Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2013,10
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

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

Fakultät für **Informatik**

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 – RS^3".

Amtoft and Banerjee [2] were the first to give semantical definitions of information 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 `m1()` and `m2()` both do not have an information flow, then `m1();m2()` does not have an information flow." Moreover, self-composition can be "overkill"; the program "`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 propagation 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 functional 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, JavaDL as it is used in the KeY system. An extensive account of the semantics of the data type *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 Java programs. It also contains an informal summary of the attacker model we have in mind. This report considers two ways to formalise the notation of an observation in 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 independently 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*.

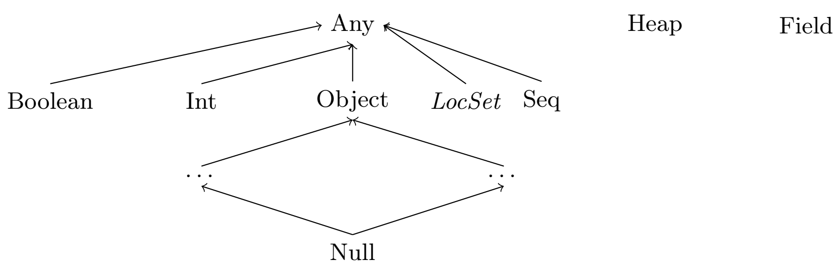

Figure 1. The JavaDL Type Hierarchy

The vocabulary Σ_{DL} of JavaDL is made up of two parts $\Sigma_{DL} = \Sigma_r \cup \Sigma_{nr}$. The symbols in Σ_r are called *rigid* symbols, their interpretation does not depend of the program state. The remaining part Σ_{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 $F : T_1 \times T_2 \to T$ 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 $\doteq$ 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 Σ_r.

The only state-dependent symbols in JavaDL i.e., the only symbols in Σ_{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.*
   $type(x)$, $type(v)$, $type(c)$ are the types declared of these symbols.
2. *If $f : T_1 \times \ldots \times T_n \to T$ is an n-place function symbols and $t_1, \ldots, t_n$ such that $type(t_i) \sqsubseteq T_i$ are terms so is $f(t_1, \ldots, t_n)$ with $type(f(t_1, \ldots, t_n)) = T$.*

all function and predicate symbols for Int, e.g., $+,*,<\dots$		
$Boolean$ constants $TRUE$, $FALSE$		
Heap modeling	$select_A : Heap \times Object \times Field \to A$ for any type $A \sqsubseteq Any$
	$store : Heap \times Object \times Field \times Any \to Heap$
	$created : Field$
	$create : Heap \times Object \to Heap$
	$anon : Heap \times LocSet \times Heap \to Heap$
	$arr : Int \to Field$
	$f : Field$ for all JAVA fields
$LocSet$	$\emptyset$, $allLoc : LocSet$
	$singleton : Object \times Field \to LocSet$
	$\cup, \cap : LocSet \times LocSet \to LocSet$
	$allFields : Object \to LocSet$
	$arrayRange : Object \times Int \times Int \to LocSet$
	$unusedLocs : Heap \to LocSet$
	$\in (Object, Field, LocSet)$
	$\subseteq (LocSet, LocSet)$, $disjoint(LocSet, LocSet)$
Seq	$seqEmpty : Seq$
	$seqSingleton : Any \to Seq$
	$seqConcat : Seq \times Seq \to Seq$
	$seqSub : Seq \times Int \times Int \to Seq$
	$seqReverse : Seq \to Seq$
	$seqGet_A : Seq \times Int \to A$ for any type $A \sqsubseteq Any$
	$seqLen : Seq \to Int$
JAVA	**null** : $Null$
	length : $Object \to Int$
	$cast_A : Any \to A$ for any type $A \sqsubseteq Any$
	$instance_A(Any)$ for any type $A \sqsubseteq Any$
	$exactInstance_A(Any)$ for any type $A \sqsubseteq Any$
Miscellaneous	$\doteq (\top, \top)$

Figure 2. Σ_r the heap-independent symbols of JAVADL

3. *If ϕ is a first-order formula, and t_1, t_2 are terms with $type(t_1) = type(t_2) = T$ 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. Σ_{nr}, Program Variables in JAVADL

JAVADL 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 ϕ is a formula. The construct $\{a := t\}$ is called an* update,
2. $\langle\alpha\rangle\phi$, $[\alpha]\phi$ *are* JavaDL *formulas for any* JavaDL *formula ϕ and any sequential* Java *program α.*[1]
3. *For every integer variable iv,* JavaDL *terms t_1, t_2 with type Int, not containing iv and* JavaDL *expression e*

$$seq\_def\{iv\}(t_1, t_2, e)$$

   is a term of type Seq.
4. *For every integer variable iv and* JavaDL *expression e of type LocSet*

$$infiniteUnion\{iv\}(e)$$

   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 puposes we need to consider here.

The basis for the semantics of JavaDL is provided by a structure $\mathcal{D}$ for typed first-order logic, called the *computation domain.*

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

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

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

The subset inclusion relations among $T^{\mathcal{D}}$ follow from the hierarchy shown in Figure 1. In particular, Heap, Field, and Any are pairwise disjoint. For any Java *class T its interpretation $T^{\mathcal{D}}$ 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., $created^s(o) = tt$. 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 $T^{\mathcal{D}}$ that are not in $T_0^{\mathcal{D}}$ for any subtype $T_0 \sqsubseteq T$. For any class C that occurs in a program to be analysed*

[1] The definition is in fact more liberal in that α 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* $C[]^{\mathcal{D}}$ *is partitioned into infinite subsets* $C^n[]$ *for* $n \geq 1$*. 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 $\mathcal{D}$ we need to give the definition of all rigid symbols in Σ_r. 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 Σ_r.

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* $TRUE^{\mathcal{D}} = tt$ *and* $FALSE^{\mathcal{D}} = ff$.
2. $select^{\mathcal{D}}_A(h, o, f) = cast^{\mathcal{D}}_A(h(o, f))$
3. *For arguments* h, o, f, x *of appropriate types the function value* $h^* = store^{\mathcal{D}}(h, o, f, x)$*, which is itself a function, is given by*
$$h^*(o', f') = \begin{cases} x & \text{if } o' = o, f = f' \text{ and } f \neq created^{\mathcal{D}} \\ h(o', f') & \text{otherwise} \end{cases}$$
4. *For arguments* h, o *of appropriate types the function value* $h^* = create^{\mathcal{D}}(h, o)$ *is given by*
$$h^*(o', f) = \begin{cases} tt & \text{if } o' = o, o \neq null \text{ and } f = created^{\mathcal{D}} \\ h(o', f) & \text{otherwise} \end{cases}$$
5. *For arguments* h, s, h' *of the appropriate types the function value* $h^* = anon^{\mathcal{D}}(h, s, h')$ *is given by*
$$h^*(o, f) = \begin{cases} h'(o, f) & \text{if } ((o, f) \in s \text{ and } f \neq created^{\mathcal{D}}) \\ & \quad \text{or } (o, f) \in unusedLocs^{\mathcal{D}}(h) \\ h(o, f) & \text{otherwise} \end{cases}$$
6. $arr^{\mathcal{D}}$ *is an injective function from* $\mathbb{Z}$ *into* $Field^{\mathcal{D}}$*,* $created^{\mathcal{D}}$ *and* $f^{\mathcal{D}}$ *for each* Java *field are pairwise different elements in* $Field^{\mathcal{D}}$ *and also not in the range of* $arr^{\mathcal{D}}$.
7. $\emptyset^{\mathcal{D}} = \emptyset$, $allLocs^{\mathcal{D}} = Object^{\mathcal{D}} \times Field^{\mathcal{D}}$
8. $singleton^{\mathcal{D}}(o, f) = \{(o, f)\}$
9. $\cup^{\mathcal{D}}$ *and* $\cap^{\mathcal{D}}$ *are the set theoretical union and intersection of sets of locations.*
10. $allFields^{\mathcal{D}}(o) = \{(o, f) \mid f \in Field^{\mathcal{D}}\}$
11. $arrayRange^{\mathcal{D}}(o, i, j) = \{(o, arr^{\mathcal{D}}(x)) \mid x \in \mathbb{Z}, i \leq x, x \leq j\}$
12. $unusedLocs^{\mathcal{D}}(h) = \{(o, f) \in allLocs^{\mathcal{D}} \mid o \neq null, h(o, created^{\mathcal{D}}) = ff\}$
13. *The usual set theoretic definitions:*
$\in^{\mathcal{D}} = \{(o, f, s) \in Object^{\mathcal{D}} \times Field^{\mathcal{D}} \times LocSet^{\mathcal{D}} \mid (o, f) \in s\}$
$\subseteq^{\mathcal{D}} = \{(s, s') \mid \forall o, f((o, f) \in s \to (o, f) \in s'\}$
$disjoint^{\mathcal{D}} = \{(s, s') \mid s \cap s' = \emptyset\}$
14. $\mathbf{null}^{\mathcal{D}} = null$.
15. $cast^{\mathcal{D}}_A(o) = \begin{cases} o & \text{if } o \in A^{\mathcal{D}} \\ default_A & \text{otherwise} \end{cases}$
The default element $default_A$ *for type* A *is as follows:*
$$default_A = \begin{cases} null & \text{if } A \sqsubseteq Object \\ \emptyset & \text{if } A = LocSet \\ seqEmpty & \text{if } A = Seq \\ ff & \text{if } A = Boolean \end{cases}$$
16. $instance^{\mathcal{D}}_A = A^{\mathcal{D}}$
17. $exactInstance^{\mathcal{D}}_A = A^{\mathcal{D}} \setminus \bigcup\{B^{\mathcal{D}} \mid B \sqsubset A\}$

18. $\mathbf{length}^{\mathcal{D}}(o) = \begin{cases} n \text{ if } o \in C^n[] \text{ for some } C \\ 0 \text{ otherwise} \end{cases}$

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., $(5/0)^{\mathcal{D}}$ 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:

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

Instead of one computation domain $\mathcal{D}$ we consider all computation domains $\mathcal{D}'$ covering all possible assignments of values to $n/^{\mathcal{D}'}0$. The prover will use the axiom

$$\begin{aligned} \forall x \forall y (y \neq 0 \to |y| * |x/y| \leq |x| \land |y| * (|x/y|+1) > |x| \land \\ ((x/y \geq 0 \land x \geq 0 \land y \geq 0) \lor (x/y \geq 0 \land x \leq 0 \land y \leq 0) \lor \\ (x/y \leq 0 \land x \leq 0 \land y \geq 0) \lor (x/y \leq 0 \land x \geq 0 \land y \leq 0))) \end{aligned}$$

Then a formula ϕ can be derived using this axiom iff it is true in all computation domains. Thus $\exists z(5/0 \doteq z)$ can be derived but $5/0 \doteq 7/0$ cannot. For ease of presentation we will in the following nevertheless speak of *the* computation domain $\mathcal{D}$ 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 $cast_A^{\mathcal{D}}(o) = \text{null}$ if $o \notin A^{\mathcal{D}}$. 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 $default_A$ for some types A have been fixed. Notice that $default_{int}$ 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 $\mathcal{D}$.*
2. *A computation domain $\mathcal{D}$ and a state s together yield a Σ_{DL} structure for first order logic, denoted by $\mathcal{D} + s$.*
   $\mathcal{D} + s$ has the same domain as $\mathcal{D}$ and the interpretation of the rigid symbols is in $\mathcal{D} + s$ the same as in $\mathcal{D}$. The interpretation of the program variables v in Σ_{nr} is fixed as $v^{\mathcal{D}+s} = s(v)$.

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* ϕ *then* t_1 *else* t_2. If t contains logical variables a variable assignment β is needed to evaluate the term to $t^{s,\beta}$. In the following, we will omit β whenever it is not essential.

The recursive definition of when a formula ϕ is true in state s with assignment β for free (logical) variables, in symbols $(s, \beta) \models \phi$, follows the usual pattern. We again omit β whenever it is not essential. To be really precise we should even write $(\mathcal{D} + s, \beta) \models \phi$ to document that truth of ϕ also depends on the computation domain. But, since $\mathcal{D}$ 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`:

$$\begin{aligned}
\{o.f := t\} &:= \{\mathtt{heap} := \mathit{store}(\mathtt{heap}, o, f, t)\}\\
\{a[i] := t\} &:= \{\mathtt{heap} := \mathit{store}(\mathtt{heap}, a, \mathit{arr}(i), t)\}\\
\{\mathit{sf} := t\} &:= \{\mathtt{heap} := \mathit{store}(\mathtt{heap}, \mathtt{null}, \mathit{sf}, t)\}
\end{aligned}$$

With this we define for a JavaDL formula ϕ and state s:

1. $s \models \{a := t\}\phi$ iff $s' \models \phi$, where s' coincides with s except for $s'(a) = t^s$.
2. $s \models \langle\alpha\rangle\phi$ iff $s' \models \phi$ for some s' such that α started in s terminates in s'.
3. $s \models [\alpha]\phi$ iff $s' \models \phi$ for all s' such that α started in s terminates in s'.
4. $(\mathit{seq\_def}\{iv\}(t_1, t_2, e))^{(s,\beta)}$ is the sequence of the elements $e^{(s,\beta[n/iv])}$ for all n with $t_1^s \leq n < t_2^s$ in this order. If $t_1^s \geq t_2^s$ then $\mathit{seq\_def}\{iv\}(t_1, t_2, e))^s = \langle\rangle$.
5. $(\mathit{infiniteUnion}\{iv\}(e))^{(s,\beta)} = \{e^{(s,\beta[n/iv])} \mid i \in \mathbb{Z}\ \}$

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

Digression In Figure 2 that lists the rigid symbols Σ_r of JavaDL for 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 $f : C_1 \to C_2$ in Σ_{nr}. In that approach a state is a Σ_{nr} structure $\mathcal{S}$ (with universe D), which is certainly a more complicated concept than our simple mapping from program variables to values in D. The non-rigid function symbol $f : C_1 \to C_2$ would in $\mathcal{S}$ be interpreted as a function $f^{\mathcal{S}} : C_1^{\mathcal{D}} \to C_2^{\mathcal{D}}$. There is an easy correspondence between these two approaches. A state s in our sense defines a corresponding Σ_{nr} structure $\mathcal{S}$ via the definition

$$f^{\mathcal{S}}(o) = \mathit{select}^{\mathcal{D}}_{C_2}(\mathbf{heap}^s, o, f^{\mathcal{D}})$$

for all $o \in C_1^{\mathcal{D}}$.

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 JavaDL $\mathit{select}_{C_2}(\mathbf{heap}, t, f)$. 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 $\mathit{select}_{C_2}(\mathbf{heap}, t, f)$. Note, that the (current) heap in a state s is implicitely 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 $\mathit{select}^{\mathcal{D}}_C(\mathbf{heap}^s, o, f^{\mathcal{D}})$, 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 $\mathit{select}_C(\mathbf{heap}, t, f)$ 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 $\mathit{select}_C(\mathbf{heap}, \mathbf{null}, f)$ and on the semantic side f^s for the value $\mathit{select}^{\mathcal{D}}_C(\mathbf{heap}^s, \mathit{null}, f^{\mathcal{D}})$, with C the type of f. This conforms with [22, page 102]

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

$$\begin{aligned}
&\forall \mathit{Int}\ i((0 \leq i\ \wedge\ i < \mathit{MAX_VALUE})\ \to\\
&\quad\{a := i\}\langle\alpha\rangle(0 \leq \mathtt{r}\ \wedge\ \mathtt{r} * \mathtt{r} \leq i\ \wedge\ (\mathtt{r}+1) * (\mathtt{r}+1) > i))
\end{aligned} \tag{1}$$

$$\begin{aligned}
&\forall \mathit{Heap}\ h, h'\ \forall \mathit{Int}\ i, i'((\mathit{select}_{\mathit{Any}}(h, \mathtt{this}, f) \doteq \mathit{select}_{\mathit{Any}}(h', \mathtt{this}, f) \wedge\\
&\quad\{\mathtt{heap} := h\}\langle m()\rangle i \doteq \mathtt{r}\ \wedge \{\mathtt{heap} := h'\}\langle m()\rangle i' \doteq \mathtt{r}) \to i \doteq i')
\end{aligned} \tag{2}$$

Formula (1) expresses that program α 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 $\texttt{this}.f$.

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 $o.created$ from ff to tt. The objects for this change are chosen depending on the computation domain $\mathcal{D}$. If $\mathcal{D}_1$, $\mathcal{D}_2$ are computations domains, and s a state, then the next new object created in $\mathcal{D}_1 + s$ may differ from the next new object created in $\mathcal{D}_2 + s$. On the semantic level we have for each type T an (underspecified) function $nextToCreate_T{}^{\mathcal{D}}$ which determines for each interpretation $\mathcal{D} + s$ the the value $nextToCreate_T{}^{\mathcal{D}}(s)$ of the next object to be created of type T. We consider only functions $nextToCreate_T{}^{\mathcal{D}}$ where $exactInstance_T{}^{\mathcal{D}+s}(nextToCreate_T{}^{\mathcal{D}}(s)) = tt$ and $created^{\mathcal{D}+s}(nextToCreate_T{}^{\mathcal{D}}(s)) = ff$ holds.

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

$$
\begin{array}{ll}
1\ \{o \mid select_{Boolean}(h, o, created) \doteq TRUE\} \textit{ is finite} & \wedge \\
2\ \forall Field\ f\ \forall Object\ o(select_{Object}(h, o, f) \doteq \textbf{null} \vee & \\
\quad select_{Boolean}(h, select_{Object}(h, o, f), created) \doteq TRUE) & \wedge \\
3\ \forall Field\ f, f'\ \forall Object\ o, o'(& \\
\quad (o', f') \in select_{LocSet}(h, o, f) \to select_{Boolean}(h, o', created) \doteq TRUE) & \wedge \\
4\ \bigwedge_{\text{JAVA } field\ f} \forall Object\ o(instance_{type(f)}(select_{Any}(h, o, f))) & \wedge \\
5\ \bigwedge_{\text{JAVA } type\ A \sqsubseteq Any} \forall Int\ i\ \forall Object\ o(& \\
\quad (i \geq 0 \wedge instance_{A[]}(o) \wedge o \neq \texttt{null}) & \\
\qquad \to (instance_A(select_{Any}(h, o, arr(i))))) &
\end{array}
$$

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 $type(f) \not\sqsubseteq Object$ the semantics definition yields $select^{\mathcal{D}}_{Object}(h, o, f) = null$

Property 3 requires that only those location sets L can be values of fields of type *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 *LocSet* the semantics definition yields $select^{\mathcal{D}}_{LocSet}(h, o, f) = \emptyset$ and property 3 is trivially true.

Property 4 depends on the JAVA program α under verification. The initial conjunction ranges of all fields occuring in α and is thus finite. This conjunction cannot be replaced by a universal quantifier since the *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 α under verification. The leading conjunction is finite, since α 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.

```
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 $\mathtt{m}_i$.

If equivalence means identity, method m1 would not be deemed secure. The reason is that the values of $\mathtt{x}^{s_1}$ and $\mathtt{x}^{s'_1}$ 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 $\mathtt{x}^{s_1}$ and $\mathtt{x}^{s'_1}$ are identical (nor that they are different). On the other hand, method m1 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 m2 of Figure 4, the observation of an attacker depends on the value of the secret variable h. The attacker can deduce that $\mathtt{h}^{s_0}$ is true if and only if the value of x changes. Information is leaked here. In contrast, method m3 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 m4, 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 $\{\mathtt{x}^{s_4}, \mathtt{y}^{s_4}, \mathtt{z}^{s_4}\}$ and $\{\mathtt{x}^{s'_4}, \mathtt{y}^{s'_4}, \mathtt{z}^{s'_4}\}$ 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. *$e_0.f$ if e_0 is an expression of type C and f is a field declared in C (static or not).*
4. *$e_a[t]$ if e_a is an expression of array type, and t is an expression of integer type.*
5. *$op(e_1, \ldots, e_k)$ where op is a data type operation and e_i expressions of matching type. Most frequently arithmetic operations will occur.*
6. *$b?e_1 : e_2$ the usual conditional operator. We (still) assume that e_1, e_2 are of the same type.*
7. *Auxiliary ghost variables, for the purpose of exposition only.*

Expressions $q(p_1, \ldots, p_n)$ for queries q are not included. For uniformity of notation we will frequently write $f(e_0)$ instead of $e_0.f$ and assume that $e_a[t]$ is presented as $at(e_a, t)$.

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 R_1 and R_2 are observation expressions, so is $R = R_1; R_2$.*
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. $[seq\{i\}(from, to, e)]^s =$
$\langle ([e^{[i \to n]}]^s)), ([e^{[i \to n+1]}]^s), \ldots, ([e^{[i \to m-1]}]^s) \rangle$,
if $from^s = n < m = to^s$.
Here $e^{[i \to n]}$ *is the expression obtained from e by replacing all occurrences of the variable i by the literal n.*

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

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

Given an observation expression R and a state s, an attacker is able to see the tuple $([R]^s, R^s)$, where $R^s = \langle e^s_1, \ldots, e^s_k \rangle$ if $[R]^s = \langle e_1, \ldots, e_k \rangle$. Hence he is able to deduce for any $0 \le i < length([R]^s)$ 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 $[R_1]^s$ 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 $e.getClass()$.

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 `m3` could deduce that $\mathtt{h}^{s_0}$ 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 `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 π 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 ρ be an automorphism of $\mathcal{D}$, s a state, ϕ a formula, e an expression, and β a variable assignment into $\mathcal{D}$.*
Then

1. $(s, \beta) \models \phi \Leftrightarrow (\rho(s), \rho(\beta)) \models \phi$
   which reduces to $s \models \phi \Leftrightarrow \rho(s) \models \phi$ if ϕ contains no free variables.
2. $\rho(e^{(s,\beta)}) = e^{(\rho(s),\rho(\beta))}$
   which reduces to $e^{\rho(s)} = \rho(e^s)$ 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 ρ 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 π_0 of $Obj^{\mathcal{D}}$ satisfying*

1. $\pi_0(null) = null$
2. *π_0 preserves the exact types of its arguments.*
3. *π_0 preserves the length of array objects.*

can be extended to an automorphism π of $\mathcal{D}$.

Proof We first describe how to extend π_0 to a bijection on D. For the following the reader might want to have again a look at the type hierarchy in Figure 1. On $Obj^{\mathcal{D}}$ we let π necessarily coincide with π_0 We set π equal to the identity function on the type universes $Boolean^{\mathcal{D}}$, $Int^{\mathcal{D}}$, $Field^{\mathcal{D}}$.

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

Next we describe the action of the bijection π on $Seq^{\mathcal{D}}$. By Definition 19 on page 31 $Seq^{\mathcal{D}} = \bigcup_{n \geq 0} D^n_{Seq}$.1 By construction $D^i_{Seq} \cap D^0_{Seq} = \emptyset$ for $i > 0$ and $D^i_{Seq} \cap D^j_{Seq} = \{\langle\rangle\}$ for $i > 0$, $j > 0$, $i \neq j$. We inductively define permuations π^n_{seq} of $\bigcup_{0 \leq i \leq n} D^i_{Seq}$. We start with π^0_{seq} equal to the mapping π defined so far for

$D^0_{Seq} = Boolean^{\mathcal{D}} \cup Int^{\mathcal{D}} \cup Object^{\mathcal{D}} \cup LocSet^{\mathcal{D}}$. $\pi^{n+1}_{seq}(\langle o_0, \ldots, o_{n-1}\rangle) = \langle \pi^n_{seq}(o_0), \ldots, \pi^n_{seq}(o_{n-1})\rangle$ for $o_i \in \bigcup_{0 \leq i \leq n} D^i_{Seq}$. It is easily checked that $\pi_{seq} = \bigcup_{n \geq 0} \pi^n_{seq}$ is a permutation of $Seq^{\mathcal{D}}$.

The most involved case remains, to define π on $Heap^{\mathcal{D}}$. Every $h \in Heap^{\mathcal{D}}$ is a mapping $h : Object^{\mathcal{D}} \times Field^{\mathcal{D}} \to Any^{\mathcal{D}}$. The mapping $\pi(h) : Object^{\mathcal{D}} \times Field^{\mathcal{D}} \to Any^{\mathcal{D}}$ 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 $\pi : Heap^{\mathcal{D}} \to Heap^{\mathcal{D}}$ thus defined is a bijection.

This completes the definition of the bijection $\pi : D \to D$. We now embark on the lengthy verification that π is an Σ_r isomorphism. Consideration of the symbols in Σ_{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.
$$\begin{aligned} \pi(cast_A^{\mathcal{D}}(o)) &= \pi(o) && \text{if } o \in A^{\mathcal{D}} \\ &= \pi(o) && \text{if } \pi(o) \in A^{\mathcal{D}} \\ &= cast_A^{\mathcal{D}}(\pi(o)) \end{aligned}$$
The other three cases in the semantic definition of $cast_A$ follow along the same line.
2.
$$\begin{aligned} \pi(select_A^{\mathcal{D}}(h,o,f)) &= \pi(cast_A^{\mathcal{D}}(h(o,f))) && \text{semantics of } select_A \\ &= cast_A^{\mathcal{D}}(\pi(h(o,f))) && \text{see first item} \\ &= cast_A^{\mathcal{D}}(\pi(h)(\pi(o),f)) && \text{def. of } \pi(h) \\ &= select_A^{\mathcal{D}}(\pi(h),\pi(o),f) && \text{semantics of } select_A \end{aligned}$$
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 $\pi(store^{\mathcal{D}}(h,o,f,x)) = store^{\mathcal{D}}(\pi(h),\pi(o),f,\pi(x))$. To this end we show for any argument pair (o', f')
$$\pi(store^{\mathcal{D}}(h,o,f,x))(o',f') = store^{\mathcal{D}}(\pi(h),\pi(o),f,\pi(x))(o',f'). \qquad (3)$$
By definition of π the lefthand side equals $\pi(store^{\mathcal{D}}(h,o,f,x)(\pi^{-1}(o'),f'))$. In case, $\pi^{-1}(o') \neq o$ or $f \neq f'$ or $f = created^{\mathcal{D}}$ the semantics of *store* yields the further rewriting: $\pi(h(\pi^{-1}(o'),f'))$. Which again be the definition of π is equal to $\pi(h)(o',f')$.
The case assumption implies $o' \neq \pi(o)$ or $f \neq f'$ or $f = created^{\mathcal{D}}$. By the semantics of store this leads to following rewritting of the righhand side of the equation (3)
$store^{\mathcal{D}}(\pi(h),\pi(o),f,\pi(x)) = \pi(h)(o',f')$ and we are done for the first case.
In case $\pi^{-1}(o') = o$ and $f = f'$ and $f \neq created^{\mathcal{D}}$ the semantics definition yields: $\pi(store^{\mathcal{D}}(h,o,f,x)) = \pi(x)$.
Since the case condition implies $o' = \pi(o)$ and $f = f'$ and $f \neq created^{\mathcal{D}}$ the righthand side of (3) evaluates to $\pi(x)$, as desired.
4. Since *created* is a constant of type $Field$ the definition of π yields $\pi(created) = created$.
5. We need to show for all pairs (o', f):
$$\pi(create^{\mathcal{D}}(h,o))(o',f) = create^{\mathcal{D}}(\pi(h),\pi(o))(o',f) \qquad (4)$$
By definition of π the left side can be rewritten to
$$\pi(create^{\mathcal{D}}(h,o)(\pi^{-1}(o'),f)).$$
In case $\pi^{-1}(o') \neq o$ or $o = \mathbf{null}$ or $f \neq created^{\mathcal{D}}$ the semantics of *create* yields
$\pi(create^{\mathcal{D}}(h,o)(\pi^{-1}(o'),f)) = \pi(h(\pi^{-1})o'),f))$. Again using the definition of π ge get $\pi(h(\pi^{-1})o'),f)) = \pi(h)(o',f)$.
The case assumption implies $o' \neq \pi(o)$ or $\pi(o) = \mathbf{null}$ or $f \neq created^{\mathcal{D}}$. The semantics of *create* for the righthand side in equation (4) yields
$create^{\mathcal{D}}(\pi(h))\pi(o))(o',f) = \pi(h)(o',f)$ as desired.
In case $\pi^{-1}(o') = o$ and $o \neq \mathbf{null}$ and $f = created^{\mathcal{D}}$ the semantics of *create*

yields $\pi(\mathit{create}^{\mathcal{D}}(h,o)(\pi^{-1}(o'),f)) = tt$.
The current case assumption implies $o' = \pi(o)$ and $\pi(o) \neq \mathbf{null}$ and $f \neq \mathit{created}^{\mathcal{D}}$. Thus, according to the semantics of *create* the righthand side of (4) evaluates also to *true*.

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

$$\pi(\mathit{anon}^{\mathcal{D}}(h,s,h'))(o,f) = \mathit{anon}^{\mathcal{D}}(\pi(h),\pi(s),\pi(h'))(o,f) \tag{5}$$

By definition of π the lefthand side of equation (5) can be rewritten as

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

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

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

By definition of π on the type universe $\mathit{LocSet}^{\mathcal{D}}$ we see that $(\pi^{-1}(o),f) \in s$ is equivalent to $(o,f) \in \pi(s)$ and $(\pi^{-1}(o),f) \in \mathit{unusedLocs}^{\mathcal{D}}(h)$ is equivalent to $(o,f) \in \pi(\mathit{unusedLocs}^{\mathcal{D}}(h))$. The case assumption thus implies $(o,f) \in \pi(s)$ and $f \neq \mathit{created}^{\mathcal{D}}$ or $(o,f) \in \pi(\mathit{unusedLocs}^{\mathcal{D}}(h))$. We will see below that furthermore $\pi(\mathit{unusedLocs}^{\mathcal{D}}(h)) = \mathit{unusedLocs}^{\mathcal{D}}(\pi(h))$ 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*

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

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

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

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 π on the type universe $\mathit{LocSet}^{\mathcal{D}}$.

9. $$\begin{aligned} \pi(\mathit{allLocs}^{\mathcal{D}}) &= \pi(\mathit{Object}^{\mathcal{D}} \times \mathit{Field}^{\mathcal{D}}) && \text{semantics of } \mathit{allLocs} \\ &= \pi(\mathit{Object}^{\mathcal{D}}) \times \pi(\mathit{Field}^{\mathcal{D}}) && \text{def of } \pi \text{ on pairs} \\ &= \mathit{Object}^{\mathcal{D}} \times \mathit{Field}^{\mathcal{D}} && \text{surjectivity of } \pi \\ &= \mathit{allLocs}^{\mathcal{D}} && \text{semantics of } \mathit{allLocs} \end{aligned}$$

10. $$\begin{aligned} \pi(\mathit{singleton}^{\mathcal{D}}(o,f) &= \pi(\{(o,f)\}) && \text{semantics of } \mathit{singleton} \\ &= \{(\pi(o),\pi(f))\} && \text{def of } \pi \\ &= \mathit{singleton}^{\mathcal{D}}(\pi(o),\pi(f)) && \text{semantics of } \mathit{singleton} \end{aligned}$$

11. $$\begin{aligned} \pi(LS_1 \cap LS_2) &= \{(\pi(o),\pi(f)) \mid (o,f) \in LS_1 \cap LS_2\} && \text{def. of } \pi \\ &= \{(\pi(o),\pi(f)) \mid (o,f) \in LS_1\} \cap \\ &\qquad \{(\pi(o),\pi(f)) \mid (o,f) \in LS_2\} && \text{set theory} \\ &= \pi(LS_1) \cap \pi(LS_2) && \text{def. of } \pi \end{aligned}$$

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

12. $$\begin{aligned} \pi(\mathit{allFields}^{\mathcal{D}}(o)) &= \pi(\{(o,f) \mid f \in \mathit{Fields}^{\mathcal{D}}\}) && \text{semantics of } \mathit{allFields} \\ &= \{(\pi(o),f) \mid f \in \mathit{Fields}^{\mathcal{D}}\} && \text{def. of } \pi \\ &= \mathit{allFields}^{\mathcal{D}}(\pi(o)) && \text{semantics of } \mathit{allFields} \end{aligned}$$

13. $\pi(arrayRange^{\mathcal{D}}(o,i,j)) = \pi(\{(o, arr^{\mathcal{D}}(x)) \mid z \in \mathbb{Z}, i \leq x, x \leq j\})$
semantics of $arrayRange$
$= \{(\pi(o), \pi(arr^{\mathcal{D}}(x))) \mid z \in \mathbb{Z}, i \leq x, x \leq j\}$
def of π
$= \{(\pi(o), arr^{\mathcal{D}}(\pi(x))) \mid z \in \mathbb{Z}, i \leq x, x \leq j\}$
item 7
$= \{(\pi(o), arr^{\mathcal{D}}(x)) \mid z \in \mathbb{Z}, i \leq x, x \leq j\}$
π is identity on $\mathbb{Z}$
$= arrayRange^{\mathcal{D}}(\pi(o), i, j)$
semantics of $arrayRange$
14. $\pi(unusedLocs^{\mathcal{D}}(h)) =$
$\pi(\{(o,f) \in allLocs^{\mathcal{D}} \mid o \neq null, h(o, created^{\mathcal{D}}) = ff\})$
semantics of $unusedLocs$
$= \{(\pi(o),f) \in allLocs^{\mathcal{D}} \mid o \neq null, h(o, created^{\mathcal{D}}) = ff\}$
def of π on $LocSet^{\mathcal{D}}$
$= \{(o',f) \in allLocs^{\mathcal{D}} \mid o' \neq null, \pi(h)(o', created^{\mathcal{D}}) = ff\}$
def of $\pi(h)$
$= unusedLocs^{\mathcal{D}}(\pi(h)))$ semantics of $unusedLocs$
15. We need to show $(o,f) \in LS \Leftrightarrow (\pi(o),f) \in \pi(LS)$. But, this is the very definition of $\pi(LS)$.
16. $LS_1 \subseteq LS_2 \Leftrightarrow \pi(LS_1) \subseteq \pi(LS_2)$
and $disjoint(LS_1, LS_2) \Leftrightarrow disjoint(\pi(LS_1), \pi(LS_2))$
follow easily from the definition of $\pi(LS_i)$.

Next in the list in Figure 2 would be the symbols under the heading *Seq*.

17. $\pi(seqEmpty^{\mathcal{D}}) = \pi(\langle\rangle) = \langle\rangle = seqEmpty^{\mathcal{D}}$.
18. $\pi(seqSingleton^{\mathcal{D}}(o)) = \pi(\langle o\rangle) = \pi(o) = seqSingleton^{\mathcal{D}}(\pi(o))$.
19. Having seen the previous two examples we trust that the reader can do the remaining cases by himself.
20. $\pi(\mathbf{null}^{\mathcal{D}}) = \mathbf{null}^{\mathcal{D}} = null$, $\pi(\mathbf{length}^{\mathcal{D}}(o)) = \mathbf{length}^{\mathcal{D}}(\pi(o))$ and the equivalence $\pi(exactInstance_A^{\mathcal{D}}(o)) \Leftrightarrow exactInstance_A^{\mathcal{D}}(\pi(o))$ follow directly from the definition of the bijection π.

This completes the proof that π is an automorphism of $\mathcal{D}$. □

Lemma 4. *Let π' be a bijection from X onto Y for finite subsets $X, Y \subseteq Obj^{\mathcal{D}}$ with*

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

Then there is an automorphism π on $\mathcal{D}$ extending π'.

Proof To define an extension π_0 of π' on $Obj^{\mathcal{D}}$ it suffices to explain what π_0 does on the sets $T_e^{\mathcal{D}}$ of objects of exact type T for every Java class T. First, we set $\pi_0(null) = null$. By assumption this is compatible with π'. By the assumed preservation of exact types and finiteness of X and Y we know that $T_e^{\mathcal{D}} \cap X$ and $T_e^{\mathcal{D}} \cap Y$ have the same finite number of elements. Since $T_e^{\mathcal{D}}$ is infinite we find a bijection π'_0 from $T_e^{\mathcal{D}} \setminus X$ onto $T_e^{\mathcal{D}} \setminus Y$. The bijection π_0 on $T_e^{\mathcal{D}}$ is the disjoint union of π' and π'_0. This, of course, also applies to array types $T = C[]$. In this case π_0 is constructed in such a way that $C^n[]$, is bijectively mapped onto itself for all $n \geq 1$. Again, by assumption this is compatible with π'.

By Lemma 3 there is an isomorphism π of $\mathcal{D}$ extending π_0 and thus π'. □

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 $\pi : Obj(R^{s_1}) \rightarrow Obj(R^{s_2})$ 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 π 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 π according to Lemma 4 is a (total) isomorphism from $\mathcal{D} + s_1$ onto $\mathcal{D} + s_2$ since $\pi(p^{s_1}) = p^{s_2}$ by the last requirement.

Not every partial isomorphism can be extended to a total isomophism, 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 ρ an isomorphism on $\mathcal{D}$.*

Let α be a program which started in s_1 terminates in s_2.

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

(See Definition 11 for the definition of $\rho(s_i)$)

Proof. The reason why we cannot assume $\rho = \rho'$, is that α 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 ρ-image of the new element o generated in the run of α starting in state s_1.

Let $N_T^{s_1}$ 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 $N_T^{s_1}$ 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 Java semantics. We take them as postulates, and a very plausible postulates, how 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 π_0 the injective mapping defined on $G \cup \bigcup_T N_T^{s_1}$ such that $\pi_0(o) = \rho(o)$ for $o \in G$ and the restrictions of π_0 map $N_T^{s_1}$ bijectively on $N_T^{\rho(s_1)}$. By Lemma 4 there is an automorphism ρ' of $\mathcal{D}$ extending π_0. This ρ' 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 $agree(R, s, s')$, iff

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

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

Notice, that because of our tacit agreement on the values of partial isomorphisms on primitive values $\text{agree}(R, s, s')$ 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 α (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 α be a program and R_1 and R_2 be two observation expressions (of type Seq)

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

iff

for all states s'_1, s_2, s'_2 such that
α started in s_1 terminates in s_2 and
α started in s'_1 terminates in s'_2,
we have

$$\begin{array}{ll} \textit{if} & agree(R_1, s_1, s'_1, \pi^1) \\ \textit{then} & agree(R_2, s_2, s'_2, \pi^2) \textit{ and } \pi^2 \textit{ is compatible with } \pi^1 \end{array}$$

where π^2 is said to be compatible with π^1 if
$\pi^2(o) = \pi^1(o)$ for all $o \in Obj(R_1^{s_1}) \cap Obj(R_2^{s_2})$ with $created^{s_1}(o) = tt$.

We extend JavaDL *by a new three-place modal operator $flow(\cdot, \cdot, \cdot))$ 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*

$$s \models flow(\alpha, R_1, R_2) \qquad \textit{iff} \qquad flow(s, \alpha, R_1, R_2) \textit{ holds} .$$

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:
```
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., $R^s = \{self.x\}$ for all states s; then flow($\mathtt{m()}, R, R$) 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.
```
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., $R^s = \{self.x\}$ for all states s; then flow($\mathtt{m()}, R, R$) is again not satisfied. The mapping π_2 defined by $\pi_2(x^{s_2}) = x^{s_2'}$ with s_2, s_2' the poststates of m() when started in s_1, respectively s_1', is certainly a partial isomorphism. But, π_2 is not in the cases compatible with the isomorphims π_1 given by $\pi_1(x^{s_1}) = x^{s_1'}$, e.g., not in the case $h^{s_1} = h^{s_2} = tt$, $x^{s_1} = y^{s_1}$, 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 refered 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'

$$agree(R_1, s, s', \pi_1) \quad implies \quad agree(R_2, s, s', \pi_2)$$

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

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

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

By Lemma 4 there is an automorphism ρ of the computation structure $\mathcal{D}$ such that for all $o \in Obj(R_1^s)$ 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 ℓ. In particular,

$\rho(n) = n$ for all $n \in \mathbb{N}$. Only object may be *moved* by ρ. It is thus safe to assume $\rho(R_i^s) = R_i^s$. We cannot prove this here, since we have not fixed a syntax for observations expressions. Alltogether, we get $e_i^{\rho(s)} = \rho(e_i^s) = e_i^s$ (See Definition 11 for the definition of $\rho(s)$ and Lemma 1 for the equation.) This entails $agree(R_1, s, \rho(s), \rho)$.

On the other hand because of $d_1^{\rho(s)} = \rho(d_1^s) = \rho(o_1) \neq o_1 = d_1^s$ we cannot have $agree(R_2, s, \rho(s))$. □

If for observation expressions R_2, R_1 we have $R_2^s \subseteq R_1^s$ for all states s then certainly $R_2 \subseteq R_1$. But for integer fields x, y we also have $R_2 = \{x, y, x+y\} \subseteq \{x, y\} = R_1$. Note, that in this example we have $Obj(R_2^s) = Obj(R_1^s) = \emptyset$ for all s.

The following lemma has been used to prove soundness of the rules of the caclucus 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 FlowSplit in our calculus.

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

1. *flow*(ϵ, R_1, R_2) *if* $R_2 \subseteq R_1$.
2. *flow*(α, R_1, R_2) *implies flow*(α, R_1, R_2') *if* $R_2' \subseteq R_2$.
3. *if flow*(α_1, R_1, R_2), *flow*(α_2, R_2, R_3) *and* $Obj(R_1^s) \cap Obj(R_3^s) \subseteq Obj(R_2^s)$ *for all* s *then flow*$(\alpha_1; \alpha_2, R_1, R_3)$. *Here,* $\alpha_1;\alpha_2$ *is the concatenation of* α_1 *and* α_2.

Proofs
ad(1) By Definition 14 we need to show for any states s_1, s_1', s_2 ,s_2' such that ϵ started in s_1 terminates in s_2 and started in s_1' terminates in s_2' that agree(R_1, s_1, s_1', π_1) implies agree(R_2, s_2, s_2', π_2) and π_1, π_2 are compatible For the empty program ϵ we have $s_1 = s_2$ and $s_1' = s_2'$. The claim thus reduces to showing that agree(R_1, s_1, s_1', π_1) implies agree(R_2, s_1, s_1', π_2) and the compatibility of π_1, π_2. But, this follows from the definition of $R_2 \subseteq R_1$ and Lemma 6.
ad(2) To prove flow(α, R_1, R_2') we need to show
for any states s_1, s_1', s_2 ,s_2' such that
α started in s_1 terminates in s_2 and
α started in s_1' terminates in s_2' that
agree(R_1, s_1, s_1', π_1) implies agree$(R_2', s_2, s_2', \pi_2')$ and the compatibility of π_1 and π_2'.

By the assumption flow(α, R_1, R_2) we know agree(R_2, s_2, s_2', π_2) plus compatibility of π_1 and π_2. The claim follows from $R_2' \subseteq R_2$. In particular compatibility of π_1 and π_2' follows from the compatibility of π_1 and π_2 since $Obj((R_2')^{s_2}) \subseteq Obj(R_2^{s_2})$ by Lemma 6.
ad(3) We are given states $s_1, s_1', s_2, s_2', s_3, s_3'$ such that $s_1 \overset{\alpha_1}{\leadsto} s_2$, $s_2 \overset{\alpha_2}{\leadsto} s_3$, $s_1' \overset{\alpha_1}{\leadsto} s_2'$, $s_2' \overset{\alpha_2}{\leadsto} s_3'$, and we know from flow$(\alpha_1, R_1, R_2)$, flow$(\alpha_2, R_2, R_3)$ that
agree(R_1, s_1, s_1', π_1) implies agree(R_2, s_2, s_2', π_2) and
agree(R_2, s_2, s_2', π_2) implies agree(R_3, s_3, s_3', π_3).
Thus agree(R_1, s_1, s_1', π_1) certainly implies agree(R_3, s_3, s_3', π_3) and it remains only to show compatibility of π_1 and π_3. We may make use of teh facts that π_1 and π_2 on one hand and π_2 and π_3 on the other are compatible. So we fix $o \in Obj(R_1^{s_1}) \cap Obj(R_3^{s_1})$ and want to show $\pi_1(o) = \pi_3(o)$. Since by assumption $o \in Obj(R_2^{s_1})$ we get $\pi_1(o) = \pi_2(o)$ from the compatibility of π_1 and π_2 and $\pi_2(o) = \pi_3(o)$ from the compatibility of π_2 and π_3. □

Example 6. It might be tempting to conjecture that flow(α, R_1, R_2) implies flow(α, R_1', R_2) if $R_1 \subseteq R_1'$. Here comes a counterexample.

```
class C {
 C x, y;
```

```
 static boolean h;

 static void ce(){
   if (h) {x = new C()} else {x = y}
  }
}
```

We argue that flow($ce()$, $\emptyset$, $\{x\}$) is true. Thus consider states s_1, s'_1, s_2, s'_2 with $s_1 \overset{ce()}{\leadsto} s_2$, $s'_1 \overset{ce()}{\leadsto} s'_2$, and agree($\emptyset, s_1, s'_1$). We omit π_1 here since it is the empty function. We need to convince ourselves that agree($\{x\}, s_2, s'_2, \pi_2$). But, this is easy since π_2 is the mapping from the singleton $\{x^{s_2}\}$ onto the singleton $\{x^{s'_2}\}$.

On the other hand flow($ce()$, $\{y\}$, $\{x\}$) is not true. In this case we start from agree($\{y\}, s_1, s'_1, \pi_1$) and get agree($\{x\}, s_1, s'_1, \pi_2$) as before. But, now π_2 and π_1 may not be compatible in some case, e.g., if $h^{s_1} = \mathit{ff}$, $h^{s'_1} = \mathit{tt}$ then π_2 maps y^{s_1} onto a new element, while $\pi_1(y^{s_1}) = y^{s'_1}$ is an existing element.

Example 7. The following example illustrates why condition

$$Obj(R_1^s) \cap Obj(R_3^s) \subseteq Obj(R_2^s)$$

for part 3 of Lemma 7 is needed. Let

$\alpha_1 \quad = \quad$ `if (h != null) {h = l;}`

$\alpha_2 \quad = \quad$ `if (h != null) {l = h;} else {l = new C();}`

The program α_1 satisfies flow(α_1, $\{l\}$, $\emptyset$). This is because an attacker cannot learn anything from running α_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 `l` in the pre-state is stored in `h` if `h` was not `null` (as the attacker knows the program).

Considering (only) α_2, an attacker who observes the (low) output variable `l` 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(α_2, $\emptyset$, $\{l\}$).

Ignoring the extra condition $Obj(R_1^s) \cap Obj(R_3^s) \subseteq Obj(R_2^s)$ 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 `l` in the pre-state to its value in the post-state: If `l` 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*

$$R = seq\_def\{iv\}(t_1, t_2, e)$$

where t_i are expression of type integer with no occurence 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 $next(n+1, x) \doteq y \leftrightarrow \exists z(next(n, x) \doteq z \wedge next(z) \doteq y)$. Then we may write

$$R = seq\_def\{i\}(0, \mathbf{this}.len, next(i, \mathbf{this}).v)$$

with len axiomatized by $next(len, \mathbf{this}) = \mathbf{null}$ and $\forall j(0 \leq j \wedge j < len \rightarrow next(j, \mathbf{this}) \neq \mathbf{null})$

If $R^s = \langle a_1, \ldots, a_{n-1} \rangle$ is the interpretation of observation R in state s the type of a_i will usually by a JAVA class or JAVA data type. But, the given definition does not impose this restriction.

In examples we will sometimes use a comma separated list of observations 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 α be a program, and let R_1, R_2 be observation expressions.*

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

Proof. The proof consists of a constructive definition of the formula ϕ_{α,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 ϕ_{α,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 $select_C(h, e, f)$. 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., `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 ϕ_{α,R_1,R_2}:

$$\begin{aligned}\phi_{\alpha,R_1,R_2} \equiv\ & \forall Heap\ h'_1, h_2, h'_2 \forall T o' \forall T_r r, r' \forall \ldots v', v^2, (v^2)' \ldots \\ & (Agree_{pre} \wedge \langle\alpha\rangle \text{save}\{s_2\} \wedge \text{in}\{s'_1\}\langle\alpha\rangle\text{save}\{s'_2\} \\ & \rightarrow (Agree_{post} \wedge Ext))\end{aligned}$$

To maintain readability we have used suggestive abbreviations:

1. $\{\text{in } s'_1\}\langle\alpha\rangle$ signals that an update $\{\texttt{heap} := h'_1 \mid\mid \texttt{this} := o' \mid\mid \ldots a_i := v' \ldots\}$ is placed before the modal operator. The a_i cover all relevant parameters and local variables.
2. The construct save$\{s_2\}$ abbreviates a conjunction of equations $h_2 = \texttt{heap}$, $r = \texttt{result}$, $\ldots$, $v^2 = a_i$, $\ldots$.
3. Analogously, save$\{s'_2\}$ stands for the primed version $h'_2 = \texttt{heap}$, $r' = \texttt{result}$, $\ldots$, $(v^2)' = a_i$, $\ldots$.
4. The shorthand $\{\text{in } s_2\}\{\text{in } s'_2\}E$ in front of a formula is resolved by (a) prefixing every occurence of a heap dependent expression e with the update $\{\texttt{heap} := h_2\}$ and (b) every primed expression e' with $\{\texttt{heap} := h'_2\}$.
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 ϕ_{α,R_1,R_2} is evaluated in state s_1.

In the following we will also use the notation R'_i, R^2_i, $(R^2_i)'$ for the terms obtained from R_i by replacing each state dependend designator v 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 $seqGet_{Any}(r,i)$ and also $t \sqsubseteq A$ for $instance_A(t)$.

We now supply the definitions of the abbreviations used above:

$$
\begin{array}{ll}
Agree_{pre} & \equiv R_1.\mathbf{length} \doteq R'_1.\mathbf{length} \\
& \wedge \\
& \forall i(0 \leq i < R_1.\mathbf{length} \rightarrow \\
& \bigwedge_{A \text{ in } \alpha}(exactInstance_A(R_1[i]) \leftrightarrow exactInstance_A(R'_1[i]))) \\
& \wedge \\
& \forall i((0 \leq i < R_1.\mathbf{length} \wedge R_1[i] \not\sqsubseteq Object \rightarrow R_1[i] \doteq R'_1[i]) \\
& \wedge \\
& \forall i,j(0 \leq i < j < R_1.\mathbf{length} \wedge R_1[i] \sqsubseteq Object \wedge R_1[j] \sqsubseteq Object \\
& \rightarrow (R_1[i] \doteq R_1[j] \leftrightarrow R'_1[i] \doteq R'_1[j])) \\
Agree_{post} & \equiv R^2_2.\mathbf{length} \doteq (R^2_2)'.\mathbf{length} \\
& \wedge \\
& \forall i(0 \leq i < R^2_2.\mathbf{length} \rightarrow \\
& \bigwedge_{A \text{ in } \alpha}(exactInstance_A(R^2_1[i]) \leftrightarrow exactInstance_A((R^2_1)'[i]))) \\
& \wedge \\
& \forall i((0 \leq i < R^2_2.\mathbf{length} \wedge R^2_2[i] \not\sqsubseteq Object \rightarrow R^2_2[i] \doteq (R^2_2)'[i]) \\
& \wedge \\
& \forall i,j(0 \leq i < j < R^2_2.\mathbf{length} \wedge R^2_2[i] \sqsubseteq Object \wedge R^2_2[j] \sqsubseteq Object \\
& \rightarrow (R^2_2[i] \doteq R^2_2[j] \leftrightarrow (R^2_2)'[i] \doteq (R^2_2)'[j])) \\
Ext & \equiv \forall i \forall j(0 \leq i < R_1.\mathbf{length} \wedge 0 \leq j < R^2_2.\mathbf{length} \wedge \\
& R_1[i] \sqsubseteq Object \wedge R^2_2[j] \sqsubseteq Object \wedge R_1[i] \doteq R^2_2[j] \\
& \rightarrow R'_1[i] \doteq (R^2_2)'[j])
\end{array}
$$

In many cases these definitions are much simpler. Frequently it is the case that R_i.**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_i is a variable of type *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 $flow(s_1,\alpha,R_1,R_2)$ fix states s'_1, s_2, s'_2 such that α started in s_1 terminates in s_2, α started in s'_1 terminates in s'_2, and agree(R_1, s_1, s'_1, π^1). We need to show that agree(R^2, s_2, s'_2, π^2) and π^2 is compatible with π^1.

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

$$(s_1,\beta) \models \text{in}\{s'_1\}\langle\alpha\rangle\text{save}\{s'_2\}.$$

Thus, our assumption $s_1 \models \phi_{\alpha,R_1,R_2}$ implies $(s_1,\beta) \models \{\text{in } s_2\}\{\text{in } s'_2\}(Agree_{post} \wedge Ext)$. The part $(s_1,\beta) \models \{\text{in } s_2\}\{\text{in } s'_2\}Agree_{post}$ implies agree(R_2, s_2, s'_2, π^2) while $(s_1,\beta) \models \{\text{in } s_2\}\{\text{in } s'_2\}Ext$ guarantees that π^2 is compatible with π^1. In total $flow(s_1,\alpha,R_1,R_2)$ has been shown.

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

$$
\begin{array}{l}
(s_1,\beta) \models Agree_{pre} \wedge \langle\alpha\rangle\text{save}\{s_2\} \wedge \text{in}\{s'_1\}\langle\alpha\rangle\text{save}\{s'_2\} \\
\qquad \rightarrow \{\text{in } s_2\}\{\text{in } s'_2\}(Agree_{post} \wedge Ext)
\end{array}
$$

$$
\begin{aligned}
\phi_{m5(),R,R} \equiv\ & \forall Heap\ h'_1, h_2, h'_2 \forall C\ o' \forall x', x^2, (x^2)', y', y^2, (y^2)'(\\
& (x \doteq y \leftrightarrow x' \doteq y' \wedge \\
& \langle m_5()\rangle(x^2 \doteq x \wedge y^2 \doteq y) \wedge \\
& \{\mathbf{this} := o', x := x', y := y'\}\langle m_5()\rangle((x^2)' \doteq x \wedge (y^2)' \doteq y)) \\
& \rightarrow \\
& (x^2 \doteq y^2 \leftrightarrow (x^2)' \doteq (y^2)' \wedge \\
& x \doteq x^2 \rightarrow x' \doteq (x^2)' \ \wedge y \doteq x^2 \rightarrow y' \doteq (x^2)' \wedge \\
& x \doteq y^2 \rightarrow x' \doteq (y^2)' \ \wedge y \doteq y^2 \rightarrow y' \doteq (y^2)'))
\end{aligned}
$$

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

We may assume $(s_1, \beta) \models Agree_{pre} \wedge \langle\alpha\rangle\text{save}\{s_2\} \wedge \text{in}\{s'_1\}\langle\alpha\rangle\text{save}\{s'_2\}$ 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 ϕ_{α,R_1,R_2}. It is easy to see that $(s_1, \beta) \models Agree_{pre}$ implies agree(R_1, s_1, s'_1, π^1). Now, $(s_1, \beta) \models \langle\alpha\rangle\text{save}\{s_2\}$ implies in particular that α started in s_1 terminates. Let us call the final state s_2. Likewise, $(s_1, \beta) \models \text{in}\{s'_1\}\langle\alpha\rangle\text{save}\{s'_2\}$ implies first $(s'_1, \beta) \models \langle\alpha\rangle\text{save}\{s'_2\}$ and then that α 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 $flow(s_1, \alpha, R_1, R_2)$ and conclude agree(R_2, s_2, s'_2, π^2) and π^2 is compatible with π^1. Except termination we obtain from $(s_1, \beta) \models \langle\alpha\rangle\text{save}\{s_2\}$ also $\beta(h_2) = s_2(\mathbf{heap})$, $\beta(r) = s_2(\mathbf{result})$, and $\beta(v^2) = s_2(v)$ for all other relevant program variables. From $(s'_1, \beta) \models \langle\alpha\rangle\text{save}\{s'_2\}$ we obtain likewise $\beta(h'_2) = s'_2(\mathbf{heap})$, $\beta(r') = s'_2(\mathbf{result})$, and $\beta((v^2)') = s'_2(v)$ for all other relevant program variables. From agree(R_2, s_2, s'_2, π^2) we thus can conclude

$$(s_1, \beta) \models \{\text{in } s_2\}\{\text{in } s'_2\}Agree_{post}$$

and from the fact that π^2 is compatible with π^1 we get

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

In total we have shown $s_1 \models \phi_{\alpha,R_1,R_2}$, as desired. □

Example 8. To illustrate the construction used in the proof of Theorem 1 by an example. We reconsider method $m_5()$ from Figure 4 on page 10 and $R = \langle x, y\rangle$, which is shorthand for

$$
\begin{aligned}
& seqConcat(\ seqSingleton(select_C(\mathbf{heap}, \mathbf{null}, x)), \\
& \qquad\qquad seqSingleton(select_C(\mathbf{heap}, \mathbf{null}, y)))
\end{aligned}
$$

Note, that we have $(R.\mathbf{length})^s = 2$ for all states s and the exact type of both fields x, y is always C. Thus $Agree_{pre}$ equals $x \doteq y \leftrightarrow x' \doteq y'$. $Agree_{post}$ equals $x^2 \doteq y^2 \leftrightarrow (x^2)' \doteq (y^2)'$ and Ext is the conjunction $x \doteq x^2 \rightarrow x' \doteq (x^2)' \ \wedge y \doteq x^2 \rightarrow y' \doteq (x^2)' \ \wedge x \doteq y^2 \rightarrow x' \doteq (y^2)' \ \wedge y \doteq y^2 \rightarrow y' \doteq (y^2)'$. Figure 5 shows the complete formula $\phi_{m5(),R,R}$.

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

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

We say that M is a modifies set *for α, denoted by $mod(\alpha, M)$, iff for all states s the following holds: if there is a state s' such that α 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 $v \notin V$ we obtain $v^s = v^{s'}$.*

5.2 A Simplified Version

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

Proof. From the assumption agree(R, s, s', π) we get by definition:

1. $R^s = \langle a_0, \dots a_{n-1} \rangle$, $R^{s'} = \langle a'_0, \dots a'_{n-1} \rangle$,
2. for all $0 \le i < n$: $type(a_i) = type(a'_i)$,
3. for all $0 \le i < n$ such that $type(a_i) \not\sqsubseteq Object$: $a_i = a'_i$,
4. for all $0 \le i < n$ such that $type(a_i) \sqsubseteq Object$: $a_i = null \Leftrightarrow a'_i = null$,
5. for all $0 \le i < n$ such that $type(a_i) \sqsubseteq Object$ and a_i is an object of array type : $a_i.\mathbf{length}^s = a'_i.\mathbf{length}^{s'}$,
6. for all $0 \le i < j < n$ such that $type(a_i) \sqsubseteq Object$ and $type(a_j) \sqsubseteq Object$: $a_i = a_j \Leftrightarrow a'_i = a'_j$
7. $\pi(a_i) = a'_i$

By the basic properties of isomorphism, see Lemma 1, we obtain using notation from Definition 11:

1. $R^{s'} = \langle a'_0, \dots a'_{n-1} \rangle$, $R^{\rho(s')} = \langle \rho(a'_0), \dots \rho(a'_{n-1}) \rangle$,
2. for all $0 \le i < n$: $type(a'_i) = type(\rho(a'_i))$,
3. for all $0 \le i < n$ such that $type(a_i) \not\sqsubseteq Object$: $\rho(a'_i) = a'_i$ since isomorphisms are the identity outside $Object^{\mathcal{D}}$,
4. for all $0 \le i < n$ such that $type(a_i) \sqsubseteq Object$: $a'_i = null \Leftrightarrow \rho(a'_i) = null$,
5. for all $0 \le i < n$ such that $type(a_i) \sqsubseteq Object$ and a_i is an object of array type : $a'_i.\mathbf{length}^{s'} = \rho(a'_i).\mathbf{length}^{\rho(s')}$,
6. for all $0 \le i < j < n$ such that $type(a_i) \sqsubseteq Object$ and $type(a_j) \sqsubseteq Object$: $a'_i = a'_j \Leftrightarrow \rho(a'_i) = \rho(a'_j)$
7. $\rho \circ \pi(a_i) = \rho(a'_i)$

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

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 α is run in two states that agree on the *low* values then the states that are reached by executing α 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 α *be a program and* R_1 *and* R_2 *be two observation expressions (of type Seq)*

We say that α *allows simple information flow only from* R_1 *to* R_2 *when started in* s_1*, denoted by flow*$^*(s_1, \alpha, R_1, R_2)$*, iff, for all states* s'_1, s_2, s'_2 *such that* α *started in* s_1 *terminates in* s_2 *and* α *started in* s'_1 *terminates in* s'_2*, we have*

if $agree(R_1, s_1, s'_1, id)$
then $agree(R_2, s_2, s'_2, \pi^2)$ *and*
$\pi^2(o) = o$ *for all* $o \in obj^{s_2}(R_2) \cap obj^{s_1}(R_1)$ *with* $created^{s_1}(o) = tt$.

Note, that $agree(R_1, s_1, s'_1, id)$ *implies in particular* $obj^{s_1}(R_1) = obj^{s'_1}(R_1)$ *since* $\pi^1 = id$ *is a bijection from* $obj^{s_1}(R_1)$ *onto* $obj^{s'_1}(R_1)$.

Lemma 9. *For all programs α, any two observation expressions R_1 and R_2 , and any state s_1*

$$flow^*(s_1, \alpha, R_1, R_2) \quad \Rightarrow \quad flow(s_1, \alpha, R_1, R_2)$$

Since the reverse implication is obviously true Lemma 9 entails that $flow$ and $flow^*$ are equivalent.

Proof. To prove flow(s_1, α, R_1, R_2) we fix, in addition to s_1, states s'_1, s_2, s'_2 such that α started in s_1 terminates in s_2 and α started in s'_1 terminates in s'_2, and assume agree(R_1, s_1, s'_1, π^1). We need to show agree(R_2, s_2, s'_2, π^2) with π^2 extending π^1.

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

From agree(R_1, s_1, s'_1, π^1) we conclude agree$(R_1, s_1, \rho(s'_1), \rho \circ \pi^1)$ by Lemma 8. Since ρ extends $(\pi^1)^{-1}$ we have agree$(R_1, s_1, \rho(s'_1), id)$. As noted in Lemma 5 there is a state s'_3 such that α started in $\rho(s'_1)$ terminates in s'_3. This enables us to make use of the assumption flow$^*(s_1, \alpha, R_1, R_2)$ and conclude agree(R_2, s_2, s'_3, π^3). Furthermore $\pi^3(o) = o$ for all $o \in obj^{s_1}(R_1) \cap obj^{s_2}(R_2)$.

Applying Lemma 5 to the inverse isomorphism ρ^{-1} to the situation that α started in $\rho(s'_1)$ terminates in s'_3, we obtain an automorphism ρ' such that α started in $\rho^{-1}(\rho(s'_1)) = s'_1$ terminates in $\rho'(s'_3)$ and ρ' coincides with ρ^{-1} on all objects in $E = \{o \in Object^{\mathcal{D}} \mid created^{\rho(s_1)}(o) = tt\}$.

Again using Lemma 8, this time for the isomorphism ρ', we obtain from agree(R_2, s_2, s'_3, π^3) also agree$(R_2, s_2, \rho'(s'_3), \rho' \circ \pi^3)$. Since α is a deterministic program and we have already defined s'_2 to be the final state of α when started in s_2 we get $s'_2 = \rho'(s'_3)$ and thus 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 π^1, i.e., for every $o \in obj^{s_1}(R_1) \cap obj^{s_2}(R_2)$ with $created^{s_1}(o) = tt$ we need to show $\rho' \circ \pi^3(o) = \pi^1(o)$.

By the definition of isomorphic states we obtain from $created^{s_1}(o) = tt$ also $created^{\rho(s_1)}(o) = tt$. Thus we can infer $\rho'(o) = \rho^{-1}(o)$ and by choice of ρ 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 $\pi^2(R^{s_2}[i]) = R^{s'_2}[i]$ for all i such that $0 \leq i < R^{s_2}.length = R^{s'_2}.length$. On the other hand π^3 is defined by $\pi^3(R^{s_2}[i]) = R^{s'_3}[i]$ for all i such that $0 \leq i < R^{s_2}.length = R^{s'_3}.length$. Thus $\rho' \circ \pi^3(R^{s_2}[i]) = \rho'(R^{s'_3}[i]) = R^{\rho'(s'_3)}[i])$. Since, as noted above, $\rho'(s'_3) = s'_2$ we have arrived at $\rho' \circ \pi^3(R^{s_2}[i]) = R^{s'_2}[i]$. □

Lemma 9 leads to the following corollary to Theorem 1.

Corollary 1. *Let α be a program, and let R_1, R_2 be observation expressions.*

There is a formula ϕ_{α,R_1,R_2} in JavaDL *making use of self-composition such that:* $s_1 \models \phi_{\alpha,R_1,R_2}$ *iff* $flow(s_1, \alpha, R_1, R_2)$
with

$$\begin{aligned}\phi_{\alpha,R_1,R_2} \equiv\ & \forall Heap\ h'_1, h_2, h'_2 \forall T o' \forall T_r r, r' \forall \ldots v', v^2, (v^2)' \ldots \\ & (Agree_{pre}\ \wedge \langle\alpha\rangle save\{s_2\} \wedge in\{s'_1\}\langle\alpha\rangle save\{s'_2\} \\ & \to (Agree_{post} \wedge Ext))\end{aligned}$$

$Agree_{post}$ and Ext remain as in the proof of Theorem 1 but $Agree_{pre}$ simplifies to

$$\begin{aligned}Agree_{pre} \equiv\ & R_1.\mathbf{length} \doteq R'_1.\mathbf{length} \\ & \wedge \\ & \forall i((0 \leq i < R_1.\mathbf{length} \to R_1[i] \doteq R'_1[i])\end{aligned}$$

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

$$\begin{array}{l} seqLen(R_1) \geq seqLen(R_2) \wedge \\ \forall i (0 \leq i \wedge i < seqLen(R_2) \rightarrow seqGet(R_2, i) \doteq seqGet(R_1, i))) \end{array}$$

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

$$\begin{array}{l} \forall i (0 \leq i \wedge i < seqLen(R_1) \rightarrow R_1[i] \doteq R'_1[i]) \\ \rightarrow \\ \forall j (0 \leq j \wedge j < seqLen(R_2) \rightarrow R_2[j] \doteq R'_2[j]) \end{array}$$

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 *Seq*. An interesting instantiation is given in the next simple lemma.

Lemma 12. *Let $R_1 = seq\_def\{u\}(t_1^1, t_2^1, e^1)$ and $R_2 = seq\_def\{w\}(t_1^2, t_2^2, e^2)$*
Then $R_1 \supseteq R_2$ is equivalent to the validity of the formula

$$\begin{array}{l} \forall u (t_1^1 \leq u \wedge u < t_2^1) \rightarrow e^1[u] \doteq (e^1)'[u]) \\ \rightarrow \\ \forall w (t_1^2 \leq u \wedge u < t_2^2) \rightarrow e^2[w] \doteq (e^2)'[w]) \end{array}$$

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) $\mathcal{C}_{m::T}$ for method m declared in type T is satisfied if in any state the formula flow(`this.m(ā)`, R_1, R_2) from Definition 14 is true, where program α 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 $\phi_{\mathtt{this.m(\bar{a})},R_1,R_2}$ presented after Theorem 1 we can read off the structure of the formalisation of flow($\mathtt{this.m(\bar{a})}, R_1, R_2$) in the general case:

$$\begin{aligned}
\psi_{\mathcal{C}_{m::T}} &\equiv \forall \mathit{Heap}\, h_1\ \forall T\ o\ \forall \bar{A}\ \bar{a}\ \forall A_{n+1}\ r\ \{\text{in } s_1\}\phi_{\mathtt{this.m(\bar{a})},R_1,R_2} \\
&\equiv \forall \mathit{Heap}\, h_1, h_1', h_2, h_2'\ \forall T\ o, o'\ \forall \bar{A}\ \bar{a}, \bar{a}'\ \forall A_{n+1}\ r, r' \\
&\qquad\qquad \{\text{in } s_1\}[\mathtt{this.m(\bar{a})}](\text{save } s_2) \land \{\text{in } s_1'\}[\mathtt{this.m(\bar{a})}](\text{save } s_2') \\
&\qquad\quad \land \{\text{in } s_1\}\{\text{in } s_1'\}(\mathit{ED}^{R_1} \land \mathit{EO}^{R_1}) \\
&\qquad\quad \to \{\text{in } s_2\}\{\text{in } s_2'\}(\mathit{ED}^{R_2} \land \mathit{EO}^{R_2} \land \mathit{Old}^{R_1,R_2})
\end{aligned}$$

Here we use the following suggestive abbreviations: (1) The shorthand $\{\text{in } s_1'\}\varphi$ signals that an update $\{\mathtt{heap} := h_1' \mid\mid \mathtt{this} := o' \mid\mid \ldots a_i := x_1' \ldots\}$ is placed before φ. The a_i cover all other relevant parameters and local variables. (2) The construct (save s_2) abbreviates a conjunction of equations $h_2 = \mathtt{heap}$, $r = \mathtt{result}$, $\ldots, x_2 = a_i, \ldots$. Analogously, (save s_2') stands for the primed version $h_2' = \mathtt{heap}$, $r' = \mathtt{result}, \ldots, x_2' = a_i, \ldots$. (3) The shorthand $\mathit{ED}^{R_1} \land \mathit{EO}^{R_1}$ abbreviates a formula which is valid iff s_1 and s_2 agree on R_1 in the sense of Definition 13. Analogously, $\mathit{ED}^{R_2} \land \mathit{EO}^{R_2}$ 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 guaranties that the isomorphism defined by $\mathit{ED}^{R_2} \land \mathit{EO}^{R_2}$ is an extension of the one defined by $\mathit{ED}^{R_1} \land \mathit{EO}^{R_1}$.

The difficulty in the application of method contracts for information flow arises from the fact that $\psi_{\mathcal{C}_{m::T}}$ 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 $\psi_{\mathcal{C}_{m::T}}$ has been proven valid for some method m, then it can be used as a lemma in the proof of $\psi_{\mathcal{C}_{m_2::T}}$ for another method $\mathtt{m}_2$. We extend the standard functional method contract rule by adding the predicate $MC_{T::m}(o, \bar{a}, h_1, \mathit{res}, h_2)$ to the antecedent of each premiss. 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 $\{\mathit{in}\ s_1\}[\mathtt{this.m(\bar{a})}]\{\mathit{save}\ s_2\}$ is that $MC_{T::m}$ is not decomposed by the proof search strategy. We introduce the following rule schema to make use of $\psi_{\mathcal{C}_{m::T}}$ as a lemma:

FlowContract

$$\frac{\begin{array}{l} MC_{T::m}(o, \bar{a}, h_1, \mathit{res}, h_2), MC_{T::m}(o', \bar{a}', h_1', \mathit{res}', h_2'), \\ \{\text{in } s_1\}\{\text{in } s_1'\}(\mathit{ED}^1 \land \mathit{EO}^1) \to \{\text{in } s_2\}\{\text{in } s_2'\}(\mathit{ED}^2 \land \mathit{EO}^2 \land \mathit{Old}) \end{array} \Longrightarrow}{MC_{T::m}(o, \bar{a}, h_1, \mathit{res}, h_2), MC_{T::m}(o', \bar{a}', h_1', \mathit{res}', h_2') \Longrightarrow}$$

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

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 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.

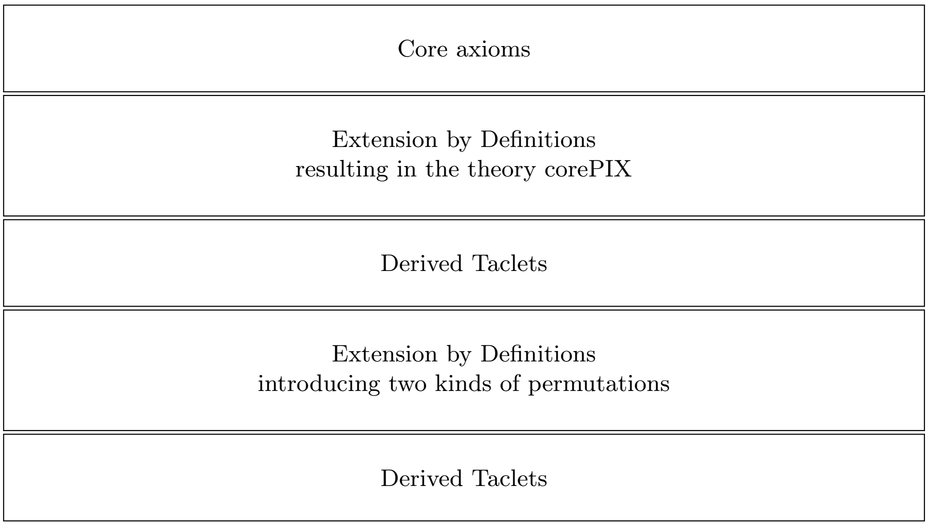

Figure 6. Structure of the file `seq.key`

A.1 The Core Theory `seqCore`

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

```
any any::seqGet(Seq, int)
any seqGetOutside
int seqLen(Seq)
```

and the generalized quantifier $seq\_def\{\}(,,)$. 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, ri, le are of type *int*, variable a is of type *any*. Furthermore $\phi\{t/u\}$ denotes the formula obtained from ϕ 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 $any :: seqGet(s, i)$.

A finite sequence s is represented as a function $i \rightsquigarrow s[i]$ from *int* into *any* plus a length $seqLen(s)$. 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

1. $\forall s(0 \leq seqLen(s))$
2. $\forall s_1 \forall s_2(s_1 \doteq s_2 \leftrightarrow$
   $seqLen(s_1) \doteq seqLen(s_2) \wedge \forall i(0 \leq i < seqLen(s_1) \rightarrow s_1[i] \doteq s_2[i]))$
3. $\forall i \forall ri \forall le($
   $((0 \leq i \wedge i < ri - le) \rightarrow seq\_def\{u\}(le, ri, t)[i] \doteq t\{(le + i)/u\})$
   $\wedge$
   $(\neg(0 \leq i \wedge i < ri - le) \rightarrow seq\_def\{u\}(le, ri, t)[i] \doteq seqGetOutside))$
4. $\forall ri \forall le($
   $(le < ri \rightarrow seqLen((seq\_def\{u\}(le, ri, t)) \doteq ri - li)$
   $\wedge$
   $(ri \leq le \rightarrow seqLen(seq\_def\{u\}(le, ri, t)) \doteq 0))$

Figure 7. Core axioms in mathematical notation

of finite sequences. Thus, the values $s[i]$ for $i < 0$ or $seqLen(s) \leq i$ are irrelevant in this respect. In particular, there is at most one empty sequence.

The main difference of our axiomatization of Seq over the traditional abstract datatype approach is the use of the generalized quantifier $seq\_def\{u\}(le, ri, t)$ with the intented 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 `seqCore` we will construct a non-empty set **MSeq** of models $\mathcal{M}$ such that $\mathcal{M} \models \phi$ for every axiom ϕ in the list of Figure A.1. Of course, one model $\mathcal{M}$ 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 `seqCore` is complete with respect to the class of structures **MSeq**, i.e., for every formula ψ with $\mathcal{M} \models \psi$ for every $\mathcal{M} \in$ **MSeq** we ask if `seqCore` $\vdash \psi$?.

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

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

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

$$Seq^{\mathcal{D}} \quad := \quad D_{Seq} \quad := \quad \bigcup_{n \geq 1} D^n_{Seq}$$

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_{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 $\langle 0, \langle\emptyset, seqEmpty, null\rangle, tt\rangle$ is a perfect element in D_{Seq}.

Definition 20 (MSeq). *A structure $\mathcal{M}$ with universe $D_{Seq} \cup D^0_{Seq}$ belongs to the set* **MSeq** *if it satiesfies the following restrictions, where i, ri, le are integers, a_k, a elements of D^0_{Seq} and $s \in S_{Seq}$ and e a term of type Any:*

1. $seq\_def\{iv\}(le, ri, e)^{\mathcal{M},\beta} = \begin{cases} \langle a_0, \ldots a_{k-1}\rangle & \text{if } ri - le = k > 0 \\ & \text{and } a_i = e^{\mathcal{M},\beta_i} \\ & \text{with } \beta_i = \beta[i/iv] \\ seqGetOutside^{\mathcal{M}} & \text{otherwise} \end{cases}$
2. $seqGet^{\mathcal{M}}_{any}(\langle a_0, \ldots, a_{n-1}\rangle, i) = \begin{cases} a_i & \text{if } 0 \leq i < n \\ seqGetOutside^{\mathcal{M}} & \text{otherwise} \end{cases}$
3. $seqLen^{\mathcal{M}}(\langle a_0, \ldots, a_{n-1}\rangle) = n$
4. $seqGetOutside^{\mathcal{M}} \in D_{Seq}$ *arbitrary.*

As an example of item 1 we present

$$seq\_def\{iv\}(-15, -10, 20 + iv)^{\mathcal{D}} = \langle 5, 6, 7, 8, 9\rangle.$$

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 $seq\_def\{iv\}(t_1, t_2, e)$ is not determined by the structure $\mathcal{M}$ alone, the variable assignment β 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 $t\{(le + i)/u\})^{\mathcal{M},\beta_i}$ evaluates to the same value as $t^{\mathcal{M},\beta'_i}$ with
$\beta'(v) = \begin{cases} \beta(v) & \text{if } v \text{ is different from } u \\ (le + i)^{\mathcal{M},\beta_i} & \text{if } v \equiv u \end{cases}$
Axioms 3 and 4 follow directly from the definitions of seq_def in (1) and $seqGet_{any}$ in (2) of Definition 20. □

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

```
Int seqDepth(Any)
```

and the axioms

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

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*

$$seqDepth^{\mathcal{M}_D}(a) = \text{ the unique } n \text{ with } a \in D^n_{Seq}$$

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

Lemma 14. *The theory* `seqCoreDepth` *is consistent.*

Proof For every structure $\mathcal{N}$ in **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 Int with its usual functions and predicates and there is a theory T_{int} such that for any model $\mathcal{D}$ of T_{int} we have $D^0_{Seq} = \mathbb{Z}$.*
The theory $T_{int} \cup$ `seqCoreDepth` is complete with respect to **MSeqD***.*
In detail this means:
Let ϕ be a formula in the signature of `seqCoreDepth` such that $\mathcal{M} \models \phi$ for all $\mathcal{M}$ in **MSeqD** *then*

$$T_{int} \cup \texttt{seqCoreDepth} \vdash \phi$$

Proof The proof proceeds by contradiction. We assume $\mathcal{N} \models \phi$ for all $\mathcal{N}$ in **MSeqD**, but $T_{int} \cup$ `seqCoreD` $\not\vdash \phi$. Thus there is a structure $\mathcal{N}_0$ with $\mathcal{N}_0 \models T_{int} \cup$ `seqCoreD` but $\mathcal{N}_0 \models \neg\phi$. We define a mapping $F : N_0 \to D_{seq} \cup \mathbb{Z}$, i.e., from the universe N_0 of $\mathcal{N}_0$ into the common universe of all structures in **MSeqD**. $F(a)$ is defined by induction on $seqDepth(a)$. By assumption $\{a \in N \mid seqDepth^{\mathcal{N}_0}(a) = 0\} = Int^{\mathcal{N}_0} = \mathbb{Z}$ and we let F be the identity on these elements. For a with $seqDepth^{\mathcal{N}_0}(a) = n + 1$ we define inductively

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

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

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 $a \in D^n_{Seq}$, 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_{Int}, but we may fairly assume that there is a term t_n with $F(t_n^{\mathcal{N}_0}) = n$, e.g. $t_n = \underbrace{1 + \ldots + 1}_{n\ times}$, $t_0 = 0$ or $t_n = \underbrace{-1 - \ldots - 1}_{n\ times}$. In the inductive step of the argument we assume that the claim is true for all $a \in D^n_{Seq}$ and fix $s = \langle s_0, \ldots s_{k-1} \rangle \in D^{n+1}_{Seq}$. Since $s_i \in D^n_{Seq}$ for all $0 \leq i < k$ there are terms t_i with $F(t_i^{\mathcal{N}_0}) = s_i$. Now, $F((seq\_def\{u\}(0, k, t))^{\mathcal{N}_0}) = s$ with

$$\begin{aligned} t = \ & \textbf{if } u = 0 \textbf{ then } t_0 \textbf{ else} \\ & (\textbf{if } u = 1 \textbf{ then } t_1 \textbf{ else} \\ & \ldots \\ & (\textbf{if } u = k-1 \textbf{ then } t_{k-1}) \ldots) \end{aligned}$$

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

$$F : N \to M \text{ is a bijection} \tag{6}$$

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

$$\begin{aligned} seq\_def\{u\}(i, j, e)^{\mathcal{M}, F(\beta)} &= F(seq\_def\{u\}(i, j, e)^{\mathcal{N}_0, \beta}) \\ seqGet^{\mathcal{M}}(F(s), i) &= F(seqGet^{\mathcal{N}_0}(s, i)) \\ seqGetOutside^{\mathcal{M}} &= F(seqGetOutside^{\mathcal{N}_0}) \\ seqLen^{\mathcal{M}}(F(s)) &= F(seqLen^{\mathcal{N}_0}(s)) \\ seqDepth^{\mathcal{M}}(F(a)) &= F(seqDepth^{\mathcal{N}_0}(a)) \end{aligned}$$

By construction F is an isomorphims from $\mathcal{N}_0$ onto $\mathcal{M}$. Thus $\mathcal{N}_0 \models \neg\phi$ implies $\mathcal{M} \models \neg\phi$.

The proof plan is to show that $\mathcal{M}$ is in **MSeqD**. This will contradict the assumption that ϕ be true in all structures in **MSeqD**.

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

$$\text{For all terms } e \text{ and variable assignments } \beta \quad F(e^{\mathcal{N}_0,\beta}) = e^{\mathcal{M},F(\beta)} \tag{7}$$

$$\text{For any formula } \phi \text{ and variable assignments } \beta \quad (\mathcal{N}_0,\beta) \models \phi \Leftrightarrow (\mathcal{M},F(\beta)) \models \phi \tag{8}$$

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 $e = seqLen(x)$ that leads to the equality $F(seqLen^{\mathcal{N}_0}(s)) = seqLen^{\mathcal{M}}(F(s))$.

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

To show $\mathcal{M} \in$ **MSeqD** we have to check Definitions 20 and 22 item by item.

1.

$$\begin{array}{rll}
seq_def\{u\}(i,j,e)^{\mathcal{M},F(\beta)} & = F(seq_def\{u\}(i,j,e)^{\mathcal{N}_0,\beta}) & \text{iso transfer} \\
& = \langle a_0, \ldots a_{k-1}\rangle & \text{def } F \\
& \quad a_r = F(seq_def\{u\}(i,j,e)^{\mathcal{N}_0,\beta}[r]) & \\
& \quad a_r = F(e^{\mathcal{N}_0,\beta_r}) & \text{axiom 3, in Fig. } A.1 \text{ case } j > i \\
& \quad a_r = e^{\mathcal{M},F(\beta)_r} & \text{eqn (7)} \\
& \quad a_r = F(seqGetOutside^{\mathcal{N}_0}) & \text{axiom 3, in Fig. } A.1 \text{ case } j \le i \\
& \quad a_r = seqGetOutside^{\mathcal{M}} & \text{iso transfer}
\end{array}$$

2.

$$\begin{array}{rll}
seqGet^{\mathcal{M}}_{any}(\langle a_0, \ldots a_{n-1}\rangle, i) & = F(seqGet^{\mathcal{N}_0}_{any}(s,i)) & \text{iso transfer} \\
& \quad \text{with } F(s) = \langle a_0, \ldots a_{n-1}\rangle & \\
& \quad seqLen^{\mathcal{N}_0} = n \text{ and} & 0 \le r \\
& \quad a_r = F(seqGet^{\mathcal{N}_0}_{any}(s,r)) & r < n
\end{array}$$

For $0 \le i < n$ this gives the desired result $seqGet^{\mathcal{M}}_{any}(\langle a_0, \ldots a_{n-1}\rangle, i) = a_i$. For $i < 0$ or $n \le i$ we argue that $\forall s \forall i((i < 0 \vee seqLen(s) \le i) \to s[i] = seqGetOutside)$ is a logical consequence of `seqCore` and thus true in $\mathcal{N}_0$. In this case we get the following chain of reasoning

$$\begin{array}{rll}
seqGet^{\mathcal{M}}_{any}(\langle a_0, \ldots a_{n-1}\rangle, i) & = F(seqGet^{\mathcal{N}_0}_{any}(s,i)) & \text{iso transfer} \\
& = F(seqGetOutside^{\mathcal{N}_0}) & \\
& = seqGetOutside^{\mathcal{M}} & \text{iso transfer}
\end{array}$$

3.

$$\begin{array}{rll}
seqLen^{\mathcal{M}}(\langle a_0, \ldots a_{n-1}\rangle) & = seqLen^{\mathcal{N}_0}(s) & \text{iso transfer} \\
& \quad \text{with } F(s) = \langle a_0, \ldots a_{n-1}\rangle &
\end{array}$$

Now, we get from the definition of F and $F(s) = \langle a_0, \ldots a_{n-1}\rangle$ immediately $seqLen^{\mathcal{N}_0}(s) = n$.

4. Nothing to show here.
5. By isomorphic transfer we defined $seqDepth^{\mathcal{M}}(F(a)) = seqDepth^{\mathcal{N}_0}(a)$. By definition of F we know $F(a) \in D_{Seq}^{seqDepth^{\mathcal{N}_0}(a)}$. Thus, the restriction on $seqDepth^{\mathcal{M}}$ in Definition 22 is satisfied.

□

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 indroduced by defining axioms.

```
alpha alpha::seqGet(Seq, int) Seq seqEmpty
Seq seqSingleton(any)         Seq seqConcat(Seq, Seq)
Seq seqSub(Seq, int, int)     Seq seqReverse(Seq)
int seqIndexOf(Seq, any)
```

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. $\forall s \forall i (alpha :: seqGet(s, i) \doteq (alpha)any :: seqGet(s, i))$
   or in shorthand
   $\forall s \forall i (alpha :: seqGet(s, i) \doteq (alpha)s[i])$
2. $seqEmpty \doteq seq\_def\{u\}(0, 0, 1)$
3. $\forall x (seqSingleton(x) \doteq seq\_def\{u\}(0, 1, x)$
4. $\forall s_1, s_2 (seqConcat(s_1, s_2) \doteq seq\_def\{u\}(0, seqLen(s_1) + seqLen(s_2),$
   $\quad \textbf{if } u < seqLen(s_1)$
   $\quad \textbf{then } s_1[u] \textbf{ else } s_2[u - seqLen(s_1)]))$
5. $\forall s \forall re, le (seqSub(s, le, ri) \doteq seq\_def\{u\}(le, ri, s[u]))$
6. $\forall s (seqReverse(s) \doteq seq\_def\{u\}(0, seqLen(s), s[seqLen(s) - u]))$
7. $\forall s \forall a \forall j ($
   $(0 \leq j \wedge j < seqLen(s) \wedge s[j] \doteq a \wedge \forall k (0 \leq k \wedge k < j \rightarrow s[k] \neq a))$
   $\rightarrow seqIndexOf(s, a) \doteq j)$

Variables s, s_1, s_2 are of type Seq, variable x of type Any and i, ri, le are of type Int.

Figure 8. First Set of Extentions by Definition

The family of function symbols `alpha alpha::seqGet(Seq, int)` defined in axiom 1 is nessecary 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 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, $(int)s[i] = (int)any :: seqGet(s, i)$. For ease of use function symbols `alpha alpha::seqGet(Seq, int)` were added for every type $alpha$.

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

Let us call the new theory `seqCore1`. At this point it is important to know, whether `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 extenstion 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_{Σ_i}.*
T_1 is called a conservative extension *of T_0 if for all sentences $\phi \in Fml_{\Sigma_0}$:*

$$T_0 \vdash \phi \Leftrightarrow T_1 \vdash \phi$$

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_{Σ_i}.*
T_1 is called a semantic conservative extension *of T_0 if*

1. *for all Σ_1-structures $\mathcal{M}_1$ with $\mathcal{M}_1 \models T_1$ the restriction $\mathcal{M}_0$ of $\mathcal{M}_1$ to Σ_0 is a model of T_0, in symbols*

$$\mathcal{M}_1 \models T_1 \Rightarrow (\mathcal{M}_1 \upharpoonright \Sigma_0) \models T_0$$

2. *for every Σ_0-structure $\mathcal{M}_0$ with $\mathcal{M}_0 \models T_0$ there is a Σ_1-expansion $\mathcal{M}_1$ of $\mathcal{M}_0$ with $\mathcal{M}_1 \models T_1$.*

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 preceeding definition is automatically true.

Lemma 16. *Let $\Sigma_0 \subseteq \Sigma_1$ be signatures, and T_i sets of sentences in Fml_{Σ_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 ϕ be a sentence in Fml_{Σ_0} with $T_0 \vdash \phi$. Let $\mathcal{M}_1$ be an arbitrary Σ_1-structure. By assumption $(\mathcal{M}_1 \upharpoonright \Sigma_0) \models T_0$. Thus we also have $(\mathcal{M}_1 \upharpoonright \Sigma_0) \models \phi$. By the coincidence lemma we also have $\mathcal{M}_1 \models \phi$. In total we have shown $T_1 \vdash \phi$. Now, assume $T_1 \vdash \phi$. If $\mathcal{M}_0$ is an arbitrary Σ_0-structure there is by the assumption an expansion of $\mathcal{M}_0$ to a Σ_1-structure $\mathcal{M}_1$. From $T_1 \vdash \phi$ we thus get $\mathcal{M}_1 \models \phi$. The coincidence lemma tells us again that also $\mathcal{M}_0 \models \phi$. In total we arrive at $T_o \vdash \phi$. □

Lemma 17 (Extension by Definition). *Let $\Sigma_0 \subseteq \Sigma_1$ be signatures, $T_0 \subseteq T_1$ sets of sentences in Fml_{Σ_0} respectively in Fml_{Σ_1}. Further assume that all sentences in $T_1 \setminus T_0$ are of the form*

$$\begin{array}{ll} \forall \bar{x}(f(\bar{x}) \doteq t) & f \in \Sigma_1 \subseteq \Sigma_0 \ t \ \textit{a term in} \ \Sigma_0 \\ \forall \bar{x}(p(\bar{x}) \leftrightarrow \phi(\bar{x}) & p \in \Sigma_1 \subseteq \Sigma_0 \ \phi \ \textit{a formula in} \ \Sigma_0 \end{array}$$

Then T_1 is a semantic conservative extension of T_0.

Proof If $\mathcal{M}_0$ is a Σ_0-model of T_0 we obtain an Σ_1-expansion $\mathcal{M}_1$ by simply setting

$$f^{\mathcal{M}_1}(\bar{a}) = t^{\mathcal{M}_0}(\bar{a})$$

and

$$p^{\mathcal{M}_1}(\bar{a}) \Leftrightarrow \mathcal{M}_0 \models \phi[\bar{a}]$$

□

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 Fml_{Σ_0} respectively in Fml_{Σ_1}. Further assume that all sentences in $T_1 \setminus T_0$ are of the form*

$\forall \bar{x} \forall y(\psi \to f(\bar{x}) \doteq y)$ $f \in \Sigma_1 \subseteq \Sigma_0$
ψ *a formla in* Σ_0

such that
$T_0 \vdash \forall \bar{x} \forall y, y'(\psi \wedge \psi\{y'/y\} \to y \doteq y')$

Then T_1 is a semantic conservative extension of T_0.

Proof We obtain a Σ_1 extension $\mathcal{M}_1$ of a Σ_0 model $\mathcal{M}_0$ of T_0 by defining

$$f^{\mathcal{M}_1}(\bar{a}) = \begin{cases} b & \text{if } \mathcal{M}_0 \models \psi[\bar{a}, b] \\ \text{arbitrary} & \text{otherwise} \end{cases}$$

Since for any $\bar{a}$ there can be at most one b satisfying $\mathcal{M}_0 \models \psi[\bar{a}, b]$ this is a sound definition. □

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

$$\begin{array}{l} \forall s \forall a \forall j, j'(\\ (0 \leq j \wedge j < seqLen(s) \wedge s[j] \doteq a \wedge \forall k(0 \leq k \wedge k < j \to s[k] \neq a)) \wedge \\ (0 \leq j' \wedge j' < seqLen(s) \wedge s[j'] \doteq a \wedge \forall k(0 \leq k \wedge k < j' \to s[k] \neq a)) \\ \to j' \doteq j) \end{array}$$

This can easily seen to be true. □

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 preleminary observations.

Definition 25 (Expansion). *Let $\Sigma_0 \subseteq \Sigma_1$ be signatures, a Σ_1-structure $\mathcal{M}_1 = (M_1, I_1)$ is called an* expansion *of a Σ_0-structure $\mathcal{M}_0 = (M_0, I_0)$ 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 $\phi \in Fml_{\Sigma_0}$. Furthermore let $\mathcal{M}_0$ be a Σ_0-structure and $\mathcal{M}_1$ an Σ_1-expansion of $\mathcal{M}_0$. Then*

$$\mathcal{M}_0 \models \phi \quad \Leftrightarrow \quad \mathcal{M}_1 \models \phi$$

Proof Obvious. □

This lemma says that the truth or falisity of a sentence ϕ in a given structure only depends on the symbols actually occuring in ϕ. 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 hierachy 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 Σ-structures.*
$\mathcal{M}_0$ is called a substructure *of $\mathcal{M}$ iff*

1. $M_o \subseteq M$
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 $a_1, \ldots, a_n \in M_0$*
$$(a_1, \ldots, a_n) \in I(p) = (a_1, \ldots, a_n) \in I_0(p)$$

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

Proof Easy induction on the complexity of ϕ. □

Definition 27. *Let $\mathcal{M}$ be a Σ-structure.*
The signature Σ_M is obtained from Σ by adding new constant symbols c_a for every element $a \in M$.

The expansion of $\mathcal{M}$ to a Σ_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 Σ-structure. The diagram of $\mathcal{M}$, in symbols $Diag(\mathcal{M})$, is defined by*

$$Diag(\mathcal{M}) = \{\phi \in Fml_{\Sigma_M} \mid \mathcal{M}^* \models \phi \text{ and } \phi \text{ is quantierfree}\}$$

Lemma 22. *Let $\mathcal{M}$ be a Σ-structure.*
If $\mathcal{N} \models Diag(\mathcal{M})$ 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_i sets of sentences in Fml_{Σ_i} 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 Σ_1-structure with $\mathcal{M}_1 \models T_1$ and $\mathcal{M}_0$ its restriction to Σ_0, i.e., $\mathcal{M}_0 = \mathcal{M}_1 \upharpoonright \Sigma_0$. For all $\phi \in T_0$ obviously $T_0 \vdash \phi$. Thus also $T_1 \vdash \phi$ 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 Σ_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 $T_1 \cup Diag(\mathcal{M}_0)$. If this theory were inconistent than already $T_1 \cup F$ for a finite subset $F \subseteq Diag(\mathcal{M}_0)$ would be inconsistent. This is the same as saying $T_1 \vdash \neg F$. Since the constants c_a do not occur in T_1 we get furthermore $T_1 \vdash \forall x_1, \ldots, x_n \neg F'$, where F' is obtained from F be replacing all occurences of constants c_a by the same variable x_i. This is equivalent to $T_1 \vdash \neg\exists x_1, \ldots, x_n F'$. Since T_1 was assume to be a conservative extension of T_0 we also get $T_0 \vdash \neg\exists x_1, \ldots, x_n F'$ and

thus $\mathcal{M}_0 \models \neg\exists x_1, \ldots, x_n F'$. This is a contradiction since by the definition of $Diag(\mathcal{M}_0)$ we have $\mathcal{M}_0 \models \exists x_1, \ldots, x_n F'$ by instantiating the quantified variable x_i that replaces the constant c_a by the element a. This contradiction shows that $T_1 \cup Diag(\mathcal{M}_0)$ is consistent. Let $\mathcal{N}$ be a model of this theory. By Lemma 22 we may assume that $\mathcal{M}_0$ is a substructure of $(\mathcal{N} \restriction \Sigma_0)$. Since by assumption only new relation symbols are added when passing from Σ_0 to Σ_1 also $(\mathcal{N} \restriction \Sigma_1)$ is a substructure of $\mathcal{N}$. By Lemma 21 we get $(\mathcal{N} \restriction \Sigma_1) \models T_1$. Obviously, $(\mathcal{N} \restriction \Sigma_1)$ is an expansion of $(\mathcal{N} \restriction \Sigma_0) = \mathcal{M}_0$ and we are finished. □

A.4 Derived Theorem

The KeY system offers *boot-strapping* verification of the correctness of taclets. On selecting in the main menue `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 diferent 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 occuring 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

$$\forall s \forall i (seqIndexOf(int :: seqGet(s,i)) = i)$$

is not derivable from `seqCore`, but

$$\begin{aligned} &\forall s \forall i\ (seqNPerm(s) \land 0 \leq i \land i < seqLen(s) \\ &\quad \rightarrow seqIndexOf(int :: seqGet(s,i)) = i) \end{aligned}$$

is.

A.5 A Second Set of Extensions by Definition

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

```
seqNPerm(Seq)               seqPerm(Seq,Seq)
Seq seqSwap(Seq,int,int) Seq seqRemove(Seq,int)
seqNPermInv(Seq)
```

Let `seqCore2` be the theory obtained form `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 and6 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 Σ_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 tupel of variables of the same length n as the argument tupel of f and, as before $\phi\{f(\bar{x})/y\}$ denotes the formula arising from ϕ by replacing all free occurrences of y by $f(\bar{x})$.

1. $\forall s(seqNPerm(s) \leftrightarrow \forall\ i(0 \leq i < seqLen(s) \rightarrow$
$\exists j(0 \leq j < seqLen(s) \wedge s[j] \doteq i)))$
2. $\forall s_1, s_2(seqPerm(s_1, s_2) \leftrightarrow seqLen(s_1) \doteq seqLen(s_2) \wedge$
$\exists s(seqNPerm(s) \wedge$
$\forall i(0 \leq i < seqLen(s_1) \rightarrow s_1[i] \doteq s_2[s[i]])))$
3. $\forall s \forall i, j(seqSwap(s, i, j) \doteq seq\_def\{u\}(0, seqLen(s),$
$\textbf{if}\ \neg(0 \leq i \wedge 0 \leq j \wedge i < seqLen(s) \wedge j < seqLen(s))\,))$
$\textbf{then}\ s[u]$
$\textbf{else if}\ u \doteq i$
$\textbf{then}\ s[j]$
$\textbf{else if}\ u \doteq j$
$\textbf{then}\ s[i]$
$\textbf{else}\ s[u]$
4. $\forall s(\forall i(seqRemove(s, i) \doteq \textbf{if}\ (i < 0 \vee seqLen(s) \leq i)$
$\textbf{then}\ s$
$\textbf{else}\ seq\_def\{u\}(0, seqLen(s), \textbf{if}\ u < i$
$\textbf{then}\ s[u]$
$\textbf{else}\ s[u+1]))$
5. $\forall s(seqLen(seqNPermInv(s)) \doteq seqLen(s))$
6. $\forall s \forall i \forall j($
$(0 \leq i \wedge i < seqLen(s) \wedge s[j] \doteq i \wedge 0 \leq j \wedge j < seqLen(s) \wedge seqNPerm(s))$
$\rightarrow seqNPermInv(s)[i] \doteq j)$

Variables s, s_1, s_2 are of type Seq, i, j are of type Int.

Figure 9. Second Set of Extentions by Definition

Proof We show that T_1 is a semantic conservative extension of T_0. Let $\mathcal{M}_0$ be a model of T_0. The structure $\mathcal{M}_1$ coincides with $\mathcal{M}_0$ for all Σ_0-sybols. We define an interpretation of the symbol f as follows

$$f^{\mathcal{M}_1}(\bar{a}) = \begin{cases} b & \text{if } \mathcal{M}_0 \models \exists y(\phi)[\bar{a}] \\ & \text{then pick } b \text{ with } \mathcal{M}_0 \models \phi[\bar{a}, b] \\ \text{arbitrary otherwise} & \end{cases}$$

Technical note, we use $\mathcal{M}_0 \models \phi[\bar{a}, b]$ as a shorthand for $(\mathcal{M}_0, \beta) \models \phi$ with the variable assignment defined by $\beta(x_i) = a_i$ for $0 \leq i < n$ and $\beta(y) = b$.

Obviously, $\mathcal{M}_1 \models \forall \bar{x}(\exists y(\phi) \rightarrow \phi[f(\bar{x})/y])$ $\square$

Lemma 25. `seqCore2` *is a conservative extension of* `seqCore`*, and thus in particular consistent.*

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

$$\begin{array}{ll} \forall s \exists t(\phi) \rightarrow \phi\{seqNPermInv(s)/t\} & \\ \text{with} & \\ \phi \quad = seqLen(t) \doteq seqLen(s) \wedge & \\ \qquad \forall i, j((seqNPerm(s) \wedge s[j] \doteq i \wedge & \\ \qquad 0 \leq i < seqLen(s) \wedge 0 \leq j < seqLen(s)) & \\ \qquad \rightarrow t[i] \doteq j) & \end{array}$$

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 $T_1 \vdash \forall s \phi\{seqNPermInv(s)/t\}$,

i.e.

$$\forall s(seqLen(seqNPermInv(s)) \doteq seqLen(s)) \text{ and}$$
$$\forall s \forall i, j$$
$$((seqNPerm(s) \land s[j] \doteq i \land 0 \leq i < seqLen(s) \land 0 \leq j < seqLen(s))$$
$$\rightarrow seqNPermInv(s)[i] \doteq j)$$

Since T_1 is a conservative extension of `seqCore`, its subtheory `seqCore2` also is. We can infact show that T_1 is equivalent to `seqCore2`. □

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

```
1  \sorts {
2      Seq;
3  }
4
5  \predicates {
6       seqPerm(Seq,Seq);
7       seqNPerm(Seq);
8  }
9
10 \functions {
11     //getters
12     alpha alpha::seqGet(Seq, int);
13     int seqLen(Seq);
14     int seqIndexOf(Seq, any);
15     any seqGetOutside;
16
17     //constructors
18     Seq seqEmpty;
19     Seq seqSingleton(any);
20     Seq seqConcat(Seq, Seq);
21     Seq seqSub(Seq, int, int);
22     Seq seqReverse(Seq);
23     Seq seqDef{false,false,true}(int, int, any);
24
25     Seq seqSwap(Seq,int,int);
26     Seq seqRemove(Seq,int);
27     Seq seqNPermInv(Seq);
28
29
30     // placeholder for values in enhanced for loop
31     Seq values;
32 }
33
34
35 \rules {
36
37 //----------------------------------------------------------
38 //  Core axioms
39 //----------------------------------------------------------
```

```

 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(s) & any::seqGet(s,jsk)=t &
         \forall m;((0<=m&m<jsk) -> any::seqGet(s,m)!=t)==>);
    \add( ==> \exists n;(0 <= n & n < seqLen(s)
                         & any::seqGet(s,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(s),any::seqGet(s,u))) ==>)
     };

  seqOutsideValue {
         \schemaVar \variables Seq s;
         \schemaVar \variables int iv;
       \find(seqGetOutside   )
       \add( \forall s;(\forall iv;((iv < 0 | seqLen(s)<= iv)
               -> any::seqGet(s,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)
   & \forall 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)
```

```

        \heuristics(concrete)
    };


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;
```

```
    \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)))
    };

  seqDef_induction_upper {
        \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-1, t),
            \if(from<to)
               \then(seqSingleton({\subst uSub; (to-1)}t))
               \else(seqEmpty)))
    };

   seqDef_induction_upper_concrete {
        \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)
    };

  seqDef_induction_lower {
        \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(
             \if(from<to)
```

```
                \then(seqSingleton({\subst uSub; (from)}t))
                \else(seqEmpty),
              seqDef{uSub;} (from+1, to, t)))
    };

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))
        \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,{\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
```

```
    \varcond (\notFreeIn(nx, s),   \notFreeIn(nx, x),
              \notFreeIn(nx, from),\notFreeIn(nx, to))
    \replacewith(sub(seqIndexOf(s,x),from));
    \add(==>
      from<=seqIndexOf(s,x) & seqIndexOf(s,x)<to &  <=from &
     \exists nx;((0<=nx&nx<seqLen(s) & any::seqGet(s,nx)=x)))
     };

//--------------------------------------------------------
//
//  Extensions by Definitions
//
//  These taclets extend the signature of corePIX by
//  the relation symbols
//            seqPerm(Seq,Seq), seqNPerm(Seq)
//  and the function symbols
//           Seq seqSwap(Seq,int,int)
//           Seq seqRemove(Seq,int)
//  by direct definitions.
//
//--------------------------------------------------------

 seqNPermDefLeft{
   \schemaVar \term Seq s1;
   \schemaVar \variables int iv,jv;

   \find(seqNPerm(s1) ==> )
   \varcond (\notFreeIn (iv,s1), \notFreeIn (jv,s1))

   \add(
    (\forall iv;(0 <= iv &  iv <seqLen(s1) ->
     \exists jv;(0<=jv & jv<seqLen(s1) &
                  int::seqGet(s1,jv) = iv))) ==> )
     };

 seqNPermDefReplace{
   \schemaVar \term Seq s1;
   \schemaVar \variables int iv,jv;

   \find(seqNPerm(s1))
   \varcond (\notFreeIn (iv,s1), \notFreeIn (jv,s1))

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

seqPermDefLeft{
   \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))
   \add(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))))))
       ==> )
   };

 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)))
```

```
    \replacewith(seqLen(s1))

    \heuristics(simplify)
  };

 getOfNPermInv {
   \schemaVar \term Seq s1;
   \schemaVar \term int i3;
   \schemaVar \skolemTerm int jsk;

   \find(int::seqGet(seqNPermInv(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(any::seqGet(s1,idx)))))

    \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) ==>)
   };

 seqPermRefl{
   \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))
         -> seqNPerm(seqSwap(s1,iv,jv)))) ==>)
     };

  seqNPermComp {
    \schemaVar \term Seq s1,s2;
    \schemaVar \variables int u;

    \assumes(seqNPerm(s2) & seqLen(s1) = seqLen(s2) ==> )
    \find( seqNPerm(s1) ==>)
    \varcond( \notFreeIn(u, s1), \notFreeIn(u, s2)   )
    \add(seqNPerm(seqDef{u;}(0,seqLen(s1),
                    int::seqGet(s1,int::seqGet(s2,u)))) ==>)
    };

  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 termini 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 V instead of $(V, \emptyset)$ and L instead of $(\emptyset, L)$.

The set of objects referenced by R is defined by

$$\begin{aligned} obj^s(R) = &\{{v_i}^s \mid type(v_i) \subseteq Object, 1 \leq i \leq k\} \cup \\ &\{o \mid (o, f) \in R^s\} \cup \{f^s(o) \mid (o, f) \in R^s, type(f) \subseteq Object\} \end{aligned}$$

Note that $obj^s(V, e.f)$ 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 $agree_{rs}(R, s, s')$*
iff
there is a partial isomorphism *π with respect to R from s to s′,*
that is π is a bijective mapping from $obj^s(R)$ onto $obj^{s'}(R)$ satisfying:

1. *π is type preserving,*
   i.e. $o \in T^{\mathcal{D}} \Leftrightarrow \pi(o) \in T^{\mathcal{D}}$ for all $o \in obj^s(R)$ and all types T.
   For objects $o \in obj^s(R)$ of array type $o.\mathbf{length}^s = \pi_0(o).\mathbf{length}^{s'}$ is required in addition.
2. *$s(v) = s'(v)$ for all $v \in V$ with $type(v_i) = Boolean$ or $type(v_i) = Int$;*
3. *$\pi(s(v)) = s'(v)$ for all $v \in V$ with $type(v_i) \sqsubseteq Object$;*
4. *$f^s(o) = f^{s'}(\pi(o))$ for all $(o, f) \in L^s$ where the $type(f) \not\sqsubseteq Object$;*
5. *$\pi(f^s(o)) = f^{s'}(\pi(o)))$ for all $(o, f) \in L^s$ with $type(f) \sqsubseteq Object$;*
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.

```
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 $R = R_\epsilon(\{x.v, y.v\})$. 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 $agree(R, s, s')$ for the end states s, s′*

reached by `m()` *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* π *such that the conditions of Definition 30 are satisfied. This is actually possible by chosing* $\pi(x^s) = y^{s'}$ *and* $\pi(y^s) = x^{s'}$*. Note, that with* $R^* = R_{x,y}(\{x.v, y.v\})$ *there is no problem.*

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

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

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 $select_C^{\mathcal{D}}(\mathbf{heap}^s, o, f^{\mathcal{D}}) = select_C^{\mathcal{D}}(\mathbf{heap}^{s'}, \pi(o), f^{\mathcal{D}})$

We use the notation $agree_{rs}(R, s, s', \pi)$ *to state that* s, s' *agree on* R *via the partial isomorphism* π.

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_{rs}.

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

Lemma 26. *Let* $R = R_{v_1,\ldots,v_k}(L)$ *be an expression of type refSet. Then* $agree_{rs}(R, s, s', id)$ *is true iff*

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

Note, that $obj^s(R) = obj^{s'}(R)$ *is also a consequence of* $agree_{rs}(R, s, s', id)$.

Proof. Easy inspection. □

The following criterion will be essential in the following.

Lemma 27. *Let* $R = R_{v_0,\ldots,v_{k-1}}(L)$ *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*,* $0 \le i < k$: $S[i] = v_i^s = v_i^{s'} = S'[i]$.
2. *For all* j *with* $k \le 2j < n-1$: $S[2j] \in Object$ *and* $S'[2j] \in Object$ *and there is a field* f *such that* $S[2j+1] = f^s(S[2j])$ *and* $S'[2j+1] = f^{s'}(S'[2j])$.
3. *for all objects* o *and all fields* f
   $(o, f) \in L^s \Leftrightarrow S[2j] = o \wedge S[2j+1] = f^s(o)$ *for some* j *with* $k \le 2j < n-1$.
4. *for all objects* o *and all fields* f
   $(o, f) \in L^{s'} \Leftrightarrow S'[2j] = o \wedge S'[2j+1] = f^{s'}(o)$ *for some* j *with* $k \le 2j < n-1$.
5. *For all integers* i *with* $k \le i < n$ *and* $type(S[i]) = type(S'[i])$ *and if* $S[i] \notin Object$ *then* $S[i] = S'[i]$.
6. *For all integers* i*,* j *with* $k \le i < j < n$ *and* $S[i] \in Object$ *and* $S[j] \in Object$:
   $S[i] \doteq S[j] \Leftrightarrow S'[i] \doteq S'[j]$.
7. *For all integers* i *with* $0 \le i < n \quad S[i] = null \Leftrightarrow S'[i] = null$
8. *For all integers* i *with* $0 \le i < n \quad created^s(S[i]) = created^{s'}(S'[i])$

Then $agree(R, s, s')$.

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

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

We first observe that

$$obj^s(R) = obj(S) \tag{9}$$

and

$$obj^{s'}(R) = obj(S') \tag{10}$$

If $o \in obj^s(R)$ then we distinguish three possibilities:

$o = v_i^s$ for some $0 \leq i < k$
By assumption 1 we have $o = S[i]$ and thus $o \in obj(S)$.

$(o, f) \in L^s$ for some f
By assumption 3 there is i with $S[2i] = o$ and thus $o \in obj(S)$.

$o = f^s(o')$ for some $(o', f) \in L^s$
Again by assumption 3 there is i with $S[2i] = o'$ and $S[2i+1] = f^s(o')$. Thus again $o \in obj(S)$.

This establishes $obj^s(R) \subseteq obj(S)$. If $o \in obj(S)$ then there is by definition an index i such that $o = S[i]$. If $0 \leq i < k$ then $S[i] = v_i^s$ and thus $o \in obj^s(R)$. If $k \leq i < n$ then there is j such that $i = 2j$ or $i = 2j + 1$. By assumption 2 there is a field f such that $S[2j + 1] = f^s(S[2j])$ and $S[2j] \in Object$. By assumption 3 this implies $(S[2j], f) \in L^s$ and thus $o \in obj^s(R)$ in any case.
This complete the proof of 9. Claim 10 is proved along the same lines using assumptions 1, 2, 4.

The mapping π is defined for $o = S[i] \in obj(S)$ by $\pi(o) = S'[i]$. Item 6 guarantees that π 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 $(o, f) \in L^s$ with $type(f) \sqsubseteq Object$. 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 π 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.

$\{(\pi(o), f) \mid (o, f) \in L^s\})$
$= \{(\pi(o), f) \mid o = S[2i], f^s(o) = S[2i + 1]$ for some $i, k \leq i < n\})$
by assumption 3
$= \{(o', f) \mid o' = S'[2i], f^{s'}(o) = S'[2i + 1]$ for some $i, k \leq i < n\})$
by definition of π
$= L^{s'}$
by assumption 4

□

The following converse of Lemma 27 is also true.

Lemma 28. *Let $R = R_{v_0,\ldots,v_{k-1}}(L)$ be a reference set expression, s, s' be states, and π a partial isomorphism from $obj^s(R)$ onto $obj^{s'}(R)$ such that $agree(R, s, s', \pi)$ then there are sequences S, S' and $n \in \mathbb{N}$ such that item 1 to 8 of Lemma 27 are satisfied.*

Proof. Let $L^s = \{(o_j, f_j) \mid 0 \leq j < m\}$. We set $n = k + 2m$ define the sequence S by $S[i] = v_i^s$ 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 $type(S[i]) \not\sqsubseteq Object$ and $S'[i] = \pi(S[i])$ otherwise.

It is easily checked that the properties of the partial isomorphism π from Definition 30 imply items 1 to 8 of Lemma 27, as desired. □

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 π^1, π^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* π^1 *there is a partial isomorphism* π^2 in the following definition.

Definition 31 (Information flow using reference sets).
Let α *be a program and* $R^1 = R_{v^1_1,\ldots,v^1_k}(L^1)$, $R^2 = R_{v^2_1,\ldots,v^2_k}(L^2)$ *expressions of type refSet*

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

iff

for all states s'_1, s_2, s'_2 *such that*
α *started in* s_1 *terminates in* s_2 *and*
α *started in* s'_1 *terminates in* s'_2*,*
we have

$$\begin{array}{ll} \text{for any partial isomorphism } \pi^1 & \text{with } agree_{rs}(R^1, s_1, s'_1, \pi^1) \\ \text{there is a partial isomorphism } \pi^2 & \text{with } agree_{rs}(R^2, s_2, s'_2, \pi^2) \\ & \quad\text{and } \pi^2 \text{ extends } \pi^1 \end{array}$$

where π^2 *is said to extend* π^1 *if*
$\pi^2(o) = \pi^1(o)$ *for all* $o \in obj^{s_1}(R_1) \cap obj^{s_2}(R_2)$ *with* $created^{s_1}(o) = tt$.

We extend JavaDL *by a new three-place modal operator* $flow_{rs}(\cdot,\cdot,\cdot))$ *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*

$$s \models flow_{rs}(\alpha, R_1, R_2) \qquad \text{iff} \qquad flow_{rs}(s, \alpha, R_1, R_2) \text{ holds} \;.$$

Let us look at a few simple examples of expressions of type *refSet*.

Example 10.
In the following expressions v is a local variable, e_1, e_2 are expresions of type C_1, C_2, f, f_1, f_2 are fields defined in the class of *this*, C_1 and C_2 respectively. Furthermore, a is an expression of array type, and $i_1 < i_2$ are integers.
$R^1_{ex} = (\{v\}, singleton(this, f))$
$R^2_{ex} = (\{\}, singleton(e_1, f_1) \cup singleton(e_2, f_2)$
$R^3_{ex} = (\{\}, arrayRange(a, i_1, i_2))$

Related observation expressions could be: (To reduce the length of expressions we will write *sC* for *seqConcat* and *sqt* for *seqSingleton*)
$R^1_{ex} = sC(sC(sqt(v), sqt(this)), sqt(this.f))$
or short $\langle v, this, this.f\rangle$
$R^2_{ex} = sC(sC(sC(sqst(e_1), sqst(e_1.f_1)), sqt(e_2)), sqt(e_2.f_2))$
or $\langle e_1, e_1.f_1, e_2, e_2.f_2\rangle$
$R^3_{ex} = seq\_def\{iv\}(i_1, i_2, a[iv])$
or $\langle a[i_1], \ldots a[i_2 - 1]\rangle$

We observe that agree$_{rs}(R^1_{ex}, s_1, s'_1)$ iff agree(R_{1-ex}, s_1, s'_1). For the other two example expressions this is not the case.

Theorem 2. *Let α be a program, and $R_1 = R_{v^1_1,\ldots,v^1_{k_1}}(L_1)$, $R_2 = R_{v^2_1,\ldots,v^2_{k_2}}(L_2)$ arbitrary reference set expressions.*

There is a formula $\phi^{rs}_{\alpha,R_1,R_2}$ in JavaDL *making use of self-composition such that:*

$$s_1 \models \phi^{rs}_{\alpha,R_1,R_2} \quad \Leftrightarrow \quad \mathit{flow}_{rs}(s_1, \alpha, R_1, R_2).$$

Proof. The proof greatly parallels the proof of Theorems 1 and 3. Nevertheless, we will repeat here the whole argument. Thus, this proof is selfcontained, 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 $\phi^{rs}_{\alpha,R_1,R_2}$.

We will explain the construction of $\phi^{rs}_{\alpha,R_1,R_2}$ 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 $select_C(h, e, f)$. 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., `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.

The main challenge in the definition of $\phi^{rs}_{\alpha,R_1,R_2}$ is that we need to express the existence of a partial isomorphims. 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 persue 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 $obj^s(R)$ 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 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 $\phi^{rs}_{\alpha,R_1,R_2}$:

$$\begin{aligned}\phi^{rs}_{\alpha,R_1,R_2} \equiv\ & \forall Heap\ h'_1, h_2, h'_2 \forall T o' \forall T_r r, r' \forall \ldots v', v^2, (v^2)' \ldots \\ & \forall Seq\ S, S' \forall Int\ n \exists Seq\ S_2, S'_2 \exists Int\ n_2 (\ldots (\\ & (Agree_{pre}\ \wedge \langle\alpha\rangle \mathrm{save}\{s_2\} \wedge \mathrm{in}\{s'_1\}\langle\alpha\rangle \mathrm{save}\{s'_2\} \\ & \rightarrow (Agree_{post} \wedge Ext) \ldots)\end{aligned}$$

To maintain readability we have used suggestive abbreviations:

1. $\{\mathrm{in}\ s'_1\}\langle\alpha\rangle$ signals that an update $\{\texttt{heap} := h'_1 \mid\mid \texttt{this} := o' \mid\mid \ldots a_i := v' \ldots\}$ is placed before the modal operator. The a_i cover all relevant parameters and local variables.
2. The construct save$\{s_2\}$ abbreviates a conjunction of equations $h_2 = \texttt{heap}$, $r = \texttt{result}$, $\ldots$, $v^2 = a_i$, $\ldots$.

3. Analogously, save$\{s_2'\}$ stands for the primed version $h_2' = \mathtt{heap}$, $r' = \mathtt{result}$, $\ldots$, $(v^2)' = a_i$, $\ldots$.
4. The shorthand $\{\text{in } s_2\}\{\text{in } s_2'\}E$ in front of a formula is resolved by (a) prefixing every occurence of a heap dependent expression e with the update $\{\mathtt{heap} := h_2\}$ and (b) every primed expression e' with $\{\mathtt{heap} := h_2'\}$.
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 $\phi^{rs}_{\alpha,R_1,R_2}$ is evaluated in state s_1.

In the following we will also use the notation R_i', R_i^2, $(R_i^2)'$ for the terms obtained from R_i by replacing each state dependend 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 $\mathbb{T}$ denotes the set of all types occuring in program α. We assume that $\mathbb{T}$ is finite. We point out that the formulas $Agree_{pre}$ and $Agree_{post}$ 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.

$$
\begin{aligned}
Agree_{pre} \equiv\ & \forall Int\ i(0 \leq i < k_1 \rightarrow (S[i] \doteq v_i \wedge S'[i] \doteq v_i' \wedge S[i] \doteq S'[i]))\wedge \\
& \forall Int\ j(k_2 \leq 2j < (n-1) \rightarrow \\
& \quad instance_{Object}(S_2[2j]) \wedge instance_{Object}(S_2'[2j])\wedge \\
& \quad \exists Field\ f(S_2[2j+1] \doteq select_{Any}(h-2, S_2[2j], f)\wedge \\
& \quad\quad S_2'[2j+1] \doteq select_{Any}(h_2', S_2'[2j], f))) \quad \wedge \\
& \forall Object\ o \forall Field\ f(\in(o,f,L_2) \leftrightarrow \exists Int\ j(k_2 \leq 2j < n-1\wedge \\
& \quad\quad S_2[2j] \doteq o \wedge S_2[2j+1] \doteq select_{Any}(h_2, o, f))) \quad \wedge \\
& \forall Object\ o \forall Field\ f(\in(o,f,L_2') \leftrightarrow \exists Int\ j(k_2 \leq 2j < n-1\wedge \\
& \quad\quad S_2'[2j] \doteq o \wedge S_2'[2j+1] \doteq select_{Any}(h_2', o, f))) \quad \wedge \\
& \forall Int\ i(0 \leq i < n \rightarrow \\
& \quad \textstyle\bigwedge_{T\in\mathbb{T}}(exactInstance_T(S_2[i]) \leftrightarrow exactInstance_T(S_2'[i]))] \quad \wedge \\
& \forall Int\ i(0 \leq i < n \wedge \neg instance_{Object}(S_2[i]) \rightarrow S_2[i] \doteq S_2'[i]) \quad \wedge \\
& \forall Int\ i,j(0 \leq i < j < n \wedge inst_{Obj}(S_2[i]) \wedge inst_{Obj}(S_2[j]) \rightarrow \\
& \quad S_2[i] \doteq S_2[j] \leftrightarrow S_2'[i] \doteq S_2'[j]) \quad \wedge \\
& \forall Int\ i(0 \leq i < n \rightarrow S_2[i] \doteq \mathbf{null} \leftrightarrow S_2'[i] \doteq \mathbf{null}) \quad \wedge \\
& \forall Int\ i(0 \leq i < n \rightarrow created(S_2[i]) \doteq created(S_2'[i]))
\end{aligned}
$$

$Agree_{post}$ is - roughly speaking - the same as $Agree_{pre}$ with S, S' replaced by S_2, S_2':

$$
\begin{aligned}
Agree_{post} \equiv\ & \forall Int\ i(0 \leq i < k_2 \rightarrow (S_2[i] \doteq v_i^2 \wedge S_2'[i] \doteq (v_i^2)' \wedge S_2[i] \doteq S_2'[i]))\wedge \\
& \forall Int\ j(k_1 \leq 2j < (n_2-1) \rightarrow \\
& \quad instance_{Object}(S[2j]) \wedge instance_{Object}(S'[2j])\wedge \\
& \quad \exists Field\ f(S[2j+1] \doteq select_{Any}(\mathbf{heap}, S[2j], f)\wedge \\
& \quad\quad S'[2j+1] \doteq select_{Any}(h_1', S'[2j], f))) \quad \wedge \\
& \forall Object\ o \forall Field\ f(\in(o,f,L_1) \leftrightarrow \exists Int\ j(k_1 \leq 2j < n_2-1\wedge \\
& \quad\quad S[2j] \doteq o \wedge S[2j+1] \doteq select_{Any}(\mathbf{heap}, o, f))) \quad \wedge \\
& \forall Object\ o \forall Field\ f(\in(o,f,L_1') \leftrightarrow \exists Int\ j(k_1 \leq 2j < n_2-1\wedge \\
& \quad\quad S'[2j] \doteq o \wedge S'[2j+1] \doteq select_{Any}(h_1', o, f))) \quad \wedge \\
& \forall Int\ i(0 \leq i < n_2 \rightarrow \\
& \quad \textstyle\bigwedge_{T\in\mathbb{T}}(exactInstance_T(S[i]) \leftrightarrow exactInstance_T(S'[i]))] \quad \wedge \\
& \forall Int\ i(0 \leq i < n_2 \wedge \neg instance_{Object}(S[i]) \rightarrow S[i] \doteq S'[i]) \quad \wedge \\
& \forall Int\ i,j(0 \leq i < j < n_2 \wedge inst_{Obj}(S[i]) \wedge inst_{Obj}(S[j]) \rightarrow \\
& \quad S[i] \doteq S[j] \leftrightarrow S'[i] \doteq S'[j]) \quad \wedge \\
& \forall Int\ i(0 \leq i < n_2 \rightarrow S[i] \doteq \mathbf{null} \leftrightarrow S'[i] \doteq \mathbf{null}) \quad \wedge \\
& \forall Int\ i(0 \leq i < n_2 \rightarrow created(S[i]) \doteq created(S'[i]))
\end{aligned}
$$

$$Ext \equiv \forall Int\ i \forall Int\ j(0 \le i < n \land 0 \le j < n_2 \rightarrow S[i] \doteq S_2[j] \rightarrow S'[i] \doteq S'_2[j])$$

It remains to show that this definition does the job.
The first part proves $s_1 \models \phi_{\alpha,R_1,R_2} \Rightarrow flow^{rs}(s_1, \alpha, R_1, R_2)$.

So let us assume $s_1 \models \phi_{\alpha,R_1,R_2}$. To prove $flow^{rs}(s_1, \alpha, R_1, R_2)$ fix states s'_1, s_2, s'_2 such that α started in s_1 terminates in s_2, α started in s'_1 terminates in s'_2, and agree(R_1, s_1, s'_1, π^1). We need to show that there exists a mapping π^2 such that agree(R^2, s_2, s'_2, π^2) and π^2 extends π^1.

We instantiate the universally quantified variables by their evaluations in state s'_1, s_2, s'_2 respectively, i.e., $\beta(v'_i) = (v^1_i)^{s'_1}$, $\beta(v^2_i) = (v^2_i)^{s_2}$, $\beta((v^2_i)') = (v^2_i)^{s'_2}$, $\beta(h'_1) = \mathbf{heap}^{s'_1}$, etc.

By Lemma 28 and agree(R_1, s_1, s'_1, π^1) there is an instantiation $\beta(n)$ und there are sequences $\beta(S)$, $\beta(S')$ such that $(s_1, \beta) \models Agree_{pre}$.

By definition of β we also have $(s_1, \beta) \models \langle\alpha\rangle\text{save}\{s_2\} \land \text{in}\{s'_1\}\langle\alpha\rangle\text{save}\{s'_2\}$

Thus $s_1 \models \phi_{\alpha,R_1,R_2}$ implies that there are instantiations $\beta(S_2)$ and $\beta(S'_2)$ such that $(s_1, \beta) \models Agree_{post} \land Ext$.

$(s_1, \beta) \models Agree_{post}$ yields by Lemma 27 an isomorphism π^2 such that agree(R^2, s_2, s'_2, π^2). Finally, $(s_1, \beta) \models Ext$ implies that π^2 is an extention of π^1. This, depends on the specific way the isomorphisms are defined.

We now turn to the second part $flow^{rs}(s_1, \alpha, R_1, R_2) \Rightarrow s_1 \models \phi_{\alpha,R_1,R_2}$. Let β be an arbitrary instantiation of the universally quantified variables in the prefix of ϕ_{α,R_1,R_2} and assume $(s_1, \beta) \models Agree_{pre} \land \langle\alpha\rangle\text{save}\{s_2\} \land \text{in}\{s'_1\}\langle\alpha\rangle\text{save}\{s'_2\}$. Otherwise, $(s_1, \beta) \models \phi_{\alpha,R_1,R_2}$ is vacuously true. $(s_1, \beta) \models Agree_{pre}$ and Lemma 27 imply the existence of a partial isomorphism π^1 such that agree(R_1, s_1, s'_1, π^1). Now, $flow^{rs}(s_1, \alpha, R_1, R_2)$ says that there is an isomorphims π^2 extending π^1 such that agree(R_2, s_2, s'_2, π^2). Lemma 28 provides instantiations of the existentially quantified variables $\beta(S_2)$ and $\beta(S'_2)$ such that $(s_1, \beta) \models Agree_{post}$. Since π^2 extends π^1 we also get $(s_1, \beta) \models Ext$. □

B.1 A Simplified Version using Reference Sets

The definition of $\phi^{rs}_{\alpha,R_1,R_2}$ 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 R_2 and provides only a sufficient condition ϕ^*_{α,R_1,R_2} for flow$_{rs}(s_1, \alpha, R_1, R_2)$. But, ϕ^*_{α,R_1,R_2} does not involve quantification over location sets.

Lemma 29. *Let R be a reference set expression.*
If $agree_{rs}(R, s, s', \pi)$ and ρ is an automorphism on $\mathcal{D}$
then also $agree_{rs}(R, s, \rho(s'), \rho \circ \pi)$.

Proof. From the assumption agree$_{rs}(R, s, s', \pi)$ we get by Definition 30 that π is a bijective mapping from $obj^s(R)$ onto $obj^{s'}(R)$ satisfying:

1. π is type preserving,
   i.e. $o \in T^{\mathcal{D}} \Leftrightarrow \pi(o) \in T^{\mathcal{D}}$ for all $o \in obj^s(R)$ and all types T.
   For objects $o \in obj^s(R)$ of array type $o.\mathbf{length}^s = \pi_0(o).\mathbf{length}^{s'}$ is required in addition.
2. $s(v) = s'(v)$ for all $v \in V$ with $type(v_i) = Boolean$ or $type(v_i) = Int$;
3. $\pi(s(v)) = s'(v)$ for all $v \in V$ with $type(v_i) \sqsubseteq Object$;
4. $f^s(o) = f^{s'}(\pi(o))$ for all $(o, f) \in L^s$ where the $type(f) \not\sqsubseteq Object$;
5. $\pi(f^s(o)) = f^{s'}(\pi(o)))$ for all $(o, f) \in L^s$ with $type(f) \sqsubseteq Object$;

6. $\pi(L^s) = \{(\pi(o), f) \mid (o, f) \in L^s\} = L^{s'}$.
7. If $null \in obj^s(R)$ then $\pi(null) = null$
8. For all $o \in obj^s(R)$: $\quad created^s(o) = tt \Leftrightarrow created^{s'}(\pi(o)) = tt$.

Then

1. $\rho \circ \pi$ is type preserving since ρ is an isomorphism.
2. $s(v) = \rho(s'(v)) = \rho(s')(v)$ for all $v \in V$ with $type(v_i) = Boolean$ or $type(v_i) = Int$, since ρ 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 $type(v_i) \sqsubseteq Object$ by tghe laws of equality.
4. $f^s(o) = \rho(f^s(o)) = f^{\rho(s')}(\rho \circ \pi(o))$ for all $(o, f) \in L^s$ where the $type(f) \not\sqsubseteq Object$ since ρ is the identity on basic data types and Lemma 1.
5. $\rho \circ \pi(f^s(o)) = f^{\rho(s')}(\rho \circ \pi(o)))$ for all $(o, f) \in L^s$ with $type(f) \sqsubseteq Object$ by tzhe 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 ρ is defined in type *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 $null \in obj^s(R)$ then $\rho \circ \pi(null) = null$, since $\rho(null) = null$ for any isomorphism
8. For all $o \in obj^s(R)$: $\quad created^s(o) = tt \Leftrightarrow created^{\rho(s')}(\rho \circ \pi(o)) = tt$ 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 α be a program and $R^1 = R_{v_1^1,\ldots,v_k^1}(L^1)$, $R^2 = R_{v_1^2,\ldots,v_k^2}(L^2)$ expressions of type refSet

Program α allows simple information flow only from R^1 to R^2 when started in s_1, denoted by $flow^*_{rs}(s_1, \alpha, R^1, R^2)$
iff
for all states s'_1, s_2, s'_2 such that
α started in s_1 terminates in s_2 and
α started in s'_1 terminates in s'_2,
we have

if $agree_{rs}(R^1, s_1, s'_1, id)$
then there is a partial isomorphism π^2 with $agree_{rs}(R^2, s_2, s'_2, \pi^2)$
and π^2 extends id

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

Lemma 30. *For all programs α, any two reference expressions R_1 and R_2 , and any state s_1*

$$flow^*_{rs}(s_1, \alpha, R_1, R_2) \quad \Rightarrow \quad flow_{rs}(s_1, \alpha, R_1, R_2)$$

Since the reverse implication is obviously true Lemma 30 entails that flow_{rs} and 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 π^2.

To prove $\text{flow}_{rs}(s_1, \alpha, R_1, R_2)$ we fix, in addition to s_1, states s'_1, s_2, s'_2 such that α started in s_1 terminates in s_2 and α started in s'_1 terminates in s'_2, and assume $\text{agree}_{rs}(R_1, s_1, s'_1, \pi^1)$. We need to show that there exists π^2 with $\text{agree}_{rs}(R_2, s_2, s'_2, \pi^2)$ and π^2 extends π^1.

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

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

Again using Lemma 29, this time for the isomorphism ρ', we obtain from $\mathrm{agree}_{rs}(R_2, s_2, s_3', \pi^3)$ also $\mathrm{agree}_{rs}(R_2, s_2, \rho'(s_3'), \rho' \circ \pi^3)$. Since α is a deterministic program and we have already defined s_2' to be the final state of α when started in s_2 we get $s_2' = \rho'(s_3')$ and thus $\mathrm{agree}_{rs}(R_2, s_2, s_2', \rho' \circ \pi^3)$.

It remains to convince ourselves that $\rho' \circ \pi^3$ extends π^1, i.e., for every $o \in obj^{s_1}(R_1) \cap obj^{s_2}(R_2)$ with $created^{s_1}(o) = tt$ we need to show $\rho' \circ \pi^3(o) = \pi^1(o)$. By the definition of isomorphic states we obtain from $created^{s_1}(o) = tt$ also $created^{\rho(s_1)}(o) = tt$. Thus we can infer $\rho'(o) = \rho^{-1}(o)$ and by choice of ρ further $\rho^{-1}(o) = \pi^1(o)$, as desired. □

Definition 33. *The set CLE of* constructive *location set expressions is a subset of all expressions of type LocSet definied 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 $type(f), type(e) \neq$ 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 $infiniteUnion\{iv\}(e) \doteq infiniteUnion\{iv\}(\mathit{if}\ iv < t\ \mathit{then}\ e\ \mathit{else}\ \emptyset)$ 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 Int such that for all states s, objects o and fields f*

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

Here $sq_e[i]$ abbreviates $seqGet_{Any}(sq_e, i)$.

Proof. We set $t_\emptyset = 0$, while $sq_\emptyset$ is arbitrary, e.g., $sq_\emptyset = \mathbf{null}$. The claimed correspondence between $\emptyset$ and $sq_\emptyset$ and $t_\emptyset$ is trivially satisfied.

Also for $sq_{singleton(e_0,f)} = \langle e_0, e_0.f \rangle$ and $t_{singleton(e_0,f)} = 2$ 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 $e = arrayRange(a, b_1, b_2)$ we set $t_e = 2 * (b_2 - b_1)$ and $sq_e = seq\_def\{iv\}(0, t_e, \mathit{if}\ even(iv)\ \mathit{then}\ a\ \mathit{else}\ a[b_0 + (iv/2)])$. 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 $sq_{e_1 \cup e_2} = seq\_def\{iv\}(0, t_e, \mathit{if}\ iv < t_{e_1}\ \mathit{then}\ sq_{e_1}[iv]\ \mathit{else}\ sq_{e_1}[t_{e_1} + iv])$.

The last case in the inductive definition is $e = infiniteUnion\{iv < t\}(e_0)$. 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 assigments of iv the expressions sq_{e_0} will evaluate to sequences of differing length. Let $t_m = max_{0 \leq iv < t} t_{e_0}$ and
$sq = if\ 0 \leq iv < t_{e_0}\ then\ sq_{e_0}\ else\ if\ even(iv)\ then\ sq_{e_0}[0]\ else\ sq_{e_0}[1]$. Thus if in state s with variable assignment β 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^s. Since iv does no longer occur free in t_m the evaluation t_m^s 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 $t_e = t * t_m$ and $sq_e = seq\_def\{iv\}(0, t_e, sq(iv/t_m/iv)[mod(t_m, iv)])$ Here $sq(x/iv)$ is the term arising from sq by replacing every occurence 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. □

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:

$obj^s(v) = \{v_i{}^s \mid type(v_i) \subseteq Object, 1 \leq i \leq k\} \cup$
$obj^s(L) = \{o \mid (o, f) \in R^s\} \cup \{f^s(o) \mid (o, f) \in R^s, type(f) \subseteq Object\}$
Thus
$obj^s(R) = obj^s(v) \cup obj^s(L)$

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 π defined by $\pi(sq_e^s[i]) = sq_e^{s'}[i]$ for $0 \leq i < t_e^s$ is bijective then it as a partial isomorphism from $obj^s(L_2)$ onto $obj^{s'}(L_2)$.

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

Theorem 3. *Let α be a program, and let $R_1 = R_{v_1^1,\ldots,v_{k_1}^1}(L_1)$ be an arbitrary reference set expression and $R_2 = R_{v_1^2,\ldots,v_{k_2}^2}(L_2)$ a reference set expression with L_2 in CLE.*

There is a formula $\phi^{rs}_{\alpha,R_1,R_2}$ in JavaDL *making use of self-composition such that:*

$$s_1 \models \phi^{rs}_{\alpha,R_1,R_2} \quad \Rightarrow \quad \mathit{flow}_{rs}(s_1, \alpha, R_1, R_2).$$

Proof. By Lemma 30 it suffices to find $\phi^{rs}_{\alpha,R_1,R_2}$ such that

$$s_1 \models \phi^{rs}_{\alpha,R_1,R_2} \quad \Rightarrow \quad \mathrm{flow}^*_{rs}(s_1, \alpha, R_1, R_2).$$

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

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

We will explain the construction of $\phi^{rs}_{\alpha,R_1,R_2}$ 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 $select_C(h, e, f)$. 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., `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 $\phi^{rs}_{\alpha,R_1,R_2}$:

$$\begin{aligned}\phi^{rs}_{\alpha,R_1,R_2} \equiv\ & \forall Heap\ h'_1, h_2, h'_2 \forall T o' \forall T_r r, r' \forall \ldots v', v^2, (v^2)' \ldots \\ & (Agree_{pre}\ \wedge \langle\alpha\rangle \mathrm{save}\{s_2\} \wedge \mathrm{in}\{s'_1\}\langle\alpha\rangle\mathrm{save}\{s'_2\} \\ & \to (Agree_{post} \wedge Ext))\end{aligned}$$

To maintain readability we have used suggestive abbreviations:

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

In the following we will also use the notation R'_i, R^2_i, $(R^2_i)'$ for the terms obtained from R_i by replacing each state dependend 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:

$$\begin{aligned}Agree_{pre} \equiv\ & \textstyle\bigwedge_{1\le i\le k_1} v^1_i \doteq (v^1_i)' \quad \wedge \\ & \forall o \forall f((o,f) \in L_1 \to select_{Any}(h_i, o, f) \doteq select_{Any}(h'_i, o, f)) \wedge \\ & \forall o \forall f((o,f) \in L_1 \leftrightarrow (o,f) \in L'_1\end{aligned}$$

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

$$\begin{aligned}Agree_{post} \equiv\ & t_2 \doteq t'_2 \\ & \textstyle\bigwedge_{1\le i\le k_2, type(v^2_i) \not\sqsubseteq Object} (v^2_i \doteq (v^2_i)') \wedge \\ & \textstyle\bigwedge_{1\le i<j\le k_2, type(v^2_i), type(v^2_j) \sqsubseteq Object} ((v^2_i \doteq v^2_j) \leftrightarrow ((v^2_i)' \doteq (v^2_j))') \wedge \\ & \forall i, j (0 \le i < j < t_2 \to (sq_2[i] \doteq sq_2[j] \leftrightarrow sq'_2[i] \doteq sq'_2[j])) \wedge \\ & \textstyle\bigwedge_{1\le i\le k_2, type(v^2_i) \sqsubseteq Object} \forall j (0 \le j < t_2 \to \\ & \quad (v^2_i \doteq sq_2[j] \leftrightarrow (v^2_i)' \doteq (sq_2[j])'))\end{aligned}$$

$$Ext \equiv \textstyle\bigwedge_{1\le i\le k_2} \forall j (0 \le j < t_2 \wedge v^2_i \doteq sq_2[j] \to (v^2_i)' \doteq (sq_2[j])')$$

It remains to show that this definition does the job.
So let us assume $s_1 \models \phi_{\alpha,R_1,R_2}$. To prove $flow^*(s_1,\alpha,R_1,R_2)$ fix states s_1', s_2, s_2' such that α started in s_1 terminates in s_2, α 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 π^2 such that agree(R^2, s_2, s_2', π^2) and π^2 extends id.

We instantiate the universally quantified variables by their evaluations in state s_1', s_2, s_2' respectively, i.e., $\beta(v_i') = (v_i^1)^{s_1'}$, $\beta(v_i^2) = (v_i^2)^{s_2}$, $\beta((v_i^2)') = (v_i^2)^{s_2'}$, $\beta(h_1') = \mathbf{heap}^{s_1'}$, etc.

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

By definition of β we also have $(s_1,\beta) \models \langle\alpha\rangle\text{save}\{s_2\} \wedge \text{in}\{s_1'\}\langle\alpha\rangle\text{save}\{s_2'\}$

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

We define π^2 by $\pi^2((sq_2[j])^{s_2}) = (sq_2[j])^{s_2'}$ for $0 \leq j < t_2^{s_2}$. Now, $(s_1,\beta) \models Agree_{post}$ (due to line 1 and line 4 in the definition of $Agree_{post}$) implies that π^2 thus definied is a bijection. Lemma 32 says that π^2 is a partial isomorphism from $obj^s(L_2)$ onto $obj^{s'}(L_2)$. We extend π^2 to a mapping from $obj^s(R_2)$ onto $obj^{s'}(R_2)$ by $\pi^2((v_i^2)^{s_2}) = v_i^2)^{s_2'}$. Line 2 and 3 in the definition of $Agree_{post}$ guarantee that this is a bijection and line 5 makes sure that this definition is compatible with π^2 defined on $obj^s(L_2)$. Altogether, we see that π^2 is a partial isomorphism from $obj^s(R_2)$ onto $obj^{s'}(R_2)$ and agree(R^2, s_2, s_2', π^2) is true.

Finally, $(s_1,\beta) \models Ext$ implies that π^2 is an extention of the identity. We may thus conclude $flow^*(s_1,\alpha,R_1,R_2)$ as desired. □

References

1. T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In J. G. Morrisett and S. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006*, pages 91–102. ACM, 2006.
2. T. Amtoft and A. Banerjee. Information flow analysis in logical form. In R. Giacobazzi, editor, *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*, LNCS 3148, pages 100–115. Springer, 2004.
3. A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *J. Funct. Program.*, 15(2):131–177, 2005.
4. G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In M. Butler and W. Schulte, editors, *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, LNCS 6664, pages 200–214. Springer, 2011.
5. G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*. IEEE Computer Society, 2004.
6. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer, 2007.
7. Á. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Proceedings, Security in Pervasive Computing*, LNCS 3450. Springer, 2005.
8. H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Einführung in die mathematische Logik (5. Aufl.).* Spektrum Akademischer Verlag, 2007.
9. R. Hähnle, J. Pan, P. Rümmer, and D. Walter. Integration of a security type system into a program logic. In U. Montanari, D. Sanella, and R. Bruni, editors, *Proc. Trustworthy Global Computing, Lucca, Italy*, LNCS 4661. Springer, 2007.
10. C. Hammer, J. Krinke, and G. Snelting. Information flow control for Java based on path conditions in dependence graphs. In *IEEE International Symposium on Secure Software Engineering (ISSSE 2006)*, pages 87–96. IEEE, March 2006.
11. R. R. Hansen and C. W. Probst. Non-interference and erasure policies for Java Card bytecode. In *6th International Workshop on Issues in the Theory of Security (WITS '06)*, 2006.

12. S. Hunt and D. Sands. On flow-sensitive security types. In J. G. Morrisett and S. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006*, pages 79–90. ACM, 2006.
13. R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Sci. Comput. Program.*, 37(1-3):113–138, 2000.
14. T. Lindholm and F. Yellin. *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.
15. J. Monk. *Mathematical Logic*, volume 37 of *Graduate Texts in Mathematics.* Springer, 1976.
16. A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, pages 228–241, 1999.
17. F. Ruch. Efficient logic-based information flow analysis of object-oriented programs. Bachelor thesis, Karlsruhe Institute of Technology, 2013.
18. C. Scheben and P. H. Schmitt. Verification of information flow properties of Java programs without approximations. In *Formal Verification of Object-Oriented Software International Conference, FoVeOOS 2011, Revised Selected Papers*, LNCS. Springer, 2012. To appear. Earlier version in Technical Report 2011-26, KIT, Department of Informatics. Available at http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/1977984.
19. J. R. Shoenfield. *Mathematical Logic.* Addison–Wesley Publ. Comp., Reading, Massachusetts, 1967.
20. M. Strecker. Formal analysis of an information flow type system for MicroJava (extended version). Technical report, Technische Universität München, July 2003.
21. D. M. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *10th Computer Security Foundations Workshop (CSFW '97), June 10-12, 1997, Rockport, Massachusetts, USA*, pages 156–169, 1997.
22. B. Weiß. *Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction.* PhD thesis, Karlsruhe Institute of Technology, 2011.