Secure Information Flow for Java
A Dynamic Logic Approach

– Extended Version –

Bernhard Beckert, Daniel Bruns, Vladimir Klebanov, Christoph Scheben,
Peter H. Schmitt, and Mattias Ulbrich

2013

KIT – University of the State of Baden-Wuerttemberg and National
Research Center of the Helmholtz Association
Please note:
This Report has been published on the Internet under the following Creative Commons License:
http://creativecommons.org/licenses/by-nc-nd/3.0/de.
Secure Information Flow for Java
A Dynamic Logic Approach
– Extended Version –

Bernhard Beckert, Daniel Bruns, Vladimir Klebanov, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich

Karlsruhe Institute of Technology (KIT), Dept. of Informatics
Am Fasanengarten 5, 76131 Karlsruhe, Germany

Abstract This is the full version of the paper submitted to FM 2012. In this paper we discuss and define an information flow property for sequential Java that takes into account information leakage through objects (as opposed to primitive values). We present proof rules for compositional reasoning over information-flow in Java programs. Our calculus rules apply at Java code-level (not at an abstraction), and they tie in with rules for functional verification. The new proof rules can be added to a Dynamic Logic calculus, as used in the KeY program verification system. The expressiveness of Dynamic Logic allows to specify and verify complex properties with high precision. The main novelty of our approach is that it uses efficient compositional information-flow reasoning wherever possible, but can resort to precise functional reasoning whenever necessary. In case none of the compositional rules apply, the information-flow property to be verified as formalised in Dynamic Logic using a variation of self-composition. Proof search then proceeds without sacrifice in precision.

1 Introduction
As distributed software systems are about to become ubiquitous in everyday life, there is more and more information that becomes electronically available. This raises the demand for confidentiality and integrity – and for the precise specification and verification of these security properties. In this paper, we target information-flow properties of Java programs. That is, we want to verify that an attacker who can observe “low” (or public) locations cannot deduce knowledge about the values of “high” (or secret) locations.

There have been static security-enforcing techniques based on syntax or types for a long time. While static checking of security type systems provides an attractive and efficient means to enforce non-interference, it is often overly conservative in practice. The reason is that type-based techniques are non-functional, i.e., they do not (and cannot) take functional properties into account. For example, a program like “low = high * 0” is secure, but to verify this one needs to reason about the functionality of *. Similarly, to verify that “if (high) {low = f1(low)} else {low = f2(low)}” is secure, one has to verify that f1 and f2 compute the same.

In contrast, functional program verification techniques tend to be very precise; they can handle the above examples. But the topic of information flow has reached the program verification world only recently. Joshi and Leino [13] and

⋆ This work was supported by the German National Science Foundation (DFG) under project “Program-level Specification and Deductive Verification of Security Properties” within priority programme 1496 “Reliably Secure Software Systems – RS³.”
Amtoft and Banerjee [2] were the first to give semantical definitions of informa-
tion flow. Their approaches are based on a comparison of two runs of the same 
program, which is sometimes known as relational verification. Many security 
properties can be defined in this way; the most widely used is non-interference: 
If any two runs start with the same public inputs, they must agree on the public 
outputs. In other words, the secret inputs must not influence public outputs. 

An easy way to encode relational properties in program logics – so that 
they can be verified using program verification calculi – is self-composition of 
programs (as proposed, e.g., in [5,7]): Through simple renaming one can make 
two copies of a program operate on disjoint variable sets. These copies can then 
be sequentially composed and their inputs and outputs compared to each other. 
In an earlier paper, we have presented a program-level specification language for 
information-flow properties (an extension of the Java Modeling Language) and 
a formalisation of self-composition in Dynamic Logic for Java [18].

It is a great advantage of the self-composition methodology that existing 
(functional) program verification systems and theorem provers can be used to 
verify information-flow properties – with very high precision. 

However, the self-composition approach was – so far – not compositional in 
the sense that it did not allow reasoning of the form “If \texttt{m1()} and \texttt{m2()} both 
do not have an information flow, then \texttt{m1();m2()} does not have an information 
flow.” Moreover, self-composition can be “overkill”; the program “\texttt{low = 0}”, e.g.,
does not need to be self-composed to verify that it has no information flow.

In this paper, we present compositional proof rules which allow the prop-
agation of information-flow properties from component programs to composite 
programs. They tie in with rules for functional verification. And they can be 
added to a Dynamic Logic calculus, as used in the KeY program verification 
system. In situations where none of these rules is applicable, we are still able 
to resort to self-composition. Thus, precision is not sacrificed for compositional 
reasoning.

A further main contribution of this paper is to discuss and define information-
flow properties for sequential Java that take into account information leakage 
through objects and heap structures (as opposed to primitive values).

Further, we introduce a rule to use information-flow contracts within func-
tional proofs, such that it becomes possible to use the results of compositional 
information-flow reasoning within functional reasoning.

Plan The plan of this extended technical report is as follows. In section 2 we 
present an almost complete introduction into the syntax and semantics of the 
Java Dynamic Logic, \texttt{JavaDL} as it is used in the KeY system. An extensive 
account of the semantics of the data type \texttt{Seq} of finite sequences is delegated to 
Appendix A. This material is completely independent of applications of KeY to 
information flow problems and may be used in other contexts as well. Section 3 
describes the observation of objects as opposed to the observation of primitive 
values in \texttt{Java} programs. It also contains an informal summary of the attacker 
model we have in mind. This report considers two ways to formalise the nota-
tion of an observation in \texttt{JavaDL}. The first possibility of the representation of 
observations by what is called observation sequences is studied in Subsections 
5.1 and 5.2. The second possibility using what we call reference set expressions 
is covered in Subsections B and B.1. The presentation is deliberately redundant. 
The observation sequences approach (Subsections 5.1 and 5.2) can be read inden-
dependently from the reference set approach (Subsections 5.1 and 5.2) and vice 
versa. The necessary prerequisites on isomorphisms needed in both cases are 
collected in Subsection 4. Proof rules for the information flow predicate are not 
covered here, but can be found in the thesis [17] extending this report. Section
6 discusses issues of the implementation of self-composition with particular emphasis on information flow contracts. Related work is cited in Section 7 while the notorious round-up, conclusions and future work, is given in Section 8.

2 Dynamic Logic for Java

In this section, we introduce syntax and semantics of a Dynamic Logic for Java, JAVADL as far as it is needed in this paper. An in-depth account can be found in [6,22]. JAVADL is an extension of classical typed first-order logic, with which we assume the reader is familiar. The following explanations only address particularities and the modal extension.

The type hierarchy for JAVADL is shown in Figure 1. Between Object and Null the class types from the Java code to be investigated will appear. There might also be additional data types at the level immediately below Any except Boolean, Int, LocSet and Seq.

![Figure 1. The JAVADL Type Hierarchy](image)

The vocabulary $\Sigma_{DL}$ of JAVADL is made up of two parts $\Sigma_{DL} = \Sigma_r \cup \Sigma_{nr}$. The symbols in $\Sigma_r$ are called rigid symbols, their interpretation does not depend of the program state. The remaining part $\Sigma_{nr}$ are the non-rigid symbols. Figure 2 shows a summary of all rigid function and predicate symbols. Besides the the symbol names Figure 2 also contains the typing information. For function symbols $f : T_1 \times T_2 \rightarrow T$ means that $f$ has two arguments required to be of type $T_1$ and $T_2$ respectively, and the return type of $f$ is $T$. For predicate symbols $p(T_1, T_2)$ indicates the $p$ is a binary predicate with argument types $T_1$ and $T_2$. For the typing of the equality symbol $=$ we have used the universal type $\top$ not shown in Figure 1. In many cases the name of a function or predicate symbol suggests its meaning. A precise definition, however, has to wait till we give the semantics of $\Sigma_r$.

The only state-dependent symbols in JAVADL i.e., the only symbols in $\Sigma_{nr}$, are program variables summarized in Figure 3.

The only other category of term-forming symbols we have not mentioned so far are logical variables. We thus arrive at the usual definition of terms $t$ and their (static) type type$(t)$:

**Definition 1.**

1. A logical variable $x$, a program variable $v$, or a rigid constant symbol $c$ are terms.
   
   $\text{type}(x)$, $\text{type}(v)$, $\text{type}(c)$ are the types declared of these symbols.

2. If $f : T_1 \times \ldots \times T_n \rightarrow T$ is an $n$-place function symbols and $t_1, \ldots, t_n$ such that $\text{type}(t_i) \subseteq T_i$ are terms so is $f(t_1, \ldots, t_n)$ with $\text{type}(f(t_1, \ldots, t_n)) = T$. 

3
all function and predicate symbols for Int, e.g., +,∗,<…

**Boolean constants** **TRUE, FALSE**

**Heap modeling**
- \( \text{select}_A : \text{Heap} \times \text{Object} \times \text{Field} \to A \) for any type \( A \subseteq \text{Any} \)
- \( \text{store} : \text{Heap} \times \text{Object} \times \text{Field} \times \text{Any} \to \text{Heap} \)
- \( \text{create} : \text{Heap} \times \text{Object} \to \text{Heap} \)
- \( \text{anon} : \text{Heap} \times \text{LocSet} \times \text{Heap} \to \text{Heap} \)
- \( \text{arr} : \text{Int} \to \text{Field} \)
- \( f : \text{Field} \) for all JAVA fields

**LocSet**
- \( \emptyset, \text{allLoc} : \text{LocSet} \)
- \( \text{singleton} : \text{Object} \times \text{Field} \to \text{LocSet} \)
- \( \cup, \cap : \text{LocSet} \times \text{LocSet} \to \text{LocSet} \)
- \( \text{allFields} : \text{Object} \to \text{LocSet} \)
- \( \text{arrayRange} : \text{Object} \times \text{Int} \times \text{Int} \to \text{LocSet} \)
- \( \text{unusedLocs} : \text{Heap} \to \text{LocSet} \)
- \( \in (\text{Object, Field, LocSet}) \)
- \( \subseteq (\text{LocSet, LocSet}), \text{disjoint}(\text{LocSet, LocSet}) \)

**Seq**
- \( \text{seqEmpty} : \text{Seq} \)
- \( \text{seqSingleton} : \text{Any} \to \text{Seq} \)
- \( \text{seqConcat} : \text{Seq} \times \text{Seq} \to \text{Seq} \)
- \( \text{seqSub} : \text{Seq} \times \text{Int} \times \text{Int} \to \text{Seq} \)
- \( \text{seqReverse} : \text{Seq} \to \text{Seq} \)
- \( \text{seqGet}_A : \text{Seq} \times \text{Int} \to A \) for any type \( A \subseteq \text{Any} \)
- \( \text{seqLen} : \text{Seq} \to \text{Int} \)

**JAVA**
- \( \text{null} : \text{Null} \)
- \( \text{length} : \text{Object} \to \text{Int} \)
- \( \text{cast}_A : \text{Any} \to A \) for any type \( A \subseteq \text{Any} \)
- \( \text{instance}_A(\text{Any}) \) for any type \( A \subseteq \text{Any} \)
- \( \text{exactInstance}_A(\text{Any}) \) for any type \( A \subseteq \text{Any} \)

**Miscellaneous**
- \( \equiv (\top, \top) \)

**Figure 2.** \( \Sigma_r \), the heap-independent symbols of JAVA DL

3. If \( \phi \) is a first-order formula, and \( t_1, t_2 \) are terms with \( \text{type}(t_1) = \text{type}(t_2) = T \)
then if \( \phi \) then \( t_1 \) else \( t_2 \) is a term of type \( T \).

**JAVA program variables**
- **this** denoting the current object
- **method parameters**
- **local variables** these are e.g., needed in the investigations of loop bodies
- **modeling program variables**
- **heap** modeling the current heap
- **result** modeling the return value of a method

**Figure 3.** \( \Sigma_{nr} \), Program Variables in JAVA DL

JAVA formulas and terms are inductively built up from atomic formulas using propositional operators and quantifiers, as usual, except for the clauses in the following definition. The operators in items 1 and 2 are modal operators. The constructs in items 3 and 4 are usually referred to as generalized quantifiers. They share with the usual existential and universal quantifiers the fact that they bind variables.

**Definition 2.** This definition lists clauses for constructing terms and formulas that are not present in textbook versions of first-order logic.
1. \(\{a := t\}\phi\) is a JAVADL formula, where \(a\) refers to a location (a program variable, a static or dynamic field, or an array entry), \(t\) is a JAVADL term \(t\), and \(\phi\) is a formula. The construct \(\{a := t\}\) is called an update.

2. \((\alpha)\phi, [\alpha]\phi\) are JAVADL formulas for any JAVADL formula \(\phi\) and any sequential Java program \(\alpha\).

3. For every integer variable \(iv\), JAVADL terms \(t_1, t_2\) with type \(Int\), not containing \(iv\) and JAVADL expression \(e\)

\[
\text{seq Def}\{iv\}(t_1, t_2, e)
\]

is a term of type \(Seq\).

4. For every integer variable \(iv\) and JAVADL expression \(e\) of type \(LocSet\)

\[
\text{infiniteUnion}\{iv\}(e)
\]

is a term of type \(LocSet\).

The KeY system is more general and also allows the infinite union construction with \(iv\) a variable of arbitrary type. The case included here, with \(iv\) an integer variable is strong enough for all purposes we need to consider here.

The basis for the semantics of JAVADL is provided by a structure \(D\) for typed first-order logic, called the computation domain.

**Definition 3.** The universe \(D\) of \(D\) is divided into the interpretations \(T^D\) for the types \(T\) occurring in the language. This definition will be extended by the description of \(Seq^D\) in Definition 19 on page 31. For now we have:

- \(Int^D = \mathbb{Z}\),
- \(\text{Boolean}^D = \{t, f\}\),
- \(\text{Object}^D = \text{the set of all Java objects},\)
- \(\text{LocSet}^D = \mathcal{P}\{\{o, f\} \mid o \in \text{Object}^D, f \in \text{Field}^D}\),
- \(\text{Any}^D = \text{Int}^D \cup \text{Boolean}^D \cup \text{Object}^D\),
- \(\text{Null}^D = \{\text{null}\}\),
- \(\text{Heap}^D = \text{the set of all functions} h : \text{Object}^D \times \text{Field}^D \rightarrow \text{Any}^D\),
- \(\text{Field}^D\) contains for every field \(f\) occurring in the Java program under investigation its interpretation \(f^D\). There might, however, be other element in \(\text{Field}^D\).

We have used the notation \(\mathcal{P}(S)\) to denote the set of subsets of \(S\). Thus \(\text{LocSet}^D\) consists of all sets of pairs \(\{o, f\}\) with \(o \in \text{Object}^D\) and \(f \in \text{Field}^D\). Restriction to finite sets would probably not hurt, but we do not require this.

The subset inclusion relations among \(T^D\) follow from the hierarchy shown in Figure 1. In particular, \(\text{Heap}, \text{Field},\) and \(\text{Any}\) are pairwise disjoint. For any Java class \(T\) its interpretation \(T^D\) is infinite. It comprises all potential objects of type \(T\). Below we will define the notation of a state \(s\) that covers the intuitive understanding of a program state. Even without a formal definition of a state, we can at this point already explain to the reader that the objects already created in state \(s\) will be those for which the implicit Boolean field created evaluates to true in \(s\), i.e., \(\text{created}(t) = t\). Having said this, we will deliver the whole truth. For every Java class \(T\) we require that there are infinitely many elements of exact type \(T\). That is to say there are infinitely many objects in \(T^D\) that are not in \(T_0^D\) for any subtype \(T_0 \subseteq T\). For any class \(C\) that occurs in a program to be analysed

\footnote{The definition is in fact more liberal in that \(a\) need not be a compilable program. Precisely, which program sequences are allowed is explained in [6, Section 3.2.4]. We will nevertheless use the term ‘program’ synonymously.}
C will be available as a type in JAVADL, and also all associated array types C[], C[][], etc. In the JAVADL semantics model we furthermore assume that the type universe C]^D is partitioned into infinite subsets C^n[] for n ≥ 1. The intention is that C^n[] contains the array object of length n. Corresponding provisions are made for arrays of higher dimensions.

The inclusion of the type Field in the syntax provides a weak reflection facility. It is possible to quantify in JAVADL over syntactic elements, in this case fields, themselves.

To complete the definition of the semantic domain D we need to give the definition of all rigid symbols in Σr. The integer operations are defined as usual. We postpone the explanation how we treat undefined values, e.g., division by 0 after first presenting the semantics of all symbols in Σr.

**Definition 4.** The semantics of the data type Seq of finite sequences will be presented in Appendix A. The interpretations of the other symbols are as follows:

1. For the Boolean constants we have TRUE^D = tt and FALSE^D = ff.
2. select^D_A(h, o, f) = cast^D_A(h(o, f))
3. For arguments h, o, f, x of appropriate types the function value h^* = store^D(h, o, f, x), which is itself a function, is given by
   \[ h^*(o', f') = \begin{cases} x & \text{if } o' = o, f = f' \text{ and } f \neq \text{created}^D \\ h(o', f') & \text{otherwise} \end{cases} \]
4. For arguments h, o of appropriate types the function value h^* = create^D(h, o) is given by
   \[ h^*(o', f) = \begin{cases} tt & \text{if } o' = o, o \neq \text{null and } f = \text{created}^D \\ h(o', f) & \text{otherwise} \end{cases} \]
5. For arguments h, s, h' of the appropriate types the function value h^* = anon^D(h, s, h') is given by
   \[ h^*(o, f) = \begin{cases} h'(o, f) & \text{if } ((o, f) \in s \text{ and } f \neq \text{created}^D) \\ h(o, f) & \text{otherwise} \end{cases} \]
6. arr^D is an injective function from \(\mathbb{Z}\) into Field^D,
   created^D and \(\text{f}^D\) for each JAVA field are pairwise different elements in Field^D and also not in the range of arr^D.
7. \(\emptyset^D = \emptyset\), allLocs^D = Object^D \times Field^D
8. singleton^D(o, f) = \{(o, f)\}
9. \(\cup^D\) and \(\cap^D\) are the set theoretical union and intersection of sets of locations.
10. allFields^D(a) = \{(o, f) \mid f \in Field^D\}
11. arrayRange^D(o, i, j) = \{(o, arr^D(x)) \mid x \in \mathbb{Z}, i \leq x, x \leq j\}
12. unusedLocs^D(h) = \{(o, f) \in \text{allLocs}^D \mid o \neq \text{null}, h(o, \text{created}^D) = \text{ff}\}
13. The usual set theoretic definitions:
   \(\in^D = \{(o, f, s) \in \text{Object}^D \times \text{Field}^D \times \text{LocSet}^D \mid (o, f) \in s\}\)
   \(\subseteq^D = \{(s, s') \mid \forall o, f((o, f) \in s \implies (o, f) \in s')\}\)
   \(\text{disjoint}^D = \{(s, s') \mid s \cap s' = \emptyset\}\)
14. null^D = null
15. cast^D_A(o) = \begin{cases} o & \text{if } o \in A^D \\ \text{default}_A & \text{otherwise} \end{cases}

The default element \(\text{default}_A\) for type A is as follows:

\[
\text{default}_A = \begin{cases} \emptyset & \text{if } A \subseteq \text{Object} \\ \text{seqEmpty} & \text{if } A = \text{LocSet} \\ \text{ff} & \text{if } A = \text{Boolean} \end{cases}
\]
16. instance^D_A = A^D \cup \{B^D \mid B \supset A\}
18. \( \text{length}^D(o) = \begin{cases} n & \text{if } o \in C^n \text{ for some } C \\ 0 & \text{otherwise} \end{cases} \)

We now come back to the issue of undefinedness. In our semantics all function symbols are interpreted by total functions, total with respect to their typing. Thus division is a total function \( \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \), and e.g., \((5/0)^D\) is a number in \( \mathbb{Z} \). The trick is that we do not know which value this is. Thus nothing can be logically derived, except that there is a value. This way to deal with undefindness is called *underspecification* and we illustrate its technical workings be giving the semantics of integer division:

\[
\begin{align*}
    n^D m &= \begin{cases} 
        \text{the uniquely defined } k \text{ such that } \\
        |m| \cdot |k| \leq |n| \text{ and } |m| \cdot (|k| + 1) > |n| \text{ and} \\
        k \geq 0 \text{ if } m, n \text{ are both positive or both negative and} \newline
        k \leq 0 \text{ otherwise} \\
        \text{undefined} \quad &\text{if } m \neq 0 \\
        \text{otherwise} 
    \end{cases}
\end{align*}
\]

Instead of one computation domain \( D \) we consider all computation domains \( D' \) covering all possible assignments of values to \( n^D \). The prover will use the axiom

\[
\forall x \forall y (y \neq 0 \to |y| \cdot |x/y| \leq |x| \land |y| \cdot ((x/y) + 1) > |x|/\land \\
( (x/y) \geq 0 \land x \geq 0 \land y \geq 0) \lor (x/y \geq 0 \land x \leq 0 \land y \leq 0) \lor \\
( x/y \leq 0 \land x \leq 0 \land y \geq 0) \lor (x/y \leq 0 \land x \geq 0 \land y \leq 0) )
\]

Then a formula \( \phi \) can be derived using this axiom if it is true in all computation domains. Thus \( \exists z (5/0 \equiv z) \) can be derived but \( 5/0 \equiv 7/0 \) cannot. For ease of presentation we will in the following nevertheless speak of the computation domain \( D \) and take care not to make use in any proof of the particular value assigned to undefined expressions.

The second way to deal with undefined values is to define them. For the *cast* function e.g., we have \( \text{cast}^D_A(o) = \text{null} \) if \( o \notin A^D \). Of course, when working with the *cast* function, specifying or reasoning, you have to remember this definition.

In clause 15 of Definition 4 special values for default elements \( \text{default}_A \) for some types \( A \) have been fixed. Notice that \( \text{default}_{int} \) does not occur in this list. It remains an underspecified constant symbol.

**Definition 5.**

1. A state is a function mapping all program variables to properly typed values in \( D \).
2. A computation domain \( D \) and a state \( s \) together yield a \( \Sigma_{DL} \) structure for first order logic, denoted by \( D + s \). \( D + s \) has the same domain as \( D \) and the interpretation of the rigid symbols is in \( D + s \) the same as in \( D \). The interpretation of the program variables \( v \) in \( \Sigma_{nr} \) is fixed as \( v^{D+s} = s(v) \).

For any state \( s \) and term \( t \) without logical variables the evaluation \( t^s \) is as usual. We trust that the reader is familiar with the semantics of conditional terms if \( \beta \) then \( t_1 \) else \( t_2 \). If \( t \) contains logical variables a variable assignment \( \beta \) is needed to evaluate the term to \( t^s, \beta \). In the following, we will omit \( \beta \) whenever it is not essential.

The recursive definition of when a formula \( \phi \) is true in state \( s \) with assignment \( \beta \) for free (logical) variables, in symbols \( (s, \beta) \models \phi \), follows the usual pattern. We again omit \( \beta \) whenever it is not essential. To be really precise we should even write \( (D + s, \beta) \models \phi \) to document that truth of \( \phi \) also depends on the
computation domain. But, since $D$ is understood to be there and leaving aside undefined values is uniquely determined, we do not mention it.

Only the additional semantic definitions from Definition 2 need explanation. First, we define updates $\{a := t\} \phi$ in which the left-hand-side is a location to be syntactic sugar for updates assigning to the program variable heap:

$$
\{a.f := t\} := \{\text{heap} := \text{store}(\text{heap}, a, f, t)\} \\
\{a[i] := t\} := \{\text{heap} := \text{store}(\text{heap}, a, arr(i), t)\} \\
\{sf := t\} := \{\text{heap} := \text{store}(\text{heap, null, sf, t})\}
$$

With this we define for a JAVA formula $\phi$ and state $s$:

1. $s \models \{a := t\} \phi$ iff $s'$ coincides with $s$ except for $s'(a) = t^*$.
2. $s \models (a) \phi$ iff $s' \models \phi$ for some $s'$ such that $\alpha$ started in $s$ terminates in $s'$.
3. $s \models [a] \phi$ iff $s' \models \phi$ for all $s'$ such that $\alpha$ started in $s$ terminates in $s'$.
4. $(\text{seq}_{- \text{def}}^f(\{iv\})(t_1, t_2, e))^{(s, \beta)}$ is the sequence of the elements $e^{(s, \beta[n/iv])}$ for all $n$ with $t_1^t \leq n < t_2^t$ in this order. If $t_1^t \geq t_2^t$ then $(\text{seq}_{- \text{def}}^f(\{iv\})(t_1, t_2, e))^{s} = \emptyset$.
5. $(\text{finiteUnion}(\{iv\})(e))^{(s, \beta)} = \{e^{(s, \beta[n/iv])} | i \in \mathbb{Z}\}$

Note that, if program $\alpha$ does not terminate when started in state $s$, then $s \models [a] \phi$ is trivially true for all formulas $\phi$, including $\phi \equiv \text{false}$.

**Digression** In Figure 2 that lists the rigid symbols $\Sigma$ of JAVA of every JAVA field $f$ in class $C_1$ of type $C_2$ a constant $f$ of type Field is included. In other approaches one would instead (or in addition) have a non-rigid function symbol $f : C_1 \rightarrow C_2$ in $\Sigma_{nr}$. In that approach a state is a $\Sigma_{nr}$ structure $S$ (with universe $D$), which is certainly a more complicated concept than our simple view from program variables to values in $D$. The non-rigid function symbol $f : C_1 \rightarrow C_2$ would in $S$ be interpreted as a function $f^S : C_1^D \rightarrow C_2^D$. There is an easy correspondence between these two approaches. A state $s$ in our sense defines a corresponding $\Sigma_{nr}$ structure $S$ via the definition

$$
\text{def}(s) = \text{select}^D_{C_2}(h, o, f^D)
$$

for all $o \in C_1^D$.

When we want to refer to the value of the field $f$ for an argument given by the expression $t$ we need to write in JAVA $\text{select}_{C_2}(\text{heap}, t, f)$. This is the price to be paid for the simplicity of our states. We will, nevertheless, sometimes write $f(t)$ or more JAVA-like $t.f$ as a shorthand for $\text{select}_{C_2}(\text{heap}, t, f)$. Note, that the (current) heap in a state $s$ is implicitly understood in this shorthand. Also on the semantic side we will write $f^s(o)$ for the value of field $f$ for object $o$ in state $s$ instead of $\text{select}^D_{C_2}(\text{heap}, o, f^D)$, with $C$ the type of $f$.

A decision had to be taken, how to treat static fields. In order to keep implementation efforts low the same mechanism $\text{select}_{C}(\text{heap}, t, f)$ is used to access values of a static field $f$. Since the value of a static field does not depend on any object the expression $t$ is taken to be null. For static fields $f$ we thus use $f$ as a shorthand for $\text{select}_{C}(\text{heap, null, f})$ and on the semantic side $f^s$ for the value $\text{select}^D_{C}(\text{heap}, null, f^D)$, with $C$ the type of $f$. This conforms with [22, page 102]

**Example 1.** Let us look at two examples of JAVA formulas:

$$
\forall \text{Int } i. \{0 \leq i \land i < \text{MAX _ VALUE}\} \rightarrow \\
\{a := i\} \{0 \leq r \land r * r \leq i \land (r + 1) * (r + 1) > i\} \tag{1}
$$

$$
\forall \text{Heap } h, h'. \forall \text{Int } i, i' \{(\text{select}_{\text{Any}}(h, \text{this}, f) \equiv \text{select}_{\text{Any}}(h', \text{this}, f) \land \\
\{\text{heap} := h\} (m) i \equiv r \land \{\text{heap} := h'\} (m) i' \equiv r\} \rightarrow i \equiv i'\} \tag{2}
$$

8
Formula (1) expresses that program \( \alpha \) with input variable \( a \) computes the positive integer square root for any positive JAVA integer (for ease of readability we have abbreviated result by \( r \)). Formula (2) states that the return value of method \( m \) only depends on the field \( \text{this}.f \).

Logical variables cannot occur in programs and program variables may not be quantified. As the above examples demonstrate, updates can be used as an interface between both types of variables.

We adopt the constant domain approach, i.e., all computation domains share the same universe \( D \). All potential objects are contained in \( D \) from the start. The generation of a new object \( o \) is effected by changing the value of \( o,\text{created} \) from \( \text{ff} \) to \( \text{tt} \). The objects for this change are chosen depending on the computation domain \( D \). If \( D_1 \), \( D_2 \) are computations domains, and \( s \) a state, then the next new object created in \( D_1 + s \) may differ from the next new object created in \( D_2 + s \). On the semantic level we have for each type \( T \) an (underspecified) function \( \text{nextToCreate}_T^D \) which determines for each interpretation \( D + s \) the value \( \text{nextToCreate}_T^D(s) \) of the next object to be created of type \( T \). We consider only functions \( \text{nextToCreate}_T^D \) where \( \text{exactInstance}_{T+1}(\text{nextToCreate}_T^D(s)) = \text{tt} \) and \( \text{created}_{T+1}(\text{nextToCreate}_T^D(s)) = \text{ff} \) holds.

**Definition 6.** The predicate \( \text{wellformed}(h) \) for a variable \( h \) of type Heap is an abbreviation of the formula

\[
1 \{ o \mid \text{select}_{\text{Boolean}}(h, o, \text{created}) = \text{TRUE} \} \text{ is finite} \\
2 \forall \text{Field } f \forall \text{Object } o (\text{select}_{\text{Object}}(h, o, f) = \text{null} \lor \text{select}_{\text{Boolean}}(h, \text{select}_{\text{Object}}(h, o, f), \text{created}) = \text{TRUE}) \\
3 \forall \text{Field } f, f' \forall \text{Object } o, o' (o', f') \in \text{select}_{\text{LocSet}}(h, o, f) \rightarrow \text{select}_{\text{Boolean}}(h, o', \text{created}) = \text{TRUE} \lor \\
4 \bigwedge_{\text{JAVA field } f} \forall \text{Object } o (\text{instance}_{\text{type}(f)}(\text{select}_{\text{Any}}(h, o, f))) \\
5 \bigwedge_{\text{JAVA type } A \subseteq \text{Any}} \forall \text{Int } i \forall \text{Object } o (i \geq 0 \land \text{instance}_{A}(o) \land o \neq \text{null}) \rightarrow (\text{instance}_{A}(\text{select}_{\text{Any}}(h, o, \text{arr}(i))))
\]

Some comments on Definition 6 are in order. Property 1 is obviously true for any state that can be reached via a JAVA program. Our experiments with program verification showed, surprisingly, that it is rarely ever used.

The value of an expression \( e.f \) in a JAVA program, where \( f \) is a fields of object type, always is an object, either the null object or a real object. By the language design of JAVA there are no dangling references. Property 2 formalizes this fact. Indeed, property 2 is much more general. It says that there are no dangling references for all fields in the model (not only those arising from a JAVA program), even for fields that are not of object type and for all objects, not only those that can be reached by the evaluation of JAVA expressions. Note, that for fields \( f \) with \( \text{type}(f) \not\subseteq \text{Object} \) the semantics definition yields \( \text{select}_{\text{Object}}^D(h, o, f) = \text{null} \).

Property 3 requires that only those location sets \( L \) can be values of fields of type \( \text{LocSet} \) that do not contain not-created objects \( o \). Of course objects do not occur directly as elements of \( L \), but only as the first component of a pair \( (o, f') \). Note as above, that for fields \( f \) that are not of type \( \text{LocSet} \) the semantics definition yields \( \text{select}_{\text{LocSet}}^D(h, o, f) = \emptyset \) and property 3 is trivially true.

Property 4 depends on the JAVA program \( \alpha \) under verification. The initial conjunction ranges of all fields occurring in \( \alpha \) and is thus finite. This conjunction cannot be replaced by a universal quantifier since the \( \text{type} \) function is not part of the JAVADL vocabulary. Property 4 is the substitute for depending types.

Also property 5 depends on the JAVA program \( \alpha \) under verification. The leading conjunction is finite, since \( \alpha \) contains only finite many types. As with
property 4 this property is needed since JAVADL does not use dependent types. It says: all entries in an array of type \( A[] \) are of type \( A \).

3 Information Flow in Java

Information leakage through object references is more involved than leakage through primitive data values. We will argue by way of examples that it is too strict to require different program runs to lead to identical behaviour. We pursue a language-based approach, this means that an attacker is only able to employ means provided by the Java language itself, i.e., they are able to evaluate expressions. They are not able to observe changes in the memory directly.

```java
final class C {
    static C x, y, z; // low variables
    static boolean h; // high variable

    static void m1() { x = new C(); }
    static void m2() { if (h) { x = new C(); } }
    static void m3() {
        if (h) { x = new C(); y = new C();}
        else { y = new C(); x = new C();}
    }
    static void m4() {
        if (h) { x = y; } else { x = z; }}
}
```

**Figure 4.** Information leaks through objects

We start with an informal discussion of the examples in Fig. 4 which will eventually lead us to a formal definition of information flow. In these examples \( x \), \( y \), and \( z \) are the low locations and \( h \) is the only high location. The non-interference property for any of the four methods would require that two independent executions of the method in two low-equivalent states result in states which are again low-equivalent. Let \( s_0 \) and \( s'_0 \) be low-equivalent states and \( s_i \) and \( s'_i \) the respective post-states for each \( m_i \).

If equivalence means identity, method \( m_1 \) would not be deemed secure. The reason is that the values of \( x^{s_1} \) and \( x^{s'_1} \) depend on the behaviour of the virtual machine which chooses the freshly created objects. The Java Virtual Machine Specification [14] does not impose any restrictions on the choice of new object references apart from the fact that they are not already in use. Therefore, we cannot ensure that the values \( x^{s_1} \) and \( x^{s'_1} \) are identical (nor that they are different). On the other hand, method \( m_1 \) obviously does not leak information. Thus, a simple non-interference condition based on object identity is too strict for an object-sensitive setting.

For method \( m_2 \) of Figure 4, the observation of an attacker depends on the value of the secret variable \( h \). The attacker can deduce that \( h^{s_0} \) is true if and only if the value of \( x \) changes. Information is leaked here. In contrast, method \( m_3 \) does not leak any information. Here, although the concrete values of \( x \) and \( y \) depend the value of \( h \), an attacker is not able to distinguish them.

In method \( m_4 \), it is important to notice that the attacker does not only observe the values of expressions, but knows the evaluation (i.e., the mapping from expressions to values) itself. The sets of values \( \{ x^{s_1}, y^{s_1}, z^{s_1} \} \) and \( \{ x^{s'_1}, y^{s'_1}, z^{s'_1} \} \) are equal in any case. However, the change made to \( x \) is observable.
We adopt the following passive attacker model: An attacker can evaluate a specified set of simple Java expressions in the pre- and post-state of a method. They see the expression and the corresponding evaluation as if they were printed on a screen. Further, we assume that the attacker knows the program-code. This allows them to trace back the observed differences in low values in the post-state to high values in the pre-state. In summary, an attacker

- can compare observed values that are of a primitive type to each other and to literals (of that type) as by using ==;
- can compare observed values of object reference type to each other and to null as by using the == predicate and observe their (runtime) type;
- cannot learn more than object identity from object references (e.g., the order in which objects have been generated cannot be learned).

Since an attacker sees the evaluations as if they were printed on a screen, they explicitly have not the power to dereference high fields, i.e., they cannot observe the value of o.f even if o is observable. An attacker that can dereference fields, can be modelled in this setting by declaring all locations o.f as low for which o may be an observable value of some other low expression. We will elaborate on this issue after the following clarifications:

Definition 7 (JavaDL Expressions). An expression e in JavaDL can be:

1. A program variable, most commonly the variable self.
2. Method parameters are also considered to be program variables.
3. e0.f if e0 is an expression of type C and f is a field declared in C (static or not).
4. ea[t] if ea is an expression of array type, and t is an expression of integer type.
5. op(e1,...,ek) where op is a data type operation and ei expressions of matching type. Most frequently arithmetic operations will occur.
6. b?e1:e2 the usual conditional operator. We (still) assume that e1, e2 are of the same type.
7. Auxiliary ghost variables, for the purpose of exposition only.

Expressions q(p1,...,pn) for queries q are not included. For uniformity of notation we will frequently write f(e0) instead of e0.f and assume that ea[t] is presented as at(ea,t).

Definition 8 (Observation Expressions). Observation expressions are recursively defined using generalized expressions as an auxiliary concept.

1. Any JavaDL expression is a generalized expression.
2. If e is a generalized expression of type T, f is an attribute defined in class T and also of type T, i an expression of type integer then it(f,i)(e) is a generalized expression.

1. A generalized expression e is an observation expression.
2. If R1 and R2 are observation expressions, so is R = R1; R2.
3. If e is a generalized expression, i a variable and from, to expressions of type integer then R = seq{[i]}(from, to, e) is an observation expression.

Definition 9 (Semantics of Observations Expressions). Let s be a state. The semantics of generalized expressions e and observation expressions R in state s is a kind of lazy evaluation, denoted by [e]s, [R]s, defined as

1. [e]s = e if e does not contain the construct it.
2. \([it(f,i)(e)]^s = f \ldots f([e]^s) \) (k times) with \(k = i^s\).

1. \([e]^s = \langle [e]^s \rangle\) for a generalized expression \(e\), i.e., the singleton sequence of the expression \([e]^s\).

2. \([R_1;R_2]^s = [R_1]^s;[R_2]^s\), i.e. the concatenation of the sequences \([R_1]^s\) and \([R_2]^s\).

3. \([\text{seq}(i)\{\text{from},\text{to},e\}]^s = (([e^{i\rightarrow 0}]^s),([e^{i\rightarrow n+1}]^s),\ldots,([e^{i\rightarrow m-1}]^s))\),
   if \(\text{from}^i = n < m = \text{to}^i\).
   Here \(e^{i\rightarrow n}\) is the expression obtained from \(e\) by replacing all occurrences of the variable \(i\) by the literal \(n\).

Example 2. Let \(R = \text{seq}\{i\}\{2,\text{to},a\text{.it}\text{.next.i}.\text{val}\}\) and assume \(s\) to be a state with \(\text{to}^s = 4\) then
\([R]^s = \langle \text{a\text{.next.next.val}} \rangle\).

**Definition 10.** By \(R_{\text{Obj}}^e\) we denote the subset of the expressions in the sequence \([R]^s\) that are of object type. On the other hand \(\text{Obj}(R^e) = \{e^s \mid e \in R_{\text{Obj}}\}\).

Given an observation expression \(R\) and a state \(s\), an attacker is able to see the tuple \(\langle [R]^s, R^e \rangle\), where \(R^e = \langle e^1, \ldots, e^k \rangle\) if \([R]^s = \langle e_1, \ldots, e_k \rangle\). Hence he is able to deduce for any \(0 \leq i < \text{length}([R]^s)\) that \(e^s_i\) is the value of the expression \(e_i\).

An attacker that can dereference fields, can be modelled in this setting by using only observations subject to the following closure property: Whenever \(e \in [R]^s\) with \(e^s = o\) for an object \(o\), then also expression \(e.f\) is in \([R]^s\) for all fields \(f\). We model the assumption that an attacker can observe the runtime type of an object similarly: Whenever \(e \in [R]^s\) with \(e^s = o\) for an object \(o\), then the observation expression \(R\) implicitly contains the expression \(e\text{.getClass()}\).

As the examples of Figure 4 show, information may flow through references, but non-identical behaviour is not a sufficient indication of a leak. Executions need not behave identically for different high inputs, but they must behave congruently with respect to reference comparison. This means that the post-states may be different as long as there is a kind of one-to-one correspondence between their references that is compatible with the identity comparison operation. In particular, the values of two locations storing references need to coincide in one post-state exactly if they do in the other.

In Java, object references are treated as opaque values. In a programming language where references have more structure (e.g., numeric pointers in C), attackers might be able to deduce more from the comparison of observations. If a particular memory manager happens to allocate memory in ascending order, an attacker of a C program analogous to \texttt{m3} could deduce that \(h^n\) is true if and only if the numerical value of \(x\) is less than the value of \(y\). Such inference is not possible in the Java language. Implementations of native methods, however, may provide some loopholes which leak structural information on references. Most notably, the native method \texttt{Object::hashCode()} returns the (encoded) memory address of a reference. This leakage potential can be dealt with by assigning a high security level to the output of native methods.

4 Isomorphisms

In the following we will repeatedly need the notion of an isomorphism of the computational domain \(\mathcal{D}\) and of isomorphic states. This section provides the necessary definitions.
We assume that the reader is familiar with the mathematical concept of isomorphism. We will consider isomorphisms only on the computational domain \( \mathcal{D} \), and the structures \( \mathcal{D} + s \) for different states \( s \).

We stipulate the following terminology.

**Definition 11.** If \( \pi \) is an isomorphism from \( \mathcal{D} + s_1 \) onto \( \mathcal{D} + s_2 \) we will say that \( s_2 \) is isomorphic to \( s_1 \) and write \( s_2 = \pi(s_1) \).

We will need the following - folklore - results:

**Lemma 1.** Let \( \rho \) be an automorphism of \( \mathcal{D} \), \( s \) a state, \( \phi \) a formula, \( e \) an expression, and \( \beta \) a variable assignment into \( \mathcal{D} \). Then

1. \( (s, \beta) \models \phi \iff (\rho(s), \rho(\beta)) \models \phi \)
   which reduces to \( s \models \phi \iff \rho(s) \models \phi \) if \( \phi \) contains no free variables.
2. \( \rho(\pi(s, \beta)) = \pi(\rho(s), \rho(\beta)) \)
   which reduces to \( \rho(\pi(s)) = \rho(e) \) if \( e \) contains no variables.

Since lazy evaluation only depends on the value of integer expressions and any automorphism is the identity on integers, we obtain:

**Lemma 2.** Let \( \rho \) be an automorphism of \( \mathcal{D} \), \( s \) a state, \( e \) a generalized expression and \( R \) an observation expression, both without variables.

1. \( [e]^\rho(s) = [e]^s \)
2. \( [R]^\rho(s) = [R]^s \)

The lemma can obviously be extended to allow variables.

**Lemma 3.** Any permutation \( \pi_0 \) of \( \text{Obj}^\mathcal{D} \) satisfying

1. \( \pi_0(\text{null}) = \text{null} \)
2. \( \pi_0 \) preserves the exact types of its arguments.
3. \( \pi_0 \) preserves the length of array objects.

can be extended to an automorphism \( \pi \) of \( \mathcal{D} \).

**Proof** We first describe how to extend \( \pi_0 \) to a bijection on \( \mathcal{D} \). For the following the reader might want to have again a look at the type hierarchy in Figure 1. On \( \text{Obj}^\mathcal{D} \) we let \( \pi \) necessarily coincide with \( \pi_0 \). We set \( \pi \) equal to the identity function on the type universes \( \text{Boolean}^\mathcal{D} \), \( \text{Int}^\mathcal{D} \), \( \text{Field}^\mathcal{D} \).

The action of the bijection \( \pi \) on \( \text{LocSet}^\mathcal{D} \) is obtained by extending the definition given so far on \( \text{Object}^\mathcal{D} \) and \( \text{Field}^\mathcal{D} \) to a mapping on sets of pairs: \( \pi(\text{LS}) = \{ (\pi_1(o), f) \mid (o, f) \in \text{LS} \} \). Since \( \pi : \text{Object}^\mathcal{D} \to \text{Object}^\mathcal{D} \) and \( \pi : \text{Field}^\mathcal{D} \to \text{Field}^\mathcal{D} \) are bijections also \( \pi : \text{LocSet}^\mathcal{D} \to \text{LocSet}^\mathcal{D} \) is a bijection.

Next we describe the action of the bijection \( \pi \) on \( \text{Seq}^\mathcal{D} \). By Definition 19 on page 31 \( \text{Seq}^\mathcal{D} = \bigcup_{n \geq 0} D^\mathcal{D}_{\text{Seq}} \). By construction \( D^\mathcal{D}_{\text{Seq}} \cap D^\mathcal{D}_{\text{Seq}} = \emptyset \) for \( i > 0 \) and \( D^\mathcal{D}_{\text{Seq}} \cap D^\mathcal{D}_{\text{Seq}} = \{ \langle \rangle \} \) for \( i > 0 \), \( j > 0 \), \( i \neq j \). We inductively define permutations \( \pi_{\text{seq}}^n \) of \( \bigcup_{0 \leq i \leq n} D^\mathcal{D}_{\text{Seq}} \). We start with \( \pi_{\text{seq}}^0 \) equal to the mapping \( \pi \) defined so far for

\[
D^\mathcal{D}_{\text{Seq}} = \text{Boolean}^\mathcal{D} \cup \text{Int}^\mathcal{D} \cup \text{Object}^\mathcal{D} \cup \text{LocSet}^\mathcal{D}.
\]

\[
\pi_{\text{seq}}^{n+1}(\langle \pi_{\text{seq}}(o_1), \ldots, \pi_{\text{seq}}(o_{n-1}) \rangle) = \langle \pi_{\text{seq}}(o_1), \ldots, \pi_{\text{seq}}(o_{n-1}) \rangle
\]

for \( o_i \in \bigcup_{0 \leq i \leq n} D^\mathcal{D}_{\text{Seq}} \). It is easily checked that \( \pi_{\text{seq}} = \bigcup_{n \geq 0} \pi_{\text{seq}}^n \) is a permutation of \( \text{Seq}^\mathcal{D} \).

The most involved case remains, to define \( \pi \) on \( \text{Heap}^\mathcal{D} \). Every \( h \in \text{Heap}^\mathcal{D} \) is a mapping \( h : \text{Object}^\mathcal{D} \times \text{Field}^\mathcal{D} \to \text{Any}^\mathcal{D} \). The mapping \( \pi(h) : \text{Object}^\mathcal{D} \times \text{Field}^\mathcal{D} \to \text{Any}^\mathcal{D} \) is defined by \( \pi(h)(o', f) = \pi(h(\pi^{-1}(o'), f)) \). As a consequence
of this definition we note \( \pi(h(o, f)) = \pi(h)(\pi(o), f) \). It is easily seen that \( \pi : \text{Heap}^D \rightarrow \text{Heap}^D \) thus defined is a bijection.

This completes the definition of the bijection \( \pi : D \rightarrow D \). We now embark on the lengthy verification that \( \pi \) is an \( \Sigma_r \) isomorphism. Consideration of the symbols in \( \Sigma_{nr} \) has to wait. We run through the list in Figure 2 from top to bottom; with the exception of the first item, making use of Definition 4 in each case.

1. \( \pi(cast_A^R(o)) = \pi(o) \) 
   \( = \pi(o) \) if \( o \in A^D \) 
   \( = \pi(o) \) if \( \pi(o) \in A^D \)
   
   The other three cases in the semantic definition of \( cast_A \) follow along the same line.

2. \( \pi(select_A^R(h, o, f)) = \pi(cast_A^R(h(o, f))) \) 
   \( = \pi(cast_A^R(h(o, f))) \) semantics of \( select_A \) 
   \( = \pi(cast_A^R(h(o, f))) \) see first item 
   \( = cast_A^R(\pi(h)(\pi(o), f)) \) def. of \( \pi(h) \) 
   \( = select_A^R(\pi(h), \pi(o), f) \) semantics of \( select_A \)
   
   In this argument we used \( \pi(f) = f \) for all fields. In the following we will throughout tacitly apply this equality.

3. We want to show that \( \pi(\text{store}^D(h, o, f, x)) = \text{store}^D(\pi(h), \pi(o), f, \pi(x)) \). 
   To this end we show for any argument pair \((o', f')\)

   \[ \pi(\text{store}^D(h, o, f, x))(o', f') = \text{store}^D(\pi(h), \pi(o), f, \pi(x))(o', f'). \] (3)

   By definition of \( \pi \) the lefthand side equals \( \pi(\text{store}^D(h, o, f, x)(\pi^{-1}(o'), f')) \). 
   In case, \( \pi^{-1}(o') \neq o \) or \( f \neq f' \) or \( f = \text{created}^D \) the semantics of \( \text{store} \) yields the further rewriting: \( \pi(h(\pi^{-1}(o'), f')) \). Which again be the definition of \( \pi \) is equal to \( \pi(h)(o', f') \).

   The case assumption implies \( o' \neq \pi(o) \) or \( f \neq f' \) or \( f = \text{created}^D \). By the semantics of \( \text{store} \) this leads to following rewriting of the righthand side of the equation (3)

   \[ \text{store}^D(\pi(h), \pi(o), f, \pi(x)) = \pi(h)(o', f') \] and we are done for the first case.

   In case \( \pi^{-1}(o') = o \) and \( f = f' \) and \( f \neq \text{created}^D \) the semantics definition yields: 
   \[ \pi(\text{store}^D(h, o, f, x)) = \pi(x) \].

   Since the case condition implies \( o' = \pi(o) \) and \( f = f' \) and \( f \neq \text{created}^D \) the righthand side of (3) evaluates to \( \pi(x) \), as desired.

4. Since \( \text{created} \) is a constant of type \( \text{Field} \) the definition of \( \pi \) yields \( \pi(\text{created}) = \text{created} \).

5. We need to show for all pairs \((o', f)\):

   \[ \pi(\text{create}^D(h, o))(o', f) = \text{create}^D(\pi(h), \pi(o))(o', f) \] (4)

   By definition of \( \pi \) the left side can be rewritten to

   \[ \pi(\text{create}^D(h, o)(\pi^{-1}(o'), f)). \]

   In case \( \pi^{-1}(o') \neq o \) or \( o = \text{null} \) or \( f \neq \text{created}^D \) the semantics of \( \text{create} \) yields

   \[ \pi(\text{create}^D(h, o)(\pi^{-1}(o'), f)) = \pi(h(\pi^{-1}(o'), f)). \] Again using the definition of \( \pi \) get \( \pi(h(\pi^{-1}(o'), f)) = \pi(h)(o', f) \).

   The case assumption implies \( o' \neq \pi(o) \) or \( \pi(o) = \text{null} \) or \( f \neq \text{created}^D \). The semantics of \( \text{create} \) for the righthand side in equation (4) yields

   \[ \text{create}^D(\pi(h)(\pi(o))(o', f)) = \pi(h)(o', f) \] as desired.

   In case \( \pi^{-1}(o') = o \) and \( o \neq \text{null} \) and \( f = \text{created}^D \) the semantics of \( \text{create} \)
yields $\pi(create^D(h, o)(\pi^{-1}(o'), f)) = \text{tt}$.

The current case assumption implies $o' = \pi(o)$ and $\pi(o) \neq \text{null}$ and $f \neq \text{created}^D$. Thus, according to the semantics of $create$ the righthand side of (4) evaluates also to true.

6. We want to show $\pi(anon^D(h, s, h')) = anon^D(\pi(h), \pi(s), \pi(h'))$.

The proof follows the pattern already seen in items 3 and 5. For the convenience of the reader we again give the details. To proof the goal just stated we show for all $o$ and $f$

$$\pi(anon^D(h, s, h'))(o, f) = anon^D(\pi(h), \pi(s), \pi(h'))(o, f)$$

(5)

By definition of $\pi$ the lefthand side of equation (5) can be rewritten as

$$\pi(anon^D(h, s, h'))(o, f) = \pi(anon^D(h, s, h') (\pi^{-1}(o), f))$$

In case $(\pi^{-1}(o), f) \in s$ and $f \neq \text{created}^D$ or $(\pi^{-1}(o), f) \in \text{unusedLocs}^D(h)$ this further evaluates to $\pi(h'(\pi^{-1}(o), f))$. Again by the definition of $\pi$ this can be further rewritten to yield the equation

$$\pi(anon^D(h, s, h'))(o, f) = \pi(h'(o, f))$$

By definition of $\pi$ on the type universe $LocSet^D$ we see that $(\pi^{-1}(o), f) \in s$ is equivalent to $(o, f) \in \pi(s)$ and $(\pi^{-1}(o), f) \in \text{unusedLocs}^D(h)$ is equivalent to $(o, f) \in \pi(\text{unusedLocs}^D(h))$. The case assumption thus implies $(o, f) \in \pi(s)$ and $f \neq \text{created}^D$ or $(o, f) \in \pi(\text{unusedLocs}^D(h))$. We will see below that furthermore $\pi(\text{unusedLocs}^D(h)) = \text{unusedLocs}^D(\pi(h))$ Thus the righthand side of equation (5) evaluates to $\pi(h'(o, f))$ as desired.

If the case assumption does not hold we obtain by the semantics of $anon$

$$\pi(anon^D(h, s, h') (\pi^{-1}(o), f)) = \pi(h(\pi^{-1}(o), f)) = \pi(h(o, f))$$

In this case the the righthand side of equation (5) also evaluates to $anon^D(\pi(h), \pi(s), \pi(h'))(o, f) = \pi(h(o, f))$ and we are done.

7. Since $arr^D(n) \in \text{Fields}^D$ we have $\pi(arr^D(n)) = arr^D(n)$. On the other hand $arr^D(\pi(n)) = arr^D(n)$ which in total gives $\pi(arr^D(n)) = arr^D(\pi(n))$.

We continue to run through the list in Figure 2 and now turn to the symbols under the heading $LocSet$.

8. $\pi(\emptyset) = \emptyset$ by the definition of $\pi$ on the type universe $LocSet^D$.

9. $\pi(allLocs^D) = \pi(Object^D \times Field^D)$ semantics of allLocs
   $= Object^D \times \pi(Field^D)$ def of $\pi$ on pairs
   $= Object^D \times Field^D$ surjectivity of $\pi$
   $= allLocs^D$ semantics of allLocs

10. $\pi\{\text{singleton}^D(o, f) = \pi(\{(o, f)\})\}$ semantics of $\text{singleton}$
    $= \{(\pi(o), \pi(f))\}$ def of $\pi$
    $= \text{singleton}^D(\pi(o), \pi(f))$ semantics of $\text{singleton}$

11. $\pi(LS_1 \cap LS_2) = \{(\pi(o), \pi(f)) \mid (o, f) \in LS_1 \cap LS_2\}$ def of $\pi$
    $= \{(\pi(o), \pi(f)) \mid (o, f) \in LS_1 \cap LS_2\}$ set theory
    $= \pi(LS_1) \cap \pi(LS_2)$ def of $\pi$

Similarly we can show $\pi(LS_1 \cup LS_2) = \pi(LS_1) \cup \pi(LS_2)$.

12. $\pi(allFields^D(o)) = \pi(\{(o, f) \mid f \in Fields^D\})$ semantics of allFields
    $= \{(\pi(o), f) \mid f \in Fields^D\}$ def of $\pi$
    $= allFields^D(\pi(o))$ semantics of allFields
13. \( \pi(\text{arrayRange}^D(o, i, j)) = \pi((\{o, arr^D(x) \mid z \in \mathbb{Z}, i \leq x \leq j\}) \)
semantics of \(\text{arrayRange}\)
\(= \{(\pi(o), \pi(\text{arr}^D(x))) \mid z \in \mathbb{Z}, i \leq x \leq j\} \)
def of \(\pi\)
\(= \{(\pi(o), \text{arr}^D(\pi(x))) \mid z \in \mathbb{Z}, i \leq x \leq j\} \)
item 7
\(= \{(\pi(o), \text{arr}^D(x)) \mid z \in \mathbb{Z}, i \leq x \leq j\} \)
\(\pi\) is identity on \(\mathbb{Z}\)
\(= \text{arrayRange}^D(\pi(o), i, j) \)
semantics of \(\text{arrayRange}\)

14. \( \pi(\text{unusedLocs}^D(h)) = \)
\(\pi((\{o, f \in \text{allLocs}^D \mid o \neq \text{null}, h(o, \text{created}^D) = \text{ff}\}) \)
semantics of \(\text{unusedLocs}\)
\(= \{(\pi(o), f) \in \text{allLocs}^D \mid o \neq \text{null}, h(o, \text{created}^D) = \text{ff}\} \)
def of \(\pi\) on \(\text{LocSet}^D\)
\(= \{(o', f) \in \text{allLocs}^D \mid o' \neq \text{null}, \pi(h)(o', \text{created}^D) = \text{ff}\} \)
def of \(\pi\)
\(= \text{unusedLocs}^D(\pi(h)) \)
semantics of \(\text{unusedLocs}\)

15. We need to show \((o, f) \in \text{LS} \iff (\pi(o), f) \in \pi(\text{LS})\). But, this is the very definition of \(\pi(\text{LS})\).

16. \(\text{LS}_1 \subseteq \text{LS}_2 \iff \pi(\text{LS}_1) \subseteq \pi(\text{LS}_2)\)
and \(\text{disjoint}(\text{LS}_1, \text{LS}_2) \iff \text{disjoint}(\pi(\text{LS}_1), \pi(\text{LS}_2))\)
follow easily from the definition of \(\pi(\text{LS}_i)\).

Next in the list in Figure 2 would be the symbols under the heading \(\text{Seq}\).

17. \( \pi(\text{seqEmpty}^D) = \pi(\langle \rangle) = \langle \rangle = \text{seqEmpty}^D \).
18. \( \pi(\text{seqSingleton}^D(o)) = \pi(\langle o \rangle) = \pi(o) = \text{seqSingleton}^D(\pi(o)) \).
19. Having seen the previous two examples we trust that the reader can do the remaining cases by himself.

20. \( \pi(\text{null}^D) = \text{null}^D = \text{null}, \pi(\text{length}^D(o)) = \text{length}^D(\pi(o))\) and the equivalence \(\pi(\text{exactInstance}^D(o)) \iff \text{exactInstance}^D(\pi(o))\) follow directly from the definition of the bijection \(\pi\).

This completes the proof that \(\pi\) is an automorphism of \(D\). \(\square\)

**Lemma 4.** Let \(\pi'\) be a bijection from \(X\) onto \(Y\) for finite subsets \(X, Y \subseteq \text{Obj}^D\) with

1. If \(\text{null} \in X\) then \(\pi'(\text{null}) = \text{null}\) and \(\text{null} \in Y\) implies \(\text{null} \in X\).
2. \(\pi'\) preserves the exact types of its arguments.
3. \(\pi'\) preserves the length of array objects.

Then there is an automorphism \(\pi\) on \(D\) extending \(\pi'\).

**Proof** To define an extension \(\pi_0\) of \(\pi'\) on \(\text{Obj}^D\) it suffices to explain what \(\pi_0\) does on the sets \(T_e^D\) of objects of exact type \(T\) for every JAVA class \(T\). First, we set \(\pi_0(\text{null}) = \text{null}\). By assumption this is compatible with \(\pi'\). By the assumed preservation of exact types and finiteness of \(X\) and \(Y\) we know that \(T_e^D \cap X\) and \(T_e^D \cap Y\) have the same finite number of elements. Since \(T_e^D\) is infinite we find a bijection \(\pi_0^\prime\) from \(T_e^D \setminus X\) onto \(T_e^D \setminus Y\). The bijection \(\pi_0\) on \(T_e^D\) is the disjoint union of \(\pi'\) and \(\pi_0^\prime\). This, of course, also applies to array types \(T = \text{C}[\square]\). In this case \(\pi_0\) is constructed in such a way that \(\text{C}[\square]\) is bijectively mapped onto itself for all \(n \geq 1\). Again, by assumption this is compatible with \(\pi'\).

By Lemma 3 there is an isomorphism \(\pi\) of \(D\) extending \(\pi_0\) and thus \(\pi'\). \(\square\)
Definition 12 (Partial Isomorphism). Let \( R \) be a observation expression and \( s_1, s_2 \) be two states such that \([R]^{s_1} = [R]^{s_2}\). A partial isomorphism with respect to \( R \) from \( s_1 \) to \( s_2 \) is a bijection \( \pi : Obj([R]^{s_1}) \to Obj([R]^{s_2}) \) such that the requirements of Lemma 4 hold.

Additionally \( \pi(e^{s_1}) = e^{s_2} \) must hold for all \( e \in [R]^{s_1} \).

It will greatly simplify notation in the following if we assume that every partial isomorphism \( \pi \) is also defined on all primitive values \( w \) with \( \pi(w) = w \).

In particular, if \( p \in [R]^{s_1} \) for all program variables \( p \), every automorphism extending a partial isomorphism \( \pi \) according to Lemma 4 is a (total) isomorphism from \( D + s_1 \) onto \( D + s_2 \) since \( \pi(p^{s_1}) = p^{s_2} \) by the last requirement.

Not every partial isomorphism can be extended to a total isomorphism, on the other hand. If \( q \) is a program variable such that \( q \) does not appear as a subterm in \([R]^{s_1}\), then \( \pi(q^{s_1}) = q^{s_2} \) is not required.

Example 3. To clarify the role of the additional condition in Definition 12 let \( x \) be a program variable of type \( C \) and \( f \) a field in \( C \), say of type integer such that \([R]^{s_1} = [R]^{s_2} = \langle x, f(x) \rangle \) for states \( s_1, s_2 \). In this case the condition implies

\[
\pi((f(x))^{s_1}) = (f(x))^{s_2} = f^{s_2}(x^{s_2}) = f^{s_2}(\pi(x^{s_1}))
\]

This amount to the usual requirements of isomorphisms on mathematical structures.

For later reference we state:

Lemma 5. Let \( s_1, s_2 \) be states and \( \rho \) an isomorphism on \( D \).

Let \( \alpha \) be a program which started in \( s_1 \) terminates in \( s_2 \).

Then \( \alpha \) started in \( \rho(s_1) \) terminates in \( \rho'(s_2) \), where \( \rho' \) is an isomorphism on \( D \) that coincides with \( \rho \) on all objects existing in state \( s_1 \), i.e. for all \( o \in Object^D \) with \( created^{s_1} = tt \) we know \( \rho(o) = \rho'(o) \).

(See Definition 11 for the definition of \( \rho(s_1) \))

Proof. The reason why we cannot assume \( \rho = \rho' \), is that \( \alpha \) may generate new objects and there is no reason why a new element \( o' \) generated in the run starting in state \( \rho(s_1) \) should be the \( \rho \)-image of the new element \( o \) generated in the run of \( \alpha \) starting in state \( s_1 \).

Let \( N_{T_1}^{s_1} \) be the set of new elements of exact type \( T \) generated in the run starting in state \( s_1 \) and \( N_{T}^{\rho(s_1)} \) be the set of new elements of exact type \( T \) generated in the run starting in state \( \rho(s_1) \). For the proof we need that both runs show the same termination behaviour and that \( N_{T_1}^{s_1} \) and \( N_{T}^{\rho(s_1)} \) have the same number of elements for each \( T \).

A strict proof of these statements would require a formal definition of \( \text{JAVA} \) semantics. We take them as postulates, and a very plausible postulates, how \( \text{JAVA} \) programs work.

Let \( G = \{ d \in D \mid created^{s_1} = tt \} \) be the finite set of elements that exist in state \( s_1 \) and \( \pi_0 \) the injective mapping defined on \( G \cup \bigcup_T N_{T_1}^{s_1} \) such that \( \pi_0(o) = \rho(o) \) for \( o \in G \) and the restrictions of \( \pi_0 \) map \( N_{T_1}^{s_1} \) bijectively on \( N_{T}^{\rho(s_1)} \). By Lemma 4 there is an automorphism \( \rho' \) of \( D \) extending \( \pi_0 \). This \( \rho' \) serves our purpose.

5 Formalizing Information Flow Properties

5.1 First Definition

Definition 13 (Agreement of states).

Let \( R \) be an observation expression.
We say that two states \( s, s' \) agree on \( R \), abbreviated by \( \text{agree}(R, s, s') \), iff

1. \( R\ = R' = \{e_1, \ldots, e_k\} \)
2. The mapping \( \pi \) defined by \( \pi((e_i)^s) = (e_i)^s' \) for \( e_i \in \text{Obj}(R) \) is a partial isomorphism.

The partial mapping \( \pi \) is uniquely determined by \( R\), \( s \) and \( s' \). We use the notation \( \text{agree}(R, s, s', \pi) \) to indicate that \( \text{agree}(R, s, s') \) is true and \( \pi \) is the mapping thus defined.

Notice, that because of our tacit agreement on the values of partial isomorphisms on primitive values \( \text{agree}(R, s, s', \pi) \) entails \( (e_i)^s = (e_i)^s' \) if \( e_i \) is an expression of primitive type.

We now define what it means for a program \( \alpha \) (when started in a state \( s \)) to allow information flow only from \( R_1 \) to \( R_2 \), which we denote by \( \text{flow}(s, \alpha, R_1, R_2) \).

The intuition is that \( R_1 \) describes the low location in the pre-state and \( R_2 \) describes the low locations in the post-state. Thus, the values of the variables and locations in \( R_2 \) in the post-state must at most depend – up to isomorphism of states – on the values of the variables and locations in \( R_1 \) in the pre-state and on nothing else.

The definition of flow is an extension of the one given by Amtoft and Banerjee [1], where a similar relation is defined using a different semantics formalism.

We consider here the termination insensitive case. Extensions taking termination into account, and also differentiate between normal and abnormal termination, are possible.

**Definition 14 (Information flow of a program).**

Let \( \alpha \) be a program and \( R_1 \) and \( R_2 \) be two observation expressions (of type \( \text{Seq} \))

Program \( \alpha \) allows information to flow only from \( R_1 \) to \( R_2 \) when started in \( s_1 \), denoted by \( \text{flow}(s_1, \alpha, R_1, R_2) \)

iff

- for all states \( s'_1, s_2, s'_2 \) such that
  - \( \alpha \) started in \( s_1 \) terminates in \( s_2 \) and
  - \( \alpha \) started in \( s'_1 \) terminates in \( s'_2 \),
- we have

  if \( \text{agree}(R_1, s_1, s'_1, \pi^1) \)

  then \( \text{agree}(R_2, s_2, s'_2, \pi^2) \) and \( \pi^2 \) is compatible with \( \pi^1 \)

where \( \pi^2 \) is said to be compatible with \( \pi^1 \) if

\( \pi^2(o) = \pi^1(o) \) for all \( o \in \text{Obj}(R_1^1) \cap \text{Obj}(R_2^2) \) with \( \text{created}^+(o) = \text{tt} \).

We extend JavaDL by a new three-place modal operator \( \text{flow}(\cdot, \cdot, \cdot) \) that expects a program as its first and reference set expressions as its second and third arguments. Its semantics is defined, for all states \( s \), by

\[ s \models \text{flow}(\alpha, R_1, R_2) \iff \text{flow}(s, \alpha, R_1, R_2) \text{ holds} \]

We think of \( R_1, R_2 \) as the publicly available information of a state of the system. In the simplest case what goes into \( R_i \) is determined by explicit declarations which program variables, and which fields are considered low. In more sophisticated scenarios views on the system for different users might be defined from which the \( R_i \) can then be inferred. In the most common case the low locations before program execution will be the same as the low locations after program execution. But, that might not be true in all cases. Thus we cover the more general case from the start.
Example 4.
The definition of information flow from Definition 14 is rather strict. Consider
the following program:

```java
class C {
    Int x, y, z;
    static boolean h;
    static void m(){
        if (h) {x = y} else {x = z}
    }
}
```

Let \( x \) be the only observable value, i.e., \( R_s = \{ \text{self.x} \} \) for all
states \( s \); then \( \text{flow}(m(), R, R) \) is not satisfied. The attacker can only learn that the
value of \( x \) he observes in the poststate is either the value of \( y \) or of \( z \) in the prestate. This
is already treated as information leakage.

Example 5.
This is a slight variation of the previous Example 4. The only difference is that
fields \( x, y, z \) now refer to objects rather than primitive values.

```java
class C {
    C x, y, z;
    static boolean h;
    static void m(){
        if (h) {x = y} else {x = z}
    }
}
```

Let again \( x \) be the only observable expression, i.e., \( R_s = \{ \text{self.x} \} \) for all
states \( s \); then \( \text{flow}(m(), R, R) \) is again not satisfied. The mapping \( \pi_2 \) defined by
\( \pi_2(x_s^i) = x_s^i \) with \( s, s' \) the poststates of \( m() \) when started in \( s_1 \), respectively \( s'_1 \),
is certainly a partial isomorphism. But, \( \pi_2 \) is not in the cases compatible with the
isomorphisms \( \pi_1 \) given by \( \pi_1(x_s^1) = x_s^1 \), e.g., not in the case \( h^1 = h^2 = \text{tt}, \ x_s^1 = y_s^1 \),
and \( x_s^1 \neq y_s^1 \).

The attacker can only see the object referred to by \( x \) in the poststate. Since
he knows that this equals either the object referred to by \( y \) or by \( z \) in the prestate, this
is considered an information leakage.

An often useful notion is subsumption of one observation by another. Here
is the most general definition.

**Definition 15.** Let \( R_1, R_2 \) be two observations. \( R_1 \) subsumes \( R_2 \), in symbols
\( R_2 \subseteq R_1 \), if for any two states \( s, s' \)
\[ \text{agree}(R_1, s, s', \pi_1) \quad \text{implies} \quad \text{agree}(R_2, s, s', \pi_2) \]

**Lemma 6.** Let \( R_1, R_2 \) be two observations such that \( R_2 \subseteq R_1 \), then for all
states \( s \)
\[ \text{Obj}(R_2^s) \subseteq \text{Obj}(R_1^s) \]

**Proof** This proof will make use of concepts and results from Subsection 4.

Assume, that there is a state \( s \) such that \( R_1 = \{ e_1, \ldots, e_n \} \), \( R_2 = \{ d_1, \ldots, d_m \} \),
and \( \text{Obj}(R_2^s) \not\subseteq \text{Obj}(R_1^s) \), i.e., there is an object \( o_1 \), say \( o_1 = d_1^s \), with \( o_1 \in \text{Obj}(R_2^s) \) but \( o_1 \not\in \text{Obj}(R_1^s) \).

By Lemma 4 there is an automorphism \( \rho \) of the computation structure \( D \)
such that for all \( o \in \text{Obj}(R_1^s) \) it is the identity, \( \rho(o) = o \), but \( \rho(o_1) \neq o_1 \). As
for any automorphism we have \( \rho(\ell) = \ell \) for any primitive value \( \ell \). In particular,
\(\rho(n) = n\) for all \(n \in \mathbb{N}\). Only object may be moved by \(\rho\). It is thus safe to assume \(\rho(R'_{i}) = R'_{i}\). We cannot prove this here, since we have not fixed a syntax for observations expressions. Altogether, we get \(e^{\rho(s)} = \rho(e^{s}) = e^{s}\) (See Definition 11 for the definition of \(\rho(s)\) and Lemma 1 for the equation.) This entails \(\text{agree}(R_{1}, s, \rho(s), \rho)\).

On the other hand because of \(d^{\rho(s)}_{i} = \rho(d^{i}_{s}) = \rho(a_{i}) \neq a_{i} = d^{i}_{s}\) we cannot have \(\text{agree}(R_{2}, s, \rho(s))\).

If for observation expressions \(R_{2}, R_{1}\) we have \(R^{2}_{i} \subseteq R^{1}_{i}\) for all states \(s\) then certainly \(R_{2} \subseteq R_{1}\). But for integer fields \(x, y\) we also have \(R_{0} = \{x, y, x + y\} \subseteq \{x, y\} = R_{1}\). Note, that in this example we have \(\text{Obj}(R^{2}_{i}) = \text{Obj}(R^{1}_{i}) = \emptyset\) for all \(s\).

The following lemma has been used to prove soundness of the rules of the calculus not included in this report, but is interesting in itself. The transitivity property, item 3 of Lemma 7, is the basis for compositional reasoning over the flow modality. It implies soundness of the rule \text{FlowSplit} in our calculus.

**Lemma 7.** The flow predicate satisfies the following properties:

1. \(\text{flow}(\epsilon, R_{1}, R_{2})\) if \(R_{2} \subseteq R_{1}\).
2. \(\text{flow}(\alpha, R_{1}, R_{2})\) implies \(\text{flow}(\alpha, R_{1}, R'_{2})\) if \(R'_{2} \subseteq R_{2}\).
3. if \(\text{flow}(\alpha_{1}, R_{1}, R_{2}), \text{flow}(\alpha_{2}, R_{2}, R_{3})\) and \(\text{Obj}(R^{1}_{i}) \cap \text{Obj}(R^{2}_{i}) \subseteq \text{Obj}(R^{2}_{i})\) for all \(s\) then \(\text{flow}(\alpha_{1}; \alpha_{2}, R_{1}, R_{3})\). Here, \(\alpha_{1}; \alpha_{2}\) is the concatenation of \(\alpha_{1}\) and \(\alpha_{2}\).

**Proofs**

**ad(1)** By Definition 14 we need to show for any states \(s_{1}, s'_{1}, s_{2}, s'_{2}\) such that \(\epsilon\) started in \(s_{1}\) terminates in \(s_{2}\) and started in \(s'_{1}\) terminates in \(s'_{2}\) that \(\text{agree}(R_{1}, s_{1}, s'_{1}, \pi_{1})\) implies \(\text{agree}(R_{2}, s_{2}, s'_{2}, \pi_{2})\) and \(\pi_{1}, \pi_{2}\) are compatible. For the empty program \(\epsilon\) we have \(s_{1} = s_{2}\) and \(s'_{1} = s'_{2}\). The claim thus reduces to showing that \(\text{agree}(R_{1}, s_{1}, s'_{1}, \pi_{1})\) implies \(\text{agree}(R_{2}, s_{2}, s'_{2}, \pi_{2})\) and the compatibility of \(\pi_{1}\) and \(\pi_{2}\). But, this follows from the definition of \(R_{2} \subseteq R_{1}\) and Lemma 6.

**ad(2)** To prove \(\text{flow}(\alpha, R_{1}, R'_{2})\) we need to show for any states \(s_{1}, s'_{1}, s_{2}, s'_{2}\) such that \(\alpha\) started in \(s_{1}\) terminates in \(s_{2}\) and \(\alpha\) started in \(s'_{1}\) terminates in \(s'_{2}\) that \(\text{agree}(R_{1}, s_{1}, s'_{1}, \pi_{1})\) implies \(\text{agree}(R'_{2}, s_{2}, s'_{2}, \pi'_{2})\) and the compatibility of \(\pi_{1}\) and \(\pi'_{2}\).

By the assumption \(\text{flow}(\alpha, R_{1}, R_{2})\) we know \(\text{agree}(R_{2}, s_{2}, s'_{2}, \pi_{2})\) plus compatibility of \(\pi_{1}\) and \(\pi_{2}\). The claim follows from \(R_{2} \subseteq R_{2}\). In particular compatibility of \(\pi_{1}\) and \(\pi'_{2}\) follows from the compatibility of \(\pi_{1}\) and \(\pi_{2}\) since \(\text{Obj}(R'_{2}) \subseteq \text{Obj}(R'_{2})\) by Lemma 6.

**ad(3)** We are given states \(s_{1}, s'_{1}, s_{2}, s'_{2}, s_{3}, s'_{3}\) such that \(s_{1} \overset{\alpha_{1}}{\rightarrow} s_{2}, s_{2} \overset{\alpha_{2}}{\rightarrow} s_{3}, s'_{1} \overset{\alpha_{1}}{\rightarrow} s'_{2}, s'_{2} \overset{\alpha_{2}}{\rightarrow} s'_{3}\), and we know from \(\text{flow}(\alpha_{1}, R_{1}, R_{2})\), \(\text{flow}(\alpha_{2}, R_{2}, R_{3})\) that \(\text{agree}(R_{1}, s_{1}, s'_{1}, \pi_{1})\) implies \(\text{agree}(R_{2}, s_{2}, s'_{2}, \pi_{2})\) and \(\text{agree}(R_{2}, s_{2}, s'_{2}, \pi_{2})\) implies \(\text{agree}(R_{3}, s_{3}, s'_{3}, \pi_{3})\). Thus \(\text{agree}(R_{1}, s_{1}, s'_{1}, \pi_{1})\) certainly implies \(\text{agree}(R_{3}, s_{3}, s'_{3}, \pi_{3})\) and it remains only to show compatibility of \(\pi_{1}\) and \(\pi_{3}\). We may make use of the facts that \(\pi_{1}\) and \(\pi_{2}\) on one hand and \(\pi_{2}\) and \(\pi_{3}\) on the other are compatible. So we fix \(o \in \text{Obj}(R'_{3}) \cap \text{Obj}(R'_{3})\) and want to show \(\pi_{1}(o) = \pi_{3}(o)\). Since by assumption \(o \in \text{Obj}(R'_{3})\) we get \(\pi_{1}(o) = \pi_{2}(o)\) from the compatibility of \(\pi_{1}\) and \(\pi_{2}\) and \(\pi_{2}(o) = \pi_{3}(o)\) from the compatibility of \(\pi_{2}\) and \(\pi_{3}\).

**Example 6.** It might be tempting to conjecture that \(\text{flow}(\alpha, R_{1}, R_{2})\) implies \(\text{flow}(\alpha, R'_{1}, R_{2})\) if \(R_{1} \subseteq R'_{1}\). Here comes a counterexample.

```c
class C {  
  x, y;
```
We argue that flow functions. We need to convince ourselves that agree easy since may not be compatible in some case, e.g., if agree onto a new element, while Example 7.

is nothing it could be compared to. Correspondingly, we have flow does not learn anything, as he only sees an object different from was not attacker knows (this statement is true for all programs). But that is not the whole story: The learn anything from running for part 3 of Lemma 7 is needed. Let we could conclude flow a run of the concatenation unchanged, then comparing the value of An state, this problem is avoided.

The example mentioned at the end of Section 3 can be handled as follows. We can talk and reason abstractly about observations by letting R just be a variable of type Seq. Thus satisfying the second requirement discussed above.

The example mentioned at the end of Section 3 can be handled as follows. We can talk and reason abstractly about observations by letting R just be a variable of type Seq. Thus satisfying the second requirement discussed above.

The example mentioned at the end of Section 3 can be handled as follows. We can talk and reason abstractly about observations by letting R just be a variable of type Seq. Thus satisfying the second requirement discussed above.

Example 7. The following example illustrates why condition

\[ \text{Obj}(R_1^1) \cap \text{Obj}(R_3^1) \subseteq \text{Obj}(R_2^2) \]

for part 3 of Lemma 7 is needed. Let

\[ \alpha_1 = \text{if} (h \neq \text{null}) \{ h = 1; \} \]

\[ \alpha_2 = \text{if} (h \neq \text{null}) \{ l = h; \} \text{ else } \{ l = \text{new } C(); \} \]

The program \( \alpha_1 \) satisfies flow(\( \alpha_1 \), \{l\}, \emptyset). This is because an attacker cannot learn anything from running \( \alpha_1 \) if he cannot observe anything in the post-state (this statement is true for all programs). But that is not the whole story: The attacker knows that the object he observed in 1 in the pre-state is stored in h if h was not null (as the attacker knows the program).

Considering (only) \( \alpha_2 \), an attacker who observes the (low) output variable 1 does not learn anything, as he only sees an object different from null and there is nothing it could be compared to. Correspondingly, we have flow(\( \alpha_2 \), \emptyset, \{l\}).

Ignoring the extra condition \( \text{Obj}(R_1^1) \cap \text{Obj}(R_3^1) \subseteq \text{Obj}(R_2^2) \) in Lemma 7(3), we could conclude flow(\( \alpha_1; \alpha_2 \), \{l\}, \{l\}). But that is not correct. By observing a run of the concatenation \( \alpha_1; \alpha_2 \), an attacker can learn something about h by comparing the value of 1 in the pre-state to its value in the post-state: If 1 is unchanged, then h was not null in the pre-state.

By demanding that all objects that an attacker knows from the pre-state and that are observable in the post-state must be observable in the intermediate state, this problem is avoided.

Definition 16. An observation expression R is of the form

\[ R = \text{seq \_ def \{} iv \text{\}}(t_1, t_2, e) \]

where \( t_i \) are expression of type integer with no occurrence of iv, and e is a expression of arbitrary type. Thus R is of type Seq.

For an explanation of the generalized quantifier seq \_ def \{iv\}(t_1, t_2, e) see Definition 2 (3) on page 4.

We can talk and reason abstractly about observations by letting R just be a variable of type Seq. Thus satisfying the second requirement discussed above.

The example mentioned at the end of Section 3 can be handled as follows. We first introduce a new binary function next(\( n, x \)), that we may also need for other purposes as well, by the recursive definition next(0, x) \( \doteq x \) and next(\( n+1, x \)) \( \doteq y \leftrightarrow \exists z (\text{next}(n, x) \doteq z \land \text{next}(z) \doteq y) \). Then we may write

\[ R = \text{seq \_ def \{} 0, this.\_len, next(i, this).v \text{\}} \]
Let Theorem 1. object-sensitive non-interference. section on information-flow in Java lead to the following formal definition of text only consider a single observation expression. The findings from the previous instead of one observation sequence. Without loss of generality, we will in this text only consider a single observation expression. The findings from the previous section on information-flow in Java lead to the following formal definition of object-sensitive non-interference.

**Theorem 1.** Let $\alpha$ be a program, and let $R_1, R_2$ be observation expressions. There is a formula $\phi_{\alpha,R_1,R_2}$ in JAVADL making use of self-composition such that:

$$s_1 \models \phi_{\alpha,R_1,R_2} \iff \text{flow}(s_1, \alpha, R_1, R_2).$$

Proof. The proof consists of a constructive definition of the formula $\phi_{\alpha,R_1,R_2}$ such that $s_1 \models \phi_{\alpha,R_1,R_2}$ iff $\text{flow}(s_1, \alpha, R_1, R_2)$.

We will explain the construction of $\phi_{\alpha,R_1,R_2}$ top down. The property to be formalized requires quantification over states. According to Definition 5 a state $s$ is determined by the value of the heap $h^s$ in $s$ and the values of the (finitely many) program variables $a^s$ in $s$. We can directly quantify over heaps $h$ and refer to the value of a field $f$ of type $C$ for object $o$ referenced by expression $e$ as $\text{select}_C(h,e,f)$. We cannot directly quantify over program variables, as opposed to quantifying over the values of program variables, which is perfectly possible. Thus we use quantifiers $\forall x, \exists x$ over the type domain of the variable and assign $x$ to $a$ via an update $a := x$. There are four states involved, the two pre-states $s_1, s_1'$ and the post-states $s_2, s_2'$. Correspondingly, there will be, for every program variable $v$, four universally quantifier variables $v, v', v^2, (v^2)'$ of appropriate type representing the values of $v$ in states $s_1, s_1', s_2, s_2'$. There are some program variables that make only sense in pre-states, e.g., $\text{this}$, and variables that make only sense in post-state, e.g., $\text{result}$. There will be only two logical variables that supply values to them instead of four. This leads to the following schematic form of $\phi_{\alpha,R_1,R_2}$:

$$\phi_{\alpha,R_1,R_2} \equiv \forall \text{Heap} \ h_1', h_2, h'_2 \forall T \forall T' r, r' \forall v, v', v^2, (v^2)' \ldots \left(\text{Agree}_\text{pre} \land (\langle \alpha \rangle \text{save}\{s_2\} \land \text{in}\{s_1'\} \langle \alpha \rangle \text{save}\{s_2'\} \right) \rightarrow (\text{Agree}_\text{post} \land \text{Ext}))$$

To maintain readability we have used suggestive abbreviations:

1. $\{\text{in } s_1\} \langle \alpha \rangle$ signals that an update $\{\text{heap} := h'_1 || \text{this} := o' || \ldots a_i := v_i' \ldots\}$ is placed before the modality operator. The $a_i$ cover all relevant parameters and local variables.
2. The construct $\text{save}\{s_2\}$ abbreviates a conjunction of equations $h_2 = \text{heap}$, $r = \text{result}$, $v = a_i$, $\ldots$.
3. Analogously, $\text{save}\{s'_2\}$ stands for the primed version $h'_2 = \text{heap}$, $r' = \text{result}$, $v' = a_i$, $\ldots$.
4. The shorthand $\{\text{in } s_2\} \in \{s'_2\} E$ in front of a formula is resolved by (a) prefixing every occurrence of a heap dependent expression $e$ with the update $\{\text{heap} := h_2\}$ and (b) every primed expression $e'$ with $\{\text{heap} := h'_2\}$.
5. The same applies to $\{\text{in } s_1\} E$. Note, there is no $\{\text{in } s_1\}$, and nor quantified variables $o, v^1$ since the whole formula $\phi_{\alpha,R_1,R_2}$ is evaluated in state $s_1$.

In the following we will also use the notation $R'_1$, $R'_2$, $(R'_2)'$ for the terms obtained from $R_i$ by replacing each state dependent designator $e$ by $v', v^2, (v^2)'$ respectively. Technically, these substitutions are effected by prefixing $R_i$ with an appropriate update.
For conciseness we use \( R[i] \) instead of \( seqGetAny(r, i) \) and also \( t \in A \) for \( instance_A(t) \).

We now supply the definitions of the abbreviations used above:

\[
Agree_{\text{pre}} \equiv R_1.\text{length} \equiv R'_1.\text{length}
\]
\[
\land \forall i (0 \leq i < R_1.\text{length} \rightarrow \bigwedge A \in \alpha \left( \text{exactInstance}_A(R_1[i]) \leftrightarrow \text{exactInstance}_A(R'_1[i]) \right))
\]
\[
\land \forall i (0 \leq i < R_1.\text{length} \land R_1[i] \notin \text{Object} \rightarrow R_1[i] \equiv R'_1[i])
\]
\[
\land \forall i, j (0 \leq i < j < R_1.\text{length} \land R_1[i] \in \text{Object} \land R_1[j] \in \text{Object} \rightarrow (R_1[i] \equiv R'_1[i] \leftrightarrow R'_1[j] \equiv R_1[j]))
\]

\[
Agree_{\text{post}} \equiv R_2.\text{length} \equiv (R'_2).\text{length}
\]
\[
\land \forall i (0 \leq i < R_2.\text{length} \rightarrow \bigwedge A \in \alpha \left( \text{exactInstance}_A(R_2[i]) \leftrightarrow \text{exactInstance}_A((R'_2)[i]) \right))
\]
\[
\land \forall i (0 \leq i < R_2.\text{length} \land R_2[i] \notin \text{Object} \rightarrow R_2[i] \equiv (R'_2)[i])
\]
\[
\land \forall i, j (0 \leq i < j < R_2.\text{length} \land R_2[i] \in \text{Object} \land R_2[j] \in \text{Object} \rightarrow (R_2[i] \equiv (R'_2)[i] \leftrightarrow (R'_2)[j] \equiv R_2[j]))
\]

\[
Ext \equiv \forall \forall j (0 \leq i < R_1.\text{length} \land 0 \leq j < R_2.\text{length} \land
R_1[i] \in \text{Object} \land R_2[j] \in \text{Object} \land R_1[i] \equiv R_2[j]
\rightarrow (R_1[i] \equiv (R'_2)[j])
\]

In many cases these definitions are much simpler. Frequently it is the case that \( R_1.\text{length} \) is not state dependend, then quantification over index \( i \) reduces to a disjunction of fixed length. Also the exact type of an expression can often be checked syntactically and need not be part of the formula. In other cases however, e.g., if \( R_1 \) is a variable of type \( \text{Seq} \), the full definition is necessary.

It remains to show that this definition does the job. There are two implications to be proved.

Let us first assume \( s_1 \models \phi_{\alpha, R_1, R_2} \). To prove \( \text{flow}(s_1, \alpha, R_1, R_2) \) fix states \( s'_{1, 2}, s'_2, s'_2 \) such that \( \alpha \) started in \( s_1 \) terminates in \( s_2 \), \( \alpha \) started in \( s'_1 \) terminates in \( s'_2 \), and \( \text{agree}(R_1, s_1, s'_1, \pi^1) \). We need to show that \( \text{agree}(R'^2, s_2, s'_2, \pi^2) \) and \( \pi^2 \) is compatible with \( \pi^1 \).

The universally quantified variables of \( \phi_{\alpha, R_1, R_2} \) will be instantiated by the variable assignment \( \beta \) as follows \( \beta(h'_1) = s'_1(\text{heap}) \), \( \beta(o') = s'_1(\text{this}) \), and \( \beta(v') = s'_1(v) \) for all other \( v \). From \( \text{agree}(R_1, s_1, s'_1, \pi^1) \) we see that \( (s_1, \beta) \models Agree_{\text{pre}} \) is true. Extending \( \beta \) by \( \beta(v^2) = s_2(v) \) for all \( v \) we obtain \( (s_1, \beta) \models (\alpha)\text{save}(s_2) \) and, finally, setting \( \beta(\langle v^2 \rangle') = s'_2(v) \) we also have

\[
(s_1, \beta) \models \text{in}(s'_1)(\alpha)\text{save}(s'_2).
\]

Thus, our assumption \( s_1 \models \phi_{\alpha, R_1, R_2} \) implies \( (s_1, \beta) \models (\text{in } s_2)\{ (\text{Agree}_{\text{post}} \land \text{Ext}) \). The part \( (s_1, \beta) \models (\text{in } s_2)\{ (\text{Agree}_{\text{post}} \land \text{Ext}) \) implies \( (s_1, \beta) \models (\text{in } s_2)\{ (\text{Ext}) \) while \( (s_1, \beta) \models (\text{in } s_2)\{ (\text{Ext}) \) Ext guarantees that \( \pi^2 \) is compatible with \( \pi^1 \). In total \( \text{flow}(s_1, \alpha, R_1, R_2) \) has been shown.

For the reverse implication assume \( \text{flow}(s_1, \alpha, R_1, R_2) \). We set out to prove \( s_1 \models \phi_{\alpha, R_1, R_2} \). Let \( \beta \) be an arbitrary assignment for the universally quantified variables of this formula. Our task is reduced to showing

\[
(s_1, \beta) \models \text{Agree}_{\text{pre}} \land (\alpha)\text{save}(s_2) \land \text{in}(s'_1)(\alpha)\text{save}(s'_2)
\rightarrow (\text{in } s_2)\{ (\text{Agree}_{\text{post}} \land \text{Ext})
\]

23
\[ \phi_{m5}(R, R) \equiv \forall \text{Heap } h', h_2, h_2' \forall \alpha \forall x', x^2, (x^2)', y', y^2, (y^2)'( \]

\[ (x = y \leftrightarrow x' = y' \land (m_5())(x^2 = x \land y^2 = y) \land \{ \text{this } \omega x := x', y := y' \}(m_5())((x^2)' = x \land (y^2)' = y)) \]

\[ \rightarrow (x^2 = y^2 \leftrightarrow (x^2)' = (y^2)' \land x = x^2 \rightarrow x' = (x^2)' \land y = x^2 \rightarrow y' = (y^2)' \land x = y^2 \rightarrow x' = (y^2)' \land y = y^2 \rightarrow y' = (y^2)' \} \]

\[ \]

**Figure 5.** Formula \( \phi_{m5}(R, R) \) for method \( m5() \) from Figure 4 and \( R = (x, y) \).

We may assume \( (s_1, \beta) \models \text{Agree}_{\text{pre}} \land (\alpha)\text{save}\{s_2\} \land \text{in}\{s'_1\}(\alpha)\text{save}\{s'_2\} \) since otherwise the implication is trivially true.

Let \( s'_1 \) be the state that differs from \( s_1 \) by \( s'_1(v) = \beta(v') \) or all variables \( v \) in the universal quantifier prefix of \( \phi_{\alpha, R_1, R_2} \). It is easy to see that \( (s_1, \beta) \models \text{Agree}_{\text{pre}} \) implies \( \text{agree}(R_1, s_1, s'_1, \pi^1) \). Now, \( (s_1, \beta) \models (\alpha)\text{save}\{s_2\} \) implies in particular that \( \alpha \) started in \( s_1 \) terminates. Let us call the final state \( s_2 \). Likewise, \( (s_1, \beta) \models \text{in}\{s'_1\}(\alpha)\text{save}\{s'_2\} \) implies first \( (s'_1, \beta) \models (\alpha)\text{save}\{s'_2\} \) and then that \( \alpha \) started in \( s'_1 \) terminates. Let us call this final state \( s'_2 \). We are now in a position to make use of our assumption \( \text{flow}(s_1, \alpha, R_1, R_2) \) and conclude \( \text{agree}(R_2, s_2, s'_2, \pi^2) \) and \( \pi^2 \) is compatible with \( \pi^1 \). Except termination we obtain from \( (s_1, \beta) \models (\alpha)\text{save}\{s_2\} \) also \( \beta(h_2) = s_2(\text{heap}), \beta(v) = s_2(\text{result}) \), and \( \beta(v') = s'_2(v) \) for all other relevant program variables. From \( (s'_1, \beta) \models (\alpha)\text{save}\{s'_2\} \) we obtain likewise \( \beta(h'_2) = s'_2(\text{heap}), \beta(v') = s'_2(\text{result}) \), and \( \beta((v^2)' ) = s'_2(v) \) for all other relevant program variables. From \( \text{agree}(R_2, s_2, s'_2, \pi^2) \) we thus can conclude

\[ (s_1, \beta) \models \{\text{in } s_2\}\{\text{in } s'_2\}\text{Agree}_{\text{post}} \]

and from the fact that \( \pi^2 \) is compatible with \( \pi^1 \) we get

\[ (s_1, \beta) \models \{\text{in } s_2\}\{\text{in } s'_2\}\text{Ext}. \]

In total we have shown \( s_1 \models \phi_{\alpha, R_1, R_2} \), as desired. \( \square \)

**Example 8.** To illustrate the construction used in the proof of Theorem 1 by an example. We reconsider method \( m5() \) from Figure 4 on page 10 and \( R = (x, y) \), which is shorthand for \( \text{seqConcat}(\text{seqSingleton(selectC(\text{heap, null, x}))}, \text{seqSingleton(selectC(\text{heap, null, y}))}) \).

Note that, we have \((R, \text{length})^s = 2\) for all states \( s \) and the exact type of both fields \( x, y \) is always \( C \). Thus \( \text{Agree}_{\text{pre}} \) equates \( x = y \leftrightarrow x' = y' \). \( \text{Agree}_{\text{post}} \) equates \( x^2 = y^2 \leftrightarrow (x^2)' = (y^2)' \) and \( \text{boolz} \) is the conjunction \( x = x^2 \rightarrow x' = (x^2)' \land y = x^2 \rightarrow y' = (y^2)' \land x = y^2 \rightarrow x' = (y^2)' \land y = y^2 \rightarrow y' = (y^2)' \). Figure 5 shows the complete formula \( \phi_{m5}(R, R) \).

Another concept we need is **modifies sets**, which are reference set expressions describing which variables and locations a program modifies (at most).

**Definition 17 (Modifies set).** Let \( \alpha \) be a program and \( M = (V, L) \) a reference set expression.

We say that \( M \) is a modifies set for \( \alpha \), denoted by \( \text{mod}(\alpha, M) \), iff for all states \( s \) the following holds: if there is a state \( s' \) such that \( \alpha \) started in \( s \) terminates in \( s' \), then (a) for all locations \((o, f) \notin L^s \) we obtain \( f^s(o) = f^{s'}(o) \) and (b) for all variables \( \varphi \notin V \) we obtain \( \varphi^s = \varphi^{s'} \).

24
5.2 A Simplified Version

Lemma 8. If agree($R, s, s', \pi$) and $\rho$ is an automorphism on $\mathcal{D}$ then also agree($R, s, \rho(s'), \rho \circ \pi$).

Proof. From the assumption agree($R, s, s', \pi$) we get by definition:
1. $R^a = \langle a_0, \ldots, a_{n-1} \rangle$, $R^a(s') = \langle a'_0, \ldots, a'_{n-1} \rangle$,
2. for all $0 \leq i < n : \text{type}(a_i) = \text{type}(a'_i)$,
3. for all $0 \leq i < n$ such that $\text{type}(a_i) \not\subseteq \text{Object}$ : $a_i = a'_i$,
4. for all $0 \leq i < n$ such that $\text{type}(a_i) \subseteq \text{Object}$ : $a_i = \text{null} \Leftrightarrow a'_i = \text{null}$,
5. for all $0 \leq i < n$ such that $\text{type}(a_i) \subseteq \text{Object}$ and $a_i$ is an object of array type : $\text{type}(a_i).\text{length}^a = \text{type}(a'_i).\text{length}^{a'}$,
6. for all $0 \leq i < j < n$ such that $\text{type}(a_i) \subseteq \text{Object}$ and $\text{type}(a_j) \subseteq \text{Object}$ : $a_i \equiv a_j$,
7. $\pi(a_i) = a'_i$.

By the basic properties of isomorphism, see Lemma 1, we obtain using notation from Definition 11:
1. $R^{s'} = \langle a'_0, \ldots, a'_{n-1} \rangle$, $R^{\rho(s')} = \langle \rho(a'_0), \ldots, \rho(a'_{n-1}) \rangle$,
2. for all $0 \leq i < n : \text{type}(a'_i) = \text{type}(\rho(a'_i))$,
3. for all $0 \leq i < n$ such that $\text{type}(a'_i) \not\subseteq \text{Object}$ : $\rho(a'_i) = a'_i$ since isomorphisms are the identity outside Object$^\mathcal{D}$,
4. for all $0 \leq i < n$ such that $\text{type}(a'_i) \subseteq \text{Object}$ : $a'_i = \text{null} \Leftrightarrow \rho(a'_i) = \text{null}$,
5. for all $0 \leq i < n$ such that $\text{type}(a'_i) \subseteq \text{Object}$ and $a_i$ is an object of array type : $\text{type}(a'_i).\text{length}^{a'} = \rho(\text{type}(a'_i)).\text{length}^{\rho(s')}$,
6. for all $0 \leq i < j < n$ such that $\text{type}(a'_i) \subseteq \text{Object}$ and $\text{type}(a'_j) \subseteq \text{Object}$ : $a'_i \equiv \rho(a'_i) = \rho(a'_j)$,
7. $\rho \circ \pi(a_i) = \rho(a'_i)$.

This is, precisely, the definition of agree($R, s, \rho(s'), \rho \circ \pi$). $\square$

The information flow property in Definition 14 follows a pattern widely accepted in the research community, which in a nutshell can be phrased as: If program $\alpha$ is run in two states that agree on the low values then the states that are reached by executing $\alpha$ also agree on the the low values. Agreement for low values of non-object type means equality. The novelty in Definition 14 is that when low values of object type are involved we replace the requirement of equality by the relaxed requirement of the existence of a partial isomorphism. But, maybe we have gone too far. What would be lost if we insist that the bijection between objects in the prestates is the identity and only the bijection in the poststates may be arbitrary? To investigate this question rigorously we first introduce the following variation of Definition 14.

Definition 18 (Simple Information flow of a program).
Let $\alpha$ be a program and $R_1$ and $R_2$ be two observation expressions (of type Seq).
We say that $\alpha$ allows simple information flow only from $R_1$ to $R_2$ when started in $s_1$, denoted by flow$^\alpha(s_1, \alpha, R_1, R_2)$, iff, for all states $s'_1, s_2, s'_2$ such that $\alpha$ started in $s_1$ terminates in $s_2$ and $\alpha$ started in $s'_1$ terminates in $s'_2$, we have

\[
\text{if agree}(R_1, s_1, s'_1, \text{id}) \quad \text{then agree}(R_2, s_2, s'_2, \pi^2) \quad \text{and} \\
\pi^2(\alpha) \equiv \alpha \quad \text{for all } \alpha \in \text{obj}^{s_2}(R_2) \cap \text{obj}^{s_1}(R_1) \text{ with created}^{s_1}(\alpha) \equiv \text{id}.
\]

Note, that agree($R_1, s_1, s'_1, \text{id}$) implies in particular $\text{obj}^{s_1}(R_1) = \text{obj}^{s'_1}(R_1)$ since $\pi^1 = \text{id}$ is a bijection from $\text{obj}^{s_1}(R_1)$ onto $\text{obj}^{s'_1}(R_1)$.
Lemma 9. For all programs \( \alpha \), any two observation expressions \( R_1 \) and \( R_2 \), and any state \( s_1 \)

\[ \text{flow}^*(s_1, \alpha, R_1, R_2) \implies \text{flow}(s_1, \alpha, R_1, R_2) \]

Since the reverse implication is obviously true Lemma 9 entails that \( \text{flow} \) and \( \text{flow}^* \) are equivalent.

Proof. To prove \( \text{flow}(s_1, \alpha, R_1, R_2) \) we fix, in addition to \( s_1 \), states \( s_1', s_2, s_2' \) such that \( \alpha \) started in \( s_1 \) terminates in \( s_2 \) and \( \alpha \) started in \( s_1' \) terminates in \( s_2' \), and assume \( \text{agree}(R_1, s_1, s_1', \pi^1) \). We need to show \( \text{agree}(R_2, s_2, s_2', \pi^3) \) with \( \pi^3 \) extending \( \pi^1 \).

By Lemma 4 there is an automorphism \( \rho \) on \( D \) extending \( (\pi^1)^{-1} \).

From \( \text{agree}(R_1, s_1, s_1', \pi^1) \) we conclude \( \text{agree}(R_1, s_1, \rho(s_1'), \rho \circ \pi^1) \) by Lemma 8. Since \( \rho \) extends \( (\pi^1)^{-1} \) we have \( \text{agree}(R_1, s_1, \rho(s_1'), \text{id}) \). As noted in Lemma 5 there is a state \( s_3' \) such that \( \alpha \) started in \( \rho(s_1') \) terminates in \( s_3' \). This enables us to make use of the assumption \( \text{flow}^*(s_1, \alpha, R_1, R_2) \) and conclude \( \text{agree}(R_2, s_2, s_3', \pi^3) \).

Furthermore \( \pi^3(o) = o \) for all \( o \in \text{obj}^{R_1}(R_1) \cap \text{obj}^{R_2}(R_2) \).

Applying Lemma 5 to the inverse isomorphism \( \rho^{-1} \) to the situation that \( \alpha \) started in \( \rho(s_1') \) terminates in \( s_3' \), we obtain an automorphism \( \rho' \) such that \( \alpha \) started in \( \rho^{-1}(\rho(s_1')) = s_1' \) terminates in \( \rho'(s_3') \) and \( \rho' \) coincides with \( \rho^{-1} \) on all objects in \( E = \{ o \in \text{Object}^D \mid \text{created}^\rho(s_1')(o) = \text{tt} \} \).

Again using Lemma 8, this time for the isomorphism \( \rho' \), we obtain from \( \text{agree}(R_2, s_2, s_3', \pi^3) \) also \( \text{agree}(R_2, s_2, \rho'(s_3'), \rho' \circ \pi^3) \). Since \( \alpha \) is a deterministic program and we have already defined \( s_2' \) to be the final state of \( \alpha \) when started in \( s_2 \) we get \( s_2' = \rho'(s_3') \) and thus \( \text{agree}(R_2, s_2, s_2', \rho' \circ \pi^3) \).

It remains to convince ourselves that \( \rho' \circ \pi^3 = \pi^2 \) and that \( \rho' \circ \pi^3 \) extends \( \pi^1 \), i.e., for every \( o \in \text{obj}^{R_1}(R_1) \cap \text{obj}^{R_2}(R_2) \) with \( \text{created}^{\rho'}(o) = \text{tt} \) we need to show \( \rho' \circ \pi^3(o) = \pi^1(o) \).

By the definition of isomorphic states we obtain from \( \text{created}^{\rho'}(o) = \text{tt} \) also \( \text{created}^{\rho'(s_1')}(o) = \text{tt} \). Thus we can infer \( \rho'(o) = \rho^{-1}(o) \) and by choice of \( \rho \) further \( \rho^{-1}(o) = \pi^1(o) \), as desired.

The proof of the equality \( \rho' \circ \pi^3 = \pi^2 \) is still open. By Definition 13 we have \( \pi^2(R^{R_2}[i]) = R^{R_2}[i] \) for all \( i \) such that \( 0 < i < R^{R_2}.\text{length} = R^{R_2}.\text{length} \). On the other hand \( \pi^3 \) is defined by \( \pi^3(R^{R_2}[i]) = R^{R_2}[i] \) for all \( i \) such that \( 0 < i < R^{R_2}.\text{length} = R^{R_2}.\text{length} \). Thus \( \rho' \circ \pi^3(R^{R_2}[i]) = \rho'(R^{R_2}[i]) = R^{R_2}[i] \).

Since, as noted above, \( \rho'(s_3') = s_2' \) we have arrived at \( \rho' \circ \pi^3(R^{R_2}[i]) = R^{R_2}[i] \).

Lemma 9 leads to the following corollary to Theorem 1.

**Corollary 1.** Let \( \alpha \) be a program, and let \( R_1, R_2 \) be observation expressions.

There is a formula \( \phi_{\alpha,R_1,R_2} \) in JAVADL making use of self-composition such that: \( s_1 \models \phi_{\alpha,R_1,R_2} \) iff \( \text{flow}(s_1, \alpha, R_1, R_2) \)

\[ \phi_{\alpha,R_1,R_2} \equiv \forall \text{Heap} \, h_1', h_2, h_2' \forall \text{To} \, t' \forall \text{TT} \, t, t' \forall v' \forall v^2 \forall v^2 \ldots \langle \text{Agree}_\text{pre} \land \langle \alpha \rangle \text{save}(s_2) \land \in\{s'_1\} \langle \alpha \rangle \text{save}(s'_2) \rangle \rightarrow (\text{Agree}_\text{post} \land \text{Ext}) \]

Agree\_post and Ext remain as in the proof of Theorem 1 but Agree\_pre simplifies to

\[ \text{Agree}_\text{pre} \equiv R_1.\text{length} \equiv R'_1.\text{length} \]

\[ \forall i(0 < i < R_1.\text{length} \rightarrow R_1[i] = R'_1[i]) \]

**Proof.** Immediate from Theorem 1 and Lemma 9.
5.3 Subsumption

We come back to the notion of subsumption defined in Definition 15.

In many cases subsumption may be established immediately by observing that any expression in $R_2$ also occurs literally in $R_1$.

Lemma 10. We assume that observations $R_1$, $R_2$ are represented as sequences (Definition 16).

If

$$
\text{seqLen}(R_1) \geq \text{seqLen}(R_2) \land
\forall i (0 \leq i \land i < \text{seqLen}(R_2) \rightarrow \text{seqGet}(R_2, i) = \text{seqGet}(R_1, i))
$$

is universally valid then $R_1 \supseteq R_2$.

Proof

Obvious. ☐

Lemma 11. We assume again that $R_1$, $R_2$ are observations represented as sequences according to Definition 16.

Then $R_1 \supseteq R_2$ is equivalent to the validity of the formula

$$
\forall i (0 \leq i \land i < \text{seqLen}(R_1) \rightarrow R_1[i] = R'_1[i])
\rightarrow
\forall j (0 \leq j \land j < \text{seqLen}(R_2) \rightarrow R_2[j] = R'_2[j])
$$

The use of primed symbols is explained at the beginning of the proof of Theorem 1 on page 22.

Proof

Again obvious. ☐

Lemma 11 is of limited use in case $R_i$ are e.g., variables of type $\text{Seq}$. An interesting instantiation is given in the next simple lemma.

Lemma 12. Let $R_1 = \text{seq_def}\{u\}(t_1^1, t_2^1, e^1)$ and $R_2 = \text{seq_def}\{w\}(t_1^2, t_2^2, e^2)$

Then $R_1 \supseteq R_2$ is equivalent to the validity of the formula

$$
\forall u(t_1^1 \leq u \land u < t_2^1) \rightarrow e^1[u] = (e^1)'[u])
\rightarrow
\forall w(t_1^2 \leq u \land u < t_2^2) \rightarrow e^2[w] = (e^2)'[w])
$$

Proof Instance of Lemma 11 ☐

6 Modular Self-composition with Contracts

In the context of functional verification, modularity is achieved through method contracts: If it is proven that an implementation of a method $m$ adheres to its contract, then we can replace calls to $m$ in proofs by this contract without looking at the implementation code. We want to carry this approach over to the verification of information flow properties. In previous work [18], we have introduced information flow contracts: An information flow contract (in short: flow contract) $C_{m:T}$ for method $m$ declared in type $T$ is satisfied if in any state the formula $\text{flow}(\text{this.m}(\alpha), R_1, R_2)$ from Definition 14 is true, where program $\alpha$ has been instantiated to method $m$ and quantification over parameters and return value are included. In [18] flow contracts may include preconditions and declassifications. For the sake of readability we exclude those features in this
presentation. Including them is straightforward. From the example of the formula \( \phi_{\text{this.m}(\bar{a}), R_1, R_2} \) presented after Theorem 1 we can read off the structure of the formalisation of flow \((\text{this.m}(\bar{a}), R_1, R_2)\) in the general case:

\[
\psi_{\text{MC} - \bar{a}} \equiv \forall \text{Heap } h_1 \forall T o \forall \bar{a} \forall A_{n+1} r \{\text{in } s_1\} \phi_{\text{this.m}(\bar{a}), R_1, R_2}
\]

\[
\equiv \forall \text{Heap } h_1, h_1', h_2, h_2' \forall T o, o' \forall \bar{a}, \bar{a}' \forall A_{n+1} r, r'
\]

\[
\{\text{in } s_1\}\{\text{this.m}(\bar{a})\}(\text{save } s_2) \land \{\text{in } s_1'\}\{\text{this.m}(\bar{a})\}(\text{save } s_2')
\]

\[
\land \{\text{in } s_1\}\{\text{in } s_1'\}(ED^{R_1} \land EO^{R_1})
\]

\[
\rightarrow \{\text{in } s_2\}\{\text{in } s_2'\}(ED^{R_2} \land EO^{R_2} \land Old^{R_1, R_2})
\]

Here we use the following suggestive abbreviations: (1) The shorthand \( \{\text{in } s_1\} \psi \) signals that an update \( \{\text{heap} := h'_1 || \text{this} := o' || \ldots a_i := x'_i \ldots\} \) is placed before \( \varphi \). The \( a_i \) cover all other relevant parameters and local variables. (2) The construct (save \( s_2 \)) abbreviates a conjunction of equations \( h_2 = \text{heap}, r = \text{result}, \ldots, x_2 = a_i, \ldots \). Analogously, (save \( s_2' \)) stands for the primed version \( h'_2 = \text{heap}, r' = \text{result}, \ldots, x'_2 = a_i, \ldots \). (3) The shorthand \( ED^{R_1} \land EO^{R_1} \) abbreviates a formula which is valid iff \( s_1 \) and \( s_2 \) agree on \( R_1 \) in the sense of Definition 13. Analogously, \( ED^{R_2} \land EO^{R_2} \) abbreviates a formula which is valid iff \( s_2 \) and \( s_2' \) agree on \( R_2 \). (4) \( Old^{R_1, R_2} \) abbreviates a formula which guarantees that the isomorphism defined by \( ED^{R_2} \land EO^{R_2} \) is an extension of the one defined by \( ED^{R_1} \land EO^{R_1} \).

The difficulty in the application of method contracts for information flow arises from the fact that \( \psi_{\text{MC} - \bar{a}} \) refers to two invocations of a method \( m \) in different contexts. Therefore a flow contract cannot be used directly if the first symbolic execution in a self-composition proof reaches a method invocation: the second execution might not yet have reached such an invocation. This is in particular a problem if the first program has to be executed completely before the execution of the second starts. The remainder of this section explains how flow contracts can be integrated into the calculus in order to achieve modular and feasible proofs.

The main idea of the integration is to delay the application of flow contracts.

If \( \psi_{\text{MC} - \bar{a}} \) has been proven valid for some method \( m \), then it can be used as a lemma in the proof of \( \psi_{\text{MC} - \bar{a}} \) for another method \( m_2 \). We extend the standard functional method contract rule by adding the predicate \( MC_{T:m}(o, \bar{a}, h_1, res, h_2) \) to the antecedent of each premise. The predicate intuitively states that the method contract rule for \( m \) applied on the object \( o \) with parameters \( \bar{a} \) in state \( h_1 \) results in state \( h_2 \) and result value \( res \). The reason to introduce the predicate and not the equivalent formula \( \{\text{in } s_1\}\{\text{this.m}(\bar{a})\}\{\text{save } s_2\} \) is that \( MC_{T:m} \) is not decomposed by the proof search strategy. We introduce the following rule schema to make use of \( \psi_{\text{MC} - \bar{a}} \) as a lemma:

**FlowContract**

\[
MC_{T:m}(o, \bar{a}, h_1, res, h_2), MC_{T:m}(o', \bar{a}', h'_1, res', h'_2),
\]

\[
\{\text{in } s_1\}\{\text{in } s_1'\}(ED^1 \land EO^1) \rightarrow \{\text{in } s_2\}\{\text{in } s_2'\}(ED^2 \land EO^2 \land Old)
\]

\[
MC_{T:m}(o, \bar{a}, h_1, res, h_2) \land MC_{T:m}(o', \bar{a}', h'_1, res', h'_2) \Rightarrow
\]

The rule matches two instances of \( MC_{T:m} \) and introduces an implication to the antecedent: the implication resulting from \( \psi_{\text{MC} - \bar{a}} \) through instantiation of the quantifiers with the heaps and actual parameters of the two instances of \( MC_{T:m} \). The condition \( \{\text{in } s_1\}\{\text{this.m}(\bar{a})\}\{\text{save } s_2\} \) of \( \psi_{\text{MC} - \bar{a}} \) and its primed counterpart are valid by construction since \( MC_{T:m}(o, \bar{a}, h_1, res, h_2) \) and its primed counterpart hold. Intuitively the rule is sound, because it is a combination of two intuitively obviously sound rules: first \( \psi_{\text{MC} - \bar{a}} \) is introduced as an axiom to the sequent and afterwards the quantifiers of \( \psi_{\text{MC} - \bar{a}} \) are instantiated in such a way that the condition \( \{\text{in } s_1\}\{\text{this.m}(\bar{a})\}\{\text{save } s_2\} \) and its primed counterpart are valid by construction.

28
7 Related Work

Techniques for Enforcing Secure Information Flow. The most widely used approach to secure information flow is type systems as introduced by Volpano and Smith [21]. This was done for a small while language. Later contributions extended this approach to sequential Java [3,16,20]. Hunt and Sands introduce floating types [12] that may change throughout a program execution. In this approach, the security levels are not assigned a-priori. Instead, through a Hoare-style calculus, the program gives rise to a mapping from variables to sets of variables on which they depend at most.

In [9], dynamic logic is used to encode the Hunt/Sands type system. This approach is similar to ours in that it combines an abstract view of programs (type system) with the power of a theorem prover. However, information-flow policies are still imposed through typing (as opposed to a proof obligation in dynamic logic).

Another approach extracts a dependence graph from programs, which is in turn analysed for graph-theoretical reachability properties. This has been done for a significant subset of Java [10]. However this technique suffers from a similar precision issue as type systems.

Self-composition has been proposed [5,7] as a technique to introduce non-interference properties into program logics. While it avoids false-positives, this technique suffers – as we have explained above – from a lack of scalability. One way to improve this method is to replace sequential composition of two programs by a single product program that partially parallelizes the two executions [4].

Information Flow in Object-oriented Languages. Most approaches to secure information flow either apply only to a simple while language without taking object-orientation into account or implicitly assign the lowest security level to object references. One of the first works to mention the restrictions w.r.t. object-orientation of static methods like type systems is [1]. There, the authors propose region logic, a kind of Hoare logic with concepts from separation logic in order to deal with aliasing of object references.

Hansen et al. [11] were the first to relax the definition of low-equivalence in non-interference for object identity. In their formalization, two heaps are low-equivalent up to a partial isomorphism (similar to our Def. 13).

8 Conclusions and Future Work

We have introduced an approach to verify Java programs w.r.t. information-flow properties in a compositional manner. We have defined a notion of low-equivalence between heaps modulo isomorphism. Although we have introduced a new modality to reason about information flow on a higher level of abstraction, the flow modality can be expressed in dynamic logic.

Proof obligations for non-interference using self-composition have already been implemented in the KeY tool. We have recently added a prototype implementation of the flow operator.

A first extension of the work presented here will be to take termination into consideration. Also, while throughout this paper, we have always defined secrecy in terms of a two-element security lattice, the approach will be extended to work with any lattice.

The concept of declassification can be easily added to the flow modality and the calculus. This can be done by adding a formula as an extra parameter to flow that describes what the attacker is allowed to learn (i.e., what flow is permissible).
We also plan to investigate whether it is useful to add the set of objects that an attacker knows as an explicit parameter to the flow modality, so as to avoid the problem discussed in Example 7 and simplify the flowSplit rule. And one may add a parameter restricting over what values the high locations range.

To further explore the applicability of our approach beyond simple textbook examples, we are currently applying it to an e-voting case study.

A Appendix: Finite Sequences

The goal of this appendix is to present the data type Seq. More precisely, we will run through the file seq.key that contains the axioms (taclets) for Seq providing arguments for their consistency. At the time of this writing seq.key was not yet on the main branch of the KeY system.

| Core axioms
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Extension by Definitions resulting in the theory corePIX</td>
</tr>
</tbody>
</table>

| Derived Taclets
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Extension by Definitions introducing two kinds of permutations</td>
</tr>
</tbody>
</table>

A.1 The Core Theory seqCore

The core consists of four axioms altogether using the following function symbols:

\[
\text{any any::seqGet(Seq, int)}
\]

\[
\text{any seqGetOutside}
\]

\[
\text{int seqLen(Seq)}
\]

and the generalized quantifier \( \text{seq\_def}\{\},() \). Figure A.1 shows the axioms in mathematical notation in a typed first-order logic. Variables \( s, s_1, s_2 \) are of type Seq, variables \( i, j, k, r, l, e \) are of type int, variable \( a \) is of type any. Furthermore \( \phi\{t/u}\) denotes the formula obtained from \( \phi \) by replacing all free occurences of the variable \( u \) by the term \( t \). The taclets version of the seqCore are reproduced in lines 41 – 89 in the listing in Subsection A.7.

We use \( s[i] \) as a short-hand for \( \text{any::seqGet}(s, i) \).

A finite sequence \( s \) is represented as a function \( i \leadsto s[i] \) from int into any plus a length \( \text{seqLen}(s) \). Axiom 1 says that the length of a sequence is a positive integer, in particular this says that it is finite. Axioms 2 characterizes equality.
of finite sequences. Thus, the values \( s[i] \) for \( i < 0 \) or \( \text{seqLen}(s) \leq i \) are irrelevant in this respect. In particular, there is at most one empty sequence.

The main difference of our axiomatization of \( \text{Seq} \) over the traditional abstract datatype approach is the use of the generalized quantifier \( \text{seq\_def}(u)(l,e,r,t) \) with the intended meaning formalized in axioms 3 and 4: it defines a sequence of length \( ri - le \) whose entry at position \( i \) is obtained by evaluating the expression \( t \) with the variable \( u \) replaced by \( i \). If \( ri \leq le \) the empty sequence is defined.

### A.2 Consistency of the Core Theory

To prove consistency of the theory \( \text{seqCore} \) we will construct a non-empty set \( \text{MSeq} \) of models \( M \) such that \( M \models \phi \) for every axiom \( \phi \) in the list of Figure A.1. Of course, one model \( M \) with this property would be enough to ascertain consistency, but it just so turns out that there is a natural class of them. Furthermore, it raises the interesting question whether \( \text{seqCore} \) is complete with respect to the class of structures \( \text{MSeq} \), i.e., for every formula \( \psi \) with \( M \models \psi \) for every \( M \in \text{MSeq} \) we ask if \( \text{seqCore} \models \psi ? \).

We turn to the construction of the models \( M \) in \( \text{MSeq} \). Let \( D \) be a structure satisfying the stipulations from Definition 3. The universe of \( M \) will depend on the choice of \( D \). To avoid unwieldy notation we will not show \( D \) as an explicit parameter. We will remember the dependance on \( D \) when needed.

**Definition 19 (The type domain \( \text{Seq}^D \)).** The type domain \( \text{Seq}^D \) is defined via the following induction.

- \( U^D = \text{Boolean}^D \cup \text{Int}^D \cup \text{Object}^D \cup \text{LocSet}^D \)
- \( D^0_{\text{Seq}} = \{ \emptyset \} \)
- \( D^{n+1}_{\text{Seq}} = \{ \langle a_1, \ldots, a_k \rangle \mid k \in \mathbb{N} \text{ and } a_i \in D^n_{\text{Seq}} \cup U^D, 1 \leq i \leq k \}, \quad n \geq 0 \)

\[
\text{Seq}^D := D_{\text{Seq}} := \bigcup_{n \geq 1} D^n_{\text{Seq}}
\]

In this definition we use the notion of a finite sequence \( \langle a_0, \ldots, a_{n-1} \rangle \) as a primitive concept. Those that want a more foundational approach may think of a finite sequences as equivalence classes of functions from \( \mathbb{Z} \) into values, or as sets of pairs \( \{ (i, a) \mid 0 \leq i < n \text{ and } a \text{ a value} \} \).

We point out that the definition of \( D_{\text{Seq}} \) is very liberal we allow unrestricted nesting, i.e. there can be sequences of sequences of sequences etc. and the entries in a sequence need not be of the same type. Thus \( \{ 0, \emptyset, \text{seqEmpty}, \text{null}, \text{tt} \} \) is a perfect element in \( D_{\text{Seq}} \).
Definition 20 (MSeq). A structure $\mathcal{M}$ with universe $D_{Seq} \cup D_{Seq}^0$ belongs to the set MSeq if it satisfies the following restrictions, where \( i, ri, le \) are integers, \( a_k, a \) elements of $D_{Seq}^0$ and \( s \in S_{Seq} \) and \( e \) a term of type Any:

1. $\text{seq}_\text{def}\{iv\} (le, ri, e)^{M, \beta} = \begin{cases} \{a_0, \ldots, a_{k-1}\} & \text{if } ri - le = k > 0 \\ \text{seqGetOutside}^{M} & \text{otherwise} \end{cases}$

2. seqGet\(_\text{any}\)(\(\langle a_0, \ldots, a_{n-1}\rangle, i\)) = \begin{cases} a_i & \text{if } 0 \leq i < n \\ \text{seqGetOutside}^{M} & \text{otherwise} \end{cases}

3. seqLen\(_M\)(\(\langle a_0, \ldots, a_{n-1}\rangle\)) = \( n \)

4. seqGetOutside\(_{M}\) \(\in D_{Seq}\) arbitrary.

As an example of item 1 we present $\text{seq}_\text{def}\{iv\}(-15, -10, 20 + iv)^{D} = (5, 6, 7, 8, 9)$.

Because there is no restriction on the interpretation of the constant seqGetOutside this definition defines not just one model but a whole class of them.

Note, in clause 1 that the meaning of seq\(_\text{def}\)\{iv\}\((t_1, t_2, e)\) is not determined by the structure $\mathcal{M}$ alone, the variable assignment $\beta$ needs to be taken into account.

Lemma 13. For every structure $\mathcal{M}$ in MSeq we have

\[ \mathcal{M} \models \phi \]

for all axioms of seqCore (see Figure A.1).

Proofs $\mathcal{M} \models \phi$ for the first two axioms 1 and 2 are obvious properties of finite sequences. It uses the technical lemma that $t(\{le + i/u\})^{M, \beta}$ evaluates to the same value as $t^{M, \beta'}$ with

$\beta'(v) = \begin{cases} \beta(v) & \text{if } v \text{ is different from } u \\ \{le + i\}^{M, \beta} & \text{if } v \equiv u \end{cases}$

Axioms 3 and 4 follow directly from the definitions of seq\(_\text{def}\) in (1) and seqGet\(_\text{any}\) in (2) of Definition 20.

Definition 21 (seqCoreDepth). The theory seqCoreDepth is the extension of seqCore by adding a new function symbol

Int seqDepth(Any)

and the axioms

5. $\forall x (\neg \text{instance}_\text{Seq}(x) \rightarrow \text{seqDepth}(x) \equiv 0)$
6. $\forall s (\text{seqDepth}(s) \equiv \max\{\text{seqDepth}(s[i]) \mid 0 \leq i < \text{seqLen}(s)\} + 1)$

Here $x$ is a variable of type Any and $s$ a variable of type Seq.

First we need to convince ourselves that the extended theory seqCoreDepth is consistent. To this end we extend definition 13.

Definition 22 (MSeqD). The set MSeqD consists exactly of those structures $\mathcal{M}_D$ that arise from $\mathcal{M}$ in MSeq by defining the new function symbol seqDepth by

$\text{seqDepth}^{M_D}(a) = \text{the unique } n \text{ with } a \in D_{Seq}^0$

for all $a$ in the universe of $\mathcal{M}$.

Lemma 14. The theory seqCoreDepth is consistent.
Proof For every structure \( \mathcal{N} \) in \textbf{MSeqD} it is easily checked that \( \mathcal{N} \models \phi \) for the two axioms (5) and (6) from Definition 21.

**Lemma 15 (Relative Completeness).** Assume that \( \mathcal{D} \) only consists of the type \textit{Int} with its usual functions and predicates and there is a theory \( T_{\text{int}} \) such that for any model \( \mathcal{D} \) of \( T_{\text{int}} \) we have \( D^0_{\text{Seq}} = \mathbb{Z} \).

The theory \( T_{\text{int}} \cup \text{seqCoreDepth} \) is complete with respect to \textbf{MSeqD}.

In detail this means:

Let \( \phi \) be a formula in the signature of \text{seqCoreDepth} such that \( \mathcal{M} \models \phi \) for all \( \mathcal{M} \) in \textbf{MSeqD} then

\[
T_{\text{int}} \cup \text{seqCoreDepth} \vdash \phi
\]

**Proof** The proof proceeds by contradiction. We assume \( \mathcal{N} \models \phi \) for all \( \mathcal{N} \) in \textbf{MSeqD}, but \( T_{\text{int}} \cup \text{seqCoreDepth} \not\models \phi \). Thus there is a structure \( \mathcal{N}_0 \) with \( \mathcal{N}_0 \models T_{\text{int}} \cup \text{seqCoreDepth} \) but \( \mathcal{N}_0 \models \neg \phi \). We define a mapping \( F : \mathcal{N}_0 \rightarrow D_{\text{seq}} \cup \mathbb{Z} \), i.e., from the universe \( \mathcal{N}_0 \) of \( \mathcal{N}_0 \) into the common universe of all structures in \textbf{MSeqD}. \( F(a) \) is defined by induction on \text{seqDepth}(a). By assumption \( \{ a \in \mathcal{N}_0 \mid \text{seqDepth}^{\mathcal{N}_0}(a) = 0 \} = \text{Int}^{\mathcal{N}_0} = \mathbb{Z} \) and we let \( F \) be the identity on these elements. For \( a \) with \text{seqDepth}^{\mathcal{N}_0}(a) = n + 1 \ we define inductively

\[
F(a) = \langle F(a[0]), \ldots, F(a[k-1]) \rangle
\]

with \( k = \text{seqLen}^{\mathcal{N}_0}(a) \), \( a[i] \) again shorthand for any \( \text{seqGet}^{\mathcal{N}_0}(a, i) \). Since \( \mathcal{N}_0 \) satisfies axiom 6 from Definition 21 we know \text{seqDepth}^{\mathcal{N}_0}(a[i]) \leq n.

From axiom (2) in Definition A.1 we get immediately that \( F \) thus defined is an injective function. We want to argue that \( F \) is also surjective. We will exhibit for every \( a \in D^0_{\text{Seq}} \), by induction on \( n \), a term \( t \) such that \( F(t^{\mathcal{N}_0}) = a \). For \( n = 0 \), we know that \( a \) has to be an integer and \( F(a) = a \). We did not specifically fix the signature of \( T_{\text{int}} \), but we may fairly assume that there is a term \( t_n \) with \( F(t_n^{\mathcal{N}_0}) = n \), e.g. \( t_n = 1 + \ldots + 1 \), \( t_0 = 0 \) or \( t_n = -1 - \ldots -1 \). In the inductive step of the argument we assume that the claim is true for all \( a \in D^0_{\text{Seq}} \) and fix \( s = \langle s_0, \ldots s_{k-1} \rangle \in D^{n+1}_{\text{Seq}} \). Since \( s_i \in D^0_{\text{Seq}} \) for all \( 0 \leq i < k \) there are terms \( t_i \) with \( F(t_i^{\mathcal{N}_0}) = s_i \). Now, \( F(\langle \text{seq}_\text{def}\{u\}(0, k, t)\rangle^{\mathcal{N}_0}) = s \) with

\[
t = \text{if } u = 0 \text{ then } t_0 \text{ else } (\text{if } u = 1 \text{ then } t_1 \text{ else } \ldots (\text{if } u = k - 1 \text{ then } t_{k-1}) \ldots)
\]

Since \( \mathcal{N}_0 \) satisfies axiom 3 from Definition A.1 we have \( \mathcal{N}_0 \models t[i] = t_i \) and thus by induction hypothesis and definition of \( F \)

\[
F(t^{\mathcal{N}_0}) = \langle F(t_0^{\mathcal{N}_0}), \ldots, F(t_{k-1}^{\mathcal{N}_0}) \rangle = \langle s_0, \ldots s_{k-1} \rangle = s
\]

In total he have verified

\[
F : N \rightarrow M \text{ is a bijection (6)}
\]

We define a structure \( \mathcal{M} \) with universe \( D^0_{\text{Seq}} \cup D_{\text{Seq}} \) by isomorphic transfer via \( F \), i.e.,

\[
\text{seq}_\text{def}\{u\}(i, j, e)^{\mathcal{M}, F(\beta)} = F(\text{seq}_\text{def}\{u\}(i, j, e)^{\mathcal{N}_0, \beta})
\]

\[
\text{seqGet}^{\mathcal{M}}(F(s), i) = F(\text{seqGet}^{\mathcal{N}_0}(s, i))
\]

\[
\text{seqGetOutside}^{\mathcal{M}} = F(\text{seqGetOutside}^{\mathcal{N}_0})
\]

\[
\text{seqLen}^{\mathcal{M}}(F(s)) = F(\text{seqLen}^{\mathcal{N}_0}(s))
\]

\[
\text{seqDepth}^{\mathcal{M}}(F(a)) = F(\text{seqDepth}^{\mathcal{N}_0}(a))
\]

33
By construction \( F \) is an isomorphism from \( N_0 \) onto \( M \). Thus \( N_0 \models \neg \phi \) implies \( M \models \neg \phi \).

The proof plan is to show that \( M \) is in \textit{MSeqD}. This will contradict the assumption that \( \phi \) be true in all structures in \textit{MSeqD}.

We will make use of the following two fundamental properties of \( F \), in fact of any isomorphism

For all terms \( e \) and variable assignments \( \beta \)

\[
F(e^{N_0, \beta}) = e^{M, F(\beta)}
\]

(7)

For any formula \( \phi \) and variable assignments \( \beta \)

\[
(N_0, \beta) \models \phi \iff (M, F(\beta)) \models \phi
\]

(8)

Here \( F(\beta) \) stands for the variable assignment defined by \( F(\beta)(x) = F(\beta(x)) \). As an instance of (7) think of the term \( e = \text{seqLen}(x) \) that leads to the equality \( F(\text{seqLen}^N_0(s)) = \text{seqLen}^M(F(s)) \).

Both (7) and (8) are routinely proved by induction on the complexity of \( e \) and \( \phi \).

To show \( M \in \textit{MSeqD} \) we have to check Definitions 20 and 22 item by item.

1. \( \text{seq\_def}\{u\}(i, j, e)^M, F(\beta) = \text{seq\_def}\{u\}(i, j, e)^{N_0, \beta} \) iso transfer

\[
a_r = F(\text{seq\_def}\{u\}(i, j, e)^{N_0, \beta})
\]

axiom 3, in Fig. A.1

\[
a_r = F(\text{seqGetOutside}^N_0)
\]

axiom 3, in Fig. A.1

\[
a_r = \text{seqGetOutside}^M
\]

iso transfer

2. \( \text{seqGet}^M_{any}(\langle a_0, \ldots, a_{n-1} \rangle, i) = \text{seqGet}^N_{any}(s, i) \) iso transfer

with \( F(s) = \langle a_0, \ldots, a_{n-1} \rangle \)

\[
\text{seqLen}^N_0 = n \text{ and } 0 \leq r
\]

\[
a_r = F(\text{seqGet}^N_{any}(s, r))
\]

\( r < n \)

For \( 0 \leq i < n \) this gives the desired result \( \text{seqGet}^M_{any}(\langle a_0, \ldots, a_{n-1} \rangle, i) = a_i \).

For \( i < 0 \) or \( n \leq i \) we argue that \( \forall s \forall i ((i < 0 \lor \text{seqLen}(s) \leq i) \rightarrow s[i] = \text{seqGetOutside}) \) is a logical consequence of \textit{seqCore} and thus true in \( N_0 \). In this case we get the following chain of reasoning

\[
\text{seqGet}^M_{any}(\langle a_0, \ldots, a_{n-1} \rangle, i) = F(\text{seqGet}^N_{any}(s, i)) \text{ iso transfer}
\]

\[
= F(\text{seqGetOutside}^N_0)
\]

\[
= \text{seqGetOutside}^M \text{ iso transfer}
\]

3. \( \text{seqLen}^M(\langle a_0, \ldots, a_{n-1} \rangle) = \text{seqLen}^N_0(s) \) iso transfer

with \( F(s) = \langle a_0, \ldots, a_{n-1} \rangle \)

Now, we get from the definition of \( F \) and \( F(s) = \langle a_0, \ldots, a_{n-1} \rangle \) immediately

\( \text{seqLen}^N_0(s) = n \).
4. Nothing to show here.
5. By isomorphic transfer we defined \( \text{seqDepth}^M(F(a)) = \text{seqDepth}^{N_0}(a) \). By definition of \( F \) we know \( F(a) \in D^{\text{seqDepth}^{N_0}(a)}_{\text{Seq}} \). Thus, the restriction on \( \text{seqDepth}^M \) in Definition 22 is satisfied.

\[ \square \]

We did not add the depth function and the accompanying axioms from Definition 21 to the core theory, since we anticipated that it will rarely be used in program verification. If that proves wrong we at least know what to do.

### A.3 First Extension by Definition

The following functions will be introduced by defining axioms.

\[
\begin{align*}
\alpha \alpha &::= \text{seqGet}(\text{Seq}, \text{Int}) \text{ seqEmpty} \\
\text{Seq seqSingleton}(\text{any}) &::= \text{seqConcat}(\text{Seq}, \text{Seq}) \\
\text{Seq seqSub}(\text{Seq}, \text{Int}, \text{Int}) &::= \text{seqReverse}(\text{Seq}) \\
\text{int seqIndexOf}(\text{Seq}, \text{any}) &::= \text{seqGet}(\text{Seq}, \text{Int})
\end{align*}
\]

The defining axioms are shown in Figure 8 in mathematical notation. The corresponding taclets may be found on lines 98 – 190 in the listing in Subsection A.7.

1. \( \forall s[i](\alpha :: \text{seqGet}(s, i) \equiv (\alpha)\text{any} :: \text{seqGet}(s, i)) \)
   or in shorthand
   \( \forall s[i](\alpha :: \text{seqGet}(s, i) \equiv (\alpha)s[i]) \)
2. \( \text{seqEmpty} \equiv \text{seq_def}\{u\}(0,0,1) \)
3. \( \forall x(\text{seqSingleton}(x) \equiv \text{seq_def}\{u\}(0,1,x)) \)
4. \( \forall s_1, s_2(\text{seqConcat}(s_1, s_2) \equiv \text{seq_def}\{u\}(0,\text{seqLen}(s_1) + \text{seqLen}(s_2), \text{if } u < \text{seqLen}(s_1) \text{ then } s_1[u] \text{ else } s_2[u - \text{seqLen}(s_1)])) \)
5. \( \forall s, r, e(\text{seqSub}(s, r, e) \equiv \text{seq_def}\{u\}(0, r, e, s[u])) \)
6. \( \forall s(\text{seqReverse}(s) \equiv \text{seq_def}\{u\}(0, \text{seqLen}(s), s[\text{seqLen}(s) - u])) \)
7. \( \forall s, a, j(0 \leq j \land j < \text{seqLen}(s) \land s[j] \equiv a \land \forall k(0 \leq k \land k < j \rightarrow s[k] \neq a)) \rightarrow \text{seqIndexOf}(s, a, j) \)

Variables \( s, s_1, s_2 \) are of type \( \text{Seq} \), variable \( x \) of type \( \text{any} \) and \( i, r, e, l \) are of type \( \text{Int} \).

**Figure 8. First Set of Extensions by Definition**

The family of function symbols \( \alpha \alpha :: \text{seqGet}(\text{Seq}, \text{Int}) \) defined in axiom 1 is necessary since the type system of the first-order language of the KeY system has deliberately been kept simple. In particular there are no parametrized types. All we know is that the entries \( s[i] \) of every sequence \( s \) are of type \( \text{any} \). If we know for sure that the entries in \( s \) are more specific, e.g., we know they are all integers, we can use the cast function, \( (\text{int})s[i] = (\text{int})\text{any} :: \text{seqGet}(s, i) \). For ease of use function symbols \( \alpha \alpha :: \text{seqGet}(\text{Seq}, \text{Int}) \) were added for every type \( \alpha \).

Sometimes it is useful to have a function \( \text{seqIndexOf} \) that is inverse to sequence access. More precisely, we want \( \text{seqIndexOf}(s, a) \) to be the least index \( i \) with \( s[i] = a \) if it exists and undefined otherwise. Definition 7 defines the partial function \( \text{seqIndexOf} \).

Let us call the new theory \( \text{seqCore1} \). At this point it is important to know, whether \( \text{seqCore1} \) is still consistent. An inconsistent theory is for our purposes
totally useless. We will show a bit more: the new theory is even a conservative extension of seqCore. We need some terminology first.

**Definition 23 (Conservative Extension).** Let \( \Sigma_0 \subseteq \Sigma_1 \) be signatures, and \( T_i \) set of sentences in \( Fml_{\Sigma_i} \).

\( T_1 \) is called a conservative extension of \( T_0 \) if for all sentences \( \phi \in Fml_{\Sigma_0} \):

\[ T_0 \vdash \phi \iff T_1 \vdash \phi \]

Note, if \( T_0 \) is consistent and \( T_1 \) is a conservative extension of \( T_0 \) then \( T_1 \) is also consistent.

Conservative extension is a well-known property in mathematical logic, see e.g., [15, pp. 208 – 210], [19, Section 4.1], [8, Kapitel VIII §1]

**Definition 24 (Semantic Conservative Extension).** Let \( \Sigma_0 \subseteq \Sigma_1 \) be signatures, and \( T_i \) sets of sentences in \( Fml_{\Sigma_i} \).

\( T_1 \) is called a semantic conservative extension of \( T_0 \) if

1. for all \( \Sigma_1 \)-structures \( M_1 \) with \( M_1 \models T_1 \) the restriction \( M_0 \) of \( M_1 \) to \( \Sigma_0 \) is a model of \( T_0 \), in symbols
   \[ M_1 \models T_1 \Rightarrow (M_1 \upharpoonright \Sigma_0) \models T_0 \]

2. for every \( \Sigma_0 \)-structure \( M_0 \) with \( M_0 \models T_0 \) there is a \( \Sigma_1 \)-expansion \( M_1 \) of \( M_0 \) with \( M_1 \models T_1 \).

Note, in case \( T_0 \subseteq T_1 \) is true, which is the most typical case, but not required in Definitions 23 and 24, then item 1 of the proceeding definition is automatically true.

**Lemma 16.** Let \( \Sigma_0 \subseteq \Sigma_1 \) be signatures, and \( T_i \) sets of sentences in \( Fml_{\Sigma_i} \).

If \( T_1 \) is a semantic conservative extension of \( T_0 \) then \( T_1 \) is also a conservative extension of \( T_0 \).

**Proof** Let \( \phi \) be a sentence in \( Fml_{\Sigma_0} \) with \( T_0 \vdash \phi \). Let \( M_1 \) be an arbitrary \( \Sigma_1 \)-structure. By assumption \( (M_1 \upharpoonright \Sigma_0) \models T_0 \). Thus we also have \( (M_1 \upharpoonright \Sigma_0) \models \phi \).

By the coincidence lemma we also have \( M_1 \models \phi \). In total we have shown \( T_1 \vdash \phi \).

Now, assume \( T_1 \vdash \phi \). If \( M_0 \) is an arbitrary \( \Sigma_0 \)-structure there is by the assumption an expansion of \( M_0 \) to a \( \Sigma_1 \)-structure \( M_1 \). From \( T_1 \vdash \phi \) we thus get \( M_1 \models \phi \). The coincidence lemma tells us again that also \( M_0 \vdash \phi \). In total we arrive at \( T_0 \vdash \phi \). \( \Box \)

**Lemma 17 (Extension by Definition).** Let \( \Sigma_0 \subseteq \Sigma_1 \) be signatures, \( T_0 \subseteq T_1 \) sets of sentences in \( Fml_{\Sigma_0} \) respectively in \( Fml_{\Sigma_1} \). Further assume that all sentences in \( T_1 \setminus T_0 \) are of the form

\[ \forall \overline{x} (f(\overline{x}) \equiv t) \quad f \in \Sigma_1 \subseteq \Sigma_0 \ t \ a \ term \ in \ \Sigma_0 \]

\[ \forall \overline{x} (p(\overline{x}) \leftrightarrow \phi(\overline{x})) \ p \in \Sigma_1 \subseteq \Sigma_0 \ \phi \ a \ formula \ in \ \Sigma_0 \]

Then \( T_1 \) is a semantic conservative extension of \( T_0 \).

**Proof** If \( M_0 \) is a \( \Sigma_0 \)-model of \( T_0 \) we obtain an \( \Sigma_1 \)-expansion \( M_1 \) by simply setting

\[ f^{M_1}(\overline{a}) = t^{M_0}(\overline{a}) \]

and

\[ p^{M_1}(\overline{a}) \iff M_0 \models \phi[\overline{a}] \]  

\( \Box \)

In the situation of Lemma 17 \( T_1 \) is called an extension by definitions of \( T_0 \).

We tacitly assume – of course – that \( T_1 \) contains only one definition for each new function or relation symbol.
Lemma 18 (Unique Conditional Extension by Definition). Let $\Sigma_0 \subseteq \Sigma_1$ be signatures, $T_0 \subseteq T_1$ sets of sentences in $\text{Fml}_{\Sigma_0}$ respectively in $\text{Fml}_{\Sigma_1}$. Further assume that all sentences in $T_1 \setminus T_0$ are of the form

$$\forall \bar{x} \forall y (\psi \rightarrow f(\bar{x}) \equiv y)$$

such that

$$T_0 \vdash \forall \bar{x} \forall y, y'(\psi \land \psi[y'/y] \rightarrow y \equiv y')$$

Then $T_1$ is a semantic conservative extension of $T_0$.

Proof. We obtain a $\Sigma_1$ extension $M_1$ of a $\Sigma_0$ model $M_0$ of $T_0$ by defining

$$f^{M_1}(\bar{a}) = \begin{cases} b & \text{if } M_0 \models \psi[\bar{a}, b] \\ \text{arbitrary otherwise} & \end{cases}$$

Since for any $\bar{a}$ there can be at most one $b$ satisfying $M_0 \models \psi[\bar{a}, b]$ this is a sound definition. \(\square\)

Lemma 19. seqCore1 is a conservative extension of seqCore, and thus in particular consistent.

Proof. Inspection of the axioms shows that they are all of the syntactic form required by Lemma 17, except for the definition of seqIndexOf which follows that pattern offered in Lemma 18. The formula to be proved in seqCore is in this case

$$\forall s \forall a \forall j, j'(0 \leq j \land j < \text{seqLen}(s) \land s[j] \equiv a \land \forall k(0 \leq k \land k < j \rightarrow s[k] \neq a)) \land (0 \leq j' \land j' < \text{seqLen}(s) \land s[j'] \equiv a \land \forall k(0 \leq k \land k < j' \rightarrow s[k] \neq a)) \rightarrow j' \equiv j)$$

This can easily seen to be true. \(\square\)

A further criterion for conservative extensions will be needed and presented in Subsection A.5

Digression

In some cases the reverse implication of Lemma 16 is also true. We proceed towards this result by some preliminary observations.

Definition 25 (Expansion). Let $\Sigma_0 \subseteq \Sigma_1$ be signatures, a $\Sigma_1$-structure $M_1 = (M_1, I_1)$ is called an expansion of a $\Sigma_0$-structure $M_0 = (M_0, I_0)$ if $M_0 = M_1$ and for all $f, p \in \Sigma_0$ $I_1(f) = I_0(f)$ and $I_1(p) = I_0(p)$.

Lemma 20 (Coincidence Lemma). Let $\Sigma_0 \subseteq \Sigma_1$ be signatures, and $\phi \in \text{Fml}_{\Sigma_0}$. Furthermore let $M_0$ be a $\Sigma_0$-structure and $M_1$ an $\Sigma_1$-expansion of $M_0$. Then

$$M_0 \models \phi \iff M_1 \models \phi$$

Proof. Obvious. \(\square\)

This lemma says that the truth or falsity of a sentence $\phi$ in a given structure only depends on the symbols actually occurring in $\phi$. It is hard to imagine a logic where this would not hold true. There are in fact, rare cases, e.g., a typed first-order logic with a type hierarchy containing subtypes and abstract types, where the coincidence lemma does not apply.
Definition 26 (Substructure). Let $\mathcal{M} = (M, I)$ and $\mathcal{M}_0 = (M_0, I_0)$ be $\Sigma$-structures. $\mathcal{M}_0$ is called a substructure of $\mathcal{M}$ iff

1. $M_0 \subseteq M$
2. for every $n$-ary function symbol $f \in \Sigma$ and any $n$ of elements $a_1, \ldots, a_n \in M_0$
   \[
   I(f)(a_1, \ldots, a_n) = I_0(f)(a_1, \ldots, a_n)
   \]
3. for every $n$-ary relation symbol $p \in \Sigma$ and any $n$ of elements $a_1, \ldots, a_n \in M_0$
   \[
   (a_1, \ldots, a_n) \in I(p) \iff (a_1, \ldots, a_n) \in I_0(p)
   \]

Lemma 21. Let $\mathcal{M}_0$ be a substructure of $\mathcal{M}$ and $\phi$ logically equivalent to a universal sentence. Then
\[
\mathcal{M} \models \phi \Rightarrow \mathcal{M}_0 \models \phi
\]

Proof Easy induction on the complexity of $\phi$. □

Definition 27. Let $\mathcal{M}$ be a $\Sigma$-structure.

The signature $\Sigma_{\mathcal{M}}$ is obtained from $\Sigma$ by adding new constant symbols $c_a$ for every element $a \in M$.

The expansion of $\mathcal{M}$ to a $\Sigma_{\mathcal{M}}$-structure $\mathcal{M}^* = (M, I^*)$ is effected by the obvious $I^*(c_a) = a$.

Definition 28 (Diagram of a structure). Let $\mathcal{M}$ be a $\Sigma$-structure. The diagram of $\mathcal{M}$, in symbols $\text{Diag}(\mathcal{M})$, is defined by
\[
\text{Diag}(\mathcal{M}) = \{ \phi \in \text{Fml}_{\Sigma_{\mathcal{M}}} \mid \mathcal{M}^* \models \phi \text{ and } \phi \text{ is quantifier free} \}
\]

Lemma 22. Let $\mathcal{M}$ be a $\Sigma$-structure.

If $\mathcal{N} \models \text{Diag}(\mathcal{M})$ then $\mathcal{M}$ is (isomorphic to) a substructure of $\mathcal{N}$.

Proof Easy. □

Lemma 23. Let $\Sigma_0 \subseteq \Sigma_1$ be signatures, and $T_1$ sets of sentences in $\text{Fml}_{\Sigma_1}$ and assume that

1. $T_1$ contains only universal sentences and
2. $\Sigma_1 \setminus \Sigma_0$ contains only relation symbols.

If $T_1$ is a conservative extension of $T_0$ then $T_1$ is also a semantic conservative extension of $T_0$.

Proof We need to show the two clauses in Definition 24.

(1): Let $\mathcal{M}_1$ be a $\Sigma_1$-structure with $\mathcal{M}_1 \models T_1$ and $\mathcal{M}_0$ its restriction to $\Sigma_0$, i.e., $\mathcal{M}_0 = \mathcal{M}_1 \upharpoonright \Sigma_0$. For all $\phi \in T_0$ obviously $T_0 \proves \phi$. Thus also $T_1 \proves \phi$ and therefore $\mathcal{M}_1 \models \phi$. By the coincidence lemma this gives $\mathcal{M}_0 \models \phi$. Thus, we get $\mathcal{M}_0 \models T_0$ as desired.

(2): Here we look at a $\Sigma_0$-structure $\mathcal{M}_0$ with $\mathcal{M}_0 \models T_0$. We set out to find an expansion $\mathcal{M}_1$ of $\mathcal{M}_0$ with $\mathcal{M}_1 \models T_1$. To this end we consider the theory $T_1 \cup \text{Diag}(\mathcal{M}_0)$. If this theory were inconsistent than already $T_1 \cup F$ for a finite subset $F \subseteq \text{Diag}(\mathcal{M}_0)$ would be inconsistent. This is the same as saying $T_1 \proves \neg F$. Since the constants $c_a$ do not occur in $T_1$ we get furthermore $T_1 \proves \forall x_1, \ldots, x_n \neg F'$, where $F'$ is obtained from $F$ be replacing all occurrences of constants $c_a$ by the same variable $x_i$. This is equivalent to $T_1 \proves \neg \exists x_1, \ldots, x_n F'$. Since $T_1$ was assumed to be a conservative extension of $T_0$ we also get $T_0 \proves \neg \exists x_1, \ldots, x_n F'$ and...
thus $M_0 \models \neg \exists x_1, \ldots, x_n F'$. This is a contradiction since by the definition of $\text{Diag}(M_0)$ we have $M_0 \models \exists x_1, \ldots, x_n F'$ by instantiating the quantified variable $x_i$ that replaces the constant $c_a$ by the element $a$. This contradiction shows that $T_1 \cup \text{Diag}(M_0)$ is consistent. Let $N'$ be a model of this theory. By Lemma 22 we may assume that $M_0$ is a substructure of $(N' \upharpoonright \Sigma_0)$. Since by assumption only new relation symbols are added when passing from $\Sigma_0$ to $\Sigma_1$ also $(N' \upharpoonright \Sigma_1)$ is a substructure of $N$. By Lemma 21 we get $(N' \upharpoonright \Sigma_1) \models T_1$. Obviously, $(N' \upharpoonright \Sigma_1)$ is an expansion of $(N' \upharpoonright \Sigma_0) = M_0$ and we are finished.

\[
\square
\]

A.4 Derived Theorem

The KeY system offers bootstrapping verification of the correctness of taclets. On selecting in the main menu file -> prove -> KeY’s taclets the user may select a taclet, he wants to verify in an interaction window showing all loaded taclets. Taclets are loaded from different files. A proof obligation is generated that shows the correctness of the selected taclet on the basis of all taclets contained in different files and all taclets occurring in the same file but textually before the selected taclet. The order of taclets in the file Seq.key has been carefully chosen such that all taclets shown on lines 198 – 758 in Section A.7 can be proved.

We point out that

\[
\forall s \forall i \left( \text{seqIndexOf)(int :: seqGet}(s,i)) = i \right)
\]

is not derivable from seqCore, but

\[
\forall s \forall i \left( \text{seqNPerm}(s) \land 0 \leq i \land i < \text{seqLen}(s) \rightarrow \text{seqIndexOf}(\text{int :: seqGet}(s,i)) = i \right)
\]

is.

A.5 A Second Set of Extensions by Definition

The following predicates and functions will be introduced by defining axioms.

\[
\begin{align*}
\text{seqNPerm}(\text{Seq}) & \quad \text{seqPerm}(\text{Seq}, \text{Seq}) \\
\text{Seq} & \quad \text{seqSwap}(\text{Seq}, \text{int}, \text{int}) \\
\text{Seq} & \quad \text{seqRemove}(\text{Seq}, \text{int}) \\
\text{seqNPermInv}(\text{Seq}) & 
\end{align*}
\]

Let seqCore2 be the theory obtained from seqCore1 by adding the axioms from Figure 9. The corresponding taclets are to be found in Section A.7 on lines 776 to 893. Again we are concerned with proving the consistency of seqCore2. We will eventually show that seqCore2 is a conservative extension of seqCore1 and thus also of seqCore. That addition of the axioms 1 to 4 in the list of Figure 9 lead to conservative extensions directly follows from Lemma 17, these are direct definitions. But, axioms 5 and 6 confront us with another situation. It will turn out that the extension by these two axioms can be reduced to a Skolem extension. For the reader’s convenience we repeat here the classical Skolem extension lemma.

**Lemma 24.** Let $T_0$ be a $\Sigma_0$-theory, $\Sigma_1 = \Sigma_0 \cup \{ f \}$ where $f$ is a new $n$-place function symbol and let $T_1$ be obtained from $T_0$ by adding an axiom of the following form

\[
\forall \bar{x} \exists y (\phi) \rightarrow \phi(f(\bar{x})/y)
\]

then $T_1$ is a conservative extension of $T_0$.

**Here $\bar{x}$ is a tuple of variables of the same length $n$ as the argument tuple of $f$ and, as before $\phi(f(\bar{x})/y)$ denotes the formula arising from $\phi$ by replacing all free occurrences of $y$ by $f(\bar{x})$.**
1. \( \forall s (\text{seqNPerm}(s) \leftrightarrow \forall (0 \leq i < \text{seqLen}(s) \rightarrow 3j(0 \leq j < \text{seqLen}(s) \land s[j] = i))) \)

2. \( \forall s_1, s_2 (\text{seqPerm}(s_1, s_2) \leftrightarrow \text{seqLen}(s_1) \equiv \text{seqLen}(s_2) \land 3s(\text{seqNPerm}(s) \land \forall i(0 \leq i < \text{seqLen}(s_1) \rightarrow s_1[i] = s_2[i]))) \)

3. \( \forall \forall i, j (\text{seqSwap}(s, i, j) \equiv \text{seq_def}\{u\}(0, \text{seqLen}(s), \text{if } \neg (0 \leq i \land 0 \leq j \land i < \text{seqLen}(s) \land j < \text{seqLen}(s))) \land \text{then } s[u] \land \text{else if } u = i \land \text{then } s[j] \land \text{else if } u = j \land \text{then } s[i] \land \text{else } s[u] \) \)

4. \( \forall s (\forall (i < \text{seqLen}(s)) \rightarrow \text{seqRemove}(s, i) \equiv \text{seq_def}\{u\}(0, \text{seqLen}(s), \text{if } u < i \land \text{then } s[u] \land \text{else } s[u + 1]) \) \)

5. \( \forall (s, s_1, s_2 \text{ are of type } \text{Seq}, i, j \text{ are of type } \text{Int}) \) \)

\textbf{Figure 9.} Second Set of Extentions by Definition

\textbf{Proof} We show that \( T_1 \) is a semantic conservative extension of \( T_0 \). Let \( M_0 \) be a model of \( T_0 \). The structure \( M_1 \) coincides with \( M_0 \) for all \( \Sigma_0 \)-symbols. We define an interpretation of the symbol \( f \) as follows

\[
 f^{M_1}(\bar{a}) = \begin{cases} 
 b & \text{if } M_0 \models \exists y(\phi)\{\bar{a}\} \\
 \text{then pick } b \text{ with } M_0 \models \phi[\bar{a}, b] & \text{arbitrary otherwise}
\end{cases}
\]

Technical note, we use \( M_0 \models \phi[\bar{a}, b] \) as a shorthand for \( (M_0, \beta) \models \phi \) with the variable assignment defined by \( \beta(x_i) = a_i \) for \( 0 \leq i < n \) and \( \beta(\bar{y}) = b \).

Obviously, \( M_1 \models \forall x(\exists y(\phi) \rightarrow \phi[f(x)/\bar{y}]) \)

\( \square \)

\textbf{Lemma 25.} \( \text{seqCore2} \) is a conservative extension of \( \text{seqCore} \), and thus in particular consistent.

\textbf{Proof} The theory \( T_0 \) that is obtain by adding axioms 1 to 4 from the list of Figure 9 to \( \text{seqCore1} \) is, as observed above, a conservative extension of \( \text{seqCore} \). Let \( T_1 \) be the theory obtained from \( T_0 \) by adding the following formula

\[
 \forall s \exists t(\phi) \rightarrow \phi(\text{seqNPermInv}(s)/t) \\
\text{with} \\
\phi = \text{seqLen}(t) = \text{seqLen}(s) \land \forall i, j(\text{seqPerm}(s) \land s[j] = i \land 0 \leq i < \text{seqLen}(s) \land 0 \leq j < \text{seqLen}(s)) \rightarrow t[i] = j)
\]

By Lemma 25 \( T_1 \) is a conservative extension of \( T_0 \). We can easily prove \( T_0 \vdash \forall s \exists t(\phi) \). Thus we know \( T_1 \vdash \forall s \phi(\text{seqNPermInv}(s)/t) \),

40
\[ \forall s (\text{seqLen}(s) \rightarrow \text{seqNPermInv}(s)) \]

\[ \forall s \forall i, j 
\begin{align*}
((\text{seqNPerm}(s) \land s[j] = i) & \land 0 \leq i < \text{seqLen}(s) \land 0 \leq j < \text{seqLen}(s)) \\
& \rightarrow \text{seqNPermInv}(s)[i] = j
\end{align*}
\]

Since \( T_1 \) is a conservative extension of \texttt{seqCore}, its subtheory \texttt{seqCore2} also is. We can in fact show that \( T_1 \) is equivalent to \texttt{seqCore2}. \hfill \Box

### A.6 Derived Theorem

See the first paragraph of Section A.4 for general comments.

All taclets in Section A.7 lines 902 to 1121 have been proved using the KeY system. All proofs have been saved and can be replayed.

### A.7 Taclets

```plaintext
\sorts { Seq; }
\predicates { seqPerm(Seq, Seq); seqNPerm(Seq); }
\functions { // getters
  alpha alpha::seqGet(Seq, int);
  int seqLen(Seq);
  int seqIndexOf(Seq, any);
  any seqGetOutside;
}

// constructors
Seq seqEmpty;
Seq seqSingleton(any);
Seq seqConcat(Seq, Seq);
Seq seqSub(Seq, int, int);
Seq seqReverse(Seq);
Seq seqDef{false, false, true}(int, int, any);
Seq seqSwap(Seq, int, int);
Seq seqRemove(Seq, int);
Seq seqNPermInv(Seq);

// placeholder for values in enhanced for loop
Seq values;
}

\rules { // Core axioms
```

41
lenNonNegative {
    \schemaVar \term Seq seq;
    \find(seqLen(seq)) \sameUpdateLevel
    \add(0 <= seqLen(seq) =>)
    \heuristics(inReachableStateImplication)
};

equalityToSeqGetAndSeqLen {
    \schemaVar \term Seq s, s2;
    \schemaVar \variables int iv;
    \find(s = s2)
    \varcond(\notFreeIn(iv, s, s2))
    \replacewith(seqLen(s) = seqLen(s2)
      & \forall iv; (0 <= iv & iv < seqLen(s)
        -> any::seqGet(s, iv) = any::seqGet(s2, iv)))
};

getOfSeqDef {
    \schemaVar \term int idx, from, to;
    \schemaVar \term any t;
    \schemaVar \variables int uSub, uSub1, uSub2;
    \find(alpha::seqGet(seqDef{uSub;}(from, to, t), idx))
    \varcond(\notFreeIn(uSub, from),
      \notFreeIn(uSub, to))
    \replacewith(\if(0 <= idx & idx < (to - from))
      \then( {\subst uSub; (idx + from)}t)
      \else(seqGetOutside))
    \heuristics(simplify)
};

lenOfSeqDef {
    \schemaVar \term int from, to;
    \schemaVar \term any t;
    \schemaVar \variables int uSub, uSub1, uSub2;
    \find(seqLen(seqDef{uSub;} (from, to, t)))
    \replacewith(\if(from<to)\then((to-from))\else(0))
    \heuristics(simplify_enlarging)
};

// ----------------------------------------------
// // Extensions by Definitions
// ----------------------------------------------
castedGetAny {
\schemaVar \term Seq seq;
\schemaVar \term int idx;
\find((beta)any::seqGet(seq, idx))
\replacewith(beta::seqGet(seq, idx))
\heuristics(simplify)
};

seqGetAlphaCast {
\schemaVar \term Seq seq;
\schemaVar \term int at;
\find( alpha::seqGet(seq, at) )
\add((alpha)any::seqGet(seq, at)=alpha::seqGet(seq, at) ==>)
};

defOfEmpty {
\schemaVar \term any te;
\schemaVar \variables int uSub;
\find(seqEmpty)
\varcond ( \notFreeIn(uSub, te))
\replacewith(seqDef{uSub;}(0, 0, te))
};

defOfSeqSingleton {
\schemaVar \term any x;
\schemaVar \variables int uSub;
\find(seqSingleton(x))
\varcond ( \notFreeIn(uSub, x))
\replacewith(seqDef{uSub;}(0,1,x))
};

defOfSeqConcat {
\schemaVar \term Seq seq1, seq2;
\schemaVar \variables int uSub;
\find(seqConcat(seq1, seq2))
\varcond (\notFreeIn(uSub, seq1),
\notFreeIn(uSub, seq2))
\replacewith(seqDef{uSub;}(0,seqLen(seq1)+seqLen(seq2),
\if (uSub < seqLen(seq1))
\then (any::seqGet(seq1,uSub))
\else (any::seqGet(seq2, uSub - seqLen(seq1)))))
};

defOfSeqSub {
\schemaVar \term Seq seq;
\schemaVar \term int from, to;
\schemaVar \variables int uSub;
\find{seqSub}{seq, from, to)}
\varcond{\notFreeIn{uSub, seq),
\notFreeIn{uSub, from), \notFreeIn{uSub, to))
\replacewith{seqDef{uSub;}{from, to, any::seqGet{seq, uSub))
};

defOfSeqReverse {
\schemaVar \term Seq seq;
\schemaVar \variables int uSub;
\find{seqReverse}{seq)}
\varcond{\notFreeIn{uSub, seq})
\replacewith{seqDef{uSub;}{0, seqLen{seq),
    any::seqGet{seq, seqLen{seq)-uSub-1))}
};

seqIndexOf {
\schemaVar \term Seq s;
\schemaVar \term any t;
\schemaVar \skolemTerm int jsk;
\schemaVar \variables int n, m;
\find{seqIndexOf}{s, t)}
\varcond{\new{jsk, \dependingOn{t),
    \notFreeIn{n, s), \notFreeIn{n, t),
    \notFreeIn{m, s), \notFreeIn{m, t})
    \replacewith{jsk}
    \add{ 0 <= jsk & jsk < seqLen{seq) & any::seqGet{seq, jsk)=t &
        forall m:((0<=m&m<jsk) -> any::seqGet{seq, m)!=t)==>);
    \add{ ==> \exists n;(0 <= n & n < seqLen{seq)
    & any::seqGet{seq, n) = t)}
};

// ------------------------------------------
// Derived taclets
// ------------------------------------------

seqSelfDefinition {
\schemaVar \term Seq seq;
\schemaVar \variables Seq s;
\schemaVar \variables int u;
\find{seq )
\add{ forall s;
    \ s = seqDef{u;}{0, seqLen{seq),any::seqGet{seq, u}) ===>}
};

seqOutsideValue {
\schemaVar \variables Seq s;
\schemaVar \variables int iv;
\find{seqGetOutside )
\add{ forall s,\forall iv;((iv < 0 | seqLen{seq)<= iv)
    -> any::seqGet{seq, iv) = seqGetOutside)) ===>)
};
getOfSeqSingleton {  
\schemaVar \term any x;  
\schemaVar \term int idx;  
\find(any::seqGet(seqSingleton(x), idx))  
\replacewith(\if(idx = 0)  
 \then( x )  
 \else( seqGetOutside ))  
\heuristics(simplify)  
};

getOfSeqConcat {  
\schemaVar \term Seq seq, seq2;  
\schemaVar \term int idx;  
\find(any::seqGet(seqConcat(seq, seq2), idx))  
\replacewith(\if(idx < seqLen(seq))  
 \then(any::seqGet(seq, idx))  
 \else(any::seqGet(seq2,idx-seqLen(seq))))  
\heuristics(simplify_enlarging)  
};
etOfSeqSub {  
\schemaVar \term Seq seq;  
\schemaVar \term int idx, from, to;  
\find(any::seqGet(seqSub(seq, from, to), idx))  
\replacewith(\if(0 <= idx & idx < (to - from))  
 \then(any::seqGet(seq, idx + from))  
 \else( seqGetOutside ))  
\heuristics(simplify)  
};

getOfSeqReverse {  
\schemaVar \term Seq seq;  
\schemaVar \term int idx;  
\find(any::seqGet(seqReverse(seq), idx))  
\replacewith(any::seqGet(seq, seqLen(seq) - 1 - idx))  
\heuristics(simplify_enlarging)  
};

lenOfSeqEmpty {  
\find(seqLen(seqEmpty))  
\replacewith(0)  
\heuristics(concrete)  
};
lenOfSeqSingleton {
  \schemaVar \term alpha x;
  \find(seqLen(seqSingleton(x)))
  \replacewith(1)
  \heuristics(concrete)
};

lenOfSeqConcat {
  \schemaVar \term Seq seq, seq2;
  \find(seqLen(seqConcat(seq, seq2)))
  \replacewith(seqLen(seq) + seqLen(seq2))
  \heuristics(simplify)
};

lenOfSeqSub {
  \schemaVar \term Seq seq;
  \schemaVar \term int from, to;
  \find(seqLen(seqSub(seq, from, to)))
  \replacewith(\if(from < to)\then(to - from)\else(0))
  \heuristics(simplify_enlarging)
};

lenOfSeqReverse {
  \schemaVar \term Seq seq;
  \find(seqLen(seqReverse(seq)))
  \replacewith(seqLen(seq))
  \heuristics(simplify)
};

equalityToSeqGetAndSeqLenLeft {
  \schemaVar \term Seq s, s2;
  \schemaVar \variables int iv;
  \find(s = s2 =>)
  \varcond(\notFreeIn(iv, s, s2))
  \add(seqLen(s) = seqLen(s2)
    & \forallall iv; (0 <= iv & iv < seqLen(s)
    -> any::seqGet(s, iv) = any::seqGet(s2, iv)) =>)
    \heuristics(inReachableStateImplication)
};

equalityToSeqGetAndSeqLenRight {
\schemaVar \term Seq s, s2;
\schemaVar \variables int iv;
\find(==> s = s2)
\varcond(\notFreeIn(iv, s, s2))
\replacewith(==> seqLen(s) = seqLen(s2)
& \forall iv; (0 <= iv & iv < seqLen(s)
-> any::seqGet(s, iv) = any::seqGet(s2, iv)))
\heuristics(simplify_enlarging)
};

getOfSeqSingletonEQ {
\schemaVar \term any x;
\schemaVar \term int idx;
\schemaVar \term Seq EQ;
\assumes(seqSingleton(x) = EQ ==>)
\find(any::seqGet(EQ, idx))
\sameUpdateLevel
\replacewith(if(idx = 0)
  \then( x)
  \else(seqGetOutside))
\heuristics(simplify)
};

getOfSeqConcatEQ {
\schemaVar \term Seq seq, seq2;
\schemaVar \term int idx;
\schemaVar \term Seq EQ;
\assumes(seqConcat(seq, seq2) = EQ ==>)
\find(any::seqGet(EQ, idx))
\sameUpdateLevel
\replacewith(if(idx < seqLen(seq))
  \then(any::seqGet(seq, idx))
  \else(any::seqGet(seq2, idx-seqLen(seq))))
\heuristics(simplify_enlarging)
};

getOfSeqSubEQ {
\schemaVar \term Seq seq;
\schemaVar \term int idx, from, to;
\schemaVar \term Seq EQ;
\assumes(seqSub(seq, from, to) = EQ ==>)
\find(any::seqGet(EQ, idx))
\sameUpdateLevel
\replacewith(if(0 <= idx & idx < (to - from))
  \then(any::seqGet(seq, idx + from))
  \else(seqGetOutside))
\heuristics(simplify)
};
getOfSeqReverseEQ { 
  \schemaVar \term Seq seq; 
  \schemaVar \term int idx; 
  \schemaVar \term Seq EQ; 
  \assumes(seqReverse(seq) = EQ =>) 
  \find(any::seqGet(EQ, idx)) 
  \sameUpdateLevel 
  \replacewith(any::seqGet(seq, seqLen(seq) - 1 - idx)) 
  \heuristics(simplify_enlarging) 
}; 

lenOfSeqEmptyEQ { 
  \schemaVar \term alpha x; 
  \schemaVar \term Seq EQ; 
  \assumes(seqEmpty = EQ =>) 
  \find(seqLen(EQ)) 
  \sameUpdateLevel 
  \replacewith(0) 
  \heuristics(concrete) 
}; 

lenOfSeqSingletonEQ { 
  \schemaVar \term alpha x; 
  \schemaVar \term Seq EQ; 
  \assumes(seqSingleton(x) = EQ =>) 
  \find(seqLen(EQ)) 
  \sameUpdateLevel 
  \replacewith(1) 
  \heuristics(concrete) 
}; 

lenOfSeqConcatEQ { 
  \schemaVar \term Seq seq, seq2; 
  \schemaVar \term Seq EQ; 
  \assumes(seqConcat(seq, seq2) = EQ =>) 
  \find(seqLen(EQ)) 
  \sameUpdateLevel 
  \replacewith(seqLen(seq) + seqLen(seq2)) 
  \heuristics(simplify) 
}; 

lenOfSeqSubEQ { 
  \schemaVar \term Seq seq; 
  \schemaVar \term int from, to; 
  \schemaVar \term Seq EQ;
\assumes(seqSub(seq, from, to) = EQ ==>)
\find(seqLen(EQ))
\sameUpdateLevel
\replacewith(\if(from < to)\then(to - from)\else(0))
\heuristics(simplify_enlarging)
}

lenOfSeqReverseEQ {
\schemaVar \term Seq seq;
\schemaVar \term Seq EQ;
\assumes(seqReverse(seq) = EQ ==>)
\find(seqLen(EQ))
\sameUpdateLevel
\replacewith(seqLen(seq))
\heuristics(simplify)
}

getOfSeqDefEQ {
\schemaVar \term int idx, from, to;
\schemaVar \term Seq EQ;
\schemaVar \term any t;
\schemaVar \variables int uSub, uSub1, uSub2;

\assumes(seqDef{uSub;} (from, to, t) = EQ ==>)
\find(any::seqGet(EQ, idx))
\varcond ( \notFreeIn(uSub, from),
  \notFreeIn(uSub, to))
\replacewith(\if(0 <= idx & idx < (to - from))
  \then((\subst uSub; (idx + from))t)
  \else (seqGetOutside))
\heuristics(simplify)
}

lenOfSeqDefEQ {
\schemaVar \term int from, to;
\schemaVar \term Seq EQ;
\schemaVar \term any t;
\schemaVar \variables int uSub, uSub1, uSub2;

\assumes(seqDef{uSub;} (from, to, t) = EQ ==>)
\find(seqLen(EQ))
\replacewith(\if(from<=to)\then((to-from))\else(0))
\heuristics(simplify_enlarging)
}

seqConcatWithSeqEmpty1 {
\schemaVar \term Seq seq;
\find(seqConcat(seq, seqEmpty))
\replacewith(seq)
seqConcatWithSeqEmpty2 { |
  \schemaVar \term Seq seq;
  \find(seqConcat(seqEmpty, seq))
  \replacewith(seq)
  \heuristics(concrete)
};

seqReverseOfSeqEmpty { |
  \find(seqReverse(seqEmpty))
  \replacewith(seqEmpty)
  \heuristics(concrete)
};

subSeqComplete { |
  \schemaVar \term Seq seq;
  \find(seqSub(seq, 0, seqLen(seq)))
  \replacewith(seq)
  \heuristics(concrete)
};

subSeqTail { |
  \schemaVar \term Seq seq;
  \schemaVar \term any x;
  \find(seqSub(seqConcat(seqSingleton(x), seq), 1, seqLen(seq)+1))
  \replacewith(seq)
  \heuristics(concrete)
};

subSeqTailEQ { |
  \schemaVar \term Seq seq;
  \schemaVar \term any x;
  \schemaVar \term int EQ;
  \assumes(seqLen(seq) = EQ ==>)
  \find(seqSub(seqConcat(seqSingleton(x), seq), 1, EQ+1))
  \sameUpdateLevel
  \replacewith(seq)
  \heuristics(concrete)
};

seqDef_split { |
  \schemaVar \term int idx, from, to;
\textbf{seqDef_induction_upper} {
\begin{verbatim}
\schemaVar \term int idx, from, to;
\schemaVar \term any t;
\schemaVar \variables int uSub, uSub1, uSub2;
\find(seqDef{uSub;} (from, to, t))
\varcond ( \notFreeIn(uSub1, from),
\notFreeIn(uSub1, idx),
\notFreeIn(uSub1, to),
\notFreeIn(uSub, from),
\notFreeIn(uSub, idx),
\notFreeIn(uSub, to),
\notFreeIn(uSub1, t) )
\replacewith(\if(from <=idx & idx < to)
  \then(seqConcat(
    seqDef{uSub;}(from, idx, t),
    seqDef{uSub1;}(idx,to,{\subst uSub;uSub1}t)))
  \else(seqDef{uSub;}(from, to, t)))
};
\end{verbatim}
}

\textbf{seqDef_induction_upper_concrete} {
\begin{verbatim}
\schemaVar \term int idx, from, to;
\schemaVar \term any t;
\schemaVar \variables int uSub, uSub1, uSub2;
\find(seqDef{uSub;} (from, 1+to, t))
\varcond ( \notFreeIn(uSub, from),
\notFreeIn(uSub, to))
\replacewith(seqConcat(
  seqDef{uSub;} (from, to, t),
  \if(from<to)
    \then(seqSingleton({\subst uSub; (to)}t))
    \else(seqEmpty)))
\heuristics(simplify)
};
\end{verbatim}
}

\textbf{seqDef_induction_lower} {
\begin{verbatim}
\schemaVar \term int idx, from, to;
\schemaVar \term any t;
\schemaVar \variables int uSub, uSub1, uSub2;
\find(seqDef{uSub;} (from, to, t))
\varcond ( \notFreeIn(uSub, from),
\notFreeIn(uSub, to))
\replacewith(seqConcat(
  seqDef{uSub;} (from, to, t),
  \if(from<to)
    \then(seqSingleton({\subst uSub; (to)}t))
    \else(seqEmpty)))
\heuristics(simplify)
};
\end{verbatim}
}
seqDef_induction_lower_concrete {
  \schemaVar \term int idx, from, to;  
  \schemaVar \term any t;  
  \schemaVar \variables int uSub, uSub1, uSub2;  
  \find(seqDef{uSub;}(-1+from, to, t)) \sameUpdateLevel  
  \varcond ( \notFreeIn(uSub, from), \notFreeIn(uSub, to))  
  \replacewith(seqConcat(  
    \if(-1+from<to)  
      \then(seqSingleton({\subst uSub; (-1+from)}t))  
      \else(seqEmpty),  
      seqDef{uSub;} (from, to, t))))  
  \heuristics(simplify)  
};

seqDef_split_in_three {
  \schemaVar \term int idx, from, to;  
  \schemaVar \term any t;  
  \schemaVar \variables int uSub, uSub1, uSub2;  
  \find(seqDef{uSub;} (from, to, t)) \sameUpdateLevel  
  \varcond ( \notFreeIn(uSub, idx), \notFreeIn(uSub1, t), \notFreeIn(uSub1, idx), \notFreeIn(uSub, from), \notFreeIn(uSub1, to))  
  "Precondition": \add(==> (from<=idx & idx<to));  
  "Splitted_seqDef": \replacewith(  
    seqConcat(seqDef{uSub;} (from, idx, t),  
    seqConcat(seqSingleton({\subst uSub; idx}t),  
    seqDef{uSub1;}(idx+1,to,\text{\subst uSub;uSub1}t))))
};

seqDef_empty {
  \schemaVar \term int idx, from, to;  
  \schemaVar \term any t;  
  \schemaVar \variables int uSub, uSub1, uSub2;  
  \find(seqDef{uSub;} (from, idx, t)) \sameUpdateLevel  
  \varcond ( \notFreeIn(uSub, from), \notFreeIn(uSub, idx))  
  "Precondition": \add(==> idx<=from);  
  "Empty_seqDef": \replacewith(seqEmpty)
};

seqDef_one_summand {
  \schemaVar \term int idx, from, to;  
  \schemaVar \term any t;  
  \schemaVar \variables int uSub, uSub1, uSub2;  
  \find(seqDef{uSub;} (from, idx, t)) \sameUpdateLevel  
  \varcond ( \notFreeIn(uSub, from), \notFreeIn(uSub, idx))
\replacewith(\if(from+1=idx)
   \then(seqSingleton(\subst uSub; from)t))
\else(seqDef{uSub;} (from, idx, t)))
);

seqDef_lower_equals_upper {
   \schemaVar \term int idx, from, to;
   \schemaVar \term any t;
   \schemaVar \variables int uSub, uSub1, uSub2;
   \find(seqDef{uSub;} (idx, idx, t))\sameUpdateLevel
   \varcond (\notFreeIn(uSub, idx))
   \replacewith(seqEmpty)
   \heuristics(simplify)
};

indexOfSeqSingleton {
   \schemaVar \term any x;
   \find(seqIndexOf(seqSingleton(x),x))\sameUpdateLevel
   \replacewith(0)
   \heuristics(concrete)
};

indexOfSeqConcatFirst {
   \schemaVar \term Seq s1, s2;
   \schemaVar \term any x;
   \schemaVar \variables int idx;
   \find(seqIndexOf(seqConcat(s1,s2),x))\sameUpdateLevel
   \varcond (\notFreeIn(idx,s1,s2,x))
   \replacewith(seqIndexOf(s1,x)) ;
   \add(==> \exists idx; (0 <= idx & idx < seqLen(s1) &
      any::seqGet(s1,idx) = x))
};

indexOfSeqConcatSecond {
   \schemaVar \term Seq s1, s2;
   \schemaVar \term any x;
   \schemaVar \variables int idx;
   \find(seqIndexOf(seqConcat(s1,s2),x))\sameUpdateLevel
   \varcond (\notFreeIn(idx,s1,s2,x))
   \replacewith(add(seqIndexOf(s2,x),seqLen(s1))) ;
   \add(==> (!
      \exists idx; (0<=idx & idx<seqLen(s1) & any::seqGet(s1,idx)=x)
      & \exists idx;
      (0<=idx & idx<seqLen(s2) & any::seqGet(s2,idx)=x)))
};

indexOfSeqSub {
   \schemaVar \term Seq s;
   \schemaVar \term int from, to, n;
   \schemaVar \term any x;
   \schemaVar \variables int nx;
   \find(seqIndexOf(seqSub(s,from,to),x))\sameUpdateLevel
\textbf{varcond} (\textbf{notFreeIn}(nx, s), \textbf{notFreeIn}(nx, x),
\textbf{notFreeIn}(nx, from), \textbf{notFreeIn}(nx, to))
\textbf{replacewith}(\textbf{sub}(\textbf{seqIndexOf}(s, x), from));
\textbf{add}(==>
\textbf{from}<=\textbf{seqIndexOf}(s, x) \& \textbf{seqIndexOf}(s, x)<\textbf{to} \& \textbf{<=from} \&
\textbf{exists nx;}(0<=nx\&nx<\textbf{seqLen}(s) \& \textbf{any::seqGet}(s, nx)=x)))
);
seqPermDef{
  \schemaVar \term Seq s1, s2, s3;
  \schemaVar \variables int iv;
  \schemaVar \variables Seq s;

  \find(seqPerm(s1,s2))

  \varcond ( \notFreeIn (iv,s1,s2),
      \notFreeIn (s,s1,s2))

  \replacewith( seqLen(s1) = seqLen(s2) &
    (\exists s; (seqLen(s) = seqLen(s1) & seqNPerm(s) &
      (\forall iv; (0 <= iv & iv < seqLen(s) ->
        any::seqGet(s1,iv)=any::seqGet(s2,int::seqGet(s,iv)))))
  )
};

defOfSeqSwap {
  \schemaVar \term Seq s;
  \schemaVar \term int iv,jv;
  \schemaVar \variables int uSub;

  \find(seqSwap(s,iv,jv))

  \varcond ( \notFreeIn(uSub, s),
      \notFreeIn(uSub, iv),
      \notFreeIn(uSub, jv) )

  \replacewith(seqDef{uSub;}(0,seqLen(s),
    \if (!(0<=iv & 0<=jv & iv<seqLen(s) & jv<seqLen(s)))
      \then (any::seqGet(s,uSub))
    \else ( \if(uSub = iv)
      \then(any::seqGet(s,jv))
    \else(\if(uSub = jv)
      \then(any::seqGet(s,iv))
    \else(any::seqGet(s,uSub))))
  )
};

defOfSeqRemove {
  \schemaVar \term Seq s;
  \schemaVar \term int iv;
  \schemaVar \variables int uSub;

  \find(seqRemove(s,iv))

  \varcond ( \notFreeIn(uSub, s),
      \notFreeIn(uSub, iv) )

  \replacewith(
    \if (iv < 0 | seqLen(s) <= iv )
      \then (s)
    \else (seqDef{uSub;}(0,seqLen(s)-1,
      \if (uSub < iv)
        \then (any::seqGet(s,uSub))
      \else (any::seqGet(s,uSub+1))))
  )
};

lenOfNPermInv {
  \schemaVar \term Seq s1;
  \find(seqLen(seqNPermInv(s1)))
getOfNPPermInv {
    \schemaVar \term Seq s1;
    \schemaVar \term int i3;
    \schemaVar \skolemTerm int jsk;
    \find(int::seqGet(seqNPPermInv(s1), i3)) \varcond ( \new(jsk, \dependingOn(i3)))
    \replacewith(jsk)
    \add (int::seqGet(s1,jsk)=i3 & 0<=jsk&jsk<seqLen(s1)==>);
    \add ( ==> 0<= i3 & i3 < seqLen(s1))
    \heuristics(simplify)
};

// Second set of derived taclets

lenOfSwap {
    \schemaVar \term Seq s1;
    \schemaVar \term int iv1, iv2;
    \find(seqLen(seqSwap(s1,iv1,iv2))) \replacewith(seqLen(s1))
    \heuristics(simplify)
};

getOfSwap {
    \schemaVar \term Object o;
    \schemaVar \term Seq s1;
    \schemaVar \term int iv, jv, idx;
    \schemaVar \term Heap h;
    \find(any::seqGet(seqSwap(s1,iv,jv), idx))
    \replacewith(
        \if (!(0<=iv & 0<=jv & iv<seqLen(s1) & jv<seqLen(s1)))
        \then (any::seqGet(s1,idx))
        \else ( \if(idx = iv)
            \then(any::seqGet(s1,jv))
            \else(\if(idx = jv)
                \then(any::seqGet(s1,iv))
                \else(\if(idx = jv))))
    \heuristics(simplify)
};

lenOfRemove {
    \schemaVar \term Seq s1;
    \schemaVar \term int iv1;
    \find(seqLen(seqRemove(s1,iv1)))
    \replacewith(
\if (0 <= iv1 & iv1 < seqLen(s1))
  \then (seqLen(s1)-1)
  \else (seqLen(s1)))
\heuristics(simplify)
};

getOfRemoveAny {
  \schemaVar \term Seq s1;
  \schemaVar \term int i3,i2;
  \find(any::seqGet(seqRemove(s1,i2), i3))
  \replacewith(\if (i2 < 0 | seqLen(s1) <= i2)
    \then(any::seqGet(s1,i3))
    \else(\if(i3 < i2)
      \then(any::seqGet(s1,i3))
      \else(\if(i2<=i3 & i3<seqLen(s1)-1)
        \then(any::seqGet(s1,i3+1))
        \else (seqGetOutside)))))
\heuristics(simplify)
};

getOfRemoveInt {
  \schemaVar \term Seq s1;
  \schemaVar \term int i3,i2;
  \find(int::seqGet(seqRemove(s1,i2), i3))
  \replacewith(\if (i2 < 0 | seqLen(s1) <= i2)
    \then(int::seqGet(s1,i3))
    \else(\if(i3 < i2)
      \then(int::seqGet(s1,i3))
      \else(\if(i2<=i3 & i3<seqLen(s1)-1)
        \then(int::seqGet(s1,i3+1))
        \else((int)seqGetOutside)))))
\heuristics(simplify)
};

lenOfRemoveConcrete1 {
  \schemaVar \term Seq s1;
  \assumes(seqLen(s1)>= 1 ==>)
  \find(seqLen(seqRemove(s1,seqLen(s1)-1)))
  \replacewith(seqLen(s1)-1)
\heuristics(simplify)
};

lenOfRemoveConcrete2 {
  \schemaVar \term Seq s1;
  \assumes(seqLen(s1)>= 1 ==>)
  \find(seqLen(seqRemove(s1,0)))
  \replacewith(seqLen(s1)-1)
\heuristics(simplify)
};
getOfRemoveAnyConcrete1 { 
  \schemaVar \term Seq s1; 
  \schemaVar \term int i3,i2; 
  \assumes(seqLen(s1) >= 1 ==>) 
  \find(any::seqGet(seqRemove(s1,seqLen(s1)-1), i3)) 
  \replacewith(\if (i3 < seqLen(s1)-1) 
    \then (any::seqGet(s1,i3)) 
    \else (seqGetOutside)) 
  \heuristics(simplify) 
}; 

getOfRemoveAnyConcrete2 { 
  \schemaVar \term Seq s1; 
  \schemaVar \term int i3,i2; 
  \assumes(seqLen(s1) >= 1 ==>) 
  \find(any::seqGet(seqRemove(s1,0), i3)) 
  \replacewith(\if (0 <= i3 & i3 < seqLen(s1)-1) 
    \then (any::seqGet(s1,i3+1)) 
    \else (seqGetOutside)) 
  \heuristics(simplify) 
}; 

seqNPermRange { 
  \schemaVar \term Seq s; 
  \schemaVar \variables int iv; 
  \find(seqNPerm(s) ==>) 
  \varcond( \notFreeIn (iv,s) ) 
  \add(\forall iv;((0 <= iv & iv < seqLen(s)) -> 
    (0<=int::seqGet(s,iv)&int::seqGet(s,iv)<seqLen(s)))==>) 
}; 

seqNPermInjective { 
  \schemaVar \term Seq s; 
  \schemaVar \variables int iv,jv; 
  \find(seqNPerm(s) ==>) 
  \varcond( \notFreeIn (iv,s), \notFreeIn (jv,s) ) 
  \add(\forall iv;\forall jv;((0 <= iv & iv < seqLen(s) & 0 <= jv & jv < seqLen(s)
    & int::seqGet(s,iv) = int::seqGet(s,jv) )
  -> iv = jv )) ==>) 
}; 

seqPermTrans{
  \schemaVar \term Seq s1, s2, s3; 
  \assumes( seqPerm(s2,s3) ==>) 
  \find(seqPerm(s1,s2) ==>) 
  \add(seqPerm(s1,s3) ==>) 
}; 

seqPermRef1{
  \schemaVar \term Seq s1; 
  \add(seqPerm(s1,s1) ==>) 
}; 

seqNPermSwapNPerm { 
}
\schemaVar \term Seq s1;
\schemaVar \variables int iv,jv;
\find( seqNPerm(s1) ==> )
\varcond( \notFreeIn(iv, s1), \notFreeIn(jv, s1) )
\add(\forall iv;\forall jv;
(0 <= iv & 0 <= jv & iv < seqLen(s1) & jv < seqLen(s1))
\rightarrow seqNPerm(seqSwap(s1,iv,jv))) ==> )
};

seqNPermComp {
\schemaVar \term Seq s1,s2;
\schemaVar \variables int u;
\find( seqNPerm(s2) & seqLen(s1) = seqLen(s2) ==> )
\varcond( \notFreeIn(u, s1), \notFreeIn(u, s2) )
\add(\forall iv;\forall jv;
(0 <= iv & 0 <= jv & iv < seqLen(s1) & jv < seqLen(s1))
\rightarrow seqNPerm(seqSwap(s1,iv,jv))) ==> )
};

seqGetSInvS {
\schemaVar \term Seq s;
\schemaVar \term int t;
\find( int::seqGet(s, int::seqGet(seqNPermInv(s), t)))
\replacewith( t );
\add( ==> seqNPerm(s) & 0 <= t & t < seqLen(s))
\heuristics(simplify)
};

seqNPermInvNPermLeft{
\schemaVar \term Seq s1;
\find( seqNPerm(s1) ==> )
\add(seqNPerm(seqNPermInv(s1)) ==> )
};

seqPermSym{
\schemaVar \term Seq s1,s2;
\find( seqPerm(s1, s2) ==> )
\add(seqPerm(s2, s1) ==> )
};

seqNPermInvNPermReplace{
\schemaVar \term Seq s1;
\find( seqNPerm(seqNPermInv(s1)))
\replacewith(seqNPerm(s1))
}

seqnormalizeDef{
\schemaVar \term Seq s1;
\schemaVar \term int le,ri;
\schemaVar \term any t;
\schemaVar \variables int u;
}
\find(seqDef{u;}(le, ri, t))
\varcond( \notFreeIn(u, le), \notFreeIn(u, ri))
\replacewith(
  \if(le < ri)
    \then (seqDef{u;}(0, (ri - le), (\subst u; (u + le)t)))
  \else (seqEmpty))
);
B Appendix: Observations using Reference Sets

comment PHS: I have finally put this approach in the appendix. I think definition 30 below is flawed, see Example 9.

In the verification of functional properties or separation properties of programs location sets play a dominant role. So it is tempting to formulate information flow properties also in terms of location sets. This is the topic of this section.

In addition to the type LocSet, see Figures 1 and 2 we need another type refSet.

An expression of type refSet is a pair consisting of a set of program variables and static fields besides and a location set expression:

Definition 29 (Reference set expression). If \( v_1, \ldots, v_k \) (\( k \geq 0 \)) are local variables and static fields, and \( L \) is an expression of type LocSet, then \( R = (\{v_1, \ldots, v_k\}, L) \) is an expression of type refSet.

For a state \( s \), the semantics of \( R \) is defined by \( R^s = (\{v_1, \ldots, v_k\}, L^s) \). To simplify notation, we write \( v \in R \) if \( v \in \{v_1, \ldots, v_k\} \) and \( (o, f) \in R^s \) if \( (o, f) \in L^s \). Moreover, we write just \( \mathsf{V} \) instead of \( (\mathsf{V}, \emptyset) \) and \( L \) instead of \( (\emptyset, L) \).

The set of objects referenced by \( R \) is defined by

\[
\text{obj}^s(R) = \{v^s \mid \text{type}(v) \subseteq \text{Object}, 1 \leq i \leq k\} \cup \\
\{o \mid (o, f) \in R^s\} \cup \{f^s(o) \mid (o, f) \in R^s, \text{type}(f) \subseteq \text{Object}\}
\]

Note that \( \text{obj}^s(V,e,f) \) contains both the object \( e^s \) and the object \( (e,f)^s \).

Definition 30. Let \( R = R_{v_1,\ldots,v_k}(L) \) be an expression of type refSet. We say that two states \( s, s' \) agree on \( R \), abbreviated by \( \text{agree}_s(R, s, s') \), iff there is a partial isomorphism \( \pi \) with respect to \( R \) from \( s \) to \( s' \), that is \( \pi \) is a bijective mapping from \( \text{obj}^s(R) \) onto \( \text{obj}^{s'}(R) \) satisfying:

1. \( \pi \) is type preserving, i.e. \( o \in T^0 \Leftrightarrow \pi(o) \in T^D \) for all \( o \in \text{obj}^s(R) \) and all types \( T \).
2. \( s(v) = s'(v) \) for all \( v \in \mathsf{V} \) with \( \text{type}(v_i) = \text{Boolean} \) or \( \text{type}(v_i) = \text{Int} \);
3. \( \pi(s(v)) = s'(v) \) for all \( v \in \mathsf{V} \) with \( \text{type}(v_i) \subseteq \text{Object} \);
4. \( f^s(o) = f^{s'}(\pi(o)) \) for all \( (o, f) \in L^s \) where the \( \text{type}(f) \subseteq \text{Object} \);
5. \( \pi(f^s(o)) = f^{s'}(\pi(o)) \) for all \( (o, f) \in L^s \) with \( \text{type}(f) \subseteq \text{Object} \);
6. \( \{(\pi(o), f) \mid (o, f) \in L^s\} = L^{s'} \).

Using the intuitive notation \( \pi(L^s) = \{(\pi(o), f) \mid (o, f) \in L^s\} \) we may also write this requirement as \( \pi(L^s) = L^{s'} \).

Example 9.

```java
class C {
    static C x, y;
    public v;
    static boolean h;

    static void m() {
        x = new C();
        y = new C();
        x.v = 0, y.v = 0;
        if (h) { x.v = 1; y.v = 0; }
    }
}
```

Let \( R = R_{x,v}(\{x.v, y.v\}) \). Intuitively, method \( m() \) leaks information about \( h \). The attacker can observe whether the values of \( x.v \) and \( y.v \) coincide or not. But, according to Definition 30 we would have \( \text{agree}(R, s, s') \) for the end states \( s, s' \).
reached by $\pi(Q)$ regardless of the value of $h$ in the prestates. If in one prestate $h = false$ and in the other $h = true$, then we are allowed to chose an isomorphism $\pi$ such that the conditions of Definition 30 are satisfied. This is actually possible by choosing $\pi(x^*) = y^*$ and $\pi(y^*) = x^*$. Note, that with $R^* = R_{x,y}((x,v,y,v))$ there is no problem.

The definition of type $\text{LocSet}$ is very liberal. We did not exclude $(o,f) \in L^*$ with $o = \text{null}$ or created$^*(o) = \text{ff}$. For this reason we need to include the following two clauses in this definition.

7. If $null \notin \text{obj}^*(R)$ then $\pi(\text{null}) = \text{null}$
8. For all $o \in \text{obj}^*(R)$: created$^*(o) = tt \Leftrightarrow \text{created}^*(\pi(o)) = tt$.

We will sometimes also use the phrase “partial $R$-isomorphism” in place of “partial isomorphism with respect to $R$”. Notice, that we have used the shorthand notation for semantics as explained in the paragraph above Example 1 on page 8. Unfolding the shorthand, e.g., item 4 reads select$^\text{fpt}_R(\text{heap}^*, o, f^D) = \text{select}_R^\text{fpt}(\text{heap}^*, \pi(o), f^D)$.

We use the notation agree$_r(R, s, s', \pi)$ to state that $s, s'$ agree on $R$ via the partial isomorphism $\pi$.

We could have used overloading in the designation of agree since the type of the first argument determines whether Definition 13 or Definition 30 applies. For ease of reading we chose to make the difference explicit, agree vs agree$_r$.

For later reference we we write down the requirements from Definition 30 for the special case $\pi = \text{id}$.

**Lemma 26.** Let $R = R_{v_1,..,v_n}(L)$ be an expression of type $\text{refSet}$. Then agree$_r(R, s, s', \text{id})$ is true iff

1. $s(v) = s'(v)$ for all $v \in V$
2. $f^*(o) = f^*(o)$ for all $(o,f) \in L^*$
3. $L^* = L^*$.

Note, that obj$^*(R) = \text{obj}^*(R)$ is also a consequence of agree$_r(R, s, s', \text{id})$.

**Proof.** Easy inspection. \hfill \square

The following criterion will be essential in the following.

**Lemma 27.** Let $R = R_{v_0,..,v_{n-1}}(L)$ be a reference set expression, $s$, $s'$ be states, and $S$, $S'$ be sequences and $n \in \mathbb{N}$ such that

1. For all $i$, $0 \leq i < k$: $S[i] = v^*_i = v^*_i = S'[i]$.
2. For all $j$ with $k \leq 2j < n - 1$: $S[2j] \in \text{Object}$ and $S'[2j] \in \text{Object}$ and there is a field $f$ such that $S[2j + 1] = f^*(S[2j])$ and $S'[2j + 1] = f^*(S'[2j])$.
3. For all objects $o$ and all fields $f$
   
   $(o,f) \in L^* \Rightarrow S[2j] = o \wedge S'[2j + 1] = f^*(o)$ for some $j$ with $k \leq 2j < n - 1$.

4. For all objects $o$ and all fields $f$
   
   $(o,f) \in L^* \Rightarrow S'[2j] = o \wedge S'[2j + 1] = f^*(o)$ for some $j$ with $k \leq 2j < n - 1$.

5. For all integers $i$ with $k \leq i < n$ and type$(S[i]) = \text{type}(S'[i])$ and if $S[i] \notin \text{Object}$ then $S'[i] = S'[i]$.
6. For all integers $i$, $j$ with $k \leq i < j < n$ and $S[i] \in \text{Object}$ and $S'[j] \in \text{Object}$:
   

7. For all integers $i$ with $0 \leq i < n$
   
   $S[i] = \text{null} \Rightarrow S'[i] = \text{null}$

8. For all integers $i$ with $0 \leq i < n$
   
   $\text{created}^*(S[i]) = \text{created}^*(S'[i])$

Then agree$(R, s, s')$. 62
Proof. We need to exhibit a bijection $\pi$ from $\text{obj}^*(R)$ onto $\text{obj}^{s'}(R)$ such that $\text{agree}(R, s, s', \pi)$.

In keeping with Definition 16 we use the notation $\text{obj}(S) = \{S[i] \mid S[i] \in \text{Object}, 0 \leq i < n\}$

We first observe that

$$\text{obj}^*(R) = \text{obj}(S)$$

and

$$\text{obj}^{s'}(R) = \text{obj}(S')$$

If $o \in \text{obj}^*(R)$ then we distinguish three possibilities:

1. $o = v_i^o$ for some $0 \leq i < k$
   - By assumption 1 we have $o = S[i]$ and thus $o \in \text{obj}(S)$.
   - $(o, f) \in L^*$ for some $f$
   - By assumption 3 there is $i$ with $S[2i] = o$ and thus $o \in \text{obj}(S)$.
   - $o = f^s(o')$ for some $(o', f) \in L^*$
   - Again by assumption 3 there is $i$ with $S[2i] = o'$ and $S[2i+1] = f^s(o')$. Thus again $o \in \text{obj}(S)$.

This establishes $\text{obj}^*(R) \subseteq \text{obj}(S)$. If $o \in \text{obj}(S)$ then there is by definition an index $i$ such that $o = S[i]$. If $0 \leq i < k$ then $S[i] = v_i^o$ and thus $o \in \text{obj}^*(R)$. If $k \leq i < n$ then there is $j$ such that $i = 2j$ or $i = 2j + 1$. By assumption 2 there is a field $j$ such that $S[2j+1] = f^s(S[2j])$ and $S[2j] \in \text{Object}$. By assumption 3 this implies $(S[2j], f) \in L^*$ and thus $o \in \text{obj}^*(R)$ in any case.

This complete the proof of 9. Claim 10 is proved along the same lines using assumptions 1, 2, 4.

The mapping $\pi$ is defined for $o = S[i] \in \text{obj}(S)$ by $\pi(o) = S'[i]$. Item 6 guarantees that $\pi$ is well defined and bijective.

It is easily checked that the requirement of Definition 30 are satisfied. We present here the arguments for items 5 and 6.

For item 5 consider $(o, f) \in L^*$ with $\text{type}(f) \subseteq \text{Object}$. By assumption 3 of the present lemma there is an index $i$, with $S[2i] = o$ and $S[2i+1] = f^s(o)$.

Thus by definition of $\pi$ and assumption 2 we have $\pi(f^s(o)) = \pi(S[2i+1]) = S'[2i+1] = f^{s'}(S'[2i]) = f^{s'}(\pi(S[2i])) = f^{s'}(\pi(o))$.

For item 6 we argue as follows.

\[
\{(\pi(o), f) \mid (o, f) \in L^*\} = \{(\pi(o), f) \mid o = S[2i], f^s(o) = S[2i+1] \text{ for some } i, k \leq i < n\}
\]

by assumption 3

\[
= \{(o', f) \mid o' = S'[2i], f^{s'}(o) = S'[2i+1] \text{ for some } i, k \leq i < n\}
\]

by definition of $\pi$

\[
= L^{s'}
\]

by assumption 4

The following converse of Lemma 27 is also true.

**Lemma 28.** Let $R = R_{n, \ldots, s_{k-1}}(L)$ be a reference set expression, $s$, $s'$ be states,
and $\pi$ a partial isomorphism from $\text{obj}^*(R)$ onto $\text{obj}^{s'}(R)$ such that $\text{agree}(R, s, s', \pi)$
then there are sequences $S$, $S'$ and $n \in \mathbb{N}$ such that item 1 to 8 of Lemma 27 are satisfied.

**Proof.** Let $L^* = \{(o_j, f_j) \mid 0 \leq j < m\}$. We set $n = k + 2m$ define the sequence $S$ by $S[i] = v_i^o$ for $0 \leq i < k$ and $S[k+2j] = o_j$, $S[k+2j+1] = f^s(o_j)$ for
0 \leq j < m. The sequence \( S' \) is defined by \( S'[i] = S[i] \) if \( \text{type}(S[i]) \not\subseteq \text{Object} \) and \( S'[i] = \pi(S[i]) \) otherwise.

It is easily checked that the properties of the partial isomorphism \( \pi \) from Definition 30 imply items 1 to 8 of Lemma 27, as desired. \( \square \)

The next definition is the variation of Definition 14 now using reference set expressions in place of observation expressions. When using observation expressions the isomorphisms \( \pi^1, \pi^2 \) are uniquely determined, if they exists. When using reference set expressions this is not the case. This explains the quantifications for any partial isomorphisms \( \pi^1 \) there is a partial isomorphism \( \pi^2 \) in the following definition.

**Definition 31 (Information flow using reference sets).**

Let \( \alpha \) be a program and \( R^1 = R_{\pi^1}^{1_1},...,1_n \{L^1\}, R^2 = R_{\pi^2}^{2_1},...,2_n \{L^2\} \) expressions of type \( \text{refSet} \).

Program \( \alpha \) allows information to flow only from \( R^1 \) to \( R^2 \) when started in \( s_1 \), denoted by \( \text{flow}_{rs}(s_1, \alpha, R^1, R^2) \)

iff

for all states \( s'_1, s_2, s'_2 \) such that

\( \alpha \) started in \( s_1 \) terminates in \( s_2 \) and

\( \alpha \) started in \( s'_1 \) terminates in \( s'_2 \), we have

for any partial isomorphism \( \pi^1 \) with \( \text{agree}^r_{rs}(R^1, s_1, s'_1, \pi^1) \)

there is a partial isomorphism \( \pi^2 \) with \( \text{agree}^r_{rs}(R^2, s_2, s'_2, \pi^2) \)

and \( \pi^2 \) extends \( \pi^1 \)

where \( \pi^2 \) is said to extend \( \pi^1 \) if

\( \pi^2(o) = \pi^1(o) \) for all \( o \in \text{obj}^{i^1}(R_1) \cap \text{obj}^{i^2}(R_2) \) with \( \text{created}^{i^1}(o) = \text{tt} \).

We extend \( \text{JAADL} \) by a new three-place modal operator \( \text{flow}_{rs}(\cdot, \cdot, \cdot) \) that expects a program as its first and reference set expressions as its second and third arguments. Its semantics is defined, for all states \( s \), by

\[ s \models \text{flow}_{rs}(\alpha, R_1, R_2) \quad \text{iff} \quad \text{flow}_{rs}(s, \alpha, R_1, R_2) \text{ holds} \]

Let us look at a few simple examples of expressions of type \( \text{refSet} \).

**Example 10.**

In the following expressions \( v \) is a local variable, \( e_1, e_2 \) are expressions of type \( C_1, C_2, f, f_1, f_2 \) are fields defined in the class of \( \text{this} \), \( C_1 \) and \( C_2 \) respectively. Furthermore, \( a \) is an expression of array type, and \( i_1 < i_2 \) are integers.

\[ R^1_{cz} = \{ v \}, \text{singleton}(\text{this}, f) \]
\[ R^2_{cz} = \{ e_1, e_2 \}, \text{singleton}(f_1, f) \]
\[ R^3_{cz} = \{ a \}, \text{arrayRange}(a, i_1, i_2) \]

Related observation expressions could be: (To reduce the length of expressions we will write \( sC \) for \( \text{seqConcat} \) and \( sqt \) for \( \text{seqSingleton} \))

\[ R^1_{cz} = sC(sC(v, \text{sqt}(\text{this})), \text{sqt}(f)) \]

or short \( (v, \text{this}, f) \)

\[ R^2_{cz} = sC(sC(sqt(e_1), sqt(e_1, f)), \text{sqt}(f_1)), \text{sqt}(f_2)) \]

or \( (e_1, e_2, f_1, f_2) \)

\[ R^3_{cz} = \text{seq}_{de}[i]v(i_1, i_2, a[i1]) \]

or \( (a[i], a[i2 - 1]) \)

We observe that \( \text{agree}^r_{rs}(R^1_{cz}, s_1, s'_1) \) iff \( \text{agree}(R_1 - cz, s_1, s'_1) \). For the other two example expressions this is not the case.
Theorem 2. Let $\alpha$ be a program, and $R_1 = R_{v_1^1},...,v_1^k (L_1)$, $R_2 = R_{v_2^1},...,v_2^k (L_2)$ arbitrary reference set expressions.

There is a formula $\phi_{o,R_1,R_2}^s$ in JAVADL making use of self-composition such that:

$$s_1 \models \phi_{o,R_1,R_2}^s \iff \text{flow}_{rs}(s_1, \alpha, R_1, R_2).$$

Proof. The proof greatly parallels the proof of Theorems 1 and 3. Nevertheless, we will repeat here the whole argument. Thus, this proof is self-contained, the reader is not required to have read the proof of Theorem 1 or 3 before.

The proof consists of a constructive definition of the formula $\phi_{o,R_1,R_2}^s$.

We will explain the construction of $\phi_{o,R_1,R_2}^s$ top down. The property to be formalized requires quantification over states. According to Definition 5 a state $s$ is determined by the value of the heap $h^s$ in $s$ and the values of the (finitely many) program variables $a^s$ in $s$. We can directly quantify over heaps $h$ and refer to the value of a field $f$ of type $C$ for object $o$ referenced by expression $e$ as $\text{select}_C(h,e,f)$. We cannot directly quantify over program variables, as opposed to quantifying over the values of program variables, which is perfectly possible. Thus we use quantifiers $\forall x, \exists y$ over the type domain of the variable and assign $x$ to $a$ via an update $a := x$. There are four states involved, the two pre-states $s_1$, $s'_1$ and the post-states $s_2$, $s'_2$. Correspondingly, there will be, for every program variable $v$, four universally quantifier variables $v$, $v'$, $v^2$, $(v^2)'$ of appropriate type representing the values of $v$ in states $s_1$, $s'_1$, $s_2$, $s'_2$. There are some program variables that make only sense in pre-states, e.g., $\text{this}$, and variables that make only sense in post-state, e.g., $\text{result}$. There will be only two logical variables that supply values to them instead of four.

The main challenge in the definition of $\phi_{o,R_1,R_2}^s$ is that we need to express the existence of a partial isomorphisms. On the face of it this is a second order property. One could hope that the higher order aspects of dynamic logic could be harnessed for this purpose. After a short period of preliminary exploration we decided not to pursue this avenue since the outcome would be - we feared - rather circuitous and cumbersome to deal with. So, we are left with the resources of typed first-order logic. The existence of a bijective mapping between two sequences of objects, where the $i$-th element of the source sequence is mapped to the $i$-th element of the target sequence can easily be formulated: the sequences should be of equal length and if the objects at two positions in the source sequence coincide the objects at the corresponding positions in the target sequence also coincide. It remains to code the objects in $\text{obj}^s(R)$ by appropriate sequences. Groundwork for this has already be laid by Lemmas 27 and 28. This idea can be made to work since JAVADL provides the data type $\text{Seq}$. In particular, quantification over sequences is possible. This motivates for the moment the occurence of the variables $S$, $S'$, $S_2$, and $S'_2$ of type sequence in the formula to follow.

This leads to the following schematic form of $\phi_{o,R_1,R_2}^s$:

$$\phi_{o,R_1,R_2}^s \equiv \forall \text{Heap } h'_1, h_2, h'_2 \forall T o \forall T_i r, r', ... v', v^2, (v^2)' ... \forall \text{Seq } S, S' \forall \text{Int } n \exists \text{Seq } S_2, S'_2 \exists \text{Int } n_2 (...(\text{Agree}_{\text{pre}} \land (\langle a \rangle \text{save}(s_2) \land \text{in}(s'_1) \langle a \rangle \text{save}(s'_2) \implies (\text{Agree}_{\text{post}} \land \text{Ext})...)...})

To maintain readability we have used suggestive abbreviations:

1. $\{\text{in } s'_1\} (\langle a \rangle)$ signals that an update $[\text{heap} := h'_1 \mid \text{this} := o' \mid ... a_i := v' ...]$ is placed before the modal operator. The $a_i$ cover all relevant parameters and local variables.
2. The construct $\text{save}(s_2)$ abbreviates a conjunction of equations $h_2 = \text{heap}$, $r = \text{result}$, $..., v^2 = a_i$, ....
3. Analogously, save \{s_2^i\} stands for the primed version \(h'_2 = \text{heap}, r' = \text{result}\), \(i, \ldots, (v^2)' = a_i, \ldots\).

4. The shorthand \{(in s_2^i)\} E in front of a formula is resolved by (a) prefixing every occurrence of a heap dependent expression \(e\) with the update \{\text{heap} := h'_2\} and (b) every primed expression \(e'\) with \(\text{heap} := h'_2\).

5. The same applies to \{(in s_1^i)\} E. Note, there is no \{(in s_1)\}, and nor quantified variables \(a, v\) since the whole formula \(\phi_{\alpha,R_1,R_2}^a\) is evaluated in state \(s_1\).

In the following we will also use the notation \(R_1', R_2', (R_2)^2\)' for the terms obtained from \(R_i\) by replacing each state dependent designator \(v\) by \(v', v^2, (v^2)'\) respectively. Technically, these substitutions are effected by prefixing \(R_i\) with an appropriate update.

We now supply the definitions of the abbreviations used above. In the following formulas \(T\) denotes the set of all types occurring in program \(\alpha\). We assume that \(T\) is finite. We point out that the formulas \(\text{Agree}_{\text{pre}}\) and \(\text{Agree}_{\text{post}}\) formalize the 8 requirements of Lemma 27. In fact, when writing Lemma 27 we had already taken care, that only requirements be imposed that can be formalized in JAVADL.

\[
\begin{align*}
\text{Agree}_{\text{pre}} & \equiv \forall \text{Int } i (0 \leq i < k_1 \rightarrow (S[i] = v_i \land S'[i] = v'_i \land S[i] = S'[i])) \land \forall \text{Int } j (k_2 \leq j < (n - 1) \rightarrow \\
& \quad \text{instanceObject}(S_2[j]) \land \text{instanceObject}(S_2'[j]) \land \\
& \quad \exists \text{Field } f(S_2[j] + 1) = \text{select}_{\text{any}}(h_2, S_2[j], f) \land \\
& \quad S_2'[j] + 1 = \text{select}_{\text{any}}(h'_2, S_2'[j], f)) \land \\
& \forall \text{Object } o, f, L_2 \rightarrow \exists \text{Int } j (k_3 \leq j < n - 1 \land \\
& \quad S_2[j] = \text{null} \land S_2'[j] = \text{null} \
\end{align*}
\]

\(\text{Agree}_{\text{post}}\) is - roughly speaking - the same as \(\text{Agree}_{\text{pre}}\) with \(S, S'\) replaced by \(S_2, S_2'\):

\[
\begin{align*}
\text{Agree}_{\text{post}} & \equiv \forall \text{Int } i (0 \leq i < k_2 \rightarrow (S_2[i] = v_i \land S_2'[i] = (v^2)_i \land S_2[i] = S_2'[i])) \land \\
& \forall \text{Int } j (k_1 \leq j < (n_2 - 1) \rightarrow \\
& \quad \text{instanceObject}(S_2[j]) \land \text{instanceObject}(S_2'[j]) \land \\
& \quad \exists \text{Field } f(S_2[j] + 1) = \text{select}_{\text{any}}(\text{heap}, S_2[j], f) \land \\
& \quad S_2'[j] + 1 = \text{select}_{\text{any}}(h'_1, S_2'[j], f)) \land \\
& \forall \text{Object } o, f, L_1 \rightarrow \exists \text{Int } j (k_3 \leq j < n_2 - 1 \land \\
& \quad S_2[j] = \text{null} \land S_2'[j] = \text{null} \
\end{align*}
\]
Ext ≡ ∀Int ∀Int j(0 ≤ i < n ∧ 0 ≤ j < n2 →
S[i] = S2[j] → S′[i] = S′2[j])

It remains to show that this definition does the job.
The first part proves s1 ⊨ φo, R1, R2 ⇒ flow∗(s1, α, R1, R2).
So let us assume s1 ⊨ φo, R1, R2. To prove flow∗(s1, α, R1, R2) fix states
s1′, s2′ such that α started in s1 terminates in s2, α started in s1′ terminates
in s2′, and agree(R1, s1, s1′, π1). We need to show that there exists a mapping π2
such that agree(R2, s2′, s2, π2) and π2 extends π1.

We instantiate the universally quantified variables by their evaluations in
state s1′, s2′, s2 respectively, i.e., β(v1′) = (v1′)s1′, β(v2′) = (v2′)s2, β((v2′)′) = (v2′)s2,
β(k′i) = heapi, etc.

By Lemma 28 and agree(R1, s1, s1′, π1) there is an instantiation β(n) and
there are sequences β(S), β(S′) such that (s1, β) ⊨ Agreeprec.

By definition of β we also have (s1, β) ⊨ (α)save{s2} ∧ in{s1} (α)save{s2′}

Thus s1 ⊨ φo, R1, R2 implies that there are instantiations β(S2) and β(S′2)
such that (s1, β) ⊨ Agreepost ∧ Ext.

(s1, β) ⊨ Agreepost yields by Lemma 27 an isomorphism π2 such that agree(R2, s2, s2′, π2).

Finally, (s1, β) ⊨ Ext implies that π2 is an extention of π1. This, depends on
the specific way the isomorphisms are defined.

We now turn to the second part flow∗(s1, α, R1, R2) ⇒ s1 ⊨ φo, R1, R2. Let β
be an arbitrary instantiation of the universally quantified variables in the prefix
of φo, R1, R2 and assume (s1, β) ⊨ Agreeprec ∧ (α)save{s2} ∧ in{s1} (α)save{s2′}.
Otherwise, (s1, β) ⊨ φo, R1, R2 is vacuously true. (s1, β) ⊨ Agreeprec and Lemma
27 imply the existence of a partial isomorphism π1 such that agree(R1, s1, s1′, π1).
Now, flow∗(s1, α, R1, R2) says that there is an isomorphism π2 extending π1
such that agree(R2, s2′, s2, π2). Lemma 28 provides instantiations of the existen-
tially quantified variables β(S2) and β(S′2) such that (s1, β) ⊨ Agreepost. Since
π2 extends π1 we also get (s1, β) ⊨ Ext.

B.1 A Simplified Version using Reference Sets

The definition of φo,R1,R2 in the proof of Theorem 2 uses quantifications over
location sets. This complicates the derivation of this formula. In this subsection
we establish Theorem 3 which is weaker than Theorem 2 in that it only applies for
constructive reference set expressions R2 and provides only a sufficient condition
φo,R1,R2 for flowrs (s1, α, R1, R2). But, φo,R1,R2 does not involve quantification
over location sets.

Lemma 29. Let R be a reference set expression.
If agree[s, R, s′, π] and ρ is an automorphism on D
then also agree[s, R, s, ρ(s′), ρ ◦ π].

Proof. From the assumption agree[s, R, s′, π] we get by Definition 30 that π
is a bijective mapping from obj∗(R) onto obj∗(R) satisfying:

1. π is type preserving,
    i.e. o ∈ T ∗ ⇐⇒ π(o) ∈ T ∗ for all o ∈ obj∗(R) and all types T.
For objects o ∈ obj∗(R) of array type o.length′ = π0(o).lengths′ is required
in addition.
2. s(v) = s′(v) for all v ∈ V with type(v) = Boolean or type(v) = Int;
3. π(s(v)) = s′(v) for all v ∈ V with type(v) ⊆ Object;
4. f∗(o) = f∗(π(o)) for all (o, f) ∈ L∗ where the type(f) ⊆ Object;
5. π(f∗(o)) = f∗(π(o)) for all (o, f) ∈ L∗ with type(f) ⊆ Object;

67
Then

1. \( \rho \circ \pi \) is type preserving since \( \rho \) is an isomorphism.
2. \( s(v) = \rho(s'(v)) = \rho(s)(v) \) for all \( v \in V \) with \( \text{type}(v) = \text{Boolean} \) or \( \text{type}(v) = \text{Int} \), since \( \rho \) is the identity on basic data types. Furthermore, we have used the terminology \( \rho(s') \) from Definition 11.
3. \( (\rho \circ \pi)(s(v)) = \rho(s)(v) \) for all \( v \in V \) with \( \text{type}(v) \subseteq \text{Object} \) by the laws of equality.
4. \( f^* (o) = \rho(f^*(o)) = f^\rho(s')(\rho \circ \pi(o)) \) for all \( (o, f) \in L^s \) where the \( \text{type}(f) \not\subseteq \text{Object} \) since \( \rho \) is the identity on basic data types and Lemma 1.
5. \( \rho \circ \pi(f^*(o)) = f^\rho(s') (\rho \circ \pi(o)) \) for all \( (o, f) \in L^s \) with \( \text{type}(f) \subset \text{Object} \) by the laws of equality and again Lemma 1.
6. \( \rho \circ \pi(L^s) = \{(\rho \circ \pi(o), f) \mid (o, f) \in L^s\} = L^{\rho(s')} \).

To see this we first note that \( \rho \circ \pi(L^s) = \{(\rho \circ \pi(o), f) \mid (o, f) \in L^s\} \) is true by the way \( \rho \) is defined in type \( \text{LocSet} \). \( \rho \circ \pi(L^s) = \rho(L^s) \) follows from the assumption and \( \rho(L^s) = L^{\rho(s')} \) from Lemma 1.

7. If \( \rho \in \text{obj}(R) \) then \( \rho \circ \pi(\text{null}) = \text{null} \), since \( \rho(\text{null}) = \text{null} \) for any isomorphism.
8. For all \( o \in \text{obj}(R) \): \( \text{created}^*(o) = \text{tt} \Leftrightarrow \text{created}^\rho(\rho \circ \pi(o)) = \text{tt} \) follows from Lemma 1.

This is exactly the definition of \( \text{agree}_{rs}(R, s, \rho(s'), \rho \circ \pi) \).

**Definition 32 (Simple Information flow using reference sets).**

Let \( \alpha \) be a program and \( R^1 = R_{s_1^1, \ldots, s_k^1}^1(L^1) \), \( R^2 = R_{s_2^1, \ldots, s_k^2}^2(L^2) \) expressions of type \( \text{refSet} \).

Program \( \alpha \) allows simple information flow only from \( R^1 \) to \( R^2 \) when started in \( s_1 \), denoted by \( \text{flow}^\rho_{rs}(s_1, \alpha, R^1, R^2) \)

if

- for all states \( s_1^1, s_2^1, s_2^2 \) such that
  - \( \alpha \) started in \( s_1 \) terminates in \( s_2^1 \) and
  - \( \alpha \) started in \( s_1^1 \) terminates in \( s_2^2 \),
we have

  if \( \text{agree}_{rs}(R^1, s_1, s_1^1, \text{id}) \)

then there is a partial isomorphism \( \pi^2 \) with \( \text{agree}_{rs}(R^2, s_2^1, s_2^2, \pi^2) \)

and \( \pi^2 \) extends \( \text{id} \).

The statement of the following lemma parallels that of Lemma 9.

**Lemma 30.** For all programs \( \alpha \), any two reference expressions \( R_1 \) and \( R_2 \), and any state \( s_1 \)

\[ \text{flow}^\rho_{rs}(s_1, \alpha, R_1, R_2) \Rightarrow \text{flow}^\rho_{rs}(s_1, \alpha, R_1, R_2) \]

Since the reverse implication is obviously true Lemma 30 entails that \( \text{flow}^\rho_{rs} \) and \( \text{flow}^*_{rs} \) are equivalent.

**Proof.** The proof follows very closely the proof of Lemma 9 with the minor difference that it suffices to show the existence of isomorphism \( \pi^2 \).

To prove \( \text{flow}^\rho_{rs}(s_1, \alpha, R_1, R_2) \) we fix, in addition to \( s_1 \), states \( s_1^1, s_2, s_2^2 \) such that
- \( \alpha \) started in \( s_1 \) terminates in \( s_2 \) and
- \( \alpha \) started in \( s_1^1 \) terminates in \( s_2^2 \),
and assume \( \text{agree}_{rs}(R_1, s_1, s_1^1, \pi^1) \). We need to show that there exists \( \pi^2 \) with \( \text{agree}_{rs}(R_2, s_2^1, s_2^2, \pi^2) \) and \( \pi^2 \) extends \( \pi^1 \).

68
By Lemma 4 there is an automorphism $\rho$ on $D$ extending $(\pi^1)^{-1}$, i.e., the inverse of $\pi^1$. From agree$_{rs}(R_1, s_1, s_1', \pi^1)$ we conclude agree$_{rs}(R_1, s_1, \rho(s_1'), \rho \circ \pi^1)$ using Lemma 29. Since $\rho$ extends $(\pi^1)^{-1}$ we have agree$_{rs}(R_1, s_1, \rho(s_1'), \text{id})$. By Lemma 5 there is a state $s'_3$ such that $\alpha$ started in $\rho(s'_1)$ terminates in $s'_3$. This enables us to make use of the assumption flow$_{rs}(s_1, \alpha, R_1, R_2)$ and conclude that there exists a partial isomorphism $\pi^3$ satisfying agree$_{rs}(R_2, s_2, s_3', \pi^3)$ and extending the identity, i.e., $\pi^3(o) = o$ for all $o \in \text{obj}^s(R_1) \cap \text{obj}^s(R_2)$.

Applying Lemma 5 to the inverse isomorphism $\rho^{-1}$ and the situation that $\alpha$ started in $\rho(s'_1)$ terminates in $s'_3$, we obtain an automorphism $\rho'$ such that $\alpha$ started in $\rho^{-1}(\rho(s'_1)) = s'_1$ terminates in $\rho'(s'_3)$ and $\rho'$ coincides with $\rho^{-1}$ on all objects in $E = \{o \in \text{Object}^D \mid \text{created}^{\rho(s'_1)}(o) = tt\}$.

Again using Lemma 29, this time for the isomorphism $\rho'$, we obtain from agree$_{rs}(R_2, s_2, s'_3, \pi^3)$ also agree$_{rs}(R_2, s_2, \rho'(s'_3), \rho' \circ \pi^3)$. Since $\alpha$ is a deterministic program and we have already defined $s'_2$ to be the final state of $\alpha$ when started in $s_2$ we get $s'_2 = \rho'(s'_3)$ and thus agree$_{rs}(R_2, s_2, s'_2, \rho' \circ \pi^3)$.

It remains to convince ourselves that $\rho' \circ \pi^3$ extends $\pi^1$, i.e., for every $o \in \text{obj}^s(R_1) \cap \text{obj}^s(R_2)$ with created$^{\pi^1}(o) = tt$ we need to show $\rho' \circ \pi^3(o) = \pi^1(o)$. By the definition of isomorphic states we obtain from created$^{\pi^1}(o) = tt$ also created$^{\rho(s'_1)}(o) = tt$. Thus we can infer $\rho'(o) = \rho^{-1}(o)$ and by choice of $\rho$ further $\rho^{-1}(o) = \pi^1(o)$, as desired. $\square$

**Definition 33.** The set CLE of constructive location set expressions is a subset of all expressions of type $\text{LocSet}$ defined by the following inductive definition.

1. $\emptyset$ is in CLE.
2. If $e$ is an expression of type $C$ and $f$ is a field in $C$ with type$(f), \text{type}(e) \neq \text{LocSet}$ then singleton$(e, f)$ is in CLE.
3. If $a$ is an expression of array type, and $t_1$, $t_2$ are integer expressions then arrayRange$(a, t_1, t_2)$ is in CLE.
4. For $e_1, e_2$ in CLE also $e_1 \cup e_2$ is in CLE.
5. For $e$ in CLE also infiniteUnion$(\{iv\}(e))$ is in CLE provided that there is an integer expression $t$ not containing $iv$ such that infiniteUnion$(\{iv\}(e)) = \text{infiniteUnion}(\{iv\}(t < t \text{ then } e \text{ else } \emptyset))$ is universally valid. We will write infiniteUnion$(\{iv < t\}(e))$ in this case.

**Lemma 31.** For every CLE expression $e$ there is an expression $sq_e$ and an expression $t_e$ of type $\text{Int}$ such that for all states $s$, objects $o$ and fields $f$

$$(o, f) \in e^s \iff \text{there is } i, 0 \leq i < t_e^s \text{ such that } sq_e[i] = o \text{ and } sq_e[i + 1] = f^s(o).$$

Here $sq_e[i]$ abbreviates seqGet$_{\text{Any}}(sq_e, i)$.

**Proof.** We set $t_0 = 0$, while $sq_0$ is arbitrary, e.g., $sq_0 = \text{null}$. The claimed correspondence between $\emptyset$ and $sq_0$ and $t_0$ is trivially satisfied.

Also for $s_{\text{singleton}(e, f)} = \langle eo, eo, f \rangle$ and $t_{\text{singleton}(e, f)} = 2$ the claim of the lemma is obviously true.

The shorthand notation $\langle \ldots \rangle$ has been introduced in the paragraph following Definition 16 on page 21.

For $e = \text{arrayRange}(a, b_1, b_2)$ we set $t_e = 2 * (b_2 - b_1)$ and $sq_e = \text{seqDef}\{\{iv\}(0, t_e, \text{if even}(iv) \text{ then } a[b_1 + (iv/2)]\}$. Remembering the semantics of arrayRange (Item 11 of Definition 4 on page 6) it is again easily seen that the claim of the lemma is satisfied.

Now assume that for $e_1, e_2$ expressions $sq_{e_1}, t_{e_1}, q_{e_2}$, and $t_{e_2}$ satisfying the claim of the lemma have already been found. We set $t_e = t_{e_1 \cup e_2} = t_{e_1} + t_{e_2}$ and $sq_{e_1 \cup e_2} = \text{seqDef}\{\{iv\}(0, t_e, \text{if } iv < t_e \text{ then } sq_{e_1}[iv] \text{ else } sq_{e_1}[t_{e_1} + iv]$. 

69
The last case in the inductive definition is \( e = \text{infiniteUnion}\{iv < t\}(e_0) \). We assume that \( sq_{e_0} \) and \( t_{e_0} \) for \( e_0 \) have been found satisfying the claim of the lemma. Typically, both expressions contain \( iv \) as a free variable. For different assignments of \( iv \) the expressions \( sq_{e_0} \) will evaluate to sequences of differing length. Let \( t_m = \max_{0 \leq i < t_{e_0}}t_{e_0} \) and
\[
sq = \begin{cases} 0 & \text{if } 0 \leq iv < t_{e_0} \text{ else if } even(iv) \text{ then } sq_{e_0}[0] \text{ else } sq_{e_0}[1]. \end{cases}
\]
Thus if in state \( s \) with variable assignment \( \beta \) we have \( sq_{e_0}^{(s,\beta)} = \langle a_0, a_1 \ldots a_k \rangle \) for \( k = t_{e_0}^{(s,\beta)} \) then \( sq^{(s,\beta)} \) is the sequence \( \langle a_1, \ldots, a_k, a_0, a_1, \ldots, a_0, a_1 \rangle \) of length \( t_m \). Since \( iv \) does no longer occur free in \( t_m \) the evaluation \( t_m^iv \) is independent of \( \beta(iv) \). Also \( e_0, sq, t_m \) still satisfy the claim of the lemma, since repetition in \( sq \) do not hurt.

We set \( t_e = t \ast t_m \) and \( sq_e = sq \mod \{iv(0, t_e, sq(iv/t_m + iv[mod(t_m, iv)])]) \}

Here \( sq(x/iv) \) is the term arising from \( sq \) by replacing every occurrence of the variable \( iv \) by \( x \). In the present case \( x \) is the integer division term \( iv/t_m \). Furthermore, \( mod(t_m, iv) \) is the remainder of \( iv \) in the division by \( t_m \). Thus \( iv = (iv/t_m) + mod(t_m, iv) \). It is now not hard to see that \( e, sq_e, \) and \( t_e \) satisfy the claim of the lemma. \( \square \)

Let \( R = \{\{v_1, \ldots, v_k\}, L\} \) be a reference set expression. We establish the following notation to be used in the next lemma:

\[
\begin{align*}
\text{obj}^s(iv) &= \{v_i \mid \text{type}(v_i) \subseteq \text{Object}, 1 \leq i \leq k\} \cup \\
\text{obj}^s(L) &= \{o \mid (o, f) \in R^s\} \cup \{f^s(o) \mid (o, f) \in R^s, \text{type}(f) \subseteq \text{Object}\}
\end{align*}
\]

Thus
\[
\text{obj}^s(R) = \text{obj}^s(iv) \cup \text{obj}^s(L)
\]

**Lemma 32.** Let \( e \) be a CLE expression and \( sq_e, t_e \) as provided by Lemma 31 and \( s, s' \) some states

If the mapping \( \pi \) defined by \( \pi(sq^e_{t_e}[i]) = sq^{s'}_{t_e}[i] \) for \( 0 \leq i < t_{e_0} \) is bijective then it is a partial isomorphism from \( \text{obj}^s(L_2) \) onto \( \text{obj}^{s'}(L_2) \).

**Proof.** By definition of \( \text{obj}^s(L_2) \), \( \text{obj}^{s'}(L_2) \) and definition of \( \pi \) and the correspondence between \( L_2 \) and \( sq_{t_e} \), \( t_e \) from Lemma 31 we see that \( \pi \) is a bijection from \( \text{obj}^s(L_2) \) onto \( \text{obj}^{s'}(L_2) \). To see that also the isomorphism property is satisfied consider \( (o, f) \in L_2^s \). By Lemma 31 there is \( i \) such that \( (sq_{t_e}[i])^s = o \) and \( (sq_{t_e}[i + 1])^s = f^s(o) \). By definition of \( \pi \) we have \( \pi(f^s(o)) = \pi(sq_{t_e}[i + 1])^s = sq^s_{t_e}[i + 1])^s = f^s(\pi(sq_{t_e}[i])^s) = f^s(\pi(o)) \). \( \square \)

**Theorem 3.** Let \( \alpha \) be a program, and let \( R_1 = R_{v_1}^{s_1}, \ldots, v_{k_1}^{s_1}(L_1) \) be an arbitrary reference set expression and \( R_2 = R_{v_2}^{s_2}, \ldots, v_{k_2}^{s_2}(L_2) \) a reference set expression with \( L_2 \) in CLE.

There is a formula \( \phi^{rs}_{n, R_1, R_2} \) in JAVA DL making use of self-composition such that:

\[
s_1 \models \phi^{rs}_{n, R_1, R_2} \Rightarrow \text{flow}_{rs}(s_1, \alpha, R_1, R_2).
\]

**Proof.** By Lemma 30 it suffices to find \( \phi^{rs}_{n, R_1, R_2} \) such that

\[
s_1 \models \phi^{rs}_{n, R_1, R_2} \Rightarrow \text{flow}_{rs}(s_1, \alpha, R_1, R_2).
\]

The proof greatly parallels the proof of Theorem 1. Nevertheless, we will repeat here the whole argument. Thus, this proof is self-contained, the reader is not required to have read the proof of Theorem 1 before.

The proof consists of a constructive definition of the formula \( \phi^{rs}_{n, R_1, R_2} \).

We will explain the construction of \( \phi^{rs}_{n, R_1, R_2} \) top down. The property to be formalized requires quantification over states. According to Definition 5 a state \( s \) is determined by the value of the heap \( h^s \) in \( s \) and the values of the (finitely
Thus we use quantifiers to quantifying over the values of program variables, which is perfectly possible. We cannot directly quantify over program variables, as opposed to quantifying over the values of program variables, which is perfectly possible. Thus we use quantifiers $\forall x, \exists x$ over the type domain of the variable and assign $x$ to $a$ via an update $a := x$. There are four states involved, the two pre-states $s_1$, $s_1'$ and the post-states $s_2$, $s_2'$. Correspondingly, there will be, for every program variable $v$, four universally quantifier variables $v, v', v''$, $(v'^2)'$ of appropriate type representing the values of $v$ in states $s_1, s_1', s_2, s_2'$. There are some program variables that make only sense in pre-states, e.g., this, and variables that make only sense in post-state, e.g., result. There will be only two logical variables that supply values to them instead of four. This leads to the following schematic form of $\phi_{a,R_1,R_2}$:

$$\phi_{a,R_1,R_2} \equiv \forall \text{Heap } h_1', h_2, h_2' \forall T \forall T_r, r', r'' \ldots \forall v', v'' (v'^2)' \ldots$$

$$(\text{Agree}_{\text{pre}} \land \langle \alpha \rangle \text{save}\{s_2\} \land \text{in}\{s_1'\} \langle \alpha \rangle \text{save}\{s_2'\}$$

$$(\rightarrow (\text{Agree}_{\text{post}} \land \text{Ext}))$$

To maintain readability we have used suggestive abbreviations:

1. $\{\text{in } s_1'\} \langle \alpha \rangle$ signals that an update $\{\text{heap} := h_1' \mid \text{this} := o' \mid \ldots a_i := v' \ldots\}$ is placed before the modal operator. The $a_i$ cover all relevant parameters and local variables.
2. The construct $\text{save}\{s_2\}$ abbreviates a conjunction of equations $h_2 = \text{heap}$, $r = \text{result}$, $\ldots$, $v^2 = a_i$, $\ldots$.
3. Analogously, $\text{save}\{s_2'\}$ stands for the primed version $h_2' = \text{heap}$, $r' = \text{result}$, $\ldots$, $(v'^2)' = a_i$, $\ldots$.
4. The shorthand $\{\text{in } s_2\} \{\text{in } s_1\} E$ in front of a formula is resolved by (a) prefixing every occurrence of a heap dependent expression $e$ with the update $\{\text{heap} := h_2\}$ and (b) every primed expression $e'$ with $\{\text{heap} := h_2'\}$.
5. The same applies to $\{\text{in } s_1\} E$. Note, there is no $\{s_1\}$, and nor quantified variables $v, v'$ since the whole formula $\phi_{a,R_1,R_2}$ is evaluated in state $s_1$.

In the following we will also use the notation $R_1', R_2', (R_2')'$ for the terms obtained from $R_i$ by replacing each state dependent designator $v$ by $v'$, $v''$, $(v'^2)'$ respectively. Technically, these substitutions are effected by prefixing $R_i$ with an appropriate update.

We now supply the definitions of the abbreviations used above:

$$\text{Agree}_{\text{pre}} \equiv \bigwedge_{1 \leq i \leq k_1} v_i^1 \equiv (v_i^1)' \land$$

$$\forall \forall \forall \forall \forall \forall f ((o, f) \in L_1 \rightarrow \text{select}_{\text{Any}}(h_i, o, f) \equiv \text{select}_{\text{Any}}(h_i', o, f)) \land$$

$$\forall \forall \forall \forall \forall \forall f ((o, f) \in L_1 \leftrightarrow (o, f) \in L_1')$$

For the next definition we denote by $sq_2, t_2$ the expressions of type $\text{Seq}$ and $\text{Ind}$ respectively that exists by Lemma 31 for $L_2$.

$$\text{Agree}_{\text{post}} \equiv t_2 \equiv t_2'$$

$$\bigwedge_{1 \leq i \leq k_2, \text{type}(v_i^2) \subseteq \text{Object}} (v_i^2 \equiv (v_i^2)') \land$$

$$\bigwedge_{1 \leq i < j \leq k_2, \text{type}(v_j^2) \subseteq \text{Object}} ((v_j^2 \equiv v_j^2) \leftrightarrow ((v_j^2)' \equiv (v_j^2)')) \land$$

$$\forall i, j (0 \leq i < j < t_2 \rightarrow (sq_2[i] \equiv sq_2[j] \leftrightarrow sq_2[j] \equiv (sq_2[j])) \land$$

$$\bigwedge_{1 \leq i \leq k_2, \text{type}(v_j^2) \subseteq \text{Object}} \forall j (0 \leq j < t_2 \rightarrow$$

$$(v_j^2 \equiv sq_2[j] \leftrightarrow (v_j^2)' \equiv (sq_2[j]))$$

$$\text{Ext} \equiv \bigwedge_{1 \leq i \leq k_2} \forall j (0 \leq j < t_2 \land v_j^2 \equiv sq_2[j] \rightarrow (v_j^2)' \equiv (sq_2[j]))$$
It remains to show that this definition does the job.

So let us assume \( s_1 \models \phi \). Using Lemma 26, we may agree that \( s_1 \) terminates in \( s_2 \), \( s \) started in \( s_1 \) terminates in \( s_2 \), and agree \((R_1, s_1, s_1', id)\). We need to show that there exists a mapping \( \pi^2 \) such that agree \((R^2, s_2, s_2', \pi^2)\) and \( \pi^2 \) extends id.

We instantiate the universally quantified variables by their evaluations in state \( s_1, s_2, s_2' \) respectively, i.e., \( \beta(v_1) = v_1^s, \beta(v_2) = v_2^s, \beta((v_2')^s) = v_2'^s, \beta(h_1) = \text{heap}^s, \) etc.

Now agree \((R_1, s_1, s_1', id)\) implies \((s_1, \beta) \models Agree_{\text{pre}}\), as can be easily seen using Lemma 26.

By definition of \( \beta \) we also have \((s_1, \beta) \models (\alpha)\text{save}\{s_2\} \land \text{in}\{s_1'\}(\alpha)\text{save}\{s_2'\}\)

Thus \( s_1 \models \phi_{\alpha,R_1,R_2} \) implies \((s_1, \beta) \models Agree_{\text{post}} \land \text{Ext}\).

We define \( \pi^2 \) by \( \pi^2((s_{q_2,j})^s) = (s_{q_2,j})^s \) for \( 0 \leq j < t_{q_2}^s \). Now, \((s_1, \beta) \models Agree_{\text{post}}\) due to line 1 and line 4 in the definition of \( Agree_{\text{post}}\) implies that \( \pi^2 \) thus defined is a bijection. Lemma 32 says that \( \pi^2 \) is a partial isomorphism from \( \text{obj}^s(L_2) \) onto \( \text{obj}^s(L_2) \). We extend \( \pi^2 \) to a mapping from \( \text{obj}^s(L_2) \) onto \( \text{obj}^s(L_2) \) by \( \pi^2((v_2')^s) = v_2'^s \). Line 2 and 3 in the definition of \( Agree_{\text{post}}\) guarantee that this is a bijection and line 5 makes sure that this definition is compatible with \( \pi^2 \) defined on \( \text{obj}^s(L_2) \). Altogether, we see that \( \pi^2 \) is a partial isomorphism from \( \text{obj}^s(R_2) \) onto \( \text{obj}^s(R_2) \) and agree \((R^2, s_2, s_2', \pi^2)\) is true.

Finally, \((s_1, \beta) \models \text{Ext}\) implies that \( \pi^2 \) is an extension of the identity. We may thus conclude \( flow^s(s_1, \alpha, R_1, R_2) \) as desired. \( \square \)

References


