\ St\ (\mathsf{test}^p[\![\ell_0, \mathtt{B}, \ell_t]\!]\ R)\ \ell \\ \mid \ell \in \mathsf{in}[\![\mathtt{S}_f]\!]\ \mathbf{?}\ Sf\ (\overline{\mathsf{test}}^p[\![\ell_0, \mathtt{B}, \ell_f]\!]\ R)\ \ell \\ \mid \ell = \ell_2\ \mathbf{?}\ St\ (\mathsf{test}^p[\![\ell_0, \mathtt{B}, \ell_t]\!]\ R)\ \ell \sqcup Sf\ (\overline{\mathsf{test}}^p[\![\ell_0, \mathtt{B}, \ell_f]\!]\ R)\ \ell \\ \mathbf{:}\ \bot\ )\end{array} \tag{25}$$

- *Abstract prefix semantics of an iteration statement* $\ell_0\mathtt{S}\ell_2$ ::= **`while`** ℓ_0 `(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 $\ell_2 = \mathsf{after}[\![\mathtt{S}]\!]$) [7]

$$\mathsf{iter}[\![\mathtt{S}]\!](\mathit{Sb})\ R\ \ell \ \triangleq\ \mathsf{lfp}^{\dot\sqsubseteq}\ (\mathcal{F}^p[\![\mathit{Sb}]\!]\ R)\ \ell \tag{26}$$

$$\mathcal{F}^p[\![\mathit{Sb}]\!] \ \in\ \mathbb{P}^p \to ((\mathsf{L} \to \mathbb{P}^p) \to (\mathsf{L} \to \mathbb{P}^p))$$

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

- *Abstract prefix semantics of a break statement* $\mathtt{S}$ ::= ℓ_0 **`break ;`** (where $\ell_0 = \mathsf{at}[\![\mathtt{S}]\!]$)

$$\mathsf{break}[\![\mathtt{S}]\!]\ R\ \ell \ \triangleq\ (\ \ell = \ell_0\ ?\ \mathsf{break}^p[\![\ell_0, \mathsf{break\text{-}to}[\![\mathtt{S}]\!]]\!]R\ :\ \bot\,) \tag{27}$$

- *Abstract prefix semantics of a compound statement* $\mathtt{S}$ ::= ℓ_0**`{`** `Sl` **`}`**ℓ_1 (where $\ell_0 = \mathsf{at}[\![\mathtt{Sl}]\!] = \mathsf{at}[\![\mathtt{S}]\!]$ and $\ell_0 = \mathsf{after}[\![\mathtt{S}]\!] = \mathsf{after}[\![\mathtt{Sl}]\!]$)

$$\mathsf{compound}[\![\mathtt{S}]\!](\mathit{Sl})\ R\ \ell \ \triangleq\ \mathit{Sl}(R)^{\ell} \tag{28}$$

- *Abstract prefix semantics of a breakable statement* $\mathtt{S}$ ::= ℓ_0**`{|`** `Sl` **`|}`**ℓ_1 (where $\ell_0 = \mathsf{at}[\![\mathtt{Sl}]\!] = \mathsf{at}[\![\mathtt{S}]\!]$ and $\ell_1 = \mathsf{after}[\![\mathtt{S}]\!] = \mathsf{after}[\![\mathtt{Sl}]\!]$)

$$\begin{aligned}
\mathsf{breakable}[\![\mathtt{S}]\!](\mathit{Sl})\ R\ \ell \ \triangleq\ & (\ \ell \in \mathsf{in}[\![\mathtt{S}]\!]\ ?\ \mathit{Sl}(R)^{\ell} \\
& \ \mid \ell = \mathsf{after}[\![\mathtt{S}]\!]\ ?\ \mathit{Sl}(R)^{\ell} \sqcup \bigsqcup_{\ell' \in \mathsf{breaks\text{-}of}[\![\mathtt{S}_b]\!]} \mathit{Sl}(R)^{\ell'} \\
& \ :\ \bot\,)
\end{aligned} \tag{29}$$

where $\bigsqcup \emptyset = \bot$.

[7] $\mathsf{lfp}^{\sqsubseteq} f$ denotes the $\sqsubseteq$-least fixpoint of an operator f on a poset $\langle \mathbb{P},\ \sqsubseteq \rangle$, if any; for example [16].

6.3 Prefix Abstract Interpreter

The prefix abstract interpreter $\mathcal{S}^p(\!|\mathbb{D}^p|\!) \in \mathsf{Pc} \to \mathbb{P}^p \overset{\nearrow}{\to} \mathsf{L} \to \mathbb{P}^p$ 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 $\mathbb{D}^p$(15). It is the instance of the abstract interpreter $\mathcal{S}$ for the abstract domain $\mathcal{D}^p(\!|\mathbb{D}^p|\!)$:

$$\mathcal{S}^p(\!|\mathbb{D}^p|\!) \quad \triangleq \quad \mathcal{S}(\!|\mathcal{D}^p(\!|\mathbb{D}^p|\!)|\!) \tag{30}$$

It follows from this definition that

$$\forall \mathtt{S} \in \mathsf{Pc}\ .\ \forall \ell \in \mathsf{L}\ .\ \ell \notin \mathsf{labs}[\![\mathtt{S}]\!] \quad \Rightarrow \quad \mathcal{S}^p[\![\mathtt{S}]\!]\ R\ \ell \triangleq \bot \tag{31}$$

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 (15), as a parameter. This is what we do in the next section, to define the prefix trace semantics.

7 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 (30)).

Given a prelude R, that is execution traces arriving at a program component $\mathtt{S}$ or just $\mathsf{at}[\![\mathtt{S}]\!]$, 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.

7.1 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 $n = |\pi| \geqslant 0$.

7.2 Prefix Trace Abstract Domain

The prefix trace abstract domain is

$$\mathbb{D}^{pt} \triangleq \langle \mathbb{P}^{pt},\ \dot{\subseteq},\ \dot{\emptyset},\ \dot{\cup},\ \mathsf{init}^{pt},\ \mathsf{assign}^{pt},\ \mathsf{skip}^{pt},\ \mathsf{test}^{pt},\ \overline{\mathsf{test}}^{pt},\ \mathsf{break}^{pt}\rangle \tag{32}$$

of type (15), such that

$$\begin{aligned}
\mathbb{P}^{pt} &\triangleq \wp(\mathbb{T}) \stackrel{\nearrow}{\to} \wp(\mathbb{T})\\
\mathsf{init}^{pt}[\![\ell_0, \mathtt{x}, \ell_1]\!]R &\triangleq \{\pi \xrightarrow{\mathtt{a}} \langle \ell_0',\ \rho\rangle \xrightarrow{\ell_0', \mathtt{x}, \ell_1} \langle \ell_1,\ \rho[\mathtt{x} \leftarrow \mathtt{x\_0}]\rangle \mid\\
&\qquad \pi \xrightarrow{\mathtt{a}} \langle \ell_0',\ \rho\rangle \in R \wedge \ell_0' = \ell_0\}\\
\mathsf{assign}^{pt}[\![\ell_0, \mathtt{x}, \mathtt{A}, \ell_1]\!]R &\triangleq \{\pi \xrightarrow{\mathtt{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{\mathtt{a}} \langle \ell_0',\ \rho\rangle \in R \wedge \ell_0' = \ell_0\}\\
\mathsf{skip}^{pt}[\![\ell_0, \ell_1]\!]R &\triangleq \{\pi \xrightarrow{\mathtt{a}} \langle \ell_0',\ \rho\rangle \xrightarrow{\ell_0', \mathtt{;}, \ell_t} \langle \ell_t,\ \rho\rangle \mid \pi \xrightarrow{\mathtt{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{\mathtt{a}} \langle \ell_0',\ \rho\rangle \xrightarrow{\ell_0', \mathtt{B}, \ell_t} \langle \ell_t,\ \rho\rangle \mid\\
&\qquad \pi \xrightarrow{\mathtt{a}} \langle \ell_0',\ \rho\rangle \in R \wedge \mathcal{B}[\![\mathtt{B}]\!]\rho \wedge \ell_0' = \ell_0\}\\
\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{aligned}$$

Notice that $\langle \mathbb{P}^{pt},\ \dot{\subseteq}\rangle$ is a complete lattice hence both a join-lattice and a CPO.

7.3 Prefix Trace Semantics

The finite prefix trace semantics $\mathcal{S}^{pt} \in \mathsf{Pc} \to \wp(\mathbb{T}) \stackrel{\nearrow}{\to} \mathsf{L} \to \wp(\mathbb{T})$ (as defined in [4, chapter 42]) is the instance of the prefix abstract interpreter $\mathcal{S}^{p}$ for the abstract domain

$$\mathcal{S}^{pt} \triangleq \mathcal{S}^{p}(\!|\mathbb{D}^{pt}|\!) = \mathcal{S}(\!|\mathcal{D}^{p}(\!|\mathbb{D}^{pt}|\!)|\!) \tag{33}$$

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.

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 $\mathcal{S}^m \in \mathsf{Pc} \to \mathbb{P}^p \xrightarrow{\nearrow} \mathbb{P}^p \times \mathbb{P}^p$ 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 and 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 $\perp$ 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 (53)).

8.1 Maximal Abstraction

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

$$\alpha^{p\to m}[\![\mathtt{S}]\!]\ \mathcal{S}\ R \ \triangleq\ \langle \mathcal{S}[\![\mathtt{S}]\!]\ R\ \mathsf{after}[\![\mathtt{S}]\!],\ \bigsqcup_{\ell \in \mathsf{breaks\text{-}of}[\![\mathtt{S}]\!]} \mathcal{S}[\![\mathtt{S}]\!]\ R\ \ell\rangle \tag{34}$$

This is a Galois connection ([4, ex.11.21 and 11.20]). When applied to the prefix abstract interpreter $\mathcal{S}^p \in \mathsf{Pc} \to \mathbb{P}^p \xrightarrow{\nearrow} \mathsf{L} \to \mathbb{P}^p$, it yields the maximal abstract interpreter $\mathcal{S}^m \in \mathsf{Pc} \to \mathbb{P}^p \xrightarrow{\nearrow} \mathbb{P}^p$:

$$\mathcal{S}^m[\![\mathtt{S}]\!]\ R \ \triangleq\ \alpha^{p\to m}[\![\mathtt{S}]\!]\ (\mathcal{S}^p[\![\mathtt{S}]\!])\ R \tag{35}$$

Since the prefix abstract interpreter $\mathcal{S}^p$ is parameterized by the abstract domain $\mathbb{D}^p$ (15), 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

$$\mathcal{D}^m(\!|\mathbb{D}^p|\!) \triangleq \langle \mathbb{P}^m,\ \mathsf{program}^m,\ \mathsf{stmtlist}^m,\ \mathsf{empty}^m,\ \mathsf{init}^m,\ \mathsf{assign}^m,\ \mathsf{skip}^m,\ \mathsf{if}^m, \mathsf{ife}^m,\ \mathsf{iter}^m,\ \mathsf{break}^m,\ \mathsf{compound}^m,\ \mathsf{breakable}^m\rangle \tag{36}$$

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

- *Abstract maximal semantic domain* $\mathbb{P}^m \triangleq \mathbb{P}^p \xrightarrow{\nearrow} \mathbb{P}^p$
- *Abstract maximal semantics of a program* $\ell_0\ \mathtt{P}\ \ell_1 ::= \ell_0\ \mathtt{Sl}\ \ell_1$

$$\mathsf{program}^m[\![\mathtt{P}]\!]\ Sl\ R \triangleq \mathsf{let}\ \langle T,\ B\rangle = Sl(R)\ \mathsf{in}\ \langle \mathsf{program}^p[\![\ell_0, \ell_1]\!]\ (T),\ \bot\rangle \tag{37}$$

- *Abstract maximal semantics of a statement list* $\ell_0\ \mathtt{Sl}\ \ell_2 ::= \ell_0\ \mathtt{Sl}'\ \ell_1\ \mathtt{S}\ \ell_2$

$$\begin{aligned}\mathsf{stmtlist}^m[\![\mathtt{Sl}]\!](Sl', S)\ R \triangleq\ & \mathsf{let}\ \langle T',\ B'\rangle = Sl'(R)\ \mathsf{in}\\ & \mathsf{let}\ \langle T,\ B\rangle = S(T')\ \mathsf{in}\\ & \langle T,\ B' \sqcup B\rangle\end{aligned} \tag{38}$$

- *Abstract maximal semantics of an empty statement list* $\ell_0\mathtt{Sl}\ell_0 ::= \ell_0\ \epsilon\ \ell_0$

$$\mathsf{empty}^m[\![\mathtt{Sl}]\!] \triangleq \langle \mathsf{empty}^p[\![\ell_0, \ell_0]\!]R,\ \bot\rangle \tag{39}$$

- *Abstract maximal semantics of an intilzation statement* $\ell_0\mathtt{S}\ell_1 ::= \ell_0\ \mathtt{\textbf{int}\ x\ ;}\ell_1$

$$\mathsf{init}^m[\![\mathtt{S}]\!]\ R \triangleq \langle \mathsf{init}^p[\![\ell_0, \mathtt{x}, \ell_1]\!]R,\ \bot\rangle \tag{40}$$

- *Abstract maximal semantics of an assignment statement* $\ell_0\mathtt{S}\ell_1 ::= \ell_0\mathtt{x\ =\ A\ ;}\ell_1$

$$\mathsf{assign}^m[\![\mathtt{S}]\!]\ R \triangleq \langle \mathsf{assign}^p[\![\ell_0, \mathtt{x}, \mathtt{A}, \ell_1]\!]R,\ \bot\rangle \tag{41}$$

- *Abstract maximal semantics of a skip statement* $\ell_0\mathtt{S}\ell_1 ::= \mathtt{;}$

$$\mathsf{skip}^m[\![\mathtt{S}]\!]\ R \triangleq \langle \mathsf{skip}^p[\![\ell_0, \ell_1]\!]R,\ \bot\rangle \tag{42}$$

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

$$\begin{aligned}\mathsf{if}^m[\![\mathtt{S}]\!](S)\ R \triangleq\ & \mathsf{let}\ \langle T,\ B\rangle = S\,(\mathsf{test}^p[\![\ell_0, \mathtt{B}, \ell_t]\!]\ R)\ \mathsf{in}\\ & \langle T \sqcup \overline{\mathsf{test}}^p[\![\ell_0, \mathtt{B}, \ell_2]\!]\ R,\ B\rangle\end{aligned} \tag{43}$$

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

$$\begin{aligned}\mathsf{ife}^m[\![\mathtt{S}]\!](St, Sf)\ R \ \triangleq\ & \mathsf{let}\ \langle T_t,\ B_t\rangle = St\ (\mathsf{test}^p[\![\ell_0, \mathtt{B}, \ell_t]\!]\ R)\ \mathsf{in}\\ & \mathsf{let}\ \langle T_f,\ B_f\rangle = Sf\ (\overline{\mathsf{test}}^p[\![\ell_0, \mathtt{B}, \ell_f]\!]\ R)\ \mathsf{in}\\ & \langle T_t \sqcup T_f,\ B_t \sqcup B_f\rangle\end{aligned} \tag{44}$$

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

$$\mathsf{iter}^m[\![\mathtt{S}]\!](\mathcal{S}[\![\mathtt{S}_b]\!])\ R \ \triangleq\ \mathsf{let}\ \langle T,\ B\rangle = \mathsf{lfp}^{\sqsubseteq_2}\,(\mathcal{F}^m[\![Sb]\!]\ R)\ \mathsf{in}\ \langle T \sqcup B,\ \bot\rangle \tag{45}$$

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

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

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* `S` ::= ${}^{\ell_0}$ **`break ;`** (where $\ell_0 = \mathsf{at}[\![\mathtt{S}]\!]$)

$$\mathsf{break}^m[\![\mathtt{S}]\!]\ R \ \triangleq\ \langle\bot,\ \mathsf{break}^p[\![\ell_0, \mathsf{break\text{-}to}[\![\mathtt{S}]\!]]\!]R\rangle \tag{46}$$

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

$$\begin{aligned}\mathsf{compound}[\![\mathtt{S}]\!](Sl)R \ \triangleq\ & \mathsf{let}\ \langle T,\ B\rangle = Sl(R)\ \mathsf{in}\\ & \langle\mathsf{compound}^p[\![\ell_0, \ell_1]\!](T),\ B\rangle\end{aligned} \tag{47}$$

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

$$\begin{aligned}\mathsf{breakable}^m[\![\mathtt{S}]\!](Sl)R \ \triangleq\ & \mathsf{let}\ \langle T,\ B\rangle = Sl(R)\ \mathsf{in}\\ & \langle\mathsf{breakable}^p[\![\ell_0, \ell_1]\!](T \sqcup B),\ \bot\rangle\end{aligned} \tag{48}$$

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 $\mathcal{D}^m(\!|\mathbb{D}^p|\!)$:

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

Using both structural and fixpoint induction, $\mathcal{S}^m$ yields the abstract version of Hoare logic of [4, 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 (34) and (a proof by calculational design similar to) [4, 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 $\mathbb{D}^{pt}$ (32), the operations of which have been defined in section 7.3.

$$\mathcal{S}^{mt}(\!|\mathbb{D}^{pt}|\!) \;\triangleq\; \mathcal{S}^m(\!|\mathbb{D}^{pt}|\!) \;=\; \mathcal{S}(\!|\mathcal{D}^m(\!|\mathbb{D}^{pt}|\!)|\!) \tag{49}$$

10 Relational Semantics

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

10.1 Properties

Following [4, 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

$$\begin{aligned}\alpha^{mt\to r}(\langle T,\, B\rangle)R \;\triangleq\; \{\langle \rho,\, \rho_n\rangle \mid \langle \rho,\, \rho_0\rangle \in R \wedge \exists n \in \mathbb{N} \;.\; \langle \ell_0,\, \rho_0\rangle \xrightarrow{\mathsf{a}_0} \\ \ldots \xrightarrow{\mathsf{a}_{n-1}} \langle \ell_n,\, \rho_n\rangle \in T \cup B\}\end{aligned} \tag{50}$$

This is a Galois connection [4, ex.11.8]. Typically, R is the identity $\mathbb{E}$ on environments $\mathbb{E}$, in which case $\alpha^{mt\to r}(\mathcal{S})\mathbb{E}$ is the program input-output relation for the trace semantics $\mathcal{S}$.

The relational semantics is

$$\mathcal{S}^r[\![\mathtt{S}]\!]R \;\triangleq\; \alpha^{mt\to r}(\mathcal{S}^{mt}(\!|\mathbb{D}^{pt}|\!)[\![\mathtt{S}]\!]R) \tag{51}$$

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 (50) and (a proof by calculational design similar to) [4, chapter 20] that the relational semantics $\mathcal{S}^r[\![\mathtt{S}]\!]$ is an instance of the maximal abstract interpreter for the following abstract domain:

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

such that

$$\begin{aligned}
\mathsf{init}^r[\![\mathtt{x}]\!]R &\;\triangleq\; \{\langle\rho,\ \rho'[\mathtt{x}\leftarrow\mathtt{x\_0}]\rangle \mid \langle\rho,\ \rho'\rangle\in R\}\\
\mathsf{assign}^r[\![\mathtt{x},\mathtt{A}]\!]R &\;\triangleq\; \{\langle\rho,\ \rho'[\mathtt{x}\leftarrow\mathcal{A}[\![\mathtt{A}]\!]\rho']\rangle \mid \langle\rho,\ \rho'\rangle\in R\}\\
\mathsf{test}^r[\![\mathtt{B}]\!]\ R &\;\triangleq\; \{\langle\rho,\ \rho'\rangle \mid \langle\rho,\ \rho'\rangle\in R\wedge\mathcal{B}[\![\mathtt{B}]\!]\rho'\}\\
\overline{\mathsf{test}}^r[\![\mathtt{B}]\!]\ R &\;\triangleq\; \{\langle\rho,\ \rho'\rangle \mid \langle\rho,\ \rho'\rangle\in R\wedge\neg\mathcal{B}[\![\mathtt{B}]\!]\rho'\}
\end{aligned}$$

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

$$\begin{aligned}
\mathcal{D}^r(\!|\mathbb{D}^r|\!) \;\triangleq\; &\langle\mathbb{P}^r,\ \dot\subseteq_2,\ \dot\emptyset_2,\ \dot\cup_2,\ \mathsf{program}^r,\ \mathsf{empty}^r,\ \mathsf{init}^r,\ \mathsf{assign}^r,\ \mathsf{skip}^r,\\
&\mathsf{test}^r,\ \overline{\mathsf{test}}^r,\ \mathsf{break}^r,\ \mathsf{compound}^r,\ \mathsf{breakable}^r\rangle
\end{aligned} \tag{52}$$

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

$$\mathsf{program}^r[\![\ell_0,\ell_1]\!]R \;=\; \mathsf{empty}^r[\![\ell_0,\ell_1]\!]R$$

11 Boundedness Hypothesis

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

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

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

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

$$\mathtt{S} \textit{ is bounded} \text{ if and only if } \beta[\![\mathtt{S}]\!] \in \mathbb{N}^+. \tag{53}$$

Static analysis can be used to determine this value, or bound it; see for example [3, 18, 5].

12 Program Equivalence

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

$$(\mathtt{P} \equiv \mathtt{P}') \triangleq (\mathcal{S}^r[\![\mathtt{P}]\!]\mathbb{E} = \mathcal{S}^r[\![\mathtt{P}']\!])\mathbb{E} \tag{54}$$

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.

13 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 than 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.

14 Program Optimization

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

15 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 [15], which is an abstract interpretation, could be equally considered.

15.1 Bounding Unrolling

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

`S`	::=		statement $\texttt{S} \in \mathbb{S}^e$
		`int` `x ;`	initilization
	|	`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

$\texttt{Sl} ::= \texttt{Sl S} \mid \epsilon$ 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 11, this can be over approximated by a static analysis.

15.2 Unrolling Abstract Domain

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

$$\mathbb{D}^u \triangleq \langle \mathbb{P}^u, \mathsf{program}^u, \mathsf{stmtlist}^u, \mathsf{empty}^u, \mathsf{init}^u, \mathsf{assign}^u, \mathsf{skip}^u, \mathsf{if}^u, \mathsf{ife}^u, \mathsf{iter}^u, \mathsf{break}^u, \mathsf{compound}^u, \mathsf{breakable}^u \rangle \tag{55}$$

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* $\texttt{P} ::= \texttt{Sl}$

$$\mathsf{program}^u[\![\texttt{P}]\!](Sl)\langle \texttt{P}, n\rangle \triangleq \mathsf{let}\ \langle \texttt{Sl}', m\rangle = Sl\langle \texttt{Sl}, n\rangle\ \mathsf{in}\ (m < 0\ \text{?}\ \langle \texttt{Sl}'\ \mathsf{error;}, m\rangle : \langle \texttt{Sl}', m\rangle) \tag{56}$$

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

$$\mathsf{stmtlist}^u[\![\texttt{Sl}]\!](Sl', S)\langle \texttt{Sl}, n\rangle \triangleq \mathsf{let}\ \langle \texttt{Sl}'', m\rangle = Sl'\langle \texttt{Sl}', n\rangle\ \mathsf{in}\ \mathsf{let}\ \langle \texttt{S}'', p\rangle = S\langle \texttt{S}, m\rangle\ \mathsf{in}\ \langle \texttt{Sl}''\ \texttt{S}'', p\rangle \tag{57}$$

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

$$\mathsf{empty}^u[\![\texttt{Sl}]\!]\langle \texttt{Sl}, n\rangle \triangleq \langle \texttt{Sl}, n\rangle \quad \text{(no computation is involved)} \tag{58}$$

- *Unrolling semantics of an initialization statement* $\texttt{S} ::= \textbf{int}\ \texttt{x}\ \texttt{;}$

$$\mathsf{init}^u[\![\texttt{S}]\!]\langle \texttt{S}, n\rangle \triangleq \langle \texttt{S}, n-1\rangle \tag{59}$$

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

$$\mathsf{assign}^u[\![\texttt{S}]\!]\langle \texttt{S}, n\rangle \triangleq \langle \texttt{S}, n-1\rangle^{9} \tag{60}$$

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* $\mathtt{S} ::= \mathtt{;}$

$$\mathsf{skip}^u[\![\mathtt{S}]\!]\langle\mathtt{S},\ n\rangle \ \triangleq\ \langle\mathtt{S},\ n-1\rangle \qquad \text{(a NOP is generated)} \tag{61}$$

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

$$\begin{aligned}\mathsf{if}^u[\![\mathtt{S}]\!](S_t)\langle\mathtt{S},\ n\rangle \ \triangleq\ & \mathsf{let}\ \langle\mathtt{S}'_t,\ n_t\rangle = S_t\langle\mathtt{S}_t,\ n-1\rangle^{10}\ \mathsf{in}\\ & \langle\mathtt{if\ (B)\ S}'_t,\ n_t\rangle\end{aligned} \tag{62}$$

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

$$\begin{aligned}\mathsf{ife}^u[\![\mathtt{S}]\!](S_t, S_f)\langle\mathtt{S},\ n\rangle \ \triangleq\ & \mathsf{let}\ \langle\mathtt{S}'_t,\ n_t\rangle = S_t\langle\mathtt{S}_t,\ n-1\rangle\\ & \mathsf{and}\ \langle\mathtt{S}'_f,\ n_f\rangle = S_f\langle\mathtt{S}_t,\ n-1\rangle\ \mathsf{in}\\ & \langle\mathtt{if\ (B)\ S}'_t\ \mathtt{else\ S}'_f,\ \max(n_t, n_f)\rangle\end{aligned} \tag{63}$$

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

$$\begin{aligned}\mathcal{F}^u[\![\mathtt{S}]\!](S_b)\langle\mathtt{S},\ k\rangle \ \triangleq\ & (\,(k \leqslant 0)\ \mathbf{?}\ \langle\mathsf{error};,\ 0\rangle\\ & \mathbf{:}\ \mathsf{let}\ \langle\mathtt{S}'_b,\ m\rangle = S_b\langle\mathtt{S}_b,\ k\rangle\ \mathsf{in}\\ & \quad \mathsf{let}\ \langle\mathtt{S}',\ p\rangle = \mathcal{F}^u[\![\mathtt{S}]\!](S_b)\langle\mathtt{S},\ m\rangle\ \mathsf{in}\\ & \quad\ \langle\mathtt{if\ (B)}\ \{\!|\ \mathtt{S}'_b\ \mathtt{S}'\ |\!\},\ p-1\rangle\,)\\ \mathsf{iter}^u[\![\mathtt{S}_b]\!](S_b)\langle\mathtt{S},\ n\rangle \ \triangleq\ & \mathsf{let}\ \langle\mathtt{S}'',\ p\rangle = \mathcal{F}^u[\![\mathtt{S}]\!](S_b)\langle\mathtt{S},\ \beta[\![\mathtt{S}]\!]\rangle\ \mathsf{in}\\ & \langle\mathtt{S}'',\ n-(\beta[\![\mathtt{S}]\!]-p)\rangle\end{aligned} \tag{64}$$

(where $\beta[\![\mathtt{S}]\!] \in \mathsf{Pc} \not\rightarrow \mathbb{N}$ specifies a sound bound (53) on the number of steps in the loop S which deduced from the global counter n)

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

$$\mathsf{break}[\![\mathtt{S}]\!]\langle\mathtt{S},\ n\rangle \ \triangleq\ \langle\mathtt{S},\ n-1\rangle \tag{65}$$

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

$$\begin{aligned}\mathsf{compound}[\![\mathtt{S}]\!](Sl)\langle\mathtt{S},\ n\rangle \ \triangleq\ & \mathsf{let}\ \langle\mathtt{Sl}',\ m\rangle = Sl(\langle\mathtt{Sl},\ n\rangle)\ \mathsf{in}\\ & \langle\{\ \mathtt{Sl}'\ \},\ m\rangle\end{aligned} \tag{66}$$

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

$$\begin{aligned}\mathsf{breakable}[\![\mathtt{S}]\!](Sl)\langle\mathtt{S},\ n\rangle \ \triangleq\ & \mathsf{let}\ \langle\mathtt{Sl}',\ m\rangle = Sl(\langle\mathtt{Sl},\ n\rangle)\ \mathsf{in}\\ & \langle\{\!|\ \mathtt{Sl}'\ |\!\},\ m\rangle\end{aligned} \tag{67}$$

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

15.3 Unroller

The unrolling of a program component is

$$\mathcal{S}^u[\![\mathtt{P}]\!] \;\triangleq\; \mathcal{S}(\!|\mathbb{D}^u|\!)[\![\mathtt{P}]\!] \tag{68}$$

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

16 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 (33), and so, by abstraction (35) the same maximal abstract semantics and by instantiation (49), the same maximal trace semantics. It follows, by abstraction (51), that they have the same relational semantics and so, by definition (54) of equivalence, that they are equivalent $\mathtt{S}^u \equiv \mathtt{S}$. In conclusion the informal requirement (1) is satisfied.

17 Static Program Analysis

Program analysis is an instance of an abstract interpreter for a reduced product [4, chapter 36] of abstract domains. For example, Astrée [8] 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 [7]. For this project we chose to use three different numerical domains, namely the polyhedral domain, the octagonal domain, and the constancy domain. The polyhedral and octagonal domains where built using the Apron [10] and Elina [**DBLP:journals/pacmpl/SinghPV18**] libraries, while the constancy domain was built specifically for this project.

17.1 Abstract Domain of a Static Analysis

A static analysis is fully specified by an abstract domain

$$\mathbb{D}^a \;\triangleq\; \langle \mathbb{P}^a,\ \sqsubseteq^a,\ \bot^a,\ \top^a,\ \sqcup^a,\ \mathsf{init}^a,\ \mathsf{assign}^a,\ \mathsf{test}^{a11},\ \overline{\mathsf{test}}^a \rangle \tag{69}$$

This abstract domain is in general complex and decomposed into many subdomains composed, for example by a reduced product, whose effect is to present the composition of these subdomains in the form (70).

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

$$\begin{aligned} \mathcal{D}^a(\!|\mathbb{D}^a|\!) \;\triangleq\; & \langle \mathbb{P}^a,\ \sqsubseteq^a,\ \bot^a,\ \top^a,\ \sqcup^a,\ \mathsf{program}^a,\ \mathsf{empty}^a,\ \mathsf{init}^a,\ \mathsf{assign}^a, \\ & \mathsf{skip}^a,\ \mathsf{test}^a,\ \overline{\mathsf{test}}^a,\ \mathsf{break}^a,\ \mathsf{compound}^a,\ \mathsf{breakable}^a \rangle \end{aligned} \tag{70}$$

defined (similarly to $\mathcal{D}^r$ at (52)) as:

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

provides an abstract domain of the same type as $\mathbb{D}^p$ in (15), 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

[11] We note that for static analysis, we use ternary logic to evaluate Boolean expressions and therefore we augment test from having a type of $\mathsf{Pc} \to (\mathsf{L} \times \mathsf{B} \times \mathsf{L}) \to \mathbb{P}^p \xrightarrow{\nearrow} \mathbb{P}^p$ to having a type of $\mathsf{Pc} \to (\mathsf{L} \times \mathsf{B} \times \mathsf{L}) \to \mathbb{P}^p \xrightarrow{\nearrow} \mathbb{P}^p * \mathsf{T}$, where $\mathsf{T} = \{t, f, tf\}$

$$\begin{aligned}\mathbb{D}^s &\triangleq \langle \mathbb{P}^s,\ \sqsubseteq^s,\ \bot^s,\ \top^s,\ \sqcup^s,\ \mathsf{init}^s,\ \mathsf{assign}^s,\ \mathsf{test}^s,\ \overline{\mathsf{test}}^s\rangle \\ \mathcal{S}^s(\!|\mathbb{D}^s|\!) &\triangleq \mathcal{S}^m(\!|\mathcal{D}^s(\!|\mathbb{D}^s|\!)|\!) \;=\; \mathcal{S}(\!|\mathcal{D}^m(\!|\mathcal{D}^s(\!|\mathbb{D}^s|\!)|\!)|\!)\end{aligned} \tag{71}$$

and the abstract domain defining the program analysis for this same abstract interpreter $\mathcal{S}^a$ is (71) then, given an increasing concretization function:

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

the following pointwise *local soundness* conditions:

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

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 [4, 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 is applied to the unrolled program, it will produce (under the boundedness hypothesis) refined results because no extrapolation (widening) / interpolation (narrowing) is necessary [4, chapter 34].

18 Abstract Domain and Their Implementations

One of the major benefits of using an abstract interpreter is that it is very modular. One can develop many different abstract domains that they can parameterize the abstract interpreter with in order to fit their needs. As stated above, for this project, we have implemented three different abstract domains: A polyhedral domain, an octagonal domain, and a constancy domain. Below we will discuss each one in more detail, as well as go over some implementation details.

18.1 Apron and Elina

To implement the polyhedra and octagonal domains, we heavily utilize the Apron [10] and Elina [**DBLP:journals/pacmpl/SinghPV18**] libraries in OCaml. While these libraries are very efficient and useful, they do have some limitations. For example, they do not support the $\neq$ operator, so we replace all statements of the form $\mathtt{B}_1 \neq \mathtt{B}_2$ with the equivalent Boolean expressions $\mathtt{B}_1 < \mathtt{B}_2 \vee \mathtt{B}_1 > \mathtt{B}_2$. Furthermore, the provided polyhedra domain only allows loose comparisons, and therefore we soundly approximate $>$ and $<$ with $\geq$ and $\leq$ respectively. Lastly, we chose to not support Boolean statements with *nand* in these two domains, as it was difficult to implement within the confines of the libraries. In future work, if we decide to implement these domains from scratch, we will support Boolean statements that contain *nand*.

18.2 Polyhedra Domain

The polyhedra domain is one which uses the tools of static analysis in order to infer linear relationships between variables in the program. To implement this domain, we used Elina's polyhedral domain, which interfaces with the Apron library. For this domain, instead of having an environment which is a mapping of variables to values, it abstracts the invariant of a program into a finite set of linear constraints of the form $Ax \leq b$, or a polyhedra, and stores them in its environment. Then, using these linear constraints, the domain can infer values and perform the required abstract functions like $\sqcup$, assign, and test. This domain is very popular as a numerical abstract domain, as it is very robust. However, the domain is very computationally expensive, with the number of constraints growing exponentially as we analyze the program. Due to this issue, and the other polyhedral domain specific problems that stem from using the Apron library, we stopped using this domain.

18.3 Octagonal Domain

The octagonal domain is a subset of the polyhedral domain, where each invariant is of the form $\pm\mathtt{x} \pm \mathtt{y} \leq c$, where x and y are variables and c is a constant. This domain is referred to as an octagon, because linear inequalities of this form will have at most eight edges on a 2-dimensional plane. While not as precise as a general polyhedral domain, an octagonal domain can be built to run in polynomial time. For the implementation of this domain we again use the Elina library.
The octagon domain also uses a set of invariants instead of a mapping of values for its environment. This allows great flexibility when testing inequalities as even if an exact value of a variable is not known, there is still information known about the

possible values of a variable. Furthermore, in certain cases, we can even use the linear relationships to infer the exact value of a variable.

One issue that arose from using the Apron library is that the function that tests a given Boolean expression will only return true if for all possible assignments of the variables in the statement the Boolean expressions is true. In every other case, the function will return false. This is an issue because if there is a possible assignment of variables which causes the statement to be false, even if there may also be a possible assignment which causes the statement to be true, the function will return false. This issue can clearly be seen in the code below.

```
([],0) int x; //[| T |]
([],1) if(x<10) { //[|-x+9.>=0|]
([],2)    if(x>5){ //[|x-6.>=0; -x+9.>=0|]
([],4)        x=6;
          }
      }
```

At the label ([],2), we know that [|-x+9.>=0|]. Therefore we know an assignment of x=6...9 would satisfy the inequality of x>5. However, due to how the Apron library was implemented, for reasons that are correct in other usages of the library, calling the function to see if x>5 is satisfiable would return false. This becomes important for specific types of program simplifications which we conduct.
We circumvent this issue using the ternary logic discussed above. For a given Boolean statement `B`, if we test both `B` and `!B`, we can determine the ternary result of the expressions. If `B` is always satisfiable, then it is known that `B` is true for all assignments of variables and therefore $\mathsf{test}[\![\mathtt{B}]\!] = T$. If `!B` is always satisfiable, then it is known that `B` is never true for any assignment of variables, and therefore $\mathsf{test}[\![\mathtt{B}]\!] = F$. Lastly if both `B` and `!B` are satisfiable for only some assignments of the variables, but not for all assignments, then $\mathsf{test}[\![\mathtt{B}]\!] = TF$

18.4 Constancy Domain

The constancy domain is a domain in which each variable is mapped to either Bot (unreachable), an integer value, or Top (unknown). Unlike the other two domains we used, no partial information about the value of a variable is stored in the environment – Either the exact value of the variable is known or it is not. This makes the domain very lightweight and fast, as there is no overhead from having a plethora of invariants. The downside of this domain is that it is very imprecise as you lose any relation. For example, consider the following code:

```
([],0) int x;
([],1) int y;
([],2) x = 1;
([],3) if(y<2) {
([],4)    if(y<3){
([],5)        x=1;
          }
([],6)   else {
              x=2;
          }
       }
```

Since if $y < 2$ would imply that $y < 3$ it is trivial for a user, and a more complex domain, to see that at the termination of the program $\rho(\mathtt{x})$ should be equal to 1. However when using the constancy domain, at program termination $\rho(\mathtt{x}) = \top$ as $\mathsf{test}[\![y < 2]\!]R = R$, meaning no information is learned after a test operation.
While a lot of information is lost in the abstraction, there are some optimizations one can make. Consider the following example:

```
([],0) int x;
([],1) int y;
([],2) x = 0;
([],3) y = y * x;
```

Normally at program termination $\rho(y) = \top$. However, since we know that any number multiplied by 0 is 0, we known that $\rho(y) = 0$. A similar optimization can be done for dividing by 0, where the evaluation of an expressions which has division by 0 is $\perp$, regardless of the value of anything else in the expression.

19 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 unrolling 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 [4, 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 [4, chapter 29].

20 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 14. 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 [4, definition 36.7]. As shown by [4, theorem 36.23], the reduced product is obtained by applying a meaning-preserving reduction operator 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 r. The solution is to define a weaker reduction operator $\mathrm{t} \in (\mathbb{D}^u \times \mathcal{D}^m(\!|\mathcal{D}^a(\!|\mathbb{D}^a|\!)|\!)) \rightarrow (\mathbb{D}^u \times \mathcal{D}^m(\!|\mathcal{D}^a(\!|\mathbb{D}^a|\!)|\!))$ 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

$$\mathtt{S}' \;=\; \mathrm{t}(\mathtt{S}, P)$$

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

$$\begin{aligned}\mathrm{t}(\mathtt{S}, P) \;&=\; \mathsf{let}\,\mathtt{S}' = \mathrm{t}(\mathtt{S}, P)\ \mathsf{in}\\ &\qquad \langle \mathtt{S}',\ P \sqcap^a \mathcal{S}^a(\!|\mathbb{D}^a|\!)[\![\mathtt{S}']\!]\rangle^{12}\end{aligned}$$

It follows that the pair $\langle \mathtt{S},\ P\rangle$ can be replaced by $\mathrm{t}(\mathtt{S}, P)$. Observe that if t is sound, increasing, and reductive then, by [4, theorem 29.2], the transformation can be improved by considering $\check{\mathrm{t}}(\mathtt{S}, P) \triangleq \mathsf{gfp}^{\sqsubseteq_2}_{\langle \mathtt{S},\ P\rangle}\ \mathrm{t}$ 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, [4, corollary 29.3] shows that it is sound, but less precise, to stop at any iterate, including doing none. Applied at the program level, $\check{\mathrm{t}}$ essentially consists in iterating the abstract interpreter.

21 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

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

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 [4, chapter 47] for the prefix trace semantics of section 7.3 and moreover, to return a refined precondition, using a backward analysis [4, chapter 50] or an iterated combination of a forward and backward analysis [4, chapter 51].

The reductive transformation $\mathsf{t}[\![\mathtt{S}]\!]$ 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* P ::= Sl
$$\begin{aligned}\mathsf{program}^{uat}[\![\mathtt{P}]\!]\langle\mathtt{P},\ n\rangle R \ \triangleq\ & \mathsf{let}\ \langle\langle\mathtt{Sl}',\ m\rangle,\ \langle T,\ B\rangle\rangle = Sl\langle\mathtt{Sl},\ n\rangle\ \mathsf{in}\\ & \mathbf{(}\,m < 0\ \mathbf{?}\ \mathsf{t}[\![\mathtt{P}]\!](\langle\mathtt{Sl}'\ \mathsf{error;},\ m\rangle, R, \langle\bot,\ B\rangle)\\ & \mathbf{:}\ \mathsf{t}[\![\mathtt{P}]\!](\langle\mathtt{Sl}',\ m\rangle, R, \langle T,\ B\rangle)\,\mathbf{)}\end{aligned} \tag{72}$$

- *Unrolling, analysis, and transformation semantics of a statement list* Sl ::= Sl′ S
$$\begin{aligned}&\mathsf{stmtlist}^{uat}[\![\mathtt{Sl}]\!](Sl', S)\langle\mathtt{Sl},\ n\rangle R \ \triangleq\\ &\quad \mathsf{let}\ \langle\langle\mathtt{Sl}'',\ m\rangle,\ \langle T',\ B'\rangle\rangle = Sl'\langle\mathtt{Sl}',\ n-1\rangle R\ \mathsf{in}\\ &\qquad \mathsf{let}\ \langle\langle\mathtt{S}'',\ p\rangle,\ \langle T'',\ B''\rangle\rangle = S\langle\mathtt{S},\ m\rangle T'\ \mathsf{in}\\ &\qquad\quad \mathsf{t}[\![\mathtt{Sl}]\!](\langle\mathtt{Sl}''\ \mathtt{S}'',\ p\rangle, R, \langle T'',\ B' \sqcup B''\rangle)\end{aligned} \tag{73}$$

- *Unrolling, analysis, and transformation semantics of an empty statement list* Sl ::= ϵ
$$\mathsf{empty}^{uat}[\![\mathtt{Sl}]\!]\langle\mathtt{Sl},\ n\rangle R \ \triangleq\ \langle\langle\mathtt{Sl},\ n\rangle,\ \langle R,\ \bot\rangle\rangle \tag{74}$$

- *Unrolling, analysis, and transformation semantics of an initilization statement* S ::= **int** x **;**
$$\mathsf{init}^{uat}[\![\mathtt{S}]\!]\langle\mathtt{S},\ n\rangle R \ \triangleq\ \mathsf{t}[\![\mathtt{S}]\!](\langle\mathtt{S},\ n-1\rangle, R, \langle\mathsf{init}^a[\![\mathtt{x}]\!]R,\ \bot\rangle) \tag{75}$$

- *Unrolling, analysis, and transformation semantics of an assignment statement* S ::= x **=** A **;**
$$\mathsf{assign}^{uat}[\![\mathtt{S}]\!]\langle\mathtt{S},\ n\rangle R \ \triangleq\ \mathsf{t}[\![\mathtt{S}]\!](\langle\mathtt{S},\ n-1\rangle, R, \langle\mathsf{assign}^a[\![\mathtt{x}, \mathtt{A}]\!]R,\ \bot\rangle) \tag{76}$$

- *Unrolling, analysis, and transformation semantics of a skip statement* S ::= **;**
$$\mathsf{skip}^{uat}[\![\mathtt{S}]\!]\langle\mathtt{S},\ n\rangle R \ \triangleq\ \mathsf{t}[\![\mathtt{S}]\!](\langle\mathtt{S},\ n-1\rangle, R, \langle R,\ \bot\rangle) \tag{77}$$

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

$$\mathsf{if}^{uat}[\![\mathtt{S}]\!](S_t)\langle \mathtt{S},\ n\rangle R \triangleq \mathsf{let}\ \langle\langle \mathtt{S}'_t,\ n_t\rangle,\ T,\ B\rangle = S_t\langle \mathtt{S}_t,\ n\rangle(\mathsf{test}^a[\![\mathtt{B}]\!]\ R)\ \mathsf{in}\quad \mathsf{t}[\![\mathtt{S}]\!](\langle \mathtt{\textbf{if}\ (B)\ S}'_t,\ n_t\rangle, R, \langle T \sqcup \overline{\mathsf{test}}^p[\![\mathtt{B}]\!]\ R,\ B\rangle) \tag{78}$$

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

$$\begin{aligned}&\mathsf{ife}^{uat}[\![\mathtt{S}]\!](S_t, S_f)\langle \mathtt{S},\ n\rangle R \triangleq\\ &\quad \mathsf{let}\ \langle\langle \mathtt{S}'_t,\ n_t\rangle,\ T_t,\ B_t\rangle = S_t\langle \mathtt{S}_t,\ n\rangle(\mathsf{test}^a[\![\mathtt{B}]\!]\ R)\\ &\quad \mathsf{and}\ \langle\langle \mathtt{S}'_f,\ n_f\rangle,\ T_f,\ B_f\rangle = S_f\langle \mathtt{S}_f,\ n\rangle(\overline{\mathsf{test}}^a[\![\mathtt{B}]\!]\ R)\ \mathsf{in}\\ &\qquad \mathsf{t}[\![\mathtt{S}]\!](\langle \mathtt{\textbf{if}\ (B)\ S}'_t\ \mathtt{\textbf{else}}\ \mathtt{S}'_f,\ \max(n_t, n_f)\rangle, R, \langle T_t \sqcup T_f,\ B_t \sqcup B_f\rangle)\end{aligned} \tag{79}$$

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

$$\begin{aligned}\mathcal{F}^{uat}[\![\mathtt{S}]\!](S_b)\langle \mathtt{S},\ k\rangle R \;\triangleq\;& (\,(k \leqslant 0)\ \mathbf{?}\ \langle \mathsf{error}\mathtt{;},\ 0\rangle\\ &\ \mathbf{:}\ \mathsf{let}\ \langle\langle \mathtt{S}'_b,\ m\rangle,\ \langle T_b,\ B_b\rangle\rangle = S_b\langle \mathtt{S}_b,\ k\rangle R\ \mathsf{in}\\ &\qquad \mathsf{let}\ \langle\langle \mathtt{S}',\ p\rangle,\ \langle T',\ B'\rangle\rangle = \mathcal{F}^{uat}[\![\mathtt{S}]\!](S_b)\langle \mathtt{S},\ m\rangle T_b\ \mathsf{in}\\ &\qquad\quad \mathsf{t}[\![\mathtt{S}]\!](\langle \mathtt{\textbf{if}\ (B)\ \{\!|\ S}'_b\ \mathtt{S}'\ \mathtt{|\!\}},\ p\rangle, R, \langle T',\ B_b \sqcup B'\rangle)\,)\\ \mathsf{iter}^{uat}[\![\mathtt{S}_b]\!](S_b)\langle \mathtt{S},\ n\rangle R \;\triangleq\;& \mathsf{let}\ \langle\langle \mathtt{S}'',\ p\rangle,\ \langle T,\ B\rangle\rangle = \mathcal{F}^{uat}[\![\mathtt{S}]\!](S_b)\langle \mathtt{S},\ \beta[\![\mathtt{S}]\!]\rangle R\ \mathsf{in}\\ &\quad \langle\langle \mathtt{S}'',\ n - (\beta[\![\mathtt{S}]\!] - p)\rangle,\ \langle T,\ B\rangle\rangle\end{aligned} \tag{80}$$

(where $\beta[\![\in]\!]\mathsf{Pc} \not\rightarrow \mathbb{N}$ specifies a sound bound (53) on the number of steps in the loop which deducted from the global counter n)

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

$$\mathsf{break}[\![\mathtt{S}]\!]\langle \mathtt{S},\ n\rangle R \triangleq \mathsf{t}[\![\mathtt{S}]\!](\langle \mathtt{S},\ n-1\rangle, R, \langle \bot,\ R\rangle) \tag{81}$$

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

$$\begin{aligned}&\mathsf{compound}[\![\mathtt{S}]\!](Sl)\langle \mathtt{S},\ n\rangle R \triangleq\\ &\quad \mathsf{let}\ \langle\langle \mathtt{Sl}',\ m\rangle,\ \langle T',\ B'\rangle\rangle = Sl(\langle \mathtt{Sl},\ n\rangle)R\ \mathsf{in}\\ &\qquad \mathsf{t}[\![\mathtt{S}]\!](\langle \mathtt{\{\ Sl}'\ \mathtt{\}},\ m\rangle, R, \langle T',\ B'\rangle)\end{aligned} \tag{82}$$

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

$$\begin{aligned}
&\mathsf{breakable}[\![\mathtt{S}]\!](Sl)\langle \mathtt{S},\ n\rangle R \ \triangleq \qquad (83)\\
&\quad \mathsf{let}\ \langle\langle \mathtt{Sl}',\ m\rangle,\ \langle T',\ B'\rangle\rangle = Sl(\langle \mathtt{Sl},\ n\rangle)R\ \mathsf{in}\\
&\qquad \mathsf{t}[\![\mathtt{S}]\!](\langle \mathtt{\{|\ Sl'\ |\}},\ m\rangle, R, \langle T' \sqcup B',\ \bot\rangle)
\end{aligned}$$

22 Program Optimizing Transformations and Corresponding Abstract Domains

The objective of the transformer reduction $\mathsf{t}[\![\mathtt{S}]\!](\langle \mathtt{S}',\ m\rangle, R, \langle T,\ B\rangle)$ 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[13] 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 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 [11].

22.1 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 [7, 1], between abstract domains themselves and with the transformer; see [4, sction 36.4.6].

22.2 Evaluation of Expressions

A major form of simplification that is done in this project is the simplification of expressions (both arithmetic and Boolean) so that they are more easily evaluated at

[13]and maybe other analyzes as already mentioned.

runtime. Given an expression `A` we execute the following steps: [14] [15]

1. For each variable `x`, where $\rho(\mathtt{x})$ is an integer, we replace all instances of that variable with $\rho(\mathtt{x})$. This includes all array indices as well.

2. We recursively distribute all negative signs, so that there is no subtractions, only addition between negative numbers or negative constants. For example the expression $\mathtt{x} - (\mathtt{y} + \mathtt{z} + 1)$ would become $\mathtt{x} + ((-1\mathtt{y}) + (-1\mathtt{z}) + (-1))$. This is done so that the expressions become associative and commutative, allowing us to freely move each addend in the expression. We note that we would not be able to do this with the float data type, as it is not associative.

3. Once all the addends are able to be freely moved around in an expression, we combine all equal addends. To do this, we create a hashmap, where the keys are arithmetic expressions and the values are how many times those expressions appear as an addend.
   We note that if an arithmetic expression is a product not between a constant and a variable, then we recursively simplify each factor of the product such that they are in their simplest form, and then we take the entire multiplication as a single addend.

4. We sum all constant values in the expression.

5. Using the hashmap generated in step 3 and the summation of the constants in step 4, we rebuild a simplified expression in which all repeated expressions are combined, and all constants are summed. For example $\mathtt{x}+3\mathtt{x}+5+\mathtt{x}*\mathtt{y}+\mathtt{x}*\mathtt{y}+3$ would simplify to the expression $4\mathtt{x} + 2\mathtt{x} * \mathtt{y} + 8$.

6. We convert all negative coefficients back into subtraction operations. For example, $\mathtt{x} + (-1 * \mathtt{y})$ would become $\mathtt{x} - \mathtt{y}$. While this step provides very little actual code optimization, it makes the expressions much easier to read.

7. If the expression is a Boolean expression, and if all values in the expression are constants, we evaluate the expression; We replace the expression with 1 if it evaluates to true and 0 if it is false.

[14]We note that practically some of these are done in parallel with one another, how ever the relative order of these steps are kept (Meaning step 1 and 2 may happen in parallel, but step 2 will not happen before step 1).

[15]Since array indices are themselves an expression, we first recursively evaluate each array index in an expression before we attempt to evaluate an expression itself.

22.3 Elimination of Dead Code

Dead code is code which is never run. This can be because it is part of a conditional statement which is never met, or it is after a break statement. Using ternary logic to evaluate Boolean statements, we are able to statically check which branches condition for a conditional will never be met for any assignment of initial values for a variable, and then remove those branches. In a similar vein, we can also check which branches have conditions that are always met, and remove everything but the body of that branch, as there is no need to test an expression which will always be true. If a condition is met for some initial values, but not all initial values, then we leave the conditional statement as is.
In addition to the removal of dead code, we also remove unnecessary code. This includes all skip statements, and empty compound or breakable statements.

22.4 Changing Breakable to Compound

As stated above, while unrolling we change all **`while`** `(B) S` to the following form: **`if`** `(B)` **`{|`** `S` **`while`** `(B) S` **`|}`**. However, if $\mathsf{escape}[\![\mathsf{S}]\!] = F$, then there is no need to use a breakable statement; a compound statement would work. Therefore, we make the small optimization that if a we have a breakable statement of the form **`{|`** `S` **`|}`** and $\mathsf{escape}[\![\mathsf{S}]\!] = F$, we convert the breakable statement to a compound statement.

23 Future Work

23.1 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 to replace 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.

23.2 Expanding Numerical Types

Currently the unroller and transformer only work with integers. However it is an aim to allow other types such as machine integers and floats. This becomes complicated because while floats are sound in intervals, they are not always sound when using an octagonal domain. Furthermore, as stated above, the process in which we simplify the code does not work with floats as they are not associative under addition. Therefore, it would require some retooling of the domain to make all approximations sound.

23.3 Tracking Specific Variables

In many programs, only a small number of variables contain useful information as many variables are used as sentinel values or loop counters. Therefore we will suggest that we remove all assignments to variables which do not effect the tracked variables. For example, consider that we are only interested in x in the following code:

```
([],0) int x;
([],1) int y;
([],2) y = 0;
([],3) while(y<100){
([],4)    x=x+1;
([],5)    y=y+1;
       }
```

After being unrolled and optimized, the code would look like:

```
([],0) int x;
([],1) int y;
([],2) y = 0;
([1],3) x=x+1;
([1],4) y=1;
([2],3) x=x+1;
([2],4) y=2;
...
([100],3) x=x+1;
([100],4) y=100;
```

In this code there are a lot of extraneous assignments to y. Therefore we propose the remove all assignments to y, as they are not needed, shrinking the size of the code in half.

23.4 Streaming

Observe that, in full generality, the optimizing transformation t at (72) 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 t at assignments (76), skips (77), tests (78), (79), and (80), and breaks (81) only. Then theoretically the unrolled program can be streamed out, thus considerably reducing memory consumption.

23.5 Piecewise Partial Unrolling

Once unrolled, it might be that the program is much too large to be accepted by compilers. Usually, a compiler transforms a programs into an intermediate representation, which is analyzed and transformed into the output code. 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 [14].

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] 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.

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

[3] 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.

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

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

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

[7] 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.

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

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

[10] 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.

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

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

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

[14] 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.

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

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

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

[18] 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.

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

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