On the Relative Completeness of Bytecode Analysis Versus Source Code Analysis

Francesco Logozzo and Manuel Fähndrich

Microsoft Research
{logozzo,maf}@microsoft.com

Abstract. We discuss the challenges faced by bytecode analyzers designed for code verification compared to similar analyzers for source code. While a bytecode-level analysis brings many simplifications, *e.g.*, fewer cases, independence from source syntax, name resolution, etc., it also introduces precision loss that must be recovered either via preprocessing, more precise abstract domains, more precise transfer functions, or a combination thereof.

The paper studies the *relative* completeness of a static analysis for bytecode compared to the analysis of the program source. We illustrate it through examples originating from the design and the implementation of `Clousot`, a generic static analyzer based on Abstract Interpretation for the analysis of MSIL.

1 Introduction

We are interested in static program analysis for program verification, where the goal is to infer *invariants* that are sufficient to discharge assertions which appear in the program either explicitly (specified by the user through assertions) or implicitly (*e.g.*, array bound checks, null dereferences, division by zero, etc.). Such analyses need to be precise enough to validate the assertions. In this paper, we will focus our attention on static analyses for program verification and we call these PSA, *Precise enough Static Analyses*.

PSA are often designed to work at the program source level, *e.g.*, [5,17,18,6,26]). There are many reasons for that. The program source provides a uniform view which abstracts machine details. Source code analysis is also able to directly exploit program structure, such as loops, to increase the precision via techniques such as reductive iterations [12], and the narrowing application by re-execution from a post-fixpoint [8].

As we will see in this paper, the most immediate benefit of source analysis however is that it provides the analysis designer with a *large code window*, allowing him/her to specialize transfer functions for extra precision.

The analysis of low level code provides different advantages: 1) it is more faithful, as it analyzes the code that is actually executed (or closer to), 2) it enables the analysis of libraries when source code is not available, 3) the analyzer avoids redundant work that the compiler performed, such as name resolution, type checking, template/generics instantiation, 4) the semantics of high-level

L. Hendren (Ed.): CC 2008, LNCS 4959, pp. 197–212, 2008.
© Springer-Verlag Berlin Heidelberg 2008

constructs that are expanded by the compiler, such as `try...catch...finally`, delegates, partial classes in C#, or generics in C# and Java, need not be duplicated. As a consequence a low-level code analyzer needs to deal with many fewer constructs than a source analyzer, reducing its complexity. Finally, 5) the analyzer can be language independent; e.g., analyzing the common target language MSIL of the .NET platform provides analysis of C#, VB, Managed C++, F#.

Because of these advantages, plenty of static analyses have been developed for low-level code. Most of them address non-relational properties like type checking [14,16,25], non-cyclicity [27], nullness [10], etc. Others target numerical properties, *e.g.*, to check buffer overruns [3] or array accesses [20].

Our observation is that while writing a static analyzer for a low-level language or bytecode is simpler than writing one for source code due to the above advantages, it is non-trivial to match the precision of a similar analysis performed at source level, due to the missing high-level structure and the reduced size of the *code window* used by transfer functions. The rest of this paper elucidates this observation with examples and general principles.

Example 1 (Motivating Example). Suppose we analyze a program containing the high level statement $\mathcal{S} \equiv \texttt{assume}\ \texttt{x} - \texttt{y} \leq 7$, using the difference bounds abstract domain [22]. At source level, the constraint $\texttt{x} - \texttt{y} \leq 7$ is a difference constraint, and it can be represented faithfully by the abstract state. Now consider the compilation of $\mathcal{S}$ into three address code:

$$\begin{aligned} 0 &: \texttt{t}_1 \leftarrow \texttt{x} - \texttt{y} \\ 1 &: \texttt{t}_2 \leftarrow \texttt{t}_1 \leq 7 \\ 2 &: \texttt{assume}\ \texttt{t}_2 \end{aligned}$$

Analyzing this code sequence with the same domain used at the source level raises immediate problems:

Expression complexity. The assignment at line 0 involves three variables, which cannot be captured precisely by the difference bounds domain. As a consequence, the abstract value for $\texttt{t}_1$ is $\top$.

Type complexity. At line 1, $\texttt{t}_2$ is assigned the result of a boolean expression[1]. At the source level there was no such boolean assignment, and in fact, the domain used at source level cannot encode the relation between $\texttt{t}_1$ and $\texttt{t}_2$.

As a result, the analysis of the code sequence using the same domain as at the source level produces an abstract state that contains no information about the relation of x and y. Several solutions are possible to mitigate the above problems.

- Use a more precise numerical abstract domain for the low-level analysis that handles relations among more than two variables, such as Octahedra [7], or Polyhedra [9,2]. This approach however leads to scaling problems, as these domains exhibit exponential complexity. No polynomial domains are known that can handle more than two variables [23].

[1] Please note that this case is orthogonal to the previous one, *i.e.*, the problem shows up even if the assignment was $\texttt{t}_2 \leftarrow (\texttt{x} - \texttt{y} \leq 7)$.

- Split the current abstract domain in two at the boolean assignment: one where $\texttt{t}_2 == \texttt{true}$ and one where $\texttt{t}_2 == \texttt{false}$. This method has two main drawbacks: (i) it may lead to exponential explosion by doubling the abstract states at each conditional branch; and (ii) it still introduces loss of precision, because the relation to be assumed at line 2 is lost.
- A more general solution which addresses both of the problems and all others related to the limited code window, is to use a lightweight symbolic abstract domain to compute available expressions at each program point.

Let us briefly sketch how the use of a symbolic domain to recover expressions works on the example. At line `2`, the analysis *first* asks the symbolic domain to refine variable $\texttt{t}_2$. This refinement, using line 1, produces $\texttt{t}_2 \equiv \texttt{t}_1 \leq 7$, which can be further refined, using line 0, to produce $\texttt{t}_2 \equiv \texttt{x} - \texttt{y} \leq 7$. The analysis *then* passes the refined expression $\texttt{x} - \texttt{y} \leq 7$ to the difference bounds domain, which handles it exactly as the source analysis does. □

As the example shows, PSA of low-level code requires more than just reusing the domains suitable for high-level code, otherwise, precision is lost. In this paper, we investigate the *relative* completeness of low-level code analysis versus source code analysis, i.e., what is required for bytecode analysis to be as precise as source code analysis, without requiring the use of domains with worse complexity.

We present representative issues that crop up when designing precise and scalable bytecode analyses. We faced those issues during the design and implementation of `Clousot` [19], a PSA for .NET based on abstract interpretation. The issues described are not specific to .NET, but arise for all low-level analyses. They manifest in (i) the precise handling of assignments, tests and branches, and (ii) the fixpoint iteration strategy, in particular for narrowing and reductive iterations. We discuss how to overcome these issues, and the solutions we have adopted in `Clousot`. In general, quantifing the impact of such issues is hard. We tried a rough (under-)estimation by switching off some precision refinements discussed in this paper (not all of them could be switched off, as many are buried deep in the architecture of `Clousot`). We obtained a loss of precision of 10% in the analysis of the array accesses of `mscorlib.dll`, the main library in the .NET framework. Such loss of precision is enough to generate more than 1400 false positives, *i.e.*, to make the analysis *de facto* unuseful.

2 Languages

We use a while-language as a representative for high-level languages, and a three address code instruction set as a representative of low-level code.

2.1 While-Language

Our high level language is a simple while-language with no dynamic memory allocation, shown in Fig. 1. The semantics is standard. We use a single type, integers. Following widespread convention, we assume that `0` stands for `false`

```
 Stm ::= skip; | Var := Exp; | Stm Stm | while(BExp) {Stm}; | if(BExp) {Stm }else {Stm }; |
         assume BExp; | assert BExp;
 Exp ::= Lit | Exp op Exp
BExp ::= Lit | Exp relop Exp |  !(BExp) | BExp && BExp | BExp || BExp
 Lit ::= Var | int          Var ::= ··· | x | y | ...        int ::= ··· | −1 | 0 | 1 | ...
  op ::= + | − | ∗ | /        relop ::=< | ≤ | ==
```

Fig. 1. The while-language: a high-level language

```
IstrStream ::= Label : Istr | Label : Istr '\n' IstrStream | ϵ
     Label ::= 0 | ... | 2^32
      Istr ::= Var ← ExpTwoOps |
               jmp Label | jmpIf Var Label | assert Lit | assume Lit | nop
 ExpTwoOps ::= Lit | Lit op Lit | Lit relop Lit | Lit && Lit | Lit || Lit
```

Fig. 2. Three address code: a low-level language

and all the other integers for `true`. Boolean expressions *shortcut* evaluation. We also consider assert and assume statements, which enable assume/ guarantee reasoning, *e.g.*, to (abstract) method calls. The statement `assert e`; checks if the expression `e` holds. If it does not, then the program fails. The statement `assume e`; acts as an execution guard for the following statements. If the condition does not hold, execution gets stuck.

2.2 Three Address Code

Our low-level language is a three address code instruction set shown in Fig. 2. This language is higher level than MSIL, Java bytecode, or assembly, but it simplifies our presentation and is sufficient to exhibit the problems of interest.

An instruction stream is a sequence of labeled instructions. An assignment instruction $\mathtt{x} \leftarrow \mathtt{e}_{\mathtt{2ops}}$ updates the value of the variable `x` with the result of the evaluation of the expression $\mathtt{e}_{\mathtt{2ops}}$ which contains *at most* two operands. As a consequence the expressions that can be atomically evaluated and assigned at low level are a subset of those at higher level, *i.e.*, $\mathtt{ExpTwoOps} \subseteq \mathtt{Exp} \cup \mathtt{BExp}$. In the next sections, we will see how this impacts the precision and performances of PSA.

2.3 Compilation

We assume two compilation functions: $\mathcal{C} \in [\mathtt{Stm} \rightarrow \mathtt{IstrStream}]$ compiles a program expressed in the high-level language into a low-level instruction stream one, and $\mathcal{C}_e \in [(\mathtt{Exp} \cup \mathtt{BExp}) \rightarrow \mathtt{IstrStream}]$ compiles expressions into a sequence of instructions for evaluating them. The result of the evaluation is in a (reserved) variable `res`. We expect the functions $\mathcal{C}$ and $\mathcal{C}_e$ to perform naive compilation, *i.e.*, a straightforward translation without any program optimization [1].

3 Abstract Interpretation

Abstract interpretation is a theory of approximations [8]. It formalizes the intuition that semantics are more or less precise depending on the observation level. The more precise the abstract semantics, the more precise the properties about the execution of the program it captures. A static analysis is an abstract semantics which is rough enough to be computable. A precise static analysis is a static analysis which is precise *enough* to capture the properties of interests, *e.g.*, those needed to prove the absence of certain runtime errors.

3.1 Abstract Domains

An abstract domain $\bar{\mathsf{D}}$ is the complete lattice $\langle \mathsf{E}, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$, where E is the set of abstract elements, ordered according to the relation $\sqsubseteq$. The smallest abstract element is $\bot$, the largest is $\top$. The join $\sqcup$, and the meet $\sqcap$, are also defined. With a slight abuse of notation, we will confuse an abstract domain $\bar{\mathsf{D}}$ with the set of its elements E.

The elements of an abstract domain are related to the concrete domain D (also a complete lattice), by means of a monotonic concretization function $\gamma \in [\bar{\mathsf{D}} \to \mathsf{D}]$. In this paper we assume the concrete domain to be the complete boolean lattice $\mathcal{P}(\Sigma)$, where $\Sigma = [\mathtt{Var} \to \mathbb{Z}]$.

Given two abstract domains, $\bar{\mathsf{D}}_1$ and $\bar{\mathsf{D}}_2$, their reduced cartesian product is $\bar{\mathsf{D}}_1 \otimes \bar{\mathsf{D}}_2$, whose elements are pairs which satisfy the reduction condition:

$$\forall \langle \bar{\mathsf{d}}_1, \bar{\mathsf{d}}_2 \rangle \in \bar{\mathsf{D}}_1 \otimes \bar{\mathsf{D}}_2.\ \gamma_{\bar{\mathsf{D}}_1 \otimes \bar{\mathsf{D}}_2}(\langle \bar{\mathsf{d}}_1, \bar{\mathsf{d}}_2 \rangle) \subseteq \gamma_{\bar{\mathsf{D}}_1}(\bar{\mathsf{d}}_1) \cap \gamma_{\bar{\mathsf{D}}_2}(\bar{\mathsf{d}}_2)\ .$$

An abstract domain is said to be *relational* if it keeps relations between program variables. Otherwise it is said to be *non-relational*.

The elements of the abstract domain of intervals, Intv, are $\{[i, s] \mid i, s \in \mathbb{Z} \cup \{-\infty, +\infty\}\}$. The concretization function, $\gamma_{\mathsf{Intv}} \in [\mathsf{Intv} \to \mathcal{P}(\mathbb{Z})]$ is defined as $\gamma_{\mathsf{Intv}}([i, s]) = \{z \in \mathbb{Z} \mid i \leq z \leq s\}$. The abstract domain of boxes, Boxes, is the functional lifting of Intv, *i.e.*, $\mathsf{Boxes} = [\mathtt{Vars} \to \mathsf{Intv}]$. The concretization of a box, $\gamma_{\mathsf{Boxes}} \in [\mathsf{Boxes} \to \mathcal{P}(\Sigma)]$ is defined as $\gamma_{\mathsf{Boxes}}(f) = \{\sigma \in \Sigma \mid \forall \mathtt{x}.\mathtt{x} \in \mathrm{dom}(f) \Longrightarrow \sigma(\mathtt{x}) \in \gamma_{\mathsf{Intv}}(f(\mathtt{x}))\}$. From the definition of γ_{Boxes}, it follows that the meaning of a variables in Boxes is independent from all the others, which implies that Boxes is a non-relational abstract domain. The time and space complexity of the operations on Boxes is $\mathcal{O}(n)$, where n is the number of variables.

The abstract domain of Polyhedra, Poly [9], captures linear constraints between program variables: $\sum_{i=0}^{i<n} a_i * \mathtt{x}_i \leq z$, with $a_i, z \in \mathbb{Z}$. The concretization function $\gamma_{\mathsf{Poly}} \in [\mathsf{Poly} \to \mathcal{P}(\Sigma)]$ is defined as the intersection of all the constraints : $\gamma_{\mathsf{Poly}}(P) = \bigcap_{\sum_{i=0}^{i<n} a_i * \mathtt{x}_i \leq z \in P} \{\sigma \in \Sigma \mid \sum_{i=0}^{i<n} a_i * \sigma(\mathtt{x}_i) \leq z\}$. From the concretization function, it follows that Poly can capture properties between an arbitrary number of variables, thus it is a relational domain. The complexity of Poly is $\mathcal{O}(2^n)$ both in space and time.

3.2 Transfer Functions

Abstract interpreters implement an upper approximation $\bar{\tau}$ of the best abstract transformer $\bar{\tau}^*$, *i.e.* $\forall \bar{\mathsf{d}} \in \bar{\mathsf{D}}.\ \bar{\tau}^*(\bar{\mathsf{d}}) \sqsubseteq \bar{\tau}(\bar{\mathsf{d}})$. An abstract transfer function $\bar{\tau}$ is (i) usually hand-crafted, and (ii) tuned to maximize the precision/cost trade-off.

It is common practice for the implementation of an abstract domain $\bar{\mathsf{D}}$ to provide two abstract transfer functions: one for the assignment and one for the handling of tests [5,18,28]. The assignment abstract transfer function, $\bar{\mathsf{D}}.\mathsf{assign}$, is an over-approximation of the states reached with the concrete assignment:

$$\forall \mathtt{x}, \mathsf{e}. \forall \bar{\mathsf{d}}.\ \{\sigma[\mathtt{x} \mapsto v] \mid \sigma \in \gamma(\bar{\mathsf{d}}), [\![\mathsf{e}]\!](\sigma) = v\} \subseteq \gamma(\bar{\mathsf{D}}.\mathsf{assign}(\bar{\mathsf{d}}, \mathtt{x}, \mathsf{e})).$$

The test abstract transfer function, $\bar{\mathsf{D}}.\mathsf{test}$, acts as a kind of filter to the input states:

$$\forall \mathsf{e}. \forall \bar{\mathsf{d}}.\ \{\sigma \in \gamma(\bar{\mathsf{d}}) \mid [\![\mathsf{e}]\!](\sigma) \neq 0\} \subseteq \gamma(\bar{\mathsf{D}}.\mathsf{test}(\bar{\mathsf{d}}, \mathsf{e})).$$

It is vital for a PSA to provide a precise approximation of test.

4 Relative Completeness of Precise Analysis of Bytecode

In this section, we define a generic abstract semantics for the high level language, $\bar{\mathbb{H}}[\![\cdot]\!] \in [\mathtt{Stm} \rightarrow \bar{\mathsf{D}} \rightarrow \bar{\mathsf{D}}]$, by structural induction. In parallel, we define the abstract semantics for the low level language, $\bar{\mathbb{L}}[\![\cdot]\!] \in [\mathtt{IstrStream} \rightarrow \bar{\mathsf{D}} \rightarrow \bar{\mathsf{D}}]$. For each kind of statement and expression, we (i) express whether and under what conditions $\bar{\mathbb{L}}[\![\cdot]\!]$ is complete w.r.t. $\bar{\mathbb{H}}[\![\cdot]\!]$, *i.e.*, when $\bar{\mathbb{L}}[\![\cdot]\!]$ is as precise as $\bar{\mathbb{H}}[\![\cdot]\!]$, and (ii) show how best to overcome precision problems, *e.g.*, by refining the abstract domain or the transfer functions.

4.1 Notions of Relative Completeness

We distinguish two notions of relative completeness: strong and weak. Strong relative completeness requires the low-level analysis not to lose information when using the *same* abstract domain. Weak relative completeness allows the low-level analysis to use a *refinement* of the abstract domain used at source level.

Definition 1 (Strong Relative Completeness). *Given statement* $\mathtt{Stm}$*, abstract domain* $\bar{\mathsf{D}}$*, and projection function* $\pi \in [\bar{\mathsf{D}} \rightarrow \bar{\mathsf{D}}]$*, which removes all the temporary variables introduced by compilation, if*

$$\forall \bar{\mathsf{d}} \in \bar{\mathsf{D}}.\ \pi(\bar{\mathbb{L}}[\![\mathcal{C}(\mathtt{Stm})]\!](\bar{\mathsf{d}})) \sqsubseteq \bar{\mathbb{H}}[\![\mathtt{Stm}]\!](\bar{\mathsf{d}}), \tag{1}$$

then $\bar{\mathbb{L}}[\![\cdot]\!]$ *is strong-relatively complete w.r.t. to* $\bar{\mathbb{H}}[\![\cdot]\!]$ *for statement* $\mathtt{Stm}$*.*

Note that the definition above does not require equality of precision, only subsumption. It may be the case that the analysis at the bytecode level is more precise in some cases.

Definition 2 (Weak Relative Completeness). *Given statement* $\mathtt{Stm}$*, two abstract domains* $\bar{\mathsf{D}}$ *and* $\bar{\mathsf{D}}^+$ *such that* $\bar{\mathsf{D}}^+$ *is more precise than* $\bar{\mathsf{D}}$: $\bar{\mathsf{D}}^+ \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \bar{\mathsf{D}}$*, and projection function* $\pi \in [\bar{\mathsf{D}}^+ \rightarrow \bar{\mathsf{D}}^+]$*, which removes all the temporary variables introduced by compilation, if*

$$\forall \bar{\mathsf{d}} \in \bar{\mathsf{D}}.\ \alpha(\pi(\bar{\mathbb{L}}[\![\mathcal{C}(\mathtt{Stm})]\!](\gamma(\bar{\mathsf{d}})))) \sqsubseteq \bar{\mathbb{H}}[\![\mathtt{Stm}]\!](\bar{\mathsf{d}}), \tag{2}$$

then $\bar{\mathbb{L}}[\![\cdot]\!]$ *is weak-relatively complete w.r.t. to* $\bar{\mathbb{H}}[\![\cdot]\!]$ *for statement* $\mathtt{Stm}$ *up to the refined domain* $\bar{\mathsf{D}}^+$*.*

Weak relative completeness relaxes the previous definition by enabling the use of a more precise abstract domain for the analysis of the bytecode. It is evident that strong relative completeness implies weak relative completeness.

4.2 Skip

Handling of `skip` is straightforward: $\bar{\mathbb{H}}[\![\mathtt{skip}]\!] = \lambda\bar{\mathsf{d}}.\bar{\mathsf{d}}$. The `skip` statement is compiled with a `nop`: $\mathcal{C}(\mathtt{skip}) = n : \mathtt{nop}$, and $\bar{\mathbb{L}}[\![n : \mathtt{nop}]\!] = \lambda\bar{\mathsf{d}}.\bar{\mathsf{d}}$. As a consequence, in this case the bytecode analysis is trivially strongly complete.

4.3 Sequence

The analysis of a sequence of statements is usually just the composition of the analyses:

$$\bar{\mathbb{H}}[\![\mathtt{Stm}_1\mathtt{Stm}_2]\!] = \bar{\mathbb{H}}[\![\mathtt{Stm}_2]\!] \circ \bar{\mathbb{H}}[\![\mathtt{Stm}_1]\!]. \tag{3}$$

The compilation is the juxtaposition of two sequences of instructions:

$$\mathcal{C}(\mathtt{Stm}_1\mathtt{Stm}_2) = \begin{bmatrix} \mathcal{C}(\mathtt{Stm}_1) \\ \mathcal{C}(\mathtt{Stm}_2) \end{bmatrix} .$$

The abstract semantics of a sequence of instructions is the compositions of the analyses:

$$\bar{\mathbb{L}}[\![k : \mathtt{Istr}\ '\backslash n'\ \mathtt{IstrStream}]\!] = \bar{\mathbb{L}}[\![\mathtt{IstrStream}]\!] \circ \bar{\mathbb{L}}[\![k : \mathtt{Istr}]\!]. \tag{4}$$

Assuming that low-level analysis is complete (resp. weakly complete) for the subsequences, from (i) the fact that projection is an abstraction; and (ii) the monotonicity of the abstract functions, it follows that the low-level analysis of the sequence is complete (resp. weakly complete) w.r.t. the high-level analysis.

Note that in general, sequencing may cause loss of precision for both high- and low-level analysis w.r.t. the concrete semantics.

4.4 Assignments

A source language analysis just passes the assignment to the underlying abstract domain $\bar{\mathsf{D}}$:

$$\bar{\mathbb{H}}[\![\mathtt{x} := \mathsf{e};]\!] = \lambda\bar{\mathsf{d}}.\bar{\mathsf{D}}.\mathsf{assign}(\bar{\mathsf{d}}, \mathtt{x}, \mathsf{e}). \tag{5}$$

The compilation of the assignment generates a sequence of instructions to evaluate $\mathtt{e}$, and an assignment of the result to $\mathtt{x}$:

$$\mathcal{C}(\mathtt{x := e;}) = \begin{bmatrix} \mathcal{C}_e(\mathtt{e}) \\ k : \mathtt{x} \leftarrow \mathtt{res} \end{bmatrix}. \tag{6}$$

Without loss of generality, we will assume in the sequel that the last instruction of $\mathcal{C}_e(\mathtt{e})$ assigns directly to the target variable $\mathtt{x}$ instead of $\mathtt{res}$. Thus, the final assignment is similarly passed to underlying abstract domain:

$$\bar{\mathbb{L}}[\![k : \mathtt{x} \leftarrow \mathsf{e2op}]\!] = \lambda\sigma.\bar{\mathsf{D}}.\mathsf{assign}(\sigma, \mathtt{x}, \mathsf{e2op}). \tag{7}$$

If the source expression $\mathtt{e}$ is such that $\mathtt{e} \equiv \mathtt{l}$ or $\mathtt{e} \equiv \mathtt{l}_1 \mathtt{op}\, \mathtt{l}_2$, where $\mathtt{l}, \mathtt{l}_1, \mathtt{l}_2 \in \mathtt{Lit}$, and $\mathtt{op}$ is as in Fig. 1, then (5), (6) and (7) imply the strong relative completeness of $\bar{\mathbb{L}}[\![k : \mathtt{x} \leftarrow \mathsf{e2op}]\!]$. However, this is not the case for more complex expressions, as the next (counter-) examples show.

Example 2 (Precision Loss using Interval Arithmetic). Suppose we use the Boxes domain to analyze the assignment $\mathcal{A} \equiv \mathtt{z} := (\mathtt{x}+\mathtt{y}) * \mathtt{y}$. Let $\bar{\mathsf{b}}_0 = [\mathtt{x} \mapsto [2,3], \mathtt{y} \mapsto [-1,1]]$ be the abstract input state. Then

$$\bar{\mathbb{H}}[\![\mathtt{z} := (\mathtt{x}+\mathtt{y}) * \mathtt{y};]\!](\bar{\mathsf{b}}_0) = \bar{\mathsf{b}}_0[\mathtt{z} \mapsto [-2,4]],$$

using a specialized source transfer function. On the other hand, the compilation of $\mathcal{A}$ is

$$\mathcal{C}(\mathtt{z} := (\mathtt{x}+\mathtt{y}) * \mathtt{y};) = \begin{bmatrix} 0 : \mathtt{t} \leftarrow \mathtt{x}+\mathtt{y} \\ 1 : \mathtt{z} \leftarrow \mathtt{t} * \mathtt{y} \end{bmatrix}, \tag{8}$$

so that the abstract state after the program point 0 is $\bar{\mathsf{b}}_0[\mathtt{t} \mapsto [1,4]]$, and hence the abstract post-state is $\bar{\mathbb{L}}[\![\mathcal{C}(\mathtt{z} := (\mathtt{x}+\mathtt{y}) * \mathtt{y};)]\!](\bar{\mathsf{b}}_0) = \bar{\mathsf{b}}_0[\mathtt{t} \mapsto [1,4], \mathtt{z} \mapsto [-\mathbf{4},4]]$. □

The example shows that the analysis of the compiled code introduces a loss of precision w.r.t. to a specialized source level transfer function. Intuitively, it is caused by the fact that the domain Boxes is non-relational, and hence at program point 1 it has lost the information that $\mathtt{t}$ depends on $\mathtt{y}$, so that two spurious cases are introduced.

As the incompleteness originates from the use of a non-relational numerical domain, one may advocate the usage of a relational domain. If we chose to analyze (8) with Oct, the problem, unfortunately, does not go away. At program point 0, we have an assignment that involves *three* variables. The domain cannot track the relation between $\mathtt{t}$, $\mathtt{x}$ and $\mathtt{y}$. As a consequence, no improvement is obtained at 1 using Octagons.

If we chose instead to analyze (8) with Poly, then the assignment at 0 can be precisely captured by this domain. So the abstract post-state is $\bar{\mathsf{p}} = \{2 \leq \mathtt{x} \leq 3, -1 \leq \mathtt{x} \leq 1, \mathtt{t} - \mathtt{x} - \mathtt{y} = 0\}$. The instruction at 1 involves a quadratic expression (the multiplication of two variables), which a naive implementation of Poly.assign may simply decide to ignore. However, it is easy to see how a more refined implementation can figure out that, because of $\bar{\mathsf{p}}$, $\mathtt{t} = \mathtt{x} + \mathtt{y}$ it can use this equality to simplify the multiplication and infer the tightest lower bound $-2 \leq \mathtt{z}$, and hence satisfy (2).

Example 3 (Precision Loss using Octagons). Let us analyze the assignment $\mathcal{B} \equiv \texttt{z} := 2*\texttt{x}-\texttt{y};$ with the Oct domain. Let the initial abstract state be $\bar{\mathsf{o}}_0 = \{\texttt{x}-\texttt{y} \leq 1, \texttt{y}-\texttt{x} \leq -1\}$. Even if the source expression is not in the octagonal form, the designer of the domain can refine $\mathsf{Oct.assign}$ (i) to replace $\texttt{x}$ in the right hand side of the $\mathcal{B}$ by $\texttt{y}-1$, and (ii) to perform the basic algebraic simplifications, so that

$$\bar{\mathbb{H}}[\![\texttt{z} := 2*\texttt{x}-\texttt{y};]\!](\bar{\mathsf{o}}_0) = \bar{\mathsf{o}}_0 \cup \{\texttt{z}-\texttt{y} \leq 2, \texttt{y}-\texttt{z} \leq -2\}.$$

On the other hand, the compilation of $\mathcal{B}$ is

$$\mathcal{C}(\texttt{z} := 2*\texttt{x}-\texttt{y};) = \begin{bmatrix} 0 : \texttt{t} \leftarrow 2*\texttt{x} \\ 1 : \texttt{z} \leftarrow \texttt{t}-\texttt{y} \end{bmatrix}. \tag{9}$$

At program point 0, there is no way one can refine $\mathsf{Oct.assign}$ to provide an octagonal constraint for $\texttt{t}$. For instance, the substitution of $\texttt{x}$ by $\texttt{y}-1$ produces $\texttt{t} \leftarrow 2*\texttt{y}-2$, which cannot be represented by an octagon constraint, too. As a consequence, no constraint can be inferred on $\texttt{t}$ and hence $\texttt{z}$: $\bar{\mathbb{L}}[\![\mathcal{C}(\texttt{z} := 2*\texttt{x}-\texttt{y};)]\!](\bar{\mathsf{o}}_0) = \bar{\mathsf{o}}_0$. □

Intuitively, the precision loss in the previous example is caused by splitting "large" expressions into smaller chunks, thereby reducing the expression window seen by the atomic operations in the abstract domain, and hence limiting their ability to infer relations.

If we chose instead to analyze (9) with Poly, then both assignments at program points 0 and 1 are linear constraints that are represented exactly by this abstract domain. As a consequence, the low-level analysis, when performed on a more precise abstract domain is (weak-relatively) complete.

Discussion: Choosing the Right Abstract Domain. The previous examples suggest that we can obtain weak completeness by systematically using Poly. This is the direction taken by some analyzers for low-level code, *e.g.*, [11,20,4]. We do not advocate this approach, as Poly exhibits an exponential complexity in practice (in the number of variables). In order to overcome this issue in `Clousot`, we have chosen to not refine directly the numerical domain $\bar{\mathsf{D}}$, but to combine it with a symbolic domain Symb to propagate expressions, [1,24]. In other words the analysis is done on the refined abstract domain $\mathsf{Symb} \otimes \bar{\mathsf{D}}$. The analysis of $k : \texttt{z} \leftarrow \texttt{e2op}$ with an abstract element $\langle \bar{\mathsf{s}}, \bar{\mathsf{d}} \rangle$, first uses $\bar{\mathsf{s}}$ to refine $\texttt{e2op}$ to an expression $\texttt{e2op}^+$, then it performs the assignment over the basic numerical domain: $\bar{\mathsf{D}}.\mathsf{assign}(\bar{\mathsf{d}}, \mathsf{z}, \texttt{e2op}^+)$.

4.5 Assumptions and Assertions

We consider just the `assume` statement, the case for `assert` being similar. At source level, the PSA just passes the expression to be assumed to the underlying domain:

$$\bar{\mathbb{H}}[\![\texttt{assume e;}]\!] = \lambda\bar{\mathsf{d}}.\bar{\mathsf{D}}.\mathsf{test}(\bar{\mathsf{d}}, \mathsf{e}).$$

The compilation generates code to evaluate the condition $\texttt{e}$, and it assumes the result:

$$\mathcal{C}(\texttt{assume e;}) = \begin{bmatrix} \mathcal{C}_e(\texttt{e}) \\ k : \texttt{assume res} \end{bmatrix}. \quad (10)$$

The bytecode semantics passes the literal to the underlying abstract domain:

$$\bar{\mathbb{L}}[\![k : \texttt{assume l}]\!] = \lambda\bar{\mathsf{d}} \in \bar{\mathsf{D}}.\mathsf{test}(\bar{\mathsf{d}}, \texttt{l}).$$

The compilation schema (10), which is common to *e.g.*, the C# and Java compilers, introduces severe imprecision in analyses, as illustrated by Ex. 1 and by:

Example 4 (Precision Loss in Tests). Consider the statement $\mathcal{D} \equiv \texttt{assume } 0 \leq \texttt{x};$ to be analyzed with Oct, in the initial state $\top_{\mathsf{Oct}} = \emptyset$. Then, $\bar{\mathbb{H}}[\![\texttt{assume } 0 \leq \texttt{x};]\!](\top_{\mathsf{Oct}}) = \{-\texttt{x} \leq 0\}$. The compilation of $\mathcal{D}$ is

$$\mathcal{C}(\texttt{assume } 0 \leq \texttt{x};) = \begin{bmatrix} 0 : \texttt{res} \leftarrow 0 \leq \texttt{x} \\ 1 : \texttt{assume res} \end{bmatrix}. \quad (11)$$

At program point 0, `res` is assigned the result of evaluating the boolean condition. Since nothing is known in the input state about `x`, nothing can be concluded about the truth of $0 \leq \texttt{x}$, and hence `res` is unconstrained. As a consequence, $\bar{\mathbb{L}}[\![\mathcal{C}(\texttt{assume } 0 \leq \texttt{x};)]\!](\top_{\mathsf{Oct}}) = \top_{\mathsf{Oct}}$. □

The previous example shows that strong relative completeness does not hold. If we analyze (11) with Poly, the situation does not change, because even Poly cannot capture the relation between a variable and the truth value of an expression. Thus, if we seek weak relative completeness, we need to refine the abstract domain with either an abstract domain for tracking boolean expressions, or more generally use the symbolic abstract domain Symb introduced in Sect 4.4 to "reconstruct" larger expressions, that can then be passed to the underlying numerical abstract domain.

Whereas in Sect 4.4 the use of Symb was just an alternative w.r.t. the use of a more precise numerical domain, it becomes a *necessity* for handling boolean expressions. The use of the symbolic domain during low-level analysis requires a refinement of the transfer functions, as shown by the next example.

Example 5 (Precision Loss Induced by Compilation). Consider a slight modification of the previous example: $\mathcal{F} \equiv \texttt{assume } !(0 \leq \texttt{x});$ to be analyzed with Oct, in the entry state $\top_{\mathsf{Oct}}$. $\bar{\mathbb{H}}[\![\texttt{assume } !(0 \leq \texttt{x});]\!](\top_{\mathsf{Oct}}) = \{\texttt{x} \leq -1\}$. The compilation of $\mathcal{F}$ (*e.g.*, by C#) is

$$\mathcal{C}(\texttt{assume } !(0 \leq \texttt{x});) = \begin{bmatrix} 0 : \texttt{t} \leftarrow 0 \leq \texttt{x} \\ 1 : \texttt{res} \leftarrow \texttt{t} == 0 \\ 2 : \texttt{assume res} \end{bmatrix}. \quad (12)$$

At program point 2, the analysis of the compiled code, using the refined domain $\mathsf{Symb} \otimes \mathsf{Oct}$ infers the abstract state $\bar{\mathsf{r}} = \langle[\texttt{t} \mapsto 0 \leq \texttt{x}, \texttt{res} \mapsto \texttt{t} == 0], \top_{\mathsf{Oct}}\rangle$. Then, `res` is refined to the expression $\texttt{res}^+ \equiv (0 \leq \texttt{x}) == 0$, which cannot be generated by the syntax in Fig. 1. As a consequence, Oct.assign,

designed for the high level, does not understand $\texttt{res}^+$, and hence ignores it: $\bar{\mathbb{L}}[\![\mathcal{C}(\texttt{assume !}(0 \leq \texttt{x});)]\!](\langle \top_{\mathsf{Symb}}, \top_{\mathsf{Oct}} \rangle) = \bar{\mathsf{r}}$. □

Discussion: Refining the Transfer Functions, and Program Transformations. The example above underlines the fact that, in order to obtain weak completeness, one must also refine the transfer functions. For instance, in the example $\mathsf{Oct.assign}$ must be refined to perform the semantic preserving rewritings $(0 \leq \texttt{x}) == 0 \rightsquigarrow !(0 \leq \texttt{x}) \rightsquigarrow \texttt{x} < 0$.

In practice, a PSA designer has two choices: perform the rewriting phase online or offline. In the first case, a transfer function first rewrites the boolean expressions, *e.g.*, by applying the De Morgan laws, by rewriting $\mathsf{e} == 0$ as $!(\mathsf{e})$, etc., and then proceeds. In the second case, in a pre-processing step, a program $\mathcal{S}$ is analyzed with just Symb, all the expressions in $\mathcal{S}$ are first refined and then simplified as above, to obtain a refined program $\mathcal{S}^+$. Then, $\mathcal{S}^+$ is analyzed using $\bar{\mathsf{D}}$. In `Clousot`, we have adopted the first approach.

4.6 Conditionals

The analysis of conditional statements (i) refines the input abstract state with the guard, (ii) analyzes the two branches in the refined state, and (iii) joins the results at the exit point. Precise handling of guards is essential for a PSA.

$$\begin{aligned}&\bar{\mathbb{H}}[\![\texttt{if}(\mathsf{e})\ \{\texttt{Stm}_1\}\texttt{else}\ \{\texttt{Stm}_2\};]\!] = \\ &\qquad \lambda\bar{\mathsf{d}}.\bar{\mathbb{H}}[\![\texttt{Stm}_1]\!](\bar{\mathsf{D}}.\mathsf{test}(\bar{\mathsf{d}}, \mathsf{e})) \sqcup \bar{\mathbb{H}}[\![\texttt{Stm}_2]\!](\bar{\mathsf{D}}.\mathsf{test}(\bar{\mathsf{d}}, !(\mathsf{e}))).\end{aligned} \tag{13}$$

One possible compilation is:

$$\mathcal{C}(\texttt{if}(\mathsf{e})\ \{\texttt{Stm}_1\}\texttt{else}\ \{\texttt{Stm}_2\};) = \left[\begin{array}{rl} & \mathcal{C}_e(\mathsf{e}) \\ k: & \texttt{b} \leftarrow \texttt{res} == 0 \\ k+1: & \texttt{jmpIf b}\ t \\ & \mathcal{C}(\texttt{Stm}_1) \\ & \texttt{jmp}\ \mathit{out} \\ t: & \mathcal{C}(\texttt{Stm}_2) \\ \mathit{out}: & \texttt{nop} \end{array}\right. . \tag{14}$$

The low level analysis of (14) can be made very similar to (13), provided that some preprocessing of the bytecode is performed. The first step is to construct the control flow graph from (14), as in Fig. 3. However, that is not enough, because one wants to know that $!(\texttt{b})$ (resp. $\texttt{b}$) holds at program point $k+2$ (resp. t). Propagating such an information during a dataflow analysis is non-trivial.

A better approach is to provide another view of the code (14), in which the guard of the conditional is made explicit in the true-branch and the false-branch as `assume` statements. This is the direction we have taken in `Clousot`. In general, let $\boxed{\texttt{B}}$ the block which computes the truth value of the guard e, $\boxed{\texttt{T(e)}}$ and $\boxed{\texttt{F(e)}}$ the (compilation of the) two branches of the conditional dominated by (resp.)

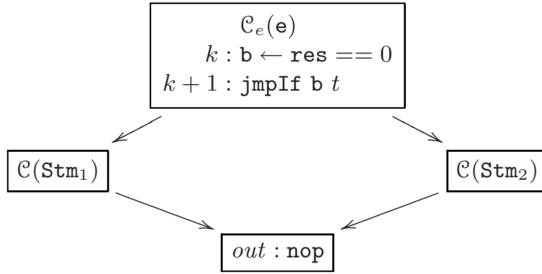

Fig. 3. The control flow graph constructed from $\mathcal{C}(\mathtt{if(e)}\ \{\mathtt{Stm}_1\}\mathtt{else}\ \{\mathtt{Stm}_2\};)$

assume b and assume !(b), and $\boxed{\mathtt{0}}$ be the exit block. Then the low level semantics can be defined as:

$$\bar{\mathbb{L}}\left[\!\!\left[\begin{array}{ccc} & \boxed{\mathtt{B}} & \\ \swarrow & & \searrow \\ \boxed{\mathtt{T(e)}} & & \boxed{\mathtt{F(e)}} \\ \searrow & & \swarrow \\ & \boxed{\mathtt{0}} & \end{array} \right]\!\!\right] = \begin{array}{l} \lambda \bar{\mathsf{r}} \in \bar{\mathsf{D}} \otimes \mathsf{Symb}. \\ \quad \text{let } \bar{\mathsf{r}}_1 = \bar{\mathbb{L}}[\![\mathtt{B}]\!](\bar{\mathsf{r}}) \text{ in} \\ \quad \text{let } \bar{\mathsf{r}}_t = \bar{\mathbb{L}}[\![\mathtt{T(e)}]\!]((\bar{\mathsf{D}} \otimes \mathsf{Symb}).\mathsf{test}(\bar{\mathsf{r}}_1, \mathsf{e})) \text{ in} \\ \quad \text{let } \bar{\mathsf{r}}_f = \bar{\mathbb{L}}[\![\mathtt{F(e)}]\!]((\bar{\mathsf{D}} \otimes \mathsf{Symb}).\mathsf{test}(\bar{\mathsf{r}}_1, !(\mathsf{e}))) \text{ in} \\ \quad \text{in } \bar{\mathsf{r}}_t \sqcup \bar{\mathsf{r}}_f \ . \end{array} \tag{15}$$

However, incompleteness can still show up if the compilation scheme is different from (14), in particular for the handling of expressions. The next example is inspired by the way the C# compiler [21], generates code for shortcutting boolean expressions.

Example 6 (Loss of Precision Induced by Compilation of Shortcut Expressions). Let $\mathcal{G}$ be the code snippet $\mathtt{if}(0 \leq \mathtt{i}\ \&\&\ \mathtt{i} < \mathtt{len})\ \{\mathtt{Stm}_1\}\ \mathtt{else}\ \{\mathtt{Stm}_2\}$. The *C#*2.0 compiler generates code that looks like the one in Fig. 4. Briefly, if one of the operands of && is false, then it jumps to line 8, which sets res to 0 . Otherwise, it sets res to 1. The two flows are then merged at program point 9, which implies that $\mathtt{res} == 0$ and $\mathtt{res} == 1$ are joined, *i.e.*, the information about the truth of the guard, $\mathtt{res} == 0 \Longleftrightarrow !(0 \leq \mathtt{i}\ \&\&\ \mathtt{i} < \mathtt{len})$ and $\mathtt{res} == 1 \Longleftrightarrow (0 \leq \mathtt{i}\ \&\&\ \mathtt{i} < \mathtt{len})$ is lost. So it cannot be further propagated in the two branches of the conditional. □

The incompleteness in the previous example can be resolved either by precisely modeling the relation between boolean variables and boolean expressions with BDDs as in [15], or by approximating the double implication with a simple implication, *e.g.*, using trace partitioning, [13]. As a consequence, the underlying abstract domain must be refined to the reduced cardinal power $\mathcal{P}(\mathtt{Lit}) \rightarrow (\bar{\mathsf{D}} \otimes \mathsf{Symb})$, so as to obtain the weak relative completeness for shortcut conditionals.

$$\mathcal{C}(\texttt{if}(0 \leq \texttt{i} \;\&\&\; \texttt{i} < \texttt{len})\ \{\texttt{Stm}_1\}\texttt{else}\{\texttt{Stm}_2\}) =$$

$$\begin{array}{lll}
0 : \texttt{t}_1 \leftarrow 0 \leq \texttt{i} & & 9 : \texttt{jmpIf res}\ k+1 \\
1 : \texttt{b}_1 \leftarrow \texttt{t}_1 == 0 & 5 : \texttt{jmpIf b}_2\ 8 & 10 : \mathcal{C}(\texttt{Stm}_2) \\
2 : \texttt{jmpIf b}_1\ 8 & 6 : \texttt{res} \leftarrow 1 & k : \texttt{jmp}\ out \\
3 : \texttt{t}_2 \leftarrow \texttt{i} < \texttt{len} & 7 : \texttt{jmp}\ 9 & k+1 : \mathcal{C}(\texttt{Stm}_1) \\
4 : \texttt{b}_2 \leftarrow \texttt{t}_2 == 0 & 8 : \texttt{res} \leftarrow 0 & out : \texttt{nop}
\end{array}$$

Fig. 4. The (simplified version of the) code generated by the *C#*2.0 compiler for the statement $\texttt{if}(0 \leq \texttt{i} \;\&\&\; \texttt{i} < \texttt{len})\ \{\texttt{Stm}_1\}\texttt{else}\{\texttt{Stm}_2\}$

4.7 Loops

The semantics of a loop is given as a least fixpoint over a suitable partial order:

$$\bar{\mathbb{H}}[\![\texttt{while(e) \{ Stm \};}]\!] = \lambda\bar{\mathsf{d}}.\ \text{let}\ \bar{\mathsf{inv}} = \mathsf{lfp}_{\bot}^{\sqsubseteq}\lambda X.\ \bar{\mathsf{d}}\sqcup\bar{\mathbb{H}}[\![\texttt{Stm}]\!](\bar{\mathsf{D}}.\mathsf{test}(X,\texttt{e}))\ \text{in}\ \bar{\mathsf{D}}.\mathsf{test}(\bar{\mathsf{inv}}, !(\texttt{e})).$$

The least fixpoint equals the limit of the increasing iterations starting from $\bot$. In general the iterations may not converge, so that a widening operator [8] is used to force convergence to a post-fixpoint. Then, a narrowing operator [8] is applied to recover some precision. An easy yet generic and useful form of narrowing is given by doing one more iteration starting from the post-fixpoint, as shown by the next example.

Example 7 (Narrowing by Re-Execution). Let $\mathcal{W} \equiv \texttt{z} := 0;\ \texttt{while(z} < 100)\ \{\ \texttt{z} := \texttt{z} + 1;\};\texttt{assert z} == 100;$ and let us analyze it with the Intv abstract domain. The fixpoint iterations produce the increasing chain of intervals $[0,0]\sqsubseteq[0,1]\sqsubseteq[0,2]\ldots\sqsubseteq[0,n]$, which is extrapolated by the standard widening on intervals to $[0,+\infty]$, so that $\mathsf{inv}^{\triangledown} = [\texttt{z} \mapsto [0,+\infty]]$ is an invariant for the loop. On the other hand, it is not precise enough to prove the assertion after the loop. By *first* re-executing the body starting from the fixpoint, one gets $[0,0]\sqcup[1,100] = [0,100]$, so that $\mathsf{inv}^{\vartriangle} = [\texttt{z} \mapsto [0,100]]$. *Then*, $\mathsf{inv}^{\vartriangle}$ intersected with the negation of the loop guard is enough to prove the assertion. □

The compilation of a while statement looks like

$$\mathcal{C}(\texttt{while(e) \{ Stm \};}) = \left[\begin{array}{l} b : \mathcal{C}_e(\texttt{e}) \\ k : \texttt{b} \leftarrow \texttt{res} == 0 \\ \quad\ \texttt{jmpIf b}\ out \\ \quad\ \mathcal{C}(\texttt{Stm}) \\ \quad\ \texttt{jmp}\ b \\ out : \texttt{nop} \end{array}\right. . \tag{16}$$

A typical analysis of the unstructured code above first detects the back edges, in order to find the program points where widening is needed. However, back edges detection is not enough to ensure relative completeness when extrapolating operators are used, as shown by the next example.

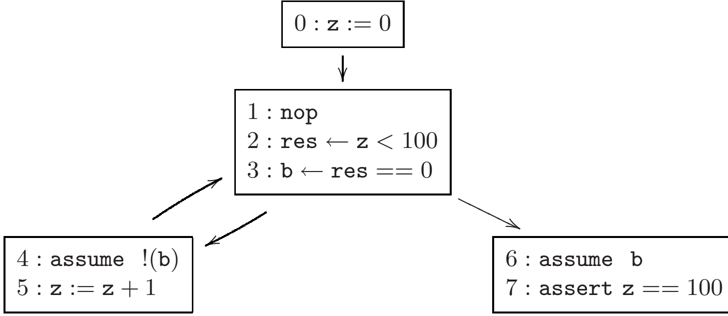

Fig. 5. The enhanced CFG graph for the three addresses compilation for the code in Ex. 7. Exact narrowing requires the knowledge that the left branch leads to a cycle.

Example 8 (Narrowing by Re-Execution, continued). The CFG graph for $\mathcal{W}$ is in Fig. 5. A standard back-edges analysis detects that the block starting at 1 is the target of a back edge, and hence the widening point. Then, we analyze the program on the domain $\mathsf{Intv} \otimes \mathsf{Symb}$, and we infer the invariant $\mathtt{z} \mapsto [0, +\infty]$ at program point 1. Now we want to refine it using the re-execution based narrowing. In the source level case, we just proceeded by induction on the structure. At the low-level, we don't know which edge leads into the loop, and which edge leads out of the loop. If we push the invariant *first* onto the left branch (*i.e.*, on program point 4), then we obtain the desired refined $\mathtt{z} \mapsto [0, 100]$, which is then pushed onto the right branch, where it is enough to prove the assertion is not violated. On the other hand, if we push the invariant *first* onto the right branch (*i.e.*, on program point 6), we obtain no invariant refinement. □

The example shows that applying standard narrowing techniques from source level analysis is tricky on low-level code, as the necessary high-level loop structures are not apparent. Symbolic expression recovery is not sufficient, as control flow is involved. Thus, to obtain relative completeness for loops, some form of loop recovery must be performed.

5 Conclusions

We have presented a series of issues faced by low-level code analyzers if their precision is to match the precision typically achieved by a source analysis. We have formalized the relation between the low-level and high-level analyses via the concepts of strong and weak relative completeness. By analysis on the program constructs, we have shown: (i) how strong relative completeness can be obtained only for trivial cases, and (ii) how weak relative completeness can be obtained by refining the underlying domain for the analysis, the transfer functions, and by pre-processing of the program. However, it turns out that the refinement step must be handled with care by the designer of the precise static analysis, in order

to avoid transforming a polynomial problem (*e.g.*, the analysis of the source program with Octagons) into an exponential one.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison Wesley, Reading (1986)
2. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library., `http://www.cs.unipr.it/ppl/`
3. Balakrishnan, G., Reps, T.W.: Analyzing memory accesses in x86 executables. In: Duesterwald, E. (ed.) CC 2004. LNCS, vol. 2985, Springer, Heidelberg (2004)
4. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for Object-Oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, Springer, Heidelberg (2006)
5. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI 2003, ACM Press, New York (2003)
6. Bourdoncle, F.: Abstract debugging of higher-order imperative languages. In: PLDI 2003, ACM Press, New York (1993)
7. Clarisó, R., Cortadella, J.: The octahedron abstract domain. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, Springer, Heidelberg (2004)
8. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977, ACM Press, New York (1977)
9. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL 1978, ACM Press, New York (1978)
10. Fähndrich, M.A., Leino, K.R.M.: Declaring and checking non-null types in an Object-Oriented language. In: OOPSLA 2003, pp. 302–312. ACM Press, New York (2003)
11. Gopan, D., Reps, T.W.: Lookahead widening. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, Springer, Heidelberg (2006)
12. Granger, P.: Improving the results of static analyses programs by local decreasing iteration. In: FSTTCS, pp. 68–79. Springer, Heidelberg (1992)
13. Handjieva, M., Tzolovski, S.: Refining static analyses by trace-based partitioning using control flow. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, Springer, Heidelberg (1998)
14. ECMA Int. Standard ECMA-355, common language infrastructure (June 2006)
15. Jeannet, B.: Representing and approximating transfer functions in abstract interpretation of hetereogeneous datatypes. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, Springer, Heidelberg (2002)
16. Leroy, X.: Bytecode verification on Java smart cards. Software - Practice and Experience (SPE) 32(4) (2002)
17. Lev-Ami, T., Manevich, R., Sagiv, S.: TVLA: A system for generating abstract interpreters. In: 18th IFIP Congress Topical Sessions, August 2004, Kluwer, Dordrecht (2004)
18. Logozzo, F.: Cibai: An abstract interpretation-based static analyzer for modular analysis and verification of Java classes. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, Springer, Heidelberg (2007)

19. Logozzo, F., Fähndrich, M.A.: Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. In: ACM SAC 2008 - OOPS, ACM Press, New York (2008)
20. Hermenegildo, M.V., Mendez, M., Navas, J.: An efficient, parametric fixpoint algorithm for analysis of Java bytecode. In: Bytecode 2007, Elsevier, Amsterdam (2007)
21. Microsoft Inc. Visual C#. `http://msdn2.microsoft.com/-us/vcsharp/`
22. Miné, A.: A new numerical abstract domain based on difference-bounds matrices. In: Danvy, O., Filinski, A. (eds.) PADO 2001. LNCS, vol. 2053, Springer, Heidelberg (2001)
23. Miné, A.: Weakly Relational Numerical Abstract Domains. PhD thesis, École Polytechnique (2004)
24. Miné, A.: Symbolic methods to enhance the precision of numerical abstract domains. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, Springer, Heidelberg (2005)
25. Palacz, K., Baker, J., Flack, C., Grothoff, C., Yamauchi, J., Vitek, H.: Engineering a common intermediate representation for Ovm framework. The Science of Computer Programming 57(3), 357–378 (2005)
26. RopasWork, Inc. Airac5, `http://ropas.snu.ac.kr/airac5/`
27. Rossignoli, S., Spoto, F.: Detecting non-cyclicity by abstract compilation into boolean functions. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, Springer, Heidelberg (2005)
28. Venet, A., Brat, G.P.: Precise and efficient static array bound checking for large embedded c programs. In: PLDI 2004, ACM Press, New York (2004)