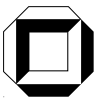
Universität Karlsruhe (TH)
Forschungsuniversität · gegründet 1825

Fakultät für Informatik
Institut für Theoretische Informatik

Diplomarbeit

Inferring Invariants by Static Analysis in KeY

Benjamin Weiß

30. März 2007

Betreuer: Prof. Dr. Peter H. Schmitt

Danksagung

Ich möchte mich ganz herzlich bei meinem Betreuer Prof. Dr. Peter H. Schmitt für das spannende Thema und seine Unterstützung beim Erstellen der Arbeit bedanken. Mein besonderer Dank gilt außerdem Richard Bubel für die vielen hilfreichen Anregungen, die ich von ihm bekam; Achim Kuwertz für seine Arbeit an der SMT-Schnittstelle; Margarete Sackmann fürs sorgfältige Korrekturlesen dieser Arbeit; und Holger Stenger für die Einführung in das Strategie-Rahmenwerk von KeY, die er mir gab. Schließlich danke ich dem gesamten KeY-Team für die hervorragende Arbeitsatmosphäre in den vergangenen zweieinhalb Jahren.

Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Benjamin Weiß
Karlsruhe, den 30. März 2007

Deutsche Zusammenfassung

Diese Arbeit handelt von Invarianten, also Formeln über den Variablen eines Programms, die immer dann gelten, wenn die Ausführung des Programms einen bestimmten Punkt im Programmcode erreicht. Invarianten spielen eine wichtige Rolle in der Programmverifikation, insbesondere in Form von Schleifeninvarianten, also Invarianten, die zu Beginn eines jeden Durchlaufs einer Schleife gelten.

Der Kontext dieser Arbeit ist KeY, ein System zur deduktiven Verifikation von Java-Programmen. Die Regeln des hierzu verwendeten Sequenzenkalküls führen eine symbolische Ausführung des untersuchten Programmes durch, d.h. eine Ausführung mit Symbolen anstelle von konkreten Werten der Variablen. Schleifen können dabei mit Hilfe einer Invariantenregel behandelt werden, welche die manuelle Spezifikation einer Schleifeninvariante durch den Benutzer des KeY-Systems voraussetzt.

Verschiedene Ansätze zum automatischen Ableiten von Invarianten aus Programmen sind bekannt. Viele davon basieren auf statischer Analyse, d.h. der automatischen Untersuchung des Programms ohne es mit konkreten Werten auszuführen. Insbesondere basieren viele auf der Technik der Datenflußanalyse, die verstanden werden kann als eine wiederholte approximative symbolische Ausführung des Programms bis zum Erreichen eines Fixpunktes.

Diese Arbeit untersucht die Beziehung zwischen Datenflußanalyse und der symbolischen Ausführung im Sequenzenkalkül des KeY-Systems, und schlägt einen Weg vor, eine Datenflußanalyse zum Ableiten von Invarianten innerhalb des Sequenzenkalküls zu realisieren. Die auf diese Weise automatisch bestimmten Schleifeninvarianten können dann beispielsweise zur Verifikation des untersuchten Programms mit dem KeY-System verwendet werden.

Der vorgeschlagene Ansatz basiert auf einer Unterart der Datenflußanalyse namens “predicate abstraction”. Hier ist die symbolische Ausführung selbst nicht approximativ, sondern erfolgt – wie auch im KeY-System – so genau wie möglich. Die notwendige Ungenauigkeit, ohne die das Verfahren nicht terminieren würde, wird mit Hilfe von expliziten Abstraktionsschritten eingeführt, die eine vorgegebenen Menge von Formeln verwenden. Diese Menge von Formeln (hier “Prädikate” genannt) definiert das Vokabular, aus dem die abgeleiteten Invarianten zusammengesetzt werden können. Nach Hinzufügen einer überschaubaren Anzahl von Regeln zum Sequenzenkalkül des KeY-Systems ist es möglich, mit KeY Beweise auf eine Art zu konstruieren, die einer “predicate abstraction”-Analyse entspricht.

Der Ansatz ist prototypisch als Teil des KeY-Systems implementiert worden. Neben den neuen Regeln selbst umfaßt diese Implementierung auch eine Heuristik zum automatischen Generieren von Prädikaten für die Abstraktion, und eine Beweissuchstrategie zum automatischen Anwenden der Regeln. Erste Experimente demonstrieren die Funktionstüchtigkeit der Implementierung und ermöglichen Vergleiche mit anderen Ansätzen zum automatischen Ableiten von Invarianten, wie zum Beispiel dem Werkzeug Daikon.

Contents

1 Introduction

1.1 Invariants

An *invariant* of an imperative program is a formula which always holds when, during the execution of the program, control flow reaches a specific point in the program code. It describes properties of the current *state*, i.e. the current memory contents of the executing computer or virtual machine. This idea goes back at least to [Flo67].

Many programming languages allow to insert formulas which are assumed to be invariants directly into the program in the form of `assert` statements. Invariant specifications given in this or any other way can then be used for various purposes, including documentation, runtime checking, and static verification.

A particularly interesting kind of invariants are *loop invariants*. A loop invariant is a formula which always holds before testing the condition of a loop. Knowing loop invariants can, in addition to the uses stated above, be a necessary prerequisite for statically verifying other specifications of programs containing loops.

The concept of invariants is related to that of *pre-* and *postconditions* of program fragments [Hoa69, Mey92]. A pair of a pre- and a postcondition has the meaning that *if* the program fragment (e.g. a method body) is entered in a state which satisfies the precondition, *then* when exiting it the postcondition must hold. Formulas like postconditions, which are required to hold at a program point only if a precondition has been satisfied before, could be called *relative* invariants to distinguish them from formulas which must hold in absolutely all cases. We will however simply use the term "invariants" for such formulas, too.

Another related concept is that of *class invariants* in object-oriented programs [Mey92]. A class invariant (sometimes also referred to as an *object invariant* or an *instance invariant*) is a formula which is not only required to hold when control flow is at a single, specific program point, like classical invariants, but in a multitude of situations. Exactly defining those situations is non-trivial.[1] In this work we do not consider class invariants.

A helpful tool for thinking about programs and their (classical) invariants are *control flow graphs*. A control flow graph is a directed graph whose nodes are basic commands and whose edges stand for control flow between these commands. A textual sequence of

[1] Cf. "observed-state correctness" in [BHS07].

statements can be transformed into an equivalent control flow graph in a straightforward manner.

Example 1.1. Consider the following program fragment, which computes the maximum element of an integer array (assume `max` and `i` are integer variables, and `a` is an integer array variable):

```
max = 0;
i = 0;
while(i < a.length) {
  if(a[i] > max) max = a[i];
  i++;
}
```

Its control flow graph is depicted in Figure 1.1; interesting program points are labeled with numbers 0 to 6. If we assume that `a` is never `null` when entering the program fragment, then the following formulas are examples for invariants at the numbered program points:

0. $\texttt{a} \neq \texttt{null}$
1. $\texttt{a} \neq \texttt{null} \land \texttt{max} \doteq 0$
2. $\texttt{a} \neq \texttt{null} \land \forall x; (0 \leq x \land x < \texttt{i} \rightarrow \texttt{a}[x] \leq \texttt{max}) \land 0 \leq \texttt{i}$
3. $\texttt{a} \neq \texttt{null} \land \forall x; (0 \leq x \land x < \texttt{i} \rightarrow \texttt{a}[x] \leq \texttt{max}) \land 0 \leq \texttt{i} \land \texttt{i} < \texttt{a.length}$
4. $\texttt{a} \neq \texttt{null} \land \forall x; (0 \leq x \land x < \texttt{i} \rightarrow \texttt{a}[x] \leq \texttt{max}) \land 0 \leq \texttt{i} \land \texttt{i} < \texttt{a.length} \land \texttt{a[i]} > \texttt{max}$
5. $\texttt{a} \neq \texttt{null} \land \forall x; (0 \leq x \land x \leq \texttt{i} \rightarrow \texttt{a}[x] \leq \texttt{max}) \land 0 \leq \texttt{i} \land \texttt{i} < \texttt{a.length}$
6. $\texttt{a} \neq \texttt{null} \land \forall x; (0 \leq x \land x < \texttt{i} \rightarrow \texttt{a}[x] \leq \texttt{max}) \land 0 \leq \texttt{i} \land \texttt{i} \geq \texttt{a.length}$

In particular, the invariant at (3) is a loop invariant, and the one at (6) is a postcondition of the program fragment.

1.2 KeY

The *KeY* tool [BHS07, ABB+05] is a verification system for programs which are written in the programming language *Java* [GJSB00], or more precisely *Java Card* [JC06]. Java Card is roughly a subset of Java slimmed down for use on smart cards and other embedded systems. Mainly, it lacks concurrency and floating point types. In this work we gloss over these differences and just call the language Java.

KeY can be used as a stand-alone system and as a plugin for the development environments *Borland Together Control Center* and *Eclipse*. Its core is a theorem prover for

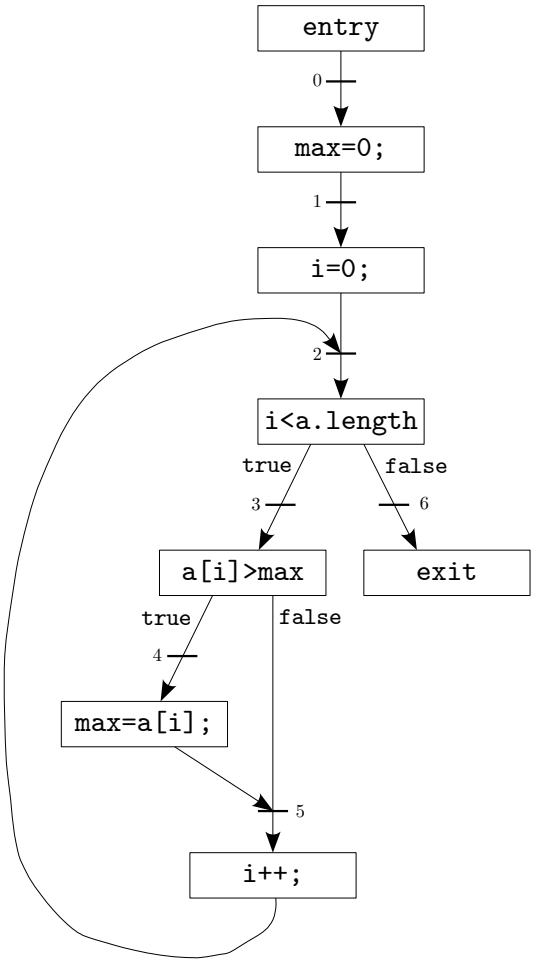

Figure 1.1: Control flow graph for the array maximum program.

formulas expressed in a program logic called *JavaDL* [Bec01]. KeY also includes front-ends which automatically generate such JavaDL formulas from specifications written in the specification languages UML/OCL [OCL06] and JML [LBR06].

The proof process itself is semi-automatic: Most steps can be done automatically, but some require user interaction. An important source of interaction is the treatment of loops, which can in general only be handled if the user supplies a loop invariant (or a more complex inductive condition).

1.3 Goal of This Work

Having to identify loop invariants manually can be tedious and sometimes difficult. This work aims at increasing the degree of automation in the KeY system by adding

an automated invariant inference mechanism. Optimal loop invariants are in general uncomputable [BG01], but this does not mean that it is impossible to unburden the user to a significant extent.

A number of approaches for inferring invariants are reported in the literature; most are based on static analysis techniques originating from compiler construction. The most obvious way to make use of these techniques would be calling an external inference algorithm from the KeY system whenever an invariant is needed. However, there are a number of fundamental similarities between KeY and said static analysis techniques. Investigating these similarities, and striving for a closer integration between the inference mechanism and KeY itself, are additional objectives.

1.4 Outline

Chapter 2 introduces JavaDL in some detail. Then, Chapter 3 provides an overview of relevant static analysis techniques. Chapter 4 is the core of this work: Building on the previous chapters, it describes an approach for a static analysis which infers invariants within KeY itself. A concrete implementation of this approach is then detailed in Chapter 5, and first experimental results with this implementation are reported in Chapter 6. Chapter 7 gives an overview of the considerable body of related work. Finally, Chapter 8 contains a summary and a few ideas for future improvements.

2 JavaDL

This chapter provides an overview of JavaDL, the logic used by the KeY system. JavaDL is a version of dynamic logic [HKT00], which in turn can be seen as an extension of Hoare logic [Hoa69]. As the name suggests, JavaDL is tailored to the Java programming language.

JavaDL is a typed logic, i.e. each term has a type on the syntactic level. However, an untyped version is presented here in order to avoid overly lengthy definitions. A full account can be found in [BHS07]. Apart from this simplification, the presentation in this chapter roughly follows [BHS07] and sometimes [Rüm06].

The syntax of JavaDL is defined in Section 2.1, its semantics in Section 2.2. Section 2.3 contains a few observations about JavaDL that are needed in later chapters. Finally, Section 2.4 sketches the sequent calculus used in KeY for reasoning about JavaDL formulas.

2.1 Syntax

The foundation for the definition of the syntax are *signatures*, which fix the available function and predicate symbols. JavaDL distinguishes between *rigid* symbols, which have the same semantics in all states, and *non-rigid* symbols, whose semantics can be different from state to state.

Definition (Signatures). A *signature* is a tuple $(\Pi, \mathcal{V}, \mathcal{F}, \mathcal{P}, \mathit{arity})$ consisting of

- a Java program Π, i.e. a set of class and interface declarations,
- a set $\mathcal{V}$ of variables,
- a set $\mathcal{F}$ of function symbols, partitioned into the sets $\mathcal{F}^r$ of rigid function symbols and $\mathcal{F}^{nr}$ of non-rigid function symbols,
- a set $\mathcal{P}$ of predicate symbols, partitioned into the sets $\mathcal{P}^r$ of rigid predicate symbols and $\mathcal{P}^{nr}$ of non-rigid function symbols,
- a function $\mathit{arity} : \mathcal{F} \cup \mathcal{P} \to \mathbb{N}$ fixing the arities of the function and predicate symbols.

Every signature contains at least the following function and predicate symbols (whose arities, where not explicitly mentioned, are defined as usual):

- for all $z \in \mathbb{Z}$: $z \in \mathcal{F}^r$
- $+, -, *, \texttt{null}, \texttt{true}, \texttt{false} \in \mathcal{F}^r$
- $\doteq, \not\doteq, <, \leq, >, \geq, \mathit{true}, \mathit{false} \in \mathcal{P}^r$
- for all local program variables and static fields $\texttt{x}$ declared in Π: $\texttt{x} \in \mathcal{F}^{nr}$, $\mathit{arity}(\texttt{x}) = 0$
- for all instance fields $\texttt{f}$ declared in Π: $\texttt{f} \in \mathcal{F}^{nr}$, $\mathit{arity}(\texttt{f}) = 1$
- there is a special array access symbol $\texttt{[]} \in \mathcal{F}^{nr}$, $\mathit{arity}(\texttt{[]}) = 2$

Nullary function symbols are also called constant symbols. The local variables and fields of the Java program are represented in JavaDL as non-rigid function symbols. For the rest of this document, we assume a given signature $(\Pi, \mathcal{V}, \mathcal{F}, \mathcal{P}, \mathit{arity})$.

The basic syntactical elements of JavaDL are *terms*, *formulas*, and another concept called *updates*. The definitions of these elements are mutually dependent; nevertheless, we define them one after the other.

2.1.1 Terms

Definition (Terms). The set *Terms* of terms is the smallest set such that

- for all $x \in \mathcal{V}$: $x \in \mathit{Terms}$
- for all $f \in \mathcal{F}$, $\mathit{arity}(f) = n$, $t_1, \ldots, t_n \in \mathit{Terms}$: $f(t_1, \ldots, t_n) \in \mathit{Terms}$
- for all $\varphi \in \mathit{Formulas}$, $t_1, t_2 \in \mathit{Terms}$: $\mathit{if}(\varphi)\mathit{then}(t_1)\mathit{else}(t_2) \in \mathit{Terms}$
- for all $u \in \mathit{Updates}$, $t \in \mathit{Terms}$: $\{u\}t \in \mathit{Terms}$

For instance fields $\texttt{f}$, we often write $t.\texttt{f}$ instead of $\texttt{f}(t)$. Similarly, for the array access symbol $\texttt{[]}$, we write $t_1\texttt{[}t_2\texttt{]}$ instead of $\texttt{[]}(t_1, t_2)$. Side-effect free Java expressions are identified with terms; for example, the Java expression `this.f` is considered identical to the JavaDL term `this.f`.

As an aside, the JavaDL counterpart for Java expressions of type `boolean` is sometimes actually a formula, not a term: While, for example, the expression `b` (where `b` is a boolean Java variable) is a term, the expression `this.f <= 27` is a formula, because $\leq$ is a predicate symbol, not a function symbol. However, this subtlety is not of much consequence, and we ignore it in this chapter. The issue comes up again briefly in Chapter 4.

2.1.2 Formulas

Definition (Formulas). The set *Formulas* of formulas is the smallest set such that

- for all $p \in \mathcal{P}$, $arity(p) = n$, $t_1, \ldots, t_n \in Terms$: $p(t_1, \ldots, t_n) \in Formulas$
- for all $\varphi \in Formulas$: $\neg\varphi \in Formulas$
- for all $\varphi_1, \varphi_2 \in Formulas$: $(\varphi_1 \wedge \varphi_2) \in Formulas$, $(\varphi_1 \vee \varphi_2) \in Formulas$, $(\varphi_1 \rightarrow \varphi_2) \in Formulas$, $(\varphi_1 \leftrightarrow \varphi_2) \in Formulas$
- for all $\varphi_1, \varphi_2, \varphi_3 \in Formulas$: $if(\varphi_1)then(\varphi_2)else(\varphi_3) \in Formulas$
- for all $x \in \mathcal{V}$, $\varphi \in Formulas$: $(\forall x; \varphi) \in Formulas$, $(\exists x; \varphi) \in Formulas$
- for all $u \in Updates, \varphi \in Formulas$: $\{u\}\varphi \in Formulas$
- for all sequences of statements `p` valid in the context of Π [GJSB00], and all $\varphi \in Formulas$: $[\mathtt{p}]\varphi \in Formulas$, $\langle\mathtt{p}\rangle\varphi \in Formulas$

Besides updates, the elements going beyond first-order logic here are the *modalities* $[\cdot]$ (called "box") and $\langle\cdot\rangle$ (called "diamond"). A formula $[\mathtt{p}]\varphi$ expresses that if `p` terminates normally, then the final state satisfies φ (*partial* correctness). A formula $\langle\mathtt{p}\rangle\varphi$ expresses that `p` terminates normally and that the final state satisfies φ (*total* correctness). *Normal* termination means termination without throwing an exception.

In this work, we only make use of the box modality. The underlying reason is that we are concerned with invariants, and invariants are fundamentally unconnected with termination: They are *safety* properties, i.e. statements that something holds *if* a program point is reached. Termination on the other hand is a *liveness* property, i.e. a requirement that a program point *will* eventually be reached.

Example 2.1. The formula `i` $\doteq 27 \rightarrow$ `[i = i + 1;]i` $\doteq 28$ expresses that if the local program variable `i` has the value 27 before executing the statement `i = i + 1`, then afterwards its value is 28.

2.1.3 Updates

Definition (Updates). The set *Updates* of updates is the smallest set such that

- for all $f(t_1, \ldots, t_n), t \in Terms$ with $f \in \mathcal{F}^{nr}$: $(f(t_1, \ldots, t_n) := t) \in Updates$ (*function* update)
- for all $u_1, u_2 \in Updates$: $(u_1; u_2) \in Updates$ (*sequential* update)
- for all $u_1, u_2 \in Updates$: $(u_1 || u_2) \in Updates$ (*parallel* update)
- for all $\varphi \in Formulas$, $u \in Updates$: $(if\ \varphi; u) \in Updates$ (*guarded* update)
- for all $x \in \mathcal{V}$, $u \in Updates$: $(for\ x; u) \in Updates$ (*quantified* update)

Updates are similar to the programs occurring in modalities. Function updates correspond to assignments; for example, $\{\texttt{x} := \texttt{e}\}\varphi$ has the same meaning as $[\texttt{x = e;}]\varphi$. Sequential and parallel updates correspond to sequential and parallel composition of statements, respectively, and guarded updates correspond to conditional statements. Quantified updates are a generalisation of parallel updates.

The *targets* of an update are those non-rigid function symbols which occur as top level operators of the left hand sides of its function updates (2^S stands for the power set of the set S):

Definition (Update targets).

$$targets : Updates \to 2^{\mathcal{F}^{nr}}$$

$$targets(u) = \begin{cases} \{f\} & \text{if } u = (f(t_1, \ldots, t_n) := t) \\ targets(u_1) \cup targets(u_2) & \text{if } u = (u_1; u_2) \text{ or } u = (u_1 || u_2) \\ targets(u') & \text{if } u = (\mathit{if}\ \varphi; u') \text{ or } u = (\mathit{for}\ x; u') \end{cases}$$

2.2 Semantics

The semantics of terms, formulas and updates is based on so-called *Kripke structures*:

Definition (Kripke structures). A *Kripke structure* is a triple ($Values, States, \rho$) such that

- *Values* is a set with $\mathbb{Z} \subseteq Values$, $null \in Values$; furthermore it contains an infinite number of (Java) objects
- *States* is the largest set such that for all $s, s' \in States$:
  - for all $f \in \mathcal{F}$, $arity(f) = n$: $s(f)$ is a function $s(f) : Values^n \to Values$
  - for all $p \in \mathcal{P}$, $arity(p) = n$: $s(p)$ is a relation $s(p) \subseteq Values^n$
  - for all $op \in \mathcal{F}^r \cup \mathcal{P}^r$: $s(op) = s'(op)$
  - for all $z \in \mathbb{Z}$: $s(z) = z$
  - $s(+)$, $s(-)$, $s(*)$ are the usual functions; $s(\texttt{null}) = null$
  - $s(\doteq)$, $s(\not\doteq)$, $s(<)$, $s(\leq)$, $s(>)$, $s(\geq)$, $s(true)$, $s(false)$ are the usual relations
- ρ is a function associating with each sequence of statements $\texttt{p}$ a *transition relation* $\rho(\texttt{p}) \subseteq States \times States$ such that $(s_1, s_2) \in \rho(\texttt{p})$ iff $\texttt{p}$, when started in s_1, terminates normally in s_2 (according to [GJSB00]).

States are defined as interpretations of the function and predicate symbols. This is consistent with the intuitive understanding used in Chapter 1, where states were considered to be snapshots of the executing machine's memory: The assignment of values to memory locations is modeled in JavaDL as interpretation of the non-rigid function symbols representing program variables and fields. Note that this common concept of state does *not* include the status of control flow.

Rigid symbols have the same interpretation in all states. In particular, the arithmetic function and predicate symbols are interpreted as usual in every state, just like the symbols for equality, true and false.

Since sequences of Java statements `p` are always deterministic, the transition relation $\rho(\mathtt{p})$ is even a (partial) function: For each $s \in \mathit{States}$, there can be at most one s' such that $(s, s') \in \rho(\mathtt{p})$.

For the rest of this document, we assume a given Kripke structure $(\mathit{Values}, \mathit{States}, \rho)$. Still missing is a concept for interpreting variables; this is done by *variable assignments*:

Definition (Variable assignments). A *variable assignment* is a function $\beta : \mathcal{V} \to \mathit{Values}$. If β is a variable assignment, $x \in \mathcal{V}$, and $v \in \mathit{Values}$, then β^v_x is the variable assignment which is identical to β except that $\beta^v_x(x) = v$.

Like the syntax definitions, the definitions of the semantics of terms, formulas and updates are mutually dependent.

2.2.1 Semantics of Terms

Definition (Semantics of terms). For all $s \in \mathit{States}$ and all variable assignments β:

$$\mathit{val}_{s,\beta} : \mathit{Terms} \to \mathit{Values}$$

$$\mathit{val}_{s,\beta}(t) = \begin{cases} \beta(t) & \text{if } t \in \mathcal{V} \\ s(f)(\mathit{val}_{s,\beta}(t_1), \ldots, \mathit{val}_{s,\beta}(t_n)) & \text{if } t = f(t_1, \ldots, t_n) \\ \mathit{val}_{s,\beta}(t_1) & \text{if } t = \mathit{if}(\varphi)\mathit{then}(t_1)\mathit{else}(t_2) \text{ and } (s,\beta) \models \varphi \\ \mathit{val}_{s,\beta}(t_2) & \text{if } t = \mathit{if}(\varphi)\mathit{then}(t_1)\mathit{else}(t_2) \text{ and } (s,\beta) \not\models \varphi \\ \mathit{val}_{s',\beta}(t) & \text{if } t = \{u\}t, \text{ where } s' = \mathit{val}_{s,\beta}(u)(s) \end{cases}$$

2.2.2 Semantics of Formulas

Definition (Semantics of formulas). For all $s \in \mathit{States}$ and all variable assignments β:

- $(s,\beta) \models p(t_1, \ldots, t_n)$ iff $(\mathit{val}_{s,\beta}(t_1), \ldots, \mathit{val}_{s,\beta}(t_n)) \in s(p)$
- $(s,\beta) \models \neg\varphi_1$ iff $(s,\beta) \not\models \varphi_1$

- $(s, \beta) \models (\varphi_1 \wedge \varphi_2)$ iff $(s, \beta) \models \varphi_1$ and $(s, \beta) \models \varphi_2$; analogously for $(\varphi_1 \vee \varphi_2)$, $(\varphi_1 \rightarrow \varphi_2)$ and $(\varphi_1 \leftrightarrow \varphi_2)$
- $(s, \beta) \models \mathit{if}(\varphi_1)\mathit{then}(\varphi_2)\mathit{else}(\varphi_3)$ iff $(s, \beta) \models (\varphi_1 \rightarrow \varphi_2) \wedge (\neg\varphi_1 \rightarrow \varphi_3)$
- $(s, \beta) \models (\forall x; \varphi)$ iff for all $v \in \mathit{Values}$: $(s, \beta^v_x) \models \varphi$; analogously for $(\exists x; \varphi)$
- $(s, \beta) \models \{u\}\varphi$ iff $(s', \beta) \models \varphi$, where $s' = \mathit{val}_{s,\beta}(u)(s)$
- $(s, \beta) \models [\mathtt{p}]\varphi$ iff for all s' with $(s, s') \in \rho(\mathtt{p})$: $(s', \beta) \models \varphi$
- $(s, \beta) \models \langle\mathtt{p}\rangle\varphi$ iff there is an s' such that $(s, s') \in \rho(\mathtt{p})$ and $(s', \beta) \models \varphi$

$(s, \beta) \models \varphi$ means that φ is valid in the state s for the variable assignment β. If φ does not contain free variables (i.e. variables which are not bound by a quantifier), its validity in a state is independent of the variable assignment. The *models* of a formula are those states in which it is valid (for all variable assignments):

Definition (Models of a formula).

$$\mathit{models} : \mathit{Formulas} \rightarrow 2^{\mathit{States}}$$

$$\mathit{models}(\varphi) = \{s \in \mathit{States} \mid \text{for all } \beta : (s, \beta) \models \varphi\}$$

A formula φ is (logically) valid, denoted by $\models \varphi$, iff $\mathit{models}(\varphi) = \mathit{States}$. For $\models \varphi_1 \rightarrow \varphi_2$, $\models \varphi_2 \rightarrow \varphi_1$ and $\models \varphi_1 \leftrightarrow \varphi_2$ we also write $\varphi_1 \Rightarrow \varphi_2$, $\varphi_1 \Leftarrow \varphi_2$ and $\varphi_1 \Leftrightarrow \varphi_2$, respectively.

2.2.3 Semantics of Updates

The semantics of updates is somewhat more intricate than that of terms and formulas. We first need some preliminary definitions, beginning with *(memory) locations*:

Definition (Locations). The set *Locations* of locations is defined as

$$\mathit{Locations} = \{(f, (v_1, \ldots, v_n)) \mid f \in \mathcal{F}^{nr},\ \mathit{arity}(f) = n,\ v_1, \ldots, v_n \in \mathit{Values}\}$$

Next on the list are *semantic updates*, which is what updates are evaluated to:

Definition (Semantic updates). The set *SemUpdates* of semantic updates is the largest set $\mathit{SemUpdates} \subseteq 2^{\mathit{Locations} \times \mathit{Values}}$ such that for all $U \in \mathit{SemUpdates}$ and all $l \in \mathit{Locations}$ there is at most one $v \in \mathit{Values}$ such that $(l, v) \in U$.

Example 2.2. If $o_1, o_2 \in \mathit{Values}$ are objects, and $\mathtt{f} \in \mathcal{F}^{nr}$ is a field, then $(\mathtt{f}, o_1)$ and $(\mathtt{f}, o_2)$ are locations, and $\{(\mathtt{f}, o_1, 1), (\mathtt{f}, o_2, 2)\}$ is a semantic update. However, $\{(\mathtt{f}, o_1, 1), (\mathtt{f}, o_1, 2)\}$ is not a semantic update.

Semantic updates can also be understood as state-transforming functions:

Definition (Application of semantic updates). For all $U \in \mathit{SemUpdates}$ and all $s \in \mathit{States}$, $U(s) \in \mathit{States}$ is defined by

$$U(s)(f)(v_1, \ldots, v_n) = \begin{cases} v & \text{if } (f, (v_1, \ldots, v_n), v) \in U \\ s(f)(v_1, \ldots, v_n) & \text{otherwise} \end{cases}$$

for all $f \in \mathcal{F}$, $\mathit{arity}(f) = n$, $v_1, \ldots, v_n \in \mathit{Values}$; and by $U(s)(p) = s(p)$ for all $p \in \mathcal{P}$.

If updates are to be evaluated to semantic updates, there has to be a way to resolve clashes, i.e. cases in which one location is assigned several values by the same update. For sequential and parallel updates, this is accomplished by a "last-win semantics": Updates on the right override updates on the left. This behaviour is captured by the *overriding operator*:

Definition (Overriding operator).

$$\oplus : \mathit{SemUpdates} \times \mathit{SemUpdates} \to \mathit{SemUpdates}$$

$$U_1 \oplus U_2 = \{(f, (v_1, \ldots, v_n), v) \in U_1 \mid \text{there is no } (f, (v_1, \ldots, v_n), v') \in U_2\} \cup U_2$$

Now, we can finally define the semantics of updates.

Definition (Semantics of updates). For all $s \in \mathit{States}$ and all variable assignments β:

$$\mathit{val}_{s,\beta} : \mathit{Updates} \to \mathit{SemUpdates}$$

$$\mathit{val}_{s,\beta}(u) = \begin{cases} \{(f, \mathit{val}_{s,\beta}(t_1), \ldots, \mathit{val}_{s,\beta}(t_n), \mathit{val}_{s,\beta}(t))\} & \text{if } u = (f(t_1, \ldots, t_n) := t) \\ \mathit{val}_{s,\beta}(u_1) \oplus \mathit{val}_{\mathit{val}_{s,\beta}(u_1)(s),\beta}(u_2) & \text{if } u = (u_1; u_2) \\ \mathit{val}_{s,\beta}(u_1) \oplus \mathit{val}_{s,\beta}(u_2) & \text{if } u = (u_1 || u_2) \\ \mathit{val}_{s,\beta}(u') & \text{if } u = (\mathit{if}\ \varphi; u') \text{ and } (s, \beta) \models \varphi \\ \emptyset & \text{if } u = (\mathit{if}\ \varphi; u') \text{ and } (s, \beta) \not\models \varphi \\ \mathit{unclash}\big(\bigcup_{v \in \mathit{Values}} \mathit{val}_{s,\beta^v_x}(u')\big) & \text{if } u = (\mathit{for}\ x; u') \end{cases}$$

where $\mathit{unclash} : 2^{\mathit{Locations} \times \mathit{Values}} \to \mathit{SemUpdates}$ is a function which resolves possible clashes by removing elements from its argument. This is achieved by assuming a well-ordering on *Values*, and letting the update which assigns the smallest value according to this well-ordering override the other updates to the same location. The details are defined e.g. in [BHS07], but they are not relevant for this work.

2.3 Observations

In this section, we gather a couple of observations about the entities defined in the previous sections, which are later needed in Chapter 4. The validity of these observations should be obvious, although formally proving them would be rather tedious.

The following lemma states that updates of a certain form completely replace the interpretation of one function symbol by that of another:

Lemma 2.1 (Function-replacing updates). *If $f \in \mathcal{F}^{nr}$, $f' \in \mathcal{F}$ such that $arity(f) = arity(f') = n$, and $u = \big(for\ x_1; \ldots; for\ x_n; f(x_1, \ldots, x_n) := f'(x_1, \ldots, x_n)\big)$, then for all $s \in States$ and all $op \in \mathcal{F} \cup \mathcal{P}$:*

$$val_{s,\beta}(u)(s)(op) = \begin{cases} s(f') & \text{if } op = f \\ s(op) & \text{otherwise} \end{cases}$$

Next, we observe that an update does not affect the interpretation of symbols other than its target symbols:

Lemma 2.2 (Non-targeted symbols). *For all $s \in States$, all variable assignments β, all $u \in Updates$, and all $op \in \mathcal{F} \cup \mathcal{P}$: If $op \notin targets(u)$, then*

$$val_{s,\beta}(u)(s)(op) = s(op)$$

Obviously, the validity of a formula only depends on the interpretation of those symbols which occur in it:

Lemma 2.3 (Formulas in similar states). *For all $s_1, s_2 \in States$, all variable assignments β, and all $\varphi \in Formulas$: If for all $op \in \mathcal{F} \cup \mathcal{P}$ occurring in φ it holds that $s_1(op) = s_2(op)$, then*

$$(s_1, \beta) \models \varphi \quad \text{iff} \quad (s_2, \beta) \models \varphi$$

An analogous statement holds for updates:

Lemma 2.4 (Updates in similar states). *For all $s_1, s_2 \in States$, all variable assignments β, and all $u \in Updates$: If for all $op \in \mathcal{F} \cup \mathcal{P}$ occurring in u it holds that $s_1(op) = s_2(op)$, then*

$$val_{s_1,\beta}(u) = val_{s_2,\beta}(u)$$

2.4 Sequent Calculus

Reasoning about the validity of JavaDL formulas is done in KeY by means of a *sequent calculus*. A *sequent* is a construct of the form $\Gamma \vdash \Delta$, where Γ (called the *antecedent* of the sequent) and Δ (called the *succedent* of the sequent) are sets of closed formulas. Its semantics is the same as that of the formula $\bigwedge \Gamma \rightarrow \bigvee \Delta$. Sequents like $\Gamma \vdash \{\varphi\} \cup \Delta$ are often abbreviated as in $\Gamma \vdash \varphi, \Delta$.

The calculus consists of *rules*, which are constructs of the form

$$\frac{p_1 \ldots p_n}{c}\ name$$

where the p_i (called the *premisses* of the rule) and c (called the *conclusion* of the rule) are sequents. A rule is *sound* if the validity of its premisses implies the validity of its conclusion. Rules are usually given in a schematic way; for example, in

$$\frac{\Gamma \vdash \varphi_1, \Delta \qquad \Gamma \vdash \varphi_2, \Delta}{\Gamma \vdash \varphi_1 \wedge \varphi_2, \Delta} \; \textit{and\_right}$$

φ_1 and φ_2 stand for arbitrary formulas, and Γ and Δ for arbitrary sets of formulas. Thus, *and_right* is actually not a single rule, but an infinite set of rules. We nevertheless call such schemata "rules", too.

A *proof tree* is a tree whose nodes are sequents, such that each sequent's children are the premisses of a rule whose conclusion is the sequent itself. If all leaves of such a tree are trivially valid, and all applied rules are sound, then this constitutes a proof that the root sequent is valid as well.

Modalities are handled by rules which perform a *symbolic execution* of the Java program: The program is "executed" without knowing the concrete values of the program variables. The next statement to be executed in this way is called the *active statement*; it is the first basic command within the modality following a non-active prefix of opening braces, beginnings of `try` blocks and the like. This prefix is usually denoted by π, and the rest of the program behind the active statement by ω. As an example, consider the program fragment below, where the active statement is `i = 0`:

$$\underbrace{\texttt{try\{}}_{\pi} \; \texttt{i = 0;} \; \underbrace{\texttt{i++; \} catch(Exception e)\{ i = 27; \}}}_{\omega}$$

Symbolic execution stepwise shortens the program. Once all modalities have been executed in this way, the leaves of the proof tree contain only formulas without modalities, which are comparatively simple to handle. In the following subsections, we take a look at the rules which symbolically execute the three basic kinds of statements, that is assignments, conditionals, and loops.

2.4.1 Assignments

The execution of assignment statements can be done in one of several ways. A typical rule (based on the assignment rule of Hoare logic) is the following:

Definition (Rule *assign_outer*).

$$\frac{\Gamma', \; \mathtt{x} \doteq \mathtt{e}' \vdash [\pi \, \omega]\psi, \; \Delta'}{\Gamma \vdash [\pi \; \mathtt{x = e;} \; \omega]\psi, \; \Delta} \; \textit{assign\_outer}$$

where `e` must not have side effects, and Γ', $\mathtt{e}'$ and Δ' result from Γ, `e` and Δ, respectively, by substituting a fresh non-rigid constant symbol x' for `x`.

Like many other rules presented in this work, this one requires the occurring Java expression to be free from side-effects. This restriction is never severe, because a program can always be transformed so that an expression is separated from its side-effects.

Example 2.3. The following proof tree contains an application of *assign_outer* (the root sequent of proof trees is displayed at the bottom):

$$\frac{i' \doteq 27,\ \mathtt{i} \doteq i' + 1 \ \vdash\ \mathtt{i} \doteq 28}{\mathtt{i} \doteq 27 \ \vdash\ \mathtt{[i = i + 1;]i} \doteq 28}\ (\mathit{assign\_outer})$$

In *assign_outer*, the assigned variable is renamed in all places which are not affected by the assignment. It is also possible to proceed the other way round, and rename the assigned variable in those places which *are* affected by the assignment:

Definition (Rule *assign_inner*).

$$\frac{\Gamma,\ x' \doteq \mathtt{e} \ \vdash\ [\pi'\,\omega']\psi',\ \Delta}{\Gamma \ \vdash\ [\pi\ \mathtt{x = e;}\ \omega]\psi,\ \Delta}\ \mathit{assign\_inner}$$

where `e` must not have side effects, x' is a fresh non-rigid constant symbol, and π', ω' and ψ' result from π, ω and ψ, respectively, by substituting x' for `x`.

Otherwise, the two rules are very similar. In particular, they share the limitation that only assignments to local variables are supported. Assignments to fields are more complicated because of *aliasing*, i.e. the phenomenon that the same memory location can be referred to by different names. For example, the formula `[a.f = 1; b.f = 2;]a.f` $\doteq 1$ is not valid, because it is possible that `a.f` and `b.f` refer to the same location. While the rules above can be adapted to support complex assignments anyway, this leads to many case distinctions and is thus rather inefficient. In KeY, assignments are instead handled by the rule below.

Definition (Rule *assign_update*).

$$\frac{\Gamma \ \vdash\ \{u;\mathtt{x} := \mathtt{e}\}[\pi\,\omega]\psi,\ \Delta}{\Gamma \ \vdash\ \{u\}[\pi\ \mathtt{x = e;}\ \omega]\psi,\ \Delta}\ \mathit{assign\_update}$$

where `e` must not have side effects.

This rule is related to *assign_inner*, but it does not apply any substitutions immediately; instead, it just turns the assignment into an update. The update which accumulates in this way in front of a modality can be aggressively simplified using a complex rule called *simpl_update*, which we do not formally define here. It is finally applied to the remaining formula once the modality has been executed completely. This approach allows to avoid many case distinctions.

Example 2.4. Consider the following proof tree (whose root sequent is not valid):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash\ \mathit{if}(\mathtt{a} \doteq \mathtt{b})\mathit{then}(2)\mathit{else}(1) \doteq 1}\ (\mathit{simpl\_update})}{\vdash\ \{\mathtt{a.f} := 1; \mathtt{b.f} := 2\}(\mathtt{a.f} \doteq 1)}\ (\mathit{assign\_update})}{\vdash\ \{\mathtt{a.f} := 1;\}[\mathtt{b.f = 2;}]\mathtt{a.f} \doteq 1}\ (\mathit{simpl\_update})}{\vdash\ \{\mathtt{a.f} := 0; \mathtt{a.f} := 1;\}[\mathtt{b.f = 2;}]\mathtt{a.f} \doteq 1}\ (\mathit{assign\_update})}{\vdash\ \{\mathtt{a.f} := 0\}[\mathtt{a.f = 1; b.f = 2;}]\mathtt{a.f} \doteq 1}\ (\mathit{assign\_update})}{\vdash\ [\mathtt{a.f = 0; a.f = 1; b.f = 2;}]\mathtt{a.f} \doteq 1}$$

One after the other, the assignment statements are turned into updates. Since the first function update is overridden by the second, it can by simplified away. As soon as the modality is empty, the update is applied to the formula on its right.

2.4.2 Conditionals

Conditional statements are executed by the following rule:

Definition (Rule *ifthenelse*).

$$\frac{\Gamma\ \vdash\ \{u\}\mathit{if}(\mathtt{e} \doteq \mathtt{true})\mathit{then}([\pi\ \mathtt{p}\ \omega]\psi)\mathit{else}([\pi\ \mathtt{q}\ \omega]\psi),\ \Delta}{\Gamma\ \vdash\ \{u\}[\pi\ \mathtt{if(e)\ p\ else\ q}\ \omega]\psi,\ \Delta}\ \mathit{ifthenelse}$$

where `e` must not have side effects.

It transforms the conditional statement into a conditional formula. In some cases, the sequent may contain the information that the condition is true or that it is false; then a proof split can be avoided. Otherwise, the rule below is used to split the proof tree into one branch for either case.

Definition (Rule *ifthenelse_split*).

$$\frac{\begin{array}{c}\Gamma,\ \{u\}\varphi\ \vdash\ \{u\}\psi_1,\ \Delta\\ \Gamma,\ \{u\}\neg\varphi\ \vdash\ \{u\}\psi_2,\ \Delta\end{array}}{\Gamma\ \vdash\ \{u\}\mathit{if}(\varphi)\mathit{then}(\psi_1)\mathit{else}(\psi_2),\ \Delta}\ \mathit{ifthenelse\_split}$$

2.4.3 Loops

The following rule can be used to execute loops:

Definition (Rule *unwind_while*).

$$\frac{\Gamma\ \vdash\ \{u\}[\pi\ \mathtt{if(e)\{p\ while(e)\ p\}}\ \omega]\psi,\ \Delta}{\Gamma\ \vdash\ \{u\}[\pi\ \mathtt{while(e)\ p}\ \omega]\psi,\ \Delta}\ \mathit{unwind\_while}$$

where `e` must not have side effects and `p` must not use `break` or `continue`.

We do not consider `for` loops in this work, but they are very similar to `while` loops. The rule actually used in KeY can also handle loops containing `break` or `continue` statements.

The unwind rule is enough only for loops which terminate after a statically known and sufficiently small number of iterations. In the general case, loops cannot be handled by symbolic execution alone. One possible solution is using an *invariant rule*, i.e. a rule which makes use of a loop invariant provided by the user.

Before we can define the invariant rule used in KeY [BSS05], we first need to define *location terms* and *anonymising updates*.

Definition (Location terms). The set $\mathit{LocationTerms}$ of location terms is the smallest set such that

- for all $f(t_1, \ldots, t_n) \in \mathit{Terms}$ with $f \in \mathcal{F}^{nr}$: $f(t_1, \ldots, t_n) \in \mathit{LocationTerms}$
- for all $\varphi \in \mathit{Formulas}$, $lt \in \mathit{LocationTerms}$: $(\mathit{if}\ \varphi; lt) \in \mathit{LocationTerms}$
- for all $x \in \mathcal{V}$, $lt \in \mathit{LocationTerms}$: $(\mathit{for}\ x; lt) \in \mathit{LocationTerms}$

Subsets of $\mathit{LocationTerms}$ are also called *modifier sets*.

Location terms and modifier sets syntactically describe sets of locations. They can be used to indicate which locations a program fragment may and which ones it may not change: Informally speaking, a modifier set is *correct* for a program fragment if the program never changes locations not referenced by the elements of the modifier set. For example, a correct modifier set for the array maximum program from Example 1.1 (p. 2) is $\{\texttt{max}, \texttt{i}\}$.

Definition (Anonymising updates). An anonymising update for a given modifier set $\{lt_1, \ldots, lt_n\}$ is an update $\mathit{anon}(lt_1)||\ldots||\mathit{anon}(lt_n)$, where anon is defined by

$$\mathit{anon} : \mathit{LocationTerms} \to \mathit{Updates}$$

$$\mathit{anon}(lt) = \begin{cases} f(t_1, \ldots, t_m) := f'(t_1, \ldots, t_m) & \text{if } lt = f(t_1, \ldots, t_m) \\ (\mathit{if}\ \varphi; \mathit{anon}(lt')) & \text{if } lt = (\mathit{if}\ \varphi; lt') \\ (\mathit{for}\ x; \mathit{anon}(lt')) & \text{if } lt = (\mathit{for}\ x; lt') \end{cases}$$

where for each f, f' is a fresh rigid function symbol with $\mathit{arity}(f) = \mathit{arity}(f')$.

Intuitively, an anonymising update is an update which assigns unknown values to all locations described by a modifier set. Now, we can finally define the invariant rule itself:

Definition (Rule *while_invariant*).

$$\frac{\begin{array}{l}\Gamma \vdash \{u\}\mathit{Inv},\ \Delta \\ \Gamma,\ \{u;v\}(\mathit{Inv} \wedge \texttt{e}) \vdash \{u;v\}[\texttt{p}]\mathit{Inv},\ \Delta \\ \Gamma,\ \{u;v\}(\mathit{Inv} \wedge \neg\texttt{e}) \vdash \{u;v\}[\pi\,\omega]\psi,\ \Delta\end{array}}{\Gamma \vdash \{u\}[\pi\ \texttt{while(e) p}\ \omega]\psi,\ \Delta}\ \mathit{while\_invariant}$$

where `e` must not have side effects, `p` must not terminate abruptly, and v is an anonymising update for a correct modifier set of the loop.

The formula *Inv* has to be provided by the user, such that the three premisses are satisfied: (i) *Inv* holds when the loop is entered, (ii) *Inv* is preserved by the loop body, and (iii) if *Inv* holds when exiting the loop, then after executing the rest of the program ψ holds. (i) and (ii) together mean that *Inv* is a loop invariant. The anonymising update v is used to model the effect of an arbitrary number of loop iterations: All locations which may be modified by the loop are set to unknown values.

3 Static Analysis

Static analysis is the analysis of a program's syntactic structure or runtime behaviour without actually running it. Its counterpart is *dynamic analysis*, which gathers information about a program by executing it on concrete input values. In this sense, KeY is a static analysis tool (even though its logic is called "dynamic"). However, the term is usually applied only to fully automatic methods such as *data-flow analysis* [ASU86, NNH99], a technique traditionally used in compilers to collect information necessary for performing code optimisations. In particular, the collected information can be invariants.

Section 3.1 provides an informal introduction to data-flow analysis. In Section 3.2, this is continued by describing *abstract interpretation*, which is a theoretical underpinning for data-flow analysis. Section 3.3 introduces a related static analysis technique called *predicate abstraction*.

3.1 Data-Flow Analysis

Data-flow analyses calculate properties associated with program points. A data-flow analysis for a program can be defined by giving for each program point a *data-flow equation*. Such an equation specifies how to derive the property of this program point from the properties of its predecessors in the control flow graph.[1] If the program contains loops, these equations will be mutually recursive. A run of the analysis amounts to repeatedly computing the properties of all program points until all of the equations are satisfied.

A classical example of a data-flow analysis is *reaching definitions analysis*, which calculates for each program point the assignment statements which may "reach" the program point. An assignment statement "reaches" a program point during a run of the program if control flow gets from the assignment to the program point without passing another assignment to the same memory location.

Reaching definitions analysis and many other data-flow analyses are concerned with complex properties referring to the computational past or future. However, there is also a class of data-flow analyses which compute pure, local properties of the states occurring at a program point—in other words, these data-flow analyses infer invariants. A simple example is *constant propagation analysis*, which determines which variables have a statically known, constant value whenever control flow reaches the program point.

[1] There are also "backwards" analyses which derive the properties of a program point from its successors.

Example 3.1. Recall the array maximum program introduced in Example 1.1 (p. 2), and in particular its control flow graph shown in Figure 1.1. The results of running a constant propagation analysis on this program are given below:

	Init.	Iteration 1	Iteration 2	Iteration 3
1	*false*	$\texttt{max} \doteq 0$	$\texttt{max} \doteq 0$	$\texttt{max} \doteq 0$
2	*false*	$\texttt{max} \doteq 0, \texttt{i} \doteq 0$	*true*	*true*
3	*false*	$\texttt{max} \doteq 0, \texttt{i} \doteq 0$	*true*	*true*
4	*false*	$\texttt{max} \doteq 0, \texttt{i} \doteq 0$	*true*	*true*
5	*false*	$\texttt{i} \doteq 0$	*true*	*true*
6	*false*	$\texttt{max} \doteq 0, \texttt{i} \doteq 0$	*true*	*true*

Each line corresponds to one of the program points 1 to 6 marked in Figure 1.1. The analysis proceeds as follows:

- Initialisation: All properties are initialised with *false*, meaning that the program point cannot be reached. For the entry point 0 we assume no restrictions, i.e. an invariant of *true*.
- Iteration 1: Beginning with program point 1, we recompute the properties for all program points. Step by step, this means:
  1. The predecessor of program point 1 is point 0. We take its property (*true*) and adapt it to reflect the effects of the assignment `max = 0`: We remove all information about `max` (there is none) and add $\texttt{max} \doteq 0$.
  2. Here there are two predecessors, points 1 and 5, but since the property at point 5 is *false*, only point 1 has to be considered. We get $\texttt{max} \doteq 0$ and $\texttt{i} \doteq 0$.
  3. We simply adopt the property of point 2, because only information about constant values of variables is tracked, not information like $\texttt{i} < \texttt{a.length}$.
  4. Again, we just adopt the property of the predecessor.
  5. Like point 2, this one has two predecessors. Coming directly from 3, we get $\texttt{max} \doteq 0$ and $\texttt{i} \doteq 0$; coming from 4, `max` has been assigned an unknown value, so we only get $\texttt{i} \doteq 0$. We merge these properties to $\texttt{i} \doteq 0$, which holds in either case.
  6. The predecessor here is point 2. Like for points 3 and 4, we simply adopt its property.
- Iteration 2: Again, we recompute all properties. For point 2, we get $\texttt{i} \doteq 0$ if coming from 1, and $\texttt{i} \doteq 1$ if coming from 5. Disjunctions are not allowed; the only allowed property which holds in either case is *true*. This weakest of all invariants then propagates to all successors of point 2.
- Iteration 3: Once again, we recompute all properties. Thereby we note that nothing changes any more, so we are done.

The constant propagation example illustrates the basic mode of operation of data-flow analyses, which, like the proof process in KeY, can be understood as a kind of symbolic execution of the analysed program. In the case of constant propagation, the invariants which can be found are clearly too weak to be overly useful, but there are other data-flow analyses which can infer more complex invariants. A generic term for data-flow analyses which infer invariants is "abstract interpretations".[2]

3.2 Abstract Interpretation

Besides denoting an invariant-inferring data-flow analysis, *abstract interpretation* is also the name of a theoretical framework for such analyses introduced by Cousot and Cousot [CC77, JN95]. This section provides an informal overview of this framework.

The domain of properties considered by an abstract interpretation is usually a partially ordered set $(D, \sqsubseteq, \bot)$ with least element $\bot$. The partial order $\sqsubseteq$ is chosen such that smaller elements are more "precise". One is then interested in finding for each program point the least element of D which always holds there. We denote by $\sqsubseteq^n$ the pointwise extension of $\sqsubseteq$ to D^n , and $\bot^n = (\bot, \ldots, \bot)$ (where $n \in \mathbb{N}$ is the number of considered program points). The analysis can then be seen as the repeated application of a monotonic function $next : D^n \to D^n$ (called *transition function*) with $\bot^n$ as initial argument until a fixed point is reached. The transition function depends on the analysed program.

The most precise imaginable abstract interpretation is called the *static semantics* or *accumulating semantics*. Its domain is $(2^{States}, \subseteq, \emptyset)$, called the *concrete semantic domain*. That is, the accumulating semantics associates with each program point the exact set of states which can occur there. If we assume that for every relevant set of states S there is a formula φ such that $models(\varphi) = S$, and if we consider equivalent formulas to be identical, then this is the same as finding the strongest invariant for each program point, and we can consider the concrete semantic domain to be $(Formulas, \Rightarrow, false)$.

The accumulating semantics is obviously uncomputable—its transition function can be defined, but no fixed point will be reached in general. In order to be computable, real abstract interpretations use simpler, coarser domains, called *abstract domains*. These can always be viewed as subsets of the concrete semantic domain. For example, the domain of the constant propagation analysis can be understood to be $(A, \Rightarrow, false)$, where $A \subseteq Formulas$ contains all possible conjunctions of formulas $\mathtt{x} \doteq z$, where $\mathtt{x}$ is a program variable and z an integer.

Abstract interpretations can form a hierarchy, where one analysis is an abstraction of another: An analysis with the domain $(A, \sqsubseteq_A, \bot_A)$ and the transition function $next_A : A^n \to A^n$ is a correct abstraction of an analysis with the domain $(C, \sqsubseteq_C, \bot_C)$ and the transition function $next_C : C^n \to C^n$ iff there are functions $\alpha : C \to A$ (called

[2] The term is sometimes applied to other kinds of analyses as well, but we stick with this narrower definition here.

abstraction function) and $\gamma : A \to C$ (called *concretisation function*) such that the following conditions hold:

1. α and γ are monotonic
2. for all $c \in C$: $c \sqsubseteq_C \gamma(\alpha(c))$
3. for all $a \in A$: $a = \alpha(\gamma(a))$
4. for all $\underline{c} \in C^n$: $\alpha^n(next_C(\underline{c})) \sqsubseteq_A^n next_A(\alpha^n(\underline{c}))$
   (where $\alpha^n : C^n \to A^n$ is the pointwise extension of α)

The second condition requires that abstracting can only make an element larger, i.e. less precise, never smaller. The third says that concretising and re-abstracting an already abstract element does not change it. The fourth states that the abstract transition function computes approximations of the concrete transition function; this is illustrated in Figure 3.1. An abstract interpretation is correct iff it is a correct abstraction of the accumulating semantics.

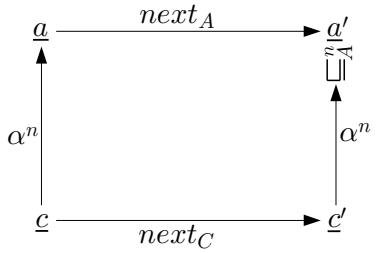

Figure 3.1: Relationship between the abstract and the concrete transition function.

Many abstract domains can be defined by giving a set $P \subseteq \mathit{Formulas}$ of "predicates", which then induces a domain $(A_P, \Rightarrow, \mathit{false})$ by $A_P = \{\bigwedge P' \mid P' \subseteq P\}$. That is, A_P contains all possible conjunctions of the given predicates. A couple of classical abstract domains are defined in this way below (where $\mathit{ProgVars}^{int} \subseteq \mathcal{F}^{nr}$ denotes the set of integer program variables):

- Constants: $P = \{\mathtt{x} \doteq z \mid \mathtt{x} \in \mathit{ProgVars}^{int},\ z \in \mathbb{Z}\}$
- Signs [CC77]: $P = \{0 \leq \mathtt{x}, \mathtt{x} < 0 \mid \mathtt{x} \in \mathit{ProgVars}^{int}\}$
- Intervals [CC77]: $P = \{z_1 \leq \mathtt{x} \wedge \mathtt{x} \leq z_2 \mid \mathtt{x} \in \mathit{ProgVars}^{int},\ z_1, z_2 \in \mathbb{Z}\}$
- Octagons [Min06]: $P = \{\pm\mathtt{x} \pm \mathtt{y} \leq z \mid \mathtt{x}, \mathtt{y} \in \mathit{ProgVars}^{int},\ z \in \mathbb{Z}\}$
- Polyhedra [CH78]: $P = \{\sum_{i=1}^{m} z_i * \mathtt{x}_i \leq z \mid z_1, \ldots, z_m, z \in \mathbb{Z}\}$
  (where $\mathit{ProgVars}^{int} = \{\mathtt{x}_1, \ldots, \mathtt{x}_m\}$)

As defined above, all of these domains only allow constraints on local integer variables. They can nevertheless be used for analysing programs containing fields, but this leads to very coarse overapproximations, since nothing can be said about the values of complex locations. A way of extending such domains to support fields is described in [CL05].

Some abstract domains have the following property, which guarantees that the corresponding analysis terminates:

Definition (Ascending chain condition). A partially ordered set $(D, \sqsubseteq, \bot)$ satisfies the *ascending chain condition* iff for every ascending chain $d_1 \sqsubseteq d_2 \sqsubseteq \ldots$ of elements of D there is an $m_0 \in \mathbb{N}$ such that for all $m > m_0$: $d_m = d_{m_0}$.

If $(D, \sqsubseteq, \bot)$ satisfies the ascending chain condition, then $(D^n, \sqsubseteq^n, \bot^n)$ also does. Since the transition function $next : D^n \to D^n$ is always monotonic, $\bot^n \sqsubseteq^n next(\bot^n) \sqsubseteq^n next(next(\bot^n)) \sqsubseteq^n \ldots$ is an ascending chain. Thus, it is guaranteed in this case that a fixed point is reached eventually. If, in addition, *next* satisfies a continuity requirement, then it is also ensured that this fixed point is the *least* fixed point of *next*, i.e. the most precise result possible within the domain.

The ascending chain condition is satisfied by the constants domain: All ascending chains, for example $(\mathit{false}) \Rightarrow (\mathtt{max} \doteq 0 \wedge \mathtt{i} \doteq 0) \Rightarrow (\mathtt{i} \doteq 0) \Rightarrow (\mathit{true}) \Rightarrow (\mathit{true}) \Rightarrow \ldots$, eventually stabilise, at the latest when they reach *true* (remember that equivalent formulas are considered identical). The same holds for the signs domain. The intervals, octagons and polyhedra domains do however not satisfy the ascending chain condition. Consider for example the following infinite, strictly ascending chain in the interval domain: $(0 \leq \mathtt{i} \wedge \mathtt{i} \leq 0) \Rightarrow (0 \leq \mathtt{i} \wedge \mathtt{i} \leq 1) \Rightarrow (0 \leq \mathtt{i} \wedge \mathtt{i} \leq 2) \Rightarrow \ldots$. For such domains, termination of the analysis has to be enforced (at the cost of precision) by additional tricks, e.g. a technique called "widening" [CC77].

3.3 Predicate Abstraction

Predicate abstraction [GS97] is a variant of abstract interpretation where an arbitrary but *finite* set of predicates P is used to induce the abstract domain A_P. This means that A_P is also finite (again, remember that equivalent formulas are considered identical), and therefore trivially satisfies the ascending chain condition. An abstraction function from the concrete semantic domain to A_P is then defined in the following way:

Definition (Predicate abstraction function). For $P \subseteq \mathit{Formulas}$:

$$\alpha_P : \mathit{Formulas} \to A_P$$

$$\alpha_P(\varphi) = \bigwedge \{p \in P \mid \varphi \Rightarrow p\}$$

Since the abstract domain is a subset of the concrete domain, the concretisation function can simply be chosen as $\gamma = id$, i.e. the identity function.

The transition function $next_P$ is defined in terms of the transition function $next_C$ of the accumulating semantics: $next_P(\varphi) = \alpha_P^n(next_C(\varphi))$. This means that, instead of using a tailor-made transition function operating directly on the abstract domain, transitions are done in the concrete semantic domain, and termination is ensured by explicit abstraction operations in between. This approach is probably less efficient, because calculating α_P can be expensive—in fact, α_P is even uncomputable, but (since P is finite) it can be approximated by calling an automatic first-order logic theorem prover. On the other hand, the approach is very straightforward and works unchanged for arbitrary predicates. Furthermore, its correctness immediately follows from the correctness of the accumulating semantics:

Lemma 3.1. *A predicate abstraction analysis is a correct abstraction of the accumulating semantics, i.e. (as defined in Section 3.2):*

1. *α_P and γ_P are monotonic*
2. *for all $\varphi \in Formulas$: $\varphi \Rightarrow \gamma_P(\alpha_P(\varphi))$*
3. *for all $\varphi \in A_P$: $\varphi \Leftrightarrow \alpha_P(\gamma_P(\varphi))$*
4. *for all $\underline{\varphi} \in Formulas^n$: $\alpha_P^n(next_C(\underline{\varphi})) \Rightarrow^n next_P(\alpha_P^n(\underline{\varphi}))$*

Proof.

1. Since $\gamma_P = id$, we only need to show something for α_P. Let $\varphi_1, \varphi_2 \in Formulas$ such that $\varphi_1 \Rightarrow \varphi_2$. Let furthermore $\alpha_P(\varphi_1) = \bigwedge P_1$ and $\alpha_P(\varphi_2) = \bigwedge P_2$.
   By definition of α_P we know that $P_1 = \{p \in P \mid \varphi_1 \Rightarrow p\}$ and $P_2 = \{p \in P \mid \varphi_2 \Rightarrow p\}$. Since $\varphi_1 \Rightarrow \varphi_2$, this implies $P_2 \subseteq P_1$, and therefore $\bigwedge P_1 \Rightarrow \bigwedge P_2$, which by definition of P_1 and P_2 means $\alpha_P(\varphi_1) \Rightarrow \alpha_P(\varphi_2)$.
2. Let $\alpha_P(\varphi) = \bigwedge P'$. By definition of α_P, we know that for each $p \in P'$: $\varphi \Rightarrow p$. Therefore $\varphi \Rightarrow \bigwedge P'$.
3. Let $\varphi \in A_P$, i.e. $\varphi = \bigwedge P'$ for some $P' \subseteq P$. Let furthermore $\alpha_P(\varphi) = \bigwedge P''$. We only need to show $\alpha_P(\varphi) \Rightarrow \varphi$, as the other direction has already been covered above.
   By definition of α_P, $P'' = \{p \in P \mid \varphi \Rightarrow p\}$. Since $P' \subseteq P$, and since obviously for all $p \in P'$: $\varphi \Rightarrow p$, it holds that $P' \subseteq P''$. This implies $\bigwedge P'' \Rightarrow \bigwedge P'$, which is the same as $\alpha_P(\varphi) \Rightarrow \varphi$.
4. Let $\underline{\varphi} \in Formulas^n$. From (2) we get $\underline{\varphi} \Rightarrow^n \alpha_P^n(\underline{\varphi})$. Since $next_C$ is monotonic (like all transition functions), this implies $next_C(\underline{\varphi}) \Rightarrow^n next_C(\alpha_P^n(\underline{\varphi}))$. Since by (1) α_P is monotonic, too, this in turn implies $\alpha_P^n(next_C(\underline{\varphi})) \Rightarrow^n \alpha_P^n(next_C(\alpha_P^n(\underline{\varphi})))$.
   Independently, the definition of $next_P$ yields $next_P(\alpha_P^n(\underline{\varphi})) = \alpha_P^n(next_C(\alpha_P^n(\underline{\varphi})))$.
   Combined, we get the desired $\alpha_P^n(next_C(\underline{\varphi})) \Rightarrow^n next_P(\alpha_P^n(\underline{\varphi}))$. □

Proposition (3) does not hold when α_P is approximated by calls to an automated theorem prover, because then it is not guaranteed that $\alpha_P(\varphi)$ contains *all* predicates from P which are implied by φ. It is not really necessary for correctness, though. The only part of Lemma 3.1 that we make use of later is proposition (2), which holds in either case.

4 Approach

Although the JavaDL proof process outlined in Chapter 2 and the static analyses described in Chapter 3 are very different at first glance, they are also fundamentally similar in a way: Both can be understood as symbolic execution of programs. Based on this common ground, it is possible to realise an automatic, invariant-inferring static analysis within the JavaDL sequent calculus by defining a relatively small number of additional rules.

This idea is elaborated on in Section 4.1, and illustrated with an extended example in Section 4.2. The necessary new sequent calculus rules are formally defined in Section 4.3.

4.1 Basic Idea

Formulas of the form $\varphi \rightarrow [\mathtt{p}]\psi$ can intuitively be read as: The program point specified by the active statement of `p` must be considered in those states which are allowed by the formula φ. Such formulas (or equivalent sequents) are typical for KeY proofs involving programs; the only feature of the symbolic execution mechanism which produces significantly different formulas is the transformation of assignments into updates.

If we use the rule *assign_outer* instead of *assign_update*, then execution steps roughly work in the following way: The active statement is removed from `p`, and φ is adapted to model the result of executing the active statement in states which satisfy φ. For example, the formula $\mathtt{x} \doteq 0 \rightarrow [\mathtt{y = 1; z = 2;}]\psi$ is first transformed into $\mathtt{x} \doteq 0 \land \mathtt{y} \doteq 1 \rightarrow [\mathtt{z = 2;}]\psi$ and then into $\mathtt{x} \doteq 0 \land \mathtt{y} \doteq 1 \land \mathtt{z} \doteq 2 \rightarrow \psi$. In this process, ψ is usually ignored—it only comes into play after symbolic execution of `p` has completed. We are interested in the intermediate formulas φ: Like invariants, they are associated with specific program points, and in fact, for programs not containing conditionals or loops, they *are* invariants.

The most significant difference between this update-less symbolic execution in KeY and abstract interpretations is related to conditionals and loops: These constructs lead to branches and subsequent confluences in the control flow graph, and at such confluences, abstract interpretations merge the information from the multiple predecessors, whereas KeY continues to treat them separately. The result of this difference is that abstract interpretations infer an *accumulated* formula for each program point—an invariant—whereas KeY infers many different formulas for each single program point, distributed

over many proof branches corresponding to different paths through the control flow graph.

The symbolic execution of KeY can be adapted so it merges control flow paths, too. For example, when executing a conditional statement, we can refrain from splitting the proof by applying *ifthenelse_split*, and instead execute the two control flow branches within the same proof branch. Once the bodies of the then-block and of the else-block have been executed, the leaf of the proof tree will contain a formula like $(\varphi_1 \rightarrow [\mathtt{p}]\psi) \wedge (\varphi_2 \rightarrow [\mathtt{p}]\psi)$, where the remaining program `p` occurs twice. "Merging" now means transforming this into the logically equivalent $(\varphi_1 \vee \varphi_2) \rightarrow [\mathtt{p}]\psi$, in which `p` only occurs once and the subformula on the left describes the combined effects of both control flow paths. An analogous procedure can be applied to loops.

With this change, the symbolic execution of KeY can be understood to be the accumulating semantics from abstract interpretation: It in principle computes the strongest invariants, but does not usually terminate—loops are unwound infinitely many times, without ever reaching a fixed point. The order in which the properties of the individual program points are iteratively recomputed is slightly different than presented in Chapter 3: There, in each iteration all properties were recomputed from top to bottom. The symbolic execution in KeY on the other hand locally iterates within each loop until a local fixed point is found, and only considers the rest of the program following the loop afterwards. However, such ordering differences are not of consequence [CC77].

Termination can only be enforced by using a less precise, more abstract analysis. Such an analysis can be defined on top of the symbolic execution using the predicate abstraction approach: Explicit abstraction operations are performed in between the iterations of symbolic execution. Since the abstract domain is finite, this ensures that a fixed point is reached eventually.

4.2 Example

To illustrate the idea presented in the previous section, we walk through a run of the intended analysis on the array maximum program introduced in Example 1.1 (p. 2). The following JavaDL formula states that after running the program, `max` contains a value larger or equal than the elements of the array `a`:

$$\begin{aligned} \mathtt{a} \neq \mathtt{null} \rightarrow [\{\ & \mathtt{max = 0;} \\ & \mathtt{i = 0;} \\ & \mathtt{while(i < a.length)\ \{} \\ & \quad \mathtt{if(a[i] > max)\ max = a[i];} \\ & \quad \mathtt{i{+}{+};} \\ & \mathtt{\}} \\ \}]\forall x; (0 \leq x \wedge x < \mathtt{a.length} \rightarrow \mathtt{a}[x] \leq \mathtt{max}) & \end{aligned} \tag{4.1}$$

We use this formula as the root of our proof tree. Actually, the nodes of the tree are sequents of the form $\emptyset \vdash \varphi$, but we write them just as formulas φ. In the following we abbreviate the subformula behind the modality by ψ, i.e. $\psi = \forall x; (0 \leq x \land x < \texttt{a.length} \rightarrow \texttt{a[}x\texttt{]} \leq \texttt{max})$.

After executing the first two statements with *assign_outer*, we get:

$$\texttt{a} \neq \texttt{null} \land \texttt{max} \doteq 0 \land \texttt{i} \doteq 0 \rightarrow [\{\ \texttt{while(i < a.length) \{ if(a[i] > max) max = a[i]; i++; \}}\ \}]\psi \tag{4.2}$$

We are now at program point 2 from Figure 1.1, the entry of the loop. This means that the current subformula on the left of the implication arrow, $\texttt{a} \neq \texttt{null} \land \texttt{max} \doteq 0 \land \texttt{i} \doteq 0$, is our first candidate for being a loop invariant. Note that except for $\texttt{a} \neq \texttt{null}$, this is the same property that is also derived for this program point in the first iteration of constant propagation analysis (see Example 3.1, p. 19).

The next step is to unwind the loop for the first time. This actually comprises several intermediate steps: First, we apply *unwind_while*, which creates a conditional statement containing the unwound loop body and the loop itself. This conditional statement is then transformed into a conditional formula by applying *ifthenelse*. Finally, we do not apply *ifthenelse_split* to split the proof, but rather expand the conditional formula to its counterpart using regular junctors. Regardless of these sub-steps, the end result is quite intuitive:

$$\begin{aligned}
&\big(\texttt{a} \neq \texttt{null} \land \texttt{max} \doteq 0 \land \texttt{i} \doteq 0 \land \texttt{i} < \texttt{a.length} \\
&\rightarrow [\{\ \{\ \texttt{if(a[i] > max) max = a[i]; i++; }\}\ \texttt{while(i < a.length) \{ if(a[i] > max) max = a[i]; i++; \}}\ \}]\psi\big) \\
&\land \big(\texttt{a} \neq \texttt{null} \land \texttt{max} \doteq 0 \land \texttt{i} \doteq 0 \land \texttt{i} \geq \texttt{a.length} \rightarrow [\{\}]\psi\big)
\end{aligned} \tag{4.3}$$

We now have two control flow branches within our proof node: If $\texttt{i} < \texttt{a.length}$ holds, then we enter the loop body; otherwise, we immediately exit the loop. The latter case is uninteresting at the moment: We just carry the corresponding subformula around while we execute the loop body in the first conjunct.

Actually, the expression `a.length` is not side-effect free, as it throws a `NullPointerException` if `a` is `null`. Unwinding the loop leads to the creation of an additional proof branch which covers this exceptional case. This case can actually never occur, because we required $\texttt{a} \neq \texttt{null}$ as a precondition. We ignore it in any event and continue exclusively on the branch shown above, where execution proceeds normally.

The next step is to execute the conditional statement contained in the loop body. Again, we do this without branching the proof; and again, we ignore the proof branches which are created along the way to cover the cases that the expression `a[i] > max` throws an exception because either `a` is `null` or `i` is not within the array bounds. We also execute the assignment `max = a[i]` in the body of the conditional statement's then-block, and obtain:

$$
\begin{aligned}
&\big(\texttt{a} \neq \texttt{null} \land max_0 \doteq 0 \land \texttt{i} \doteq 0 \land \texttt{i} < \texttt{a.length} \land \texttt{a[i]} > max_0 \land \texttt{max} \doteq \texttt{a[i]} \\
&\rightarrow [\{\ \{ \\
&\qquad \texttt{i++;} \\
&\qquad \texttt{\}} \\
&\qquad \texttt{while(i < a.length) \{} \\
&\qquad \texttt{if(a[i] > max) max = a[i];} \\
&\qquad \texttt{i++;} \\
&\qquad \texttt{\}} \\
&\quad \texttt{\}}]\psi\big) \\
&\land \big(\texttt{a} \neq \texttt{null} \land \texttt{max} \doteq 0 \land \texttt{i} \doteq 0 \land \texttt{i} < \texttt{a.length} \land \texttt{a[i]} \leq \texttt{max} \\
&\quad \rightarrow [\{\ \{ \\
&\qquad\quad \texttt{i++;} \\
&\qquad\quad \texttt{\}} \\
&\qquad\quad \texttt{while(i < a.length) \{} \\
&\qquad\quad \texttt{if(a[i] > max) max = a[i];} \\
&\qquad\quad \texttt{i++;} \\
&\qquad\quad \texttt{\}} \\
&\qquad \texttt{\}}]\psi\big) \\
&\land \big(\texttt{a} \neq \texttt{null} \land \texttt{max} \doteq 0 \land \texttt{i} \doteq 0 \land \texttt{i} \geq \texttt{a.length} \rightarrow [\{\texttt{\}}]\psi\big)
\end{aligned}
\tag{4.4}
$$

Now there are three cases. The last one is the same as before, namely the situation that the loop has never been entered. The first two correspond to the two control flow paths where the condition `a[i] > max` has been satisfied and where it has not been satisfied, respectively. In the first case, the assignment `max = a[i]` has been executed with *assign_outer*, which has lead to the old version of `max` being renamed to max_0.

Having completed execution of the conditional statement, we have now reached a confluence in the control flow graph, namely program point 5 in Figure 1.1. This manifests itself in the fact that the same remaining program occurs in two different conjuncts. We

"merge" these two cases into one:

$$\begin{aligned}
&\big((\texttt{a} \neq \texttt{null} \land max_0 \doteq 0 \land \texttt{i} \doteq 0 \land \texttt{i} < \texttt{a.length} \land \texttt{a[i]} > max_0 \land \texttt{max} \doteq \texttt{a[i]} \\
&\quad \lor \texttt{a} \neq \texttt{null} \land \texttt{max} \doteq 0 \land \texttt{i} \doteq 0 \land \texttt{i} < \texttt{a.length} \land \texttt{a[i]} \leq \texttt{max}) \\
&\quad \rightarrow [\{\ \{ \\
&\qquad\quad \texttt{i++;} \\
&\qquad \texttt{}\} \\
&\qquad \texttt{while(i < a.length) \{} \\
&\qquad\quad \texttt{if(a[i] > max) max = a[i];} \\
&\qquad\quad \texttt{i++;} \\
&\qquad \texttt{}\} \\
&\quad \texttt{}\}]\psi\big) \\
&\land \big(\texttt{a} \neq \texttt{null} \land \texttt{max} \doteq 0 \land \texttt{i} \doteq 0 \land \texttt{i} \geq \texttt{a.length} \rightarrow [\{\}]\psi\big)
\end{aligned} \tag{4.5}$$

Executing the last remaining statement of the loop body yields:

$$\begin{aligned}
&\big((\texttt{a} \neq \texttt{null} \land max_0 \doteq 0 \land i_0 \doteq 0 \land i_0 < \texttt{a.length} \land \texttt{a}[i_0] > max_0 \land \texttt{max} \doteq \texttt{a}[i_0] \\
&\quad \lor \texttt{a} \neq \texttt{null} \land \texttt{max} \doteq 0 \land i_0 \doteq 0 \land i_0 < \texttt{a.length} \land \texttt{a}[i_0] \leq \texttt{max}) \\
&\quad \land \texttt{i} \doteq i_0 + 1 \\
&\quad \rightarrow [\{\ \texttt{while(i < a.length) \{} \\
&\qquad\quad \texttt{if(a[i] > max) max = a[i];} \\
&\qquad\quad \texttt{i++;} \\
&\qquad \texttt{}\} \\
&\quad \texttt{}\}]\psi\big) \\
&\land \big(\texttt{a} \neq \texttt{null} \land \texttt{max} \doteq 0 \land \texttt{i} \doteq 0 \land \texttt{i} \geq \texttt{a.length} \rightarrow [\{\}]\psi\big)
\end{aligned} \tag{4.6}$$

Now we get back to program point 2 from Figure 1.1. Unlike in the first iteration, we have to consider both control flow predecessors this time. Let φ_1 and φ_2 be defined as follows:

$$\begin{aligned}
\varphi_1 = (&\texttt{a} \neq \texttt{null} \land max_0 \doteq 0 \land i_0 \doteq 0 \land i_0 < \texttt{a.length} \land \texttt{a}[i_0] > max_0 \land \texttt{max} \doteq \texttt{a}[i_0] \\
&\lor \texttt{a} \neq \texttt{null} \land \texttt{max} \doteq 0 \land i_0 \doteq 0 \land i_0 < \texttt{a.length} \land \texttt{a}[i_0] \leq \texttt{max}) \\
&\land \texttt{i} \doteq i_0 + 1
\end{aligned}$$

$$\varphi_2 = \texttt{a} \neq \texttt{null} \land \texttt{max} \doteq 0 \land \texttt{i} \doteq 0$$

That is, φ_1 is the subformula in the first three lines of (4.6), and φ_2 is the subformula on the left of the implication in (4.2), which is also present in the second conjunct of (4.6). We merge the control flow paths into:

$$\begin{aligned}
\varphi_1 \lor \varphi_2 \rightarrow [\{\ &\texttt{while(i < a.length) \{} \\
&\quad \texttt{if(a[i] > max) max = a[i];} \\
&\quad \texttt{i++;} \\
&\texttt{}\} \\
\texttt{}\}]\psi&
\end{aligned} \tag{4.7}$$

This means that $\varphi_1 \vee \varphi_2$ replaces φ_2 as our invariant candidate for the loop entry. The next task is to check whether a fixed point has already been reached, i.e. whether $\varphi_1 \vee \varphi_2 \Leftrightarrow \varphi_2$ holds. Since $\varphi_1 \vee \varphi_2 \Leftarrow \varphi_2$ always holds, we only need to check $\varphi_1 \vee \varphi_2 \Rightarrow \varphi_2$, which in turn can be simplified to $\varphi_1 \Rightarrow \varphi_2$.

In this case, $\varphi_1 \Rightarrow \varphi_2$ does *not* hold, the reason being for example that φ_1 requires `i` to have the value 1, whereas φ_2 requires the value of `i` to be 0. Therefore, we did not yet find a fixed point. In fact, we could go on iterating like this indefinitely; the invariant candidates would get progressively weaker as new disjuncts would be added, but a fixed point would never be reached. In the terminology of Chapter 3, the reason is that that the domain $(\mathit{Formulas}, \Rightarrow, \mathit{false})$ does not satisfy the ascending chain condition. For example, $(\texttt{i} \doteq 0) \Rightarrow (\texttt{i} \doteq 0 \vee \texttt{i} \doteq 1) \Rightarrow (\texttt{i} \doteq 0 \vee \texttt{i} \doteq 1 \vee \texttt{i} \doteq 2) \Rightarrow \ldots$ is an infinite, strictly ascending chain, whose elements describe the values that are possible for `i` after a corresponding number of analysis iterations.[1]

To avoid such infinite chains we now use predicate abstraction: We replace $\varphi_1 \vee \varphi_2$ by $\alpha_P(\varphi_1 \vee \varphi_2)$, i.e. by the conjunction of all predicates in P which are implied by $\varphi_1 \vee \varphi_2$. The result of this abstraction step of course depends on the exact set P of predicates, which has to be supplied to the analysis from the outside. For this example, we assume that P has been chosen as follows:

$$P = \{(\texttt{a} \not\doteq \texttt{null}),\ (0 \leq \texttt{i}),\ \forall x; (0 \leq x \wedge x < \texttt{i} \rightarrow \texttt{a[}x\texttt{]} \leq \texttt{max}),\ (\texttt{i} \doteq 0),\ (\texttt{i} \leq 1)\}$$

That is, we assume that P contains all the components of the loop invariant from Example 1.1 (p. 2), as well as the predicates $\texttt{i} \doteq 0$ and $\texttt{i} \leq 1$, which are not invariants. In practice, there will usually be many more such non-invariant predicates; there can be an arbitrary but finite number. Abstracting with this choice of P produces:

$$\begin{array}{l} \texttt{a} \not\doteq \texttt{null} \wedge 0 \leq \texttt{i} \wedge \texttt{i} \leq 1 \wedge \forall x; (0 \leq x \wedge x < \texttt{i} \rightarrow \texttt{a[}x\texttt{]} \leq \texttt{max}) \\ \rightarrow [\texttt{\{ while(i < a.length) \{} \\ \qquad \texttt{if(a[i] > max) max = a[i];} \\ \qquad \texttt{i++;} \\ \quad \texttt{\}} \\ \texttt{\}}]\psi \end{array} \tag{4.8}$$

The first line of (4.8) is formed by $\alpha_P(\varphi_1 \vee \varphi_2)$. It contains all predicates which are loop invariants. Also, it contains $\texttt{i} \leq 1$, which still holds because we have so far performed just one iteration. It does not contain $\texttt{i} \doteq 0$, since the possible values of `i` already comprise 0 and 1.

[1] If we take into account that the size of integers in Java is limited, then the number of values which `i` can take is actually finite. But it is still too large to be explorable in practice.

After unwinding the loop and executing its body a second time, we get:

$$
\begin{aligned}
&\big((\mathtt{a} \neq \mathtt{null} \land 0 \leq i_1 \land i_1 \leq 1 \land \forall x; (0 \leq x \land x < i_1 \rightarrow \mathtt{a}[x] \leq max_1) \\
&\quad \land i_1 < \mathtt{a.length} \land \mathtt{a}[i_1] > max_1 \land \mathtt{max} \doteq \mathtt{a}[i_1] \\
&\quad \lor \mathtt{a} \neq \mathtt{null} \land 0 \leq i_1 \land \forall x; (0 \leq x \land x < i_1 \rightarrow \mathtt{a}[x] \leq \mathtt{max}) \\
&\quad \land i_1 < \mathtt{a.length} \land \mathtt{a}[i_1] \leq \mathtt{max}) \\
&\land \mathtt{i} \doteq i_1 + 1 \\
&\rightarrow [\{\ \mathtt{while(i < a.length)\ \{} \\
&\qquad \mathtt{if(a[i] > max)\ max = a[i];} \\
&\qquad \mathtt{i{+}{+};} \\
&\quad \mathtt{\}} \\
&\ \mathtt{\}}]\psi\big) \\
&\land \big(\mathtt{a} \neq \mathtt{null} \land 0 \leq \mathtt{i} \land \mathtt{i} \leq 1 \land \forall x; (0 \leq x \land x < \mathtt{i} \rightarrow \mathtt{a}[x] \leq \mathtt{max}) \\
&\quad \land \mathtt{i} \geq \mathtt{a.length} \\
&\quad \rightarrow [\{\mathtt{\}}]\psi\big)
\end{aligned}
\tag{4.9}
$$

Again, we are getting to program point 2 from Figure 1.1. We redefine φ_1 and φ_2 as follows:

$$
\begin{aligned}
\varphi_1 = (&\mathtt{a} \neq \mathtt{null} \land 0 \leq i_1 \land i_1 \leq 1 \land \forall x; (0 \leq x \land x < i_1 \rightarrow \mathtt{a}[x] \leq max_1) \\
&\land i_1 < \mathtt{a.length} \land \mathtt{a}[i_1] > max_1 \land \mathtt{max} \doteq \mathtt{a}[i_1] \\
&\lor \mathtt{a} \neq \mathtt{null} \land 0 \leq i_1 \land \forall x; (0 \leq x \land x < i_1 \rightarrow \mathtt{a}[x] \leq \mathtt{max}) \\
&\land i_1 < \mathtt{a.length} \land \mathtt{a}[i_1] \leq \mathtt{max}) \\
&\land \mathtt{i} \doteq i_1 + 1
\end{aligned}
$$

$$
\varphi_2 = \mathtt{a} \neq \mathtt{null} \land 0 \leq \mathtt{i} \land \mathtt{i} \leq 1 \land \forall x; (0 \leq x \land x < \mathtt{i} \rightarrow \mathtt{a}[x] \leq \mathtt{max})
$$

That is, φ_1 is the subformula in the first five lines of (4.9), and φ_2 is the subformula on the left of the implication in (4.8), which is also present in the second conjunct of (4.9). As before, we merge the control flow paths into:

$$
\begin{aligned}
\varphi_1 \lor \varphi_2 \rightarrow [\{\ &\mathtt{while(i < a.length)\ \{} \\
&\mathtt{if(a[i] > max)\ max = a[i];} \\
&\mathtt{i{+}{+};} \\
&\mathtt{\}} \\
\mathtt{\}}]\psi&
\end{aligned}
\tag{4.10}
$$

However, $\varphi_1 \Rightarrow \varphi_2$ still does not hold, because φ_1 allows 1 and 2 as values of `i`, whereas φ_2 only allows `0` and `1`. Therefore, we have not reached a fixed point yet. The underlying reason is the predicate $\mathtt{i} \leq 1$, which is part of φ_2 but which is not an invariant.

Since we are not done yet, we have to perform another abstraction step, i.e. we again

replace $\varphi_1 \vee \varphi_2$ by $\alpha_P(\varphi_1 \vee \varphi_2)$:

$$\begin{aligned}
&\texttt{a} \neq \texttt{null} \wedge 0 \leq \texttt{i} \wedge \forall x; (0 \leq x \wedge x < \texttt{i} \rightarrow \texttt{a}[x] \leq \texttt{max}) \\
&\rightarrow [\{\ \texttt{while(i < a.length) \{} \\
&\qquad \texttt{if(a[i] > max) max = a[i];} \\
&\qquad \texttt{i++;} \\
&\quad \texttt{\}} \\
&\texttt{\}}]\psi
\end{aligned} \tag{4.11}$$

This time only predicates which really are invariants remain. However, we cannot know this mechanically yet; we have to perform another iteration to see that nothing changes any more. So we unwind the loop and execute its body a third time to get:

$$\begin{aligned}
&\big((\texttt{a} \neq \texttt{null} \wedge 0 \leq i_2 \wedge \forall x; (0 \leq x \wedge x < i_2 \rightarrow \texttt{a}[x] \leq max_2) \\
&\quad \wedge i_2 < \texttt{a.length} \wedge \texttt{a}[i_2] > max_2 \wedge \texttt{max} \doteq \texttt{a}[i_2] \\
&\quad \vee \texttt{a} \neq \texttt{null} \wedge 0 \leq i_2 \wedge \forall x; (0 \leq x \wedge x < i_2 \rightarrow \texttt{a}[x] \leq \texttt{max}) \\
&\qquad \wedge i_2 < \texttt{a.length} \wedge \texttt{a}[i_2] \leq \texttt{max}) \\
&\wedge \texttt{i} \doteq i_2 + 1 \\
&\rightarrow [\{\ \texttt{while(i < a.length) \{} \\
&\qquad \texttt{if(a[i] > max) max = a[i];} \\
&\qquad \texttt{i++;} \\
&\quad \texttt{\}} \\
&\texttt{\}}]\psi\big) \\
&\wedge \big(\texttt{a} \neq \texttt{null} \wedge 0 \leq \texttt{i} \wedge \forall x; (0 \leq x \wedge x < \texttt{i} \rightarrow \texttt{a}[x] \leq \texttt{max}) \wedge \texttt{i} \geq \texttt{a.length} \\
&\quad \rightarrow [\{\}]\psi\big)
\end{aligned} \tag{4.12}$$

We redefine φ_1 and φ_2 again:

$$\begin{aligned}
\varphi_1 = (&\texttt{a} \neq \texttt{null} \wedge 0 \leq i_2 \wedge \forall x; (0 \leq x \wedge x < i_2 \rightarrow \texttt{a}[x] \leq max_2) \\
&\wedge i_2 < \texttt{a.length} \wedge \texttt{a}[i_2] > max_2 \wedge \texttt{max} \doteq \texttt{a}[i_2] \\
&\vee \texttt{a} \neq \texttt{null} \wedge 0 \leq i_2 \wedge \forall x; (0 \leq x \wedge x < i_2 \rightarrow \texttt{a}[x] \leq \texttt{max}) \\
&\wedge i_2 < \texttt{a.length} \wedge \texttt{a}[i_2] \leq \texttt{max}) \\
&\wedge \texttt{i} \doteq i_2 + 1
\end{aligned}$$

$$\varphi_2 = \texttt{a} \neq \texttt{null} \wedge 0 \leq \texttt{i} \wedge \forall x; (0 \leq x \wedge x < \texttt{i} \rightarrow \texttt{a}[x] \leq \texttt{max})$$

That is, φ_1 is the subformula in the first five lines of (4.12), and φ_2 is the subformula on the left of the implication in (4.11), which is also present in the second conjunct of (4.12). We once again merge the control flow paths into:

$$\begin{aligned}
\varphi_1 \vee \varphi_2 \rightarrow [&\{\ \texttt{while(i < a.length) \{} \\
&\qquad \texttt{if(a[i] > max) max = a[i];} \\
&\qquad \texttt{i++;} \\
&\quad \texttt{\}} \\
&\texttt{\}}]\psi
\end{aligned} \tag{4.13}$$

Now, $\varphi_1 \Rightarrow \varphi_2$ holds, and thus $\varphi_1 \vee \varphi_2 \Leftrightarrow \varphi_2$. This means that our candidate loop invariant φ_2 is a fixed point and thus a real loop invariant. We can stop iterating the loop and assume that φ_2 holds afterwards:

$$\varphi_2 \wedge \texttt{i} \geq \texttt{a.length} \rightarrow [\{\}]\psi \tag{4.14}$$

4.3 Rules

The JavaDL sequent calculus rules necessary to implement the invariant inference characterised in the previous sections are defined below. In analogy to Section 2.4, they are sorted by whether they (primarily) deal with assignments, conditionals, or loops. The rule used for predicate abstraction, which does not have a counterpart in Section 2.4, is presented last.

4.3.1 Assignments

An essential prerequisite for understanding the symbolic execution of KeY as an abstract interpretation was the use of *assign_outer* instead of *assign_update* for treating assignments. However, in the actual KeY system, replacing the assignment rule is rather impractical, as there are actually many such rules: For example, assignments to fields and to arrays are handled separately. Moreover, updates can emerge in other ways as well, not only from the assignment rules. Therefore, we opted not to avoid the introduction of updates, but instead deal with them afterwards in a way which mimics the effect of using *assign_outer*.

The rule below transforms a formula of the form $\varphi \rightarrow \{u\}\psi$ into a formula $\varphi' \rightarrow \psi$, where φ' is the strongest postcondition of φ under u. The update u again occurs in φ' (it is not eliminated, only "shifted" to the left), and φ' even contains new, additional updates. The important result is however that the update is removed from ψ.

Definition (Rule *shift_update*).

$$\frac{\emptyset \;\vdash\; (\{u'\}\varphi \wedge \tau \rightarrow \psi) \wedge \varepsilon}{\emptyset \;\vdash\; (\varphi \rightarrow \{u\}\psi) \wedge \varepsilon}\ \textit{shift\_update}$$

where:

- for each $f \in \mathit{targets}(u)$: f' is a fresh rigid function symbol with $\mathit{arity}(f') = \mathit{arity}(f)$
- $u' = \mathit{par}_{f \in \mathit{targets}(u)} \big(\mathit{for}\ x_1; \ldots; \mathit{for}\ x_n; f(x_1, \ldots, x_n) := f'(x_1, \ldots, x_n)\big)$
  (where *par* stands for parallel update composition in an arbitrary order, and $\mathit{arity}(f) = n$)
- $\tau = \bigwedge_{f \in \mathit{targets}(u)} \forall x_1; \ldots; \forall x_n; f(x_1, \ldots, x_n) \doteq \{u'\}\{u\}f(x_1, \ldots x_n)$

The structure of this rule is unusual in a number of ways. Firstly, it operates not on a sequent $\varphi \vdash \{u\}\psi$, but on an equivalent formula, and it carries around a subformula ε which is not affected by the rule. The same is true for the other rules defined below. The underlying reason is that we do not split the proof at conditionals or loops, and instead carry around several conjuncts in our succedent, like in $\vdash (\varphi_1 \to \psi_1) \wedge (\varphi_2 \to \psi_2)$. This is somewhat at odds with the basic principle of the sequent calculus, which favours having only one implication at a time. Our sequents therefore contain everything within a single formula, and the rules are formulated accordingly.[2] Naturally, the conjunct to which a rule is applied does not need to be the first one, although the rules are written here in this way.

A second unusual property of *shift_update* is that it explicitly forbids having other formulas (Γ, Δ) in the sequent. This restriction obviously does not hurt in our context. It is only necessary for this rule, not for the others defined in this section.

Example 4.1. As a simple example for an appliance of *shift_update*, consider the following proof tree:

$$\dfrac{\dfrac{\vdash\ i' \doteq 27 \wedge \mathtt{i} \doteq i' + 1 \to \psi}{\vdash\ \{\mathtt{i} := i'\}\mathtt{i} \doteq 27 \wedge \mathtt{i} \doteq \{\mathtt{i} := i'\}\{\mathtt{i} := \mathtt{i} + 1\}\mathtt{i} \to \psi}\ (\mathit{simpl\_update})}{\vdash\ \mathtt{i} \doteq 27 \to \{\mathtt{i} := \mathtt{i} + 1\}\psi}\ (\mathit{shift\_update})$$

The update u' from the definition, which is $\mathtt{i} := i'$ in this case, encodes the same renaming which is performed by *assign_outer*. If φ does not contain modalities, the updates resulting from an application of *shift_update* can always be disposed by update simplification as shown here, yielding a result equivalent to that of *assign_outer* (cf. Example 2.3, p. 14).

Example 4.2. A more complicated situation involving aliasing is shown below:

$$\dfrac{\dfrac{\begin{array}{c}\vdash\ f'(\mathtt{this}) \doteq 27 \\ \wedge \forall x; x.\mathtt{f} \doteq \mathit{if}(f'(\mathtt{o}) \doteq 358 \wedge x \doteq \mathtt{o})\mathit{then}(42)\mathit{else}(f'(x)) \\ \to \psi\end{array}}{\vdash\ \{\mathit{for}\ x; x.\mathtt{f} := f'(x)\}\mathtt{this.f} \doteq 27}\ (\mathit{simpl\_update})}{\begin{array}{c}\wedge \forall x; x.\mathtt{f} \doteq \{\mathit{for}\ x; x.\mathtt{f} := f'(x)\}\{\mathit{if}\ \mathtt{o.f} = 358; \mathtt{o.f} := 42\}x.\mathtt{f} \\ \to \psi\end{array}}\ (\mathit{shift\_update})$$
$$\vdash\ \mathtt{this.f} \doteq 27 \to \{\mathit{if}\ \mathtt{o.f} = 358; \mathtt{o.f} := 42\}\psi$$

As this example shows, *shift_update* with subsequent update simplification creates just those case distinctions for possible aliasing that *assign_update* is able to avoid (or at least delay). This is inevitable in our context.

[2] Fortunately, the actual symbolic execution rules in the KeY system support execution within a complex formula with several conjuncts, too, unlike the simplified versions presented in Section 2.4.

Lemma 4.1 (Soundness of *shift_update*).

$$\models \ (\{u'\}\varphi \wedge \tau \to \psi) \wedge \varepsilon \tag{4.15}$$

implies

$$\models \ (\varphi \to \{u\}\psi) \wedge \varepsilon \tag{4.16}$$

Proof. Let (4.15) hold. Furthermore, let $s \in \mathit{States}$, and β be a variable assignment. Our goal is to show $(s,\beta) \models (\varphi \to \{u\}\psi) \wedge \varepsilon$. From (4.15) we immediately get $(s,\beta) \models \varepsilon$. If $(s,\beta) \not\models \varphi$, we are done. Let therefore $(s,\beta) \models \varphi$. The remaining task is to show $(s,\beta) \models \{u\}\psi$.
Let $U = \mathit{val}_{s,\beta}(u)$, and let $s' \in \mathit{States}$ be defined as follows:

$$s'(op) = \begin{cases} s(f) & \text{if } op = f' \text{ for some } f \in \mathit{targets}(u) \\ U(s)(op) & \text{otherwise} \end{cases}$$

That is, s' is identical to $U(s)$ except that the fresh function symbols f' are interpreted like the corresponding f are interpreted in s.
Let $U' = \mathit{val}_{s',\beta}(u')$, and $s'' = U'(s')$. By Lemma 2.1, s'' satisfies

$$s''(op) = \begin{cases} s'(f') & \text{if } op = f \in \mathit{targets}(u) \\ s'(op) & \text{otherwise} \end{cases}$$

By definition of s', this is the same as

$$s''(op) = \begin{cases} s(op) & \text{if } op \in \mathit{targets}(u) \\ s(f) & \text{if } op = f' \text{ for some } f \in \mathit{targets}(u) \\ U(s)(op) & \text{otherwise} \end{cases}$$

With the help of Lemma 2.2 we can simplify this to

$$s''(op) = \begin{cases} s(f) & \text{if } op = f' \text{ for some } f \in \mathit{targets}(u) \\ s(op) & \text{otherwise} \end{cases} \tag{4.17}$$

That is, s'' is identical to s except for the interpretation of the fresh function symbols. Since these symbols do not occur in φ, and since $(s,\beta) \models \varphi$, it follows by Lemma 2.3 that $(s'',\beta) \models \varphi$, and consequently (by choice of s'')

$$(s',\beta) \models \{u'\}\varphi \tag{4.18}$$

Let $f \in \mathit{targets}(u)$. (4.17) tells us that $s''(f) = s(f)$. Therefore $U(s'')(f) = U(s)(f)$. Independently, the definition of s' yields $s'(f) = U(s)(f)$. Combined, we have $s'(f) = U(s'')(f)$, which by definition of s'' is the same as $s'(f) = U(U'(s'))(f)$. Since this holds for all $f \in \mathit{targets}(u)$ (and since by Lemma 2.4 $U = \mathit{val}_{s'',\beta}(u)$), this implies

$$(s',\beta) \models \tau \tag{4.19}$$

Together, (4.15), (4.18) and (4.19) yield $(s',\beta) \models \psi$. As the fresh function symbols do not occur in ψ, this implies by Lemma 2.3 that $(U(s),\beta) \models \psi$, which in turn implies the desired $(s,\beta) \models \{u\}\psi$. □

4.3.2 Conditionals

Conditional statements can be turned into conditional formulas by applying *ifthenelse* as defined in Section 2.4. We do not want to split the proof by a subsequent application of *ifthenelse_split*, though. Instead, the following rule can be used to expand the conditional formula into a regular one:

Definition (Rule *ifthenelse_expand*).

$$\frac{\Gamma \vdash (\varphi \to \psi_1) \land (\neg\varphi \to \psi_2) \land \varepsilon,\ \Delta}{\Gamma \vdash \mathit{if}(\varphi)\mathit{then}(\psi_1)\mathit{else}(\psi_2) \land \varepsilon,\ \Delta}\ \mathit{ifthenelse\_expand}$$

The soundness of this rule immediately follows from the definition of the semantics of conditional formulas. It can also be applied if the conditional formula is nested more deeply, as in $\vdash \big(\varphi_1 \to \mathit{if}(\varphi_2)\mathit{then}([\mathtt{p}]\psi_1)\mathit{else}([\mathtt{p}]\psi_2)\big) \land \varepsilon$, which is a typical situation after executing a conditional statement. Applying it is the first step towards turning the formula again into a conjunction of subformulas of the form $\varphi \to [\mathtt{p}]\psi$. More normalisation steps are necessary, which can be done using the following two rules:

Definition (Rule *distribute_implication*).

$$\frac{\Gamma \vdash (\varphi \to \psi_1) \land (\varphi \to \psi_2) \land \varepsilon,\ \Delta}{\Gamma \vdash \big(\varphi \to (\psi_1 \land \psi_2)\big) \land \varepsilon,\ \Delta}\ \mathit{distribute\_implication}$$

Definition (Rule *flatten_implication*).

$$\frac{\Gamma \vdash (\varphi_1 \land \varphi_2 \to \psi) \land \varepsilon,\ \Delta}{\Gamma \vdash \big(\varphi_1 \to (\varphi_2 \to \psi)\big) \land \varepsilon,\ \Delta}\ \mathit{flatten\_implication}$$

Their soundness follows from simple propositional logic equivalences. The same is true for the following rule, which can be applied once both control flow branches of the conditional statement have been symbolically executed:

Definition (Rule *merge*).

$$\frac{\Gamma \vdash \big((\varphi_1 \lor \varphi_2) \to \psi\big) \land \varepsilon,\ \Delta}{\Gamma \vdash (\varphi_1 \to \psi) \land (\varphi_2 \to \psi) \land \varepsilon,\ \Delta}\ \mathit{merge}$$

Afterwards, ψ only occurs once, and symbolic execution of the program contained in it can continue without differentiating between the two cases.

Related to conditional statements are assignments to boolean variables, because such assignments are introduced by the symbolic execution if the condition `e` of a conditional statement `if(e) p else q` is a complex expression: Before the rule *ifthenelse* can be applied, the program is first transformed into `boolean b = e; if(b) p else q`.

This is interesting to us here because the calculus handles assignments to boolean variables differently from other assignments: As noted in Section 2.1, complex boolean expressions do not have corresponding JavaDL terms, because the boolean-valued operators of Java (like `<`, `==`, `&&`) are modeled in JavaDL as predicate symbols or junctors, not as function symbols. Therefore, assignments involving such expressions cannot be turned into updates like all other assignments. The calculus instead transforms them into conditional formulas; for example, executing the assignment in the formula $[\texttt{lhs = e}_1 \texttt{ < e}_2\texttt{;}]\psi$ yields $\mathit{if}(\texttt{e}_1 < \texttt{e}_2)\mathit{then}([\texttt{lhs = true;}]\psi)\mathit{else}([\texttt{lhs = false;}]\psi)$.

This treatment of assignments to boolean variables would normally lead to splitting the proof *twice* for each conditional statement involving a complex condition: First when the temporary boolean variable `b` is assigned, and then again when the conditional statement itself is executed. The calculus features special rules to avoid this second split: In situations like `b = true; if(b) p else q`, one branch of the conditional statement can simply be thrown away.

Nevertheless, this approach does not fit well with our invariant inference, as it deviates from the idea of symbolic execution and—at least temporarily—creates additional case distinctions. We instead use rules like the following for handling boolean assignments:

Definition (Rule *lt_nosplit*).

$$\frac{\Gamma \;\vdash\; \big(\varphi \wedge (c \doteq \texttt{true} \leftrightarrow \texttt{e}_1 < \texttt{e}_2) \rightarrow \{\texttt{lhs} := c\}[\pi\,\omega]\psi\big) \wedge \varepsilon,\; \Delta}{\Gamma \;\vdash\; \big(\varphi \rightarrow [\pi\ \texttt{lhs = e}_1\ \texttt{<}\ \texttt{e}_2\texttt{;}\ \omega]\psi\big) \wedge \varepsilon,\; \Delta}\ \mathit{lt\_nosplit}$$

where `lhs`, $\texttt{e}_1$ and $\texttt{e}_2$ must not have side effects, and where c is a fresh rigid constant symbol.

The soundness of this rule should be obvious. Analogous rules are necessary for all other boolean-valued operators of Java.

4.3.3 Loops

The following rule can be used to "merge" the control flow paths converging at a loop entry, similar to what the rule *merge* does for conditional statements:

Definition (Rule *merge_while*).

$$\frac{\Gamma \;\vdash\; \big((\varphi_1 \vee \varphi_2) \rightarrow [\pi\ \texttt{while(e) p}\ \omega]\psi\big) \wedge \varepsilon,\; \Delta}{\Gamma \vdash (\varphi_1 \rightarrow [\pi\ \texttt{while(e) p}\ \omega]\psi) \wedge (\varphi_2 \wedge \neg\texttt{e} \doteq \texttt{true} \rightarrow [\pi\,\omega]\psi) \wedge \varepsilon, \Delta}\ \mathit{merge\_while}$$

where `e` must not have side effects.

The first two conjuncts of the conclusion usually both stem from unwinding the loop: The first conjunct describes the case that the loop condition was satisfied, so the loop body has been executed and now we are about to enter the loop again. The second conjunct stands for the case that the condition did not hold; φ_2 is the previous loop invariant candidate.

Lemma 4.2 (Soundness of *merge_while*).

$$\models\ \Gamma \vdash \big((\varphi_1 \vee \varphi_2) \to [\pi\ \texttt{while(e) p}\ \omega]\psi\big) \wedge \varepsilon, \Delta \tag{4.20}$$

implies

$$\models\ \Gamma \vdash (\varphi_1 \to [\pi\ \texttt{while(e) p}\ \omega]\psi) \wedge (\varphi_2 \wedge \neg \texttt{e} \doteq \texttt{true} \to [\pi\ \omega]\psi) \wedge \varepsilon, \Delta \tag{4.21}$$

Proof. Let (4.20) hold, $s \in \mathit{States}$, and β be a variable assignment. Our goal is to show that the conclusion is valid in (s, β). If $(s, \beta) \not\models \Gamma$ or $(s, \beta) \models \Delta$, we are done. Let therefore $(s, \beta) \models \Gamma$, $(s, \beta) \not\models \Delta$. (4.20) now yields

$$(s, \beta) \models \varphi_1 \to [\pi\ \texttt{while(e) p}\ \omega]\psi \tag{4.22}$$

$$(s, \beta) \models \varphi_2 \to [\pi\ \texttt{while(e) p}\ \omega]\psi \tag{4.23}$$

$$(s, \beta) \models \varepsilon \tag{4.24}$$

From (4.23) and the semantics of Java we get $(s, \beta) \models \varphi_2 \wedge \neg \texttt{e} \doteq \texttt{true} \to [\pi\ \omega]\psi$. This, (4.22) and (4.24) imply the desired $(s, \beta) \models (\varphi_1 \to [\pi\ \texttt{while(e) p}\ \omega]\psi) \wedge (\varphi_2 \wedge \neg \texttt{e} \doteq \texttt{true} \to [\pi\ \omega]\psi) \wedge \varepsilon$. □

The rule *merge_while* determines the new, weaker invariant candidate for the loop: φ_2 is replaced by $\varphi_1 \vee \varphi_2$. Afterwards, the loop can in principle be unwound again. Still missing is a way to stop these iterations eventually. This should be done once the loop invariant candidate is a fixed point, i.e. once $\varphi_1 \vee \varphi_2 \Leftrightarrow \varphi_2$ holds. As we already observed in Section 4.2, $\varphi_1 \vee \varphi_2 \Leftrightarrow \varphi_2$ is equivalent to $\varphi_1 \Rightarrow \varphi_2$, which justifies the following rule:

Definition (Rule *end_while*).

$$\frac{\Gamma\ \vdash\ (\varphi_2 \wedge \neg \texttt{e} \doteq \texttt{true} \to [\pi\ \omega]\psi) \wedge \varepsilon,\ \Delta}{\Gamma\ \vdash\ \big((\varphi_1 \vee \varphi_2) \to [\pi\ \texttt{while(e) p}\ \omega]\psi\big) \wedge \varepsilon,\ \Delta}\ \mathit{end\_while}$$

where `e` must not have side effects, and where $\varphi_1 \Rightarrow \varphi_2$ must hold.

This rule does what we need, but unfortunately it is not sound: For that, it would need to ensure that $\varphi_2 \wedge \texttt{e} \doteq \texttt{true} \Rightarrow [\texttt{p}]\varphi_2$ holds, i.e. that φ_2 is actually an invariant. It is plausible that this is usually the case because of the ancestors which the conclusion usually has in the proof tree: Starting with $\varphi_2 \to [\pi\ \texttt{while(e) p}\ \omega]\psi$, the loop has been unwound to $\varphi_2 \wedge \texttt{e} \doteq \texttt{true} \to [\pi\ \texttt{p while(e) p}\ \omega]\psi$ (plus the exit case). The loop body `p` has then been symbolically executed, yielding $\varphi_1 \to [\pi\ \texttt{while(e) p}\ \omega]\psi$. Then, *merge_while* has been applied. So, the symbolic execution of `p` has transformed

$\varphi_2 \wedge \texttt{e} \doteq \texttt{true}$ into φ_1, and we know that $\varphi_1 \Rightarrow \varphi_2$. Together, this suggests that $\varphi_2 \wedge \texttt{e} \doteq \texttt{true} \Rightarrow [\texttt{p}]\varphi_2$ is true.

Formally requiring that the proof history is as sketched above would however be quite complicated: It would mean formalising the requirement that all rules which have been applied in between must follow the symbolic execution principle, i.e. modelling the effects of concrete executions symbolically. There are other rules in the calculus, like the cut-rule (see [BHS07]), whose application would have to be forbidden. Furthermore, it is possible to define program-shortening rules which still do not perform true symbolic execution, as the following example proof tree shows:

$$\frac{\vdash\ \texttt{i} \doteq 27 \wedge \texttt{j} \doteq 368 \rightarrow \texttt{i} \doteq 27}{\vdash\ \texttt{i} \doteq 27 \rightarrow [\texttt{j = 0;}]\texttt{i} \doteq 27}\ (\textit{irrelevant\_assign\_outer})$$

As the assignment does not influence the validity of the sequent at all, the hypothetical but sound rule *irrelevant_assign_outer* can "execute" it in a way which does not model its real effects. Another example would be rules which perform *backward* symbolic execution. Formally distinguishing such rules from true symbolic execution rules is quite intricate. We therefore decided against ensuring the soundness of *end_while*. The consequence is that the "proofs" created using our invariant inference rules do not carry the same amount of confidence as regular KeY proofs, or in other words, that the inferred invariants are not absolutely guaranteed to really be invariants. But this can be tolerated: A regular proof without unsound rules can be performed afterwards, using the inferred loop invariants to increase the degree of automation.

4.3.4 Abstraction

Given a set $P \subseteq \textit{Formulas}$ of "predicates", the following rule performs an abstraction step:

Definition (Rule *abstract*).

$$\frac{\Gamma\ \vdash\ (\varphi' \rightarrow \psi) \wedge \varepsilon,\ \Delta}{\Gamma\ \vdash\ (\varphi \rightarrow \psi) \wedge \varepsilon,\ \Delta}\ \textit{abstract}$$

where $\varphi' = \alpha_P(\varphi)$ with α_P being the predicate abstraction function defined in Section 3.3.

Lemma 4.3 (Soundness of *abstract*).

$$\models\ \Gamma \vdash (\varphi' \rightarrow \psi) \wedge \varepsilon, \Delta \tag{4.25}$$

$$\textit{implies}$$

$$\models\ \Gamma \vdash (\varphi \rightarrow \psi) \wedge \varepsilon, \Delta \tag{4.26}$$

Proof. Let (4.25) hold, $s \in \mathit{States}$, and β be a variable assignment. Our goal is to show that the conclusion is valid in (s, β). If $(s, \beta) \not\models \Gamma$ or $(s, \beta) \models \Delta$, we are done. Let therefore $(s, \beta) \models \Gamma$, $(s, \beta) \not\models \Delta$. (4.25) now yields $(s, \beta) \models \varphi' \to \psi$ and $(s, \beta) \models \varepsilon$.
If $(s, \beta) \not\models \varphi$, we are done, so assume $(s, \beta) \models \varphi$. From Lemma 3.1 (2) we know that $\varphi \Rightarrow \varphi'$, so we get $(s, \beta) \models \varphi'$. Combined with $(s, \beta) \models \varphi' \to \psi$, this implies $(s, \beta) \models \psi$. Altogether, we have $(s, \beta) \models (\varphi \to \psi) \wedge \varepsilon$. □

5 Implementation

The invariant inference approach introduced in Chapter 4 has been implemented as a part of the KeY system. This implementation comprises three major elements: Firstly, implementations of the rules themselves (Section 5.1); secondly, a heuristic which generates predicates for the abstraction rule (Section 5.2); and finally, a proof search strategy for applying the rules automatically (Section 5.3). An overview of the user's view of the implementation is provided in Section 5.4.

5.1 Rules

In the KeY system, sequent calculus rules are normally implemented as *taclets* [BHS07, BGH+04]. A taclet is a more precise and machine-readable formulation of a rule. As an example, the following taclet implements *and_right*:

```
and_right {
  \find(==> phi1 & phi2)
  \replacewith(==> phi1);
  \replacewith(==> phi2)
  \heuristics(split, beta)
};
```

The first three lines describe the logical content of the rule. Just like φ_1 and φ_2 in the original formulation in Section 2.4, `phi1` and `phi2` are placeholders for arbitrary formulas. Such placeholders, for formulas or other syntactic elements like terms or statements, are called *schema variables*. The concrete KeY syntax differs slightly from the one used so far; for example, the junctor $\wedge$ is rendered as `&` and the sequent arrow $\vdash$ as `==>`. Note that the other formulas of the sequent (Γ, Δ) do not explicitly occur in the taclet.

The last line declares that the taclet belongs to the *rule sets* "`split`" and "`beta`". Rule sets are used to guide the proof search strategies; for example, `split` contains rules which split the proof. We introduce a new rule set `invinference` which comprises all new taclets defined in this section.

An example for a symbolic execution taclet is the following implementation of *ifthenelse*:

```
ifthenelse {
  \find(\[{..  if(#e) #p else #q ...\}]psi)
  \replacewith(\if(#e = TRUE) \then(\[{..  #p ...}\]psi)
                              \else(\[{..  #q ...}\]psi)
  \heuristics(split_if)
};
```

Here, `#e` is a schema variable standing for a side-effect free Java expression, and `#p` and `#q` are schema variables standing for Java statements. The program prefix π and the postfix ω are represented by `..` and `...`, which can also be seen as special schema variables.

Unlike in the taclet `and_right`, the `\find` part of `ifthenelse` does not contain a sequent arrow. This indicates that the taclet may be applied to occurrences of the schematic formula at arbitrary, possibly nested, positions in the sequent. Such taclets are called *rewrite taclets*. This formulation as a rewrite taclet makes `ifthenelse` more powerful than the rule given in Section 2.4, which can only be applied to top-level elements of the succedent.

Not all meaningful rules are expressible with taclets alone. A way to bypass such limitations is the use of *meta operators* in taclets, which can execute arbitrary Java code as part of a taclet application. For example, the taclet for the invariant rule *while_invariant* uses a meta operator to create the anonymising update v. Even though they can execute arbitrary code, taclets with meta operators still operate locally on *parts* of the sequent. Access to the sequent as a whole is only possible with *built-in rules*, i.e. rules which are not encoded as taclets but completely built into the KeY system itself in the form of Java code. An example for a rule which is implemented as a built-in rule is update simplification. Hard-coding custom functionality through meta operators or even built-in rules is non-systematic and in general not preferable, but cannot always be avoided.

The following subsections provide an overview of the implementations of the rules defined in Section 4.3.

5.1.1 Assignments

Since *shift_update* requires that the sequent to which it is applied contains only one formula, it cannot be expressed as a taclet. Instead, it is implemented as a built-in rule. This built-in rule is a straightforward rendering of the rule as presented in Section 4.3. It applies the "update shifting" transformation to all conjuncts of the formula at once, and automatically performs update simplification on its result, so that a subsequent application of *simpl_update* is unnecessary.

5.1.2 Conditionals

The rules for conditional statements are all rather simple and can be formulated as taclets. These taclets are shown in Figure 5.1. In particular, the problem of making the rules applicable to nested elements of a conjunction (expressed by the ε in Section 4.3) is in these cases solved elegantly by the concept of rewrite taclets.

Schema variables declared as `\program LeftHandSide` or `\program SimpleExpression` only match expressions which are guaranteed to be free from side effects. The KeY system automatically instantiates schema variables declared as `\skolemTerm` with a fresh constant symbol when the taclet is applied.

5.1.3 Loops

The two loop rules *merge_while* and *end_while* are implemented together in a single taclet `merge_end_while`, which is shown in Figure 5.2. It uses a meta operator `#MergeWhile` to calculate its result, which is either that of applying *merge_while* alone or that of applying *merge_while* and subsequently *end_while*, depending on whether `phi1` implies `phi2` or not.

Whether this implication holds is determined by asking an automated first-order logic theorem prover to prove it in the background. Of course, `phi1` and `phi2` can in general be dynamic logic formulas involving modalities and updates, but usually this is not the case. Even then, the problem is undecidable, so the called theorem prover may fail to prove `phi1` $\Rightarrow$ `phi2` even if it holds. This can lead to non-termination of the invariant inference process: It is possible that a fixed point has been reached but that this cannot be detected. This risk is however mitigated at least to some extent by the fact that `phi2` normally stems from applying *abstract*: Since *abstract* is computed using the same limited theorem prover, predicates which overburden it usually do not occur in `phi2` in the first place.

The KeY system comes with support for calling various first-order theorem provers, including *Simplify* [DNS03] as well as several tools which support input in the *SMT-LIB* format [RT06], the best of which reportedly is *Yices* [DdM]. By clicking on a button, the user can pass a sequent of a KeY proof to such a tool, which then tries to prove its validity. Which tool is used can be chosen in the options dialog. The `#MergeWhile` meta operator uses this pre-existing functionality for its background theorem prover calls.

Like the taclets for the conditional rules, `merge_end_while` is a rewrite taclet. This is necessary because it, too, needs to be applicable to nested elements of a conjunction in the succedent. However, in this case, this introduces unsoundness: The premiss and the conclusion of *merge_while* are not equivalent, so e.g. applications to a formula in the antecedent would have to be forbidden. The problem is that "applicable to any element of a conjunction in the succedent, but nowhere else" is not expressible in the taclet language; only a built-in rule could enforce it. But since *end_while* is not strictly sound

```
\schemaVariables {
  \formula phi, phi1, phi2, psi, psi1, psi2;
  \program LeftHandSide #lhs;
  \program SimpleExpression #e1, #e2;
  \skolemTerm boolean c;
}
\rules {
  ifthenelse_expand {
    \find(\if(phi)\then(psi1)\else(psi2))
    \replacewith((phi -> psi1) & (!phi -> psi2))
    \heuristics(invinference)
  };
  distribute_implication {
    \find(phi -> (psi1 & psi2))
    \replacewith((phi -> psi1) & (phi -> psi2))
    \heuristics(invinference)
  };
  flatten_implication {
    \find(phi1 -> (phi2 -> psi))
    \replacewith(phi1 & phi2 -> psi)
    \heuristics(invinference)
  };
  merge {
    \find((phi1 -> psi) & (phi2 -> psi))
    \replacewith((phi1 | phi2) -> psi)
    \heuristics(invinference)
  };
  lt_nosplit {
    \find(\[{..  #lhs = #e1 < #e2; ...}\]psi)
    \replacewith((c=TRUE<->lt(#e1,#e2)) -> {#lhs:=c}\[{.. ...}\]psi)
    \heuristics(invinference)
  };
}
```

Figure 5.1: Taclets for the conditional rules.

```
\schemaVariables {
  \formula phi, phi1, phi2, psi, predicates;
  \program SimpleExpression #e;
  \program Statement #p;
}
\rules {
  merge_end_while {
    \find((phi1 -> \[{..  while(#e) #p ...}\]psi)
          & (phi2 & !(#e = TRUE) -> \[{..  ...}\]psi))
    \replacewith(#MergeWhile(phi1,
                             phi2,
                             \[{..  while(#e) #p ...}\]psi))
    \heuristics(invinference)
  };
  abstract {
    \find(phi -> \[{..  while(#e) #p ...}\]psi)
    \replacewith(#Abstract(phi, predicates)
                 -> \[{..  while(#e) #p ...}\]psi)
    \heuristics(invinference)
  };
}
```

Figure 5.2: Taclets for the loop and abstraction rules.

anyway and no unsound applications happen in practice, this issue does not matter much.

5.1.4 Abstraction

The taclet implementing *abstract* is shown in Figure 5.2. Like `merge_end_while`, its formulation as a rewrite taclet is not strictly sound. Since we only want to apply the taclet at loop entries, its applicability is restricted to formulas with a loop as active statement.

The result of `abstract` is computed by a meta operator `#Abstract`, whose arguments are the formula `phi` and another formula `predicates`. Since the latter does not occur in the `\find` part, it can be instantiated freely when applying the taclet. It represents

the set P of predicates: Since schema variables which match sets of formulas are not supported by the KeY system, P is passed to `#Abstract` as a single formula which contains the predicates as conjuncts.

In order to calculate the abstraction function, `#Abstract` performs calls to an external automated theorem prover just like `#MergeWhile`. Here, the of number necessary calls can potentially be very large: For each $p \in P$, the validity of $\varphi \Rightarrow p$ has to be determined. Since such theorem prover calls are expensive, optimisations have be employed in order to avoid very long running times.

The most important such optimisation is to exploit implication relationships between the predicates: Often, P contains predicates p_1, p_2 such that $p_1 \Rightarrow p_2$. Then, $\varphi \Rightarrow p_1$ implies $\varphi \Rightarrow p_2$, and conversely $\varphi \not\Rightarrow p_2$ implies $\varphi \not\Rightarrow p_1$. Since usually most of the predicates will not hold, this second fact should be made use of: If checking $\varphi \Rightarrow p_2$ fails, then checking $\varphi \Rightarrow p_1$ is unnecessary. `#Abstract` supports this optimisation for the following classes of predicates (where $min, min_1, min_2, max, max_1, max_2 \in \mathit{Terms}$, $\psi \in \mathit{Formulas}$, and $\lhd, \lhd_1, \lhd_2, \lhd_3, \lhd_4 \in \{<, \leq\}$):

- $min \lhd max$
- $\forall x; (min \lhd_1 x \wedge x \lhd_2 max \rightarrow \psi)$
- $\forall x; \forall y; (min \lhd_1 x \wedge x \lhd_2 y \wedge y \lhd_3 max \rightarrow \psi)$
- $\forall x; \forall y; (min_1 \lhd_1 x \wedge x \lhd_2 max_1 \wedge min_2 \lhd_3 y \wedge y \lhd_4 max_2 \rightarrow \psi)$

For example, if P contains both $min < max$ and $min \leq max$ for some terms min and max, then $\varphi \Rightarrow min \leq max$ is checked first, and if this check fails, then $min < max$ is dismissed immediately without another theorem prover call. It also does not appear in the output, in order to avoid huge formulas with many redundant elements. The number of calls which can be saved by this optimisation is considerable: For example, in the last of the listed predicate classes, there are four comparison operators, so for every choice of the boundary terms there are $2^4 = 16$ predicates. In most cases, only the weakest of them (the one using $<$ for all four comparisons) has to be checked.

A second optimisation which is used by `#Abstract` is that predicates of the last three of the listed classes are not checked at all if $min \leq max$ (or $min_1 \leq max_1$ and $min_2 \leq max_2$ in case of the last class) has been checked without success. This is not justified by a logical necessity, but by the way in which such predicates are commonly used: The quantification of the variable is restricted to an interval confined by min and max. If there are many such predicates, this, too, can drastically reduce the number of theorem prover calls.

5.2 Generating Predicates

The set of predicates used by the abstraction rule forms the vocabulary from which invariants can be constructed. The larger this set is, the more powerful but also less efficient the inference process becomes. In general, custom predicates have to be supplied by the user; this is easier than the task of manually finding the invariants themselves. But the implementation also features a heuristic predicate generator, which automatically creates many predicates commonly occurring in invariants.

The predicate generator is called in the context of an application of *abstract* to a formula $\varphi \rightarrow \psi$. It is given as input both this formula and an optional set of predicates which have already been specified by the user. Its first action is identifying the local program variables which occur both in φ and in ψ: These are the only ones which are interesting at the current program point, since (i) no information is available about those not in φ, and (ii) those not in ψ are irrelevant for both the rest of the program and for the postcondition which we are trying to prove. This in particular excludes the many temporary program variables which are introduced by the calculus to cache, for example, the results of parts of complex expressions. Normally, these would be discarded by update simplification soon after being introduced, but *shift_update* prevents this. By excluding them from our predicates, we get rid of them during abstraction.

These interesting local program variables, plus the constant symbols 0 and `null`, form an initial set of terms. This set is then extended by deriving other terms from it: New terms are constructed by applying all non-rigid function symbols to all possible argument terms from the set. For example, if the original set contains the program variables `o` and `a`, terms like `o.f` and `a[0]` are added. In principle, this step could be repeated arbitrarily often to derive complex terms such as `o.f[i].g`, but this is currently not done in order to keep the number of predicates reasonably small.

The following predicates are then constructed from the obtained set of terms:

- For all boolean terms t: $t \doteq \texttt{true}$, $\texttt{t} \doteq \texttt{false}$.
- For all (syntactically different) integer terms t_1, t_2: $t_1 < t_2$, $t_1 \leq t_2$.
- For all (syntactically different) terms of a reference type t_1, t_2: $t_1 \doteq t_2$, $\neg t_1 \doteq t_2$.

No quantified formulas are created automatically, because doing so unspecifically quickly leads to an unacceptable number of predicates. However, the predicates defined by the user are allowed to contain free variables, and for each such predicate ψ quantified versions using various guards are created as follows:

- If ψ contains one free variable x: $\forall x; (min \lhd_1 x \wedge x \lhd_2 max \rightarrow \psi)$
  (for all integer terms min, max and all $\lhd_1, \lhd_2 \in \{<, \leq\}$)

- If ψ contains two free variables x and y:
  - $\forall x; \forall y; (min \lhd_1 x \land x \lhd_2 y \land y \lhd_3 max \rightarrow \psi)$
    (for all integer terms min, max and all $\lhd_1, \lhd_2, \lhd_3 \in \{<, \leq\}$)
  - $\forall x; \forall y; (min_1 \lhd_1 x \land x \lhd_2 max_1 \land min_2 \lhd_3 y \land y \lhd_4 max_2 \rightarrow \psi)$
    (for all integer terms $min_1, max_1, min_2, max_2$ and all $\lhd_1, \lhd_2, \lhd_3, \lhd_4 \in \{<, \leq\}$)

This mechanism does generate rather large numbers of predicates: For example, the number of predicates created by the last case for each such user predicate is $2^4 * n^4$, where n is the number of generated integer terms. However, the optimisations used by the `#Abstract` meta operator described in Section 5.1 apply to just these kinds of predicates and significantly reduce the number of theorem prover calls necessary for them.

5.3 Proof Search Strategy

The automatic application of rules in the KeY system is managed by *proof search strategies*. For example, KeY includes such strategies for first-order problems and for general JavaDL problems. Their main task is to prioritise the possible rule applications, usually based on the rule sets to which the applicable rules belong. Another task is instantiating schema variables which are used by a taclet but do not occur in its `\find` part. A new strategy is necessary to perform these tasks correctly for the invariant inference rules.

This invariant inference strategy basically assigns priorities according to the following scheme: Applications of the rules from the `invinference` rule set and the built-in rule implementing *shift_update* are given very high priorities, so they are preferred to applications of all other rules. Applications of rules from undesired conventional rule sets such as `alpha` and `beta` are given the lowest possible priority, meaning that they must not be performed at all. In the other cases, the task of determining the priority of a rule application is delegated to the regular JavaDL strategy.

There are two exceptions to this basic mode of operation. The first is related to the "merging" of control flow branches: If the sequent contains several such branches, their symbolic execution has to be coordinated so they can be merged later. For example, in the sequent $\vdash (\varphi_1 \rightarrow [\texttt{max = a[i]; i++;}]\psi) \land (\varphi_2 \rightarrow [\texttt{i++;}]\psi)$, as it occurs in the array maximum example, the execution of the second conjunct must be stopped until the execution of the then-branch in the first conjunct has completed and `merge` can be applied. The invariant inference strategy recognises such situations and forbids the application of symbolic execution taclets when necessary.

The second special case is setting priorities for applications of `abstract`. Such applications are usually forbidden. The only exception is when `merge_end_while` has been applied immediately before without terminating the iterating process for the loop: Then, `abstract` is given the highest priority of all, so it is ensured that it will be applied next.

When `abstract` is applied, the schema variable `predicates` must be instantiated. The invariant inference strategy does so depending on the loop which is the current active statement: If no abstraction has been performed before for this loop, then the predicate generator described in Section 5.2 is used to generate a new set of predicates. Otherwise, as an optimisation, the result of the last abstraction for this loop is used: The process of inferring an invariant for the loop is monotonic, i.e. the invariant candidates can only get weaker with each iteration. Therfore predicates which already did not hold in the previous iteration do not need to be considered again. The abstraction result which is used must however contain *all* predicates which held in the previous iteration, even the redundant ones which were simplified away in the output of the `#Abstract` meta operator (like $min \leq max$ in the presence of $min < max$).

5.4 User's View

After opening a problem in the KeY prover, the user can select "Invariant inference" on the "Proof Search Strategy" pane. This opens a dialogue window where custom predicates can be entered.[1] It is possible to select the first-order theorem prover to be used via the "Options" menu. The strategy can then be run by pressing the button labelled "start automated proof search". It terminates once a preselected number of rules have been applied or once the invariant inference process has completed.

The inferred invariants are not currently exported explicitly; rather, they occur in the resulting proof tree. The interesting branch of this tree is the one which is displayed uppermost: It corresponds to a normal execution of the program. Here, the leaf contains the inferred postcondition, and the inner node which follows the last application of `merge_end_while` for a loop contains the inferred loop invariant for that loop. A screenshot of the KeY system in such a situation is shown in Figure 5.3.

[1] This dialog is very provisional, as it is subject to a number of technical restrictions, the most severe being that only globally declared program variables can be parsed. A better input mechanism for user-defined predicates is considered as future work.

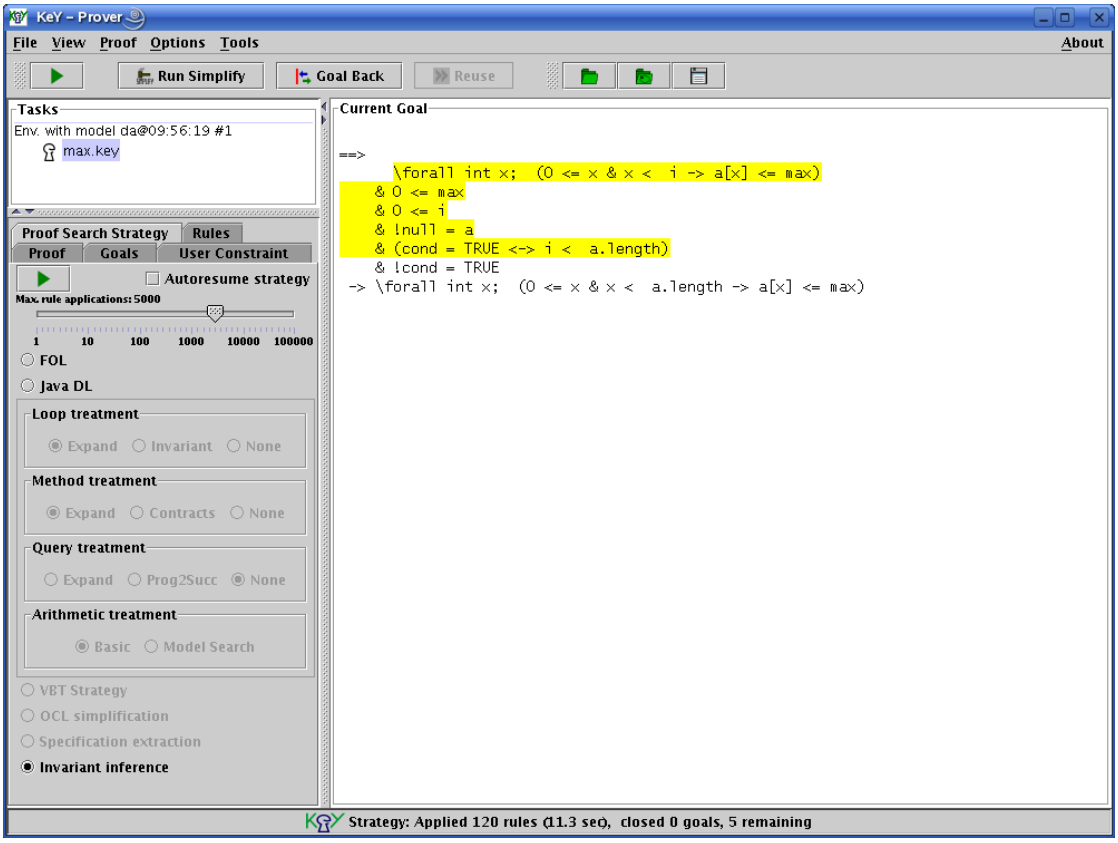

Figure 5.3: Screenshot of the KeY system after running the invariant inference strategy on a slightly modified version of the array maximum example program. The highlighted part of the the sequent on the right is the inferred loop invariant.

6 Experiments

The implementation described in Chapter 5 has so far been applied to two noteworthy examples, the first being the array maximum program familiar from the earlier chapters. How this example works out in practice is described in Section 6.1. The second, somewhat more complex example is the classical sorting algorithm *selection sort*; this is the subject of Section 6.2. All time measurements were performed on a machine with a clock frequency of 2 GHz and roughly 200 MB of free main memory.

6.1 Array Maximum

The following formula is a slightly modified version of the formula considered in Section 4.2:

$$\texttt{a} \doteq \texttt{null} \rightarrow [\{\ \texttt{max = 0;}\ \texttt{i = 0;}\ \texttt{cond = i < a.length;}\ \texttt{while(cond) \{}\ \texttt{if(a[i] > max) max = a[i];}\ \texttt{i++;}\ \texttt{cond = i < a.length;}\ \texttt{\}}\ \}]\forall x; (0 \leq x \wedge x < \texttt{a.length} \rightarrow \texttt{a}[x] \leq \texttt{max})$$

The only difference here is that the result of evaluating the expression `i < a.length` is buffered in an additional variable `cond`. This change does not affect the semantics of the program, but it is necessary due to technical limitations of the current implementation: Without it, `merge_end_while` would not be applicable, because it does not support complex expressions as loop conditions.

Using Simplify as the external first-order theorem prover, the following invariant can be inferred for the loop (cf. Section 4.2 and Figure 5.3):

$$\begin{aligned}
&\forall x; (0 \leq x \wedge x < \texttt{i} \rightarrow \texttt{a}[x] \leq \texttt{max}) \\
&\wedge\ 0 \leq \texttt{max} \\
&\wedge\ 0 \leq \texttt{i} \\
&\wedge\ \texttt{a} \doteq \texttt{null} \\
&\wedge\ (\texttt{cond} \doteq \texttt{true} \leftrightarrow \texttt{i} < \texttt{a.length})
\end{aligned}$$

When provided with this loop invariant and the modifier set $\{\texttt{max}, \texttt{i}\}$, the KeY system can automatically prove the formula to be valid in about 5 seconds.

To get the above result, the following predicates have to be manually specified by the user:

- $\texttt{cond} \doteq \texttt{true} \leftrightarrow \texttt{i} < \texttt{a.length}$
- $\texttt{a}[x] \leq \texttt{max}$ (where $x \in \mathcal{V}$)

The first one is only necessary because of the transformation which introduces the `cond` variable. In principle, this transformation could easily be automated and the predicate be added along the way.

After running predicate generation, the total number of predicates is 258. The invariant inference process terminates after two iterations. This is actually one iteration less than in Section 4.2; the reason is that there happen to be no predicates like $\texttt{i} \leq 1$ which still hold after the first iteration despite not being invariants. The number of applied rules is 120, and the total running time approximately 10 seconds. 61 calls to Simplify are needed, responsible for about 50 percent of the overall running time.

If Yices is used instead of Simplify, the same invariant is inferred in the same number of iterations. The running time however doubles to approximately 20 seconds, of which about 75 percent are used by Yices.

6.2 Selection Sort

Selection sort is a well-known sorting algorithm with quadratic time complexity. The basic idea is to find the smallest element of the array to be sorted, swap it with the first element, and iteratively repeat this process on the sub-array starting at the second position. An implementation in Java is given in Figure 6.1 as part of a formula which states that after running the algorithm the array is sorted.

Selection sort is more challenging for the invariant inference than the array maximum program mostly because it contains two nested loops. Using again Simplify as the external first-order theorem prover, the invariant given in Figure 6.2 can be inferred for the inner loop, and the one displayed in Figure 6.3 for the outer loop.

Provided with these loop invariants and the modifier sets $\{\texttt{minIndex}, \texttt{j}, \texttt{condInner}\}$ for the inner loop and $\{\texttt{minIndex}, \texttt{j}, \texttt{condInner}, \texttt{temp}, (\textit{for}\ x; \texttt{a}[x]), \texttt{i}, \texttt{condOuter}\}$ for the outer loop, the KeY system can automatically prove the formula to be valid in about 2 minutes. These modifier sets are rather obvious and could in principle be derived from the program automatically with relatively little effort.

$$\texttt{a} \neq \texttt{null} \wedge \texttt{a.length} > 0 \rightarrow [\{$$
```
  i = 0;
  condOuter = i < a.length;
  while(condOuter) {
    minIndex = i;
    j = i + 1;
    condInner = j < a.length;
    while(condInner) {
      if(a[j] < a[minIndex]) minIndex = j;
      j++;
      condInnder = j < a.length;
    }
    temp = a[i];
    a[i] = a[minIndex];
    a[minIndex] = temp;
    i++;
    condOuter = i < a.length;
  }
```
$$\}]\forall x; (0 < x \wedge x < \texttt{a.length} \rightarrow \texttt{a}[x-1] \leq \texttt{a}[x])$$

Figure 6.1: JavaDL formula for selection sort.

Again, a few predicates have to be specified manually by the user in order to make finding the mentioned invariants possible. In this case, these are:

- $\texttt{condOuter} \doteq \texttt{true} \leftrightarrow \texttt{i} < \texttt{a.length}$
- $\texttt{condInner} \doteq \texttt{true} \leftrightarrow \texttt{j} < \texttt{a.length}$
- $\texttt{a}[x] \leq \texttt{a}[y]$ (where $x, y \in \mathcal{V}$)
- $\texttt{a[minIndex]} \leq \texttt{a}[x]$ (where $x \in \mathcal{V}$)

Together with the automatically generated predicates, this leads to 8794 predicates for the inner loop and 16950 predicates for the outer loop.[1] The invariant inference process terminates after 3 iterations for the outer loop, containing 4, 4 and 2 iterations for the inner loop, respectively. 701 rules are applied in a total running time of about 13 minutes. Simplify is called 789 times, accounting for about 80 percent of the overall time consumption.

If Yices is used instead of Simplify, the invariant inference terminates after 4 iterations for the outer loop, comprising 976 rule applications and taking about 40 minutes. The

[1] The difference stems from the fact that for the outer loop, `temp` is considered as a program variable, whereas for the inner loop it is not: When reaching the inner loop, no information is ever known about `temp`.

$$
\begin{aligned}
&\forall x; \forall y; (0 \leq x \wedge x < y \wedge y \leq i \rightarrow \texttt{a}[x] \leq \texttt{a}[y]) \\
&\wedge\ \forall x; (\texttt{i} \leq x \wedge x < \texttt{minIndex} \rightarrow \texttt{a}[\texttt{minIndex}] \leq \texttt{a}[x]) \\
&\wedge\ \forall x; (\texttt{i} < x \wedge x < \texttt{j} \rightarrow \texttt{a}[\texttt{minIndex}] \leq \texttt{a}[x]) \\
&\wedge\ \forall x; (\texttt{minIndex} < x \wedge x < \texttt{j} \rightarrow \texttt{a}[\texttt{minIndex}] \leq \texttt{a}[x]) \\
&\wedge\ \texttt{a}[0] \leq \texttt{a}[\texttt{i}] \\
&\wedge\ \texttt{a}[\texttt{minIndex}] \leq \texttt{a}[\texttt{i}] \\
&\wedge\ 0 < \texttt{a.length} \\
&\wedge\ \texttt{j} \leq \texttt{a.length} \\
&\wedge\ 0 < \texttt{j} \\
&\wedge\ \texttt{minIndex} < \texttt{a.length} \\
&\wedge\ 0 \leq \texttt{minIndex} \\
&\wedge\ \texttt{minIndex} < \texttt{j} \\
&\wedge\ \texttt{i} < \texttt{a.length} \\
&\wedge\ 0 \leq \texttt{i} \\
&\wedge\ \texttt{i} < \texttt{j} \\
&\wedge\ \texttt{i} \leq \texttt{minIndex} \\
&\wedge\ \forall x; \forall y; (0 \leq x \wedge x < \texttt{i} \wedge \texttt{i} \leq y \wedge y < \texttt{a.length} \rightarrow \texttt{a}[x] \leq \texttt{a}[y]) \\
&\wedge\ \forall x; \forall y; (0 \leq x \wedge x < \texttt{i} \wedge \texttt{i} \leq y \wedge y < \texttt{j} \rightarrow \texttt{a}[x] \leq \texttt{a}[y]) \\
&\wedge\ \forall x; \forall y; (0 \leq x \wedge x < \texttt{i} \wedge \texttt{i} \leq y \wedge y < \texttt{minIndex} \rightarrow \texttt{a}[x] \leq \texttt{a}[y]) \\
&\wedge\ \texttt{a} \neq \texttt{null} \\
&\wedge\ \texttt{condOuter} \doteq \texttt{true} \\
&\wedge\ (\texttt{condOuter} \doteq \texttt{true} \leftrightarrow \texttt{i} < \texttt{a.length}) \\
&\wedge\ (\texttt{condInner} \doteq \texttt{true} \leftrightarrow \texttt{j} < \texttt{a.length})
\end{aligned}
$$

Figure 6.2: Inferred invariant for the inner loop of selection sort.

$$
\begin{aligned}
&\forall x; \forall y; (0 \leq x \wedge x < y \wedge y < \texttt{i} \rightarrow \texttt{a}[x] \leq \texttt{a}[y]) \\
&\wedge\ 0 < \texttt{a.length} \\
&\wedge\ \texttt{i} \leq \texttt{a.length} \\
&\wedge\ 0 \leq \texttt{i} \\
&\wedge\ \forall x; \forall y; (0 \leq x \wedge x < \texttt{i} \wedge \texttt{i} \leq y \wedge y < \texttt{a.length} \rightarrow \texttt{a}[x] \leq \texttt{a}[y]) \\
&\wedge\ (\texttt{condOuter} \doteq \texttt{true} \leftrightarrow \texttt{i} < \texttt{a.length}) \\
&\wedge\ \texttt{a} \neq \texttt{null}
\end{aligned}
$$

Figure 6.3: Inferred invariant for the outer loop of selection sort.

inferred invariants are very weak. It is unclear at this point to which degree this altogether disappointing performance of Yices in comparison to Simplify is to be attributed to Yices itself, and to which degree it may stem from the immaturity of the KeY module responsible for translating JavaDL sequents into the SMT-LIB format.

7 Related Work

Inferring invariants is a popular topic; recent works in this area include *Daikon* (Section 7.1) and several approaches from the contexts of *ESC/Java* and of *Spec#* (Section 7.2). Another somewhat related tool is *BLAST* (Section 7.3). Also, there are various other approaches within the KeY project for specification inference or for the integration of static analysis in one form or another (Section 7.4).

7.1 Daikon

Daikon [ECGN01] is a tool which infers invariants by *dynamic* analysis: The analysed program is instrumented at interesting program points, such as method entries and exits, with code which saves the current program state. Front-ends performing this instrumentation for a number of programming languages are available, including C, C++ and Java. The instrumented program is run through a user-specified test suite, producing a data base of states occurring at program points. This data base is then analysed for properties which held in all of the test cases and which thus are potential invariants.

Much like in our predicate abstraction based approach, only invariants from a pre-defined vocabulary can be inferred. In Daikon, this vocabulary is quite extensive, including simple properties like variables having constant values, ordering comparisons between two variables, and more complex properties like linear relationships between up to three variables, ordering relationships between array elements, or subset relationships between arrays.

The major difference to our approach is that Daikon requires the user to provide a test suite, and that the quality of its output heavily depends on this test suite: There is a realistic danger of inferring invariants which describe properties of the test cases rather than of the program itself. To some degree, Daikon counters this with statistical methods: The probability of an invariant holding "by chance" in the test suite is estimated, and the invariant is only reported if this probability is below an adjustable threshold.

When applied to the selection sort implementation from Section 6.2, using a test suite of 10 randomly generated arrays of random lengths between 100 and 250, Daikon ran for about 10 minutes and produced the output shown in Figure 7.1.[1]

[1] In its current implementation, Daikon does not support inferring loop invariants. This restriction can however be circumvented by adding calls to dummy methods at appropriate positions in the code:

```
a != null
a[] contains no duplicates
i >= 0
condOuter == true
minIndex >= 0
j >= 0
a[0..i] sorted by <
a[0..i-1] sorted by <
(minIndex == 0) ==> (i == 0)
i <= minIndex
(j == 0) ==> (i == 0)
i <= j
i <= size(a[])-1
i != a[i]
i != a[minIndex]
(j == 0) ==> (minIndex == 0)
minIndex <= j
minIndex <= size(a[])-1
minIndex != a[i]
minIndex != a[minIndex]
j <= size(a[])
j != a[i]
j != a[minIndex]
size(a[]) != a[i]
size(a[]) != a[minIndex]
size(a[])-1 != a[i]
size(a[])-1 != a[minIndex]
a[0..i] elements <= a[i]
a[0..i-1] elements < a[i]
a[i] >= a[minIndex]
a[0..i-1] elements < a[minIndex]
```

```
a != null
a[] contains no duplicates
i >= 0
a[0..i-1] sorted by <
i <= size(a[])
```

Figure 7.1: Daikon output for the inner (left) and outer (right) loop of selection sort.

This output comprises a large number of properties, but still not all of the invariants necessary for a verification of the program with KeY. For example, the essential invariant $\forall x; \forall y; (0 \leq x \wedge x < \texttt{i} \wedge \texttt{i} \leq y \wedge y < \texttt{a.length} \rightarrow \texttt{a}[x] \leq \texttt{a}[y])$ is missing, because invariants of this form are not considered by Daikon. Such limitations are present in our approach as well, since they are a direct consequence of using a finite vocabulary. The output however also shows the problem of false invariant reports, which is inherent only to the dynamic analysis approach: A significant number of the reported properties, including "`a[] contains no duplicates`" and most expressions which use the `!=` operator, are not really invariants; they just happened to be true in all of the test cases.

7.2 ESC/Java and Spec#

ESC/Java [FLL+02] is an automatic static verification system for Java which sacrifices soundness for performance and ease of use. In particular, it normally treats loops in a

The loop invariants are then inferred as postconditions for these methods.

very unsound way: A loop is unwound a fixed number of times (with default settings: once) and all further iterations are simply ignored. Optionally, a more precise treatment of loops can be switched on, which then requires user-specified loop invariants just like verification with the KeY system.

Houdini [FL01] is a tool which infers specifications in the format required by ESC/Java (which is basically a subset of JML) automatically. Its basic mode of operation is to generate a set of candidate specifications, insert them into the analysed program, and iteratively remove all specification elements which cannot be verified by ESC/Java until a fixed point is reached. Of course, this internal use of ESC/Java makes Houdini unsound, too.

A sound technique for inferring loop invariants, based on predicate abstraction and implemented as a part of ESC/Java, is described in [FQ02]. The predicate abstraction analysis is performed on the level of the intermediate language used internally by ESC/Java. The approach features support for automatically generating disjunctively composed predicates, and an optimised algorithm for computing the abstraction function in the presence of the ensuing large number of such predicates. Also, it proposes a way of manually specifying predicates in the style of JML, namely by inserting comments of the form `/*@ loop_predicate ... @*/` into the program code in front of loops.

The *Spec# programming system* [BLS05] consists of the Spec# language (which is an extension of C#), a compiler for Spec#, and an automatic static verification system for Spec# called *Boogie* [BCD+05]. The architecture of Boogie is somewhat similar to that of ESC/Java; in particular, it uses a similar internal intermediate language. Unlike ESC/Java however, Boogie is sound: Spurious error messages can occur, but no errors are systematically overlooked. Boogie, too, needs loop invariants to perform its verification task. If these have not been specified by the user, Boogie infers them by abstract interpretation based on classical domains like polyhedra [LL05] and a technique to extend them to support fields [CL05]. In cases where verification with Boogie fails, the Spec# compiler resorts to inserting runtime checks into the program.

7.3 BLAST

The *Berkeley Lazy Abstraction Software Verification Tool (BLAST)* [HJMS02, BHJM05] is an automatic static verification system for C. Its central data structure is the so-called *abstract reachability tree (ART)*, an infinite tree consisting of all paths through the control flow graph, whose nodes are program points labeled with a formula which describes the states that may occur at the program point if execution reaches it in the way defined by the tree. The ART is closely related to the proof tree created by symbolic execution in KeY if (i) loops are handled by unwinding, (ii) *assign_outer* is used for assignments, but (iii) no merging of control flow branches is performed: The paths through such a proof tree, too, correspond to the paths through the control flow graph, and its nodes, too, contain formulas which describe the occurring sets of states.

The ART is "abstract" because the formulas attached to its nodes are not necessarily the most precise descriptions of the occurring sets of states. Rather, they are overapproximations constructed from a finite set of predicates, much like in predicate abstraction. This allows BLAST to deal with the infinity of the ART due to loops: If a node has an ancestor standing for the same program point, and if the formula of the node implies that of its ancestor, then the descendants of the node do not need to be examined: All possible states have already been covered before. This corresponds to the detection of fixed points in abstract interpretation.

Unlike in predicate abstraction, the set of predicates used by BLAST is not fixed in advance. Instead, BLAST uses a methodology called *counterexample-guided abstraction refinement (CEGAR)* [CGJ+00]: It begins its verification with a very limited or even empty set of predicates. This extremely coarse abstraction typically leads to "spurious counterexamples", i.e. control flow paths in which specifications seem to be violated only because relevant properties have been abstracted away. Such counterexamples are then used to refine the set of predicates, and verification is tried again using the new set of predicates. This process is repeated until the verification succeeds or only counterexamples remain which cannot be disposed by adding predicates. BLAST innovates by making it unnecessary to start over the whole process for each discovery of a spurious counterexample: New predicates are detected and added locally in the ART during the verification, and other parts of the ART do not need to be revisited. This concept is called *lazy abstraction*.

BLAST now roughly works as follows: Using the current set of predicates, it constructs an ART in depth-first order. If a specification is violated in a node, BLAST analyses the path from the root to that node to find out whether the problem can really occur in the concrete program or whether it is a spurious counterexample. This analysis corresponds to a concrete symbolic execution of the path. If the counterexample is spurious, additional predicates are derived locally for the nodes of the path, such that tracking the values of these predicates is sufficient to show that the specification violation does not occur (this derivation of predicates is done using *Craig interpolation*; for details refer to [BHJM05]). Subtrees of the ART which are rooted in nodes for which predicates have been added are then rebuilt, and the normal ART construction process continues. It terminates when either a genuine counterexample is found or the ART is complete.

Thus, invariants are not explicitly constructed by BLAST, but they could be mined from a complete ART resulting from a run of BLAST: Disjoining all formulas attached to the ART nodes which represent a specific program point, such as a loop entry, yields an invariant for that program point.

Testing BLAST on a C implementation of the selection sort algorithm was not successful: BLAST wrongly warned that it is possible that the array is not sorted after running the algorithm. One reason for this is certainly that BLAST does not support quantified predicates. Moreover, at least in its current implementation "BLAST 2.0" using default settings, it appears to be very weak in dealing with arrays even when quantification is

unnecessary. As an example, consider the following trivial C program fragment:

```
int a[1];
a[0] = 238;
assert(a[0] == 238);
```

While BLAST can easily verify the correctness of this program, it fails on the following, only marginally more complicated variant, claiming that `a[0] != 238` may hold in the last line:

```
int a[1];
int i = 0;
a[i] = 238;
assert(a[0] == 238);
```

7.4 Related Work within KeY

[Pla04] discusses the topic of inferring postconditions from a theoretical perspective, and presents a technique which entails constructing a KeY proof tree and extracting a postcondition from the leaves of that tree. While the problem of inferring postconditions is closely related to that of inferring invariants, this technique is fundamentally different from ours in that the inferred postconditions are not in general formulas of first-order logic: In particular, for programs which contain loops, postconditions are inferred which themselves contain these loops (within modalities); this limits their practical usefulness.

An approach for implementing data-flow analyses within KeY's sequent calculus is described in [Ged05] on the example of reaching definitions analysis. Here, KeY is used to explicitly extract the data-flow equations from a program, which are then solved in a second step by an external tool. The basic idea for extracting the equations is to introduce a type for data-flow equations into JavaDL, so data-flow equations can occur as JavaDL terms. Statements of the form "the data-flow equations e correspond to the program fragment `p`" are then encoded as formulas of the form $\langle \mathtt{p} \rangle \mathit{wrapper}(e)$, where *wrapper* is a predicate symbol needed just to turn the construct into a syntactically legal formula. The equations for a program fragment `p` are extracted by constructing a "proof" for the sequent $\vdash \exists e; \langle \mathtt{p} \rangle \mathit{wrapper}(e)$, where $e \in \mathcal{V}$ is a variable of the data-flow equations type. This proof construction is done using not the normal rules of the JavaDL calculus, but entirely custom rules specific to the implemented data-flow analysis. In this process, information is propagated between branches of the proof tree through *meta variables* (see [BHS07]).

An application of a specific static analysis within KeY proofs is described in [GH06]. The idea is to have a special rule which transforms parallelisable loops, i.e. loops whose iterations are independent from each other, directly into a quantified update. This update

is constructed using repeated symbolic executions of the loop body until a fixed point is reached; this only works well if the body does not itself contain loops. Whether the loop iterations are independent from each other, i.e. whether the rule is really applicable, is then determined by running a *dependence analysis* on the constructed update. This dependence analysis is performed outside of the sequent calculus.

8 Conclusion

8.1 Summary

We have investigated an approach for automatically inferring invariants within the JavaDL sequent calculus used by the KeY system. The inferred invariants can be used for various purposes; in particular, inferred loop invariants can be employed to automatically verify loops in KeY with the invariant rule.

The approach is based on data-flow analysis, which is a static analysis technique routinely used in compilers to collect various kinds of information about the program being compiled. For one class of data-flow analyses, which are also called abstract interpretations, the collected pieces of information are invariants. Such an analysis can intuitively be understood as an approximative symbolic execution of the program, where (i) at confluences in the control flow graph, execution of the paths is merged together, and (ii) loops are iterated until the collected information does not change any more, i.e. until a fixed point is reached. That this eventually happens is ensured by the approximative nature of the used symbolic execution.

More precisely, the approach uses a variant of abstract interpretation called predicate abstraction. Here, the steps of symbolic execution are not approximative themselves. Instead, approximation is performed in between using an arbitrary, finite set of predicates: An invariant candidate formula is replaced with the conjunction of all those predicates from that set which are implied by it. This conjunction is weaker than the original formula, i.e. it carries less information. The set of predicates then defines the vocabulary from which invariants can be constructed.

Adding a small number of rules to the JavaDL sequent calculus allows to construct proofs in a way which corresponds to a predicate abstraction analysis. This is because the calculus, too, uses symbolic execution to handle programs. The additional rules serve to overcome differences in the symbolic execution concepts, such as the handling of assignments via updates and the treatment of control flow confluences, and perform approximation in the way defined by predicate abstraction.

The invariant inference approach has been implemented as a part of the KeY system. This implementation comprises the rules themselves, heuristics for automatically generating a set of predicates, and a proof search strategy which guides the automatic application of the rules. In particular, the strategy coordinates the symbolic executions of several modalities. Checking the validity of first-oder formulas, which must be done

e.g. for the predicate abstraction, is performed by calling an automated theorem prover such as Simplify in the background. A number of optimisations are used in order to reduce the number of such calls.

The quality of the inferred invariants heavily depends on the used set of predicates. On the other hand, expanding this set increases time consumption, so the default set which is generated heuristically must not be too large. In general, predicates which are desired to appear in the inferred invariants must be specified manually by the user. Apart from the predicates, the quality of the found invariants is determined by the strength of the used external theorem prover.

First experiments show the inference to work well on not-too-large examples. In order to find the invariants necessary for verification, moderate help from the user is necessary in the form of specifying predicates. The experiments also demonstrate the main advantage of static analysis compared to dynamic approaches such as Daikon: No formulas are falsely reported as invariants.

8.2 Future Work

In principle, inferring invariants together with first-order theorem proving is itself a way to verify programs: For example, whether a piece of code ensures a specified postcondition can be verified by inferring a postcondition and checking whether it implies the specified postcondition; no additional proof is necessary. However, the invariant inference currently makes use of rules which are not strictly sound, so the inferred formulas are not guaranteed to be invariants with the same level of confidence that is established by a regular KeY proof. Overcoming this limitation would be a primary direction for future work. Ensuring the soundness of *end_while* would entail formalising the requirement that a rule performs "symbolic execution", and using this formalisation to explicitly identify the rules which may be applied during invariant inference. The second soundness issue, namely the currently missing enforcement that certain taclets are applied only to elements of a conjunction in the succedent, is of a more technical nature, and could be resolved by using built-in rules instead of taclets or by extending the taclet language.

The implementation is currently not able to deal with Java exceptions properly. The problem is that exceptions introduce irregular control flow, which complicates the task of the proof search strategy to coordinate symbolic execution. As an example, consider the program fragment in Figure 8.1: Here, the conditional statement leads to two control flow paths, but they rejoin not at the end of the conditional statement (`i++`), but at a later point (`i--`). Such irregular control flow is also created by `break` and `continue` statements. Since it is not obvious from the program code (as it would be from a control flow graph), correct handling of such control flow in the proof search strategy is somewhat involved and has not yet been implemented.

```
try {
  if(b) {
    throw new Exception();
  } else {
    ...
  }
  i++;
} catch(Exception e) {
  ...
}
i--;
```

Figure 8.1: Example program using exceptions.

Another improvement would be devising a more comfortable mechanism for entering manually specified predicates, for example in the form of annotations in the program code as proposed by [FQ02]. Also, the heuristics for generating predicates can always be enhanced, although the number of generated predicates must not get too large; one could think about letting the user activate or deactivate certain classes of predicates depending on the current program and the envisaged form of the desired invariants. Related is the abstraction mechanism itself, whose optimisations are crucial in order to keep the number of theorem prover calls down, and which, too, can still be improved upon. A more fundamental direction would be to investigate how the necessary predicates could be detected more systematically, as it is for example done in BLAST.

Bibliography

[ABB+05] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[BCD+05] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects (FMCO 2005)*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.

[Bec01] Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, volume 2041 of *LNCS*, pages 6–24. Springer, 2001.

[BG01] Andreas Blass and Yuri Gurevich. Inadequacy of computable loop invariants. *ACM Transactions on Computational Logic*, 2(1):1–11, 2001.

[BGH+04] Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Andreas Roth, Philipp Rümmer, and Steffen Schlager. Taclets: A new paradigm for constructing interactive theorem provers. *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales, Serie A: Matemáticas (RACSAM)*, 98(1), 2004. Special Issue on Symbolic Computation in Logic and Artificial Intelligence.

[BHJM05] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Checking memory safety with Blast. In M. Cerioli, editor, *8th International Conference on Fundamental Approaches to Software Engineering (FASE 2005)*, volume 3442 of *LNCS*, pages 2–18. Springer, 2005.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.

[BSS05] Bernhard Beckert, Steffen Schlager, and Peter H. Schmitt. An improved rule for while loops in deductive program verification. In Kung-Kiu Lau, editor, *7th International Conference on Formal Engineering Methods (ICFEM 2005)*, volume 3785 of *LNCS*, pages 315–329. Springer, 2005.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Annual ACM Symposium on Principles of Programming Languages (POPL 1977)*, pages 238–252. ACM Press, 1977.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *12th International Conference on Computer Aided Verification (CAV 2000)*, pages 154–169. Springer, 2000.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th Annual ACM Symposium on Principles of Programming Languages (POPL 1978)*, pages 84–97. ACM Press, 1978.

[CL05] Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation (VMCAI 2005)*, volume 3385 of *LNCS*, pages 147–163. Springer, 2005.

[DdM] Bruno Dutertre and Leonardo de Moura. The YICES SMT solver. Tool paper, Computer Science Laboratory, SRI International.

[DNS03] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Laboratories Palo Alto, 2003.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[FL01] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *Formal Methods for Increasing Software Productivity (FME 2001)*, volume 2021 of *LNCS*, pages 500–517. Springer, 2001.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *ACM Conference on Programming Language Design and Implementation (PLDI 2002)*, pages 234–245. ACM Press, 2002.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In *Symposium on Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[FQ02] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *29th Annual ACM Symposium on Principles of Programming Languages (POPL 2002)*, pages 191–202. ACM Press, 2002.

[Ged05] Tobias Gedell. Embedding static analysis into tableaux and sequent based frameworks. In B. Beckert, editor, *Automated Reasoning with Analytic Tableaux and Related Methods (Tableaux 2005)*, volume 3702 of *LNAI*, pages 108–122. Springer, 2005.

[GH06] Tobias Gedell and Reiner Hähnle. Automating verification of loops by parallelization. In *13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2006)*, volume 4246 of *LNCS*, pages 332–346. Springer, 2006.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *9th International Conference on Computer Aided Verification (CAV 1997)*, pages 72–83. Springer, 1997.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *29th Annual ACM Symposium on Principles of Programming Languages (POPL 2002)*, pages 58–70. ACM Press, 2002.

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[JC06] Java Card platform specification 2.2.2. Sun Microsystems, 2006.

[JN95] Neil D. Jones and Flemming Nielson. Abstract interpretation: A semantics-based tool for program analysis. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 4, pages 527–636. Oxford University Press, 1995.

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[LL05] K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. In K. Yi, editor, *3rd Asian Symposium on Programming Languages and Systems (APLAS 2005)*, volume 3780 of *LNCS*, pages 119–134. Springer, 2005.

[Mey92] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[Min06] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19:31–100, 2006.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[OCL06] Object Constraint Language specification, version 2.0. Object Modeling Group, 2006.

[Pla04] André Platzer. Using a program verification calculus for constructing specifications from implementations. Studienarbeit, Universität Karlsruhe, 2004.

[RT06] Silvio Ranise and Cesare Tinelli. The SMT-LIB standard: Version 1.2. Technical report, University of Iowa, 2006.

[Rüm06] Philipp Rümmer. Sequential, parallel, and quantified updates of first-order structures. In *13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2006)*, volume 4246 of *LNCS*, pages 422–436. Springer, 2006.

Index