A Complexity Preserving Transformation from Jinja Bytecode to Rewrite Systems

master thesis in computer science

by

Michael Schaper

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements for the degree of Master of Science

supervisor: Assoc. Prof. Dr. Georg Moser,
Institute of Computer Science

Innsbruck, 15 May 2014

Master Thesis

A Complexity Preserving Transformation from Jinja Bytecode to Rewrite Systems

Michael Schaper (c7031025)
michael.schaper@student.uibk.ac.at

15 May 2014

Supervisor: Assoc. Prof. Dr. Georg Moser

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.
Die vorliegende Arbeit wurde bisher in gleicher oder ähnlicher Form noch nicht als Magister-/Master-/Diplomarbeit/Dissertation eingereicht.

Datum	Unterschrift

Abstract

In this thesis we revisit known transformations from object-oriented bytecode programs to rewrite systems from the viewpoint of runtime complexity. We analyse *Jinja bytecode* (JBC for short), which exhibits core features of well–known object-oriented programming languages but provides a formal semantics. Using standard techniques from static program analysis we define two alternative representations of JBC executions. First, we provide a *graph-based* abstraction of bytecode states and establish a *Galois insertion* between the abstract domain and sets of bytecode states. We define an abstract semantics from which we obtain a finite representation of JBC executions as *computation graphs*. We show that this representation is correct and that the computation graph of a program is always computable and finite. Second, we provide a *term-based* abstraction of bytecode states that we obtain from the nodes of the computation graph. From the edge relation of the computation graph we obtain a finite representation of JBC executions as *constrained term rewrite systems*. We show that the transformation is *complexity preserving*. That is, an upper bound on the runtime complexity of the resulting constrained term rewrite system implies an upper bound on the runtime complexity of the bytecode program. Moreover, we show that the transformation is *non-termination preserving*. We restrict to *non-recursive* methods and make use of existing heap shape analysis to approximate acyclicity and reachability. The transformation has been implemented in the prototype JaT.

Acknowledgments

Foremost I express my gratitude to my supervisor Georg Moser, who dedicated much time and effort to guide me throughout this master project. In particular, I am thankful for introducing me to an interesting area of research. I thank the Computational Logic group for providing a pleasant environment during my studies and especially Martin Avanzini for helpful discussions. I thank all my friends and colleagues for healthy diversions. Finally, I thank my family for supporting me during my whole studies.

Contents

1. Introduction

Automated complexity analysis of computer programs is concerned with automatically inferring upper bounds for suitable cost measures, such as the maximal number of computation steps or the maximal size of the memory consumption. Such upper bounds are usually represented as cost functions in terms of the input. Establishing upper bounds of computer programs is undecidable in general: The decision problem that Turing machine M runs in polynomial time is Σ_2^0-*complete* in the arithmetical hierarchy [29]. However, suitable techniques from static program analysis allow safe approximations. In this thesis we study the automatic runtime complexity analysis of Jinja bytecode, an object-oriented bytecode language, by means of a transformation to rewrite systems.

Object-oriented languages like Java are very popular[1] and static analysis techniques of such languages have been studied thoroughly, for example [21, 44, 45]. Though, the automatic complexity analysis of imperative and object-oriented languages is still a big challenge and an active area of research. Several methods have been proposed in recent years [2, 6, 22, 49].

In this work we study a transformational approach from object-oriented bytecode programs to *constrained term rewrite systems* (cTRSs for short). Here, cTRSs are defined as an extension of *term rewrite systems* (TRSs) that incorporate the theory of Presburger arithmetic to express integer and Boolean operations naturally. Furthermore, constraints are added to rules such that a rule is only applicable if its constraint is satisfied.

TRSs (and its derivatives) have been successfully applied before for proving termination of computer programs: In [41] term-based abstractions are used to provide termination analyses of programs in a functional language. In [16, 17] a transformation from `C` like programs with integer valued variables to cTRSs is presented: Terms correspond to program states at control flow points and rules represent state transformations between control flow points. We want to illustrate this approach with following example.

Example 1.1. Figure 1.1 illustrates a program with a single `while` loop. Rule rem_1 represents the assignment of 3 to n and rule rem_2 represents the `while` loop. A constraint is used to express the loop condition. The resulting cTRS is non-terminating since the fact that variable n is a constant of value 3 is not available in the second rule.

As already indicated in [16, 17] additional (standard) analyses to find invariants over integer valued variables improve the termination analysis. This approach was extended in [11, 39] to prove termination of Java programs including user-defined data structures. Here, an abstract state represents a set of

[1] `http://www.tiobe.com/index.php/tiobe_index`

```
int n = 3;
while(m > n){
  m = m - n;
}
```

$$\mathsf{rem}_1(m,n) \to \mathsf{rem}_2(m,3)$$
$$\mathsf{rem}_2(m,n) \to \mathsf{rem}_2(m-n,n)\,[\![m > n]\!]$$

Figure 1.1.: Simple transformation to a cTRS.

program states. A finite relation on abstract states is obtained by *symbolically evaluating* the bytecode instructions with abstract states, and suitably merging them. This relation is then transformed into rewrite rules such that rewrite steps mimic program steps.

The *runtime complexity* of TRSs forms an invariant cost model [7] and several methods have been developed in recent years to compute upper bounds of TRSs automatically [8, 35, 38]. Many of these techniques have been implemented in the *Tyrolean Complexity Tool*, also known as T$_\mathsf{C}$T[2].

This motivates to investigate aforementioned approaches in the context of automatic complexity analysis of computer programs. We provide an abstraction from bytecode programs to cTRSs by means of standard techniques from static program analysis, in particular *abstract interpretations*. Abstract interpretations allow to relate concrete and abstract domains formally and provide sufficient conditions to safely approximate the concrete domain by the abstract domain. Similar to the approach presented in [11, 39], the proposed transformation encompasses two stages. The first stage provides a finite representation of all execution paths of program P through a graph, termed *computation graph*. The nodes of the computation graph are *graph-based* abstractions of JVM states and the graph is formed by *symbolic evaluation*, essentially *joining* states with equal program location. We establish a *Galois insertion* between the abstract domain and sets of JVM states and show that the construction of the computation graph of P is correct, computable and finite. In the second stage, we encode the (finite) computation graph as cTRS, where constraints are used to express relations on program variables. We suitably transform abstract states to terms and therefore obtain a *term-based* abstraction of JVM states. Our formalism of abstract states is not rich enough to approximate acyclicity and reachability precisely, but asks for a combination with existing heap shape analyses such as [21, 42, 44]. We show that the proposed transformation from bytecode programs to cTRSs is *complexity preserving*, which allows to infer an upper bound on the runtime complexity of P by analysing the runtime complexity of the cTRSs obtained by the proposed transformation. As a corollary we obtain that the transformation is also *non-termination preserving*.

Example 1.2. Figure 1.2 illustrates the cTRS that is obtained via our transformation of the program from Figure 1.1. The resulting cTRS contains more rules as we perform our transformation on the bytecode of the program. Our transformation is capable to provide the invariant $n = 3$ for the loop.

[2] `http://cl-informatik.uibk.ac.at/software/tct`

$$
\begin{aligned}
\mathsf{rem}_I(\mathsf{Rem}, m, \mathsf{null}) &\to \mathsf{rem}_A(\mathsf{Rem}, m, 3)\\
\mathsf{rem}_A(\mathsf{Rem}, m, 3) &\to \mathsf{rem}_B(b, \mathsf{Rem}, m, 3)[\![b \equiv m > 3]\!]\\
\mathsf{rem}_B(\mathsf{true}, \mathsf{Rem}, m, 3) &\to \mathsf{rem}_C(\mathsf{true}, \mathsf{Rem}, m, 3)\\
\mathsf{rem}_C(\mathsf{true}, \mathsf{Rem}, m, 3) &\to \mathsf{rem}_A(\mathsf{Rem}, m', 3)[\![m' = m - 3]\!]\\
\mathsf{rem}_B(\mathsf{false}, \mathsf{Rem}, m, 3) &\to \mathsf{rem}_D(\mathsf{false}, m, 3)\\
\mathsf{rem}_D(\mathsf{false}, \mathsf{Rem}, m, 3) &\to \mathsf{rem}_E(\mathsf{Rem}, m, 3)
\end{aligned}
$$

Figure 1.2.: The cTRS of `rem`.

Although our transformation is capable of generating invariants over integer valued variables, standard abstract integer domains such as the octagon or polyhedra domain usually provide more sophisticated ones.

Operations on objects are captured by the *term-based* abstraction of our transformation. We think that term-based abstractions are adequate to analyse programs with composited data structures. Consider Figure 1.3 which illustrates one of the motivating examples in [39]. In [2, 45] objects are abstracted

```
class IntList{
  IntList next;
  int value;
}

class Tree{
  Tree left;
  Tree right;
  int value;
}

class TreeList{
  TreeList next;
  Tree value;
}
```

```
class Flatten {
  IntList flatten(TreeList list){
    TreeList cur = list;
    IntList result = null;
    while (cur != null){
      Tree tree = cur.value;
      if (tree != null) {
        IntList oldIntList = result;
        result = new IntList();
        result.value = tree.value;
        result.next = oldIntList;
        TreeList oldCur = cur;
        cur = new TreeList();
        cur.value = tree.left;
        cur.next = oldCur;
        oldCur.value = tree.right;
      } else {
        cur = cur.next;
      }
    }
    return result;
  }
}
```

Figure 1.3.: The `flatten` program.

to their maximal *path-length*. Suppose the parameter is initially bounded to an acyclic list. The approaches presented in [2, 45] based on path-length fail to provide an upper bound or show termination. Whereas our domain is rich

enough to capture the behaviour of the program and our implementation is able to infer a linear bound for the program fully automatically. We discuss this example in more detail in Section 8.2.

The remainder of this thesis is structured as follows: Chapter 2 introduces basic definitions and provides an overview over techniques from static program analysis that are used in sequent chapters. In Chapter 3 bytecode programs are formally introduced. Afterwards, program states and bytecode semantics are introduced in Chapter 4. The abstract state domain is introduced in Chapter 5. In Chapter 6 we present the transformation to cTRSs and provide our main result. In Chapter 7 and Chapter 8 we discuss related work and implementation details, respectively. We conclude in Chapter 9.

2. Preliminaries

Let f be a mapping from A to B, denoted $f\colon A \to B$, then $\mathsf{dom}(f) = \{x \mid f(x) \in B\}$ and $\mathsf{rg}(f) = \{f(x) \mid x \in A\}$. Let $a \in \mathsf{dom}(f)$, we define:

$$f\{a \mapsto v\}(x) := \begin{cases} v & \text{if } x = a \\ f(x) & \text{otherwise .} \end{cases}$$

We compare partial functions with *Kleene equality*: Two partial functions $f\colon \mathbb{N} \to \mathbb{N}$ and $g\colon \mathbb{N} \to \mathbb{N}$ are equal, denoted $f =_{\mathsf{k}} g$, if for all $n \in \mathbb{N}$ either $f(n)$ and $g(n)$ are defined and $f(n) = g(n)$ or $f(n)$ and $g(n)$ are not defined.

We usually use square brackets to denote a list. Furthermore, (:) denotes the cons operator, and (@) is used to denote the concatenation of two lists.

Definition 2.1. A *directed graph* $G = (V_G, Succ_G, L_G)$ over the set $\mathcal{L}$ of *labels* is a structure such that V_G is a finite set, the *nodes* or *vertices*, $Succ_G\colon V_G \to V_G^*$ is a mapping that associates a node u with an (ordered) sequence of nodes, called the *successors* of u. Note that the sequence of successors of u may be empty: $Succ_G(u) = []$. Finally $L_G\colon V_G \to \mathcal{L}$ is a mapping that associates each node u with its *label* $L_G(u)$. Let u, v be nodes in G such that $v \in Succ_G(u)$, then there is an *edge* from u to v in G; the edge from u to v is denoted as $u \to v$.

Definition 2.2. A structure $G = (V_G, Succ_G, L_G, E_G)$ is called *directed graph with edge labels* if $(V_G, Succ_G, L_G)$ is a directed graph over the set $\mathcal{L}$ and $E_G\colon V_G \times V_G \to \mathcal{L}$ is a mapping that associates each edge e with its *label* $E_G(e)$. Edges in G are denoted as $u \xrightarrow{\ell} v$, where $E_G(u \to v) = l$ and $u, v \in V_G$. We often write $u \to v$ if the label is either not important or is clear from context.

If not mentioned otherwise, in the following a *graph* is a directed graph with edge labels. Usually, nodes in a graph are denoted by $u, v, \ldots$ possibly followed by subscripts. We drop the reference to the graph G from V_G, $Succ_G$, and L_G, ie., we write $G = (V, Succ, L)$ if no confusion can arise from this. Furthermore, we also write $u \in G$ instead of $u \in V$.

Let $G = (V, Succ, L)$ be a graph and let $u \in G$. Consider $Succ(u) = [u_1, \ldots, u_k]$. We call u_i $(1 \leqslant i \leqslant k)$ the *i-th successor* of u (denoted as $u \rightharpoonup_G^i u_i$). If $u \rightharpoonup_G^i v$ for some i, then we simply write $u \rightharpoonup_G v$. A node v is called *reachable* from u if $u \rightharpoonup_G^* v$, where $\rightharpoonup_G^*$ denotes the reflexive and transitive closure of $\rightharpoonup_G$. We write $\rightharpoonup_G^+$ for $\rightharpoonup_G \cdot \rightharpoonup_G^*$. A graph G is *acyclic* if $u \rightharpoonup_G^+ v$ implies $u \neq v$. We write $G \restriction u$ for the subgraph of G reachable from u.

2.1. Lattice Theory

In static program analysis often lattices are used to represent the underlying property domain. In this section, we recapitulate important concepts of the lattice theory following the presentation of [37].

Definition 2.3. A *partially ordered set*, or *poset*, is a set L equipped with a *partial ordering* $\sqsubseteq$. A partial ordering is a relation $\sqsubseteq$ that is reflexive, transitive and anti-symmetric. Let $l \in L$ and Y be a subset of L. Then l is an *upper bound* (*lower bound*) of Y if for all $l' \in Y$ the relation $l' \sqsubseteq l$ ($l \sqsubseteq l'$) holds. An upper bound l_0 is a *least upper bound* (*greatest lower bound*) of Y if $l_0 \sqsubseteq l$ ($l \sqsubseteq l_0$) holds for all upper bounds (lower bounds) l of Y. We use $\bigsqcup Y$ ($\bigsqcap Y$) to denote the least upper bound (greatest lower bound) of Y. A subset Y of L is a *chain* if for all $l_1, l_2 \in Y$ the relation $l_1 \sqsubseteq l_2$ or $l_2 \sqsubseteq l_1$ holds. A partially ordered set satisfies the *ascending chain condition* (*descending chain condition*) iff any infinite sequence $l_0 \sqsubseteq l_1 \sqsubseteq \cdots \sqsubseteq l_n \sqsubseteq \cdots$ ($l_0 \sqsupseteq l_1 \sqsupseteq \cdots \sqsupseteq l_n \sqsupseteq \cdots$) is not strictly increasing (decreasing).

Definition 2.4. A *complete lattice* $L = (L, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$ is a partially ordered set $(L, \sqsubseteq)$ such that any subset Y has a least upper bound ($\bigsqcup Y$) and a greatest lower bound ($\bigsqcap Y$). Here, $\bot = \bigsqcup \varnothing = \bigsqcap L$ is the *least element* and $\top = \bigsqcap \varnothing = \bigsqcup L$ is the *greatest element*. We write $l_1 \sqcup l_2$ ($l_1 \sqcap l_2$) instead of $\bigsqcup\{l_1, l_2\}$ ($\bigsqcap\{l_1, l_2\}$).

We use the convention that we identify the lattice itself with its underlying domain, for example L for some lattice $(L, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$.

Example 2.5. Let S be an arbitrary set and $\subseteq$ denote the subset relation. Then $\mathcal{P}(S) := (\mathcal{P}(S), \subseteq, \bigcup, \bigcap, \varnothing, S)$ is a complete lattice.

Another simple, but interesting, example is the *sign lattice*, which is used to analyse the sign of integer valued variables in a program.

Example 2.6. Let $\mathsf{Sign} = \{0, -, +, -0, 0+, \bot, \top\}$ be the domain of signs. The elements $-, +, -0$ and $0+$ represent the negative, positive, non-positive and non-negative domain of integers respectively. The complete lattice $\mathsf{Sign} := (\mathsf{Sign}, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$ is fully defined via the Hasse diagram depicted in Figure 2.1. We have $s1 \sqsubseteq s_2$ iff there exists a path in the diagram from s_1 to s_2 in an upward manner.

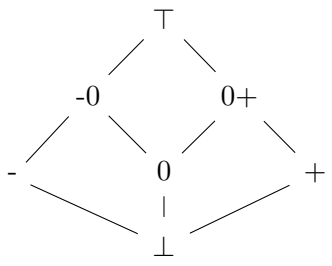

Figure 2.1.: The Hasse diagram of the complete lattice Sign.

One may observe from the previous example that $\sqsubseteq$ defines a notion of precision on the sign domain. The bottom element contains no information at all. The second level consisting of $-, 0,$ and $+$ is the most precise one. The

third level looses the information about a value being precisely zero, whereas no information about the sign of a value can be inferred at the top at all.

This observation leads to the following motivation for the usage of lattices in static program analysis. In case of the sign analysis the Sign domain represents all elements of the property domain. The order $\sqsubseteq$ induces precision along the property domain. The operation $\bigsqcup$ is used to abstract information from bottom to top when a program location is visited multiple times. Finally, $\bot$ serves as unit value with respect to $\bigsqcup$, whereas $\top$ provides a way to ensure termination of the analysis.

The former description indicates that $\bigsqcap$ is not used in the sign analysis. Indeed, often one only needs either $\bigsqcup$ or $\bigsqcap$. We see later that the choice depends on the actual analysis. Moreover, one could express $\bigsqcap$ via $\bigsqcup$ and vice versa. In the sequent, when defining lattices, we restrict to define only the operations needed for our analysis.

Definition 2.7. Let $f\colon L \to L$ be a monotone function on a complete lattice $L = (L, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$. A *fixed point* of f is an element $l \in L$ such that $f(l) = l$. A fixed point l_0 is the *least fixed point* (*greatest fixed point*) if $l_0 \sqsubseteq l$ ($l \sqsubseteq l_0$) holds for all fixed points l of L.

Let $(L, \sqsubseteq)$ be a poset and f be a monotone function. Tarski's fixed point theorem [48] states that the set of all fixed points of f forms a complete lattice with respect to $\sqsubseteq$. Therefore, f has a least fixed point and a greatest fixed point. Moreover, if $(L, \sqsubseteq)$ admits the ascending chain condition (descending chain condition) then there exists a $n \in \mathbb{N}$ such that $f^n(\bot) = f^{n+1}(\bot)$ ($f^n(\top) = f^{n+1}(\top)$) and $f^n(\bot)$ is the least fixed point ($f^n(\top)$ is the greatest fixed point). Here f^n is the iterative application of f, ie., $f^0(x) = x$ and $f^{n+1}(x) = f(f^n(x))$.

In the next section we show that a data flow problem is a set of constraints over the property domain. Transfer functions are used to describe the flow of information and fixed points are solutions to the set of constraints. The above observations suggests that one can obtain (under some conditions) the most precise solution by iterative application of the transfer functions.

2.2. Static Program Analysis

Static program analysis is a concept to generate *safe* and *computable approximations* of (dynamic) program properties without executing the actual program. Here, safe means that the desired property holds for all possible program runs. Since programs may have infinitely many program runs, approximations are necessary to provide finite analyses.

In this section we give an overview of important concepts of static program analysis that will be used in sequent chapters. We first describe the classic approach to data flow analysis. Then, we recapitulate how Galois connections can be used to construct instances of data flow analyses.

2.2.1. Data Flow Analysis

In data flow analysis programs are usually conceived as graphs, termed *control flow graphs* (CFGs for short). Nodes represent (a sequence of) instructions, whereas edges represent how information flows from one node to another. A data flow problem is then defined by the underlying domain, an initial value, and monotone functions representing the transformation of information along the program flow. This gives rise to a set of constraints, where one is interested in the most precise solution that satisfies all constraints.

We make this more formally by recapitulating the notion of *generalized monotone frameworks* as presented in [37]. Let *Label* define a set of labels. Then, an instance of a generalised monotone framework consists of:

- a complete lattice L,
- a finite flow $F \subseteq \mathit{Label} \times \mathit{Label}$,
- a finite set of extremal labels $E \subseteq \mathit{Label}$,
- an extremal value $e \in L$, and
- a mapping from labels l to monotone transfer functions $f_l \colon L \to L$.

The lattice L defines the property domain. Labels identify nodes uniquely. The flow F defines directed edges between labels and captures the sequence of statements to consider. The extremal value represents the initial property of the analysis, whereas extremal labels represent entry and exit labels of the flow. Transfer functions specify how one property is transformed along the program flow into another property.

Let l, l' be labels. An analysis A associates labels with properties in L and generalises to a set of constraints, when considering

(i) $A_\circ(l) \sqsupseteq e$, if $l \in E$ is an extremal label,

(ii) $A_\circ(l) \sqsupseteq \mathit{join}\ A_\bullet(l')$, for all edges from l' to l in F, and

(iii) $A_\bullet(l) \sqsupseteq f_l(A_\circ)$, where f_l is the transfer function associated with l.

Intuitively, $A_\circ(l)$ represents the property of l before the execution of the statements associated to l. Conversely, $A_\bullet(l)$ represents the property of l after execution of the associated statements. The combination function *join* is either $\bigsqcup$ or $\bigsqcap$ and depends on the actual analysis.

For the remainder of this chapter we will use the While programming language to exemplify concepts of data flow analysis. While consists of assignments, `while` loops and arithmetic operations over the integers. States in While are defined by the program environment, ie., a mapping $\sigma \colon V \to \mathbb{Z}$ from program variables to integers. We denote the domain of states by State.

Example 2.8. Consider the following While program, where all statements are already uniquely labelled.

$$\texttt{[i:=0;]}^1\ \texttt{[n:=100;]}^2\ \texttt{while [i<=n;]}^3\ \texttt{do [i:=i+1;]}^4$$

We keep the property domain L, the transfer functions f_l and the combination function $join \in \{\sqcup, \sqcap\}$ abstract and define a forward analysis A with extremal label 1 and extremal value e. We obtain:

$$F = \{(1,2),(2,3),(3,4),(4,3)\}$$
$$E = \{1\}$$

$$\begin{array}{ll} A_\circ(1) \sqsupseteq e & A_\circ(3) \sqsupseteq A_\bullet(2)\ join\ A_\bullet(4) \\ A_\bullet(1) \sqsupseteq f_1(A_\circ(1)) & A_\bullet(3) \sqsupseteq f_3(A_\circ(3)) \\ A_\circ(2) \sqsupseteq A_\bullet(1) & A_\circ(4) \sqsupseteq A_\bullet(3) \\ A_\bullet(2) \sqsupseteq f_2(A_\circ(2)) & A_\bullet(4) \sqsupseteq f_4(A_\circ(1)) \end{array}$$

A solution can be obtained by *chaotic iteration*. First of all, for all labels l, $A_\circ(l)$ and $A_\bullet(l)$ are initialized with the unit value. Then, left hand sides are non-deterministically updated if the constraint is not fulfilled. If all f_l are monotone, then there exists local fixed points for every $A_\circ(l), A_\bullet(l)$. If every constraint admits a local fixed point, then the set of constraints admits a global fixed point. In notation, $(A_\circ, A_\bullet) \models A^{\sqsupseteq}$. If $(L, \sqsubseteq)$ satisfies the ascending chain condition then this strategy always terminates successfully.

Next, we identify several useful characterisations of data flow analysis that will be used in sequent chapters frequently.

May/Must. A *may* analysis identifies properties that are satisfied by all paths. The desired solution is the *least solution* that satisfies the constraints and the combination operator is usually $\bigsqcup$. On the other hand, a *must* analysis identifies properties that are satisfied by at least one path. The desired solution is the *greatest solution* that satisfies the constraints and the combination operator is usually $\bigsqcap$.

Forward Flow/Backward Flow. A *forward flow* captures the standard execution order of statements. A *backward flow* is obtained by reversing the direction of the edges of the forward flow.

Flow-Sensitive/Flow-Insensitive. In a *flow-sensitive* analysis the order of the statements matters, whereas in a *flow-insensitive* analysis the order of statements does not matter.

Intraprocedural/Interprocedural. *Intraprocedural* analyses are concerned with the inspection of a single method. Method calls are usually abstracted using the top element. *Interprocedural* analyses take method calls into account. Special techniques are necessary to obtain a solution over valid paths, ie., if the information flows from a call to a procedure it should flow back from the same procedure to the call site.

Context Sensitive/Context Insensitive. In a *context sensitive* analysis different call sites are distinguished by introducing contexts. Therefore, it may be necessary to analyse a method multiple times. In a *context insensitive* setting the information of all call sites are combined, and the same result is returned back to all call sites.

We conclude this subsection with a final example that serves as motivation for the abstract interpretation framework below.

Example 2.9. We define the *sets of states* analysis, denoted SS. The sets of states analysis approximates how sets of states are transformed into sets of states. In particular, we obtain an approximation of the states reachable from a given start state if the start state is an element of the extremal value.

Let State represent all states of While, $e \subseteq \mathsf{State}$, and $\mathcal{P}(\mathsf{State}) := (\mathcal{P}(\mathsf{State}), \subseteq, \bigcup, \bigcap, \varnothing, \mathsf{State})$ be a complete lattice. Furthermore, let the transfer function of SS be defined as the component-wise application of the single-step semantics of While.

Now consider a program P and the set of constraints $\mathsf{SS}^{\supseteq}$ induced by the control flow graph of P. Suppose $(\mathsf{SS}_\circ, \mathsf{SS}_\bullet) \models \mathsf{SS}^{\supseteq}$ and state t can be derived from a start state s in P. Then $s \in e$ implies that t is an element of the union of all sets of $\mathsf{SS}_\bullet(l)$. The claim follows straightforward by an inductive argument over the derivation.

Note that $(\mathcal{P}(\mathsf{State}), \subseteq)$ does not satisfy the ascending chain condition and $\mathsf{SS}_\bullet$ is in general not computable. In particular, it is undecidable if some state t is an element of $\mathsf{SS}_\bullet(l)$ for some label l. However, in the next subsection we show how $\mathsf{SS}_\bullet$ can be *safely approximated* by means of abstract interpretation.

2.2.2. Abstract Interpretation

Abstract interpretation is a generic framework for static program analysis first introduced in the seminal paper of P. Cousot and R. Cousot [14]. The basic idea is that one relates concrete domains to abstract domains. Abstract domains are designed such that the analysis is computable and program properties can be inferred directly. If the abstract domain safely approximates the concrete domain, then properties of the concrete domain can be deduced from properties of the abstract domain.

For the complexity analysis of programs we have to consider all possible program runs for a set of initial states. Hence, the concrete domain should capture all possible traces. In this scenario one usually talks about the *concrete semantics* or *collecting semantics* of the programming language. We already have studied a possible formalization in Example 2.9, namely SS. The formalization of the information flow in the abstract domain is usually termed *abstract semantics*. One method for describing such relationship between concrete and abstract domain are Galois connections.

Definition 2.10. A *Galois connection* is a quadruple (L, α, γ, M) such that L and M are complete lattices, $\alpha\colon L \to M$ and $\gamma\colon M \to L$ are monotone functions, satisfying

$$\forall l \in L\colon l \sqsubseteq \gamma(\alpha(l)) \text{ and } \forall m \in M\colon m \sqsupseteq \alpha(\gamma(m)) \,.$$

The functions α and γ are usually termed *abstraction function* and *concretisation function*, respectively.

Here L defines the property domain of the concrete domain and M defines the property domain of the abstract domain. The first condition implies that we may lose precision, if we abstract an element of the concrete domain and then concretise the result again. The second condition implies that it is safe, if we concretise an element of the abstract domain and then abstract the result again. The following lemma provides an intuitive way for relating the concrete domain to the abstract domain.

Lemma 2.11. *If (L, α, γ, M) is a Galois connection then $\alpha(l) \sqsubseteq m \Leftrightarrow l \sqsubseteq \gamma(m)$.*

Proof. Follows directly from the properties of α and γ. □

A more strict variant of the Galois connection that is often useful in practice, is the concept of Galois insertion.

Definition 2.12. A *Galois insertion* is a quadruple (L, α, γ, M) such that L and M are complete lattices, $\alpha\colon L \to M$ and $\gamma\colon M \to L$ are monotone functions, satisfying

$$\forall l \in L\colon l \sqsubseteq \gamma(\alpha(l)) \text{ and } \forall m \in M\colon m = \alpha(\gamma(m)) \ .$$

In particular γ is injective, or equivalently α is surjective.

Obviously every Galois insertion is a Galois connection. A Galois insertion ensures that M does not contain superfluous elements, ie., there are no two elements $m_1, m_2 \in M$ such that $m_1 \sqsubset m_2$ and $\gamma(m_1) = \gamma(m_2)$. In practice this means that there exists no abstract element that is more precise than m_1 and at the same time represents the same domain in the concrete property space. A Galois insertion can be obtained from a Galois connection by means of a reduction operator.

Definition 2.13. Let (L, α, γ, M) be a Galois connection. The *reduction operator* $\varsigma\colon M \to M$ returns the most precise element representing the same concrete domain:

$$\varsigma(m) = \bigsqcap \{m' \mid \gamma(m) = \gamma(m')\} \ .$$

Then $(L, \alpha, \gamma, \varsigma^*(M))$ is a Galois insertion, where ς^* is the set extension of ς.

Under certain conditions Galois connections can be constructed in a simple way. In particular if L is a powerset domain.

Definition 2.14. Let $\mathcal{P}(V)$ and M be complete lattices. Furthermore, let $V' \subseteq V$, and $\beta\colon V \to M$ be a function mapping values of V to M. We call β the *representation function*. Then $(\mathcal{P}(V), \alpha, \gamma, M)$ is a Galois connection with

$$\alpha(V') = \bigsqcup \{\beta(v) \mid v \in V'\} \text{ ,and}$$
$$\gamma(m) = \{v \in V \mid \beta(v) \sqsubseteq m\} \ .$$

If M is also a powerset domain, for example $\mathcal{P}(D)$, then it is enough to provide an *extraction function* $\eta\colon V \to D$. Then $\bigsqcup$ is $\bigcup$ and $\beta(v) = \{\eta(v)\}$ in the previous definition of α and γ.

Example 2.15. Recall Example 2.6, where we introduced the Sign lattice. To analyse the sign of multiple variables in While programs we extend Sign to $\mathsf{State}_{\mathsf{Sign}}$. The domain is $\mathsf{State}_{\mathsf{Sign}} \colon V \to \mathsf{Sign}$, and the partial order and the join operation is defined component-wise for some program environment σ. We omit the details here. We define $\mathsf{sign} \colon \mathbb{Z} \to \mathsf{Sign}$ as follows:

$$\mathsf{sign}(z) := \begin{cases} - & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ + & \text{if } z > 0 \end{cases}$$

Furthermore, let $\beta_{\mathsf{sign}} \colon \mathsf{State} \to \mathsf{State}_{\mathsf{Sign}}$ be defined as $\beta(x) := \mathsf{sign}(\sigma(x))$. Then $\mathsf{State}_{\mathsf{Sign}} := (\mathcal{P}(\mathsf{State}), \alpha_{\mathsf{sign}}, \gamma_{\mathsf{sign}}, \mathsf{State}_{\mathsf{Sign}})$ is a Galois connection. It is easy to see that γ_{sign} is injective as all elements of Sign represents different sets of numbers. Hence $\mathsf{State}_{\mathsf{Sign}}$ is also a Galois insertion. Moreover, $(\mathsf{State}_{\mathsf{Sign}}, \sqsubseteq)$ satisfies the ascending chain condition.

We now motivate abstract interpretations by means of the monotone framework and Galois connections. Suppose A is an instance of the generalised monotone framework with complete lattice L, extremal value e and transfer functions f_l. Furthermore (L, α, γ, M) is a Galois connection. We construct an instance B with complete lattice M. We require that

- the extremal value $e^\sharp$ of B satisfies $e^\sharp \sqsupseteq \alpha(e)$, and
- the monotone transfer functions $f_l^\sharp$ of B satisfy $f_l^\sharp \sqsupseteq \alpha \circ f_l \circ \gamma$.

Then $(B_\circ, B_\bullet \models B^{\sqsupseteq})$ implies $(\gamma \circ B_\circ, \gamma \circ B_\bullet \models A^{\sqsupseteq})$, ie., the solution in B is an upper approximation to the solution in A. If B satisfies the above mentioned requirements, we say that the abstraction is *correct*. Sometimes it is convenient to restate the requirements. It holds that: (i) $e^\sharp \sqsupseteq \alpha(e) \Leftrightarrow e \sqsubseteq \gamma(e^\sharp)$, and (ii) $f_l^\sharp \sqsupseteq \alpha \circ f_l \circ \gamma \Leftrightarrow f_l \sqsubseteq \gamma \circ f_l^\sharp \circ \alpha$. The former relation ensures that the initial concrete domain is an instance of the concretisation of the initial abstract domain. The latter relation ensures that a step in the concrete domain is safely approximated by a step in the abstract domain. The main observation follows from an inductive argument. For a detailed proof, the interested reader is referred to [37].

Example 2.16. We define the abstract semantics for $\mathsf{State}_{\mathsf{Sign}}$ informally as follows: For an assignment, we first (abstractly) evaluate the right hand side. Numbers are abstracted to their sign and arithmetic operations are the expected operations on signs. The variable is then updated with the resulting expression. Conditions do not affect the state, and therefore are mapped to the identity function. The following constraints illustrates the result of the sign analysis of our While program of Example 2.16. It is easy to check that the analysis is correct, ie., $(\gamma_{\mathsf{sign}} \circ \mathsf{State}_{\mathsf{Sign}_\circ}, \gamma_{\mathsf{sign}} \circ \mathsf{State}_{\mathsf{Sign}_\bullet} \models \mathsf{SS}^{\sqsupseteq})$

$$
\begin{aligned}
A_\circ(1) &:= \{i \to \bot, n \to \bot\} \sqsupseteq \{i \to \bot, n \to \bot\} \\
A_\bullet(1) &:= \{i \to 0, n \to \bot\} \sqsupseteq f_1(A_\circ(1)) \\
A_\circ(2) &:= \{i \to 0, n \to \bot\} \sqsupseteq A_\bullet(1) \\
A_\bullet(2) &:= \{i \to 0, n \to +\} \sqsupseteq f_2(A_\circ(2))
\end{aligned}
$$

$$
\begin{aligned}
A_\circ(3) &:= \{i \to 0+, n \to +\} \sqsupseteq A_\bullet(2) \sqcup A_\bullet(4) \\
A_\bullet(3) &:= \{i \to 0+, n \to +\} \sqsupseteq f_3(A_\circ(3)) \\
A_\circ(4) &:= \{i \to 0+, n \to +\} \sqsupseteq A_\bullet(3) \\
A_\bullet(4) &:= \{i \to 0+, n \to +\} \sqsupseteq f_4(A_\circ(1))
\end{aligned}
$$

2.3. Complexity Preserving Abstraction

We express the complexity of a program in terms of the maximal derivation height with respect to its input size. An abstract relation should safely approximate the program state relation in a similar way as presented above. Additionally we employ a size condition on the start states.

Let A be a set and $\to \subseteq A \times A$ be a binary relation on A. We write $a_1 \to a_2$ instead of $(a_1, a_2) \in \to$. The maximal *derivation height* is defined by

$$\mathsf{dh}(a, \to) =_\mathsf{k} \mathsf{max}\{n \mid \exists a'.a \to^n a'\} \ .$$

Let $|\cdot| \colon A \to \mathbb{N}$ denote a suitable *size measure* on A. We define the *complexity function* as the maximal derivation height with respect to input size and input elements:

$$\mathsf{cc}(n, S, \to) =_\mathsf{k} \mathsf{max}\{\mathsf{dh}(s, \to) \mid s \in S \text{ and } |s| \leqslant n\} \ .$$

The complexity function is defined on all inputs, if $\to$ is *well-founded* and *finitely branching*. As shown in [8], well-foundedness alone is not a sufficient condition and finitely branching is not a necessary condition.

Definition 2.17. Let $\to|_S \subseteq A \times A$ denote the restriction of $\to$ to elements reachable from S. Let B be another set, $\rightsquigarrow \subseteq B \times B$ and $T \subseteq B$. Moreover, let $\gg \subseteq B \times A$ be a relation. We write $b \gg a$ instead of $(b, a) \in \gg$. We say $\rightsquigarrow|_T$ *abstracts* $\to|_S$, if $\gg \cdot \to|_S \subseteq \rightsquigarrow|_T \cdot \gg$ and $t \gg s$ for all $s \in S$ and some $t \in T$. We call $\gg$ the *abstraction* of $\to|_S$ to $\rightsquigarrow|_T$. Furthermore, $\gg$ is a *complexity preserving abstraction* if $\|t\| = O(|s|)$, for all $s \in S$ and $t \gg s$. Here $|\cdot|$ and $\|\cdot\|$ are suitable size measures for $s \in S$ and $t \in T$, respectively.

When analysing programs the binary relation $\to$ is induced by a finite description: the program code, or equivalently the control flow graph, where the edges are equipped with the concrete semantics. Recall the set of states construction of Example 2.9. The concrete semantics transforms sets of states to sets of states. In the special case where the concrete semantics is the set extension of the single-step semantics, we immediately obtain a relation on program states. We express this more formally in the following lemma.

Lemma 2.18. *Let $\mathcal{P}(A) = (\mathcal{P}(A), \subseteq, \bigcup, \bigcap, \varnothing, A)$ and B be complete lattices, and $(\mathcal{P}(A), \alpha, \gamma, B)$ be a Galois connection. Moreover $f^* \subseteq \gamma \circ f^\natural \circ \alpha$ and $S \subseteq \gamma(T)$ for extremal value $S \in \mathcal{P}(A)$ and $T \in B$ hold. Suppose $f^*(A') = \{f(a) \mid a \in A'\}$, for $A' \subseteq A$ and some function f, then $\gg \cdot \to|_S \subseteq \rightsquigarrow|_T \cdot \gg$ and $T \gg s$ for all $s \in S$, where $\to \subseteq A \times A$ and $\rightsquigarrow \subseteq B \times B$.*

Proof. By assumption $f^* \subseteq \gamma \circ f^\natural \circ \alpha$ and $S \subseteq \gamma(T)$. We set $b \gg a := \gamma(b) \ni a$. Then $T \gg s$ for all $s \in S$. Using $f_l^\natural \sqsupseteq \alpha \circ f^* \circ \gamma \Leftrightarrow f^* \subseteq \gamma \circ f^\natural \circ \alpha$, together with Lemma 2.11 we obtain $f^*(\gamma(b)) \subseteq \gamma(f^\natural(b))$, for all $b \in B$. Hence $\{f(a) \mid a \in \gamma(b)\} \subseteq \{a' \mid a' \in \gamma(f^\natural(b))\}$. Then $\{b \gg a \to f(a)\} \subseteq \{b \rightsquigarrow f^\natural(b) \gg a'\}$ follows. □

Example 2.19. Recall Example 2.16. By construction the abstraction $\mathsf{State}_{\mathsf{Sign}}$ is correct and the previous lemma applies. Hence, $\gg \cdot \to_{\mathsf{SS}}|_S \subseteq \rightsquigarrow_{\mathsf{State}_{\mathsf{Sign}}}|_T \cdot \gg$ and $T \gg s$ for all $s \in S$, where $S = \{\{i \mapsto \bot, n \mapsto \bot\}\}$ and $T = \{i \mapsto \bot, n \mapsto \bot\}$. The abstraction is also complexity preserving for a suitable size measure, yet not well-founded as nodes 3 and 4 admit a cycle in T.

3. Bytecode Programs

In this chapter we introduce Jinja bytecode (JBC for short) programs. Jinja is a Java like language that exhibits its core features, but is formally specified and verified in the proof assistant Isabelle [30, 33]. We expect the reader to be familiar with Java or a similar object-oriented language. Before introducing JBC programs formally, we provide a short overview.

Bytecode. The bytecode obtained from the compilation process provides a succinct representation of the original program, as it abstracts away complex syntax and programming features. The set of bytecode instructions we consider consists of only 20 statements. However, it is expressive enough to state most interesting programs. Bytecode is executed on a virtual machine. A program state of the virtual machine consists of a program environment, ie., a mapping from variables to values, and a global heap, ie., the dynamic memory.

Object-Oriented. The language provides a set of basic data types. Classes allow the user to define more complex data types by composing existing types, and consists of field and method declarations. Instances of classes are called objects. The distinction between the basic set of data types and user-defined data types is important as only objects can share memory in the heap, and sharing gives rise to side-effects that have to be safely approximated in the analysis.

Statically Typed. This means that we can approximate the content of program variables and object fields by their static type. We will see that the program specification gives rise to a subclass relation and the static type only provides a safe upper bound on the runtime type with respect to the subclass relation.

Static Field Access/Static Field Update. Field access and field update of an object do not depend on the runtime type of the object. In particular, the information which field is accessed or updated, is already encoded in the bytecode program.

Dynamic Method Invocation. On the other hand, method invocations depend on the (dynamic) runtime type of the object. That is, different methods may be invoked at the same program position depending on the runtime type of the object. The runtime type of an object can only be approximated with respect to the subclass relation. Hence, all possible types and methods have to be considered for a method invocation.

Well-Formed. Besides type information, well-formedness of bytecode implies several useful conditions on program states. This simplifies analysis, since malformed states can be excluded beforehand. The compiler presented in [30]

translates well-formed Jinja programs into well-formed JBC programs. Similarly a JBC program that passes the bytecode verification is also well-formed. In particular, well-formedness is a decidable property. We recapitulate relevant properties of well-formedness after introducing program states formally.

Program examples are usually illustrated using source code in a Java-like syntax, rather than the bytecode. The following example serves as running example for the rest of this thesis.

Example 3.1. Figure 3.1 depicts the `List` class. Instances of `List` have two accessible fields, `next` and `val`, and can invoke method `append`.

```
class List{
  List next;
  int val;

  void append(List ys){
    List cur = this;
    while(cur.next != null){
      cur = cur.next
    }
    cur.next = ys;
}
```

Figure 3.1.: The `List` class.

In the following we define Jinja bytecode programs formally.

Definition 3.2. A *Jinja value* can be

- a Boolean of type bool,
- an (unbounded) integer of type int,
- the dummy value unit of type void,
- the null reference null of type nullable, or
- a reference (or address).

The dummy value unit is used for the evaluation of assignments (see [30]) and to allocate uninitialised local variables. Non-null references are usually referred to as addresses. The actual type of addresses is not important and we usually identify the type of an address with the type of the object bounded to the address.

Definition 3.3. A *JBC program* P consists of a set of *class declarations*. Each class is identified by a *class name* and further consists of the name of its direct *superclass*, *field declarations* and *method declarations*. The superclass declaration is non-empty, except for a dedicated class termed *Object*. Moreover, the subclass hierarchy of P is tree-shaped. A field declaration is a pair of *field name*

and *field type*. A method declaration consists of the *method name*, a list of *parameter types*, the *result type* and the *method body*. A method body is a triple of $(mxs \times mxl \times instructionlist)$, where mxs and mxl are natural numbers denoting the maximum size of the operand stack and the number of local variables, not including the *this* reference and the parameters of the method, while *instructionlist* gives a sequence of bytecode instructions. The *this* reference can be conceived as a hidden parameter and references the object that invokes the method. Let n denote a natural number, i an integer, v a Jinja value, cn a class name, and mn a method name. The set of instructions `Ins` consists of:

$$\begin{aligned}\texttt{Ins} := {} & \texttt{Load}\ n \mid \texttt{Store}\ n \mid \texttt{Push}\ v \mid \texttt{Pop} \\ & \mid \texttt{IAdd} \mid \texttt{ISub} \mid \texttt{ICmpGte} \mid \texttt{CmpEq} \mid \texttt{CmpNeq} \mid \texttt{BAnd} \mid \texttt{BOr} \mid \texttt{BNot} \\ & \mid \texttt{Goto}\ i \mid \texttt{IfFalse}\ n \mid \\ & \mid \texttt{New}\ cn \mid \texttt{Getfield}\ fn\ cn \mid \texttt{Putfield}\ fn\ cn \mid \texttt{Checkcast}\ cn \mid \\ & \mid \texttt{Invoke}\ mn\ n \mid \texttt{Return}\end{aligned}$$

Class methods can only be invoked from class instances. Hence a program should have a class independent entry point, ie., a `main` function. Often we are only interested in a specific class method. Then we just assume that the *this* reference is not null. Moreover, we often confuse a program with the method under study.

Example 3.4. Consider the `List` class from Example 3.1. Figure 3.2 depicts the corresponding bytecode program, resulting from the compilation rules in [30]. In the following we name the registers 0, 1, and 2 as *this*, *ys*, and *cur*, respectively.

Let P be a JBC program. We fix P for the remainder of this chapter.

Definition 3.5. The class declarations of P naturally induces a strict subclass relation, denoted $\prec$. Its reflexive closure is denoted $\preceq$. We use $\mathsf{subclasses}(cn)$ to denote all subclasses of class cn for program P including cn itself.

Definition 3.6. We extend the subclass relation to a partial order on types, denoted $\leqslant_{\mathsf{type}}$. The types of P consists of $\{\mathsf{bool}, \mathsf{int}, \mathsf{void}, \mathsf{nullable}\}$ together with all classes cn defined in P. We use $\mathsf{type}(v)$ to denote the type of value v and $\mathsf{types}(P)$ to denote the collection of types in P. Recall that we usually identify the type of an address with the type of the object bound to the address. Let t, t', cn, cn' be types in P. Then $t \leqslant_{\mathsf{type}} t'$ holds if $t = t'$ or

- $t = \mathsf{void}$,
- $t = \mathsf{nullable}$ and $t' = cn$,
- $t = cn$, $t' = cn'$ and $cn \preceq cn'$.

The *least common superclass* is the least upper bound for a set of classes $CN \subseteq \mathsf{types}(P)$ and is always defined. If no confusion can arise, we abuse notation and use $\mathsf{type}(x)$ to denote the type of the value bound to variable x.

```
Class:
 Name: List                Bytecode:
 Classbody:                 00: Load 0
  Superclass: Object        01: Store 2
  Fields:                   02: Push unit
   List next                03: Pop
   int val                  04: Load 2
  Methods:                  05: Getfield next List
   Method: void append      06: Push null
    Parameters:             07: CmpNeq
     List ys                08: IfFalse 7
    Methodbody:             09: Load 2
     MaxStack:              10: Getfield next List
      2                     11: Store 2
     MaxVars:               12: Push unit
      1                     13: Pop
                            14: Goto -10
                            15: Push unit
                            16: Pop
                            17: Load 2
                            18: Load 1
                            19: PutField next List
                            20: Push unit
                            21: Return
```

Figure 3.2.: The bytecode for the `List` class.

We comment on the programming language features to clarify the scope of the programs under study. All programs are single-threaded. No integer overflow can occur as the integer type represents unbounded integers. Furthermore there is no instruction for multiplication. There is no support for arrays, strings or floating point numbers. We assume that the garbage collector collects unreachable objects immediately. In Chapter 7 we will indicate some difficulties arising from the above mentioned program features.

4. Concrete Domain

In this chapter we specify the concrete domain of our analysis. In Section 4.1 we provide a formal definition for program states based on the formalisation in [30]. In Section 4.2 we introduce (an equivalent) graph based representation of program states. Afterwards, we provide the single-step and collecting semantics in Section 4.3 and Section 4.4, respectively.

4.1. Concrete States

Bytecode is executed on the Jinja virtual machine (JVM).

Definition 4.1. A *(JVM) state* is a pair consisting of the *heap* and a list of *frames*. A *heap* is a mapping from *addresses* to *objects*, where an object is a pair $(cn, ftable)$ such that:

- cn denotes the *class name*, and
- $ftable$ denotes the fieldtable, ie., a mapping from (cn', fn) to values, where fn is a *field name* and $cn \preceq cn'$.

A *frame* is a quintuple (stk, loc, cn, mn, pc) such that:

- stk denotes the *operation stack*, ie., an array of values,
- loc denotes the *registers*, ie., an array of values,
- cn denotes the *class name*,
- mn denotes the *method name*, and
- pc is the *program counter*.

Suppose $obj = (cn, ftable)$. We define projection functions cl and ft as follows: $\mathsf{cl}(obj) := cn$ and $\mathsf{ft}(obj) := ftable$. Note that $\mathsf{cl}(heap(a))$ and $\mathsf{ft}(heap(a))$ are undefined if $heap(a)$ is undefined.

Let stk (loc) denote the operation stack (registers) of a given frame. Typically the structure of loc is as follows: the 0^{th} register holds the $this$ pointer, followed by the parameters and the local variables of the method. Uninitialised registers are preallocated with the dummy value unit. We denote the entries of stk (loc), by $stk(i)$ ($loc(i)$) for $i \in \mathbb{N}$ and write $\mathsf{dom}(stk)$ ($\mathsf{dom}(loc)$) for the set of indices of the array stk (loc).

Observe that the domain of the fieldtable for a given object of class cn contains all fields declared for cn together with all fields declared for superclasses of cn. Clearly the domain of the fieldtable is equal for any instance of cn.

Definition 4.2. Let $s = (heap, frms)$. The *program location* of s is a list of triples (cn, mn, pc) obtained by restricting the elements of $frms$ to class name, method name and program counter.

The well-formedness criterion of JBC implies several properties on the JVM: Bytecode instructions are provided with arguments of the expected type. No instruction tries to get a value from the empty stack, nor puts more elements on the stack or accesses more elements than specified. The program counter is always within the code array of the method. All registers, except the register storing *this*, must be written into before accessed. If two states have the same program location, then for all frames the domain $\mathsf{dom}(stk)$ ($\mathsf{dom}(loc)$) coincides for the two states. Moreover, corresponding entries of stk (loc) of the two states are well-typed in the sense that $v \leqslant_{\mathsf{type}} v'$ or $v' \leqslant_{\mathsf{type}} v$ holds for the two corresponding entries.

In the following we consider Jinja programs and JBC programs to be well-formed. To ease readability, we do not consider exception handling, ie., an exception yields immediate termination of the program. This is not a restriction of our analysis, as it could be easily integrated, but complicates matters without gaining additional insight.

While the above form of representing states allows for a succinct presentation, it is more natural to conceive the heap (and conclusively a state) as a graph. In the following we make this intuition precise.

4.2. State Graphs

Let s be a state and let *heap* denote the heap of s. We propose a graph-based representations of *heap* and state s called *heap graph* and *state graph*. This representation makes technical use of a set $\mathcal{I}_{heap}$ of *implicit references*. Suppose a is an address on the *heap*, cn a class name, and id a field identifier. Furthermore, suppose $ftable = \mathsf{ft}(heap(a))$ is defined and $ftable((cn, id)) = val$ such that val is not an address. Then we say the triple (a, cn, id) is an *implicit reference* for val. The set $\mathcal{I}_{heap}$ collects all implicit references of *heap*.

Definition 4.3. Let *heap* denote the heap. We represent *heap* as a directed graph with edge labels $H = (V_H, Succ_H, L_H, E_H)$, where the nodes, the successor relations and the labelling functions are defined as follows:

$$V_H := \mathsf{dom}(heap) \cup \mathsf{dom}(\mathcal{I}_{heap})$$

$$Succ_H(u) := \begin{cases} [f^*(u, (cn_1, id_1)), \ldots, & \\ f^*(u, (cn_k, id_k))] & \text{if } u \text{ is an address, } \mathsf{ft}(heap(u)) = f, \\ & \mathsf{dom}(f) = \{(cn_1, id_1), \ldots, (cn_k, id_k)\} \\ [] & \text{otherwise .} \end{cases}$$

$$L_H(u) := \begin{cases} \mathsf{cl}(heap(u)) & \text{if } u \text{ is an address} \\ val & \text{if } u \text{ is an implicit reference for } val \text{ .} \end{cases}$$

$$E_H(u \to v) := \begin{cases} (cn, id) & \text{if } u \text{ is an address, } \mathsf{ft}(heap(u)) = f, f^*(u, (cn, id)) = v \\ \varnothing & \text{otherwise .} \end{cases}$$

Here f^* is defined as follows: $f^*(u,(cn,id)) := f((cn,id))$ if $f((cn,id))$ is an address and $f^*(u,(cn,id)) := (u,cn,id)$ otherwise, where $(u,cn,id) \in \mathcal{I}_{heap}$.

Based on the heap graph, we represent s as a *state graph* S. Let $s = (heap, frms)$ be a state and let $frms = [frm_1,\ldots,frm_k]$ such that $frm_i = (stk_i, loc_i, cn_i, mn_i, pc_i)$. We define the set $Stk(s) := \{(stk,i,j) \mid 1 \leqslant i \leqslant k, 1 \leqslant j \leqslant |stk_i|\}$ that collects all *stack indices*. Similarly we define the set of *register indices*: $Loc(s) := \{(loc,i,j) \mid 1 \leqslant i \leqslant k, 1 \leqslant j \leqslant |loc_i|\}$. If s is clear from context, we write Stk (Loc) instead of $Stk(s)$ ($Loc(s)$). We extend the set $\mathcal{I}_{heap}$ to cover also non-address values stored in the stack or registers. For this it suffices to extend $\mathcal{I}_{heap}$ by a disjoint copy of $Stk(s) \cup Loc(s)$. The set of implicit references with respect to s is denoted as $\mathcal{I}_s$. The copy of a stack or register index in $\mathcal{I}_s$ is called its *implicit reference*.

Definition 4.4. Let $s = (heap, frms)$ be a state and let H denote the heap graph of *heap*. Furthermore, let $(stk,i,j) \in Stk$ and let $(loc,i',j') \in Loc$. We write $os_{(i,j)}$ and $l_{(i',j')}$ to name the indices of the operation stack and registers. We define the *state graph of* s as quadruple $S = (V_S, Succ_S, L_S, E_S)$. The nodes, the successor relation, and the labelling function of the directed graph are defined as follows:

$$V_S := Stk \cup Loc \cup V_H \cup \mathcal{I}_s$$

$$Succ_S(u) := \begin{cases} [stk_i^*(j)] & \text{if } u = (stk,i,j) \in Stk \\ [loc_i^*(j)] & \text{if } u = (loc,i,j) \in Loc \\ Succ_H(u) & \text{if } u \in V_H\ . \end{cases}$$

$$L_S(u) := \begin{cases} os_{(i,j)} & \text{if } u = (stk,i,j) \in Stk \\ l_{(i,j)} & \text{if } u = (loc,i,j) \in Loc \\ L_H(u) & \text{if } u \in V_H \\ val & \text{if } u \text{ is an implicit reference for } val\ . \end{cases}$$

$$E_S(u \to v) := \begin{cases} E_H(u \to v) & \text{if } u,v \in H \\ \varnothing & \text{otherwise}\ . \end{cases}$$

Here $stk_i^*(j)$ and $loc_i^*(j)$ is defined like f^* as introduced in Definition 4.3, ie., $stk_i^*(j) := stk_i(j)$ ($loc_i^*(j) := loc_i(j)$), if $stk_i(k)$ ($loc_i(j)$) is an address and $stk_i^*(j)(loc_i^*(j))$ is defined as the implicit reference of $(stk,i,j)((loc,i,j))$ otherwise.

Below we often confuse a state s and its representation as a state graph S. We define some relevant properties on elements of the state graph:

Definition 4.5. Let s be a state and S be its state graph. Furthermore, let r, p and q be references in V_{Heap}. We say that:

- r *alias with* p, if $r = p$;
- r *reaches* p, if $r \xrightarrow{+}_S p$;

- r *shares with* p, if there exists q such that $r \xrightarrow{*}_S q \;{}_S\!\xleftarrow{*} p$;
- r is *cyclic*, if $r \xrightarrow{+}_S r$; and
- r is *acyclic*, if r is not cyclic.

Sometimes, we use the above properties on variables, eg., two variables alias if they store the same address.

The *size* of a state is defined on a *per-reference* basis, which unravels sharing. We explicitly add 1 to the overall construction. This does not affect the results but allows a more convenient relation to the size of terms we present later.

Definition 4.6. Let s be a state and let S be its state graph. Let u, v be nodes in S and $u \xrightarrow{+}_S v$ denote a simple path in S from u to v. Then the size of a stack or register index u, denoted as $|u|$, is defined as follows:

$$|u| := \sum_{u \xrightarrow{+}_S v} |L_S(v)| \,,$$

where $|l|$ is $\mathsf{abs}(l)$ if $l \in \mathbb{Z}$, otherwise 1, for $l \in L_S$. Here, $\mathsf{abs}(z)$ denotes the absolute value of the integer z. Then the *size* of s is the sum of all sizes of stack or register indices in S plus 1. In the following we use $|s|$ to denote the size of a state s.

4.3. Bytecode Semantics

Reconsider the set of instructions provided in Definition 3.3. In the following we provide an informal description of their semantics. Appendix A illustrates the formalisation of all instructions in the same format as used below:

$$\texttt{IAdd}\ \frac{(heap, (i_2 : i_1 : stk, loc, cn, mn, pc) : frms)}{(heap, ((i_2 + i_1) : stk, loc, cn, mn, pc + 1) : frms)}$$

Here, i_1, i_2 denote arbitrary integer values.

Most instructions affect only the current frame. `Load` n pushes the value of $loc(n)$ onto the stack, whereas `Store` n pops the top value of the operand stack and stores it at $loc(n)$. `Push` v pushes the value v onto the stack, whereas `Pop` removes the top element of the stack. `IAdd`, `ISub`, `ICmpGte`, `BAnd`, `BOr`, and `BNot` denote the usual operations on integer and Boolean values. Operands are taken from the top of the stack and the result is pushed onto the stack. `CmpEq` and `CmpNeq` define equality and inequality on all values.

`Goto` i defines an unconditional (relative) jump. `IfFalse` n defines a conditional control flow instruction that depends on the value on top of the stack.

`New` cn allocates a new instance of type cn in the heap and pushes the corresponding address onto the stack. All fields of the fresh created instance are instantiated with the default value. That is, 0 for integer typed fields, `false` for Boolean typed fields, and `null` otherwise. `Getfield` fn cn dereferences the reference on top of the stack and replaces the reference by the content of the

field identified by (cn, fn) onto the stack. `Putfield` fn cn dereferences the reference next to the top of the stack and updates the content of the field identified by (cn, fn) using the value on top of the stack. The instruction `Checkcast` cn checks the type of the reference on top of the stack. The instruction `Invoke` mn n first copies the top n element of the stack in reversed order. It then identifies the method to invoke by the runtime type of the corresponding reference and constructs a new frame including *this*, the parameters, and the local variables initialised with `unit`. The program counter of the new frame is set to 0. The instruction `Return` pops the top frame and updates the stack of the next frame with the return value. If the frame stack consists only of a single frame then the program terminates.

All instructions beside the jump instructions increment the program counter by one. The instructions `Getfield`, `Putfield`, `Checkcast`, `Invoke` dereference references and may fail if the reference is `null`.

Definition 4.7. Let P be a program and let s and t be states. Then we denote by $s \to_P t$ the one-step transition relation of the JVM. If there exists a evaluation of s to t, we write $s \to_P^* t$.

Consider the `append` method in Example 3.1. It is easy to see that the complexity of `append` does not only depend on the size of the initial state but also on the shape of the objects. If *this* is bound to an acyclic `List` instance, then the complexity depends only on the length of the list. Otherwise, the `while` loop does not terminate. There is one more case to consider. If *this* is bound to an acyclic instance and *this* shares with parameter *ys* initially, then the loop does terminate, but *this* is cyclic after the putfield instruction. This information is relevant in an interprocedural analysis. Therefore, beside input size, we incorporate the set of input states into the definition of the runtime complexity of some program P.

We define the *runtime* of a JVM for a given evaluation $s \to_P^* t$ as the number of single-step executions in the course of the evaluation from s to t.

Definition 4.8. Let $\mathcal{JS}$ denote the set of JVM states, and $\mathcal{S} \subseteq \mathcal{JS}$. We define the *runtime complexity* with respect to P as follows:

$$\mathsf{rcjvm}(n) =_{\mathsf{k}} \mathsf{max} \left\{ m \;\middle|\; \begin{array}{l} i \to_P^* t \text{ holds such that the runtime is } m, \\ i \in \mathcal{S} \text{ and } |i| \leqslant n \end{array} \right\}$$

Note that we adopt a (standard) unit cost model for system calls.

4.4. Collecting Semantics

In this section we fix the collecting semantics for our analysis. Above, we already restricted our attention to well-formed JBC programs. For the proposed static analysis of these programs we also restrict ourselves to *non-recursive* methods. In the next chapter we will see that our approach cannot handle unbounded list of frames. In the following let P be a well-formed and non-recursive program.

Definition 4.9. The complete lattice $\mathcal{P}(\mathcal{JS}) := (\mathcal{P}(\mathcal{JS}), \subseteq, \cup, \cap, \varnothing, \mathcal{JS})$ defines the *concrete computation domain*. Let $\mathcal{S} \subseteq \mathcal{JS}$. We define the *collecting semantics* on the domain $\mathcal{P}(\mathcal{JS})$ as the component-wise extension of the one-step transition relation to sets: $\{t \mid s \to_P t, s \in \mathcal{S}\}$.

5. Abstract JVM Domain

In this chapter we provide an abstract representation of all program executions of some program P. In Section 5.1 we introduce abstract program states. Abstract states are similar to concrete states but incorporate variables, so that an abstract state represents a set of concrete states. We construct a Galois insertion between the set of concrete states and abstract states. In Section 5.2 we provide the abstract semantics and show that a computation step in the abstract domain safely approximates a step in the concrete domain. In Section 5.3 we introduce computation graphs as finite representations of all program runs of program P. Several concepts introduced in the previous chapters are generalised here for abstract states. We use superscript $\natural$ to distinguish them. If it is clear from the context the annotation is often omitted.

5.1. Abstract States

In this section, we introduce *abstract states* as generalisations of JVM states.

Definition 5.1. We extend Jinja expressions by countable many abstract variables $X_1, X_2, X_3, \ldots$, denoted by $x, y, z, \ldots$ An *abstract variable* may either abstract an object, an integer or a Boolean value.

In denoting abstract variables typically the name is of less importance than the type. Hence, we denote an abstract variable for an object of class cn, simply as cn, while abstract integer or Boolean variables are denoted as int, and $bool$, respectively. The subclass (cf. Definition 3.5) relation is extended in the natural way to abstract variables for classes. For brevity we sometimes refer to an abstract variable of integer or Boolean type, as *abstract integer* or *abstract Boolean*, respectively.

Definition 5.2. An *abstract value* is either a Jinja value (cf. Definition 3.2), or an abstract integer or an abstract Boolean. In turn a Jinja value is also called a *concrete value*.

Definition 5.3. An *abstract heap* is a mapping from *addresses* to *abstract objects*, where an abstract object is either a pair $(cn, ftable)$ or an abstract (class) variable cn. *Abstract frames* are defined like frames of the JVM, but registers and operand stack of an abstract frame store abstract values. We define (partial) projection functions cl and ft as follows:

$$\mathsf{cl}(obj) := \begin{cases} cn & \text{if } obj \text{ is an object and } obj = (cn, ftable) \\ cn & \text{if } obj \text{ is an abstract variable of type } cn\ , \end{cases}$$

$$\mathsf{ft}(obj) := \begin{cases} ftable & \text{if } obj \text{ is an object and } obj = (cn, ftable) \\ \text{undefined} & \text{otherwise}\ . \end{cases}$$

Furthermore, we define *annotations* of addresses in an abstract state s, denoted as iu. Formally, annotations are pairs $p \neq q$ of addresses, where $p, q \in heap$ and p differs from q.

Definition 5.4. An *abstract state* $s = (heap, frms, iu)$ is either a triple consisting of an abstract heap $heap$, a list of abstract frames $frms$, and a set of annotations iu, the *maximal abstract state*, denoted as $\top$, or the *minimal abstract state*, denoted as $\bot$. If $s = (heap, frms, iu)$, we demand that all addresses in $heap$ are reachable from local variables or stack entries in the list of frames $frms$. The set of abstract states is collected in the set $\mathcal{AS}$.

Due to the presence of abstract variables, abstract states can represent sets of states as the variables can be suitably instantiated. Thus different JVM states can be abstracted to a single abstract state. To make this precise, we will augment $\mathcal{AS}$ with a partial order $\sqsubseteq$, the *instance relation* (see Definition 5.7). We will extend the partial order $(\mathcal{AS}, \sqsubseteq)$ to a complete lattice $\mathcal{AS} := (\mathcal{AS}, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$ and establish a Galois insertion between $\mathcal{P}(\mathcal{JS})$ and $\mathcal{AS}$. The annotation $p \neq q \in iu$ will be used to disallow sharing of these addresses in states represented by the state s.

When depicting abstract states, we replace stack and register indices by intuitive names, written in roman font. Furthermore, we make use of the following conventions: we use an italic font (and lower-case) to describe abstract variables and a sans serif (and upper-case) to depict class names. Before we present the formal definition of the relation $\sqsubseteq$, we provide an example that should clarify the intuition.

Example 5.5. Recall the `List` class from Example 3.1 together with the well-formed JBC program depicted in Figure 3.2. Consider the state A depicted below:

04	$\epsilon \mid \text{this} = o_1, \text{ys} = o_2, \text{cur} = o_1$
	$o_1 = \mathsf{List}(\text{List.val} = int, \text{List.next} = o_3)$
A	$o_2 = list, o_3 = list$

The operation stack in A is empty, denoted by ϵ. The registers *this* and *cur* contain the same address o_1 and *ys* is mapped to o_2. In the heap o_1 is mapped to an object of type List whose value is abstracted to *int* and whose next element is referenced by o_3. It is not difficult to see that A forms an abstraction of any JVM state obtained with program counter 04 in the `append` program (if *this* initially references a non-empty list) before any iteration of the `while` loop. Now, consider state B:

04	$\epsilon \mid \text{this} = o_1, \text{ys} = o_2, \text{cur} = o_3$
	$o_1 = \mathsf{List}(\text{List.val} = int, \text{List.next} = o_3)$
	$o_2 = list, o_4 = list$
B	$o_3 = \mathsf{List}(\text{List.val} = int, \text{List.next} = o_4)$

Again, it is not difficult to see that B abstracts any JVM state obtained if exactly one iteration of the loop has been performed.

Definition 5.6. We define a preorder on abstract values, which are not references, and abstract objects. We extend $\mathsf{type}(v)$ (cf. Definition 3.6) to abstract values the intended way, ie., $\mathsf{type}(int) = int, \mathsf{type}(bool) = bool$ and $\mathsf{type}(cn) = cn$ for an integer variable int, a Boolean variable $bool$, and class variable cn. We define the preorder $\trianglelefteq$ as follows: We have $v \trianglelefteq w$, if either

(i) $v = w$, or

(ii) $\mathsf{type}(v) \leqslant_{\mathsf{type}} \mathsf{type}(w)$ and w is an abstract variable.

We write $w \trianglerighteq v$, if $v \trianglelefteq w$.

Let $|stk|$, $|loc|$ denote the size of the operand stack and the number of registers respectively. We make use of the following abbreviation: $w \trianglerighteq_m v$ if either $w \trianglerighteq v$ or v, w are references and we have $v = m(w)$, where m denotes a mapping on references.

Definition 5.7. Let $s = (heap, frms, iu)$ be a state in $\mathcal{AS} \setminus \{\bot, \top\}$ with $frms = [frm_1, \ldots, frm_k]$ and $frm_i = (stk_i, loc_i, cn_i, mn_i, pc_i)$. Furthermore let $t = (heap', frms', iu')$ be a state with $frms' = [frm'_1, \ldots, frm'_k]$ and $frm'_i = (stk'_i, loc'_i, cn'_i, mn'_i, pc'_i)$. Then s is an *abstraction* of t, denoted as $s \sqsupseteq t$, if the following conditions hold:

1. for all $1 \leqslant i \leqslant k$: $pc_i = pc'_i$, $cn_i = cn'_i$ and $mn_i = mn'_i$,
2. for all $1 \leqslant i \leqslant k$: $\mathsf{dom}(stk_i) = \mathsf{dom}(stk'_i)$ and $\mathsf{dom}(loc_i) = \mathsf{dom}(loc'_i)$ and
3. there exists a mapping $m\colon \mathsf{dom}(heap) \to \mathsf{dom}(heap')$ such that
   - for all $1 \leqslant i \leqslant k$, $1 \leqslant j \leqslant |stk_i|$: $stk_i(j) \trianglerighteq_m stk'_i(j)$,
   - for all $1 \leqslant i \leqslant k$, $1 \leqslant j \leqslant |loc_i|$: $loc_i(j) \trianglerighteq_m loc'_i(j)$,
   - for all $a \in \mathsf{dom}(heap)$: $heap(a) \trianglerighteq heap'(m(a))$,
   - for all $a \in \mathsf{dom}(heap)$ such that $\mathsf{ft}(heap(a))$ is defined and for all $1 \leqslant i \leqslant \ell$: $f(cn_i, id_i) \trianglerighteq_m f'(cn_i, id_i)$, where $f := \mathsf{ft}(heap(a))$ with $\mathsf{dom}(f) = \{(cn_1, id_1), \ldots, (cn_\ell, id_\ell)\}$ and $f' := \mathsf{ft}(heap'(m(a)))$ with $\mathsf{dom}(f') = \{(cn_1, id_1), \ldots, (cn_\ell, id_\ell)\}$.
4. finally, we have $iu' \supseteq m^*(iu)$.

Here, m^* denotes the lifting of the morphism m to sets: $m(\{iu_1, \ldots, iu_k\}) = \{m(iu_1), \ldots, m(iu_k)\}$. Furthermore for all $s \in \mathcal{AS}$: $s \sqsubseteq \top$ and $\bot \sqsubseteq s$.

Example 5.8 (continued from Example 5.5)**.** For the state S depicted below we obtain that $A \sqsubseteq S$ and $B \sqsubseteq S$, ie., S forms an abstraction of both states.

04	$\epsilon \mid \text{this} = o_1, \text{ys} = o_2, \text{cur} = o_4$
	$o_1 = \mathsf{List}(\text{List.val} = int, \text{List.next} = o_3)$
	$o_2 = list, o_3 = list, o_5 = list$
S	$o_4 = \mathsf{List}(\text{List.val} = int, \text{List.next} = o_5)$

The definition for the heap graph (cf. Definition 4.3) and the state graph (cf. Definition 4.4) naturally extends to abstract states, when extending $S = (V_S, Succ_S, L_S, E_S)$ to $S = (V_S, Succ_S, L_S, E_S, iu_S)$ and considering abstract values rather than concrete values only. Furthermore, we use $\top$ to denote the state graph of $\top \in \mathcal{AS}$ and the empty graph to denote $\bot \in \mathcal{AS}$. In presenting state graphs, we indicate references, but do not depict implicit references.

Example 5.9 (continued from Example 5.8)**.** The state graphs of A and B are given in Figure 5.1 and Figure 5.2, respectively. The state graph of the abstraction S is depicted in Figure 5.3.

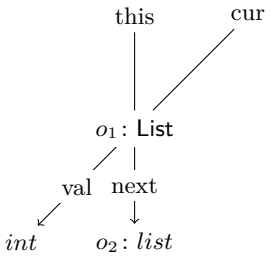

Figure 5.1.: Abstract State A.

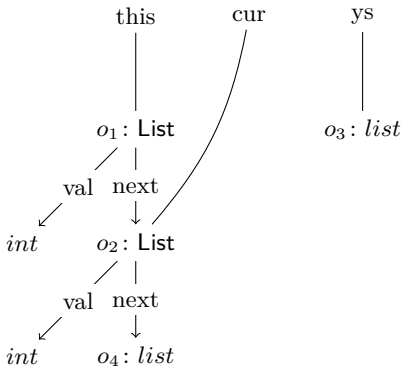

Figure 5.2.: Abstract State B.

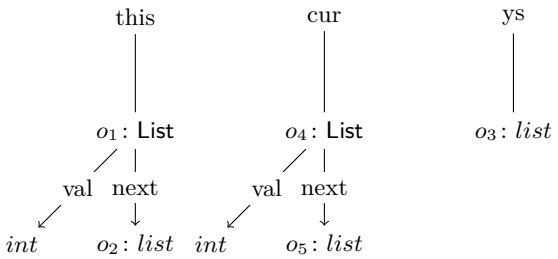

Figure 5.3.: Abstraction S.

We introduce *state homomorphisms* that allow an alternative, but equivalent definition of the instance relation $\sqsubseteq$.

Definition 5.10. Let $s, t \in \mathcal{AS} \setminus \{\bot, \top\}$ and S and T be their state graphs. A *state homomorphism* from S to T (denoted $m\colon S \to T$) is a function $m\colon V_S \to V_T$ such that

1. for all $u \in S$ and $u \in Stk \cup Loc$, $L_S(u) = L_T(m(u))$,
2. for all $u \in S \setminus (Stk \cup Loc)$, $L_S(u) \trianglerighteq L_T(m(u))$,
3. for all $u \in S$: if $u \xrightarrow{i}_S v$, then $m(u) \xrightarrow{i}_T m(v)$ and
4. for all $u \xrightarrow{\ell} v \in S$ and $m(u) \xrightarrow{\ell'} m(v) \in T$, $\ell = \ell'$.

We sometimes abuse notation and use $m\colon s \to t$ instead of $m\colon S \to T$.

If no confusion can arise we refer to a state homomorphism simply as *morphism*. It is easy to see that the composition $m_1 \circ m_2$ of two morphisms m_1, m_2 is again a morphism. We say that two states $s, t \in \mathcal{AS}$ are *isomorphic* if there exists a morphism from s to t and vice versa. Suppose the abstract states s and t are isomorphic. Then they differ only in their abstract variables and can be transformed into each other through a renaming of variables. Thus the set of JVM states represented by s and t is equal; we call s and t *equivalent* (denoted $s \sim t$).

Let $s, t \in \mathcal{AS}$ and let S and T denote their state graphs. Then $s \sqsupseteq' t$ if one of the following alternatives holds: (i) $S = \top$, (ii) $T = \varnothing$, or (iii) $S, T \neq \varnothing$ and there exists a state morphism m from S to T, $s = (heap, frms, iu)$, $t = (heap', frms', iu')$ have the same program location (cf. Definition 4.2) and $iu' \supseteq m^*(iu)$.

Lemma 5.11. *Let $s, t \in \mathcal{AS}$. Then $s \sqsubseteq t$ iff $s \sqsubseteq' t$.*

Proof. Straightforward. □

Due to Lemma 5.11 and the composability of morphisms it follows that the instance relation $\sqsubseteq$ is transitive. Hence the relation $\sqsubseteq$ is a preorder. Furthermore $\sqsubseteq$ can be lifted to a partial order, if we consider the factorisation of the set of abstract states with respect to the equivalence relation $\sim$. In order to express this fact notationally, we identify isomorphic states and replace $\sim$ by $=$. Conclusively $(\mathcal{AS}, \sqsubseteq)$ is a partial order. We are left to provide a least upper bound definition of the *join* of abstract states.

Definition 5.12. Let s and s' be states such that there exists an abstraction t of s and s'. We call t the *join* of s and s', denoted as $s \sqcup s'$, if t is a least upper bound of $\{s, s'\}$ with respect to the preorder $\sqsubseteq$.

The limit cases are handled as usual. If the program locations of s and s' differ, then $s \sqcup s' = \top$. Otherwise, we can identify invariants to construct an upper bound $t \neq \top$ and prove well-definedness of $s \sqcup s'$. Let $S = (V_S, Succ_S, L_S, E_S, iu_S)$ and $S' = (V_{S'}, Succ_{S'}, L_{S'}, E_{S'}, iu_{S'})$ be the two state graphs of state s and s', respectively. Furthermore, let t be an abstraction of s and s', and let $T = (V_T, Succ_T, L_T, E_T, iu_T)$ be its state graph. By definition we have the following properties:

1. Let Stk (Loc) collect the stack (register) indices of state s. As $s \sqsubseteq t$, Stk (Loc) coincides with the set of stack (register) indices of t. Similarly for s' and thus $V_T \supseteq Stk \cup Loc$.

2. For any node $u \in T$ there exist uniquely defined nodes $v \in V_S$, $w \in V_{S'}$ such that $L_S(v) \trianglelefteq L_T(u)$, $L_{S'}(w) \trianglelefteq L_T(u)$. We say the nodes v and w *correspond* to u.

3. For any node $u \in T$ and any successor u' of u in T there exists a successor v' (w') in S (S') of the corresponding node v (w) in S (S'). Furthermore v' and w' correspond to u'.

4. For any edge $u \xrightarrow{\ell} u' \in T$ such that v (w) corresponds to u in S (S') there is an edge $v \xrightarrow{k} v' \in S$ and an edge $w \xrightarrow{k'} w' \in S'$ such that $\ell = k = k'$.

5. For any annotation $u \neq u' \in iu_T$ there exists $v \neq v'$ in iu_S and $w \neq w'$ in $iu_{S'}$, where v (v') and w (w') correspond to u (u').

In order to construct an abstraction t of s and s' we use the above properties as invariants and define its state graph T by iterated extension. We define T^0 by setting $V_{T^0} := Stk \cup Loc$. Due to Property 1 these nodes exist in S and S' as well. The labels of stack or register indices trivially coincide in S and S', cf. Definition 5.10. Thus we set L_{T^0} accordingly. Furthermore we set $Succ_{T^0} = E_{T^0} = iu_{T^0} := \varnothing$. Then T^0 satisfies Properties 1–5.

Suppose state graph T^n has already been defined such that the Properties 1–5 are fulfilled. In order to update T^n, let $u \in V_{T^n}$ such that v and w correspond to u. Suppose $v \xrightarrow{k} v' \in S$ and $w \xrightarrow{k} w' \in S'$ such that there is no node u' in T^n where v' and w' correspond to u'. Let u' denote a node fresh to T^n. We define $V_{T^{n+1}} := V_{T^n} \cup \{u'\}$ and establish Property 2 by setting $L_{T^{n+1}}(u')$ such that $L_S(v') \trianglelefteqslant L_{T^{n+1}}(u')$ and $L_{S'}w' \trianglelefteqslant L_{T^{n+1}}(u')$ where $L_{T^{n+1}}(u')$ is as concrete as possible. If we succeed, we fix that v' and w' correspond to u'. It remains to update $iu_{T^{n+1}}$ suitably such that Property 5 is fulfilled. If this also succeeds Properties 1–5 are fulfilled for T^{n+1}. On the other hand, if no further update is possible we set $T := T^n$. By construction T is an abstraction of S and S' and indeed represents $s \sqcup s'$.

Example 5.13 (continued from Example 5.9)**.** In Figure 5.3 abstract state S is introduced as an abstraction of abstract states A and B. In particular, S results from the construction defined above, ie., $S = A \sqcup B$.

Given the definitions above $\mathcal{AS} := (\mathcal{AS}, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$ is a complete lattice.

Lemma 5.14. *The partial order $(\mathcal{AS}, \sqsubseteq)$ satisfies the ascending chain condition, ie., any ascending chain eventually stabilises.*

Proof. In order to derive a contradiction we assume the existence of an ascending sequence $(s_i)_{i \geqslant 0}$ that never stabilises. By definition for all $i \geqslant 0$: $|s_i| \geqslant |s_{i+1}|$. By assumption there exists $i \in \mathbb{N}$ such that for all $j > i$: $|s_i| = |s_j|$ and $s_i \sqsubset s_j$. The only possibility for two different states s_i, s_j of equal size that $s_i \sqsubset s_j$ holds, is that addresses shared in s_i become unshared in s_j. Clearly this is only possible for a finite amount of cases. Contradiction. □

Definition 5.15. Let $s = (heap, frms) \in \mathcal{JS}$, we define the representation function $\beta\colon \mathcal{JS} \to \mathcal{AS}$, that injects JVM states into $\mathcal{AS}$. Suppose $\mathsf{dom}(heap) = \{p_1, \ldots, p_n\}$. Define iu such that all $p_i \neq p_j \in iu$ for all different i, j. Then $\beta(s) = (heap, frms, iu)$. Let $\alpha\colon \mathcal{P}(\mathcal{JS}) \to \mathcal{AS}$ and $\gamma\colon \mathcal{AS} \to \mathcal{P}(\mathcal{JS})$ be the abstraction function and the concretisation function induced by β (cf. Definition 2.14). Then $(\mathcal{P}(\mathcal{JS}), \alpha, \gamma, \mathcal{AS})$ is a Galois connection.

It is easy to see that $\mathcal{AS}$ may contain redundant elements. Consider abstract states $s^\natural, t^\natural$. Let $s^\natural = (heap, frms, iu)$, $p, q \in \mathsf{dom}(heap)$ and $p \neq q \in iu$. Let $t^\natural$ be defined like $s^\natural$ but $p \neq q \notin iu$. Furthermore, suppose that there exists

no concrete state $s \in \gamma(s^\natural)$ such that $m(p) = m(q)$ in s for some morphism $m : s^\natural \to \beta(s)$. This happens, for example, if $s^\natural = \beta(s)$ for some concrete state s, and references $m(p), m(q)$ point to instances with a different type. Then $s^\natural \sqsubset t^\natural$ and $\gamma(s^\natural) = \gamma(t^\natural)$.

To form a Galois insertion between $\mathcal{P}(\mathcal{JS})$ and $\mathcal{AS}$, we introduce a reduction operator (cf. Definition 5.16) that adds annotations for non-aliasing addresses.

Definition 5.16. Let $s^\natural = (heap, frms, iu)$ be an abstract state. We define the *reduction operator* $\varsigma \colon \mathcal{AS} \to \mathcal{AS}$ as follows:

$$\varsigma(s^\natural) := (heap, frms, iu') \ ,$$

where $iu' := \{p \neq q \mid p, q \in \mathsf{dom}(heap)\} \setminus \{p \neq q \mid s \in \gamma(s^\natural), m : s^\natural \to \beta(s), m(p) = m(q)\}$. Then $\varsigma(s^\natural) \sqsubseteq s^\natural$ and $\gamma(\varsigma(s^\natural)) = \gamma(s^\natural)$.

In practice, we compute the reduction by a unification argument of p and q in $s^\natural$: We try to construct a new state $t^\natural \sqsubseteq s^\natural$, where $r = m(p) = m(q)$. Let $T^\natural$ and $S^\natural$ be the state graphs of $t^\natural$ and $s^\natural$. Suppose u, v, w represent r, p, q in $T^\natural$ and $S^\natural$. We can use a similar reasoning we used for the join construction, but now require $L_{T^\natural}(u) \trianglelefteq L_{S^\natural}(v)$ and $L_{T^\natural}(u) \trianglelefteq L_{S^\natural}(w)$ if v and w correspond to u. If the construction succeeds, we can easily find a concrete state from $t^\natural$ such that $m(p) = m(q)$. The construction does not succeed if, for example, successors of corresponding nodes have different concrete values; then we add $p \neq q$.

Lemma 5.17. *The maps α and γ define a Galois insertion between the complete lattices $\mathcal{P}(\mathcal{JS})$ and $\varsigma^*(\mathcal{AS})$, where ς^* denotes the set extension of ς.*

Proof. It suffices to prove that γ is injective, ie., for all $s^\natural, t^\natural \in \varsigma^*(\mathcal{AS})$ if $s^\natural \neq t^\natural$ then $\gamma(s^\natural) \neq \gamma(t^\natural)$. Suppose $s^\natural \neq t^\natural$ but $\gamma(s^\natural) = \gamma(t^\natural)$. It is a simple consequence of our morphism definition that $\gamma(s^\natural) \neq \gamma(t^\natural)$, if the state graphs of $s^\natural$ and $t^\natural$ differ. Hence, $s^\natural$ can only be different from $t^\natural$ if the annotations of $s^\natural$ and $t^\natural$ differ. However, by assumption they are equal. Contradiction. □

It follows that the reduction operator defined in Definition 5.16, indeed returns the greatest lower bound that represents the same element in the concrete domain as required. In the following we identify the $\varsigma^*(\mathcal{AS})$ with $\mathcal{AS}$.

Recall Definition 5.18 defining several properties on heap elements. We lift this functions to abstract states.

Definition 5.18. Let $s^\natural$ be an abstract state. We use S to denote the state graph of $\beta(s)$ for some concrete state s. Furthermore, let r, p and q be references in V_{Heap}. We say that:

- r and p *may-alias*, if $m(r) = m(p)$ for some state $s \in \gamma(s^\natural)$ and morphism $m \colon s^\natural \to \beta(s)$;
- r *may-reaches* p, if $m(r) \xrightarrow{+}_S m(p)$ for some state $s \in \gamma(s^\natural)$ and morphism $m \colon s^\natural \to \beta(s)$;

- r *may-share* with p, if there exists an $q \in s$ such that $m(r) \xrightarrow{*}_S q \; {}_S\!\xleftarrow{*} m(p)$, for some state $s \in \gamma(s^\natural)$ and morphism $m\colon s^\natural \to \beta(s)$;

- r is *maybe-cyclic*, if $m(r) \xrightarrow{+}_S m(r)$ for some state $s \in \gamma(s^\natural)$ and morphism $m\colon s^\natural \to \beta(s)$; and

- r is *acyclic*, if r is not maybe-cyclic.

Obviously these properties can not be computed in general, as $\gamma(s^\natural)$ may be infinite. Furthermore, our representation does not provide a precise approximation of these properties, as abstract variables generally also present cyclic instances. We will see that these properties are necessary to make our approach admissible. For now, we assume a preliminary analysis. In Section 8.1, we show how existing analyses can be incorporated in our approach.

5.2. Abstract Computation

In Section 4.4 we introduced the collecting semantics of JBC. In this section we define the *abstract semantics* on abstract states. For every JBC instruction f we define an *abstract instruction* $f^\natural$ and show that $f^\natural$ is an upper approximation of the set extension of f, denoted f^*. Moreover we introduce *state refinements*. State refinements allow to do case-analysis on abstract states such that an abstract state is refined into multiple but finitely many abstract states. The idea is that we perform state refinements if an abstract instruction $f^\natural$ can not be applied. Such refinements occur implicitly in control flow graphs by guarded edges, for example, when considering conditional jumps. Here we explicitly perform such refinements.

Appendix A illustrates the single-step semantics of JBC instructions. Based on these instructions, and actually mimicking them quite closely, we define how abstract states are *evaluated symbolically*. In most cases this is straightforward:

The instructions `Push`$^\natural$, `Pop`$^\natural$, `Goto`$^\natural$ and `New`$^\natural$ are defined identical. The instructions `Load`$^\natural$, `Store`$^\natural$, `Checkcast`$^\natural$ and `Return`$^\natural$ now consider abstract values rather than concrete values.

Next, consider instructions `IAdd`$^\natural$, `ISub`$^\natural$, `ICmpGte`$^\natural$, `BAnd`$^\natural$, `BOr`$^\natural$ and `BNot`$^\natural$. If the operands are concrete the instructions can be executed directly. Otherwise, we introduce a fresh variable and a side-condition mimicking the instructions. Figure 5.4 depicts the latter case for `IAdd`$^\natural$ and `BAnd`$^\natural$ formally. Here i_3 and b_3 are fresh variables. If the top element of the stack is a concrete value, `IfFalse`$^\natural$

$$\texttt{IAdd}^\natural \quad \frac{(heap, (i_2 : i_1 : stk, loc, cn, mn, pc) : frms, iu)}{(heap, (i_3 : stk, loc, cn, mn, pc + 1) : frms, iu)} \quad i_1 + i_2 = i_3$$

$$\texttt{BAnd}^\natural \quad \frac{(heap, (b_2 : b_1 : stk, loc, cn, mn, pc) : frms, iu)}{(heap, (b_3 : stk, loc, cn, mn, pc + 1) : frms, iu)} \quad b_2 \wedge b_1 \equiv b_3$$

Figure 5.4.: Symbolic evaluations of integer and Boolean operations.

can be executed directly. Otherwise we perform a *Boolean refinement*, replacing the variable with values true and false.

The instructions `Getfield`$^\sharp$, `Putfield`$^\sharp$ and `Invoke`$^\sharp$ access the object on the heap. In abstract states, elements of the heap may be class variables. Recall that a class variable cn represents null as well as instances of cn and its subtypes. Therefore, if the top of the stack is an address p and $heap(s) = cn$, then an *instance refinement* is performed.

Definition 5.19. Let $s^\sharp = (heap, frms, iu)$ be a state and let p be an address such that $heap(p) = cn'$. Let $cn \in \mathsf{subclasses}(cn')$. Furthermore, suppose $(cn_1, id_1), \ldots, (cn_n, id_n)$ denote fields of cn (together with the defining classes). We perform the following two *class instance* steps, where the second takes care of the case, where address p is replaced by null.

$$\frac{(heap, frms, iu)}{(heap\{p \mapsto (cn, ftable_1)\}, frms, iu)} \qquad \frac{(heap, frms, iu)}{(heap_2, frms_2, iu)}\ .$$

Here $ftable_1((cn_i, id_i)) := v_i$ such that the type of the abstract variable v_i is defined in correspondence to the type of field (cn_i, id_i), eg., a fresh int variable for integer fields. On the other hand we set $heap_2$ $(frms_2)$ equal to $heap$ $(frms)$, but $p \notin \mathsf{dom}(heap_2)$ and all occurrences of p are replaced by null.

Consider a `Putfield`$^\sharp$ instruction on address p. Then this operation affects also instances that reaches address p, ie., $q \rightarrow^+_{S^\sharp} p$ for some address q different from p. Therefore we additionally employ *unsharing refinements* before execution. The unsharing refinement resolves all cases where some address $q \in \mathsf{dom}(heap)$ may-alias with p. We consider the cases where $p = q$ and $p \neq q$. The key observation is that after the unsharing refinement p does not alias with other elements in the abstract heap. Hence only the object bound to p is affected by the putfield instruction abstract heap.

Definition 5.20. Let $s^\sharp = (heap, frms, iu)$ be a state. Let p and q denote different addresses in $heap$ such that $p \neq q \notin iu$. We perform the following *unsharing* steps: The first case forces these addresses to be distinct. The second case substitutes all occurrences of q with p.

$$\frac{(heap, frms, iu)}{(heap, frms, iu \cup \{p \neq q\})} \qquad \frac{(heap, frms, iu)}{(heap', frms', iu)}\ ,$$

where $heap'$ $(frms')$ is equal to $heap$ $(frms)$ with all occurrences of q replaced by p.

Finally, we are left to show the abstract computation steps for `CmpEq`$^\sharp$ and `CmpNeq`$^\sharp$. We have to consider multiple cases, depending on the values to compare. We only show the cases for `CmpEq`$^\sharp$. The cases for `CmpNeq`$^\sharp$ are similar.

1. Let val_1 and val_2 be addresses. If the addresses of val_1 and val_2 are the same then the test evaluates to true. Otherwise, we have to check if val_1 and val_2 may-alias and perform an unsharing refinement (cf. Definition 5.20) if necessary. In the latter case the test returns false.

2. Wlog. let val_1 be an address and val_2 be null. If $heap(val_1) = obj$ and $\mathsf{cl}(obj) = cn$, we perform an instance refinement according to Definition 5.19 on val_1 and re-consider the condition.

3. If val_1 and val_2 are concrete non-address Jinja values, then the test $(val_1 = val_2)$ can be directly executed and the symbolic evaluation equals the instruction on the JVM.

4. If val_1 or val_2 is an abstract Boolean or integer variable, then we introduce a new Boolean variable b_3 and the side condition $(val_1 = val_2) \equiv b_3$.

Example 5.21. In Figure 5.5 we present an example detailing the need for the given definition of class instantiation. Here class `B` overrides method `m` inherited from class `A`. We only know the static type of the parameter when analysing method `call(A a)`. Method `call(A a)` accepts any instance of class `A` or any instance of a subclass of `A` as parameter. In particular any instance of class `B`. Due to the overridden method `call(A a)` does not terminate for instances of class `B`.

```
class A{
  void m(){unit}
}
class B extends A{
  void m(){while(true)}
}
```

```
class C{
  void call(A a){a.m()}
  main(){
    C c = new C();
    c.call(new B());
  }
}
```

Figure 5.5.: All subclasses have to be considered.

Let $s^\natural$, $s^{\natural\prime}$ and $t^\natural$ be abstract states such that $s^{\natural\prime}$ is obtained by zero or multiple refinement steps from $s^\natural$. Furthermore, suppose $t^\natural$ is obtained from $s^{\natural\prime}$ due to a symbolic evaluation. Then we say $t^\natural$ is obtained form $s^\natural$ by an *abstract computation*.

Correctness. To prove correctness of an abstract computation step, we have to show that $f^*(\gamma(s^\natural)) \subseteq \gamma(f^\natural(s^\natural))$. Hence, to prove correctness of an symbolic evaluation step it is enough to show that for all $s \in \gamma(s^\natural)$ and $s \to_P t$ it follows that $t \in \gamma(t^\natural)$, where $t^\natural$ is obtained from a symbolic evaluation step, ie., $t^\natural = f^\natural(s^\natural)$. Similarly, to prove correctness of the refinement steps it is enough to show that for all $s \in \gamma(s^\natural)$ there exists a state $s_i^\natural$ obtained by a state refinement of $s^\natural$ such that $s \in \gamma(s_i^\natural)$. Correctness of an abstract computation step follows from the correctness of refinement and symbolic evaluation steps.

Lemma 5.22. *Let $s^\natural \in \mathcal{AS}$. Suppose $s_1^\natural, \ldots, s_n^\natural$ is obtained by a state refinement from $s^\natural$. Then $s^\natural \sqsupseteq s_i^\natural$ for all $s_i^\natural$. Furthermore, $s \in \gamma(s^\natural)$ implies that there exists an abstract state $s_i^\natural$ such that $s \in \gamma(s_i^\natural)$.*

Proof. The claim follows easily by the definition of Boolean and class variables, and the fact that two addresses in the heap of $s^\natural$ either alias or not. □

Lemma 5.23. *Let $s^\natural \in \mathcal{AS}$. Suppose $f^\natural(s^\natural)$ is applicable. Then $f^*(\gamma(s^\natural)) \subseteq \gamma(f^\natural(s^\natural))$, ie., for all $s \in \gamma(s^\natural)$ and $s \to_P t$ it follows that $t \in \gamma(t^\natural)$, where $t^\natural = f^\natural(s^\natural)$.*

Proof. The proof is straightforward in most cases. Let $s^\natural = (heap^\natural, frm^\natural : frms^\natural, iu)$ and $s = (heap, frm : frms)$. By assumption the domain of $frm^\natural : frms^\natural$ and $frm : frms$ coincide. We only treat some informative cases:

- Consider $\texttt{Push}^\natural\ v$. The step can be directly symbolically evaluated, as v is a concrete value.
- Consider $\texttt{Load}^\natural\ n$. By assumption $loc^\natural(n) \trianglerighteq_m loc(n)$. In the abstract computation step $loc^\natural(n)$ is loaded on to the top of the stack. Obviously $stk_i^\natural(n) \trianglerighteq_{m'} stk_i(n)$, where stk_i represents the top of the stack. Then $t \in \gamma(t^\natural)$.
- Consider $\texttt{IAdd}^\natural$. Let i_2, i_1 denote the first two stack elements of $s^\natural$. Wlog. suppose that i_1 is abstract. By definition of the symbolic evaluation of $\texttt{IAdd}^\natural$ we perform the step by introducing a new abstract integer i_3 and adding the constraint $i_3 = i_1 + i_2$. Then $t \in \gamma(t^\natural)$, since $i_3 \trianglerighteq z$ for all numbers z.
- Consider $\texttt{IfFalse}^\natural\ i$. Wlog. let false be the top element of the stack of $s^\natural$. Executing the symbolic step yields a state $t^\natural$, which is an abstraction of t by assumption on s and $s^\natural$. Then $t \in \gamma(t^\natural)$.
- Consider $\texttt{Putfield}^\natural\ fn\ cn$ on address p. By assumption the instruction can be symbolically evaluated and p does not alias with some address $q \in \mathsf{dom}(heap^\natural)$ different from p. The only interesting case to consider is when $heap^\natural(q)$ is a class variable and there exists $s \in \gamma(s^\natural)$ such that $m(q) \rightharpoonup_S r \overset{*}{\rightharpoonup}_S m(p)$, where $r \in \mathsf{dom}(heap)$. Then $m(q)$ reaches $m(p)$ via r and is affected by the update instruction. This does not matter, since $heap^\natural(q)$ is also a class variable in $t^\natural$, thus also representing the affected instance. Then $t \in \gamma(t^\natural)$.
- Consider $\texttt{CmpEq}^\natural$. By assumption the instruction can be symbolically executed. That is the necessary refinement steps are already performed. Then $t \in \gamma(t^\natural)$ follows directly.

□

The next theorem is an immediate result of the Lemma 5.22 and Lemma 5.23.

Theorem 5.24. *Let s and t be JBC states such that $s \to_P^* t$. Suppose $s \in \gamma(s^\natural)$ for some abstract state $s^\natural$. Then there exists an abstract computation from $s^\natural$ to $t^\natural$ such that $t \in \gamma(t^\natural)$.*

Theorem 5.24 formally proves the correctness of the proposed abstract domain with respect to the operational semantics for Jinja, established by Klein and Nipkow [30]. In order to exploit this abstract domain we require a finite representation of the abstract domain $\mathcal{AS}$ induced by P. For that we propose computation graphs as finite representations of all relevant states in $\mathcal{AS}$, abstracting JBC states in P.

5.3. Computation Graphs

In this section, we define *computation graphs* as *finite* representation of all program traces of P. We recall the standard approach to data flow analysis as presented in Section 2.2: (i) computation of the control flow graph of P; (ii) mapping functions f_l to labels; (iii) computation of a fixed point from an initial state of the abstract domain. Computation graphs unify the above mentioned approach: The abstract domain is the set of all abstract states. Nodes in the computation graph are also abstract states. Starting from an initial abstract state, states are dynamically expanded through abstract computation. We obtain a finite control flow graph and a fixed point over the abstract domain by repeatedly expanding the graph via abstract computation and suitably merging states having the same program location.

Definition 5.25. A *computation graph* $G = (V_G, E_G)$ is a directed graph with edge labels, where $V_G \subset \mathcal{AS}$ and $s^\sharp \xrightarrow{\ell} t^\sharp \in E_G$ if either $s^\sharp$ is obtained from $t^\sharp$ by an abstract computation or $s^\sharp$ is an instance of $t^\sharp$. Furthermore, if there exists a constraint C in the symbolic evaluation, then $\ell := C$. For all other cases $\ell := \varnothing$. We say that G is the computation graph of program P if for all initial states i of P there exists an abstract state $i^\sharp \in G$ such that $i \in \gamma(i^\sharp)$.

We obtain a finite representation of loops, if we suitably exploit the fact that any subset of $\mathcal{AS}$ has a least upper bound and $(\mathcal{AS}, \sqsubseteq)$ satisfies the ascending chain condition. The intuition is best conveyed by an example.

Example 5.26. Consider the `List` program from Example 3.1 and the corresponding bytecode from Example 3.4. Figure 5.6 illustrates the computation graph of `append`. For the sake of readability we omit the *val* field of the list, the unsharing annotations and some intermediate nodes.

Consider the initial node I. It is easy to see that I is an abstraction of all concrete initial states, when *this* is not null. We assume that *this* is acyclic and initially do not share with *ys*. Nodes A, B and S correspond to the situation described in Example 5.5 and Example 5.8. That is, node A is obtained after assigning *cur* to *this* before any iteration of the loop, node B is obtained after exactly one iteration of the loop and node $S = \bigsqcup\{A, B\}$. Intermediate iterations are normally removed. This is indicated by a dashed border for B.

After pushing the reference of *cur.next* and null onto the operand stack, we reach node C. At `pc` $= 7$ we want to compare the reference of *cur.next* with null. But, *cur.next* is not concrete. Therefore, a class instance refinement is performed, yielding nodes C_1 and C_2.

First, we consider that *cur.next* is not null, but references an arbitrary instance, as illustrated in node C_1. The step from C_1 to D is trivial. Let *id* denote the identity function and $m = id(V_S)$. Then $m\{o_4 \mapsto o_5, o_5 \mapsto o_6\}$ is a morphism from S to D. Therefore, D is an instance of S. Second, we consider the case when *cur.next* is null, as depicted in node C_2. Node E is obtained from C_2 after loading registers *cur* and *ys* onto the stack. At program counter 19 a `Putfield`$^\sharp$ instruction is performed. Therefore we perform a refinement according to Definition 5.20. We obtain nodes E_1, E_2 and E_3. In E_1, *this* and

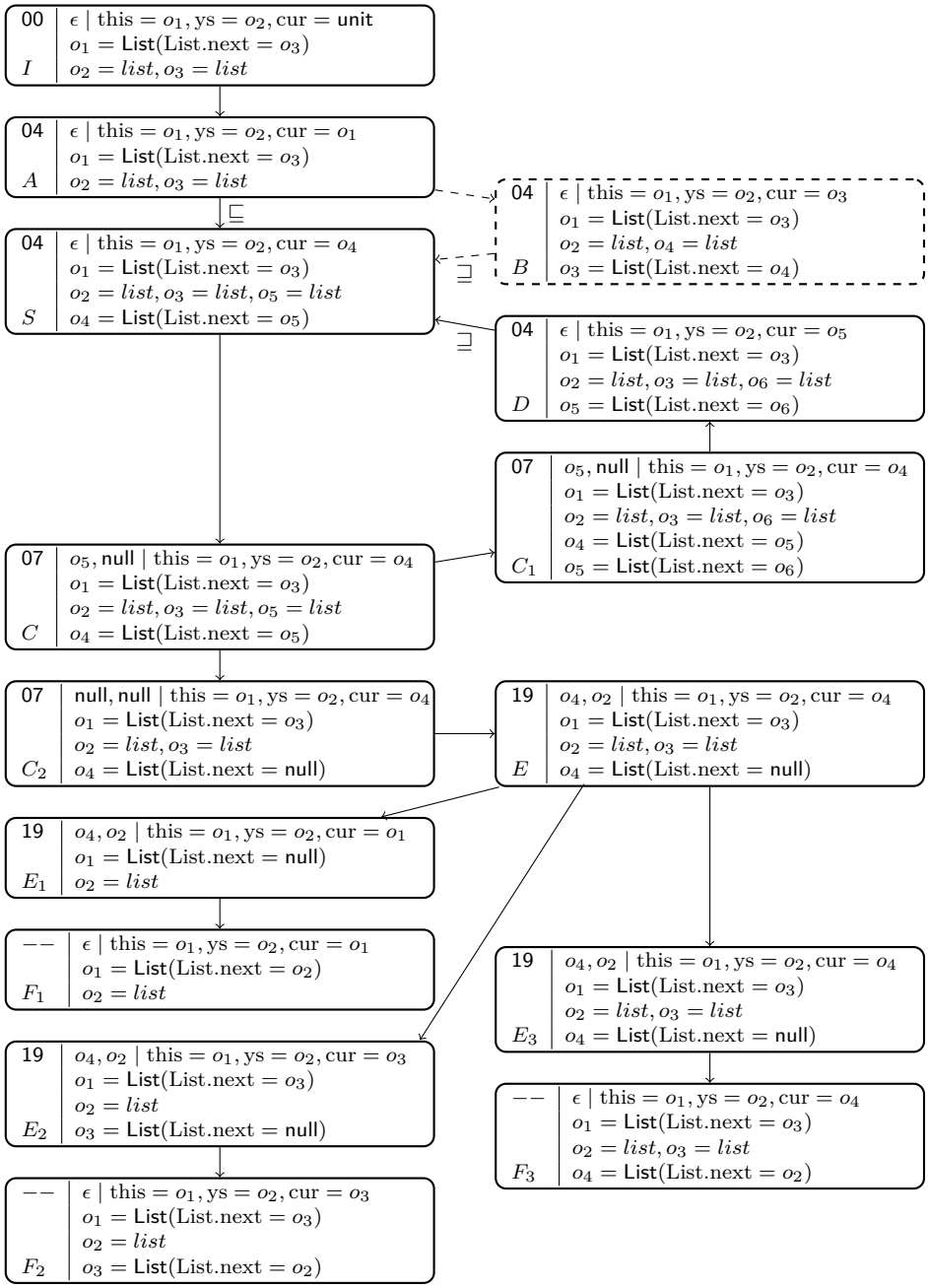

Figure 5.6.: The (incomplete) computation graph of `append`.

cur point to the same reference, in E_2 *this.next* and *cur* point to the same reference, and in E_3 the abstracted part from *cur* is distinct from *this*, yet *this* and *cur* shares. Nodes F_1, F_2 and F_3 are obtained after performing the `Putfield`$^\natural$ instruction.

The complete graph that is automatically generated from the prototype implementation, which we present in Chapter 8, consists of 38 nodes.

To concretise the employed strategy, note that whenever we are about to

finish a loop, we attempt to use an instance refinement to the state starting this loop. If this fails, for example in an attempted step from B to A in Example 5.26, we join the corresponding states. Here we collect all states that need to be abstracted and join them to obtain an abstraction. Complementing the proposed strategy, we restrict the applications of state refinements suitably, such that these refinement steps are only performed if no other steps are applicable. The next lemma shows that if this strategy is followed we are guaranteed to obtain a *finite* computation graph.

Lemma 5.27. *Let G be the computation graph of a program P such that in the construction of G the above strategy is applied. Then G is finite.*

Proof. We argue indirectly. Suppose the computation graph G of P is infinite. This is only possible if there exists an initial state i of P that is non-terminating, which implies that starting from i we reach a loop in P that is called infinitely often. As G is infinite this implies that the join operation for this loop gives rise to an infinite sequence of states $(s_j^\natural)_{j\geqslant 0}$ such that $s_j^\natural \sqsubset s_{j+1}^\natural$ for all j. However, this is impossible as any ascending chain of abstract states eventually stabilises, cf. Lemma 5.14. □

Let G be a computation graph. We write $s^\natural \rightharpoonup_G t^\natural$ to indicate that state $t^\natural$ is directly reachable in G from $s^\natural$. Sometimes we want to distinguish whether $t^\natural$ is obtained by a refinement (denoted as $s^\natural \rightharpoonup_{\text{ref}} t^\natural$) or by a symbolic evaluation (denoted as $s^\natural \rightharpoonup_{\text{eva}} t^\natural$), or whether $s^\natural$ is an instance of $t^\natural$ (denoted as $s^\natural \rightharpoonup_{\text{ins}} t^\natural$). If $t^\natural$ is reachable from $s^\natural$ in G we write $s^\natural \rightharpoonup_G^* t^\natural$. If $s^\natural \neq t^\natural$ this is denoted by $s^\natural \rightharpoonup_G^+ t^\natural$.

Lemma 5.28. *Let $s, t \in \mathcal{JS}$ such that $s \to_P t$. Let G denote the computation graph of P. Suppose $s^\natural \in G$ such that $s \in \gamma(s^\natural)$, then there exists $t^\natural \in G$ such that $t \in \gamma(t^\natural)$ and a path $s^\natural \rightharpoonup_{ins}^* \cdot \rightharpoonup_{ref}^* \cdot \rightharpoonup_{eva} t^\natural$.*

Proof. By construction of G we have to consider two cases: Suppose $t^\natural$ is obtained by an abstract computation from $s^\natural$. We employ Lemma 5.23 to conclude that $t \in \gamma(t^\natural)$. Then $s^\natural \rightharpoonup_{\text{ref}}^* \cdot \rightharpoonup_{\text{eva}} t^\natural$. Next, suppose $t^\natural$ is obtained by an abstract computation from $s^{\natural\prime}$, where $s^\natural \sqsubseteq s^{\natural\prime}$. Hence, we also have $s \in \gamma(s^{\natural\prime})$. We employ Lemma 5.23 to conclude that $t \in \gamma(t^\natural)$. Then $s^\natural \rightharpoonup_{\text{ins}}^* \cdot \rightharpoonup_{\text{ref}}^* \cdot \rightharpoonup_{\text{eva}} t^\natural$. Since G is finite we conclude that $s^\natural \rightharpoonup_{\text{ins}}^* \cdot \rightharpoonup_{\text{ref}}^* \cdot \rightharpoonup_{\text{eva}} t^\natural$ has finitely many instance and refinement steps, only depending on G. □

We arrive at the main result of this section.

Theorem 5.29. *Let $i, t \in \mathcal{JS}$ and suppose $i \to_P^* t$, where the runtime of the execution is m. Let G denote the computation graph of P. Suppose $i^\natural \in G$ such that $i \in \gamma(i^\natural)$, then there exists an abstraction $t^\natural \in G$ and a path $i^\natural \rightharpoonup_G^* t^\natural$ of length m' such that $m \leqslant m' \leqslant K \cdot m$. Here constant $K \in \mathbb{N}$ only depends on G.*

Proof. By induction on m (employing Lemma 5.28), we conclude the existence of state $t^\natural$ such that $i^\natural \rightharpoonup_G^* t^\natural$. Hence, the first part of the theorem follows. Furthermore by Lemma 5.28 there exists m' such that $m \leqslant m' \leqslant K \cdot m$. □

Corollary 5.30. *Let P be a program and $\mathcal{S} \subseteq \mathcal{IS}$. Suppose computation graph G is obtained from initial state $\alpha(\mathcal{S})$. We have $\gg \cdot \to_{P|_{\mathcal{S}}} \subseteq \rightharpoonup^*_{ins} \cdot \rightharpoonup^*_{ref} \cdot \rightharpoonup_{eva} \cdot \gg$ and $\alpha(\mathcal{S}) \gg s$ for all $s \in \mathcal{S}$, applying Lemma 2.18.*

It is easy to see that the obtained abstraction is complexity preserving as the size of abstract states are bounded by the size of some concrete state by construction. Yet, the existence of cycles in the computation graph implies that the relation of the abstraction is not well-founded as arbitrary long paths can be constructed. In the next chapter we see how computation graphs give rise to a term-based abstraction.

6. Abstract Term Domain

In this chapter we present the transformation from computation graphs to rewrite systems and provide the main result of this thesis. In Section 6.1 we introduce *constrained term rewrite systems* as a variant of term rewrite systems. In Section 6.2 we show how the rewrite rules are obtained from the computation graph. We construct a Galois connection from set of concrete states to set of terms and show that the rewrite relation obtained from the transformation is an upper approximation of all program executions.

6.1. Constrained Term Rewrite Systems

Let G be the computation graph for program P with initial state $i^\sharp$. G is kept fixed for the remainder of this chapter. In the following we describe the translation from G into a *constrained term rewrite system* (*cTRS* for short). Our definition is a variation of cTRSs as for example defined by Falke and Kapur [16, 17]. Recently, Kop and Nishida introduced a very general formalism of term rewrite systems with constraints, termed *logical constrained term rewrite systems (LCTRSs)* [31]. The proposed notion of cTRSs is not directly interchangeable with LCTRSs, yet the rewrite system resulting from the transformation could also be formalised as LCTRS.

Let $\mathcal{C}$ be a (not necessarily finite) sorted signature and let $\mathcal{V}'$ denote a countably infinite set of sorted variables. Furthermore, let T denote a theory over $\mathcal{C}$. Quantifier-free formulas over $\mathcal{C}$ are called *constraints*. Suppose $\mathcal{F}$ is a sorted signature that extends $\mathcal{C}$ and let $\mathcal{V} \supseteq \mathcal{V}'$ denote an extension of the variables in $\mathcal{V}'$. Let $\mathcal{T}(\mathcal{F}, \mathcal{V})$ denote the set of *(sorted) terms* over the signature $\mathcal{F}$ and $\mathcal{V}$. Note that the sorted signature is necessary to distinguish between *theory* variables that are to be interpreted over the theory T and *term* variables whose interpretation is free. A *constrained rewrite rule*, denoted as $l \to r \,[\![C]\!]$, is a triple consisting of terms l and r, together with a constraint C. We assert that $l \notin \mathcal{V}$, but do *not* require that $\mathsf{Var}(l) \supseteq \mathsf{Var}(r) \cup \mathsf{Var}(C)$, where $\mathsf{Var}(t)$ ($\mathsf{Var}(C)$) denotes the variables occurring in the term t (constraint C). A *constrained term rewrite system* is a finite set of constrained rewrite rules.

Let $\mathcal{R}$ denote a cTRS. A context D is a term with exactly one occurrence of a *hole* $\Box$, and $D[t]$ denotes the term obtained by replacing the hole $\Box$ in D by the term t. A substitution σ is a function that maps variables to terms, and $t\sigma$ denotes the homomorphic extension of this function to terms. We define the rewrite relation $\to_\mathcal{R}$ as follows. For terms s and t, $s \to_\mathcal{R} t$ holds, if there exists a context D, a substitution σ and a constrained rule $l \to r \,[\![C]\!] \in \mathcal{R}$ such that $s =_T D[l\sigma]$ and $t = D[r\sigma]$ with $T \vdash C\sigma$. Here $=_T$ denotes unification modulo T. For extra variables x, possibly occurring in t, we demand that $\sigma(x)$ is in normal-form.

We often drop the reference to the cTRS $\mathcal{R}$, if no confusion can arise from this. A function symbol in $\mathcal{F}$ is called *defined* if f occurs as the root symbol of l, where $l \to r \,[\![C]\!] \in \mathcal{R}$. Symbols in $\mathcal{C}$ are called *theory symbols* and function symbols in $\mathcal{F} \setminus \mathcal{C}$ that are not defined, are called *constructor* symbols.

A cTRS $\mathcal{R}$ is called *terminating*, if the relation $\to_{\mathcal{R}}$ is well-founded. For a terminating cTRS $\mathcal{R}$, we define its *runtime complexity*, denoted by rctrs. We adapt the runtime complexity with respect to a standard TRS suitable for cTRS $\mathcal{R}$. (See [25] for the standard definition.) The *derivation height* of a term t (with respect to $\mathcal{R}$) is defined as the maximal length of a derivation (with respect to $\mathcal{R}$) starting in t. The derivation height of t is denoted by $\mathsf{dh}(t)$. Note that $\to_{\mathcal{R}}$ is not necessarily finitely branching for finite cTRSs, as fresh variables on the right-hand side of a rule can occur.

Definition 6.1. We define the *runtime complexity* (with respect to $\mathcal{R}$) as follows:

$$\mathsf{rctrs}(n) =_{\mathsf{k}} \mathsf{max}\{\mathsf{dh}(t) \mid t \text{ is basic and } \|t\| \leqslant n\} \,,$$

where a term $t = f(t_1, \ldots, t_k)$ is called *basic* if f is defined, and the terms t_i are only built over constructor, theory symbols, and variables. We fix the size measure $\|\cdot\|$ below.

In the following we are only interested in cTRS over a specific theory T, namely Presburger arithmetic. That is, we have $T \vdash C$, if all ground instances of the constraint C are valid in Presburger arithmetic. Recall, that Presburger arithmetic is decidable. If $T \vdash C$, then C is *valid*. On the other hand, if there exists a substitution σ such that $T \vdash C\sigma$, then C is *satisfiable*.

To represent the basic operations in the Jinja bytecode instruction set (cf. Definition 3.3) we collect the following connectives and truth constants in $\mathcal{C}$: $\wedge$, $\vee$, $\neg$, true, and false, together with the following relations and operations: $=$, $\neq$, $\geqslant$, $+$, $-$. Furthermore, we add infinitely many constants to represent integers. We often write $l \to r$ instead of $l \to r \,[\![\mathsf{true}]\!]$. As expected $\mathcal{C}$ makes use of two sorts: bool and int. We suppose that all abstract variables $X_1, X_2, \ldots$ are present in the set of variables $\mathcal{V}$, where abstract integer (Boolean) variables are assigned sort int (bool) and all other variables are assigned sort univ. The remaining elements of the signature $\mathcal{F}$ will be defined in the course of this section. As the signature of these function symbols is easily read off from the translation given below, in the following the sort information is left implicit to simplify the presentation.

The size of a term t, denoted by $\|t\|$ is defined as follows:

$$\|t\| := \begin{cases} 1 & \text{if } t \text{ is a variable} \\ \mathsf{abs}(t) & \text{if } t \text{ is an integer} \\ 1 + \sum_{i=1}^{n} \|t_i\| & \text{if } t = f(t_1, \ldots, t_n) \text{ and } f \text{ is not an integer} \,. \end{cases}$$

6.2. CTRS Transformation

In the next definition, we show how program states become representable as terms over $\mathcal{F}$.

Definition 6.2. Let $s^\natural = (heap, frms, iu)$ be a state and let the index sets Stk and Loc be defined as above. Suppose v is a value. Then the value v is translated as follows:

$$\mathsf{tval}(v) := \begin{cases} \mathsf{null} & \text{if } v \in \{\mathsf{unit}, \mathsf{null}\} \\ v & \text{if } v \text{ is a non-address value, except } \mathsf{unit} \text{ or } \mathsf{null} \\ \mathsf{taddr}(v) & \text{if } v \text{ is an address .} \end{cases}$$

Let a be an address. Then a is translated as follows:

$$\mathsf{taddr}(a) := \begin{cases} x & \text{if } a \text{ is maybe-cyclic, } x \text{ is fresh} \\ x & \text{if } heap(a) \text{ is a class variable } x \\ cn(\mathsf{tval}(v_1), \ldots, \mathsf{tval}(v_n)) & \text{if } heap(a) = (cn, ftable) \text{ .} \end{cases}$$

Here we suppose in the last case that $\mathsf{dom}(ftable) = \{(cn_1, id_1), \ldots, (cn_n, id_n)\}$ and for all $1 \leqslant i \leqslant n$: $ftable((cn_i, id_i)) = v_i$. Finally, to translate the state $s^\natural$ into a term, it suffices to translate the values of the registers and the operand stacks of all frames in the list $frms$. Let $(stk, i, j) \in Stk$ such that $stk_i(j)$ denotes the j^{th} value in the operation stack of the i^{th} frame in $frms$. Similarly for $(loc, i', j') \in Loc$. Then we set

$$\mathsf{ts}(s) := [\mathsf{tval}(stk_1(1)), \ldots, \mathsf{tval}(stk_k(|stk_k|))), \mathsf{tval}(loc_1(1)), \ldots, \mathsf{tval}(loc_k(|loc_k|))]$$

where the list $[\ldots]$, is formalised by an auxiliary binary symbol : and the constant nil.

Example 6.3 (continued from Example 5.26)**.** Consider the simplified presentation of state C in Figure 5.6. Then $\mathsf{ts}(C)$ yields following term:

$$\mathsf{ts}(C) = [list5, \mathsf{null}, \mathsf{List}(list3), list2, \mathsf{List}(\mathsf{List}(list5))] \text{ .}$$

This transformation immediately gives rise to a Galois connection between set of states and set of terms. Let $\mathcal{P}(\mathcal{T}(\mathcal{F}, \mathcal{V})) := (\mathcal{P}(\mathcal{T}(\mathcal{F}, \mathcal{V})), \subseteq, \cup, \cap, \varnothing, \mathcal{T}(\mathcal{F}, \mathcal{V}))$ be the complete lattice of terms over signature $\mathcal{F}$ ordered by set inclusion. Recall Definition 5.15, where we defined $\beta\colon \mathcal{JS} \to \mathcal{AS}$. We set $\eta(s) := \mathsf{ts}(\beta(s))$. Then $\alpha\colon \mathcal{P}(\mathcal{JS}) \to \mathcal{P}(\mathcal{T}(\mathcal{F}, \mathcal{V}))$ and $\gamma\colon \mathcal{P}(\mathcal{T}(\mathcal{F}, \mathcal{V})) \to \mathcal{P}(\mathcal{JS})$ are the abstraction function and the concretisation function induced by η (cf. Definition 2.14). Hence, $(\mathcal{P}(\mathcal{JS}), \alpha, \gamma, \mathcal{P}(\mathcal{T}(\mathcal{F}, \mathcal{V})))$ is a Galois connection. Observe that our term representation can only fully represent acyclic data: Consider the translation of state C of the previous example. Suppose that *this* is initially cyclic, then *this* and *cur* are cyclic in C and we would have: $\mathsf{ts}(C) = [list8, \mathsf{null}, list6, list2, list7]$. However, we still obtain following lemma.

Lemma 6.4. *Let $s^\natural$ and $t^\natural$ be abstract states. If $t^\natural \sqsubseteq s^\natural$, then there exists a substitution σ such that $\mathsf{ts}(t^\natural) = \mathsf{ts}(s^\natural)\sigma$.*

Proof. Let $S^\natural$ and $T^\natural$ be the state graphs of $s^\natural$ and $t^\natural$, respectively. By assumption there exists a morphism $m\colon S^\natural \to T^\natural$. The lemma is a direct consequence of the following observations:

- Consider the terms $\mathsf{ts}(s^\natural)$ and $\mathsf{ts}(t^\natural)$. By definition these terms encode the standard term representations of the graphs $S^\natural$ and $T^\natural$.
- Let u and v be nodes in $S^\natural$ and $T^\natural$ such that $m(u) = v$. The label of u (in $S^\natural$) can only be distinct from the label of v (in $T^\natural$), if $L_{S^\natural}(u)$ is an abstract variable or null. In the former case $\mathsf{tval}(L_{S^\natural}(u))$ is again a variable and the latter case implies that $L_{T^\natural}(v) = \mathsf{unit}$. Thus in both cases, $\mathsf{tval}(L_{S^\natural}(u))$ matches $\mathsf{tval}(L_{T^\natural}(v))$.
- By correctness of our abstraction, we have $m(u)$ is maybe-cyclic, if v is maybe-cyclic. In this case $\mathsf{tval}(L_{S^\natural}(u))$ and $\mathsf{tval}(L_{T^\natural}(v))$ are fresh variables. Hence, $\mathsf{tval}(L_{S^\natural}(u))$ matches $\mathsf{tval}(L_{T^\natural}(v))$

□

The next lemma relates the size of a state to its term representation and vice versa.

Lemma 6.5. *Let $s = (heap, frms)$ be a state such that heap does not admit cyclic data. Then $\|\mathsf{ts}(\beta(s))\| = |s|$.*

Proof. As a consequence of Definition 4.6 and the above proposed variant of the term complexity we see that $\|\mathsf{ts}(\beta(s))\| = |s|$ for all states s. □

Lemma 6.6. *Let $s = (heap, frms)$ be a state such that heap may contain cyclic data. Then $\|\mathsf{ts}(\beta(s))\| \leqslant |s|$ and therefore $\|\mathsf{ts}(\beta(s))\| \in O(|s|)$.*

Proof. Follows from the previous lemma and the fact that addresses bounded to cyclic data are replaced by fresh variables. □

Let G be a computation graph. For any state $s^\natural$ in G we introduce a new function symbol $\mathsf{f}_{s^\natural}$. Suppose $\mathsf{ts}(s^\natural) = [s_1^\natural, \ldots, s_n^\natural]$. To ease presentation we write $\mathsf{f}_{s^\natural}(\mathsf{ts}(s^\natural))$ instead of $\mathsf{f}_{s^\natural}(s_1^\natural, \ldots, s_n^\natural)$.

Definition 6.7. Let G be a finite computation graph and $s^\natural = (heap, frms, iu)$ and $t^\natural$ be states in G. We define the constrained rule *corresponding* to the edge $(s^\natural, t^\natural)$, denoted by $\mathsf{rule}(s^\natural, t^\natural)$, as follows:

$$\mathsf{rule}(s^\natural, t^\natural) = \begin{cases} \mathsf{f}_{s^\natural}(\mathsf{ts}(s^\natural)) \to \mathsf{f}_{t^\natural}(\mathsf{ts}(s^\natural)) & \text{if } s^\natural \sqsubseteq t^\natural \\ \mathsf{f}_{s^\natural}(\mathsf{ts}(t^\natural)) \to \mathsf{f}_{t^\natural}(\mathsf{ts}(t^\natural)) & \text{if } t^\natural \text{ is a state refinement of } s^\natural \\ \mathsf{f}_{s^\natural}(\mathsf{ts}(s^\natural)) \to \mathsf{f}_{t^\natural}(\mathsf{ts}(t^\natural))\,[\![\mathsf{tval}(C)]\!] & \text{the edge is labelled by } C \\ \mathsf{f}_{s^\natural}(\mathsf{ts}(s^\natural)) \to \mathsf{f}_{t^\natural}(\mathsf{ts}^*(t^\natural)) & s^\natural \text{ corresponds to a } \mathtt{Putfield}^\natural \text{ on address } p,\ heap(q) \text{ is variable } cn, \text{ and } q \text{ may-reach } p \\ \mathsf{f}_{s^\natural}(\mathsf{ts}(s^\natural)) \to \mathsf{f}_t(\mathsf{ts}(t^\natural)) & \text{otherwise .} \end{cases}$$

Here $\mathsf{tval}(C)$ denotes the standard extension of the mapping tval to labels of edges and ts^* is defined as ts but employs fresh variables for any reference q that may-reach the object that is updated. The cTRS obtained from G consists of rule $\mathsf{rule}(s^\natural, t^\natural)$ for all edges $s^\natural \to t^\natural \in G$.

By construction G is finite. We comment on $\mathsf{rule}(s^\sharp, t^\sharp)$: If $s^\sharp \sqsubseteq t^\sharp$, then $s^\sharp$ is more precise than $t^\sharp$. We want to keep this information during transformation. If $t^\sharp$ is a state refinement of $s^\sharp$, then a pattern matching on $t^\sharp$ is performed. If the edge from $s^\sharp$ to $t^\sharp$ is labelled by some constraint C. Then $\mathsf{f}_{s^\sharp}$ corresponds to an operation and the right hand side is updated. When $\mathsf{f}_{s^\sharp}$ is a $\texttt{Putfield}^\sharp$, we additionally have to incorporate side-effects resulting from the update.

Example 6.8 (continued from Example 5.26)**.** Figure 6.1 illustrates the cTRS obtained from the computation graph. We use following conventions: L denotes the list constructor symbol and l followed by a number a list variable. In the last rule $l4$ is fresh on the right-hand side. This is because we update *cur* and have a side-effect on *this* that is not directly observable in the abstraction.

$$
\begin{aligned}
\mathsf{f}_I(\mathsf{L}(l3), l2, \mathsf{null}) &\to \mathsf{f}_A(\mathsf{L}(l3), l2, \mathsf{L}(l3))\\
\mathsf{f}_A(\mathsf{L}(l3), l2, \mathsf{L}(l3)) &\to \mathsf{f}_S(\mathsf{L}(l3), l2, \mathsf{L}(l3))\\
\mathsf{f}_S(\mathsf{L}(l3), l2, \mathsf{L}(l5)) &\to \mathsf{f}_C(l5, \mathsf{null}, \mathsf{L}(l3), l2, \mathsf{L}(l5))\\
\mathsf{f}_C(\mathsf{L}(l6), \mathsf{null}, \mathsf{L}(l3), l2, \mathsf{L}(\mathsf{L}(l6))) &\to \mathsf{f}_{C_1}(\mathsf{L}(l6), \mathsf{null}, \mathsf{L}(l3), l2, \mathsf{L}(\mathsf{L}(l6)))\\
\mathsf{f}_C(\mathsf{null}, \mathsf{null}, \mathsf{L}(l3), l2, \mathsf{L}(\mathsf{null})) &\to \mathsf{f}_{C_2}(\mathsf{null}, \mathsf{null}, \mathsf{L}(l3), l2, \mathsf{L}(\mathsf{null}))\\
\mathsf{f}_{C_1}(\mathsf{L}(l6), \mathsf{null}, \mathsf{L}(l3), l2, \mathsf{L}(\mathsf{L}(l6))) &\to \mathsf{f}_D(\mathsf{L}(l3), l2, \mathsf{L}(l6))\\
\mathsf{f}_D(\mathsf{L}(l3), l2, \mathsf{L}(l6)) &\to \mathsf{f}_S(\mathsf{L}(l3), l2, \mathsf{L}(l6))\\
\mathsf{f}_{C_2}(\mathsf{null}, \mathsf{null}, \mathsf{L}(l3), l2, \mathsf{L}(\mathsf{null})) &\to \mathsf{f}_E(\mathsf{L}(\mathsf{null}), l2, \mathsf{L}(l3), l2, \mathsf{L}(\mathsf{null}))\\
\mathsf{f}_E(\mathsf{L}(\mathsf{null}), l2, \mathsf{L}(\mathsf{null}), l2, \mathsf{L}(\mathsf{null})) &\to \mathsf{f}_{E_1}(\mathsf{L}(\mathsf{null}), l2, \mathsf{L}(\mathsf{null}), l2, \mathsf{L}(\mathsf{null}))\\
\mathsf{f}_{E_1}(\mathsf{L}(\mathsf{null}), l2, \mathsf{L}(\mathsf{null}), l2, \mathsf{L}(\mathsf{null})) &\to \mathsf{f}_{F_1}(\mathsf{L}(l2), l2, \mathsf{L}(l2))\\
\mathsf{f}_E(\mathsf{L}(\mathsf{null}), l2, \mathsf{L}(\mathsf{L}(\mathsf{null})), l2, \mathsf{L}(\mathsf{null})) &\to \mathsf{f}_{E_2}(\mathsf{L}(\mathsf{null}), l2, \mathsf{L}(\mathsf{L}(\mathsf{null})), l2, \mathsf{L}(\mathsf{null}))\\
\mathsf{f}_{E_2}(\mathsf{L}(\mathsf{null}), l2, \mathsf{L}(\mathsf{L}(\mathsf{null})), l2, \mathsf{L}(\mathsf{null})) &\to \mathsf{f}_{F_2}(\mathsf{L}(\mathsf{L}(l2)), l2, \mathsf{L}(l2))\\
\mathsf{f}_E(\mathsf{L}(\mathsf{null}), l2, \mathsf{L}(l3), l2, \mathsf{L}(\mathsf{null})) &\to \mathsf{f}_{E_3}(\mathsf{L}(\mathsf{null}), l2, \mathsf{L}(l3), l2, \mathsf{L}(\mathsf{null}))\\
\mathsf{f}_{E_3}(\mathsf{L}(\mathsf{null}), l2, \mathsf{L}(l3), l2, \mathsf{L}(\mathsf{null})) &\to \mathsf{f}_{F_3}(\mathsf{L}(l4), l2, \mathsf{L}(l2))
\end{aligned}
$$

Figure 6.1.: The cTRS of `append`.

Example 6.9. Figure 1.2 illustrates the cTRS obtained of the program from Figure 1.1 and shows how constraints are integrated; irrelevant intermediate nodes are omitted.

In the following we show that the rewrite relation of the obtained cTRS safely approximates the concrete semantics of the concrete domain. We first argue informally:

- By Lemma 5.28 there exists a path $s^\sharp \rightharpoonup^*_{\text{ins}} \cdot \rightharpoonup^*_{\text{ref}} \cdot \rightharpoonup_{\text{eva}} t^\sharp$ in G for $s \to_P t$ such that $s \in \gamma(s^\sharp)$ and $t \in \gamma(t^\sharp)$.
- Together with Lemma 6.4 we have to show that $\mathsf{f}_{s^\sharp}(\mathsf{ts}(s')) \to^+_{\mathcal{R}} \mathsf{f}_{t^\sharp}(\mathsf{ts}(t'))$, for $s' = \beta(s)$ and $t' = \beta(t)$.

- We do this by inspecting the rules obtained from the transformation. We will see that instance steps and refinement steps do not modify the term instance. In case of evaluation steps the effect is either directly observable in the abstract state, as it happens for $\texttt{Push}^\sharp$ for example, or indirectly by requiring that the substitution is conform with the constraint. In the case of the $\texttt{Putfield}^\sharp$ instructions we have to find a suitable substitution for fresh variables to accommodate possible side-effects.

Lemma 6.10. *Let $s^\sharp$ and $t^\sharp$ be states in G connected by an edge $s^\sharp \xrightarrow{\ell} t^\sharp$ from $s^\sharp$ to $t^\sharp$. Suppose $s \in \mathcal{JS}$ with $s \in \gamma(s^\sharp)$. Suppose further that if the constraint ℓ labelling the edge is non-empty, then s satisfies ℓ. Moreover, if $s^\sharp \xrightarrow{\ell} t^\sharp$ follows due to a refinement step, then s is consistent with the chosen refinement. Then there exists $t \in \gamma(t^\sharp)$ such that $\mathsf{f}_{s^\sharp}(\mathsf{ts}(s')) \to_{\mathsf{rule}(s^\sharp,t^\sharp)} \mathsf{f}_{t^\sharp}(\mathsf{ts}(t'))$ with $s' = \beta(s)$, $t' = \beta(t)$.*

Proof. The proof proceeds by case analysis on the edge $s^\sharp \xrightarrow{\ell} t^\sharp$ in G, where we only need to consider the following four cases. The argument for the omitted fifth case is very similar to the third case.

- *Case* $s^\sharp \xrightarrow{\ell} t^\sharp$, as $s^\sharp \sqsubseteq t^\sharp$; $\ell = \varnothing$. By assumption $s' \sqsubseteq s^\sharp \sqsubseteq t^\sharp$. Hence, $s \in \gamma(t^\sharp)$ by transitivity of the instance relation. By Lemma 6.4 there exists a substitution σ such that $\mathsf{ts}(s') = \mathsf{ts}(s^\sharp)\sigma$. In sum, we obtain:

$$\mathsf{f}_{s^\sharp}(\mathsf{ts}(s')) = \mathsf{f}_{s^\sharp}(\mathsf{ts}(s^\sharp))\sigma \to_{\mathsf{rule}(s^\sharp,t^\sharp)} \mathsf{f}_{t^\sharp}(\mathsf{ts}(s^\sharp))\sigma = \mathsf{f}_{t^\sharp}(\mathsf{ts}(t')) \ ,$$

  where we set $t' := s'$.

- *Case* $s^\sharp \xrightarrow{\ell} t^\sharp$, as $t^\sharp$ is a refinement of $s^\sharp$; $\ell = \varnothing$. By assumption $s' \sqsubseteq s^\sharp$ and s is concrete. Hence, $s' \sqsubseteq t^\sharp$ by definition of $t^\sharp$. Again by Lemma 6.4 there exists a substitution σ, such that $\mathsf{ts}(s') = \mathsf{ts}(t^\sharp)\sigma$. In sum, we obtain:

$$\mathsf{f}_{s^\sharp}(\mathsf{ts}(s')) = \mathsf{f}_{s^\sharp}(\mathsf{ts}(t^\sharp))\sigma \to_{\mathsf{rule}(s^\sharp,t^\sharp)} \mathsf{f}_{t^\sharp}(\mathsf{ts}(t^\sharp))\sigma = \mathsf{f}_{t^\sharp}(\mathsf{ts}(t')) \ ,$$

  where we again set $t' := s'$.

- *Case* $s^\sharp \xrightarrow{\ell} t^\sharp$, as $t^\sharp$ is the result of the symbolic evaluation of $s^\sharp$ and $\ell = C \neq \varnothing$. By assumption s satisfies the constraint C. More precisely, there exists a substitution σ such that $\mathsf{ts}(s') = \mathsf{ts}(s^\sharp)\sigma$ and $T \vdash \mathsf{tval}(C)\sigma$. We obtain:

$$\mathsf{f}_{s^\sharp}(\mathsf{ts}(s')) = \mathsf{f}_{s^\sharp}(\mathsf{ts}(s^\sharp))\sigma \to_{\mathsf{rule}(s^\sharp,t^\sharp)} \mathsf{f}_{t^\sharp}(\mathsf{ts}(t^\sharp))\sigma \ .$$

  Let t be defined such that $s \to_P t$. By Lemma 5.23 we obtain $t' \sqsubseteq t^\sharp$ and by inspection of the proof of Lemma 5.23 we observe that $\mathsf{ts}(t') = \mathsf{ts}(t^\sharp)\sigma$. In sum, $\mathsf{f}_{s^\sharp}(\mathsf{ts}(s')) \to_{\mathsf{rule}(s^\sharp,t^\sharp)} \mathsf{f}_{t^\sharp}(\mathsf{ts}(t'))$.

- *Case* $s^\sharp \xrightarrow{\ell} t^\sharp$, as $t^\sharp$ is the result of a $\texttt{Putfield}^\sharp$ instruction on p and there exists an address q in $s^\sharp$ that may-reaches p. By assumption $s' \sqsubseteq s^\sharp$ and thus $\mathsf{ts}(s') = \mathsf{ts}(s^\sharp)\sigma$ for some substitution σ. Let t be defined such that $s \to_P t$. Due to Lemma 5.23, we have $t' \sqsubseteq t^\sharp$ and thus there exists a substitution τ such that $\mathsf{ts}(t') = \mathsf{ts}^*(t^\sharp)\tau$.

Consider the rule $\mathsf{f}_{s^\natural}(\mathsf{ts}(s^\natural)) \to \mathsf{f}_{t^\natural}(\mathsf{ts}^*(t^\natural))$. By definition address q points in $s^\natural$ to an abstract variable x such that x occurs in $\mathsf{ts}(s^\natural)$ and $\mathsf{ts}(t^\natural)$. Furthermore, x is replaced by an extra variable x' in $\mathsf{ts}^*(t^\natural)$. Wlog., we assume that x' is the only extra variable in $\mathsf{ts}^*(t^\natural)$. Let m be a morphism such that $m\colon s^\natural \to s'$ and $m(q) \stackrel{+}{\to} m(p)$. By definition of $\mathtt{Putfield}^\natural$, $m(p)$ and $m(q)$ exist in t' and only the part of the heap reachable from these addresses can differ in s' and t'.

In order to show the admissibility of the rewrite step $\mathsf{f}_{s^\natural}(\mathsf{ts}(s')) \to \mathsf{f}_{t^\natural}(\mathsf{ts}(t'))$ we define a substitution ρ such that $\mathsf{ts}(s^\natural)\rho = \mathsf{ts}(s')$ and $\mathsf{ts}^*(t^\natural)\rho = \mathsf{ts}(t')$. We set:

$$\rho(y) := \begin{cases} \tau(x) & \text{if } y = x' \\ \sigma(y) & \text{otherwise .} \end{cases}$$

Then $\mathsf{ts}(s^\natural)\rho = \mathsf{ts}(s')$ by definition as $x' \notin \mathsf{Var}(s^\natural)$. On the other hand $\mathsf{ts}^*(t^\natural)\rho = \mathsf{ts}(t')$ follows as the definition of ρ forces the correct instantiation of x' and Lemma 5.23 in conjunction with Lemma 6.4 implies that σ and τ coincide on the portion of the heap that is not changed by the field update.

□

The next lemma emphasises that any execution step is represented by finitely many but at least one rewrite steps in $\mathcal{R}$.

Lemma 6.11. *Let $s, t \in \mathcal{JS}$ and $s^\natural \in G$ such that $s \in \gamma(s^\natural)$. Suppose $s \to_P t$, then there exists $t^\natural \in G$ such that $t \in \gamma(t^\natural)$ and $\mathsf{f}_{s^\natural}(\mathsf{ts}(\beta(s))) \xrightarrow{\leqslant K}_{\mathcal{R}} \mathsf{f}_{t^\natural}(\mathsf{ts}(\beta(t)))$. Here $K \in \mathbb{N}$ depends only on G and $\xrightarrow{\leqslant K}_{\mathcal{R}}$ denotes at least one and at most K many rewrite steps in $\mathcal{R}$.*

Proof. The lemma follows from the proof of Lemma 5.28 and Lemma 6.10. □

We arrive at the main result of this thesis.

Theorem 6.12. *Let $s, t \in \mathcal{JS}$. Suppose $s \to_P^* t$, where s is reachable in P from some initial state $i \in \mathcal{JS}$. Let $s' = \beta(s)$ and $t' = \beta(t)$. Suppose G is the computation graph of P such that $i^\natural \in G$ and $i \in \gamma(i^\natural)$. Then there exists $s^\natural, t^\natural \in G$ and a derivation $\mathsf{f}_{s^\natural}(\mathsf{ts}(s')) \to_{\mathcal{R}}^* \mathsf{f}_{t^\natural}(\mathsf{ts}(t'))$ such that $s \in \gamma(s^\natural)$ and $t \in \gamma(t^\natural)$. Furthermore, for all n: $\mathsf{rcjvm}(n) \in O(\mathsf{rctrs}(n))$.*

Proof. The existence of $s^\natural$ follows from the correctness of abstract computation together with the construction of the computation graph. Let m denote the runtime of the execution $s \to_P^* t$. Then by induction on m in conjunction with Lemma 6.11 we obtain the existence of a state $t^\natural$ such that $t \in \gamma(t^\natural)$ and a derivation:

$$\mathsf{f}_{s^\natural}(\mathsf{ts}(s')) \xrightarrow{\leqslant K \cdot m}_{\mathcal{R}} \mathsf{f}_{t^\natural}(\mathsf{ts}(t')) \; . \tag{6.1}$$

Here the constant $K \in \mathbb{N}$ depends only on G. We have $\mathsf{f}_{s^\natural}(\mathsf{ts}(s')) \to_{\mathcal{R}}^* \mathsf{f}_{t^\natural}(\mathsf{ts}(t'))$ from which we conclude the first part of the theorem.

To conclude the second part, let n be arbitrary and suppose m denotes the runtime of the execution $i \to_P^* t$, where $|i| \leqslant n$. We set $i' = \beta(i)$. As G is the

computation graph of P we obtain $i \in \gamma(i^\natural)$. From Lemma 6.6 it follows that $\|\mathsf{ts}(\beta(i))\| \leqslant |i|$. Specialising (6.1) to $i^\natural$ and i' yields $\mathsf{f}_{i^\natural}(\mathsf{ts}(i')) \xrightarrow{\leqslant K \cdot m}_{\mathcal{R}} \mathsf{f}_{t^\natural}(\mathsf{ts}(t'))$. Thus we obtain

$$\mathsf{rcjvm}(|i|) = m \leqslant K \cdot m \leqslant \mathsf{rctrs}(\|\mathsf{ts}(\beta(i))\|) \leqslant \mathsf{rctrs}(|i|) \; .$$

□

The corollary follows directly from the previous theorem.

Corollary 6.13. *Let P be a program and $\mathcal{S} \subseteq \mathcal{JS}$. Suppose computation graph G is obtained from initial state $\alpha(\mathcal{S})$. Suppose cTRS $\mathcal{R}$ is obtained from G. We set $t \gg s$ iff $\mathsf{ts}(\beta(s)) = t$. Then $\gg \cdot \to_{P|\mathcal{S}} \subseteq \to^+_{\mathcal{R}} \cdot \gg$, and $t \gg s$ for all $s \in \mathcal{S}$ and some $t \in \alpha(\mathcal{S})$. Furthermore, $\|t\| = O(|s|)$ for all $s \in \mathcal{S}$ and $t \gg s$. Hence, $\gg$ is a complexity preserving abstraction.*

It is tempting to think that the precise bound on the number of rewrite steps presented in Lemma 6.11 should translate to a linear simulation between JVM executions and rewrite derivation. Unfortunately this is not the case as the transformation is not termination preserving. For this consider the program of Figure 6.2.

```
class List{
  List next;
}

class Main{
  void inits(List ys){
    while(ys.next != null){
      List cur = ys;
        while(cur.next.next != null){
          cur = cur.next
        }
      cur.next = null;
      }
  }
}
```

Figure 6.2.: The `inits` program.

Here the outer loop cuts away the last cell until the initial list consists only of one cell whereas the inner loop is used to iterate through the list. It is easy to see that the main function terminates if the argument is an acyclic list. Since variables *ys* and *cur* share during iteration, the proposed transformation introduces a fresh variable for the `next` field of the initial argument *ys* when performing the `Putfield`$^\natural$ instruction. Termination of the resulting rewrite system can not be shown any more.

However *non-termination preservation* follows as an easy corollary of Theorem 6.12.

Corollary 6.14. *The computation graph method, that is the transformation from a given JBC program P to a cTRS $\mathcal{R}$ is non-termination preserving.*

Proof. Suppose there exists an infinite run in P, but $\mathcal{R}$ is terminating. Let i be some initial state i of P. By Theorem 6.12 there exists a state t such that $i \to_P^* t$ and $\mathsf{f}_{i^\natural}(\mathsf{ts}(i')) \to_\mathcal{R}^* \mathsf{f}_{t^\natural}(\mathsf{ts}(t'))$, where $i \in \gamma(i^\natural)$, $i' = \beta(i)$, $t \in \gamma(t^\natural)$, and $t' = \beta(t)$. Furthermore, as $\mathcal{R}$ is terminating we can assume $\mathsf{f}_{t^\natural}(\mathsf{ts}(t'))$ is in normalform. However, as t is non-terminating, there exists a successor, thus Lemma 6.11 implies that $\mathsf{f}_{t^\natural}(\mathsf{ts}(t'))$ cannot be in normalform. Contradiction. □

7. Related Work

In this chapter we review related work. First, we take a look into general programming language properties and features. For example, new challenges arise when the integer type of a programming language is bounded and an overflow can occur. Afterwards, we give an overview over related approaches that have been developed in recent years.

Integer Overflow. The examples in Figure 7.1 are taken from [18] and illustrate the problem of approximating bounded integers with overflow by unbounded integers. Program `overflow1` terminates when considering bounded integers with overflow as the counter eventually gets negative, but does not terminate when considering unbounded integers. The abstraction with unbounded integers is sound, but termination of the abstraction can not be shown. Program `overflow2` does not terminate when considering bounded integers with overflow and parameter j equals the upper integer bound, but always terminates using unbounded integer. This however gives rise to a false positive example. In [18]

```
void overflow1(int i){
  while(i>0){++i;}
}
```

```
void overflow2(int i,int j){
  while(i=<j){++i;}
}
```

Figure 7.1.: Problems arising from integer overflow.

a special abstract domain based on bitvector arithmetic is used to handle such programs correctly. In Jinja the integer domain is unbounded and this problem does not arise. This is not true for Java, as the integer domain is bounded.

Garbage Collection. We implicitly assume that objects that are not reachable from the program environment are immediately collected by the garbage collector. This is very common in practice [39, 44] as the garbage collector often depends on the implementation rather than on the language specification. In [5] different strategies are identified to incorporate garbage collection in heap space analysis.

Arrays. Arrays for Jinja are introduced in [33], together with threads. Arrays are treated similar to objects, but consists of a dedicated set of instructions. Since the size of an array may be determined dynamically, it is not clear how to represent arrays and operations on arrays using terms in a sound and proper way. One may abstract arrays similarly as we abstract cyclic data.

Threads. Programs are considered to be single-threaded. Multiple threads impose multiple challenges, such as undeterministic behaviour and access of shared

variables. A rewriting based analysis of multi-threaded Java programs can be found in [19]. The semantics are presented in a continuation based style. This results in an increase of the state space. Search and model-checking techniques are then used to verify safety properties.

Non-Linear Arithmetic. Due to abstraction, non-linear arithmetic expressions may arise in program analysis in the presence of a multiplication operator. This could happen, for example, if the concrete value of the operands are not known and abstracted to integer variables during analysis. Analysis of non-linear arithmetic expressions depends on sophisticated numerical property domains. Current tools often rely on the polyhedra domain to infer linear constraints on variables, for example [4, 45]. Variables affected by non-linear expressions are abstracted by losing all information about the variable. Here, we rely on cTRSs and constraints over Presburger arithmetic. The programs under study also do not contain float and bitwise operations.

Recursion. Our approach can not handle unbounded list of frames and therefore we restrict our analysis to non-recursive programs. A common approach to handle recursion is to consider only the program environment of the current frame (or method) [2, 45]. Then side-effects have to be safely approximated on call-sites.

7.1. Termination Graphs

As already indicated our transformation approach is based on previous achievements of Otto et al. [39] and Brockschmidt et al. [11]. We clarify the connections here:

The concept of computation graphs for object-oriented bytecode has been introduced before in [11, 39] as *termination graphs* to prove termination of bytecode programs. Several extensions have been introduced to prove termination of recursive programs [10], to prove non-termination [12], and to prove termination with cyclic data [9]. Termination graphs have also been applied for functional [47] and logic programming [43]. These techniques have been implemented in the prover AProVE[1].

In comparison to [11, 39], we employ a simpler representation of abstract states, by not including sharing and shape information of the heap directly into the abstract state. This results in a more intuitive description of abstraction by means of graph morphisms. On the other hand, additional (external) analyses are necessary to make our approach useful. In Section 8.1 we are going to present the analyses currently used in our implementation. These analyses provide data facts for all program locations and can be easily integrated in the computation graph approach to refine the transformation. However, the information obtained from state refinements in the computation graph can in general not be used that easily to refine information of the external analysis, as the flow graph in the external analyse may differ from the computation graph.

[1] `http://aprove.informatik.rwth-aachen.de`

This may result in a loss of precision. The combination of the computation graph method with domains for shape analysis is subject to future work.

In [11, 39], the main result was a non-termination preserving transformation such that termination of the resulting rewrite system implies termination of the original program. Here we have shown that our approach is in addition complexity preserving. This result extends also to their work. Furthermore, we have shown the connection of the computation graph approach to standard techniques of program analysis.

7.2. The SPEED Method

One of the major challenges for analysing imperative programs is to cope with user-defined data structures. In [23] Gulwani et al. present SPEED; a tool that is designed to compute symbolic complexity bounds for C/C++ programs, handling conditional loops as well as iterations over user-defined data structures.

The method is based on *counter instrumentation* which is already presented in [14] as a possible application for abstract interpretation. In the most simple case, a single imaginary counter variable is introduced. This counter is initialised to zero at the beginning of a procedure and incremented within every loop construct. The intuition is that the total number of iterations are counted. Safe upper approximations are computed using techniques from static analysis. In particular, relational domains, such as the polyhedra domain allow to describe an upper bound of the counter variable in terms of the input parameters. In practice this approach is limited as the polyhedra domain just provides linear invariants and the generation of invariants gets infeasible for complex programs.

To overcome this limitation Gulwani et al. proposed a method where multiple counter variables are used. Furthermore, counter variables can be reinitialised to zero within loops. Invariants are computed for each counter variable and composed adequately. To find a suitable counter instrumentation an exhaustive proof search is applied. We leave out the details for computing the bound but exemplify the general idea with the following example:

Example 7.1. Figure 7.2 depicts a program with a non-linear bound. The grey code segments correspond to the counter instrumentation of variables c and d. A single counter variable would fail as the invariant analysis with the polyhedra domain only provides linear constraints. In the `else` branch of the condition, c is reinitialised to zero whereas d is incremented by one. This means that the bound of c depends on d and for every iteration of d we have to account for the bound of c. This can be compared with a nested loop statement, where the inner loop is initialised to a constant in each iteration of the outer loop. The generated bound is $\mathsf{max}(0, n) + \mathsf{max}(0, m) \times \mathsf{max}(0, n)$.

We now address how bounds for iterations over user-defined data structures, such as lists and trees, are computed in SPEED. Rather than an automated analysis of the heap, the user is required to define quantitative functions and effects of method calls on it. Quantitative functions represent numerical properties of data structures and are represented by uninterpreted functions. Method effects

```
simplemultipledep(int n,m){
  int c=0; int d=0;
  int x=0; int y=0;
  while(x<n){
    if(y<m){
      y++; c=++;
    }else{
      y=0; x++;
      c=0; d++;
    }
  }
}
```

Figure 7.2.: Counter instrumentation for a non-linear bounded program.

are specified by constraints over the combined domain of linear constraints and uninterpreted functions. Bounds are then generated by computing invariants.

Example 7.2. Following uninterpreted function symbols provide quantitative measures for single linked lists:

$$\texttt{Len}(L) := \text{length of list } L, \text{ and}$$
$$\texttt{Pos}(e, L) := \text{position of element } e \text{ in list } L.$$

The effect of getting the next element can be described by the following constraints:

$$e = L.\texttt{GetNext}(f) := \texttt{Pos}(e, L) = \texttt{Pos}(f, L) + 1;$$
$$\texttt{Assume}(0 \leqslant \texttt{Pos}(f, L) < \texttt{Len}(L))$$

Notice the combination of the domain of linear constraints and uninterpreted functions. We consider a program iterating over a list:

$$\texttt{for}(e = f; e \neq \texttt{null}; e = L.\texttt{GetNext}(e));$$

The invariant generator of the SPEED tool can establish following invariant: $c = \texttt{Pos}(e, L) - \texttt{Pos}(f, L) \wedge \texttt{Pos}(e, L) \leqslant \texttt{Len}(L)$, which simplifies to $c \leqslant \texttt{Len}(L) - \texttt{Pos}(f, L)$.

This method requires that invariants over the combination of linear constraints and uninterpreted functions can be generated. In [24] Gulwani and Tiwari present a new method for combining abstract interpreters automatically, which is is based on the Nelson and Oppen method for combining decision procedures [36]. The proposed approach often provides intuitive and precise bounds. However, the method is not fully automatic and the constraints for the method effects can get complex for more sophisticated data structures or when side-effects have to be considered.

In contrast to SPEED our approach provides a fully automatic analysis of programs with user-defined data structures. Our termed-based abstraction together with symbolic evaluation is able to capture the operations on user-defined data structures.

7.3. Resource Static Analysis (RESA)

In [6] Atkey presents an approach for amortised resource analysis of imperative languages by embedding a logic of resources within Separation logic. The techniques developed have later been implemented for the resource analysis of Java bytecode [20].

The conceptual idea of amortised analysis is that data structures store resources that can be consumed by other operations. Separation logic [15] is an extension of Hoare logic that allows reasoning about the presence and shape of shared mutable data structures. For example, the assertion $A * B$ holds for store s and heap h, if h can be split into two disjoint heaps h_1 and h_2, and assertion A holds for s, h_1 and assertion B holds for s, h_2. Atkey proposes that information about consumable resources can be incorporated in a similar manner besides the heap.

For that purpose a resource aware program logic is introduced, including an additional instruction `consume` r that indicates the consumption of resource r. The rules of the logic mimic the operational semantics of the instructions. Furthermore, procedures are annotated with preconditions and postconditions. Preconditions state predicates about arguments, heap and available resources. Postconditions state predicates about arguments, heap, remaining resources and return value. A variant of separation logic including resource information serves as the language for assertions in the program logic. For example, following resource-aware inductive predicate represents a list segment where resources R are associated to each element:

$$\mathsf{lseg}(R, x, y) := (x = y \wedge \mathsf{emp}) \vee \exists z, z'.[x \xmapsto{\mathsf{data}} z] * [x \xmapsto{\mathsf{next}} z'] * R * \mathsf{lseg}(R, z', y)$$

Automation is achieved as follows: First, verification conditions are generated. Verification conditions represent intermediate assertions between preconditions and postconditions of a procedure. The conditions are induced by the program logic and are generated in a bottom up fashion starting from the postcondition. To resolve loops, additional loop invariants are required. Given the collection of problems generated, a goal driven proof search is performed verifying that the precondition implies the postcondition. Moreover, the proof search procedure collects linear constraints describing the resource usage of the bytecode instruction. The resulting set of linear constraints can then be solved by a linear constraint solver.

The motivating examples in [6] suggest that this analysis works well on cyclic data structures. Though, the implementation [20] indicates still some challenges: The proof search procedure depends strongly on the predicates for data structures. Therefore only lists and trees are supported. Loop invariants are not generated automatically and have to be stated by the user. Moreover, different techniques have to be used to handle loops depending on numeric quantities rather than on allocated data structures. Currently only linear bounds can be established from the generated linear constraints.

7.4. Resource Aware Java (RAJA)

In [27] a type system for amortised heap-space analysis for a Java-like programming language, termed RAJA (Resource Aware JAva), has been introduced. Later on, a fully automatic type inference algorithm has been presented in [28]. A prototype implementation is publicly available[2].

Here, data structures are associated with potentials (or resources). The potential of the data structures in the input state represents an upper bound on the total heap consumption.

Types in RAJA are refined class types. That is, a type consists of a class identifier together with a view. Field access, field update, and method invocations are extended to refined types. The idea is that different potentials and effects can be associated to a data structure depending on its refined type. One of the motivating examples in [27] is a method for copying a singly-linked list. A *rich* and a *poor* view have been introduced for list cells such that the potential for a rich cell is 1 and the potential for a poor cell is 0. For copying list cells, objects are created and potential is used. The well-typing requires that the initial list is a *rich* list and the resulting list is a *poor* list. Hence, the method can not be invoked repeatedly. A runtime object can have multiple refined types and the overall potential of the object is the sum over all access paths, thus accounting for aliasing. Typing judgements in RAJA additionally incorporate the cost of evaluating an expression and ensure that whenever an expression terminates with an unbounded memory, then it terminates with a bounded memory of size n plus the potential in the current state before the execution.

To determine upper bounds automatically, view variables were introduced in [28]. Moreover, methods are equipped with a set of subtyping and linear arithmetic constraints capturing the resource consumption of the method. Constraints are generated via an extended type inference algorithm. A solution of the constraints provides a typing judgement and therefore also a bound. Arithmetic constraints are solved via a linear constraint solver. The subtyping constraints are reduced to inequality constraints over infinite labelled trees. Current methods assume that the solution are regular infinite trees, which implies a linear bound.

7.5. The JULIA Static Analyser

The JULIA static analyser[3] is one of the first and probably the most elaborated Java bytecode analyser. It has been developed over the last few years by Spoto et al. to provide an extensive and scalable system for program analysis for full Java [45]. The tool provides various kinds of analyses, such as class, null pointer, initialisation, sharing, acyclicity, aliasing, path-length and termination.

Currently JULIA does not perform any kind of complexity analysis. However, it features a fully automatic termination check, handling user-defined data structures dynamically allocated in memory [46]. We summarise the basic idea:

[2] `http://raja.tcs.ifi.lmu.de`
[3] `http://www.juliasoft.com`

First, a *path-length* analysis of the bytecode program is performed [40]. Here variables are abstracted into an integer path-length. If a variable is bound to an integer i then its path-length is i itself. If a variable is bound to an address a then its path-length is the maximal length of a path in the graph induced by the heap starting from a. Path-lengths are represented by numerical constraints in the closed polyhedra abstract domain. The transfer functions describe the constraints on variables for the operand stack and local variables. Consider for example the `Getfield`$^\sharp$ fn cn instruction: Let s_n, s'_n denote the variable representing the top of the stack before and after executing the instruction. If the accessed field fn of class cn is of type integer, we obtain no information about its length. Note that all informations of an object, besides path-length, is disregarded. If the field is a class type and the current object is maybe-cyclic we obtain $s_n \geqslant s'_n$, ie., the new value is not larger than the old value. If the field is a class type and the current object is acyclic we obtain $s_n \geqslant 1 + s'_n$, ie., the new value is strictly smaller than the old value. As observed the path-length analysis makes use of other analysis such as maybe-cyclic, maybe-sharing, and definite aliasing to improve precision.

Afterwards, a *constraint logic program (CLP)* is extracted from the path-length analysis. For example, following CLP is obtained for the `append` example (cf. Figure 3.1), when *this* is not cyclic:

```
entry980(IL2) :- {IL2>=2,IL2-OL2>=1,OL2>=0}, entry980(OL2).
```

Here, `entry980` corresponds to the entry of the `while` loop, `IL2` and `OL2` denote the path-length of *cur* before and after executing the body of the loop. This transformation is non-termination preserving, ie., termination of the CLP program implies termination of the original bytecode program.

Finally, the obtained CLP program can be analysed by a termination prover for constraint logic programs. For example, BinTerm[4] tries to find affine ranking functions for recursive predicates after performing simplifications.

Currently, JULIA does not provide automatic complexity analysis of programs. However, the path-length abstraction described here is also used by the tool we present in the next subsection. We want to remark that the CLPs obtained by the aforementioned analysis can be directly expressed as cTRSs.

7.6. Cost and Termination Analyser (COSTA)

The COSTA (COSt and Termination Analyser for Java Bytecode)[5] tool is developed by Albert et al. and provides automatic cost and termination analysis of Java bytecode programs [2, 3, 4]. The tool provides a generic way to apply different *cost models* and often returns precise cost results. A cost model determines the cost to be assigned to an execution step, for example, the number of bytecode instructions or the heap consumption.

At first, a control flow graph of the bytecode program is constructed, where blocks are sequences of (non-branching) instructions and edges are labelled by

[4] `http://lim.univ-reunion.fr/staff/fred/dev`
[5] `http://costa.ls.fi.upm.es/web`

constraints that represent conditions on the flow. Based on the control flow graph an intermediate representation, termed *rule based representation (RBR)*, is generated. Rules correspond to blocks and define a recursive representation with a flattened stack. Rules follow roughly the following pattern:

$$m_i(loc, stk_i, ret) \leftarrow guard, b_{i,1}, \ldots, b_{i,n}, m_j(loc, stk_j, ret) \ ,$$

where m_i, m_j are block identifier, loc are the local variables, stk_i, sk_j are stack variables, ret is a variable storing the return value, $guard$ a (possibly empty) control flow constraint, $b_{i,1}, \ldots, b_{i,n}$ are the bytecode instructions of block m_i, and $m_j(loc, stk_j, ret)$, denotes the continuation of m_i. Several simplifications and optimisations are performed on the RBR. For example, a static single assignment transformation of the bytecode and stack variable elimination.

Based on the RBR (linear) *size-relations* among variables are generated. Size relations are given as conjunction of linear constraints. In particular one is interested in inferring *input-output* relations, which relates input arguments of a rule with input arguments of its continuations. These size relations are obtained using techniques from abstract interpretation. Guards and bytecode instructions are compiled into linear constraints. This is trivial for (linear) arithmetic operations. Objects are abstracted by their maximal path-length [40]. Afterwards bottom up fixed points are generated.

Rules give rise to cost equations. The cost of executing a rule consists of the cost for executing the bytecode instructions together with the cost of executing the continuation of the rule. A set of (recursive) cost equations is termed *cost relation system (CRS)*. A (non-recursive) closed form representation can be obtained using solvers for CRSs, for example the PUBS solver [1].

The basic concept of our method is similar to the COSTA approach: Abstract interpretations are used to find a representation that over-approximates the program relation. In contrast to COSTA we use a term-based abstraction of objects. We think that term-based abstractions are more suitable for programs with composited data structures such as those appearing in the `flatten` program of Figure 1.3. The COSTA tool is not able to prove termination or provide an upper bound of the program. Our transformation allows us to infer a linear bound automatically. We discuss this in more detail in Section 8.2.

7.7. Loop Bounds for C Programs (LOOPUS)

The LOOPUS tool computes loop bounds for C programs using the *size-change abstraction (SCA)* [49]. Originally, SCA has been introduced to proof termination of functional programs with well-founded data [32]. The size-change analysis generates an abstraction of a program P. This abstraction approximates the size relations between source and destination parameters in each function call. Then P terminates on all inputs if every infinite computation implies an infinite descent in some data value.

In LOOPUS SCA is used to generate loop invariants for inner loops. The domain of SCA can be represented as Boolean combinations of (in)equality constraints between variables (or norms) in disjunctive normal form. SCA is a

disjunctive abstract domain in contrast to the standard polyhedra domain for example. Therefore, this domain can naturally abstract multiple paths within a loop body. In [49] the abstract domain is termed *set of size-change relations (SCRs)*. SCRs defines the standard powerset domain of (in)equality constraints over a finite set of (primed) norms. Norms are functions from states to integers and are extracted at the beginning of the analysis using heuristics. For example, if $x \geqslant y$ is an arithmetic constraint in a cycle-free path from location l back to location l, then $x - y$ is used as a norm if $x - y$ decreases along the path.

LOOPUS infers upper bounds for program locations l. An upper bound defines how often l is visited in an execution of P wrt. to the input. The overall procedure proceeds in two steps. First, a transition system for l wrt. P is generated. A transition system represents an over approximation of the concrete transition relation. A transition relation is represented by a conjunction of linear constraints over variables. Second, the algorithm tries to infer a bound from the resulting transition system.

In the first step inner loops are summarised recursively. For example, suppose l_0 is the program location of a loop and l_1 is the program location of a loop within the body of l_0. The initial transition system for l_1 is obtained by the composition of the transition relations along (cycle-free) paths from l_1 to l_1. There can exist multiple paths in the presence of branching statements. The most precise loop invariant would be obtained by iteratively expanding the initial transition system, but is not computable in general. Hence the transitive hull (or fixed-point) of the abstraction of the initial transition system is computed. The domain SCRs is finite by construction and the fixed-point can always be computed. The transition system l_0 is then obtained by enumerating all (cycle-free) paths from l_0 to l_0. Here, l_1 is replaced by the transition invariant. SCR is a disjunctive domain. A *pathwise* analysis is obtained by considering each disjunct.

The second step first applies *contextualisation* of the resulting transition system. The contextualisation of a transition system infers which transitions can be executed from a given program location. The idea is that the transition system of a loop can be classified into multiple loop phases. Afterwards an overall bound is composed from bounds of all SCCs. LOOPUS uses the topological order from the contextualisation of the transition system to define dependencies amongst SCCs and suitably combine them with max and addition. To compute a bound from an SCC LOOPUS looks for non-increasing norms in the abstraction of the transition relation. For example, if norm n is non-increasing, ie. $n \geqslant n'$, on all transitions of the SCC but decreasing and bounded, ie. $n \geqslant n', n \geqslant 0$, on some transitions of the SCC then $\mathsf{max}(n, 0)$ provides a bound. Global invariants, which are generated by standard abstract domains such as the octagon or polyhedra domain, are used to provide upper bounds in terms of the input. Non-linearity can arise if a norm is not non-increasing on all transitions, but existing bounds can approximate how often the subgraph defined by all non-increasing edges of the norm can be entered.

The combination of pathwise analysis together with contextualisation seems to perform well in practice, as different loop phases and the dependencies among them can be extracted. In particular, for the analysis of single (possibly nested)

loops. As the actual bound depends on global invariants the analysis may fail when the bound is non-linear. For that, consider a program with two sequent loops such that the bound for the second loop depends on a value computed by the first loop. If the size of the value is non-linear wrt. to the input, then standard domains such as the octagon or polyhedra domain can not provide a bound on the value.

In [26] graph-based techniques for the analysis of term rewrite systems were introduced. This techniques are similar to the aforementioned pathwise analysis and contextualisation. Though, a possible limitation of our approach is that we do not generate sophisticated invariants among variables and therefore the path analysis is less precise. It is conceivable to support the transformation with additional constraints generated by an external invariant analyser or actually perform invariant analyses on cTRSs. An investigation of the latter option is subject to future work.

8. Implementation Details

A prototype, termed JaT[1] (Jinja Analysis Tool), has been implemented in the Haskell[2] programming language. In this chapter we discuss the details of the implementation. In Section 8.1 we show how (external) heap shape analyses are incorporated into our transformational approach. In Section 8.2 we provide some information about the tool itself.

8.1. Jinja Static Analysis

We kept parts of the bytecode transformation in previous chapters abstract. In particular, we relied on heap shape properties such as sharing and acyclicity. In this section we present the additional flow analyses used in the current implementation. First, we introduce the *set of simplified states* domain. A simplified state disregards information about the program location and allows a more concise representation of the other property domains. Afterwards, we present a *type* analysis based on the bytecode verifier of JBC [30]. The type analysis provides type information on operand stack and local variables. Finally, we introduce the *sharing* and *acyclicity* domains that provide shape information on objects bound to stack and local variables during runtime. In the following let P be a well-formed and non-recursive bytecode program.

8.1.1. Set of Simplified States

Let $\mathcal{JS}^*$ denote the *set of simplified JVM states*. We define $\mathcal{JS}^*$ like $\mathcal{JS}$ but disregard all informations about the program location in a state. That is, frames in $\mathcal{JS}^*$ only consists of operand stack and local variables. Similarly, the semantics on $\mathcal{JS}^*$ is defined identically to the semantics on $\mathcal{JS}$ with non-relevant elements ignored. Then $\mathcal{P}(\mathcal{JS}^*) := (\mathcal{P}(\mathcal{JS}^*), \subseteq, \cup, \cap, \varnothing, \mathcal{JS}^*)$ is a complete lattice.

We establish Galois connections between $\mathcal{JS}^*$ and the other domains of interest. As simplified states do not contain any information of the program location we can provide a more concise presentation of the other domains. We make our actual algorithm responsible for keeping track of the program location by defining adequate successor functions for our instructions. In the following we do not differentiate between a JVM state and a simplified JVM state if it is clear from the context or irrelevant.

[1] `http://cl-informatik.uibk.ac.at/users/georg/cbr/tools/jat`
[2] `http://www.haskell.org`

8.1.2. Type Analysis

The type analysis abstracts values of the program environment to types. In particular it provides an upper bound with respect to $(\mathsf{types}(P), \leqslant_{\mathsf{type}})$ (cf. Definition 3.6) on stack and local variables. The analysis is based on the well-typed analysis of JBC [30], but extended to an interprocedural analysis. The type information obtained from the analysis is used in the sharing and acyclicity domain.

Definition 8.1. We define the complete lattice on program types $\mathsf{types}(P) := (\mathsf{types}(P), \leqslant_{\mathsf{type}}, \bigsqcup_{\mathsf{type}}, \bigsqcap_{\mathsf{type}}, \mathsf{void}, \top)$, by extending $(\mathsf{types}(P), \leqslant_{\mathsf{type}})$ with a dedicated top symbol such that $t \leqslant_{\mathsf{type}} \top$ for all $t \in \mathsf{types}(P)$, and providing a suitable join operation:

$$t \sqcup_{\mathsf{type}} t' := \begin{cases} t' & \text{if } t \leqslant_{\mathsf{type}} t' \\ t_0 & \text{if } t_0 \text{ is the least common superclass of } t \text{ and } t' \\ \top & \text{otherwise .} \end{cases}$$

The abstract domain is denoted $\mathsf{TY} \supseteq \{\bot, \top\}$. Elements of TY are denoted τ and obtained from JVM states, when mapping the values of the operand stack and local variables to its type. More formally:

Definition 8.2. Let $s = (heap, frms)$ denote a state in $\mathcal{JS}^*$ with $frms = [frm_1, \ldots, frm_k]$ and $frm_i = (stk_i, loc_i)$. Let type^* be the component-wise extension of type to lists. Then β_{TY} is defined as follows:

$$\beta_{\mathsf{TY}}(s) := [(\mathsf{type}^*(stk_1), \mathsf{type}^*(loc_1)), \ldots, (\mathsf{type}^*(stk_k), \mathsf{type}^*(loc_k))] \; .$$

Let $\mathsf{dom}(\tau)$ collect all stack and local variables, ie., $Stk \cup Loc$. We lift operations of the complete lattice of types to type environments by a component-wise application of $\leqslant_{\mathsf{type}}$ and $\sqcup_{\mathsf{type}}$. Since P is well-formed by assumption, the domains of two environments coincide for some specific program location. In particular occurrences of $\top$ during analysis indicate a type error. Hence, in the following definition we only consider the interesting cases.

Definition 8.3. Let τ, τ' be elements of TY for some specific program location. Then $\tau \sqsubseteq_{\mathsf{TY}} \tau'$ holds, if $\mathsf{type}(v) \leqslant_{\mathsf{type}} \mathsf{type}(v')$ for all $v \in \mathsf{dom}(\tau), v' \in \mathsf{dom}(\tau')$ and $v = v'$. We obtain the supremum τ'' of τ and τ' as follows: We set $\mathsf{dom}(\tau'') := \mathsf{dom}(\tau)$, and $\mathsf{type}(v'') := \mathsf{type}(v) \sqcup_{\mathsf{type}} \mathsf{type}(v')$ for all $v \in \mathsf{dom}(\tau), v' \in \mathsf{dom}(\tau'), v'' \in \mathsf{dom}(\tau'')$, and $v = v' = v''$. Then $\mathsf{TY} := (\mathsf{TY}, \sqsubseteq_{\mathsf{TY}}, \bigsqcup_{\mathsf{TY}}, \bigsqcap_{\mathsf{TY}}, \bot, \top)$ is a complete lattice.

The representation function β_{TY} gives rise to $\alpha_{\mathsf{TY}} \colon \mathcal{P}(\mathcal{JS}^*) \to \mathsf{TY}$ and $\gamma_{\mathsf{TY}} \colon \mathsf{TY} \to \mathcal{P}(\mathcal{JS}^*)$ (cf. Definition 2.14). Then $(\mathcal{P}(\mathcal{JS}^*), \alpha_{\mathsf{TY}}, \gamma_{\mathsf{TY}}, \mathsf{TY})$ is a Galois connection. Intuitively, the concretisation of a type environment represents all states that are compatible with the type environment. In particular, the abstract domain contains no information about the shape of an object in the heap, except what can be inferred from the types in the environment. Let cn, dn denote class names and t, t' denote types in P. Figure 8.1 depicts the

abstract transfer functions of TY. We omit some instructions as they can be easily inferred from the given ones. In the definition of $\texttt{Getfield}^{\mathsf{TY}}\ fn\ cn$, type t' correspond to the type of field (cn, fn). It is easy to check that the defined functions safely approximate the concrete semantics.

$$
\begin{aligned}
\texttt{Load}^{\mathsf{TY}}\ n\ ((stk, loc) : frms) &:= (loc(n) : stk, loc) : frms \\
\texttt{Store}^{\mathsf{TY}}\ n\ ((t : stk, loc) : frms) &:= (stk, loc\{n \mapsto t\}) : frms \\
\texttt{Push}^{\mathsf{TY}}\ v\ ((stk, loc) : frms) &:= (\mathsf{type}(v) : stk, loc) : frms \\
\texttt{Pop}^{\mathsf{TY}}\ ((t : stk, loc) : frms) &:= (stk, loc) : frms \\
\texttt{IAdd}^{\mathsf{TY}}\ ((\mathsf{int} : \mathsf{int} : stk, loc) : frms) &:= (\mathsf{int} : stk, loc) : frms \\
\texttt{CmpEq}^{\mathsf{TY}}\ ((t : t : stk, loc) : frms) &:= (\mathsf{bool} : stk, loc) : frms \\
\texttt{BAnd}^{\mathsf{TY}}\ ((\mathsf{bool} : \mathsf{bool} : stk, loc) : frms) &:= (\mathsf{bool} : stk, loc) : frms \\
\texttt{Goto}^{\mathsf{TY}}\ ((stk, loc) : frms) &:= (stk, loc) : frms \\
\texttt{IfFalse}^{\mathsf{TY}}\ ((\mathsf{bool} : stk, loc) : frms) &:= (stk, loc) : frms \\
\texttt{New}^{\mathsf{TY}}\ cn\ ((stk, loc) : frms) &:= (cn : stk, loc) : frms \\
\texttt{Getfield}^{\mathsf{TY}}\ fn\ cn\ ((cn' : stk, loc) : frms) &:= (t' : stk, loc) : frms \\
\texttt{Putfield}^{\mathsf{TY}}\ fn\ cn\ ((t : cn' : stk, loc) : frms) &:= (stk, loc) : frms \\
\texttt{Checkcast}^{\mathsf{TY}}\ cn\ ((cn' : stk, loc) : frms) &:= (cn : stk, loc) : frms \\
\texttt{Invoke}^{\mathsf{TY}}\ mn\ n\ ((stk' : stk, loc) : frms) &:= \\
&\quad (\epsilon, loc' : [\mathsf{void}_0 : \cdots : \mathsf{void}_{mxl-1}]) : (stk' : stk, loc) : frms \\
&\quad \text{where } (stk', loc') = (t_{n-1} : \cdots : t_0 : cn, cn : t_0 : \cdots : t_{n-1}) \\
\texttt{Return}^{\mathsf{TY}}\ (([t], loc) : (stk' : stk, loc) : frms) &:= (t : stk, loc) : frms \\
&\quad \text{where } stk' = t_{n-1} : \cdots : t_0 : cn
\end{aligned}
$$

Figure 8.1.: The transfer functions of TY.

8.1.3. Sharing Analysis

Sharing analysis is essential to make our approach admissible. It is necessary to describe side-effects on objects which are abstracted by our approach. We apply the domain of sharing variable pairs and closely follow the presentation in [44]. Two variables share if there exists a shared part in the heap starting from the values bound to the variables. Elements of the abstract domain of pair sharing are (unordered) pairs of variables.

The pair sharing analysis is a may-analysis. That is, if two variables may-share in the abstract domain, then the variables do not necessarily share in the concrete domain. Contrary, if two variables share in the concrete domain, then they may-share in the abstract domain. We usually just say that two variables share, rather than two variables may-share.

We first comment in an informal manner on the sharing domain. Consider x, y, z to be program variables. An assignment $x := y$ replaces the value of x by the value of y. Therefore x shares with the variables y shares with. If the value of y is not an address then y shares with no variable. Consequently x shares with no variable. If the value of y is an address and y shares with some variable z then x also shares with z after the assignment. In particular x shares also with y. Now consider a field update $x.f = y$. If the value of y is not an address, then the only information assured is that no additional sharing is introduced. If the value of y is an address and y shares with z then x and y share. Furthermore, x shares with z. Similarly, if x and z share before the field update, then y and z share after it.

When restricting the analysis to non-recursive programs, side-effects of a method call can be analysed exactly by incorporating the stack and local variables of all frames in the domain. Hence, sharing is only introduced via the `Putfield` instruction.

In the following we will introduce the sharing domain more formally, based on [44]. We obtain a Galois insertion between sets of simplified states and the pair sharing domain.

We define the set of reachable classes, taking the subclass relation and the field table into account.

Definition 8.4. Recall Definition 3.5, introducing $\mathsf{subclasses}(cn)$. Let cn be a class of P. The set of *reachable classes* of cn is denoted as $\mathsf{reaches}(cn)$ and defined iteratively as follows:

$$\begin{aligned}\mathsf{reaches}(cn) &:= \textstyle\bigcup_{i\geqslant 0} \mathsf{reaches}^i(cn)\\ \mathsf{reaches}^0(cn) &:= \mathsf{subclasses}(cn)\\ \mathsf{reaches}^{i+1}(cn) &:= \textstyle\bigcup\{\mathsf{subclasses}(cn'') \mid cn' \in \mathsf{reaches}^i(cn), cn'' \in \mathsf{ft}^*(cn')\}\end{aligned}$$

Where $\mathsf{ft}^*(cn)$ returns the class types of declared fields in cn. We say that cn and cn' *statically share*, if $\mathsf{reaches}(cn) \cap \mathsf{reaches}(cn') \neq \varnothing$. Similarly, we say that two variables v_1 and v_2 statically share, if their corresponding type statically share and use $\mathsf{reaches}(v)$ to denote the classes reachable by v.

Let τ be a type environment. We parametrise abstraction, concretisation and transfer functions with a type environment, usually denoted with superscript τ. Note that we already know how to compute the type environment (cf. Subsection 8.1.2). The type environment allows to restrict the sharing domain to variable pairs that statically share.

Definition 8.5. Let τ be a type environment. The set of (unordered) pairs, whose static type shares, is defined by:

$$SV^\tau := \{(v_1, v_2) \in \mathsf{dom}(\tau) \times \mathsf{dom}(\tau) \mid \mathsf{reaches}(v_1) \cap \mathsf{reaches}(v_2) \neq \varnothing\} \ .$$

The abstract domain of pair sharing SH is:

$$\mathsf{SH}^\tau := \{sh \subseteq SV^\tau \mid \text{if } (v_1, v_2) \in sh \text{ then } (v_1, v_1) \in sh \text{ and } (v_2, v_2) \in sh\} \ .$$

Furthermore, $\mathsf{SH} := (\mathsf{SH}^\tau, \subseteq, \bigcup, \bigcap, \varnothing, SV^\tau)$ is a complete lattice.

The side condition in the definition of the sharing domain SH states, that (v_1, v_2) can only share if v_1, v_2 share with themselves, ie., there are bound to (maybe) non-null references.

After fixing the property space of the sharing domain, we are able to relate the concrete and abstract domain by means of an abstraction function and concretisation function.

Definition 8.6. Let $\mathcal{S} \subseteq \mathcal{JS}^*$ be a set of states compatible with τ. We define abstraction $\alpha_{\mathsf{SH}} \colon \mathcal{P}(\mathcal{JS}^*) \to \mathsf{SH}$ and concretisation $\gamma_{\mathsf{SH}} \colon \mathsf{SH} \to \mathcal{P}(\mathcal{JS}^*)$ as follows:

$$\alpha^\tau_{\mathsf{SH}}(\mathcal{S}) := \left\{ (x, y) \in \mathsf{dom}(\tau) \times \mathsf{dom}(\tau) \;\middle|\; \begin{array}{l}\text{there exists } s \in \mathcal{S} \text{ such that}\\ \text{variables } x \text{ and } y \text{ share in } s\end{array} \right\},$$

$$\gamma^\tau_{\mathsf{SH}}(sh) := \left\{ s \in \mathcal{S} \;\middle|\; \begin{array}{l}\text{for all } v_1, v_2 \in \mathsf{dom}(\tau) \text{ if } v_1 \text{ and } v_2 \text{ share in } s \text{ then}\\ (v_1, v_2) \in sh\end{array} \right\}.$$

Then, $(\mathcal{P}(\mathcal{JS}^*), \alpha_{\mathsf{SH}}, \gamma_{\mathsf{SH}}, \mathsf{SH})$ is a Galois insertion.

To complete the definition of the sharing analysis, we are left to define the (abstract) transfer functions. We do this informally by first defining an assignment and a field update operation. We will see that this is enough to describe almost all operations. Before, we define two operations on the sharing domain:

Definition 8.7. Let sh be a sharing domain. Then, $sh - v$ denotes the sharing domain, in which pairs including variable v are removed from sh:

$$sh - v := \{(v_1, v_2) \mid (v_1, v_2) \in sh, v \neq v_1 \text{ and } v \neq v_2\}.$$

The closure sh^*_v operation transitively closes sh with respect to v:

$$sh^*_v := sh \cup \{(v_1, v_2) \mid (v, v_1) \in sh \text{ and } (v, v_2) \in sh\}.$$

Definition 8.8. Let v, v_1, v_2 be variables.

$$(v := v_1)(sh) := \begin{cases} sh' \cup \{(v, v)\} \cup \{(v, v_2) \mid (v_1, v_2) \in sh\} & \text{if } (v_1, v_1) \in sh \\ sh' & \text{otherwise} \end{cases}$$

where $sh' = sh - v$, and

$$(v.f := v_1)(sh) := \begin{cases} ((sh \cup \{(v, v_1)\})^*_{v_1} - v_1)^*_v & \text{if } (v_1, v_1) \in sh \\ sh & \text{otherwise}. \end{cases}$$

It is easy to see that $(v := v_1)$ and $(v.f := v_1)$ correspond to the observations we stated at the beginning of this subsection. We now lift this operations to the set of instructions. The instructions $\texttt{Load}^{\mathsf{SH}}$, and $\texttt{Store}^{\mathsf{SH}}$ are actually assignments between local and stack variables. Similarly, in the non-recursive case, passing of function parameters and return values of $\texttt{Invoke}^{\mathsf{SH}}$ and $\texttt{Return}^{\mathsf{SH}}$ reduces to multiple application of the assignment operator. In the definition of field update $(v.f := v_1)$ we take into account that v_1 is a value on the stack that is removed after the $\texttt{Putfield}^{\mathsf{SH}}$ instruction. For the instruction $\texttt{New}^{\mathsf{SH}}$ we add (s_i, s_i) for the corresponding stack variable. The instructions $\texttt{Checkcast}^{\mathsf{SH}}$ and $\texttt{Getfield}^{\mathsf{SH}}$ refine the type of the top stack variable. Therefore we normalise with respect to the new type environment. All other instructions operate on non-address values and do not effect the sharing domain.

8.1.4. Acyclicity Analysis

Akin to the previous subsections we are going to present an acyclicity analysis. When transforming program states to terms, we unfold acyclic data and abstract possible cyclic data by introducing a fresh variable. The analysis presented here ensures that the data bound to a variable is acyclic, if the variable is in the defined property domain. We follow the presentation in [42] and suitably adapt it to our target language.

The acyclicity analysis is a must analysis. The property domain is the set of variables. Whenever a variable is in the acyclicity domain for some program location, then the data bound to the variable is definitely acyclic for all possible executions at the corresponding program location.

Let v_1, v_2 be variables. Furthermore, the types of v_1 and v_2 do not restrict them to be acyclic. First, consider the assignment operation $v_1 := v_2$. If v_2 is (possibly) cyclic then so is v_1. If v_2 is acyclic, then so is v_1. Next, consider the putfield operation $v_1.f := v_2$. If the type of the field $v.f$ is restricted to acyclic types then v_2 has to be acyclic and the operation does not affect the property domain. If v_2 is acyclic then v_1 is acyclic, if v_2 and v_1 do not share before the putfield instruction. Otherwise, we are not able to infer that v_1 is definitely acyclic. Furthermore, the putfield operation may effect other variables sharing with v_1.

As noted above, the acyclicity domain depends on the typing and sharing domain. The precision of the domain can be increased by requiring that the domain contains at least those variables, whose static type is acyclic. Therefore, we define a predicate that checks the desired property.

Definition 8.9. Non class types are acyclic. A class cn is acyclic if the predicate $\mathsf{acyclic}(cn)$ holds:

$$\mathsf{acyclic}(cn) := \bigwedge \{\mathsf{acyclic}^*(cn') \mid cn' \in \mathsf{reaches}(cn)\}$$
$$\mathsf{acyclic}^*(cn) := cn \notin \bigcup_{i>0} \mathsf{reaches}^i(cn)$$

Next, we define the abstract property space of the analysis.

Definition 8.10. Let τ be a type environment. The set of variables, whose static type is acyclic, is defined by:

$$\mathsf{AC}^\tau := \{v \in \mathsf{dom}(\tau) \mid \text{ type of } v \text{ is acyclic}\} .$$

The abstract domain of acyclic variables AC is:

$$\mathsf{AC}^\tau := \{ac \in \mathcal{P}(\mathsf{dom}(\tau)) \mid \mathsf{AC}^\tau \subseteq ac\} .$$

Furthermore, $\mathsf{AC} := (\mathsf{AC}^\tau, \supseteq, \bigcup, \bigcap, \mathsf{dom}(\tau), AC^\tau)$ is a complete lattice.

The acyclicity analysis is a must analysis and $\mathsf{dom}(\tau) \supseteq \mathsf{AC}^\tau$. We are interested in the greatest subset of $\mathsf{dom}(\tau)$ that holds for all paths in an execution. Values are initialised with the least element $\mathsf{dom}(\tau)$ and abstracted with respect to the order $\supseteq$. Incoming paths are combined using $\bigcap$.

Definition 8.11. Let $ac \subseteq \mathsf{dom}(\tau)$ and $\mathcal{S} \subseteq \mathcal{JS}^*$ be a set of states compatible with τ. We define abstraction $\alpha_{\mathsf{AC}} \colon \mathcal{P}(\mathcal{JS}^*) \to \mathsf{AC}$ and concretisation $\gamma_{\mathsf{AC}} \colon \mathsf{AC} \to \mathcal{P}(\mathcal{JS}^*)$ as follows:

$$\gamma^{\tau}_{\mathsf{AC}}(ac) := \{s \in \mathcal{S} \mid v \in ac \text{ and } v \text{ is acyclic in } s\} \text{ , and}$$
$$\alpha^{\tau}_{\mathsf{AC}}(\mathcal{S}) := \{v \in \mathsf{dom}(\tau) \mid v \text{ is acyclic in every } s \in \mathcal{S}\} \text{ .}$$

Then, $(\mathcal{P}(\mathcal{JS}^*), \alpha_{\mathsf{AC}}, \gamma_{\mathsf{AC}}, \mathsf{AC})$ is a Galois connection.

We are left to define the transfer functions for AC. First, we define the assignment and putfield operation in a formal manner. Afterwards, we comment on the bytecode instructions.

Definition 8.12. Let v and its variants denote variables.

$$(v := v')(ac) := \begin{cases} ac \cup \{v\} & \text{if } v' \in ac \\ ac \setminus \{v\} & \text{otherwise .} \end{cases}$$

$$(v.f := v')(ac) := \begin{cases} ac & \text{if } v.f \text{ is acyclic} \\ ac & \text{if } v' \in ac \text{ and } v, v' \text{ do not share} \\ ac \setminus \{v'' \mid v, v'' \text{ share}\} & \text{otherwise .} \end{cases}$$

The bytecode instructions $\texttt{Load}^{\mathsf{AC}}$, $\texttt{Store}^{\mathsf{AC}}$, $\texttt{Invoke}^{\mathsf{AC}}$, and $\texttt{Return}^{\mathsf{AC}}$ are simulated via the assignment operation. The data obtained by $\texttt{Push}^{\mathsf{AC}}$ and $\texttt{New}^{\mathsf{AC}}$ is always acyclic. Therefore, we add the corresponding stack variable to the property domain. When the type of the field of a $\texttt{Getfield}^{\mathsf{AC}}$ instruction is acyclic we add the corresponding stack variable to the domain. All other instructions operate on primitive values. Therefore all involved stack variables are acyclic.

8.2. The Prototype: JaT

In this section we comment on the implementation of JaT. The tool consists of three modules:

Jinja. This module provides data type, parser and general functionalities for JBC programs.

JFlow. This module provides the data flow analysis presented in the previous section. In the spirit of [13], we combine all abstract domains to a single abstract domain $\mathsf{TY} \times \mathsf{SH} \times \mathsf{AC}$. Each domain additionally provides an interface that allows to interact with other domains. We call elements of this domain data facts. The implementation of the data flow algorithm follows the graph-free approach presented in [34]. The analysis is context-sensitive and the result allows to query data facts for some arbitrary program location, that is a list of triples (cn, mn, pc). This is achieved by introducing *value based contexts*. The idea is that the result of a specific method call does not depend on the program

location but only on the context (or value) of the call-site. Hence, a mapping from contexts to data facts of a method call is established. When encountering method calls during analysis the algorithm first checks, whether the method is already analysed under the current context. If so, we obtain the return value from the mapping and can proceed. Otherwise, we first compute the fixed point of the method before updating the context mapping.

Jat. This module provides the transformation from bytecode programs to cTRSs. Figure 8.2 depicts the overall design choice of the implementation.

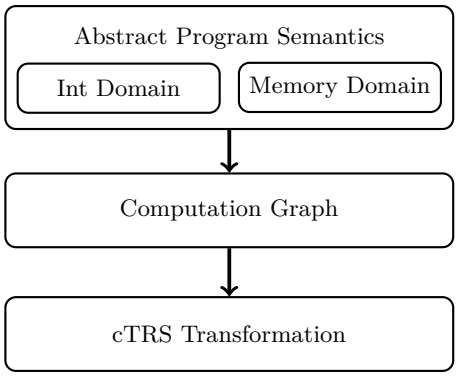

Figure 8.2.: JaT: An Overview.

As defined in Chapter 5 an abstract computation can be a refinement or a symbolic evaluation. Refinements only depend on the current state. Hence an abstract instruction should return either a finite set of refined states or a single state obtained from symbolic evaluation. To apply different approaches we make the abstract program semantics dependent on an *int domain* and a *memory domain*. An int domain provides refinement and evaluation steps for integer operations as well as integer relations. Currently only a simple domain that keeps track of the sign of integer variables is implemented. This allows to evaluate conditions like $x < y$ when x is known to be negative and y is known to be positive for example. This may result in a more concise graph representation. In the future the incorporation of more sophisticated domains is planned. A memory domain provides refinement and evaluation steps for operations accessing and modifying the heap. Currently we have implemented two domains: (i) the *sharing domain* as introduced in Chapter 5 augmented with the data facts obtained from a pre-analysis of the `JFlow` module, and (ii) the *unsharing domain* as presented in [11, 39].

Given the clear separation of the semantic operations the construction of the computation graph and the transformation to cTRSs is rather straightforward.

Figure 8.3 depicts the `flatten` example we already introduced in Chapter 1. We want to discuss the analysis of this program in more detail.

Suppose the input argument *list* is acyclic. Our pre-analysis fails to show that variable *cur* remains acyclic, therefore termination can not be shown. Due to the assignments in lines 3 and 12, *oldCur* shares with *cur* and *tree*. After

```
class IntList{
  IntList next;
  int value;
}

class Tree{
  Tree left;
  Tree right;
  int value;
}

class TreeList{
  TreeList next;
  Tree value;
}
```

```
class Flatten {
  IntList flatten(TreeList list){
    TreeList cur = list;
    IntList result = null;
    while (cur != null){
      Tree tree = cur.value;
      if (tree != null) {
        IntList oldIntList = result;
        result = new IntList();
        result.value = tree.value;
        result.next = oldIntList;
        TreeList oldCur = cur;
        cur = new TreeList();
        cur.value = tree.left;
        cur.next = oldCur;
        oldCur.value = tree.right;
      } else {
        cur = cur.next;
      }
    }
    return result;
  }
}
```

Figure 8.3.: The `flatten` program.

the field update in line 14, *cur* shares with *oldCur*. Therefore acyclicity can not be shown any more for *cur* after the update in line 15.

The automatic analysis presented in [11] also fails to show the desired property. However, the actual implementation in the AProVE tool incorporates additional reachability information to handle such examples.

A small modification of the example and our acyclicity analysis allows us to show that *cur* remains acyclic. First we swap the update operations in lines 14 and 15. This does not affect the result of the program. Then *cur* shares with *oldCur* and *tree* after 14. The update $cur.value = tree.left$ can make *cur* not cyclic, if we consider that a `Tree` object never reaches a `TreeList` object. The order of the field update is important as *cur* shares with *oldCur* after updating the *value* field.

Obviously this is not quite satisfactory, as we actually analyse a different program. A more sophisticated domain that incorporates aliasing and reachability information, as presented in [21], is able to show that *cur* remains acyclic. Here, *oldCur* does not alias or reaches *cur* after the field update in line 14. Therefore, *cur* can not get cyclic when setting $cur.next = oldCur$.

As it turned out, it is not enough to show that *cur* remains acyclic. It is also necessary to show that the remaining `TreeList` of $cur.next.next$, which is abstracted to a variable, is unaffected by the field update $oldCur.value = tree.right$. We have to check if the variable reaches *oldCur*. We argue that this is not possible as it would contradict the acyclicity of *oldCur*.

TCT is now able to show that `flatten` has a linear runtime complexity.

To test the viability of our approach is beyond the scope of this thesis as techniques to handle cTRSs in TCT are yet in development. Though, we want to comment on some general observations: The `flatten` example shows that sharing, acyclicity and reachability information are essential and fine-tuning is still necessary. We also consider to incorporate aforementioned domains into the computation graph method, as refinements could also be applied to the other domains.

The `append` example shows that even if the analysis is exact, we have to account for side-effects as only a finite part of the heap is considered in the abstraction. Suppose that we would iterate through *this* after appending another list. As we require that fresh variables are instantiated with normal-forms, we would still be able to prove termination of the program, since it is enough to consider each loop individually. The same reasoning does not apply for complexity analysis. It would be necessary to provide and incorporate an upper bound on the size of *this* after the update.

Note that a single program statement in Jinja is compiled into multiple instructions and therefore bytecode programs are usually bigger than source code programs. Furthermore, due to class refinements the computation graph can grow exponentially with respect to the number of bytecode instructions: Consider two classes `A` and `B` such that `A` is a superclass of `B`. Both classes define i methods $\mathtt{m}_{1 \leqslant k \leqslant i}\mathtt{()}$. In each method $\mathtt{m}_k\mathtt{()}$ we can enforce a class variable a using a `while` loop, and invoke $\mathtt{m}_{k+1}\mathtt{()}$ after refining a. The current method is not able merge nodes resulting from this construction as the program locations for each call differs.

The cTRS of `flatten` has 110 rules in total. TCT is able to infer a linear bound automatically. Though, many techniques require to incorporate all rules and TCT needs over 3 minutes. In practice, we will combine rules such that a single rule corresponds to a sequence of instructions. We then obtain a linear speed-up with respect to the original system. The main result still holds. Such simplifications have already been considered in [17, 39] and the current prototype also provides a (experimental) transformation. Then `flatten` consists of only 8 rules and TCT can infer the bound immediately.

9. Conclusion and Future Work

In this thesis we defined a representation of JBC executions as computation graphs from which we obtain a representation of JBC executions as constrained rewrite systems. A similar technique to computation graphs was introduced before in [11, 39] to prove termination of bytecode programs. We studied the connection to standard techniques from data flow analysis, and express the relationship between sets of program states and abstract states via a Galois insertion. Abstract states are represented as graphs incorporating (sorted) variables to abstract values and objects. Computation graphs correspond to control flow graphs, where the property domain is fixed and nodes are obtained by dynamically expanding the graph using the abstract semantics.

The computation graph construction is proven to be finite, but (currently) limited to non-recursive bytecode programs. The transformation to cTRSs requires sharing and shape information as only acyclic data can be expressed as terms and side-effects of update operations have to be taken into account. Constraints are used to express integer and Boolean operations on (abstract) values. We have shown that the resulting transformation is complexity preserving, thus upper bounds of bytecode programs can be established by analysing the complexity of the resulting rewriting system.

The prototype implementation JaT makes use of type [30], sharing [44] and acyclicity [42] analyses to express properties on heap elements. The viability of our approach has still to be shown through experiments, as current methods in complexity analysis of term rewrite systems do not support constraints.

Future work will be dedicated towards methods for complexity analysis of cTRSs and generalising the approach to recursive programs.

Bibliography

[1] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *JAR*, 46:161–203, 2011.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *Proc. 16th ESOP*, volume 4421 of *LNCS*, pages 157–172. Springer Verlag, 2007.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *Proc. 6th FMCO*, volume 5382 of *LNCS*, pages 113–132. Springer Verlag, 2008.

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *TCS*, 413:142–159, 2012.

[5] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap Space Analysis for Garbage Collected Languages. *SCP*, 78:1427–1448, 2013.

[6] R. Atkey. Amortised Resource Analysis with Separation Logic. In *Proc. 19th ESOP*, volume 6012 of *LNCS*, pages 85–103. Springer Verlag, 2010.

[7] M. Avanzini and G. Moser. Closing the Gap Between Runtime Complexity and Polytime Computability. In *Proc. 21th RTA*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[8] M. Avanzini and G. Moser. A Combination Framework for Complexity. In *Proc. 24th RTA*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[9] M. Brockschmidt, R. Musiol, C. Otto, and J. Giesl. Automated Termination Proofs for Java Bytecode with Cyclic Data. In *Proc. 24th CAV*, volume 7358 of *LNCS*, pages 105–122, 2012.

[10] M. Brockschmidt, C. Otto, and J. Giesl. Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting. In *Proc. 22nd RTA*, volume 10 of *LIPIcs*, pages 155–170, 2011.

[11] M. Brockschmidt, C. Otto, C. von Essen, and J. Giesl. Termination Graphs for Java Bytecode. In *Verification, Induction, Termination Analysis*, volume 6463 of *LNCS*, pages 17–37. Springer Verlag, 2010.

[12] M. Brockschmidt, T. Ströder, C. Otto, and J. Giesl. Automated Detection of Non-termination and NullPointerExceptions for Java Bytecode. In *FoVeOOS*, volume 7421 of *LNCS*, pages 123–141. Springer Verlag, 2011.

[13] A. Cortesi, B. L. Charlier, and P. V. Hentenryck. Combinations of Abstract Domains for Logic Programming: Open Product and Generic Pattern Construction. *SCP*, 38:27 – 71, 2000.

[14] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of 4th POPL*, pages 238–252, 1977.

[15] J. C.Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. 17th LICS*, pages 55–74. IEEE Computer Society, 2002.

[16] S. Falke and D. Kapur. A Term Rewriting Approach to the Automated Termination Analysis of Imperative Programs. In *Proc. 22nd CADE*, volume 5663 of *LNCS*, pages 277–293. Springer Verlag, 2009.

[17] S. Falke, D. Kapur, and C. Sinz. Termination Analysis of C Programs Using Compiler Intermediate Languages. In *Proc. 22nd RTA*, volume 10 of *LIPIcs*, pages 41–50. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[18] S. Falke, D. Kapur, and C. Sinz. Termination Analysis of Imperative Programs Using Bitvector Arithmetic. In *Proc. 4th VSTTE*, volume 7152 of *LNCS*, pages 261–277, 2012.

[19] A. Farzan, F. Chen, J. Meseguer, and G. Roşu. Formal Analysis of Java Programs in JavaFAN. In *Proc. 16th CAV*, volume 3114 of *LNCS*, pages 501–505. Springer Verlag, 2004.

[20] D. Fenacci and K. MacKenzie. Static Resource Analysis for Java Bytecode Using Amortisation and Separation Logic. *ENTCS*, 279:19 – 32, 2011. Proc. 6th BYTECODE.

[21] S. Genaim and D. Zanardini. Reachability-Based Acyclicity Analysis by Abstract Interpretation. *TCS*, 474:60–79, 2013.

[22] S. Gulwani. SPEED: Symbolic Complexity Bound Analysis. In *Proc. 21st CAV*, volume 5643 of *LNCS*, pages 51–62. Springer Verlag, 2009.

[23] S. Gulwani, K. Mehra, and T. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proc. 36th POPL*, pages 127–139. ACM, 2009.

[24] S. Gulwani and A. Tiwari. Combining Abstract Interpreters. In *Proc. PLDI*, pages 376–386. ACM, 2006.

[25] N. Hirokawa and G. Moser. Automated Complexity Analysis Based on the Dependency Pair Method. In *Proc. 4th IJCAR*, volume 5195 of *LNCS*, pages 364–380, 2008.

[26] N. Hirokawa and G. Moser. Complexity, Graphs, and the Dependency Pair Method. In *Proc. 15th LPAR*, volume 5330 of *LNCS*, pages 652–666. Springer Verlag, 2008.

[27] M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis. In *Proc. 15th ESOP*, volume 3942 of *LNCS*, pages 22–37. Springer Verlag, 2006.

[28] M. Hofmann and D. Rodriguez. Automatic Type Inference for Amortised Heap-Space Analysis. In *Proc. 22nd ESOP*, volume 7792 of *LNCS*, pages 593–613. Springer Verlag, 2013.

[29] P. Hájek. Arithmetical Hierarchy and Complexity of Computation. *TCS*, 8:227 – 237, 1979.

[30] G. Klein and T-Nipkow. A Machine-Checked Model for a Java-like Language, Virtual Machine, and Compiler. *ACM TOPLAS*, 28:619–695, 2006.

[31] C. Kop and N. Nishida. Term Rewriting with Logical Constraints. In *Proc. 9th FroCos*, volume 8152 of *LNCS*, pages 343–358. Springer Verlag, 2013.

[32] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The Size-Change Principle for Program Termination. In *Proc. of 28th POPL*, volume 28, pages 81–92. ACM, 2001.

[33] A. Lochbihler. Jinja With Threads. In *The Archive of Formal Proofs*. `http://afp.sf.net/entries/JinjaThreads.shtml`, 2007. Formal proof development.

[34] M. Mohnen. A Graph-Free Approach to Data-Flow Analysis. In *Proc. 11th CC*, volume 2304 of *LNCS*, pages 46–61. Springer Verlag, 2002.

[35] G. Moser. Proof Theory at Work: Complexity Analysis of Term Rewrite Systems. *CoRR*, abs/0907.5527, 2009. Habilitation Thesis.

[36] G. Nelson and D. C. Oppen. Simplification by Cooperating Decision Procedures. *TOPLAS*, 1:245–257, 1979.

[37] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Verlag, 2005.

[38] L. Noschinski, F. Emmes, and J. Giesl. A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems. In *Proc. 23rd CADE*, volume 6803 of *LNCS*, pages 422–438. Springer Verlag, 2011.

[39] C. Otto, M. Brockschmidt, C. v. Essen, and J. Giesl. Automated Termination Analysis of Java Bytecode by Term Rewriting. In *Proc. 21th RTA*, LIPIcs, pages 259–276. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[40] P. M. Hill, E. Payet and F. Spoto. Path-length Analysis of Object-Oriented Programs. In *In Proc. 1st EAAI*. Elsevier, 2006.

[41] S. E. Panitz and M. Schmidt-Schauß. TEA: Automatically Proving Termination of Programs in a Non-Strict Higher-Order Functional Language. In *Proc. 4th SAS*, volume 1302 of *LNCS*, pages 345–360. Springer Verlag, 1997.

[42] S. Rossignoli and F. Spoto. Detecting Non-cyclicity by Abstract Compilation into Boolean Functions. In *Proc. 7th VMCAI*, volume 3855 of *LNCS*, pages 95–110. Springer Verlag, 2006.

[43] P. Schneider-Kamp, J. Giesl, T. Ströder, A. Serebrenik, and R. Thiemann. Automated Termination Analysis for Logic Programs with Cut*. *TPLP*, 10:365–381, 2010.

[44] S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In *Proc. 12th SAS*, volume 3672 of *LNCS*, pages 320–335. Springer Verlag, 2005.

[45] F. Spoto. Julia: A Generic Static Analyser for the Java Bytecode. In *Proc. of 7th FTfJP*, 2005.

[46] F. Spoto, F. Mesnard, and É. Payet. A Termination Analyzer for Java Bytecode Based on Path-Length. *ACM TOPLAS*, 32:8:1–8:70, 2010.

[47] S. Swiderski and P. Schneider-Kamp. Automated Termination Analysis for Haskell: From Term Rewriting to Programming Languages. In *Proc. 17th RTA*, volume 4098 of *LNCS*, pages 297–312. Springer Verlag, 2006.

[48] A. Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[49] F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound Analysis of Imperative Programs with the Size-Change Abstraction. In *Proc. 18th SAS*, volume 6887 of *LNCS*, pages 280–297, 2011.

A. Semantics of Jinja Bytecode Instructions

$$\texttt{Load}\ n \quad \frac{(heap, (stk, loc, cn, mn, pc) : frms)}{(heap, (loc(n) : stk, loc, cn, mn, pc + 1) : frms)}$$

$$\texttt{Store}\ n \quad \frac{(heap, (v : stk, loc, cn, mn, pc) : frms)}{(heap, (stk, loc\{n \mapsto v\}, cn, mn, pc + 1) : frms)}$$

$$\texttt{Push}\ v \quad \frac{(heap, (stk, loc, cn, mn, pc) : frms)}{(heap, (v : stk, loc, cn, mn, pc + 1) : frms)}$$

$$\texttt{Pop} \quad \frac{(heap, (v : stk, loc, cn, mn, pc) : frms)}{(heap, (stk, loc, cn, mn, pc + 1) : frms)}$$

We use `BOp` together with $\otimes = \{+, -, \vee, \wedge, \geqslant, ==, \neq\}$ to define instructions `IAdd`, `ISub`, `BOr`, `BAnd`, `ICmpGt`, `CmpEq` and `CmpNeq`.

$$\texttt{BOp} \quad \frac{(heap, (v_1 : v_2 : stk, loc, cn, mn, pc) : frms)}{(heap, (v_2 \otimes v_1 : stk, loc, cn, mn, pc + 1) : frms)}$$

$$\texttt{BNot} \quad \frac{(heap, (b : stk, loc, cn, mn, pc) : frms)}{(heap, (\neg b : stk, loc, cn, mn, pc + 1) : frms)}$$

$$\texttt{IfFalse}\ i \quad \frac{(heap, (\mathsf{false} : stk, loc, cn, mn, pc) : frms)}{(heap, (stk, loc, cn, mn, pc + i) : frms)}$$

$$\frac{(heap, (\mathsf{true} : stk, loc, cn, mn, pc) : frms)}{(heap, (stk, loc, cn, mn, pc + 1) : frms)}$$

$$\texttt{Goto}\ i \quad \frac{(heap, (stk, loc, cn, mn, pc) : frms)}{(heap, (stk, loc, cn, mn, pc + i) : frms)}$$

`New` cn' creates a new instance obj of class cn'. The fields of obj are instantiated with the default values, ie., 0 for int, false for bool and null otherwise. Instance obj is mapped to by a fresh address a in $heap$. `Getfield` $fn\ cn'$ access field (cn', fn) of $\mathsf{ft}(heap(a))$. `Putfield` $fn\ cn'$ updates field (cn', fn) in $(cn'', ftable) = heap(a)$ with value v. `Checkcast` cn' fails if $cn' \preceq cn$ does not hold. `Getfield` and `Putfield` fail if a is null.

$$\texttt{New}\ cn' \quad \frac{(heap, (stk, loc, cn, mn, pc) : frms)}{(heap\{a \mapsto obj\}, (a : stk, loc, cn, mn, pc + 1) : frms)}$$

$$\texttt{Getfield}\ fn\ cn' \quad \frac{(heap, (a : stk, loc, cn, mn, pc) : frms)}{(heap, (ftable(cn', fn) : stk, loc, cn, mn, pc + 1) : frms)}$$

$$\texttt{Putfield}\ fn\ cn' \quad \frac{(heap, (v : a : stk, loc, cn, mn, pc) : frms)}{(heap\{a \mapsto (cn'', ftable')\}, (stk, loc, cn, mn, pc + 1) : frms)}$$

$$\texttt{Checkcast}\ cn' \quad \frac{(heap, (cn : stk, loc, cn, mn, pc) : frms)}{(heap, (cn : stk, loc, cn, mn, pc + 1) : frms)}$$

`Invoke` mn' n inspects the type of $heap(a)$, and performs a bottom-up search (with respect to the subclass hierarchy) for the first method declaration mn'. The new frame is $frm' = (\epsilon, loc, cn', mn', 0)$, where loc consists of the $this$ reference (address a), parameters $p_0 : \cdots : p_{n-1}$ and mxl registers instantiated with `unit` (mxl is defined in the method declaration), and cn' denotes the class where mn' is declared. The program terminates if `Return` is executed and $frms$ consists of a single frame. Otherwise, the top frame is dropped and the next frame updated; frm' drops the parameters and the reference and pushes the return value v onto the stack.

$$\texttt{Invoke}\ mn'\ n \quad \frac{(heap, (p_{n-1} : \cdots : p_0 : a : stk, loc, cn, mn, pc) : frms)}{(heap, frm' : (p_{n-1} : \cdots : p_0 : a : stk, loc, cn, mn, pc) : frms)}$$

$$\texttt{Return} \quad \frac{(heap, [frm])}{(heap, [])} \qquad \frac{(heap, (v : stk, loc, cn, mn, pc) : frm : frms)}{(heap, frm' : frms)}$$