Abstraction, Objects and Information Flow Analysis

A twofold thesis on loop abstraction with objects and value-sensitive information flow analysis in a dynamic logic

Master Thesis Computer Science,
Research Track, Kerckhoffs Institute
May 2011

Institute for Computing and Information Sciences,
Digital Security group,
Radboud University, Nijmegen,
The Netherlands

Computer Science and Engineering Department,
Software Engineering using Formal Methods group,
Chalmers University of Technology, Göteborg,
Sweden

Author: Bart van Delft
Thesis number: 650
Supervisor: Richard Bubel
Examiners: Reiner Hähnle, Erik Poll, Wojciech Mostowski

Summary

This thesis continues the research on abstract interpretation and information flow analysis from Bubel et al. (2009). The logical framework used for the formal verification of program source code is extended to allow for programs containing objects. The main challenge addressed with this extension is the automatic derivation of loop invariants. The resulting method based on abstraction is sound but also in most applications outperformed by the for objects more common shape analysis techniques. Shape analysis on the other hand is not easily adapted to fit into the existing approach. The possibility of combining shape analysis with the abstraction-based approach used in this thesis requires further investigation.

In the second part of this thesis we improve the secure information flow analysis that is based on the same logical framework. We obtain a higher level of value and control-flow sensitivity by various adaptations to the framework's signature, syntax and semantics. The combination of this extended information flow analysis with the object-extended dynamic logic is a topic that we want to investigate further in future research.

Acknowledgments

Over the course of the last months in which I have been working on my thesis I have received a lot of help from a lot of people. My friends, Dutch, Swedish and from all other nationalities that I met in Sweden: you helped making my stay as an exchange student even more rewarding. I want to thank you all very much for that, Sara in particular.

I also want to thank Benjamin Niedermann, who next to being a good friend has improved this thesis by numerous discussions and by proof-reading earlier versions.

I am very grateful to Wolfgang Ahrendt, Erik Poll, Reiner Hähnle, and my parents who have all done a great effort in making my stay in Göteborg and therefore this thesis possible.

Thanks to my examiners, Erik Poll, Reiner Hähnle and Wojciech Mostowski, for a critical reading of my thesis and pointing out flaws such as one this.

Finally I want to thank my supervisor Richard Bubel, who has kept me motivated, increased my understanding of dynamic logic, contributed ideas of which one is at the core of the value-sensitive information flow analysis and who has, in short, been a great supervisor.

Contents

Acknowledgments **5**

1. Introduction **9**

1.1. Theorem Proving . . . 9
1.2. The Logical System . . . 10
1.2.1. First-order Logic . . . 10
1.2.2. Dynamic Logic . . . 12
1.2.3. Sequent Calculus . . . 15
1.2.4. The KeY Approach . . . 17
1.3. Abstract Interpretation . . . 17
1.3.1. Fixed Points . . . 18
1.4. Information Flow Analysis . . . 18
1.4.1. Declassification . . . 20
1.4.2. Side Channels . . . 20
1.4.3. Information Flow Analysis in Dynamic Logic . . . 21
1.5. Outline . . . 21

2. Abstract Interpretation **23**

2.1. A Dynamic Logic with Objects . . . 23
2.1.1. Signature and Syntax . . . 23
2.1.2. Interpretation and Semantics . . . 26
2.1.3. Calculus . . . 29
2.2. Automatic Derived Invariants by Abstraction . . . 31
2.2.1. Abstract Domains . . . 31
2.2.2. Update Weakening . . . 35
2.2.3. Automated Fixed Point Search . . . 38
2.2.4. Replacing the Special Quantifier $\tilde{\exists}$ . . . 43
2.3. Examples and Improvements . . . 48
2.3.1. List Insertion . . . 49
2.3.2. List Reversal . . . 51
2.3.3. Field Invariants . . . 52
2.3.4. Improvements to the Fixed Point Search . . . 52
2.3.5. Improvements to the Abstract Lattice . . . 56
2.4. Comparison with Shape Analysis . . . 56

3. Information Flow Analysis **59**

3.1. Dependencies . . . 60
3.2. Including Dependencies in the Logic . . . 61
3.2.1. Extensions to Signature, Syntax and Semantics . . . 61
3.2.2. Extensions to Calculus Rules . . . 63
3.2.3. Limitations . . . 66

3.3. Improving Value and Control-Flow Sensitivity 67
3.3.1. Extensions to Signature, Syntax and Semantics 67
3.3.2. Value Sensitivity 75
3.3.3. Control Flow Sensitivity 77
3.4. Other Approaches 81
3.4.1. Dependency Tracking with Labels 81
3.4.2. Separation of Implicit and Explicit Dependencies 83

4. Conclusions and Future Research **85**

Bibliography **87**

A. Rewrite Rules **91**
A.1. Update Rewriting Rules 91
A.2. Update Rewriting Rules with Objects 91

B. Proofs **93**
B.1. Lemma: Soundness of weakenUpdate 93
B.2. Lemma: Soundness of invariantUpdate 95

1. Introduction

1.1. Theorem Proving

The development of computer software is a process consisting of several stages that can be grouped into design, implementation, testing, documenting, deployment and maintenance. These stages are often not executed entirely sequential but may be run in parallel. Partial implementations may for example be tested before finishing the entire product, and documentation can be added to the source code before the product is tested. The step of interest in this thesis is the *testing* phase, where it is verified that the implemented program does indeed match the specifications and requirements resulted from the design phase. There are several approaches to test implementations, such as manual code reviews, test case generation and running benchmark tools. In this thesis we focus on the method of *formal verification* of the implementation.

In formal verification it is attempted to give a formal (or, if one prefers that term: mathematical) proof that the provided program satisfies the requirements specified by the designers. This verification is performed on the source code of the program, which is therefore referred to as being a *static* analysis of the program. Would a different method of testing be adopted that verifies by running instances of the program, such as in the case of benchmarking or running test cases, this is referred to as a *dynamic* way of program analysis.

There are different approaches to formally verify a program, but they all share the property that a *formal language* needs to be selected in which the *formal requirements* need to be defined, that is, the specific requirements on which you want to verify the program. The usage of the formal language requires the use of solely well-defined expressions, implying that every part of the expression is formally defined in that language as well. For example, the property “This method always returns the sum of the integers in the array provided” requires you to also specify “method”, “return”, “sum”, “integers” and “array” in the formal language, which in addition require other elements to be specified. This goes on recursively until the whole (programming) language used for the program has been formally defined. Fortunately most of this formalization can be reused if you have a second program to be proven in the same language, so we have to go to the process of defining integers, objects and other elements only once. Except of course if you want to use a different formal language to prove the requirements of the other program.

Several attempts have been made on describing specific domains in specific formal languages. Some of these formalizations are provided in the form of *theorem provers* that are intended to give the software tester the benefit of proving the correctness of the program in an environment in which the lion’s share has already been formalized. In most cases it is even possible to load the source code of the program to be verified into the theorem prover such that a formal notation of the program is automatically created.

Two stages of the verification process remain. The first one is the formalization of the statement to be proved (to which we refer as the *proof obligation*) which might not have to be performed in the formal language but an intermediate language more convenient to the programmer or verifier (e.g. JML for Java programs). This statement in the intermediate language is then translated by the theorem prover to a formal notation. The last stage is

generating the actual *proof* that this formalized property does (not) hold.

In this last stage we make a last distinction between theorem provers: those that try to deduce the validity of the proof obligation automatically, called *automatic* theorem provers, and those that require the human user to guide or entirely perform the proof deduction, hence called *interactive* theorem provers. In general it may be said that interactive theorem proving is more labor-intensive for the user, however enables the user to prove or disprove properties on which an automatic theorem prover would fail.

It could be concluded that the formal verification of a program is in general a labor-intensive process. Even given the presence of a theorem prover, the user still needs to (formally) define a correct proof obligation and possibly interact with the theorem prover, implying that the user has to understand the proof deduction and underlying mechanisms used by the theorem prover. One might ask when this method could be preferred over other methods of software testing.

The main difference between formal verification and other testing methods, such as running test cases, is that at the end of the process you have obtained a formal *proof* that the formalized requirement indeed holds for the program. Most verification systems are designed with this goal in mind, which means that a proof for a requirement gives you absolute certainty that the requirement is fulfilled. This opposed to running arbitrary test cases, where the fulfillment of the requirement is unknown for untested cases.

The usage of the terms source code, program and markup languages may give the impression that theorem provers operate directly on the actual source code of the program. This holds for the setting in which we are interested in this thesis, however the process of formalizing a domain and properties to be proved also applies on other settings. Several theorem provers exist that only consist of assuring the correct usage of the formal language and have some built-in libraries for e.g. integer and boolean support, but leave it to the user to specify their own programming language or calculus such as Coq (Castéran and Bertot, 2004), Isabelle/HOL (Nipkow et al., 2002) and PVS (Owre et al., 1992). Other provers are aimed at their own formal language for proving domain-specific properties, such as ProVerif (Blanchet, 2001), a tool for protocol analysis which is in fact testing on the design instead of on the implementation.

In this chapter we introduce the *dynamic logic* that is used as the formal language in this thesis. The same language is used by the verification tool KeY (Beckert et al., 2007), which is further described in Section 1.2.4. We also introduce the concepts of abstract interpretation and fixed points in Section 1.3, as well as information flow analysis in Section 1.4, which are the two main themes discussed in this thesis. An outline of the whole thesis is presented in Section 1.5.

1.2. The Logical System

In this section a dynamic logic is introduced that the remainder of this thesis extends to incorporate objects and value-sensitive information flow analysis (Sections 1.3 and 1.4). Notation and definitions are adopted from the original work by Bubel et al. (2009).

1.2.1. First-order Logic

The dynamic logic used in this thesis can be seen as an extension to first-order logic. We give a short description here to introduce notation and such. A detailed introduction to first-order logic can be found in (Fitting, 1990). We start by defining a signature that lists all the symbols used in the logic.

Definition 1 (First-order signature)**.** *A* first-order signature *is a tuple* $\Sigma = (\mathcal{F}, \mathcal{P}, \mathcal{V})$ *where* $\mathcal{F}$ *is a set of* function symbols*,* $\mathcal{P}$ *a set of* predicate symbols *and* $\mathcal{V}$ *a set of* logic variables*.*

In the rest of this section we assume a fixed signature, so we omit it as a parameter to all subsequent definitions. We use this assumption through-out this thesis, that is, we have the similar assumption on signatures defined in later sections. Note that we do not define constants as such, but rather consider them to be 0-ary function symbols.

The following syntax is the basic one used in our first-order logic and forms the base of the syntax for the dynamic logic defined in Section 1.2.2 and the extensions contributed by this thesis in Chapters 2 and 3. The syntax provides us with constructors to create terms (t) and formulas (φ).

Definition 2 (First-order syntax)**.** *The way in which the elements from the signature can be combined, is defined with the following syntax, in which* $f \in \mathcal{F}$*,* $p \in \mathcal{P}$ *and* $y \in \mathcal{V}$*.*

$$\begin{aligned} t &::= f(t, \ldots, t) \mid y \mid \mathit{if}(\varphi)\mathit{then}(t)\mathit{else}(t) \\ \varphi &::= \text{true} \mid \text{false} \mid p(t, \ldots t) \mid \varphi \;\&\; \varphi \mid (\varphi \mid \varphi) \mid \varphi \;\text{->}\; \varphi \mid\; !\,\varphi \mid \forall y.\varphi \mid \exists y.\varphi \mid t \doteq t \end{aligned}$$

The terms of the form $f(t_1, \ldots, t_n)$ *and the formulas of the form* $p(t_1, \ldots, t_n)$ *must respect the arities of the respective symbols* f *and* p*.*

To provide the semantics of terms and formulas we require an interpretation I of function and predicate symbols and a logic variable assignment β mapping logic variables to concrete values.

Definition 3 (Interpretation and Logic Variable Assignment)**.** *Let* D *be the universe of concrete values. The* interpretation I *is a function mapping a function symbol* $f \in \mathcal{F}$ *with arity* n *to a function:* $I(f) : \mathsf{D}^n \rightarrow \mathsf{D}$ *and every predicate symbol* $p \in \mathcal{P}$ *with arity* n *to a relation* $I(p) \subseteq \mathsf{D}^n$*. The pair* (D, I) *is called a* first-order structure*.*

A logic variable assignment β *is a function mapping logic variables to concrete values:* $\beta : \mathcal{V} \rightarrow \mathsf{D}$*.*

We can now assign a meaning to these terms and propositions by means of the evaluation function val.

Definition 4 (First-order semantics)**.** *Given a universe* D*, an interpretation* I *and a logic variable assignment* β*, the function* $val_{M,\beta}$ *evaluates a term* t *to a concrete value* $val_{M,\beta}(t) \in \mathsf{D}$ *and a formula* φ *to a truth value* $val_{M,\beta}(\varphi) \in \{tt, \mathit{ff}\}$*. A formula* φ *is called* valid *if and only if* $val_{M,\beta}(\varphi) = tt$ *for all first-order structures* $M = (\mathsf{D}, I)$ *and logic variable assignments* β*. We define this evaluation function as follows.*

$$\begin{aligned} val_{M,\beta}(f(t_1, \ldots, t_n)) &= I(f)(val_{M,\beta}(t_1), \ldots, val_{M,\beta}(t_n)) \\ val_{M,\beta}(y) &= \beta(y) \\ val_{M,\beta}(\mathit{if}(\varphi)\mathit{then}(t_1)\mathit{else}(t_2)) &= \begin{cases} val_{M,\beta}(t_1) & \text{if } val_{M,\beta}(\varphi) = tt \\ val_{M,\beta}(t_2) & \text{otherwise} \end{cases} \\ val_{M,\beta}(\text{true}) &= tt \\ val_{M,\beta}(\text{false}) &= \mathit{ff} \\ val_{M,\beta}(p(t_1, \ldots, t_n)) &= tt \quad \text{iff} \quad (val_{M,\beta}(t_1), \ldots, val_{M,\beta}(t_n)) \in I(p) \end{aligned}$$

$$
\begin{aligned}
val_{M,\beta}(\varphi_1 \ \&\ \varphi_2) &= tt \quad \text{iff} \quad \mathit{ff} \notin \{val_{M,\beta}(\varphi_1), val_{M,\beta}(\varphi_2)\} \\
val_{M,\beta}(\varphi_1 \mid \varphi_2) &= tt \quad \text{iff} \quad tt \in \{val_{M,\beta}(\varphi_1), val_{M,\beta}(\varphi_2)\} \\
val_{M,\beta}(\varphi_1 \mathrel{-\!\!>} \varphi_2) &= val_{M,\beta}(!\,\varphi_1 \mid \varphi_2) \\
val_{M,\beta}(!\,\varphi) &= tt \quad \text{iff} \quad val_{M,\beta}(\varphi) = \mathit{ff} \\
val_{M,\beta}(\forall y.\varphi) &= tt \quad \text{iff} \quad \mathit{ff} \notin \{val_{M,\beta_y^v}(\varphi) \mid v \in \mathsf{D}\} \\
val_{M,\beta}(\exists y.\varphi) &= tt \quad \text{iff} \quad tt \in \{val_{M,\beta_y^v}(\varphi) \mid v \in \mathsf{D}\} \\
val_{M,\beta}(t_1 \doteq t_2) &= tt \quad \text{iff} \quad val_{M,\beta}(t_1) = val_{M,\beta}(t_2)
\end{aligned}
$$

When there exists a first-order structure $M = (\mathsf{D}, I)$ and a logic variable assignment β such that $val_{M,\beta}(\varphi) = tt$, we say that φ is *satisfiable*.

The notation β_y^v in the evaluation of formulas with a $\forall$ or $\exists$ quantifier means that we evaluate the formula φ with the same logic variable assignment function β as the original formula, except that the logic variable y is mapped to the value v.

EXAMPLE 1:

As an example, consider the evaluation of the formula (true $\mathrel{-\!\!>}$ $c - c \doteq 0$) - where $c, -, 0 \in \mathcal{F}$ (c is a constant). We use the standard integer domain for D and the standard interpretation I for $-$ (minus):

$$
\begin{aligned}
val_{M,\beta}(\text{true} \mathrel{-\!\!>} c - c \doteq 0) &= val_{M,\beta}((!\,\text{true}) \mid (c - c \doteq 0)) \\
&= tt \text{ iff } tt \in \{val_{M,\beta}(!\,\text{true}), val_{M,\beta}(c - c \doteq 0)) \\
&= tt \text{ iff } tt \in \{tt \text{ iff } val_{M,\beta}(\text{true}) = \mathit{ff}, val_{M,\beta}(c - c \doteq 0)\} \\
&= tt \text{ iff } tt \in \{\mathit{ff}, val_{M,\beta}(c - c \doteq 0)\} \\
&= tt \text{ iff } tt \in \{\mathit{ff}, tt \text{ iff } val_{M,\beta}(c - c) = val_{M,\beta}(0)\} \\
&= tt \text{ iff } tt \in \{\mathit{ff}, tt \text{ iff } I(-)(val_{M,\beta}(c), val_{M,\beta}(c)) = I(0)\} \\
&= tt \text{ iff } tt \in \{\mathit{ff}, tt \text{ iff } 0 = 0\} \\
&= tt \text{ iff } tt \in \{\mathit{ff}, tt\} \\
&= tt
\end{aligned}
$$

In the particular case of Example 1 no matter what logic variable assignment β we use the formula is always valid. In this case we call the first-order structure $M = (\mathsf{D}, I)$ a *model* for that formula.

1.2.2. Dynamic Logic

Dynamic logic extends first-order logic by adding the possibility to reason about programs. Again this introduction is relatively short and for more information we refer to Harel et al. (2000). In this thesis we follow the approach by Saul Kripke (1963). He introduced *Kripke structures* that consist of the (reachable) states a program can be in and transitions on how to go from one state to another. The interpretation over the symbols on which the structure is defined, in our case: program variables, may vary between states. In our use of Kripke structures

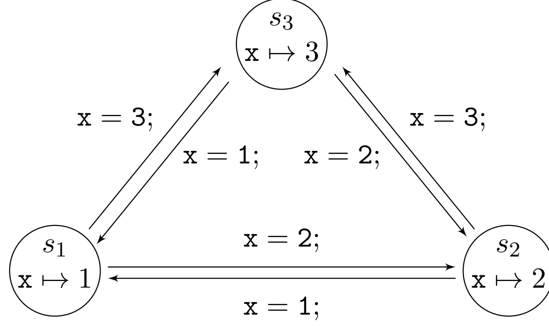

Figure 1.1.: A graph representing a Kripke structure on the program variable `x`.

we leave the concrete domain the same in all states. To make a distinction between symbols whose interpretation is allowed to vary between states (again, here: program variables) and those that are not (such as the interpretation of mathematical operators or other functions), we divide the set $\mathcal{F}$ (see Definition 2) into $\mathcal{F}_r$ and $\mathcal{F}_n$. The symbols in $\mathcal{F}_r$ are *rigid* symbols whose interpretation *cannot* be changed, while the symbols in $\mathcal{F}_n$ are called *non-rigid* symbols and their interpretation may differ between states. This means that we consider program variables to be 0-ary elements of the set $\mathcal{F}_n$.

One could represent such a Kripke structure in a graph, for example the one shown in Figure 1.1, where a simple structure is given for a single program variable `x` with a concrete domain of the integers $\{1, 2, 3\}$.

The graph is shown as a way to represent Kripke structures however in the rest of this thesis we represent a state solely by its interpretation of program variables. Transitions from one state to another are the result of executing a program. To simplify the reasoning we transform these programs into *updates* that directly describe the differences between the target state and the starting state. These updates are thus an additional modality next to programs to model transitions between states. We formally introduce these updates in the coming definitions.

To include these states, updates and program variables into our logic, we extend it as follows.

Definition 5 (Dynamic Logic Signature)**.** *A* dynamic logic signature *is a tuple* $\Sigma = (\mathcal{F}, \mathcal{P}, \mathcal{V})$ *where* $\mathcal{P}$ *and* $\mathcal{V}$ *are as in Definition 1 and* $\mathcal{F} = \mathcal{F}_n \cup \mathcal{F}_r$ *is the join of a set of* non-rigid function symbols *and a set of* rigid function symbols.

Definition 6 (Dynamic Logic Syntax)**.** *The first-order syntax is extended as follows, where* $f \in \mathcal{F}_r$*,* $p \in \mathcal{P}$*,* $y \in \mathcal{V}$ *and* $\mathtt{x} \in \mathcal{F}_n$ *(with arity 0).*

$$t ::= f(t, \ldots, t) \mid \mathtt{x} \mid y \mid \mathit{if}(\varphi)\mathit{then}(t)\mathit{else}(t) \mid \{\mathcal{U}\}t$$

$$\varphi ::= \mathrm{true} \mid \mathrm{false} \mid p(t, \ldots t) \mid \varphi \;\&\; \varphi \mid (\varphi \mid \varphi) \mid \varphi \;\text{->}\; \varphi \mid\, !\,\varphi \mid \forall y.\varphi \mid \exists y.\varphi \mid t \doteq t \mid \{\mathcal{U}\}\varphi \mid [\mathtt{p}]\varphi$$

$$\mathcal{U} ::= (\mathtt{x} := t \parallel \ldots \parallel \mathtt{x} := t)$$

$$\mathtt{p} ::= \mathtt{x = } t \mid \mathtt{p;p} \mid \mathbf{if}\ \mathtt{(}\varphi\mathtt{)\ \{p\}}\ \mathbf{else}\ \mathtt{\{p\}} \mid \mathbf{while}\ \mathtt{(}\varphi\mathtt{)\ \{p\}}$$

Terms $f(t_1, \ldots, t_n)$ *and formulas* $p(t_1, \ldots, t_n)$ *must respect the arities of the symbols* f *and* p*, respectively. Terms and formulas that appear inside programs may not contain any logic variables, quantifiers, updates, or nested programs.*

The syntax now includes updates and programs. As a short-hand, if the **`else`** part of a conditional statement is empty, we may omit it. For example, **`if`** `(i > 3) { i = 3 }` **`else`** `{}` may be written as **`if`** `(i > 3) { i = 3 }`.

Updates represent transitions between states in a Kripke structure and the symbols $\{,\}$ around an update indicate application of this transition. On account of this the term $\{\mathcal{U}\}t$ means that t is evaluated in the state that is reached after applying transition $\mathcal{U}$ in the current state.

The symbol $[\mathtt{p}]\varphi$ denotes that *if* the execution of p terminates, φ holds in the (any) state in which it terminates (box operator, *partial correctness*). Normally one has two ways to include a program p in dynamic logic formulas. The other option is the diamond operator, denoting *total correctness*: $\langle \mathtt{p} \rangle \varphi$ means that the execution of program p terminates and that in any state reached after termination the formula φ holds. In this thesis we do not concern ourselves with a situation in which p does not terminate. An interesting work on the combination of this dynamic logic and its abstraction framework described in Chapter 2 with relation to termination can be found in (Niedermann, 2011).

Definition 7 (States)**.** *A* state *is a function mapping program variables to concrete values* $s : \mathcal{F}_n \to \mathsf{D}$*. The set of all states is denoted by* $\mathcal{S}$*.*

Similar to logic variable assignments β, a state assigns values to program variables. The difference is that in a dynamic logic it is possible to refer to different states within a formula (using the $[\mathtt{p}]\varphi$ construction) and that the interpretation of a program variable may differ between those states. Opposed to that the logic variables (and other symbols) have, likewise to first-order logic, the same value independent of where they syntactically occur.

Definition 8 (Dynamic Logic Semantics)**.** *The evaluation function from first-order logic is extended with the current state s as an additional parameter. The evaluation function val is defined analogous to the first-order case. We mention here only the additional cases.*

Updates $\mathcal{U}$ *are evaluated to a result state* $val_{M,s,\beta}(\mathcal{U}) \in \mathcal{S}$ *and programs* p *to a set of states* $val_{M,s,\beta}(\mathtt{p}) \subseteq \mathcal{S}$ *where the cardinality of* $val_{M,s,\beta}(\mathtt{p})$ *is either 0 or 1.*

A formula φ *is called* valid *if and only if* $val_{M,s,\beta}(\varphi) = tt$ *for all fist-order structures* (D, I)*, states s and logic variable assignments* β*.*

The evaluation function for the terms and formulas syntax-constructors adopted from first-order logic remains the same, the additional constructors are defined below:

$$
\begin{aligned}
val_{M,s,\beta}(\mathtt{x}) &= s(\mathtt{x}) \\
val_{M,s,\beta}(\{\mathcal{U}\}t) &= val_{M,s',\beta}(t) \quad \text{where } s' = val_{M,s,\beta}(\mathcal{U}) \\
val_{M,s,\beta}(\{\mathcal{U}\}\varphi) &= val_{M,s',\beta}(\varphi) \quad \text{where } s' = val_{M,s,\beta}(\mathcal{U}) \\
val_{M,s,\beta}([\mathtt{p}]\varphi) &= tt \quad \text{iff} \quad ff \notin \{val_{M,s',\beta}(\varphi) \mid s' \in val_{M,s,\beta}(\mathtt{p})\} \\
val_{M,s,\beta}(\mathtt{x}_1 := t_1 \parallel \ldots \parallel \mathtt{x}_n := t_n) &= \big\{\mathtt{x} \mapsto s(\mathtt{x}) \mid \mathtt{x} \notin \{\mathtt{x}_1, \ldots, \mathtt{x}_n\}\big\} \cup \\
&\quad \big\{\mathtt{x} \mapsto val_{M,s,\beta}(t_k) \mid \mathtt{x} = \mathtt{x}_k \text{ and } \mathtt{x} \notin \{\mathtt{x}_{k+1}, \ldots, \mathtt{x}_n\}\big\} \\
val_{M,s,\beta}(\mathtt{x = }\ t) &= \big\{val_{M,s,\beta}(\mathtt{x} := t)\big\} \\
val_{M,s,\beta}(\mathtt{p1;p2}) &= \big\{val_{M,s',\beta}(\mathtt{p2}) \mid s' \in val_{M,s,\beta}(\mathtt{p1})\big\} \\
val_{M,s,\beta}(\mathtt{\textbf{if}(}g\mathtt{)\{p1\}\ \textbf{else}\ \{p2\}}) &= \begin{cases} val_{M,s,\beta}(\mathtt{p1}) & \text{if } val_{M,s,\beta}(g) = tt \\ val_{M,s,\beta}(\mathtt{p2}) & \text{otherwise} \end{cases} \\
val_{M,s,\beta}(\mathtt{\textbf{while}\ (}g\mathtt{)\ \{p\}}) &= \begin{cases} \bigcup_{s_1 \in S_1} val_{M,s_1,\beta}(\mathtt{\textbf{while}(}g\mathtt{)\ \{p\}}) & \text{if } val_{M,s,\beta}(g) = tt \\ \{s\} & \text{otherwise} \end{cases} \\
&\quad \text{where } S_1 = val_{M,s,\beta}(\mathtt{p})
\end{aligned}
$$

A program is evaluated to the set of all states it may terminate in. Since we only consider deterministic programs this set is either empty (in case of non-termination) or the cardinality of the set is one.

Note that for a program formula $[\mathtt{p}]\varphi$ to hold, φ should hold in *all* result states from program $\mathtt{p}$, which corresponds to the definition of partial correctness.

As can be seen from their semantics, updates describe the difference in the interpretation of program variables from the current state to the state reached after applying this update. Since all of the *elementary* updates to program variables are evaluated in parallel, conflicts may arise, i.e. when $\mathtt{x}_i = \mathtt{x}_j$ for $i \neq j$. In this case the semantics ensures that the rightmost update "wins", and thus overwrites the effect of earlier elementary updates to the same program variable (*last-one wins semantics*).

EXAMPLE 2:

To prevent this example from growing lengthy, we show how the formula $[\mathtt{y = 1}](\mathtt{y} \doteq 1)$ is evaluated.

$$\begin{aligned}
val_{M,s,\beta}([\mathtt{y = 1}](\mathtt{y} \doteq 1)) &= tt \quad \text{iff} \quad ff \notin \{val_{M,s',\beta}(\mathtt{y} \doteq 1) \mid s' \in val_{M,s,\beta}(\mathtt{y = 1})\} \\
&= tt \quad \text{iff} \quad ff \notin \{val_{M,s',\beta}(\mathtt{y} \doteq 1) \mid s' \in \{val_{M,s,\beta}(\mathtt{y} := 1)\}\} \\
&= tt \quad \text{iff} \quad ff \notin \{val_{M,s',\beta}(\mathtt{y} \doteq 1) \mid s' \in \{\mathtt{x} \mapsto s(\mathtt{x}) \mid \mathtt{x} \notin \{\mathtt{y}\}\} \cup \\
&\qquad\qquad\qquad \{\mathtt{y} \mapsto val_{M,s,\beta}(1)\}\} \\
&= tt \quad \text{iff} \quad ff \notin \{val_{M,s',\beta}(\mathtt{y} \doteq 1)\}, \ s' \text{as the only option above} \\
&= tt \quad \text{iff} \quad ff \notin \{tt \text{ iff } val_{M,s',\beta}(\mathtt{y}) = val_{M,s',\beta}(1)\} \\
&= tt \quad \text{iff} \quad ff \notin \{tt \text{ iff } s'(\mathtt{y}) = I(1)\} \\
&= tt \quad \text{iff} \quad ff \notin \{tt \text{ iff } val_{M,s,\beta}(1) = I(1)\} \\
&= tt \quad \text{iff} \quad ff \notin \{tt \text{ iff } I(1) = I(1)\} \\
&= tt \quad \text{iff} \quad ff \notin \{tt\} \\
&= tt
\end{aligned}$$

1.2.3. Sequent Calculus

To reason about the validity of formulas we use a Gentzen-style sequent calculus (Gentzen, 1934). This section provides a brief introduction to sequent calculi in general and the notation used throughout this thesis. A *sequent* has the following form:

$$\varphi_1, \ldots, \varphi_n \Longrightarrow \varphi_m, \ldots \varphi_l$$

Where we may abbreviate these lists of formulas to:

$$\Gamma \Longrightarrow \Delta$$

Or, to lift out certain formulas in particular, we can abbreviate it to:

$$\Gamma, \varphi_i \Longrightarrow \varphi_j, \Delta$$

We call the collection of formulas in Γ the *antecedent*, those in Δ the *succedent*. One could consider this sequent as a meta-formula with implication as its main connective. That is, the formula is true if one of the formulas in Γ is false or one of the formulas in Δ is true. The semantics of sequents is defined as:

$$val_{M,s,\beta}(\Gamma \Longrightarrow \Delta) = val_{M,s,\beta}(\bigwedge \Gamma \mathrel{-\!\!>} \bigvee \Delta)$$

A calculus rule is an inference rule of the following general form, using *seq* as an abbreviation of a sequent:

$$\frac{seq_1 \qquad \dots \qquad seq_n}{seq}$$

Textual one can describe this rule as follows: "if all sequents $seq_1, \dots, seq_n$ hold, then *seq* also holds". The sequent *seq* is therefore called the *conclusion* of the rule and the sequents $seq_1, \dots, seq_n$ its premises. A rule is *sound* if and only if the validity of its premises implies the validity of its conclusion.

Using these rules a *proof tree* is constructed. The root node of the tree is annotated with the sequent to be proven. For instance, assume we want to prove that formula φ is valid, we start our sequent proof with the root node:

$$\Longrightarrow \varphi$$

On every leaf sequent in the tree a calculus rule can be *applied* if the conclusion of the rule matches that leaf sequent. Application of the rule adds the rule's premises as new children to this leaf, which now becomes a node in the tree and the premises are new leaf sequents. In any tree constructed in this way, validity of all the leaves implies the validity of the root sequent, provided that all the applied rules are sound. If a tree is constructed in which all leaves are obviously valid, we have successfully proven the validity of the root sequent. Valid sequents can be identified in the calculus by having *closed* as their (single) premise.

Given the semantics of a sequent, we can provide the following rules for sequents that close a branch in the tree:

$$\textsf{closeTrue}\ \frac{closed}{\Gamma \Longrightarrow \text{true}, \Delta} \qquad \textsf{closeFalse}\ \frac{closed}{\Gamma, \text{false} \Longrightarrow \Delta} \qquad \textsf{closeAxiom}\ \frac{closed}{\Gamma, \phi \Longrightarrow \phi, \Delta}$$

Another comprehensive rule is the andRight rule for formulas with the conjunction as their main connective. To prove that the formula $\phi \ \& \ \psi$ holds, we need to show that ϕ holds and that ψ holds:

$$\textsf{andRight}\ \frac{\Gamma \Longrightarrow \phi, \Delta \qquad \Gamma \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \phi \ \& \ \psi, \Delta}$$

Similar, for rules with the implication as their main connective, we have the impRight rule. To show that $\phi \mathrel{-\!\!>} \psi$ holds we need to show that if ϕ holds, ψ has to hold as well:

$$\textsf{impRight}\ \frac{\Gamma, \phi \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \phi \mathrel{-\!\!>} \psi, \Delta}$$

Using these rules and the first-order syntax provided in Definition 2 we can now use the sequent calculus to show the logic validity of the simple formula false –> (true & true), which

we place in the sequent $\Longrightarrow$ false -> (true & true). This gives us the following small proof tree of rule applications:

$$\text{impRight}\ \dfrac{\text{andRight}\ \dfrac{\text{closeTrue}\ \dfrac{\textit{closed}}{\text{false} \Longrightarrow \text{true}} \qquad \text{closeTrue}\ \dfrac{\textit{closed}}{\text{false} \Longrightarrow \text{true}}}{\text{false} \Longrightarrow \text{true \& true}}}{\Longrightarrow \text{false -> (true \& true)}}$$

However this is not the only application of rules that shows the sequent to be valid. Another (shorter) option is:

$$\text{impRight}\ \dfrac{\text{closeFalse}\ \dfrac{\textit{closed}}{\text{false} \Longrightarrow \text{true \& true}}}{\Longrightarrow \text{false -> (true \& true)}}$$

The selection of rules to apply and in which order these rules are applied can dramatically affect the length of the proof tree. An imperfect selection or order may even permanently obstruct the discovery of a sequent's proof, while a different selection or order does achieve to prove that sequent.

1.2.4. The KeY Approach

The dynamic logic as introduced in this section is the core of the KeY System, a formal software development tool that aims to integrate design, implementation, formal specification, and formal verification of object-oriented software as seamlessly as possible (Ahrendt et al., 2005). The KeY System includes a semi-automatic theorem prover for Java programs and is able to parse proof obligations from JML markup in Java-source files. KeY does as much automatic as is possible, but allows a user to provide proof steps in the form of rule applications when automation fails at some point. This automation is internally achieved by proof search in the dynamic logic calculus, realized by an automated prover.

An important feature of the KeY System is its integration with external SMT (Satisfiability Modulo Theories) solvers that can be used to solve decision problems with respect to a certain theory (such as integers, real numbers, arrays etc.). At the time of writing the most recent version of the KeY System is 1.6.0 and integrates with CVC3, Simplify, Yices and Z3 (Barrett and Tinelli, 2007; Detlefs et al., 2005; Dutertre and de Moura, 2006; de Moura and Bjørner, 2008).

The goal of this thesis is to achieve a fully automated system for specific application cases (loop abstraction and information flow analysis), without the need for user interaction while maintaining a high degree of completeness.

1.3. Abstract Interpretation

In their 1977 paper, Patrick and Radhia Cousot introduced the concept of abstract interpretation of programs. They note that when statically analyzing a program one could, instead of tracking the values of variables in a concrete universe, do so in an abstract universe. As an intuitive example, first published by Sintzoff (1972), they consider lifting the concrete domain of integers to the abstract domain of the so-called 'sign language' or 'rule of signs' which we use only in an informal manner in this section. The concrete operation $-5 * 8$ could for example be

translated to the abstract operation $-(+) * (+)$, which evaluates to $(-) * (+)$, which evaluates to $(-)$. This gives us a sound approximation of the outcome of the operation, namely that the outcome is negative. Obviously, being called an abstraction and an approximation, this gives us incomplete results but the approach can still be useful in providing the programmer or compiler (or: automatic static verifier) means to check for certain properties of programs such as type checking, optimizations, partial correctness proofs etc.

The main advantage that can be obtained from applying abstraction during static program analysis is located in the area of proving properties of programs containing loops. Independent of the underlying mechanism causing the loops in the program, be it a while loop or a collection of goto-statements, a general problem for verification is that in many occasions the number of loop iterations is unknown when statically reviewing the source code. Cousot and Cousot use the notion of a *fixed point*, a collection of states (Definition 7) that always contains the resulting state when program execution exits the loop, independent of the number of executions of this loop.

1.3.1. Fixed Points

A fixed point can be viewed as a mathematical fixed point. That is, we have a function that when applied any number of times on a certain input, the output is always in the same set. In our case the function f is a loop-part of a program and the input the state s of the program before execution of the loop.

Using the syntax from our dynamic logic (Definition 6), consider the general loop **while**(g){b}. The search for a fixed point of this loop means that we aim to find a set of states S_{fp} such that:

$$\forall n \ . \ val_{M,s,\beta}(\mathtt{b}^n) \subseteq S_{fp}$$

Where $\mathtt{b}^n$ means n repetitive executions of the loop's body b. In a concrete domain it is for most loops not possible to find such a fixed point automatically, since the values of the program variables updated in the loop depend on the number of executions. However in an abstract domain this becomes possible. For example:

```
i = 1;
while (b) {
  i = i + 1;
}
```

In a concrete domain we are not able to automatically find a fixed point for this loop for any number of executions of the while loop. A fixed point would be here e.g. the set of all whole numbers for program variable i. Usually we want to prevent information loss by finding the smallest possible fixed point. In particular, we want to find this in an automatic fashion. In the abstract domain we can specify the fixed point $\{\mathtt{i} \mapsto (+)\}$ (assuming i is the only existing program variable). In fact, if we introduce another sign $(\pm)$, indicating all integers, we can even claim that we are always able to find such a fixed point for any program with any starting state: an (abstract) state where each program variable has the value $(\pm)$. This state is always a sound, but also very imprecise, fixed point in the abstract domain.

In Chapter 2 it is shown how this concept is adopted in our dynamic logic by Bubel et al. (2009) to automatically derive loop invariants for programs dealing with integers. The contribution made in that chapter is an extension to this adaptation for objects.

1.4. Information Flow Analysis

There exist multiple ways to restrict and regulate access to data, however what happens to this data after a process has obtained access to it cannot be controlled by regular access control. The data, or information about the data, may flow from the process that has access to it to other processes that do not. For example, a program P is given access to a file which cannot be accessed by process Q (where Q may be a human user of the program P). If P now reads the content of the file and streams that directly to process Q, the information from the file flows to the unauthorized process. This cannot be prevented by access control, since the process Q does not access the file in question but only gets the information in the file. To detect such a leak in program P, we perform a static analysis on its source code to detect where information goes during the program's execution. This type of analysis is known as *information flow analysis*.

In general when performing information flow analysis we separate the program variables in different sets of security levels (Denning, 1976). In most literature one finds only two levels, Low and High, but a different number of levels can be used. Moreover, the security levels are considered to be part of a security lattice. In the example with only the levels Low and High, we have the lattice in which information is allowed to flow between variables of the same security level, or from Low to High but not from High to Low. Again, more complex security lattices may exist. The concept is that the values of Low variables are observable to everyone, such as to the process Q in our example, while the values of High variables are not. If using information flow analysis we can prove that this is the case, we say to have proven the *non-interference* property for this program: other processes are not able to derive any information from the High variables using only the information from the Low variables.

To provide some examples of information flow and non-interference, let `l1` and `l2` be Low, and `h1` and `h2` High variables. Then the following program-fragments result in unwanted information flow:

1. `l1 = h1`
   After execution of this statement `l1` contains the same value as `h1`.

2. `l2 = h1 / h2`, where `/` is the division operator
   Here the variable `l2` does not give all information about the High variables but it provides some partial information on their ratio.

In these examples the (partial) information from the value of the High variables flows directly to the Low variables (*explicit* flow) and this information is thus observable by other processes. It is also possible that information flows indirectly from High to Low variables (*implicit* flow), as the following examples show:

1. `if (h1 > 0) {l1 = 10} else {l1 = 20}`
   The value of `l1` leaks the information whether `h1` was bigger than 0.

2. `l1 = 0; l2 = 0; if (h1 > h2) {l1 = 1} else {l2 = 1}`
   A slightly more complex program. Information is leaked since another process can observe if either `l1` or `l2` has been set to 1, and the information that is leaked is partial information on the relation $>$ between `h1` and `h2`.

One way of statically *enforcing* a safe information flow is to extend the regular type system (of bools, integers etc.) with a security type. Often these types are Low and High as can be expected from previous examples. The *security type system* is a collection of rules that enforces

information flowing only in the safe direction and not the other way around. These rules are enforced at compile-time which prevents a run-time overhead. Many different typing systems have been developed with this concept, for an overview see Sabelfeld and Myers (2003). One of the problems a typing approach to information flow analysis faces, is called ‘label creep’. Since a safe approach would always apply the highest of possible security labels/types to a term, it causes the labels to always creep upward to a higher security level and thus becomes too restrictive for practical use (Denning, 1982).

In this thesis we take a different approach to the information flow problem, via the use of *dependencies*: we identify the set of variables on which the value of a variable `x` depends, and call this set the dependency set of `x`. This is further introduced in Section 3.1.

1.4.1. Declassification

In practical applications the requirement that an outside process cannot learn anything from the High variables by only observing the program and its Low variables is too restrictive, since most programs actually *intend* to leak some information about the High variables. Consider for example a login-application. A correct username and password combination must not be leaked directly to the user (e.g. “The username and password you provided were incorrect. It should have been *jdoe* and *monday*”). However, whether or not the login combination was correct can always be derived from the fact whether or not the system allowed you in, hence this partial information is allowed to leak.

To obtain a more practical information flow analysis for realistic applications one should therefore take into account the possibility to *declassify* certain parts of High variables. For example, the variable `correctPassword` should not flow into a Low variable, but the result of the operation `correctPassword == providedPassword` is allowed to flow out of the program. A common approach is that in the program’s source code, next to the addition of security levels for each variable, the programmer can also specify certain operations that may leak (some) information. Using the terminology of Sabelfeld and Sands (2005): when specifying declassification one specifies *who* may release *what* information, and *where* in the program code or *when* during the execution this is allowed to happen.

In the remainder of this thesis we do however not come back to declassification. For an interesting approach to declassification which allows for dynamic changes in the information flow policy, we refer the reader to Broberg and Sands (2006).

1.4.2. Side Channels

Next to the direct and implicit ways in which information can leak via the Low variables, information on the High variables can also leak via other channels observable to remote processes (or adversaries). Take for example the program:

```
while (h > 0) {
  someHighLevelOperation();
}
```

Where we assume that in `someHighLevelOperation()` no information flows to Low variables. However, the execution of this method takes some *time*, and the number of executions of this method is affected by the value of `h`. So here time is a side channel that gives some information on the value of `h`.

Another example of a side channel leak would be the following program:

```
if (h == true) {
```

```
    x = readFromCD();
}
```

In this case one might deduct from whether or not the cd-drive starts spinning the value of `h` was true or false. More complicated side channels exist, and are used in methods such as Differential Power Analysis (Kocher et al., 1999).

In the rest of this thesis we assume the absence of side channel leaks. For an overview of side channel attacks, see Le et al. (2008).

1.4.3. Information Flow Analysis in Dynamic Logic

In Chapter 3 we describe the extension made by Bubel et al. (2009) on the dynamic logic from Section 1.2.2 to include information flow analysis. This extension however is too restrictive when it comes to value sensitivity and for that reason fails to prove the non-interference property for certain programs, although they are in fact secure. This thesis contributes in that same chapter to the field by extending the logic framework further such that a larger set of secure programs can be identified correctly.

1.5. Outline

The remainder of this thesis can be outlined as follows. In Chapter 2 we describe the extension made to dynamic logic by Bubel et al. (2009) for the automatic derivation of loop invariants, while simultaneously extending this for the inclusion of objects. In Chapter 3 the approach by Bubel et al. (2009) is described to extend the dynamic logic for information flow analysis, followed by a variation on this extension that allows for a better value-sensitive analysis. We summarize the conclusions in Chapter 4 and give an overview of future research topics, which includes extending the information-flow analysis for objects.

2. Abstract Interpretation

This chapter continues on the work by Bubel et al. (2009), in which the dynamic logic calculus is extended to include abstract domains and a calculus rule that allows the abstraction of a while loop, as briefly introduced in Section 1.3. The calculus and loop abstraction presented in that paper however only addresses the integer and (implicitly) the boolean domain. This chapter re-explains their signature, syntax and semantics while simultaneously extending them for the *sort* (or: object/class) domain.

Example 3:

To illustrate the extension to the definitions of signatures, syntax, semantics and calculus rules we use a running example, in which the set of sorts consists of integers and tuples:

```
Sort int;
Sort Tuple {
  int fst;
  int snd;
  Tuple next;
};
```

We use the word 'original' when referring to the work presented in the paper by Bubel et al. (2009). The intention is to distinguish between the work that is reused and the work that is contributed with this thesis.

2.1. A Dynamic Logic with Objects

2.1.1. Signature and Syntax

The signature from the dynamic logic (Definition 5) is adapted to include a set of sort symbols (such as **`int`**, `Tuple` etc.). In the original version program variables had their own set, but as discussed in Section 1.2.2 we move them in the set of functions, which becomes a collection of two different types of functions: rigid functions (their interpretation is independent of the state in which they are evaluated) and non-rigid functions (their meaning may vary with the state). This allows us to represent *program variables* as 0-ary, non-rigid function symbols and object *fields* as unary, non-rigid function symbols.

Definition 9 (Signature)**.** *A* signature *is a tuple* $\Sigma = (\mathcal{S}, \mathcal{F}, \mathcal{P}, \mathcal{V})$*, where* $\mathcal{S}$ *is a set of* sorts*,* $\mathcal{F}$ *a set of* function symbols*,* $\mathcal{P}$ *a set of* predicate symbols *and* $\mathcal{V}$ *a set of logic* variables*. Function and predicate symbols have fixed arities.* $\mathcal{F}$ *can be further partitioned into* $\mathcal{F}_r \cup \mathcal{F}_n$ *for rigid and non-rigid functions respectively. The set of rigid functions with arity a is denoted as* $\mathcal{F}_{r,a}$*, similar the set of non-rigid functions with arity a as* $\mathcal{F}_{n,a}$*.*

EXAMPLE 4:

In our example, we define $\mathcal{S} := \{\texttt{int}, \texttt{Tuple}\}$. The set of functions $\mathcal{F}$ is defined as the union of the following sets:

$$\begin{aligned}
\mathcal{F}_{n,0} &:= \text{Program variables} \\
\mathcal{F}_{n,1} &:= \{\texttt{fst} : \texttt{Tuple} \to \texttt{int}, \\
&\quad\ \texttt{snd} : \texttt{Tuple} \to \texttt{int}, \\
&\quad\ \texttt{next} : \texttt{Tuple} \to \texttt{Tuple}\} \\
\mathcal{F}_{r,0} &:= \{\ldots,\ -2 : \texttt{int},\ -1 : \texttt{int},\ 0 : \texttt{int},\ 1 : \texttt{int},\ 2 : \texttt{int},\ \ldots\} \\
\mathcal{F}_{r,1} &:= \{\texttt{idInt} : \texttt{int} \to \texttt{int},\ \texttt{idTup} : \texttt{Tuple} \to \texttt{Tuple}\} \\
\mathcal{F}_{r,2} &:= \{\texttt{intPlus} : \texttt{int} \to \texttt{int} \to \texttt{int},\ \texttt{intMinus} : \texttt{int} \to \texttt{int} \to \texttt{int}\}
\end{aligned}$$

The only information provided about the function symbols in the signature is their name and type. Their meaning is described in Definition 12. The set of predicate symbols $\mathcal{P}$ that is used in our example is:

$$\mathcal{P} := \{\texttt{leq} : \texttt{int} \times \texttt{int},\ \texttt{geq} : \texttt{int} \times \texttt{int},\ \texttt{eq} : \texttt{int} \times \texttt{int}\}$$

In the rest of the examples we use infix notation for simplicity, i.e. $\texttt{+} \mapsto \texttt{intPlus}$, $\texttt{-} \mapsto \texttt{intMinus}$, $\leq \mapsto \texttt{leq}$, $\geq \mapsto \texttt{geq}$, $\doteq \mapsto \texttt{eq}$. Finally the set of program variables is left unspecified, but we stick to the use of `i`, `j`, `k` and so on for program variables of sort **`int`** and `r`, `s`, `t` and so on for program variables of sort `Tuple`.

As usual, we assume terms, formulas, updates etc. to be well-formed with respect to the syntax. The original syntax is updated consistently to include the notion of sorts.

Definition 10 (Syntax)**.** *The syntax is defined as follows, where $f \in \mathcal{F}_r$, $\texttt{x} \in \mathcal{F}_{n,0}$, $b \in \mathcal{F}_{n,1}$, $p \in \mathcal{P}$, and $y \in \mathcal{V}$.*

$$\begin{aligned}
loc &::= \texttt{x} \mid t.b \\
t &::= \bigcup_{s \in \mathcal{S}} t_s \\
\forall s \in \mathcal{S}\ .\ t_s &::= f(t, \ldots, t) \mid \texttt{x} \mid y \mid t.b \mid \mathit{if}(\varphi)\mathit{then}(t_s)\mathit{else}(t_s) \mid \{\mathcal{U}\}t_s \\
\varphi &::= \text{true} \mid \text{false} \mid p(t, \ldots, t) \mid \varphi \mathbin{\&} \varphi \mid (\varphi \mid \varphi) \mid \varphi \mathbin{{-}{>}} \varphi \mid\ !\,\varphi \mid \\
&\qquad \forall Ty.\varphi \mid \exists Ty.\varphi \mid t \doteq t \mid \{\mathcal{U}\}\varphi \mid [\texttt{p}]\varphi \\
\mathcal{U} &::= (\forall Tx.\varphi \mathbin{{-}{>}} loc := t \parallel \ldots \parallel \forall Tx.\varphi \mathbin{{-}{>}} loc := t) \\
\texttt{p} &::= loc\ \texttt{=}\ t \mid \texttt{p;p} \mid \textbf{if}\ (\varphi)\ \{\texttt{p}\}\ \textbf{else}\ \{\texttt{p}\} \mid \textbf{while}\ (\varphi)\ \{\texttt{p}\}
\end{aligned}$$

Terms $f(t_1, \ldots, t_n)$ and formulas $p(t_1, \ldots, t_n)$ must respect the arities of the symbols f and p, respectively. Terms and formulas that appear inside programs may not contain any logic variables, quantifiers, updates, or nested programs. All terms, formulas and updates have to be well-typed according to the signature of the used symbols.

Remark 1. *By well-typed it is meant that the constructors for terms, formulas and updates should be considered as a conditional definition where necessary. E.g.:*

$$\frac{loc \in t_s \qquad t \in t_{s'} \qquad s = s'}{loc\ \texttt{=}\ t \in \texttt{p}}$$

$$\frac{f : s_1 \to \cdots \to s_n \to s \qquad t_1 \in t_{s_1} \quad \ldots \quad t_n \in t_{s_n}}{f(t_1, \ldots, t_n) \in t_s}$$

$$\frac{t \in t_{s'} \qquad b : s' \to s}{t.b \in loc}$$

Remark 2. *In the remainder of this thesis we omit the subscript to denote the sort of a term if this does not cause confusion.*

Example 5:

Some correctly typed terms and formulas in our example include:

- `3` which is a term of type **`int`**
- `t` which is a term of type `Tuple`
- `t.fst` which is a term of type **`int`**
- `t.next.snd` $\geq$ `7` which is a formula
- `t` $\doteq$ `t.next` which is a formula

Since most syntax symbols are explained in Section 1.2.2, at this point we limit the explanation to the new symbols.

By convention the symbol *loc* is used as the placeholder for locations. A location is a non-rigid functions symbol which can be either a program variable or a field with an argument. The general syntax for a function application is $f(t)$. For non-rigid functions representing a field we usually take the familiar postfix notation $t.f$ as is common in most object-oriented programming languages. To further emphasize the difference we use the symbol b for non-rigid functions representing a field, thus having the notation $t.b$.

Opposed to the syntax from Definition 6, variables and terms can now represent values of different domains. To ensure a correct sort-preservation some additional notation in sub-script is added. The sort of a quantified logic variable in formulas $\forall y.\varphi$ and $\exists y.\varphi$ is added as well. As pointed out in Remark 2, we omit the subscript that denotes the sort of a term if it does not cause confusion. To reduce the complexity of notation we also omit the sort in a quantifier (i.e. we write $\forall x$ instead of $\forall Tx$) when this is of no importance or can be easily derived from the context.

Updates have been changed as well compared to the original syntax; a universal quantified variable may appear in an elementary update (x of sort T)[1]. The need to extend the original syntax of updates arises from the way in which the fixed point of a while loop (Section 1.3.1) is derived automatically, as discussed in Section 2.2.3. The formula φ in such a quantified update serves as a guard; the update only applies when φ holds. Quantified updates as a concept for modeling the stack and heap during symbolic execution were first introduced by Rümmer (2006).

[1] The restriction that only one variable can be quantified in an elementary update can easily be lifted, but it proves sufficient for the purpose of this thesis to have only one quantified variable and simplifies the presentation.

Where desired we may write $\forall T x.\varphi(x)$ -> $loc[x] := t[x]$ to keep explicit track of the quantified variable. In case the quantified variable does not appear in the free variables of neither the locations nor the term, it may be omitted in our notation. Similar if φ is the truth constant true, φ may be omitted. For example, the elementary update $\forall$ `Tuple` $t.$ true -> `i` := 3 can be abbreviated to `i` := 3.

EXAMPLE 6:

Let `p` denote the following program:

```
t.fst = 0;
i = 1;
while (g) {
  t = t.next;
  t.fst = i;
  i = i + 1;
}
```

We can construct a simple statement on this program, such as `i` > 0 -> [`p`](`t.fst` ≥ 0). This formula holds, if when the program variable `i` holds a value higher than zero, then after execution of program `p` the `fst` field of tuple `t` holds a value greater or equal than zero. In the next section we introduce a semantics to capture this intended interpretation.

2.1.2. Interpretation and Semantics

Given a universe of values D, we take $\mathsf{D}_s \subseteq \mathsf{D}$ as the values in D of the sort $s \in \mathcal{S}$. We also assume that a partial order $\preceq$ on each D_s exists (as adopted from Beckert et al. (2007), Chapter 3).

Definition 11 (Well-Ordered Domains)**.** *Given a domain* D, $\preceq$ *is a partial well-ordering on* D *if the following holds:*

- *Reflexivity:* $x \preceq x$ *for all* $x \in \mathsf{D}$
- *Antisymmetry:* $x \preceq y$ *and* $y \preceq x$ *implies* $x = y$
- *Transitivity:* $x \preceq y$ *and* $y \preceq z$ *implies* $x \preceq z$
- *Well-Orderedness: Each non-empty subset* $\mathsf{D}_{sub} \subseteq \mathsf{D}$ *has a least element* $min_{\preceq}(\mathsf{D}_{sub})$ *such that for all* $x \in \mathsf{D}_{sub}$ $min_{\preceq}(\mathsf{D}_{sub}) \preceq x$.

EXAMPLE 7:

We use $\mathbb{Z}$ for $\mathsf{D}_{\texttt{int}}$ and for $x, y \in \mathbb{Z}$ we have that $x \preceq y$ if:

- $x \geq 0$ and $y < 0$ or
- $x \geq 0$ and $y \geq 0$ and $x \leq y$ or
- $x < 0$ and $y < 0$ and $y \leq x$

As for the object domain $\mathsf{D}_{\texttt{Tuple}}$ any well-ordering suffices, e.g. based on Gödelization (Gödel, 1931, pages 178-179).

Definition 12 (Interpretations, States, Variable Assignments). *Given a universe* D *of values, an* interpretation I *is a function mapping every rigid function symbol* $f \in \mathcal{F}$ *with arity* n *to a function* $I(f) : \mathsf{D}^n \to \mathsf{D}$*, and every predicate symbol* $p \in \mathcal{P}$ *with arity* n *to a relation* $I(p) \subseteq \mathsf{D}^n$*.*

A state *maps each non-rigid function* $b : T_1 \times \cdots \times T_N \mapsto T$*,* $n \geq 0$*, to a function* $s(b) : \mathsf{D}_{T_1} \times \cdots \times \mathsf{D}_{T_n} \mapsto \mathsf{D}_T$*; the set of all states is denoted* $\mathcal{S}$*. A* logic variable assignment *is a function* $\beta : \mathcal{V} \to \mathsf{D}$*.*

EXAMPLE 8:

We map the function $\dot{+} : \textbf{int} \times \textbf{int} \mapsto \textbf{int}$ to:

$$I(\dot{+}) : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$
$$I(\dot{+})(x, y) = x + y \quad x, y \in \mathbb{Z}$$

Where $\dot{+}$ is a rigid function symbol of arity 2 and $+$ the mathematical plus operation defined on $\mathbb{Z}$. Similar interpretations can be given to other functions and predicates.

In the rest of this thesis when quantifying over different interpretations I we assume the meaning of regular mathematical symbols such as $+, -, \geq$, etc. to be fixed to their standard interpretation. That is, when we state 'let I_0 be any interpretation…' this implies 'let I_0 be any interpretation with $+, -, \geq$, etc. defined as standard…'.

Definition 13 (Semantics). *Given a first-order structure* $M = (\mathsf{D}, I)$ *consisting of a universe* D *and an interpretation* I*, a state* s *and a logic variable assignment* β*, we evaluate terms* t *to a value* $val_{M,s,\beta}(t) \in \mathsf{D}$*, formulas* φ *to a truth value* $val_{M,s,\beta}(\varphi) \in \{tt, ff\}$*, updates* $\mathcal{U}$ *to a result state* $val_{M,s,\beta}(\mathcal{U}) \in \mathcal{S}$*, and programs* $\mathtt{p}$ *to a set of states* $val_{M,s,\beta}(\mathtt{p}) \subseteq \mathcal{S}$*, where the cardinality of* $val_{M,s,\beta}(\mathtt{p})$ *is either* 0 *or* 1*. The evaluation function* $val_{M,s,\beta}$ *is as in Definition 8. We list the semantics of the new symbols:*

$$val_{M,s,\beta}(\forall T y.\varphi) = tt \quad \text{iff} \quad ff \notin \{val_{M,s,\beta_y^v}(\varphi) \mid v \in \mathsf{D}_T\}$$
$$val_{M,s,\beta}(\exists T y.\varphi) = tt \quad \text{iff} \quad tt \in \{val_{M,s,\beta_y^v}(\varphi) \mid v \in \mathsf{D}_T\}$$

$$\begin{aligned}
& val_{M,s,\beta}(\forall T_1 x_1;\ \varphi_1;\ loc_1 := t_1 \parallel \\
& \quad \ldots \parallel \forall T_n x_n;\ \varphi_n;\ loc_n := t_n) = \big\{F(loc) \mapsto val_{M,s,\beta}(loc) \mid ((loc, v), w) \notin \cup_{i \geq 1} dom_i \text{ for all } v, w \in \mathsf{D}\big\} \cup \\
& \qquad \big\{F(loc) \mapsto v \mid loc = loc_k \text{ and } ((loc, v), a) \in dom_k \text{ and} \\
& \qquad\quad a \in \mathsf{D}_{T_k} \text{ and } b \not\preceq a \text{ for all } ((loc, v'), b) \in dom_k \text{ and} \\
& \qquad\quad ((loc, v'), w) \notin \bigcup_{i > k} dom_i \text{ for all } w \in \mathsf{D}\big\} \\
& \qquad \text{where } F(loc) = \begin{cases} \mathtt{x} & \text{if } loc = \mathtt{x}\ (loc \in \mathcal{F}_{n,0}) \\ (b, val_{M,s,\beta}(loc')) & \text{otherwise, } loc = loc'.b \end{cases} \\
& \qquad \text{and } dom_i = \{((loc_i[x_i/a], val_{M,s,\beta_{x_i}^a}(t_i)), a) \mid a \in A_i\} \\
& \qquad \text{and where } A_i = \{d \in \mathsf{D}_{T_i} \mid val_{M,s,\beta_{x_i}^d}(\varphi_i) = tt\}
\end{aligned}$$

$$val_{M,s,\beta}(loc \doteq t) = \{val_{M,s,\beta}(\forall Ty;\ tt;\ loc := t)\} \text{ where } loc \in t_T$$
$$\text{and where } y \notin freeVar(loc) \text{ and } y \notin freeVar(t)$$

We discuss the evaluation of updates (as adopted from (Beckert et al., 2007)) in more detail. First of all, all locations that are not listed on the left-hand side of any of the elementary updates in the parallel update remain unchanged, as is expressed by mapping them to $val_{M,s,\beta}(loc)$. Secondly, those locations loc_i that are changed are updated to the corresponding term t_i on the right-hand side, where the complex condition ensures that (1) this t_i is the *last* update occurring in the parallel update, conform the 'last-one wins semantics' as described in Section 1.2.2, and that (2) the possibly quantified argument x is the *least* object that results in the location loc_i on the left-hand side of the update (*least-one wins semantics*). The function F is a mere translation from the syntactical location to the mapping that is to be updated.

EXAMPLE 9:

To show that a least-one wins semantics is required for applying parallel updates consider the following example. Let f be a non-rigid function that takes an integer as argument, % the modulo operator. Then in the update

$$\forall x.\ f(x\%2) := x$$

we have a clash on updates to both $f(0)$ and $f(1)$. That is, all odd substitutes for x update the location $f(1)$ and all even substitutes the location $f(0)$. The value to which this location is updated differs however per substitute and in the state resulting from this update each location can be evaluated to only one value. Here the least-one wins semantics determines that substitutes 0 and 1 are the least substitutes for x that update these locations. Hence the above update is equivalent to:

$$f(0) := 0 \parallel f(1) := 1$$

In a more elaborate example, consider $\mathcal{U}$ the update

$$\forall x.\ f(x\%2) := x \parallel \forall x.\ f(2 \cdot x) := x + 1$$

We write each of these quantified updates on separate lines, note however that within each line the updates are currently executed in real parallel, i.e. do not have a last-one wins semantics:

$$f(0) := 0 \parallel f(1) := 1 \parallel f(0) := 2 \parallel f(1) := 3 \parallel \ldots$$
$$f(0) := 1 \parallel f(2) := 2 \parallel f(4) := 3 \parallel f(6) := 4 \parallel \ldots$$

We have several parallel updates to locations $f(0)$ and $f(1)$, so we need a way to establish what the value of these two locations is after application of this update. Given the partial ordering from Example 7 on integers the least-one wins semantics tells us that the *first update to* $f(1)$ *wins* (since $1 \preceq 3 \preceq 5 \preceq \ldots$). For $f(0)$ the first update also wins from all other updates resulting from the first quantified update because of the least-one wins semantics (i.e. x is instantiated with 0). However in the update resulting from the second quantified update the quantified variable x is instantiated with 0 as well. In this clash the last-one wins semantics tells us that the last update to $f(0)$ wins. This means that the above update is equivalent to:

$$f(0) := 1 \parallel f(1) := 1 \parallel f(2) := 2 \parallel f(4) := 3 \parallel f(6) := 4 \parallel \ldots$$

2.1.3. Calculus

While updates describe state changes in a parallel, direct way, the box operator on a program `p` describes a state change in a more complicated manner. The value of a program variable after execution of program `p` is harder to deduce compared to after an update (e.g., because of conditional statements and while loops). Also updates suffer less from the so-called *aliasing* problem, namely that multiple program variable names may refer to the same data location in memory. For example, we can postpone a proof split (e.g. the question whether an object `x` is equal to an object `y` within the update $\mathtt{y} := \mathtt{z}$) which might simplify matters in cases where the proof split turns out to be unnecessary (e.g. if this update is followed by the update $\mathtt{x} := \mathtt{null}$ and we want to prove that $\mathtt{x} \doteq \mathtt{null}$).

It is for these reasons that we prefer reasoning with updates instead of programs. For that purpose calculus rules are added that *symbolically execute* program statements by rewriting them into updates and formulas. Consider the assignment rule displayed below.

$$\textsf{assignment}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}\{loc := t\}\ [\ldots]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[loc = t; \ldots]\varphi, \Delta}$$

We use the symbols $\mathcal{U}$ and $\ldots$ to collapse all elementary updates or program statements. In each of these cases the collapsed set of elements may be empty, in which case they can be omitted during application of this rule. To combine two updates into a parallel update (or to apply them on a term) update *rewrite rules* can be used. These are displayed in Appendix A.

For the handling of conditional statements we provide the following rule:

$$\textsf{ifElse}\ \frac{\begin{array}{c}\Gamma, \{\mathcal{U}\}g \Longrightarrow \{\mathcal{U}\}[\mathtt{p1;}\ \ldots]\varphi, \Delta \\ \Gamma, \{\mathcal{U}\}!g \Longrightarrow \{\mathcal{U}\}[\mathtt{p2;}\ \ldots]\varphi, \Delta\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathtt{if\ (}g\mathtt{)\ \{p1\}\ else\ \{p2\};}\ \ldots]\varphi, \Delta}$$

Here we split the proof tree in two separate sequents that both need to be proven; one in which the guard g is true under the update $\mathcal{U}$ and one in which it is false.

Example 10:

Consider the following proof obligation:

$$\Gamma \quad \Longrightarrow \quad [\mathtt{i=3;\ if\ (i<0)\ \{i=1\}\ else\ \{i=0\}}]\mathtt{i} \doteq 0, \Delta$$

We first apply the assignment rule:

$$\Gamma \quad \Longrightarrow \quad \{\mathtt{i} := 3\}[\mathtt{if\ (i<0)\ \{i=1\}\ else\ \{i=0\}}]\mathtt{i} \doteq 0, \Delta$$

Next, we apply the ifElse rule, resulting in two branches:

$$\Gamma, \{\mathtt{i} := 3\}\mathtt{i} < 0 \Longrightarrow \{\mathtt{i} := 3\}[\mathtt{i=1}]\mathtt{i} \doteq 0, \Delta \quad (1)$$
$$\Gamma, \{\mathtt{i} := 3\}\,!(\mathtt{i} < 0) \Longrightarrow \{\mathtt{i} := 3\}[\mathtt{i=0}]\mathtt{i} \doteq 0, \Delta \quad (2)$$

We first continue with sequent (1). After applying some simplification steps we obtain:

$$\Gamma, 3 < 0 \quad \Longrightarrow \quad \{\mathtt{i} := 3\}[\mathtt{i=1}]\mathtt{i} \doteq 0, \Delta$$

Followed by

$$\Gamma, \mathrm{false} \quad \Longrightarrow \quad \{\mathtt{i} := 3\}[\mathtt{i=1}]\mathtt{i} \doteq 0, \Delta$$

Which can be closed by the closeFalse rule. We continue in sequent (2) where we again apply the assignment rule:

$$\Gamma, \{\texttt{i} := 3\}\,!(\texttt{i} < 0) \quad \Longrightarrow \quad \{\texttt{i} := 3\}\{\texttt{i} := 0\}\texttt{i} \doteq 0, \Delta$$

After some update rewriting rules (see Appendix A.2) we obtain the sequent

$$\Gamma, \{\texttt{i} := 3\}\,!(\texttt{i} < 0) \quad \Longrightarrow \quad 0 \doteq 0, \Delta$$

Which is trivially true and we close the second sequent as well, thereby completing the proof.

For a while loop, we could choose to unwind it:

$$\text{loopUnwind}\ \frac{\begin{array}{l}\Gamma, \{\mathcal{U}\}g \Longrightarrow \{\mathcal{U}\}[\texttt{b; while(}g\texttt{) \{b\}; }\ldots]\varphi, \Delta \\ \Gamma, \{\mathcal{U}\}!g \Longrightarrow \{\mathcal{U}\}[\ldots]\varphi, \Delta\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\texttt{while (}g\texttt{) \{b\}; }\ldots]\varphi, \Delta}$$

Unwinding a loop is only successful if after a statically *known* number of iterations the condition g becomes false. In the general case where the number of iterations is not known beforehand (but for example depends on a provided parameter) induction or a loop invariant rule has to be applied:

$$\text{loopInvariant}\ \frac{\begin{array}{ll}\Gamma \Longrightarrow \{\mathcal{U}\}inv, \Delta & \text{Initial} \\ inv, g \Longrightarrow [\texttt{b}]inv & \text{Invariant} \\ inv, !g \Longrightarrow [\ldots]\varphi & \text{Use case}\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\texttt{while (}g\texttt{) \{b\}; }\ldots]\varphi, \Delta}$$

Here we need to provide an invariant *inv*, where *inv* is a formula. The sequent with the while-statement is split into three separate premises that need to hold. In the first sequent we need to show that *inv* holds in the initial state, i.e. the invariant holds before the while loop is executed (or: if the while loop is not executed at all). The second premise states that the loop invariant must be an actual invariant, i.e., that an arbitrary execution of the loop in a state satisfying the loop condition g does not invalidate the invariant. Finally in the third sequent, having established in the other two that *inv* is indeed an invariant, we can continue our proof for the rest of the program.

This rule has two issues: First, we lose information present in Γ, Δ and $\mathcal{U}$. In order to preserve that information, it has to become part of our invariant. Second, and more pressing, this invariant *inv* needs to be specified somehow. Since we want to achieve *automatic* program verification, requesting this formula from a user is not the solution, but is the current approach taken by theorem provers, since automatic derivation of a suitable invariant is problematic.

In the original work in Bubel et al. (2009) *loop abstraction* is used to overcome both the problem of automatic derivation as well as the preservation of information in Γ, Δ and $\mathcal{U}$. Again we adopt this approach to enable loop abstraction with objects as well.

2.2. Automatic Derived Invariants by Abstraction

As described briefly in Section 1.3, static program analysis benefits in multiple ways from lifting the analysis from a concrete to an abstract domain. As mentioned in Section 2.1.3 we use abstraction in this thesis mainly for the purpose of automatic generation of while loop invariants. The main idea underlying this section is to repeatedly symbolically execute the body of a while loop until a fixed point is found.

In the remainder of this section we first introduce an abstract domain for both integers and objects which we use later on. We then explain how we can automatically derive an update that represents a superset of all states that might be reached by a loop. It is shown how this update is related to loop invariants and we adapt the loopInvariant rule accordingly.

2.2.1. Abstract Domains

We use the following definition for abstract domains and their logical representation, based on the one from Bubel et al. (2009):

Definition 14 (Abstract Domains)**.** *An* abstract domain *is a finite lattice $\mathcal{A}$. Let $\sqsubseteq$ denote the partial order and $\sqcup$ the join operator. $\mathcal{A}$ is an abstract domain for the concrete domain* D *if there is an* abstraction function $\alpha : 2^{\mathsf{D}} \to \mathcal{A}$ *and a* concretization function $\gamma : \mathcal{A} \to 2^{\mathsf{D}}$ *such that*

1. *α and γ are monotone with respect to $\subseteq$ and $\sqsubseteq$, i.e. if $x \subseteq y$ then $\alpha(x) \sqsubseteq \alpha(y)$ and if $a \sqsubseteq b$ then $\gamma(a) \subseteq \gamma(b)$.*
2. *For each $a \in \mathcal{A} : a \sqsupseteq \alpha(\gamma(a))$*
3. *For each $c \in 2^{\mathsf{D}} : c \subseteq \gamma(\alpha(c))$*

The first property ensures consistency of the abstraction and concretization function with respect to the abstract lattice, the second that we may lose precision when concretizing (γ)[2] and the third that concrete values are all preserved when performing abstraction. The definition for monotonicity might differ from one familiar to the reader, where ‘if’ is replaced with ‘if and only if’, however that stronger definition is not required here.

We further extend our signature Σ to $\Sigma_{\mathbb{A}}$ to represent and reason about formulas using this abstract lattice within our concrete calculus.

Definition 15 (Logical Representations of Abstract Domains)**.** *We extend the signature $\Sigma = (\mathcal{S}, \mathcal{F}, \mathcal{P}, \mathcal{V})$ to the signature $\Sigma_{\mathbb{A}} = (\mathcal{S}, \mathcal{F}', \mathcal{P}', \mathcal{V})$ as follows (where $\mathbb{A}$ denotes the set of all abstract domains, $\mathcal{A}_T$ the abstract domain for sort T):*

- $\mathcal{F}'_r = \mathcal{F}_r \cup \{\gamma_{a,z} : T_1 \times \cdots \times T_n \mapsto T \mid a \in \mathcal{A}_T, z \in \mathbb{Z}, \mathcal{A}_T \in \mathbb{A}, T_i \in \mathcal{S}, n \geq 0\}$
- $\mathcal{P}' = \mathcal{P} \cup \{\chi_a : T \mid a \in \mathcal{A}_T, \mathcal{A}_T \in \mathbb{A}\}$

Moreover, in this context we only allow for interpretations I where

- $\forall \mathcal{A} \in \mathbb{A}, a \in \mathcal{A}, z \in \mathbb{Z}, x_1 \in \mathsf{D}_1, \ldots x_n \in \mathsf{D}_n . I(\gamma_{a,z}(x_1, \ldots x_n)) \in \gamma(a)$
- $\forall \mathcal{A} \in \mathbb{A}, a \in \mathcal{A} . I(\chi_a) = \gamma(a)$

[2] This loss of precision only occurs in the abstract lattice for objects. In the one for the integers we have that for each $a \in \mathcal{A} : a = \alpha(\gamma(a))$.

With this extension, we have for each abstract element a an infinite number of concrete constants functions $\gamma_{a,z}$ which are used to represent abstract values in logical formulas. That is, the functions $\gamma_{a,1}$, and $\gamma_{a,2}(x)$, where $\gamma_{a,1}$ is a 0-ary and $\gamma_{a,2}$ a unary symbol, are both concrete elements belonging to $\gamma(a)$. Except for the fact that we know they are elements of the concretization of the same abstract element a, we have no further a priori knowledge about them. In other words, the uncertainty to which concrete element they are evaluated is treated by underspecification. Consequently, if we have the constants $\gamma_{a,0}$ and $\gamma_{a,1}$ we cannot tell if they are the same concrete element or not. Similar for $\gamma_{a,2}(x)$ and $\gamma_{a,2}(y)$ with $x \neq y$. To successfully reason about these γ-symbols, rules need to be provided for handling them. E.g., in case for the integer sign-language used in Section 1.3, we need rules to determine that $(-) * (+) \rightsquigarrow (-)$, or that $(+)$+$(-) \rightsquigarrow (\pm)$.

As long as we considered only program variables and not fields, the 0-ary gamma symbols (constants) from the original work sufficed. In the presence of field functions and as a result quantified updates, we have to extend the gamma functions to unary functions as well.

In the next two sections the abstract domains for integers and objects as used in this thesis are described. The abstract domain for integers is conform the 'sign language' as described by Cousot and Cousot (1977) and already used in (Bubel et al., 2009).

Abstract Domain for Integers

The abstract domain is constructed as a direct translation of the sign language. The concrete domain D is in this case $\mathbb{Z}$, the abstract lattice using the sign language (or: the sign lattice) is shown in Figure 2.1 (taken from Bubel et al. (2009)). To illustrate the use of this abstract lattice in a logical formula, consider the update $\mathcal{U} = \mathtt{i} := \gamma_{\leq,0} \parallel \mathtt{j} := \gamma_{\leq,1}$. With the necessary rules for the handling of γ-constant symbols provided, we are now able to formulate sequents such as $\Longrightarrow \{\mathcal{U}\}\mathtt{i} < 10$, or $!(\gamma_{\leq,1} < 0) \Longrightarrow \{\mathcal{U}\}\mathtt{j} \doteq 0$.

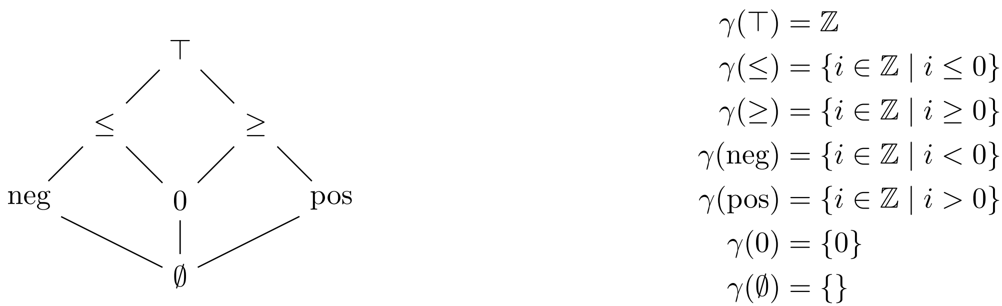

Figure 2.1.: Abstract domain lattice for sign analysis, with its concretization function γ

Abstract Domain for Objects

For the well-understood and studied domain of integers an abstract domain is readily available. Not only the sign language, but other distinctions such as odd versus even, or prime versus non-prime can be used as well. Opposed to this the concrete domain of objects is harder to translate to an abstract domain. In the research area of shape analysis several attempts have been made to capture objects by using shapes as abstraction, but this is not easily translated to

integrate with the original approach that is adopted in this thesis. See Section 2.4 for a further comparison with shape analysis.

Instead we stay in line with the original approach of abstract domains and create such an abstract domain for objects by the notion of *reachability*. That is, the only terms to which we can update an object program variable are via direct assignments to other object program variables (e.g. `t = s`) or using fields (e.g. `t = s.next`). Since we do not (yet) have the option to create new objects in a program, we can conclude that all the objects that are *reachable* during run-time are either pointed to by program variables, or reachable from the program variables using field operations.

Given a state s_0, $M = (\mathsf{D}, I)$ a first-order structure, the abstract domain $\mathcal{A}_{obj} := \wp(\mathcal{A}_{pv} \cup \mathcal{A}_{reach})$, i.e. the powerset of the join of the sets $\mathcal{A}_{pv}$ and $\mathcal{A}_{reach}$, for the objects of one sort is defined as:

- $\mathcal{A}_{pv} = \{\mathtt{x}^0 \mid \mathtt{x} \in \mathcal{F}_{n,0}\}$, with `x` a program variable for an object of the current sort, with $\gamma(\{\mathtt{x}^0\}) = \{val_{M,s_0,\beta}(\mathtt{x})\}$, i.e. the subset that is a member of the powerset $\mathcal{A}_{obj}$ containing only $\mathtt{x}^0$, represents the single object indicated by program variable `x` in the initial state before execution of the program.

- $\mathcal{A}_{reach} = \{\mathtt{x}^+_{b_1,\ldots,b_n} \mid \mathtt{x} \in \mathcal{F}_{n,0} \text{ and } \forall k; b_k \in \mathcal{F}_{n,1} \text{ and } path(\{b_1,\ldots,b_n\},T,S)\}$, where

  $$\begin{aligned} path(\{b_1,\ldots,b_n\},T,S) \;&=\; \exists k_0,\ldots,k_m \in \{1,\ldots n\}; \exists r_1,\ldots r_{m-1} \in \mathcal{S}; \\ &\quad b_{k_0} : T \to r_0 \text{ and } b_{k_m} : r_{m-1} \to S \text{ and } \forall_{1 \le i < n}\, b_i : r_{i-1} \to r_i \end{aligned}$$

  indicates that there must be a path from sort T to sort S, S being the sort for which this abstract lattice is defined. In this we can include fields that do not have the type $S \to S$ and we ensure that no abstract elements are generated where none of the fields is applicable to the sort of the lattice or none of the fields has the corresponding type of the lattice.

Since $\mathcal{A}_{obj}$ is a powerset, we can easily create an abstract lattice from this abstract domain, by taking $\subseteq$ as $\sqsubseteq$ and $\cup$ as $\sqcup$.

Example 11:

With $\mathcal{A}_{obj}$ being defined as the powerset of the union of the two sets $\mathcal{A}_{pv}$ and $\mathcal{A}_{reach}$ we obtain for the example setting $\mathcal{F}_{n,0} = \{\mathtt{x}, \mathtt{y}\}$, with both program variables being of the same type T and only one field $\mathcal{F}_{n,1} = \{\mathtt{next} : T \to T\}$, for these two sets:

- $\mathcal{A}_{pv} = \{\mathtt{x}^0, \mathtt{y}^0\}$
- $\mathcal{A}_{reach} = \{\mathtt{x}^+_{\mathtt{next}}, \mathtt{y}^+_{\mathtt{next}}\}$

The corresponding abstract lattice of $\mathcal{A}_{obj}$ is shown in Figure 2.2.

Further the concretization of abstract elements $a \in \mathcal{A}_{obj}$ is given as $\gamma(a) = \{x \mid x \in_a a \text{ and } sort(x) \doteq sort(a)\}$ where $\in_a$:

$$\begin{aligned} x \in_a a_1 \cup a_2 \quad &\text{iff} \quad (x \in_a a_1 \text{ or } x \in_a a_2) \text{ and } (a_1 \neq \emptyset \text{ or } a_2 \neq \emptyset) \\ x \in_a \{\mathtt{y}^0\} \quad &\text{iff} \quad x = val_{M,s_0,\beta}(\mathtt{y}) \\ x \in_a \{\mathtt{y}^+_{b_1,\ldots,b_n}\} \quad &\text{iff} \quad \exists i \exists z \;.\; \big(val_{M,s_0,\beta_0}(b_i)(z) = x \text{ and } (z = val_{M,s_0,\beta}(\mathtt{y}) \text{ or } z \in_a \{\mathtt{y}^+_{b_1,\ldots,b_n}\})\big) \end{aligned}$$

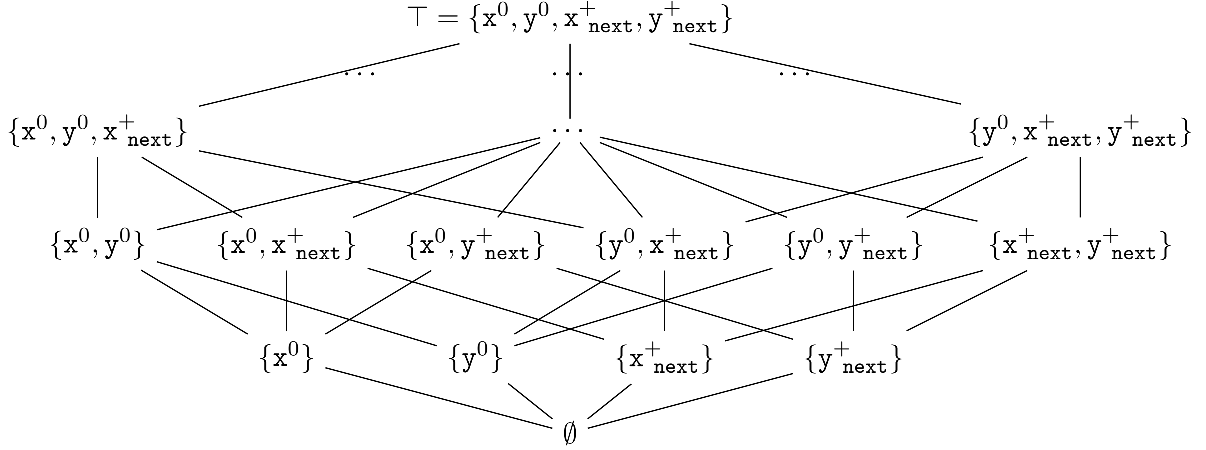

Figure 2.2.: Example abstract domain lattice for objects

The extra condition $sort(x) \doteq sort(a)$ is needed since the fields $b_1, \ldots, b_n$ might be 'intermediate' fields. For example, we could have the element $a = \mathtt{a}^+_{\mathtt{nextA},\mathtt{nextB}}$ with $\mathtt{a} : \mathtt{A}$, $\mathtt{nextA} : \mathtt{B} \mapsto \mathtt{A}$ and $\mathtt{nextB} : \mathtt{A} \mapsto \mathtt{B}$. In this case it should hold that $\mathtt{a.nextB.nextA} \in \gamma(a)$, but also $\mathtt{a.nextB} \notin \gamma(a)$, since a is an abstract element only for objects of sort A.

Using this definition the top element of the abstract lattice now indeed represents all the objects reachable by the program code. To simplify notation we may omit the set notation for singleton sets in this abstract lattice. For example, instead of writing $\gamma_{\{\mathtt{x}^+_{\mathtt{next}}\},0}$ we may write $\gamma_{\mathtt{x}^+_{\mathtt{next}},0}$.

For completion, we show how an abstraction function α can be developed for each abstract object domain, that fulfills the requirement that if $x_1 \subseteq x_2$ then $\alpha(x_1) \sqsubseteq \alpha(x_2)$:

$$\alpha(x) = \bigcup \{a \mid x \subseteq \gamma(a), a \in \mathcal{A}_{pv} \cup \mathcal{A}_{reach}\}$$

The function α returns one abstract element of the lattice $\mathcal{A}_{obj}$. Note that these abstract elements are elements of the powerset $\wp(\mathcal{A}_{pv} \cup \mathcal{A}_{reach})$ and thus are sets themselves. The above definition correctly returns exactly one abstract element on each provided subset $x \subseteq \mathsf{D}$.

EXAMPLE 12:

Let p be the program

```
while(g) {
  t = t.next;
  t.fst = 1;
}
```

Using this extension, we can now formalize properties such as

$$\forall\ \mathtt{Tuple}\ x\ .\ x.\mathtt{fst} \doteq 0 \Longrightarrow [\mathtt{p}]\big(\forall\ \mathtt{Tuple}\ x\ .\ \chi_{\mathtt{t}^+_{\mathtt{next}}}(x)\ \text{->}\ x.\mathtt{fst} \geq 0\big)$$

2.2.2. Update Weakening

Here we introduce an ordering on updates, where one update $\mathcal{U}_2$ might be *weaker* than another update $\mathcal{U}_1$. One can informally think of 'being weaker' as update $\mathcal{U}_2$ representing a less specific state than $\mathcal{U}_1$.

Definition 16 ($\lhd_{P,C}$-relation on updates)**.** *Let P denote a proof, $\mathcal{U}_1$ and $\mathcal{U}_2$ updates, and C a set of formulas (in practice arising from the context Γ and Δ). We call $\mathcal{U}_2$ P, C-*weaker *than $\mathcal{U}_1$, i.e.,*

$$\mathcal{U}_1 \lhd_{P,C} \mathcal{U}_2$$

if for any first-order structure $M = (\mathsf{D}, I)$, state s, and logic variable assignment β, where for all $\psi \in C$ we have $val_{M,s,\beta}(\psi) = tt$, the following holds:

$$val_{M,s,\beta}(\mathcal{U}_1) \in \{val_{M',s,\beta}(\mathcal{U}_2) \mid M' = (\mathsf{D}, I') \text{ and } I \simeq_{P,C} I'\}$$

where $I \simeq_{P,C} I'$ means that I and I' coincide on all function and predicate symbols occurring in P or C.[3] *In case of an empty set of context formulas C, we omit C and write P-weaker and $\lhd_P$ instead.*

EXAMPLE 13:

We show some attempts to weaken the update $\mathcal{U} = \mathtt{i} := 3 \parallel \mathtt{j} := \mathtt{j} + 1$ in the proof P:

$$\mathtt{j} > 0 \Longrightarrow \{\mathcal{U}\}\varphi$$

- $\mathtt{i} := d \parallel \mathtt{j} := e$ where d and e are new constant symbols, is P-weaker than $\mathcal{U}$, since we can choose interpretation $I' \equiv I$ with $I'(d) = s(\mathtt{i})$ and $I'(e) = s(\mathtt{j}) + 1$.
- $\mathtt{i} := \gamma_{\geq,0} \parallel \mathtt{j} := f(1)$ where $\gamma_{\geq,0}, f$ are new function symbols, is P-weaker than $\mathcal{U}$, since we can choose interpretation $I' \equiv I$ with $I'(f)(1) = s(\mathtt{j}) + 1$.
- $\mathtt{i} := \mathtt{j} \parallel \mathtt{j} := \mathtt{j} + 1$ is *not* P-weaker than $\mathcal{U}$, since for any s' with $s'(\mathtt{j}) \neq 3$ the membership requirement from Definition 16 does not hold.
- $\mathtt{i} := \gamma_{\geq,0} \parallel \mathtt{j} := \gamma_{pos,0}$ is *not* P-weaker than $\mathcal{U}$, but $\{\mathtt{j} > 0\}, P$-weaker.

We also introduce a rule that allows us to replace an update $\mathcal{U}$ in a formula by an update $\mathcal{U}'$ that is weaker than $\mathcal{U}$. In practice we only use this update weakening to replace a right-hand side term of an elementary update with a (correct) γ-symbol. The following weakenUpdate rule is adopted from the rule in the original calculus to include the handling of locations (objects) and γ-symbols as n-ary functions (as discussed in Definition 15).

$$\mathsf{weakenUpdate}\ \frac{\begin{array}{l}\Gamma, \{\mathcal{U}\}\overline{\forall}x.(\overline{loc}[x] \doteq \bar{c}(x)) \Longrightarrow \tilde{\exists}\bar{\gamma}.\{\mathcal{U}'\}\overline{\forall}x.(\overline{loc}[x] \doteq \bar{c}(x)), \Delta \\ \Gamma \Longrightarrow \{\mathcal{U}'\}\varphi, \Delta\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}\varphi, \Delta}$$

where

[3]Note that in particular $val_{I',s,\beta}(\psi) = tt$ holds for all $\psi \in C$.

- $\overline{loc}$ is a list of locations $(loc_1, \ldots, loc_n)$ of all locations updated in $\mathcal{U}$ or $\mathcal{U}'$, where $loc_i \neq_\alpha loc_j$, with $i \neq j$. That is, only one of the locations that are equal after α-conversion (note that because of quantified updates there might be a free variable occurring in each location, term or guard) is included in this list.
- $loc[x]$ means that the quantified variable in loc (if any) is substituted with x.
- $\bar{c}$ is a list of fresh rigid functions $(c_1, \ldots, c_n)$ the same length as $\overline{loc}$. If the quantified variable x is not occurring in the location on the left-hand side of the equality this implies that c returns the same value for each input.
- $\bar{\gamma}$ is a list of function symbols $(\gamma_{a_1,z_1}, \ldots, \gamma_{a_m,z_m})$ freshly introduced in $\mathcal{U}'$.
- $\tilde{\exists}$ is a special quantifier that expresses the same intention as the regular $\exists$ quantifier, however with the difference that it operates on second order variables as well, which is not possible in first order logic. Instead this quantifier requires the instantiation of its second-order variable at the moment the rule is applied, preventing the $\tilde{\exists}$ quantifier from occurring in the real logic. We show how this instantiation can be derived automatically during the search for a fixed point in Section 2.2.4, and how this rule needs to be updated consequently.
- The notation $\tilde{\exists}\bar{\gamma}.\psi$ is an abbreviation for $\tilde{\exists}\bar{y}(\forall x\ \chi_{\bar{a}}(\bar{y}(x))\ \&\ \psi[\bar{\gamma}/\bar{y}])$, where $\bar{a} = (a_1, \ldots a_m)$ are the abstract elements from $\bar{\gamma}$ described above, $\bar{y} = (y_1, \ldots y_m)$ a list of fresh logic variables of the same length as $\bar{\gamma}$, ranging over functions.
- The notation $\overline{\forall} x.(\overline{loc}[x] \doteq \bar{c}(x))$ is an abbreviation for $\forall x.(loc_1[x] \doteq c_1(x))\ \&\ \ldots\ \&\ \forall x.(loc_n[x] \doteq c_n(x))$.

In this and the next section we continue using the symbol $\tilde{\exists}$, but it is important to note that this is *not* a second-order quantifier but only a short way for representing our intention, the real rule is introduced in Section 2.2.4.

Example 14:

Consider the sequent

$$\Longrightarrow \quad \{\mathtt{i} := 7\}\varphi$$

that we want to weaken using the update $\mathtt{i} := \gamma_{\geq,0}$. We have to show that there exists a 0-ary function y_0 such that after substituting $\gamma_{\geq,0}$ with y_0, the value of i after the original update is the same as after the weakening one. That is, the first premise of the weakenUpdate rule becomes:

$$\{\mathtt{i} := 7\}(\mathtt{i} \doteq c_0) \quad \Longrightarrow \quad \tilde{\exists} y_0.(\chi_{\geq}(y_0)\ \&\ \{\mathtt{i} := \gamma_{\geq,0}\}[\gamma_{\geq,0}/y_0](\mathtt{i} \doteq c_0))$$

Which we can prove by substituting y_0 with either 7 or c_0.

Now consider a sequent with updates:

$$\Longrightarrow \quad \{\mathtt{t.snd} := 3 \parallel \mathtt{s.snd} := 6\}\varphi$$

That we want to weaken with the update $\forall x.x.\mathtt{snd} := \gamma_{\top,1}(x)$. This time we make use of a unary function γ symbol. The first sequent to prove resulting from application of the weakenUpdate

rule becomes:

$$\{\mathtt{t.snd} := 3 \parallel \mathtt{s.snd} := 6\}\forall x.(x.\mathtt{snd} \doteq c_1(x))$$
$$\Longrightarrow \tilde{\exists} y_1.(\forall x.\chi_\top(y_1(x)) \;\&\; \{\forall x.x.\mathtt{snd} := \gamma_{\top,1}(x)\}[\gamma_{\top,1}/y_1]\forall x.(x.\mathtt{snd} \doteq c_1(x))$$

In this case we can use c_1 as an instantiation of y_1 to successfully prove the sequent.

The problems for an automatic proof procedure are that (1) the "second-order" quantifier $\tilde{\exists}$ is not allowed to appear in a first-order formula, and that (2) these instantiations of y_0 and y_1 needs to be derived automatically.

In Section 2.2.4 the weakenUpdate rule is slightly altered because of $\tilde{\exists}$, however for simplicity we use the current version for the moment. A proof of the soundness for the altered rule is provided in Appendix B.1. Recalling the classical invariant rule loopInvariant (Section 2.1.3):

$$\text{loopInvariant}\; \frac{\begin{array}{ll} \Gamma \Longrightarrow \{\mathcal{U}\}inv, \Delta & \text{Initial} \\ inv, g \Longrightarrow [\mathtt{b}]inv & \text{Invariant} \\ inv, !g \Longrightarrow [\ldots]\varphi & \text{Use case} \end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathtt{while}\ (g)\ \{\mathtt{b}\};\ \ldots]\varphi, \Delta}$$

We now define a variation on this using the weaken-update approach:

invariantUpdate

$$\frac{\begin{array}{ll} \Gamma, \{\mathcal{U}\}\overline{\forall} x.(\overline{loc}[x] \doteq \overline{c}(x))) \Longrightarrow \tilde{\exists}\overline{\gamma}\{\mathcal{U}'\}\ \overline{\forall} x.(\overline{loc}[x] \doteq \overline{c}(x)), \Delta & \text{Initial} \\ \Gamma, \{\mathcal{U}'\}g, \{\mathcal{U}'\}[\mathtt{p}]\overline{\forall} x.(\overline{loc}[x] \doteq \overline{c}(x))) \Longrightarrow \tilde{\exists}\overline{\gamma}\{\mathcal{U}'\}\ \overline{\forall} x.(\overline{loc}[x] \doteq \overline{c}(x)), \Delta & \text{Invariant} \\ \Gamma, \{\mathcal{U}'\}\,!\,g \Longrightarrow \{\mathcal{U}'\}[\ldots]\varphi, \Delta & \text{Use case} \end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathtt{while}\ (g)\ \{\mathtt{p}\};\ \ldots]\varphi, \Delta}$$

where the three premises to prove represent the same intention as those from the loopInvariant rule, as discussed in Section 2.1.3. The first premise is the same as the first premise in the weakenUpdate rule, indicating that the "invariant update" can be used to reach the state that is reached with update $\mathcal{U}$. The second premise shows that $\mathcal{U}'$ is indeed an invariant over the while loop; for any state that is reached from executing the body of the while loop after $\mathcal{U}'$, there exists an interpretation for the γ symbols such that $\mathcal{U}'$ directly reaches that state. Finally the proof tree for φ is continued in the third premise.

Example 15:

Consider the following sequent:

$$\forall\ \mathtt{Tuple}\ x\ .\ x.\mathtt{fst} \doteq 0 \Longrightarrow [\mathtt{while\ (g)\ \{t=t.next;\ t.fst=1\}}](\mathtt{t.fst} \geq 0)$$

As an invariant update we use the following: $\mathcal{U}' = \mathtt{t} := \gamma_{\mathtt{t}^+_{\mathtt{next}},0} \parallel \forall x\ .\ \chi_{\mathtt{t}^+_{\mathtt{next}}}(x) \;{-}{>}\; x.\mathtt{fst} := \gamma_{\geq,1}(x)$. To show that this is an actually invariant, we need to prove the following two premises from the invariantUpdate rule (where substitution on the update $\mathcal{U}'$ has already been applied):

$$\forall x\ .\ x.\mathtt{fst} \doteq 0, (\mathtt{t} \doteq c_1 \;\&\; \forall x\ .\ x.\mathtt{fst} \doteq c_2(x))$$
$$\Longrightarrow \tilde{\exists} y_1, y_2.\forall x.(\chi_{\mathtt{t}^+_{\mathtt{next}}}(y_1) \;\&\; \chi_{\geq}(y_2(x))) \;\&$$
$$\{\mathtt{t} := y_1 \parallel \forall x\ .\ \chi_{\mathtt{t}^+_{\mathtt{next}}}(x) \;{-}{>}\; x.\mathtt{fst} := y_2(x)\}(\mathtt{t} \doteq c_1 \;\&\; \forall x\ .\ x.\mathtt{fst} \doteq c_2(x))$$

$\forall x\ .\ x.\mathtt{fst} \doteq 0, \{\mathcal{U}'\}g,$
$\{\mathcal{U}'\}[\mathtt{p}](\mathtt{t} \doteq c_1\ \&\ \forall x\ .\ x.\mathtt{fst} \doteq c_2(x))$
$\quad \Longrightarrow \tilde{\exists} y_1, y_2.\forall x.(\chi_{\mathtt{t}^+_{\mathtt{next}}}(y_1)\ \&\ \chi_{\geq}(y_2(x)))\ \&$
$\quad\quad \{\mathtt{t} := y_1 \parallel \forall x\ .\ \chi_{\mathtt{t}^+_{\mathtt{next}}}(x)\ \texttt{->}\ x.\mathtt{fst} := y_2(x)\}(\mathtt{t} \doteq c_1\ \&\ \forall x\ .\ x.\mathtt{fst} \doteq c_2(x))$

$\forall x\ .\ x.\mathtt{fst} \doteq 0, \{\mathcal{U}'\}\,!\,g$
$\quad \Longrightarrow \{\mathcal{U}'\}(\mathtt{t.fst} \geq 0)$

Provided that we can handle the "second-order" existential quantifier, instantiation of the quantifiers with c_1 for y_1 and respectively c_2 for y_2 results in logically valid and provable sequents.

Similar to the weakenUpdate rule, the invariantUpdate rule is altered in Section 2.2.4 to replace the "second-order" quantifier $\tilde{\exists}$ with the real intended meaning.

2.2.3. Automated Fixed Point Search

The automatic derivation of fixed points is highly similar to the original one described in Bubel et al. (2009). The main concept is to iterate the execution of the body of a while loop many times and updating a "weakening update" accordingly until a fixed point for this update is reached. Upon encountering a while loop, the loopUnwind rule (Section 2.1.3) is applied once. The resulting sequent is executed up to the point where we re-encounter the while loop, using the calculus rules for symbolic execution. Due to the possible presence of conditional statements in the while loop, this symbolic execution results in one or more branches.

To obtain a fixed point, we collect the updates $\mathcal{U}_1, \ldots \mathcal{U}_m$ that are the result of the symbolic execution of the while loop's body, in each of those branches. We join them together with $\mathcal{U}_0$ in one update $\mathcal{U}'$ that captures all states that can be reached with these updates. We then replace $\mathcal{U}_0$ with $\mathcal{U}'$ and unwind the loop again, until we can conclude that $\mathcal{U}_0$ is a fixed point.

An overview of this process can be found in Figure 2.3, in which the symbol $\dot{\sqcup}$ is used for the join operator on branches, which is defined later in this section.

To detect whether $\mathcal{U}_0$ is a fixed point, we check if $\mathcal{U}_0$ is weaker than $\mathcal{U}'$ (see Definition 16). That is, we try to prove that for all locations $\overline{loc}$ the states reachable from $\mathcal{U}_0$ *subsume* those of $\mathcal{U}'$:

$$\tilde{\forall}\bar{y'}.\tilde{\exists}\bar{y}.(Eq(\bar{y'}, \bar{y})\ \&\ \chi_{\bar{\gamma'}}(\bar{y'})\ \texttt{->}\ \chi_{\bar{\gamma_0}}(\bar{y})\ \&\ \forall x\{\mathcal{U}_0[\bar{\gamma_0}/\bar{y}]\}\overline{loc}[x] \doteq \{\mathcal{U}'[\bar{\gamma'}/\bar{y'}]\}\overline{loc}[x]) \qquad (2.1)$$

where

- $\bar{\gamma_0}, \bar{\gamma'}$ denote sequences of all γ symbols occurring in one of the respective updates $\mathcal{U}_0$ or $\mathcal{U}'$
- $\bar{y}, \bar{y'}$ are duplicate-free sequences of logic variables of same length as $\bar{\gamma}$ resp. $\bar{\gamma'}$
- $\chi_{\bar{\gamma_0}}(\bar{y})$ is short for $\forall x.\chi_{\bar{\gamma_0}}(\bar{y}(x))$, similar for $\chi_{\bar{\gamma'}}(\bar{y'})$

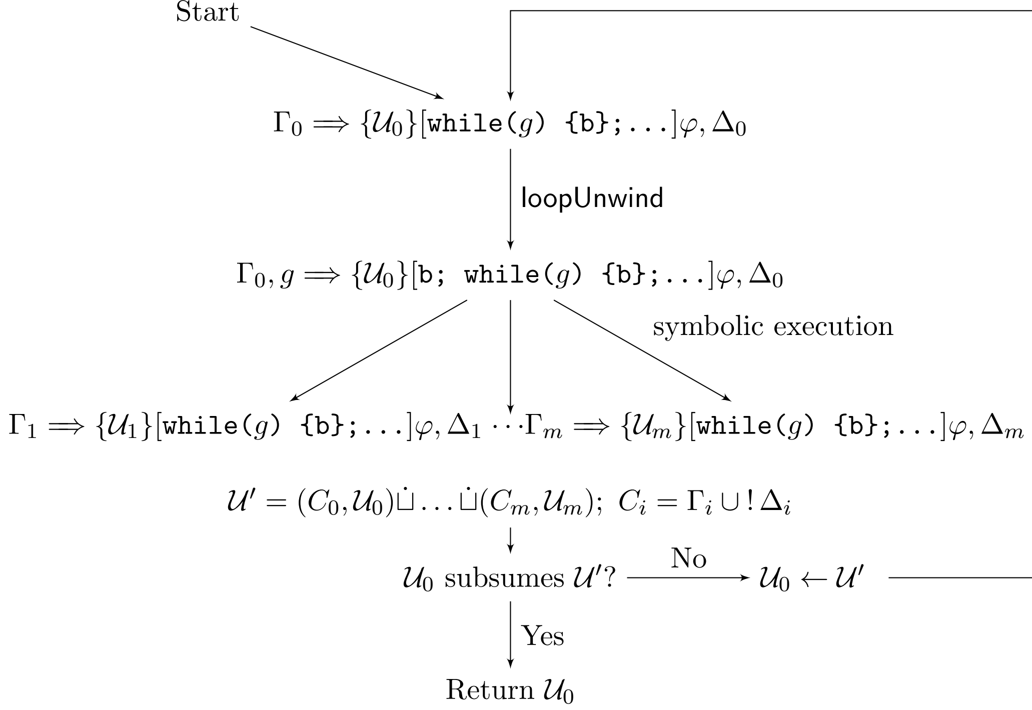

Figure 2.3.: Flowchart of the computation of a loop invariant: $\dot\sqcup$ as the *join* operator

- $\tilde\exists$, $\tilde\forall$, $\overline{x}$ and $\overline{loc}$ are defined as in the weakenUpdate rule. Note that again we here have two short-hand "second-order" quantifiers present ($\tilde\exists$ and $\tilde\forall$) in a first-order formula, which is not allowed.
- $Eq(\bar{y'}, \bar{y}) := \bigwedge_{\substack{y_i \in \bar{y}, y'_j \in \bar{y'},\\ a_i = a_j}} \forall x.(y_i(x) = y'_j(x))$ and $\chi_{\bar\gamma}(\bar{y}) := \bigwedge_{\substack{\gamma_{a_i}, i \in \bar\gamma_0,\\ y_i \in \bar{y}}} \forall x. \chi_{a_i}(y_i(x))$ (analog. $\chi_{\bar{\gamma'}}$)

As long as we have not found this fixed point we continue with another iteration of unwinding the while loop and joining the branches, as shown in Figure 2.3. Due to the way the branches are joined, a fixed pointed is guaranteed to be found as is described later in this section.

Example 16:

As a simple example, consider the following two updates:

$$\mathcal{U}' = \texttt{x} := \gamma_{\geq,0}(\texttt{x})$$
$$\mathcal{U}_0 = \texttt{x} := \gamma_{\top,0}(\texttt{x})$$

We want to show that $\mathcal{U}_0$ subsumes $\mathcal{U}'$. I.e. we need to prove the following instantiation of equation (2.1):

$$\tilde\forall y' \tilde\exists y. \big(Eq(y', y)\ \&\ \chi_{\geq}(y') \rightarrow \chi_{\top}(y)\ \&\ \forall x.(\{\texttt{x} := y(\texttt{x})\}\texttt{x} \doteq \{\texttt{x} := y'(\texttt{x})\}\texttt{x})\big)$$

Skolemization (Fitting, 1990) is a standard technique used in first-order logic to replace a universal quantified variable with a new constant symbol (which is adopted as the forallRight rule in the dynamic logic from KeY (Beckert et al., 2007)). Note that this rule is only available

for first-order logic formula and that therefore the use of this rule on the special symbol $\tilde{\forall}$ is just show the intention. This gives us (after update application):

$$\tilde{\exists}y.\big(Eq(c,y) \mathbin{\&} \chi_{\geq}(c) \rightarrow \chi_{\top}(y) \mathbin{\&} y(\mathtt{x})) \doteq c(\mathtt{x})\big)$$

We can now instantiate the "second-order" existential quantifier on the function y with c:

$$Eq(c,c) \mathbin{\&} \chi_{\geq}(c) \rightarrow \chi_{\top}(c) \mathbin{\&} (c(\mathtt{x})) \doteq c(\mathtt{x})$$

Where we mainly have to prove that for this new, unspecified non-rigid function c the formula

$$\forall x.\big(\chi_{\geq}(c(x)) \rightarrow \chi_{\top}(c(x))\big)$$

holds. Which, due to the nature of the abstract lattice for integers, is trivially the case.

As noted by Niedermann (2011) a computational slightly more expensive but highly simplifying way of detecting a fixed point is to see whether $\mathcal{U}_0 =_z \mathcal{U}'$ (that is, equal except for the z-index on γ-symbols[4]) since that is, after all, what being a fixed point implies: another execution of the loop's body yields no change in the update representing the state change. This is a sound approach and one may notice that it takes at most one iteration more than the detection of a fixed point using the subsumption method described above. It also removes the need of having the special quantifiers $\tilde{\forall}$ and $\tilde{\exists}$ to detect subsumption.

Joining Branches

The join rule introduced in this section is a special rule, in the sense that it is not a regular sequent calculus rule but a 'meta rule' that is able to combine two branches of a proof tree. The following is adopted from Bubel et al. (2009).

Let P denote a proof where we have unwinded and executed the while loop once, obtaining several open branches:

$$\begin{array}{c} \vdots \\ \Gamma_{s_0} \Longrightarrow \{\mathcal{U}_{s_0}\}[\mathtt{while}\ (g)\{\mathtt{b}\}]\varphi,\ \Delta_{s_0} \\ \vdots \\ \Gamma_{s_1} \Longrightarrow \{\mathcal{U}_{s_1}\}[\mathtt{while}\ (g)\{\mathtt{b}\}]\varphi,\ \Delta_{s_1} \quad \ldots \quad \Gamma_{s_m} \Longrightarrow \{\mathcal{U}_{s_m}\}[\mathtt{while}\ (g)\{\mathtt{b}\}]\varphi,\ \Delta_{s_m} \end{array}$$

Applying the join rule closes all except one of these open branches. The open branch that is left is extended by adding the sequent

$$\bigvee_{i=s_0}^{s_m} (\Gamma_{s_i} \mathbin{\&} !\,\Delta_{s_i}) \Longrightarrow \{(C_{s_0},\mathcal{U}_{s_0}) \dot{\sqcup} \ldots \dot{\sqcup} (C_{s_m},\mathcal{U}_{s_m})\}[\mathtt{while}\ (g)\ \{\mathtt{b}\}]\varphi$$

as a new leaf with

- formula set $C_{s_i} := \Gamma_{s_i} \cup !\,\Delta_{s_i}$ and
- $(C_1,\mathcal{U}_1) \dot{\sqcup} (C_2,\mathcal{U}_2)$ is an update join operation as defined below.

[4] That is, one z-index may be mapped to one other z-index, but not to multiple, i.e. update $\mathtt{x} := \gamma_{a,1} \parallel \mathtt{y} := \gamma_{a,1}$ is equal to update $\mathtt{x} := \gamma_{a,2} \parallel \mathtt{y} := \gamma_{a,2}$ in this context, but not equal to $\mathtt{x} := \gamma_{a,2} \parallel \mathtt{y} := \gamma_{a,3}$.

Definition 17 (Update Join $\cdot \dot\sqcup \cdot$)**.** *The* update join operation *has the signature*

$$\dot\sqcup : (\wp(For) \times Updates) \times (\wp(For) \times Updates) \to Updates$$

where $\wp(For)$ denotes the power set of formulas and is defined by the following property:

Let $\mathcal{U}_1$ and $\mathcal{U}_2$ denote arbitrary updates occurring in a proof P and let C_1, C_2 be formula sets representing constraints on the update values. Then an update $(C_1,\mathcal{U}_1) \dot\sqcup (C_2,\mathcal{U}_2)$ must be $(P, C_{1/2})$-weaker than $\mathcal{U}_1$ resp. $\mathcal{U}_2$, i.e.

$$\mathcal{U}_i \lhd_{P,C_i} (C_1,\mathcal{U}_1) \dot\sqcup (C_2,\mathcal{U}_2), \; i = 1, 2 \; .$$

Soundness of the join rule is a great convenience, but since it is only used for the automatic derivation of weakened updates, the correctness of the real application of the derived weakened update is ensured by the soundness of the invariantUpdate rule.

An Additional Rewrite Step

Up to this point the join-procedure is the same as it was in the original version, however with the inclusion of objects some alternation is needed in the following steps. Consider the while loop:

```
while (g) {
  x = x.next;
  x.car = x.car + i;
  i = i - 1;
}
```

It may be clear that we want to end up with some kind of update like $\{\mathtt{i} := \gamma_{\top,0} \,\|\, \mathtt{x} := \gamma_{\mathtt{x}^+_{\mathtt{next}},0} \,\|\, \forall \chi_{\mathtt{x}^+_{\mathtt{next}}}(x) \text{ -> } x.\mathtt{car} := \gamma_{\top,1}(x)\}$. This is where we need the quantified updates and unary γ-symbols to correctly construct the weakened update. However, as can be seen in the calculus rules for symbolic execution, quantified updates are never introduced by the application of those rules. The introduction of the quantification needs therefore to be performed on the level of this meta-rule so that they can be included in the weakened update. To continue the joining of branches in a straight-forward way for the abstract domains of all sorts, we introduce these quantifications using the following *rewrite step* during the symbolic execution of the body of the while loop. That is, we only perform this rewrite step during the fixed point search. After the symbolic execution of an assignment to a field of an abstract object variable (e.g. $\{\ldots \,\|\, loc := \gamma_{a,z} \,\|\, \ldots\}[loc.b_1 \ldots b_n \texttt{ = } t\texttt{;} \;\; \ldots]\varphi$), we end up, after parallelization, with a sequent of the form:

$$\Gamma \Longrightarrow \{\ldots \,\|\, loc := \gamma_{a,z} \,\|\, \ldots \,\|\, \gamma_{a,z}.b_1 \ldots b_n := t\}[\mathtt{p}]\varphi, \Delta$$

We now rewrite this right-most elementary update to a quantified update of the form $\forall x.\chi_a(x) \text{ -> } x.b_1 \ldots b_n := t'$ instead, where t' is a term describing all possible values the updated location can have after this update. We need to take into account that this update now also affects other objects in $\gamma(a)$ that are different from the specific $\gamma_{a,z}$. In order to create a correct term t' we make use of the *update join operation* $\dot\sqcup$ which is defined later on. We collapse the update $\ldots \,\|\, loc := \gamma_{a,z} \,\|\, \ldots$ just before the symbolic execution of the assignment to $\mathcal{U}$. Our rewrite rule is as follows:

$$\Gamma \Longrightarrow \{\mathcal{U} \,\|\, \gamma_{a,z}.b_1 \ldots b_n := t\}\varphi, \Delta \rightsquigarrow$$
$$\Gamma \Longrightarrow \{\mathcal{U} \,\|\, \forall x.\chi_a(x) \text{ -> } x.b_1 \ldots b_n := \{\mathcal{U}'\}x.b_1 \ldots b_n\}\varphi, \Delta$$

where $\mathcal{U}' = (\phi,\mathcal{U})\dot{\sqcup}(\phi,\mathcal{U} \parallel \forall x.\chi_a(x) \text{ -> } x.b_1 \ldots b_n := t)$ in which $\phi = \Gamma \cup !\,\Delta$. Since $\mathcal{U}'$ captures both the update $\mathcal{U}$ in which all locations are unaffected by the assignment, and $\mathcal{U} \parallel \forall x.\chi_a(x) \text{ -> } x.b_1 \ldots b_n := t$ in which all possible locations *are* affected, the value of $x.b_1 \ldots b_n$ after this update $\mathcal{U}'$ thus captures both the possibilities that a location was and was not affected by this update, which is what we aimed to achieve.

EXAMPLE 17:

Given the following sequent:

$$\Gamma \Longrightarrow \{\forall x.\chi_a(x) \text{ -> } x.\texttt{fst} := 0 \parallel \texttt{y} := \gamma_{a,0}\}[\texttt{y.fst = 1; ...}]\varphi, \Delta$$

After executing the assignment rule and performing some update rewriting steps, we obtain:

$$\Gamma \Longrightarrow \{\forall x.\chi_a(x) \text{ -> } x.\texttt{fst} := 0 \parallel \texttt{y} := \gamma_{a,0} \parallel \gamma_{a,0}.\texttt{fst} := 1\}[\ldots]\varphi, \Delta$$

This is the point where, during symbolic execution of the while loop's body in the search for a fixed point, the rewrite rule step is applied. The rewrite rule provides the following two input pairs to the join operator $\dot{\sqcup}$ described later in this section:

$$(\Gamma \cup !\,\Delta \quad , \quad \forall x.\chi_a(x) \text{ -> } x.\texttt{fst} := 0 \parallel \texttt{y} := \gamma_{a,0})$$
$$(\Gamma \cup !\,\Delta \quad , \quad \forall x.\chi_a(x) \text{ -> } x.\texttt{fst} := 0 \parallel \texttt{y} := \gamma_{a,0} \parallel \forall x.\chi_a(x) \text{ -> } x.\texttt{fst} := 1)$$

The join operator returns the update $\mathcal{U}' = \texttt{y} := \gamma_{a,0} \parallel \forall x.\chi_a(x) \text{ -> } x.\texttt{fst} := \gamma_{\geq,0}(x)$. Thus applying our rewrite step we obtain as a resulting sequent:

$$\Gamma \Longrightarrow \{\forall x.\chi_a(x) \text{ -> } x.\texttt{fst} := 0 \parallel \texttt{y} := \gamma_{a,0} \parallel \forall x.\chi_a(x) \text{ -> } x.\texttt{fst} := \{\mathcal{U}'\}x.\texttt{fst}\}[\ldots]\varphi, \Delta$$

And after application of the $\mathcal{U}'$ we obtain:

$$\Gamma \Longrightarrow \{\forall x.\chi_a(x) \text{ -> } x.\texttt{fst} := 0 \parallel \texttt{y} := \gamma_{a,0} \parallel \forall x.\chi_a(x) \text{ -> } x.\texttt{fst} := \gamma_{\geq,0}\}[\ldots]\varphi, \Delta$$

It should be noted that we do not prove the soundness of either the rewrite rule described above, or the join operator ($\dot{\sqcup}$) described below. Therefore, these are not to be used in the construction of a proof but only during the search for a fixed point. The resulting invariant update can then be used in the real proof using the invariantUpdate rule which guarantees soundness (as proven in Appendix B.2).

Defining the Join Operator ($\dot{\sqcup}$)

Given a thus modified update/constraints pair $(\mathcal{U}_1, C_1), (\mathcal{U}_2, C_2)$, we join them ($\dot{\sqcup}$) to the update $\mathcal{U}'$ as follows. Let loc be a location occurring on the left-hand side of $\mathcal{U}_1$ or $\mathcal{U}_2$. For each unique loc after α-conversion we add by parallel composition the elementary update $\forall x.(\varphi_1(x) \mid \ldots \mid \varphi_n(x)) \text{ -> } loc[x] := t'$ where φ_i are the guards associated with all elementary updates to loc in $\mathcal{U}_1$ or $\mathcal{U}_2$ and t' is a term constructed as follows.

1. Try to prove

$$\Longrightarrow \exists y.\forall z.((C_1 \text{ -> } (\{\mathcal{U}_1\}loc[z]) \doteq y) \;\&\; (C_2 \text{ -> } (\{\mathcal{U}_2\}loc[z]) \doteq y))$$

when successful take $t' = t$ where t is the term occurring on the right-hand side in the right-most elementary update to loc in either of the two updates. If the proof attempt fails (e.g. because of a timeout), continue with the next step.

2. For each pair $(C_i, \mathcal{U}_i)$, $i = 1, 2$, start with the smallest abstract domain element a and try to prove

$$C_i \Longrightarrow \forall z.\chi_a(\{\mathcal{U}_i\}loc[z])$$

using every element a in the abstract lattice until an a has been found for which this sequent holds. Note that we always find an a for which this holds, if not any other element then with $a = \top$ (thus the last abstract element that is tried).

Having obtained the abstract domain elements a_1, a_2 for each respective pair, we can compute $a_1 \sqcup a_2$ (or at least an upper bound). We then take $t' = \gamma_{a_1 \sqcup a_2}(x)$, where x is the quantified variable.

EXAMPLE 18:

Given program variables `t`, `i` and the constraint/update pairs

$$(\mathtt{i} > 0, \forall x.\chi_{\mathtt{t}^+_{\mathtt{next}}}(x) \text{ -> } x.\mathtt{fst} := 0)$$
$$(\mathtt{i} \geq 0, \forall x.\chi_{\mathtt{t}^+_{\mathtt{next}}}(x) \text{ -> } x.\mathtt{fst} := \mathtt{i} \,\|\, \mathtt{i} := \mathtt{i} - 1)$$

We compute the joined update for this pair as follows. Starting with location $x.\mathtt{fst}$, we generate the update $\forall x.(\chi_{\mathtt{t}^+_{\mathtt{next}}}(x) \mid \chi_{\mathtt{t}^+_{\mathtt{next}}}(x)) \text{ -> } x.\mathtt{fst} := t'$. The first attempt to find if this quantified location is evaluated to the same value under both updates obviously fails, for example if `i` is 1.

We then enter the abstraction phase, and find that the first abstract elements in the integer lattice for which the proof obligation in step 2 can be proven are 0 and $\geq$ for each pair:

$$\mathtt{i} > 0 \Longrightarrow \forall z.\chi_0(\{\forall x.\chi_{\mathtt{t}^+_{\mathtt{next}}}(x) \text{ -> } x.\mathtt{fst} := 0\}z.\mathtt{fst}) \text{ and}$$
$$\mathtt{i} \geq 0 \Longrightarrow \forall z.\chi_{\geq}(\{\forall x.\chi_{\mathtt{t}^+_{\mathtt{next}}}(x) \text{ -> } x.\mathtt{fst} := \mathtt{i} \,\|\, \mathtt{i} := \mathtt{i} - 1\}z.\mathtt{fst})$$

The join for these elements gives us $(0 \sqcup \geq) = \geq$. The first sub-update of $\mathcal{U}'$ becomes $\forall x.(\chi_{\mathtt{t}^+_{\mathtt{next}}}(x) \mid \chi_{\mathtt{t}^+_{\mathtt{next}}}(x)) \text{ -> } x.\mathtt{fst} := \gamma_{\geq,0}(x)$. A similar computation for `i` gives us the complete update:

$$\forall x.(\chi_{\mathtt{t}^+_{\mathtt{next}}}(x) \mid \chi_{\mathtt{t}^+_{\mathtt{next}}}(x)) \text{ -> } x.\mathtt{fst} := \gamma_{\geq,0}(x) \,\|\, \forall x.\,\mathrm{true} \text{ -> } \mathtt{i} := \gamma_{\geq,1}(x)$$

2.2.4. Replacing the Special Quantifier $\tilde{\exists}$

In Section 2.2.2 the following definition of the invariantUpdate is specified:

invariantUpdate

$$\frac{\begin{array}{ll} \Gamma, \{\mathcal{U}\}\forall x.(\overline{loc}[x] \doteq \overline{c}(x))) \Longrightarrow \tilde{\exists}\bar{\gamma}\{\mathcal{U}'\}\ \forall x.(\overline{loc}[x] \doteq \overline{c}(x)), \Delta & \text{Initial} \\ \Gamma, \{\mathcal{U}'\}g, \{\mathcal{U}'\}[\mathtt{p}]\forall x.(\overline{loc}[x] \doteq \overline{c}(x))) \Longrightarrow \tilde{\exists}\bar{\gamma}\{\mathcal{U}'\}\ \forall x.(\overline{loc}[x] \doteq \overline{c}(x)), \Delta & \text{Invariant} \\ \Gamma, \{\mathcal{U}'\}\,!\,g \Longrightarrow \{\mathcal{U}'\}[\ldots]\varphi, \Delta & \text{Use case} \end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathtt{while\ (}g\mathtt{)\ \{p\};\ }\ldots]\varphi, \Delta}$$

In this rule the notation $\tilde{\exists}\bar{\gamma}.\psi$ is an abbreviation for $\tilde{\exists}\bar{y}(\forall x\ \chi_a(\bar{y}(x))\ \&\ \psi[\bar{\gamma}/\bar{y}])$, where $\bar{y} = (y_1, \dots y_m)$ a list of function variables of the same length as $\bar{\gamma}$. Since γ symbols may be unary functions, the special quantifier $\tilde{\exists}$ thus quantifies over second-order variables in a first order formula, which is not possible. In the remainder of this section we assume all logic variables quantified over by $\tilde{\exists}$ to be second-order. To show this in a more clear setting, consider the following instantiation of the first premise:

$$\Gamma, \{\texttt{t.fst} := 3\}\forall x.(x.\texttt{fst} \doteq c(x)) \Longrightarrow \tilde{\exists}\bar{\gamma}.\{\forall x.x.\texttt{car} := \gamma_{\geq,0}(x)\}\varphi, \Delta$$

Since $\tilde{\exists}$ appears as a "second-order" quantifier, this is an illegal formula in our logic. We therefore remove the higher-order quantification step over y and assume that an instantiation of y is given, we then obtain:

$$\Gamma, \{\texttt{t.fst} := 3\}\forall x.(x.\texttt{fst} \doteq c(x)) \Longrightarrow \forall x.\chi_{\geq}(y(x))\ \&\ \{\forall x.x.\texttt{car} := y(x)\}\varphi, \Delta$$

No special "second-order" quantifier is present now. Recalling that the goal is to create a fully automatic proof search and that we know that this proof search always applies the removal of the "second-order" quantifier directly after the application of the invariantUpdate (or weakenUpdate) rule, one could consider what happens if we combine these two steps into one. In that case the need of a "second-order" quantifier becomes obsolete. We can combine the two steps into one by placing a "definition" of y in the antecedent:

$$\Gamma, \{\texttt{t.fst} := 3\}\forall x.(x.\texttt{fst} \doteq c(x)), \forall x.(y(x) \doteq r[x]) \Longrightarrow \forall x.\chi_{\geq}(y(x))\ \&\ \{\forall x.x.\texttt{car} := y(x)\}\varphi, \Delta$$

Here r is a regular term with at most one free variable in it, substituted with x. In the next section a method is described to automatically derive a term r that indeed captures the definition of a correct replacement function y for γ. In this example we could of course simply take $r = c(x)$ or $r = 3$. To capture this concept of applying these two steps in one, we adapt the invariantUpdate rule to:

invariantUpdate

$$\frac{\begin{array}{l}\Gamma, \{\mathcal{U}\}\overline{\forall} x.(\overline{loc}[x] \doteq \bar{c}(x)), \Xi \Longrightarrow \forall x.\chi_{\bar{a}}(\bar{y}(x))\ \&\ \{\mathcal{U}'\}[\bar{\gamma}/\bar{y}]\overline{\forall} x.(\overline{loc}[x] \doteq \bar{c}(x)), \Delta \\ \Gamma, \{\mathcal{U}'\}g, \{\mathcal{U}'\}[\mathrm{p}]\overline{\forall} x.(\overline{loc}[x] \doteq \bar{c}(x)), \Xi \Longrightarrow \overline{\forall} x.\chi_{\bar{a}}(\bar{y}(x))\ \&\ \{\mathcal{U}'\}[\bar{\gamma}/\bar{y}]\overline{\forall} x.(\overline{loc}[x] \doteq \bar{c}(x)), \Delta \\ \Gamma, \{\mathcal{U}'\}\,!\,g \Longrightarrow \{\mathcal{U}'\}[\dots]\varphi, \Delta\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\texttt{while (}g\texttt{) \{p\}; ...}]\varphi, \Delta}$$

And similar the weakenUpdate rule to:

weakenUpdate

$$\frac{\begin{array}{l}\Gamma, \{\mathcal{U}\}\overline{\forall} x.(\overline{loc}[x] \doteq \bar{c}(x)), \Xi \Longrightarrow \overline{\forall} x.\chi_{\bar{a}}(\bar{y}(x))\ \&\ \{\mathcal{U}'\}[\bar{\gamma}/\bar{y}]\overline{\forall} x.(\overline{loc}[x] \doteq \bar{c}(x)), \Delta \\ \Gamma \Longrightarrow \{\mathcal{U}'\}\varphi, \Delta\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}\varphi, \Delta}$$

where notation is the same as before and:

- $\bar{y}$ a list of fresh logic variables $(y_1, \dots, y_n)$ ranging over functions, the same length, arity and sort as $\bar{\gamma}$ $(\gamma_{a_1,1}, \dots \gamma_{a_n,n})$. For later convenience we assume that all indexes of γ variables are unique. That is, there do not exist two γ variables $\gamma_{a,i}$ and $\gamma_{b,i}$ with $a \neq b$. We could ensure this by rewriting the indices (since they do not affect the meaning of the symbol). As a result we can now state that a function symbol y_i relates to a unique $\gamma_{a,i}$ with the same index.

- Ξ a list of formulas 'defining' $\bar{y}$. Each $\xi_i \in \Xi$ is *required* to be of the form $\forall x.(y_i(x) \doteq r_i[x])$ where r_i is an automatically derived term for y_i (having a free variable x which is quantified over). Function symbols from $\bar{\gamma}$ or $\bar{y}$ are not allowed to appear in these r_i.

- $\overline{\forall} x.\chi_{\bar{a}}(\bar{y}(x))$ a list of formulas ensuring that each y_i is a correct instantiation of each γ_i. It is an abbreviation for the formula $\forall x.\chi_{a_1}(y_1(x))$ & $\ldots$ & $\forall x.\chi_{a_n}(y_n(x))$ where a_i is the abstract element from $\gamma_{a_i,i}$.

Lemma 1. *The* weakenUpdate *rule and* invariantUpdate *rule are sound.*

The proof for this lemma is given in Appendix B.

Automatic Instantiation of Substitute

With these adaptations to the weakenUpdate and invariantUpdate rule the automatic proof search not only has to return the fixed point or invariant update $\mathcal{U}'$ but also the correct substitute for the γ symbols in the first (or first two) premises of the rule. That is, we need to establish an automatic instantiation of r_i for each $\gamma_{a,i}$ as shown in the last definition of both rules. That it is possible to do so results from the fact that we know how the invariant update $\mathcal{U}'$ and the fresh γ symbols it contains were derived.

If a fresh $\gamma_{a,i}$ occurs in an elementary update in $\mathcal{U}'$, we know it appears on the right-hand side and, because of the way $\mathcal{U}'$ has been constructed, we also know that it is the only term on the right-hand side of the elementary update. To see that this is indeed the case one has to look at the definition of the automatic fixed point search and join rule operation as defined in Section 2.2.3. Recall that a general form of an update is:

$$\forall x.\varphi(x) \text{ -> } loc[x] := t[x] \parallel \forall x.\varphi(x) \text{ -> } loc[x] := t[x] \parallel \ldots$$

Now for each fresh $\gamma_{a,i}$ introduced in $\mathcal{U}'$ we need to construct a correct substitute as a term r_i to be used in the premise of either the weakenUpdate or the invariantUpdate rule. For each $\gamma_{a,i}$ we start by selecting from $\mathcal{U}'$ only the elementary updates in which this symbol occurs and combine this as the update $\mathcal{U}_\gamma$. Using the information on how $\mathcal{U}'$ is constructed, we know that $\gamma_{a,i}$ is the only symbol on the right hand side and thus that $\mathcal{U}_\gamma$ is of the form:

$$\mathcal{U}_\gamma = \forall x.\varphi_1(x) \text{ -> } loc_1[x] := \gamma_{a,i}(x) \parallel \forall x.\varphi_2(x) \text{ -> } loc_2[x] := \gamma_{a,i}(x) \parallel \ldots$$

For later steps it is important that we preserve the order of the elementary updates (that is, we have to preserve the 'last-one wins' semantics). The construction of the corresponding r_i that defines the substitute y_i for $\gamma_{a,i}$ - that is, the term to be used in the 'definition' formula-set Ξ, is now a two-step procedure. We first construct a partial term $r_{k,i}$ for each location loc_k modulo the renaming of bound (quantified) variables. Then we combine these partial terms into the final term r_i.

For the construction of $r_{k,i}$, the partial term for location loc_k, we first remove all the updates to other locations from the update $\mathcal{U}_\gamma$. This gives us:

$$\mathcal{U}_k = \forall x.\varphi_1(x) \text{ -> } loc_k := \gamma_{a,i}(x) \parallel \ldots \parallel \forall x.\varphi_n(x) \text{ -> } loc_k := \gamma_{a,i}(x)$$

From the premise(s) in both the weakenUpdate and the invariantUpdate rule we know that after substitution it has to hold that $\forall x.(loc_k[x] \doteq c_k(x))$. To obtain the basis for our term $r_{k,i}$, we take:

$$r_{k,i}[z] = \{\mathcal{U}_k\} loc_k[z]$$

Applying update simplification, this results in

$$r_{k,i}[z] = \mathit{if}(\varphi_n(z))\mathit{then}(\gamma_{a,i}(z))\mathit{else}(\ldots \mathit{if}(\varphi_1(z))\mathit{then}(\gamma_{a,i}(z))\mathit{else}(\mathit{loc}_k[z])\ldots)$$

The 'last-one wins' semantics is preserved by this conditional term. It is clear that the best substitute for $\gamma_{a,i}(z)$ is $c_k(z)$ in order to fulfill the property that $\forall x.(\mathit{loc}_k[x] \doteq c_k(x))$. We construct the term $r_{k,i}$ by performing this substitution on the right-hand term. This gives us:

$$r_{k,i}[z] = \mathit{if}(\varphi_n(z))\mathit{then}(c_k(z))\mathit{else}(\ldots \mathit{if}(\varphi_1(z))\mathit{then}(c_k(z))\mathit{else}(\mathit{loc}_k[z])\ldots)$$

We can now shorten this definition (and note that it is possible to obtain it even faster during implementation) to:

$$r_{k,i}[z] = \mathit{if}(\varphi_n(z) \mid \ldots \mid \varphi_1(z))\mathit{then}(c_k(z))\mathit{else}(\mathit{loc}_k[z])$$

We repeat this procedure for all locations (modulo bound variable renaming), giving several $r_{k',i}$ for the different locations $\mathit{loc}_{k'}$. To construct the final term r_i for $\gamma_{a,i}$ we combine these sub-definitions for each location by substituting one $r_{k,i}$ term in the *else*-branch of another $r_{k',i}$ term (repeating this process pair-wise if there are more than two different locations module bound variable renaming). The term occurring in the final *else*-branch is of no real concern. The y_i-function replacing $\gamma_{a,i}$ as defined by this r_i only occurs in situations where at least one of the conditions in one of the *if*-parts holds.

The only restriction is that the value in this *else*-part has to fall in the correct abstract domain, that is, χ_a should hold for this value where a the abstract element from $\gamma_{a,i}$. We can assure this by taking a random element from $\gamma(a)$ or by creating a new $\gamma_{a,j}$.

Example 19:

Consider that the update $\mathcal{U}_\gamma$ for $\gamma_{a,i}$ is:

$$\mathcal{U}_\gamma = \forall x.\varphi(x) \text{ -> } x.\mathtt{fst} := \gamma_{a,i}(x) \parallel \forall x.\varphi(x) \text{ -> } x.\mathtt{snd} := \gamma_{a,i}(x) \parallel \ldots$$

We first take $\mathit{loc}_k = x.\mathtt{fst}$. From update $\mathcal{U}_\gamma$ we preserve:

$$\mathcal{U}_k = \forall x.\varphi_1(x) \text{ -> } x.\mathtt{fst} := \gamma_{a,i}(x) \parallel \ldots \parallel \forall x.\varphi_n(x) \text{ -> } x.\mathtt{fst} := \gamma_{a,i}(x)$$

To obtain the basis for our term $r_{\mathtt{fst},i}$ we take:

$$r_{\mathtt{fst},i}[z] = \{\mathcal{U}_k\}z.\mathtt{fst}$$

Applying update simplification, we get:

$$r_{\mathtt{fst},i}[z] = \mathit{if}(\varphi_n(z))\mathit{then}(\gamma_{a,i}(z))\mathit{else}(\ldots \mathit{if}(\varphi_1(z))\mathit{then}(\gamma_{a,i}(z))\mathit{else}(z.\mathtt{fst})\ldots)$$

We construct the term $r_{\mathtt{fst},i}$ by performing the substitution of $\gamma_{a,i}$ by $c_{\mathtt{fst}}$ on the right-hand term (given that $c_{\mathtt{fst}}$ is the term to which $x.\mathtt{fst}$ locations are compared). This gives:

$$r_{\mathtt{fst},i}[z] = \mathit{if}(\varphi_n(z))\mathit{then}(c_{\mathtt{fst}}(z))\mathit{else}(\ldots \mathit{if}(\varphi_1(z))\mathit{then}(c_{\mathtt{fst}}(z))\mathit{else}(z.\mathtt{fst})\ldots)$$

Which can be shortened to:

$$r_{\mathtt{fst},i}[z] = \mathit{if}(\varphi_n(z) \mid \ldots \mid \varphi_1(z))\mathit{then}(c_{\mathtt{fst}}(z))\mathit{else}(z.\mathtt{fst})$$

We repeat this for the location $x.\texttt{snd}$, which gives us the following two sub-terms for r_i:

$$\begin{aligned} r_{\texttt{fst},i}[z] &= \mathit{if}(\varphi_n(z) \mid \ldots \mid \varphi_1(z))\mathit{then}(c_{\texttt{fst}}(z))\mathit{else}(z.\texttt{fst}) \\ r_{\texttt{snd},i}[z] &= \mathit{if}(\varphi_p(z) \mid \ldots \mid \varphi_m(z))\mathit{then}(c_{\texttt{snd}}(z))\mathit{else}(z.\texttt{snd}) \end{aligned}$$

We combine these sub-definitions for each location by substituting $r_{\texttt{snd}}$ in $r_{\texttt{fst}}$ giving us:

$$r_i[z] = \mathit{if}(\varphi_n(z) \mid \ldots \mid \varphi_1(z))\mathit{then}(c_{\texttt{fst}}(z))\mathit{else}(\mathit{if}(\varphi_p(z) \mid \ldots \mid \varphi_m(z))\mathit{then}(c_{\texttt{snd}}(z))\mathit{else}(z.\texttt{snd}))$$

We now have constructed the term r_i for $\gamma_{a,i}$ that should match the invariant update found by the fixed point search. When creating the formula ξ_i for Ξ that defines the function y, we ensure that the logic variable z is replaced with the quantified variable x. In the general case, this formula is thus:

$$\xi_i = \forall x.(y_i(x) \doteq r_i[x])$$

EXAMPLE 20:

Given the sequent:

$$\forall x.(x.\texttt{fst} \doteq 0) \Longrightarrow \{\texttt{t} := \texttt{t.next}\}[\texttt{while(}g\texttt{) \{t.fst = 1; t = t.next\}}]\varphi$$

Using the fixed point search as defined in Section 2.2.3, we obtain the invariant update:

$$\mathcal{U}' = \forall x.\chi_{\texttt{t}^+_{\texttt{next}}}(x) \;\texttt{->}\; x.\texttt{fst} := \gamma_{\geq,1}(x) \parallel \texttt{t} := \gamma_{\texttt{t}^+_{\texttt{next}},2}$$

Applying the invariantUpdate rule gives as first premise:

$$\begin{aligned} &\forall x.(x.\texttt{fst} \doteq 0), \{\texttt{t} := \texttt{t.next}\}(\forall x.(x.\texttt{fst} \doteq c_1(x)) \;\&\; \texttt{t} \doteq c_2), \\ &\forall x.(y_1(x) \doteq r_1[x]), \forall x.(y_2(x) \doteq r_2[x]) \\ &\quad \Longrightarrow \forall x.\chi_{\geq}(r_1[x]) \;\&\; \forall x.\chi_{\texttt{t}^+_{\texttt{next}}}(r_2[x]) \;\& \\ &\qquad \{\mathcal{U}'\}[\gamma_{\geq,0}/y_1][\gamma_{\texttt{t}^+_{\texttt{next}},2}/y_2](\forall x.(x.\texttt{fst} \doteq c_1(x)) \;\&\; \texttt{t} \doteq c_2) \end{aligned}$$

Using the method described in this section we find for r_1 and r_2 the following terms:

$$\begin{aligned} r_1[z] &= \mathit{if}(\chi_{\texttt{t}^+_{\texttt{next}}}(z))\mathit{then}(c_1(z))\mathit{else}(0) \\ r_2[z] &= \mathit{if}(\mathrm{true})\mathit{then}(c_2(z))\mathit{else}(\texttt{t.next}) \end{aligned}$$

We thus have formula ξ_1 and ξ_2 in Ξ:

$$\begin{aligned} \xi_1 &= \forall x.\big(y_1(x) \doteq \mathit{if}(\chi_{\texttt{t}^+_{\texttt{next}}}(x))\mathit{then}(c_1(x))\mathit{else}(0)\big) \\ \xi_2 &= \forall x.\big(y_2(x) \doteq \mathit{if}(\mathrm{true})\mathit{then}(c_2(x))\mathit{else}(\texttt{t.next})\big) \end{aligned}$$

With which the first premise can be proven. The same instantiations can be used to prove the second premise as well.

2.3. Examples and Improvements

In Figure 2.4 a number of example programs using lists is shown. The first two programs (a-b) are concerned with the structure of these lists, while the later three (c-e) are more concerned with the data fields of the lists. In this section we describe what we can and cannot prove using the abstract lattice as described for objects. To increase the set of propositions that can be proved with the approach described in Section 2.2, the fixed point search is extended in Section 2.3.4 with an 'iterator shape', using some concepts from *shape analysis*. However due to the difference in nature between shape analysis and our abstraction based approach this extension does not cover the same amount of shapes as is possible in shape analysis (Section 2.4).

In all of the programs in Figure 2.4 we have the assumption that `t` is an acyclic list of tuples and that each of its `fst` fields is set to 1. In the first two programs (a-b) we want to show that `t` remains acyclic. In programs (c-d) we want to show that the `fst` fields remain above the limit of 0. Example (e) is an 'object version' of the example used in the original work on integers and we merely want to show that `sum` holds afterwards a value ≥ 0.

We define Γ such that we cover all of the examples' assumptions. Since we are not able to specify that 'just' `t` is acyclic we add the assumption that the whole heap is acyclic as well[5]:

$$\begin{aligned} acyclic(l) &= \forall x.\forall y.(x \neq y \text{ -> } x.\texttt{next} \neq y.\texttt{next}), \\ &\quad \forall x.(x.\texttt{next} \neq x), \\ &\quad \forall x.(x.\texttt{next} \neq l) \\ \Gamma &= acyclic(\texttt{t}), \\ &\quad \forall x.\chi_{\{\texttt{t}^0, \texttt{t}^+_{\texttt{next}}\}}(x) \text{ -> } x.\texttt{fst} \doteq 1 \end{aligned}$$

The proof obligations for each of the example programs are as follows, where we take `p` to be the corresponding program from Figure 2.4:

(a) $\Gamma \Longrightarrow [\texttt{p}]acyclic(\texttt{t})$

(b) $\Gamma \Longrightarrow [\texttt{p}]acyclic(\texttt{t})$

(c) $\Gamma \Longrightarrow [\texttt{p}]\forall x.\chi_{\{\texttt{t}^+_{\texttt{next}}\}}(x) \text{ -> } x.\texttt{fst} \geq 0$

(d) $\Gamma \Longrightarrow [\texttt{p}]\forall x.\chi_{\{\texttt{t}^+_{\texttt{next}}\}}(x) \text{ -> } x.\texttt{fst} \geq 0$

(e) $\Gamma \Longrightarrow [\texttt{p}]\texttt{sum} \geq 0$

With our acyclicity definition we do allow for infinite lists, which may result in while loops that do not terminate when executed, which could happen in e.g. in example (a). However since we only use the box operator, if the list pointed to by `t` is infinite validity of the sequents above is automatically obtained. In each of the proof obligations the formulas indeed hold after the execution of the program. In the following sections we show for each of the examples how it can be proved in the abstraction framework, or why it is out of scope.

[5]We might be able to define acyclicity of `t` before the code is executed, however not after `t` has been changed in the program.

```
x = t;
while (x != s) {
  x = x.next;
}
e.next = x.next;
x.next = e;
```

(a) Insert

```
r = n;
x = t;
while (x != n) {
  e = x.next;
  x.next = r;
  r = x;
  x = e;
}
t = r;
```

(b) Reverse

```
while (t != null) {
  t.fst = t.fst - 1;
  t = t.next;
}
```

(c) Limits

```
while (t != null) {
  t.fst = t.fst + 1;
  t = t.next;
}
```

(d) Limits revised

```
sum = 0;
while (t != null) {
  sum = sum + t.fst;
  t = t.next;
}
```

(e) Sum

Figure 2.4.: Example programs

2.3.1. List Insertion

In this example we insert a node in an acyclic list and prove that afterwards we still have an acyclic list. Below the program from example (a) is listed again, inserting an element `e` in the list `t` just after the element `s`:

```
x = t;
while (x != s) {
  x = x.next;
}
e.next = x.next;
x.next = e;
```

We want to formulate that in the initial state there are no cycles (recall: we extended the condition of a non-cyclic list to having no cycles anywhere on the heap), thus:

$$\forall a.\forall b.(a \neq b \text{ -> } a\texttt{.next} \neq b\texttt{.next})$$

and to ensure that there is nothing pointing to the head of the list:

$$\forall x.x\texttt{.next} \neq \texttt{t}$$

Which were already present in Γ. Further we specify that `e` is not yet in the list:

$$\forall x.(x\texttt{.next} \neq \texttt{e})\ \&\ \texttt{t} \neq \texttt{e}$$

The property to prove is that after insertion acyclicity still holds, thus:

$$\forall a.\forall b.(a \neq b \text{ -> } a\texttt{.next} \neq b\texttt{.next}), \text{ and } \forall x.x\texttt{.next} \neq \texttt{t}$$

We now execute the fixed point search as described in Section 2.2.3, where $\mathcal{U}_i$ denotes the update after the i-th execution of the loop's body, and $\mathcal{U}'_i$ denotes the result of the i-th join.

$$\begin{aligned}
\mathcal{U}_0 &= \texttt{x} := \texttt{t} \\
\mathcal{U}_1 &= \texttt{x} := \texttt{t.next} \\
\mathcal{U}'_1 &= \texttt{x} := \gamma_{\texttt{t}^+_{\texttt{next}},0} \\
\mathcal{U}_2 &= \texttt{x} := \gamma_{\texttt{t}^+_{\texttt{next}},0}\texttt{.next} \\
\mathcal{U}'_2 &= \texttt{x} := \gamma_{\texttt{t}^+_{\texttt{next}},1}
\end{aligned}$$

We have found a fixed point, because both $\mathcal{U}'_1$ subsumes $\mathcal{U}'_2$, and because the updates are equivalent apart from their z-indexes. We can therefore apply the invariantUpdate rule with update $\mathcal{U}'_1$ as the weakening update. After continuing the symbolic execution of the two assignments following the while loop, we thus obtain the update:

$$\texttt{x} := \gamma_{\texttt{t}^+_{\texttt{next}},0} \parallel \texttt{e.next} := \gamma_{\texttt{t}^+_{\texttt{next}},0}\texttt{.next} \parallel \gamma_{\texttt{t}^+_{\texttt{next}},0}\texttt{.next} := \texttt{e}$$

After some rewriting and calculus rule applications we are left with the following sequent to prove (we ignore the second formula for acyclicity, to show that $\forall x.(x.next \neq \texttt{t})$, but its proof is similar to the one for this sequent).

$$\Gamma, \forall x.(x\texttt{.next} \neq \texttt{e}), \texttt{t} \neq \texttt{e}, a \neq b \Longrightarrow$$
$$\{\texttt{x} := \gamma_{\texttt{t}^+_{\texttt{next}},0} \parallel \texttt{e.next} := \gamma_{\texttt{t}^+_{\texttt{next}},0}\texttt{.next} \parallel \gamma_{\texttt{t}^+_{\texttt{next}},0}\texttt{.next} := \texttt{e}\}(a\texttt{.next} \neq b\texttt{.next})$$

We note beforehand that it cannot be that $\gamma_{\texttt{t}^+_{\texttt{next}},0} \doteq \texttt{e}$ since we have the assumption that e is not yet an element in list t.

We can prove this sequent by performing a case split on the values of a and b. To save space, we omit cases that contradict the assumptions in Γ:

- Case $a \doteq \texttt{e}$ (thus $b \neq \texttt{e}$)
  - Case $b \doteq \gamma_{\texttt{t}^+_{\texttt{next}},0}$
    After update application, we have to prove: $\gamma_{\texttt{t}^+_{\texttt{next}},0}\texttt{.next} \neq \texttt{e}$. Which is simple since we had in Γ that $\forall x.(x\texttt{.next} \neq \texttt{e})$.
  - Case $b \neq \gamma_{\texttt{t}^+_{\texttt{next}},0}$
    We have to prove: $\gamma_{\texttt{t}^+_{\texttt{next}},0}\texttt{.next} \neq b\texttt{.next}$. Note that we have $b \neq \gamma_{\texttt{t}^+_{\texttt{next}},0}$, so we can use the proposition in Γ: $\forall a.\forall b.(a \neq b \texttt{->} a\texttt{.next} \neq b\texttt{.next})$.
- Case $a \doteq \gamma_{\texttt{t}^+_{\texttt{next}},0}$ (thus $b \neq \gamma_{\texttt{t}^+_{\texttt{next}},0}$)
  We have to prove: $\texttt{e} \neq b\texttt{.next}$. For which we again can use $\forall x.(x\texttt{.next} \neq \texttt{e})$.
- Case $a \neq \texttt{e}$ and $a \neq \gamma_{\texttt{t}^+_{\texttt{next}},0}$
  - Case $b \doteq \texttt{e}$
    We get $a\texttt{.next} \neq \gamma_{\texttt{t}^+_{\texttt{next}},0}\texttt{.next}$ which we prove using $\forall a \forall b.(a \neq b \texttt{->} a\texttt{.next} \neq b\texttt{.next})$.
  - Case $b \doteq \gamma_{\texttt{t}^+_{\texttt{next}},0}$
    We get $a\texttt{.next} \neq \texttt{e}$ which we prove using $\forall x.(x\texttt{.next} \neq \texttt{e})$.

– Case $b \neq \mathtt{e}$ and $b \neq \gamma_{\mathtt{t}^+_{\mathtt{next}},0}$
We get $a.\mathtt{next} \neq b.\mathtt{next}$ for which we again can use (using the formula in the antecedent that $a \neq b$): $\forall a \forall b.(a \neq b \texttt{->} a.\mathtt{next} \neq b.\mathtt{next})$.

This shows that using the abstract lattice we are able to prove the preservation of acyclicity during element insertion.

2.3.2. List Reversal

Below the example (b) from Figure 2.4 on reversing a list:

```
r = n;
x = t;
while (x != n) {
  e = x.next;
  x.next = r;
  r = x;
  x = e;
}
t = r;
```

Note that `n` here is used to mark the end of the list (since we never introduced a real **`null`** element). This implies that multiple `next` fields may point to this objects, making it an exception to the acyclicity assumption. However, for this particular example where we start the program with only one list this has no effect.

Again we want to prove that in the resulting reversed list `t` acyclicity still holds. The updates used during the abstraction process are:

$$
\begin{aligned}
\mathcal{U}_0 &= \mathtt{r} := \mathtt{n} \parallel \mathtt{x} := \mathtt{t} \\
\mathcal{U}_1 &= \mathtt{e} := \mathtt{t.next} \parallel \mathtt{x.next} := \mathtt{n} \parallel \mathtt{r} := \mathtt{t} \parallel \mathtt{x} := \mathtt{t.next} \\
\mathcal{U}_1' &= \mathtt{r} := \gamma_{\{\mathtt{n}^0,\mathtt{t}^0\},0} \parallel \mathtt{x} := \gamma_{\{\mathtt{t}^0,\mathtt{t}^+_{\mathtt{next}}\},0} \parallel \mathtt{e} := \gamma_{\{\mathtt{e}^0,\mathtt{t}^+_{\mathtt{next}}\},0} \parallel \mathtt{x.next} := \gamma_{\{\mathtt{n}^0,\mathtt{t}^+_{\mathtt{next}}\},0} \\
\mathcal{U}_2 &= \mathtt{e} := \gamma_{\{\mathtt{t}^0,\mathtt{t}^+_{\mathtt{next}}\},0}.\mathtt{next} \parallel \forall x \,.\, \chi_{\{\mathtt{t}^0,\mathtt{t}^+_{\mathtt{next}}\}}(x) \texttt{->} x.\mathtt{next} := \gamma_{\{\mathtt{n}^0,\mathtt{t}^0\},0}(x) \parallel \\
&\quad\; \mathtt{r} := \gamma_{\{\mathtt{t}^0,\mathtt{t}^+_{\mathtt{next}}\},0} \parallel \mathtt{x} := \gamma_{\{\mathtt{t}^0,\mathtt{t}^+_{\mathtt{next}}\},0}.\mathtt{next} \\
\mathcal{U}_2' &= \mathtt{r} := \gamma_{\{\mathtt{n}^0,\mathtt{t}^0,\mathtt{t}^+_{\mathtt{next}}\},0} \parallel \mathtt{x} := \gamma_{\{\mathtt{t}^0,\mathtt{t}^+_{\mathtt{next}}\},1} \parallel \mathtt{e} := \gamma_{\{\mathtt{n}^0,\mathtt{t}^0,\mathtt{t}^+_{\mathtt{next}}\},1} \parallel \\
&\quad\; \forall x \,.\, \chi_{\{\mathtt{t}^0,\mathtt{t}^+_{\mathtt{next}}\}}(x) \texttt{->} x.\mathtt{next} := \gamma_{\{\mathtt{n}^0,\mathtt{t}^0.\mathtt{t}^+_{\mathtt{next}}\},0}(x)
\end{aligned}
$$

We dropped some elementary updates such as to `x.next` in subsequent updates to save space. After another round of abstraction we would find that $\mathcal{U}_2'$ is a suitable fixed point. Using $\mathcal{U}_2'$ as a fixed point in the weakenUpdate rule prevents us from proving that acyclicity holds after the reversal of this list. The part of $\mathcal{U}_2'$ that causes this disability is the last elementary update:

$$\{\ldots \parallel \forall x \,.\, \chi_{\{\mathtt{t}^0,\mathtt{t}^+_{\mathtt{next}}\}}(x) \texttt{->} x.\mathtt{next} := \gamma_{\{\mathtt{null}^0,\mathtt{t}^0.\mathtt{t}^+_{\mathtt{next}}\},0}(x)\}$$

Because of the abstraction performed with this update, we can neither show that

- no object gets its next field pointing to `t`, since it requires us to prove that:
  $\forall x.\chi_{\{\mathtt{t}^0,\mathtt{t}^+_{\mathtt{next}}\}}(x) \texttt{->} \gamma_{\{\mathtt{null}^0,\mathtt{t}^0.\mathtt{t}^+_{\mathtt{next}}\},0}(x) \neq \mathtt{t}$,
- no two next fields get assigned to the same object, since that requires us to prove:
  $\forall x \forall y.\ x \neq y \texttt{->} \gamma_{\{\mathtt{null}^0,\mathtt{t}^0.\mathtt{t}^+_{\mathtt{next}}\},0}(x) \neq \gamma_{\{\mathtt{null}^0,\mathtt{t}^0.\mathtt{t}^+_{\mathtt{next}}\},0}(y)$.

Both sequents cannot be proven because of our definition of γ functions, i.e. we do not know whether $\gamma_{a,z}(x) \doteq \gamma_{a,z}(y)$ holds or not, given that $x \neq y$ (Section 2.2.1). Therefore, we are not able to prove the acyclicity of `t`.

2.3.3. Field Invariants

The programs (c,d,e) listed in Figure 2.4 and the corresponding properties to prove are all concerned with the fields of the nodes in the list of tuples `t`. In proof obligation (c - Limits) we can clearly see how the fixed point search returns an over-approximated fixed point resulting in the inability to prove the proposition. For convenience we repeat the example here:

```
while (t != null) {
  t.fst = t.fst - 1;
  t = t.next;
}
```

The join procedure for the search of the fixed point is the following (note that we have the assumption in Γ that $\forall x.\chi_{\{t^0,\mathtt{t}^+_{\mathtt{next}}\}}(x) \rightarrow x.\mathtt{fst} \doteq 1$):

$$
\begin{aligned}
\mathcal{U}_0 &= \text{empty} \\
\mathcal{U}_1 &= \mathtt{t.fst} := \mathtt{t.fst} - 1 \parallel \mathtt{t} := \mathtt{t.next} \\
\mathcal{U}_1' &= \mathtt{t.fst} := \gamma_{\geq,0} \parallel \mathtt{t} := \gamma_{\mathtt{t}^+_{\mathtt{next}},0} \\
\mathcal{U}_2 &= \forall x.\chi_{\mathtt{t}^+_{\mathtt{next}}}(x) \rightarrow x.\mathtt{fst} := \gamma_{\top,0} \parallel \mathtt{t} := \gamma_{\mathtt{t}^+_{\mathtt{next}},0}.\mathtt{next} \\
\mathcal{U}_2' &= \mathtt{t.fst} := \gamma_{\geq,0} \parallel \forall x.\chi_{\mathtt{t}^+_{\mathtt{next}}}(x) \rightarrow x.\mathtt{fst} := \gamma_{\top,1} \parallel \mathtt{t} := \gamma_{\mathtt{t}^+_{\mathtt{next}},1}
\end{aligned}
$$

Since `t` gets abstracted to *anything* in the list starting from the object originally referred to by `t` using the field `next`, the fixed point search cannot come to the conclusion that the value of `t.fst` at the beginning of the while loop is always 1 (since according to the weakened updated it might have been updated in the loop-cycle before this one). Therefore for all $x.\chi_{\mathtt{t}^+_{\mathtt{next}}}(x)$ the update to $x.\mathtt{fst}$ gets abstracted to $\gamma_{\top,0}$.

This same problem arises in (d - Limits revised), however since we are increasing the value of the `fst` field instead of decreasing we abstract to $\gamma_{\geq}$. Therefore the checked property concerning the limit holds no matter the number of iterations. In this case however, we could abstract `t` straight to $\gamma_{\top,0}$ and still be able to prove that the `fst` fields all have a value above the limit after termination of the while lop (since in our abstract lattice $\top$ only represents elements reachable by the program code). We could harden the problem a little by adding an extra program variable and adding to Γ : $\mathtt{s.fst} \doteq -10$ so that we *do* need the more precise abstraction that `t` only gets updated to objects reachable from `t`.

The similar issue holds for (e - Sum): without the existence of other program variables not satisfying the conditions that their field `fst` is equal to 1 we could abstract `t` to $\gamma_{\top,0}$ and still be able to prove the proposition that $\mathtt{sum} \geq 0$.

2.3.4. Improvements to the Fixed Point Search

As can be seen in Section 2.3, we are only able to to prove some of the examples from Figure 2.4, and for some of those that we can prove an abstraction to $\top$ is just as sufficient as an abstraction to any other correct abstract element, unless we add some specific side-conditions. On the other hand, the examples that we are not able to prove result from the computed invariant update being weaker than it needs to be.

In this section an improvement extending the abstraction process is suggested, resulting in a more specific fixed point. We try to capture one of the heap shapes that can be captured by shape analysis (Section 2.4), which we call the *iterator shape*. To achieve this, additional information needs to be provided to the join process, other than the abstraction currently provided by the abstract lattice. The concept is that by providing extra information on the *fields* of the objects pointed to by program variables, an improved join procedure can be developed. This is done by introducing an additional *predicate* to be used in the search for a fixed point, adding additional *invariants* (constraints) that can be used in the join procedure and combining these in a new, extended version of the invariantUpdate rule.

A 'border' predicate for quantified updates

A new predicate, denoted by $\zeta_{a,o}$ is added, being quite similar to the predicate χ_a. The difference in semantics is that o represents a 'border' object for the abstract elements from the set $\mathcal{A}_{Reach}$ (Section 2.2.1). That is, the fields of object o are not allowed to be used to show that an object is an element of $\gamma(a)$.

The interpretation of this new predicate is defined as $I(\zeta_{a,o})(x) = x \in_{a,o} a$, where $\in_{a,o}$:

$$
\begin{aligned}
x \in_{a,o} a_1 \cup a_2 \quad & \text{iff} \quad (x \in_{a,o} a_1 \text{ or } x \in_{a,o} a_2) \text{ and } (a_1 \neq \emptyset \text{ or } a_2 \neq \emptyset) \\
x \in_{a,o} \{\mathtt{y}^0\} \quad & \text{iff} \quad x = val_{M,s_0,\beta}(\mathtt{y}) \\
x \in_{a,o} \{\mathtt{y}^+_{b_1,\ldots,b_n}\} \quad & \text{iff} \quad \exists i \exists z \;.\; val_{M,s_0,\beta}(b_i)(z) = x \text{ and} \\
& \qquad (z = val_{M,s_0,\beta}(\mathtt{y}) \text{ or } (z \neq o \text{ and } z \in_{a,o} \{\mathtt{y}^+_{b_1,\ldots,b_n}\}))
\end{aligned}
$$

Example 21:

Consider a situation in which $\mathtt{x.next.next} = \mathtt{y}$ and $\mathtt{x.next} \neq \mathtt{y}$. We then have that $\chi_{\mathtt{x}^+_{\mathtt{next}}}(\mathtt{y})$ and $\zeta_{\mathtt{x}^+_{\mathtt{next}},\mathtt{y.next}}(\mathtt{y})$ hold, but $\zeta_{\mathtt{x}^+_{\mathtt{next}},\mathtt{x.next}}(\mathtt{y})$ does not.

This new predicate is used during the search for a fixed point in a while loop. In Section 2.2.3 a rewrite step is introduced when an update is performed to fields of 'abstracted' objects:

$$
\begin{aligned}
&\Gamma \Longrightarrow \{\mathcal{U} \,\|\, \gamma_{a,z}.b_1 \ldots b_n := t\}[\mathtt{p}]\varphi, \Delta \rightsquigarrow \\
&\Gamma \Longrightarrow \{\mathcal{U} \,\|\, \forall x.\chi_a(x) \text{ -> } x.b_1 \ldots b_n := \{\mathcal{U}'\}x.b_1 \ldots b_n\}[\mathtt{p}]\varphi, \Delta
\end{aligned}
$$

where $\mathcal{U}' = (\phi,\mathcal{U})\dot{\sqcup}(\phi,\mathcal{U} \,\|\, \forall x.\chi_a(x) \text{ -> } x.b_1 \ldots b_n := t)$ in which $\phi = \Gamma \cup \,!\,\Delta$. Instead, we now replace this rewrite step with:

$$
\begin{aligned}
&\Gamma \Longrightarrow \{\mathcal{U} \,\|\, \gamma_{a,z}.b_1 \ldots b_n := t\}[\mathtt{p}]\varphi, \Delta \rightsquigarrow \\
&\Gamma \Longrightarrow \{\mathcal{U} \,\|\, \forall x.\zeta_{a,\gamma_{a,z}}(x) \text{ -> } x.b_1 \ldots b_n := \{\mathcal{U}'\}x.b_1 \ldots b_n\}[\mathtt{p}]\varphi, \Delta
\end{aligned}
$$

where $\mathcal{U}' = (\phi,\mathcal{U})\dot{\sqcup}(\phi,\mathcal{U} \,\|\, \forall x.\zeta_{a,\gamma_{a,z}}(x) \text{ -> } x.b_1 \ldots b_n := t)$ in which $\phi = \Gamma \cup \,!\,\Delta$.

In words, we rewrite the elementary update to a quantified one, as before, but now instead of having the guard $\chi_a(x)$ we have $\zeta_{a,\gamma_{a,z}}(x)$. That means that the rewritten update does not update all objects in $\gamma(a)$, but only those reachable without using fields from object $\gamma_{a,z}$. Note that for all objects o, if $\chi_a(o)$ holds, then so does $\zeta_{a,o}(o)$. The only restriction is that no fields of o are used to show this property. Since $\chi_a(o)$ holds this means that $o \in_a a$ holds, which always

can be shown without using a field of object o itself. This can be easily seen by noting that for each object p whose field is used to show that $o \in_a a$, the property $p \in_a a$ holds as well. Thus if fields of o were used to show that $o \in_a$, then these steps were not needed since this implies it was already shown that $o \in_a$. Hence this new version of this rewrite rule is still sound.

Upgrading the fixed point search

Consider the search for a fixed point in program (d - Limits), as described in Section 2.3.3. With the adoption of the ζ predicates, we still preserve the same first update of symbolic execution of the while loop, and thus the same result of the join procedure:

$$\begin{aligned}\mathcal{U}_0 &= \text{empty}\\ \mathcal{U}_1 &= \texttt{t.fst} := \texttt{t.fst} - 1 \,\|\, \texttt{t} := \texttt{t.next}\\ \mathcal{U}_1' &= \texttt{t.fst} := \gamma_{\geq,0} \,\|\, \texttt{t} := \gamma_{\texttt{t}^+_{\text{next}},0}\end{aligned}$$

If the fixed point search procedure continued in the same way as described in Section 2.2.3 consequently the same fixed point would be found. To improve the final fixed point returned, an additional loop invariant is created for each of the fields of the objects, pointed to by program variables. In that way we preserve additional information. In this example this concerns only the fields of program variable `t`.

In a more general setting, consider the two constrained updates $(C_1,\mathcal{U}_1)$ and $(C_2,\mathcal{U}_2)$ provided as input for the join. For each of the fields b of `x`, where program variable `x` was abstracted to $\gamma_{a,z}$, we now generate an additional invariant as follows.

For each pair $(C_i,\mathcal{U}_i)$, $i = 1, 2$, for any abstract domain element a' starting with the smallest one, we try to prove

$$C_i \Longrightarrow \chi_{a'}(\{\mathcal{U}_i\}\texttt{x}.b)$$

and stop processing a pair as soon as an a' has been found for which the sequent is valid, i.e. a proof has been found (within a given timeout). After termination we are left with two abstract domain elements a'_1, a'_2 for the respective pairs for which we compute $a'_1 \sqcup a'_2$ (or at least an upper bound). Finally we generate the invariant $\chi_{a'}(\texttt{x.f})$ (if an invariant already existed for `x.f` we replace it by this one).

Applying this method on $\mathcal{U}_0$ and $\mathcal{U}_1$ from our example, using the information provided in Γ, i.e. that
$\forall x.\chi_{\{\texttt{t}^0,\texttt{t}^+_{\text{next}}\}}(x) \;\texttt{->}\; x.\texttt{fst} \doteq 1$ (Section 2.3) we obtain an invariant for the field `fst`: $\chi_{\text{pos}}(\texttt{t.fst})$. Since this is an invariant, we also know that $\chi_{\text{pos}}(\{\mathcal{U}_1'\}\texttt{t.fst})$ thus $\chi_{\text{pos}}(\gamma_{\texttt{t}^+_{\text{next}},0}.\texttt{fst})$.

After a second symbolic execution of the while loop we now obtain:

$$\begin{aligned}\mathcal{U}_1' &= \texttt{t.fst} := \gamma_{\geq,0} \,\|\, \texttt{t} := \gamma_{\texttt{t}^+_{\text{next}},0}\\ \mathcal{U}_2 &= \forall x.\zeta_{\texttt{t}^+_{\text{next}},\gamma_{\texttt{t}^+_{\text{next}},0}}(x) \;\texttt{->}\; x.\texttt{fst} := \gamma_{\texttt{t}^+_{\text{next}},0}.\texttt{fst} - 1 \,\|\, \texttt{t} := \gamma_{\texttt{t}^+_{\text{next}},0}.\texttt{next}\end{aligned}$$

Now that we have the 'additional' information on $\gamma_{\texttt{t}^+_{\text{next}},0}.\texttt{fst}$ the join procedure finds for $\gamma_{\texttt{t}^+_{\text{next}},0}.\texttt{fst} - 1$ again the abstract element $\geq$ and not the abstract element $\top$ which was the result of this join in Section 2.3.3:

$$\mathcal{U}_2' = \texttt{t.fst} := \gamma_{\geq,0} \,\|\, \forall x.\zeta_{\texttt{t}^+_{\text{next}},\gamma_{\texttt{t}^+_{\text{next}},0}}(x) \;\texttt{->}\; x.\texttt{fst} := \gamma_{\geq,1}(x) \,\|\, \texttt{t} := \gamma_{\texttt{t}^+_{\text{next}},1}$$

We again construct an extra invariant on the `fst` field of $\texttt{t} = \gamma_{\texttt{t}^+_{\text{next}},1} = \gamma_{\texttt{t}^+_{\text{next}},0}.\texttt{next}$. Note that we here profit from the usage of ζ instead of χ, allowing us to prove that $\gamma_{\texttt{t}^+_{\text{next}},0}.\texttt{next}$.

`fst` is not updated by the universal update in $\mathcal{U}_2'$ and therefore we can again add the invariant $\chi_{pos}(\texttt{t.fst})$.

To detect a fixed point we again use the technique of showing that two updates resulting from subsequent join procedures are equal apart from z-indexes conversion (Section 2.2.3), hereby noting that this also has to apply for the z-index from γ-objects used as 'border object' in ζ predicates.

As a result, we find $\mathcal{U}_2'$ is a fixed point, making the proof search procedure able to construct a proof tree for example (c) in Figure 2.4, with the third premise of the invariantUpdate rule becoming:

$$\Gamma \Longrightarrow \{\mathcal{U}_2'\}(\forall x.\chi_{\texttt{t}^+_{\texttt{next}}}(x) \mathrel{->} x.fst \geq 0)$$

Updating the invariantUpdate rule

In order to allow for the use of these new invariants apart from during the search for a fixed point, the invariantUpdate rule needs to be redefined. It has to incorporate both the standard approach on invariants (using an *inv* formula as seen in Section 2.1.3) and the new approach (using weakening updates as seen in Section 2.2.2). We use the version with the short-hand symbol, "second-order" quantifier $\tilde{\exists}$ here for simplicity and readability, but this rule can be converted to a rule without this symbol using the approach described in Section 2.2.4.

$$\text{invariantUpdate}\ \frac{\begin{array}{l}\Gamma \Longrightarrow \{\mathcal{U}'\}inv, \Delta \\ \Gamma, \{\mathcal{U}'\}g, \{\mathcal{U}'\}inv \Longrightarrow \{\mathcal{U}'\}[\texttt{p}]inv, \Delta \\ \Gamma, \{\mathcal{U}\}\forall x.(\overline{loc} \doteq \overline{c}(x))) \Longrightarrow \tilde{\exists}\overline{\gamma}\{\mathcal{U}'\}\ \forall x.(\overline{loc} \doteq \overline{c}(x)), \Delta \\ \Gamma, \{\mathcal{U}'\}g, \{\mathcal{U}'\}[\texttt{p}]\forall x.(\overline{loc} \doteq \overline{c}(x))) \Longrightarrow \tilde{\exists}\overline{\gamma}\{\mathcal{U}'\}\ \forall x.(\overline{loc} \doteq \overline{c}(x)), \Delta \\ \Gamma, \{\mathcal{U}'\}\,!\,g, \{\mathcal{U}'\}inv \Longrightarrow \{\mathcal{U}'\}[\ldots]\varphi, \Delta\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\texttt{while (}g\texttt{) \{p\}; ...}]\varphi, \Delta}$$

The first premise indicates that the invariants (captured in the formula *inv*) should hold in the base case. The second premise indicates that given that the invariants hold from the previous execution and the condition of the loop is true, the invariants should also hold after execution of the body of this while loop. In the last sequent we can continue the construction of the proof tree with the additional information from the invariant formula.

Resulting extended coverage

With these new additions the proof search is able to prove programs of the form (c - Limits). One could say that the extended approach also supports one 'heap shape' for acyclic lists, which can be seen as a list iterator. In this shape we also preserve invariants on all the fields of this list, including those different from the one used for list iteration (e.g. `next`). That is, we have invariants on the *current* item in the list indicated by the iterator. However, because this 'iterator-shape' is the only shape 'supported' with this extension, we are still unable to prove programs such as (b - Reverse) since the iterator shape is not sufficient here.

It is expected that extension of the current framework with more different shapes turns out to be quite hard, since the underlying concepts of an abstract lattice are essentially different from shape analysis. The abstract lattice is used to generate a so-called *points-to set*, indicating the set of all possible concrete objects to which a program variable may point in a certain state. Opposed to that, in shape analysis the information as to which concrete objects a program variable may point is ignored and the analysis focuses on the *shape* of the heap instead which,

quite as one would expect, is better suited for shape-related properties, such as acyclicity in list reversal. A further comparison to shape analysis can be found in Section 2.4.

2.3.5. Improvements to the Abstract Lattice

The abstract lattice as described in Section 2.2.1 is a rather straight-forward one. Some improvements can be considered to create a more consistent lattice, or to allow for a simplified usage.

1. First of all, the definition of $\in_a$ could be extended to also include the object $val_{M,s_0,\beta}(\mathtt{a})$ in $\mathtt{a}^+_{\mathtt{field}}$ as well. This makes the abstract element $\mathtt{a}^0$ obsolete. This results in a simpler lattice (with only one 'kind' of abstract element) but removing the 'singleton' elements also makes the abstraction less concise. E.g. a variable that could be either in $\gamma(\{\mathtt{a}^0\})$ or $\gamma(\{\mathtt{b}^0\})$ is now joined to $\gamma_{\{\mathtt{a}^+_{\mathtt{field}},\mathtt{b}^+_{\mathtt{field}}\},0}$ instead of $\gamma_{\{\mathtt{a}^0,\mathtt{b}^0\},0}$. The decision whether or not the 'singleton' elements should be kept in the lattice depends on the type of program to which abstraction is applied. Were the abstract lattice to be used mainly for e.g. applications using list-iterators, the 'singleton' elements could be left out.

2. The abstract element $\{\mathtt{a}^+_{\mathtt{x,y}}\}$ captures both the abstract elements $\{\mathtt{a}^+_{\mathtt{x}}\}$ and $\{\mathtt{a}^+_{\mathtt{y}}\}$, that is, $\gamma(\{\mathtt{a}^+_{\mathtt{x}}\}) \subseteq \gamma(\{\mathtt{a}^+_{\mathtt{x,y}}\})$ and $\gamma(\{\mathtt{a}^+_{\mathtt{y}}\}) \subseteq \gamma(\{\mathtt{a}^+_{\mathtt{x,y}}\})$. In the current abstract lattice however, $\{\mathtt{a}^+_{\mathtt{x}}\} \sqcup \{\mathtt{a}^+_{\mathtt{y}}\} = \{\mathtt{a}^+_{\mathtt{x}}, \mathtt{a}^+_{\mathtt{y}}\}$. One could change the definition of the lattice such that this pair is joined to $\{\mathtt{a}^+_{\mathtt{x,y}}\}$ instead, as well would $\{\mathtt{a}^+_{\mathtt{x}}\} \sqcup \{\mathtt{a}^+_{\mathtt{x,y}}\}$ and $\{\mathtt{a}^+_{\mathtt{x,y}}\} \sqcup \{\mathtt{a}^+_{\mathtt{y}}\}$. This simplification of the lattice improves the consistency of joins and results in a better understanding of the results of joining two update-branches.

2.4. Comparison with Shape Analysis

Shape analysis is the general name given to static code analysis techniques in which during analysis a *shape* is updated representing the current state of the program during execution. In this thesis *loop invariants* are used, where the set of concrete objects to which a program variable may point represents the current value of that program variable, a so-called *points-to set*. Instead, in shape analysis a *"shape invariant"* is created, similar as our loop invariants describing that, no matter the number of loop-executions, the same shape of the program is preserved.

Although different approaches to shape analysis exist, they all share the property of representing the possible states of the heap as a *shape graph*. In this graph each memory or heap cell is represented by a node in the graph. Most notably there are one or more so-called *summary nodes*, that represent a set of indistinguishable memory cells; i.e. 'all the other cells on the heap, not already explicitly represented as a node in the graph'. An example of such a shape graph can be found in Figure 2.5.

An interesting approach to shape analysis is taken in (Sagiv et al., 2002) where a 2-valued logic is combined with 3-valued logic. The third value ($\frac{1}{2}$) is used to indicate that the truth value of a formula is unknown, allowing to abstract properties formulated in logic formulas. This is used to give a value to relations with the summary node(s), for example, whether the `next` field of an object in the summary node points to another object in the summary node. A similar uncertainty is in our framework represented by the comparison $\gamma_{\{\mathtt{a}^+_{\mathtt{next}}\},0} \stackrel{?}{=} \gamma_{\{\mathtt{a}^+_{\mathtt{next}}\},1}$. Using the approach by Sagiv et al. makes it simpler to generate different instantiations of their

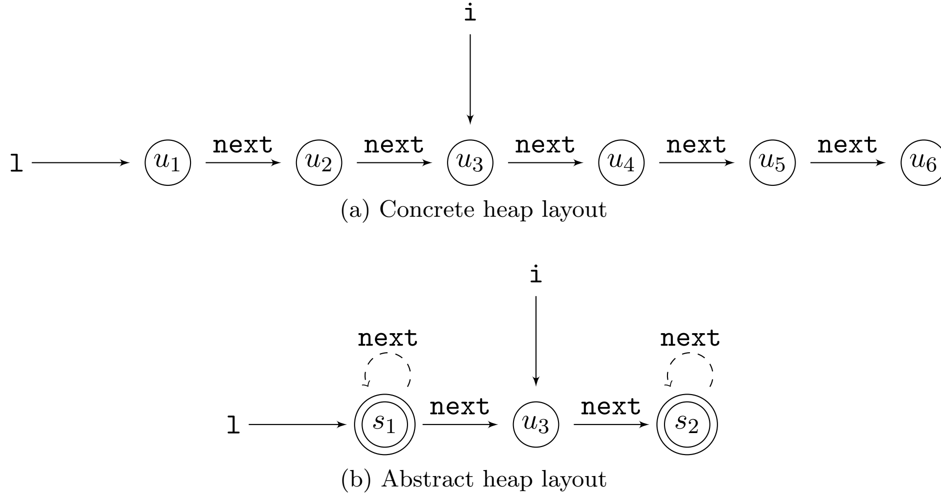

Figure 2.5.: Example of a shape graph. `i` is a program variable iterating through a list pointed to by head `l`. In (a) the concrete list is shown, in (b) an abstracted version with two summary nodes that could be used as an iterator loop's invariant.

framework by using different 3-valued logic predicates, making the class of programs that can be evaluated depend on the set of predicates used. One could for example use both the predicate

$$\forall v. \neg \exists v_1, v_2. next(v_1, v) \wedge next(v_2, v) \wedge v_1 \neq v_2$$

to (partially) define acyclicity as well as the predicate

$$\forall v_1, v_2, v. (next(v_1, v) \wedge next(v_2, v)) \rightarrow (v_1 = v_2 \ \& \ v_1 \neq v)$$

The main benefit of shape analysis over the approach suggested in this thesis is that it is very well-suited to reason on program properties related to certain heap shapes, such as lists or trees. In particular, shape analysis is good in coping with changing heap structures, whereas the approach in this thesis is rigid with respect to the chosen domain. That is, if assignments of the form `t.next = ...` occur in a program the approach using abstract domains may become rather imprecise. However, in this thesis it is attempted to integrate the abstraction of objects with the existing abstraction framework in place for integers, enabling object abstraction using the same paradigm.

Shape analysis techniques cannot be directly translated into this same approach. Using the invariant update approach each program variable has to get assigned a value, whereas in shape analysis the exact value of an individual program variable is not directly important; it is the shape of the whole heap that is abstracted. As a result it is problematic to create an abstract domain based on shape analysis and have concretized instances of each abstract element in the form of $\gamma_{a,z}$ to which program variables can be updated.

One approach that may be attempted is to create a more relational abstract lattice in which relations between program variables can be described and abstracted. An update to one program variable however may in such a setting also require to change the other program variables with which its relation changes due to that update. Consequently one assignment may result in the update of several program variables, which makes it hard to maintain a correct state. Therefore it is not trivial to convert concepts from shape analysis to the abstract lattice based approach used in this thesis.

3. Information Flow Analysis

Much research in the area of computer security has been focused on regulating access to information. A large number of protection measures exist (file permissions, encryption standards etc.), however in this thesis we look at how this information, once obtained, is propagated through a program. In the research area of information flow analysis it is aimed to detect if sensitive information handled in a program can leak to outside observers who have no permission to access that information. In Section 1.4 this research area is explained in more detail. In this chapter we look at how to create an automatic information flow analysis for the programs that can be written in the dynamic logic from Section 1.2.2.

As is customary, we use the convention that `h`-program variables contain, at the start of the program execution, information of a High security level, whereas `l`-program variables contain information of a Low security level. Furthermore, the value of `l`-program variables after the execution of the program is observable by outside processes.

Example 22:

We consider a program to be secure if an attacker cannot deduce any information on the High variables by observing the Low ones (*non-interference*).

(i) `l = h` is *insecure* because it leaks information directly from `h` to `l`.

(ii) `if (h > 0) {l = 10} else {l = 20}` is *insecure* because partial information on the value of `h` can be deduced from the value of `l`.

(iii) `if (h > 0) {l = 10} else {l = 20}; l = 0` is *secure* because the final value of `l` does not depend on `h`.

(iv) `if (h > 0) {l = 2} else {l = 2}` is *secure* because the final value of `l` is the same whether the first or second branch has been taken.

(v) `h = 0; l = h` is *secure* because the value of `l` is always 0.

(vi) `l = h; l = l - h` is *secure* because the value of `l` is always 0.

(vii) `if (h > 0) {h = l; l = h}` is *secure* because the value of `l` is not changed.

(viii) `if (h - h + l > 0) {l = 0} else {l = 2}` is *secure* because the value of `h` has no influence on the condition, and thus the final value of `l` is only affected by the value of `l` before execution of the program.

As pointed out in Section 1.4, a common approach is to use a type system that adds an additional type to program variables indicating its security level. Via type interference or type checking non-interference is ensured. These systems are always *sound*, i.e. they never classify

an insecure program as secure. However to a reasonable degree of automation, compromises need be made with respect to the completeness of these systems, i.e. they may classify secure programs as insecure, or ‘unknown’.

Analyses that have troubles correctly labeling secure programs of the kind such as (iii) and (iv) in Example 22, lack *control-flow sensitivity.* Programs of the kind as (v),(vi),(vii) and (viii) require an analysis to have *value sensitivity* in order to correctly label them as secure. In their article, Bubel et al. extend the program logic as described in detail in Chapter 2 to create an automatic information flow analysis. Although they establish an improvement in precision and achieving value-sensitivity when compared to most type-based systems, the resulting approach is still incapable of showing that programs of the kind (iv), (vi) or (viii) are secure.

In the rest of this chapter, we first introduce the concept of dependencies that serves as the basic concept for the information flow analysis. In Section 3.2 the extension by Bubel et al. is described and its limitations are identified, followed by improvements on that extension fixing those limitations in Section 3.3.

3.1. Dependencies

Considering the example programs in Example 22, one could state that in order for one of those programs to be secure, the final value of `l` should not depend on the initial value of `h`. In a more general sense, we could state that the final value of a variable `x` *depends* on a number of variables, say $\bar{\mathtt{y}}$ (possibly including `x`). We call these variables $\bar{\mathtt{y}}$ the *dependencies* of variable `x`. A more formal definition of the notion of *dependencies* is given in Definition 18 (as adopted from (Bubel et al., 2009)).

Definition 18 (Variable Dependencies)**.** *Given a program variable* `x` *and a program* `p`*, the* dependencies of `x` under `p` *form the smallest set* $\mathcal{D}(\mathtt{x},\mathtt{p}) \subseteq \mathcal{F}_{n,0}$ *of program variables such that the following holds for all first-order structures* $M = (\mathsf{D}, I)$ *and all variable assignments* β*: if* $s_1, s_2 \in \mathcal{S}$ *are such that we have* $s_1(\mathtt{y}) = s_2(\mathtt{y})$ *for all* $\mathtt{y} \in \mathcal{D}(\mathtt{x},\mathtt{p})$*, then either*

- $val_{M,s_1,\beta}(\mathtt{p}) = val_{M,s_2,\beta}(\mathtt{p}) = \emptyset$ *(i.e., from both initial states the execution of* `p` *does not terminate), or*
- $val_{M,s_1,\beta}(\mathtt{p}) = \{s_1'\}$ *and* $val_{M,s_2,\beta}(\mathtt{p}) = \{s_2'\}$ *and* $s_1'(\mathtt{x}) = s_2'(\mathtt{x})$ *(i.e., from both initial states the execution terminates and yields the same value for* `x`*).*

We use some terminology to differentiate between two classes of dependencies. The term *explicit* dependencies is used for program variables that are part of the term to which a variable is updated, e.g. in `x = y`, the program variable `y` is an explicit dependency of `x`. The term *implicit* dependencies is used for program variables that indirectly influence the update to a variable, e.g. in `if(y>0){x=1}else{x=0}`, the program variable `y` is an implicit dependency of `x`.

To be able to reason about dependencies in logical formulas in the way we talk about other program properties, we cannot use this exact definition that compares all possible runs of a program. We prefer it to be a local property that holds or does not hold in a given program state. Hence in the next section we extend the logic and semantics of programs so that we can store dependencies explicitly in states.

3.2. Including Dependencies in the Logic

In (Bubel et al., 2009) the dynamic logic from Section 1.2.2 is extended to explicitly track dependencies. In this section we discuss those extensions. To simplify the introduction of this dependency tracking we stick to the dynamic logic without the extension for objects, i.e. the logic as described in Section 1.2.2. The core concept of this approach is to extend the set of program variables with a program variable $\mathtt{x}^{dep}$ for each existing variable $\mathtt{x}$. Simultaneously with an update to $\mathtt{x}$, its corresponding dependency variable $\mathtt{x}^{dep}$ is updated to the set of variables that $\mathtt{x}$ now depends on. At the end of the symbolic execution of a program, the variable $\mathtt{x}^{dep}$ contains the dependencies of $\mathtt{x}$.

3.2.1. Extensions to Signature, Syntax and Semantics

We first extend the signature definition by adding sets of program variables used to represent dependencies and additional program variables $\mathtt{x}^{dep}$ to store dependencies.

Definition 19 (Signature with dependencies)**.** *Given a signature $\Sigma = (\mathcal{F}, \mathcal{P}, \mathcal{V})$ as defined in Definition 5, the* dependency extension *of Σ is a signature $\Sigma^{dep} = (\mathcal{S}, \mathcal{F}^{dep}, \mathcal{P}^{dep}, \mathcal{V})$, where*

- *$\mathcal{S}$ is a set of sorts to which terms can be assigned, and contains at least the sort* `LocSet`*,*
- *$\mathcal{F}_r^{dep} = \mathcal{F}_r \cup \{\mathtt{\{\}}, \dot{\cup}\} \cup \{\mathtt{\{x\}} \mid \mathtt{x} \in \mathcal{F}_{n,0}\}$, where $\mathtt{\{\}}$ is a constant symbol of type* `LocSet`*, $\dot{\cup}$ is a function symbol with arity 2 of type* $\mathtt{LocSet} \times \mathtt{LocSet} \to \mathtt{LocSet}$*, and where the $\mathtt{\{x\}}$ are function symbols with arity 0 of type* `LocSet`*,*
- *$\mathcal{F}_{n,0}^{dep} = \mathcal{F}_{n,0} \cup \{\mathtt{x}^{dep} \mid \mathtt{x} \in \mathcal{F}_{n,0}\}$, and*
- *$\mathcal{P}^{dep} = \mathcal{P} \cup \{\dot{\subseteq}\}$, where $\dot{\subseteq}$ is a predicate symbol with arity 2 of type* $\mathtt{LocSet} \times \mathtt{LocSet}$.

We further require that:

- *Programs over a signature Σ^{dep} are built only from the symbols already present in the sub-signature Σ,*
- *We only consider universes $\mathsf{D} \supseteq \wp(\mathcal{F}_{n,0})$ where every set of program variables also occurs as a value in the universe, and*
- *We only allow interpretations I that fix the meaning of the additional symbols as follows:*
  - *$I(\mathtt{\{\}}) = \emptyset$,*
  - *for all $P_1, P_2 \in \wp(\mathcal{F}_{n,0})$: $I(\dot{\cup})(P_1, P_2) = P_1 \cup P_2$,*
  - *for all $\mathtt{x} \in \mathcal{F}_{n,0}$: $I(\mathtt{\{x\}}) = \{\mathtt{x}\}$, and*
  - *$I(\dot{\subseteq}) = \{(P_1, P_2) \mid P_1 \subseteq P_2 \subseteq \mathcal{F}_{n,0}\}$.*

To emphasize the second requirement: sets such as $\{\mathtt{x}, \mathtt{y}\}$ are now part of the concrete domain. We can create arbitrary sets of program variables on the logic level, such as $\mathtt{\{x\}} \,\dot{\cup}\, \mathtt{\{y\}} \,\dot{\cup}\, \mathtt{\{z\}}$. As a shorthand, we can also write `{x,y,z}` instead.

Definition 20 (Program Semantics with Dependencies)**.** *Similar as in Definition 8, given a first-order structure $M = (\mathsf{D}, I)$, a state s and a logic variable assignment β, we evaluate programs $\mathtt{p}$ to a set of states $val'_{M,s,\beta}(\mathtt{p}) \subseteq \mathcal{S}$. As before, our programs are deterministic, so the sets always have at most one element. The evaluation of terms, formulas and updates remains the same (and uses the regular function $val_{M,s,\beta}$), the new function $val'_{M,s,\beta}$ for programs is given below.*

$$val'_{M,s,\beta}(\texttt{x = } t) = \big\{ val_{M,s,\beta}\big(\texttt{x} := t \parallel \texttt{x}^{dep} := deps(t)\big) \big\}$$

$$val'_{M,s,\beta}(\texttt{p1;p2}) = \big\{ val'_{I,s',\beta}(\texttt{p2}) \mid s' \in val'_{M,s,\beta}(\texttt{p1}) \big\}$$

$$val'_{M,s,\beta}(\textbf{if}(g)\{\texttt{p1}\}\ \textbf{else}\ \{\texttt{p2}\}) = \begin{cases} S'_1 & \text{if } val_{M,s,\beta}(g) = tt \\ S'_2 & \text{otherwise} \end{cases}$$

where $S_1 = val'_{M,s,\beta}(\texttt{p1}), S_2 = val'_{M,s,\beta}(\texttt{p2})$,

$S'_i = \emptyset$ iff $S_i = \emptyset$, otherwise $S'_i = \{s'_i\}$ where

$$s'_i(\texttt{x}) = \begin{cases} s_i(\texttt{x}) & \text{if } \texttt{x} \in \mathcal{PV} \text{ or} \\ & \texttt{x} = \texttt{y}^{dep} \text{ and} \\ & s_i(\texttt{y}) = s(\texttt{y}) \\ & \text{for } S_i = \{s_i\} \\ s_i(\texttt{x}) \cup val_{M,s,\beta}(deps(g)) & \text{otherwise} \end{cases}$$

$$val'_{M,s,\beta}(\textbf{while}\ (g)\ \{\texttt{p}\}) = \begin{cases} \bigcup_{s'_1 \in S'_1} val'_{I,s'_1,\beta}(\textbf{while}(g)\ \{\texttt{p}\}) & \text{if } val_{M,s,\beta}(g) = tt \\ \{s\} & \text{otherwise} \end{cases}$$

where $S_1 = val'_{M,s,\beta}(\texttt{p})$,

and where S'_1 is derived from S_1 as above

The two main differences with the regular $val_{M,s,\beta}$ function for program statements are in the assignment and in the handling of conditions from conditional and loop statements. When performing an assignment to x, the corresponding program variable $\texttt{x}^{dep}$ is updated as well. The term $deps(t)$ over-approximates the precise semantic dependencies of t. The function *deps* takes a term and returns a term, and is defined as follows:

$$\begin{aligned}
deps(f(t_1,\ldots,t_n)) &= deps(t_1) \,\dot\cup \ldots \dot\cup\, deps(t_n) \\
deps(\texttt{x}) &= \texttt{x}^{dep} \qquad \text{where } \texttt{x} \in \mathcal{F}_{n,0} \\
deps(if(\varphi)then(t_1)else(t_2)) &= deps(\varphi) \,\dot\cup\, deps(t_1) \,\dot\cup\, deps(t_2) \\
deps(a) &= \texttt{\{\}} \qquad \text{where } a \in \{\text{true}, \text{false}\} \\
deps(p(t_1,\ldots,t_n)) &= deps(t_1) \,\dot\cup \ldots \dot\cup\, deps(t_n) \\
deps(\varphi_1 * \varphi_2) &= deps(\varphi_1) \,\dot\cup\, deps(\varphi_2) \qquad \text{where } * \in \{\&, |, \texttt{->}\} \\
deps(!\,\varphi) &= deps(\varphi) \\
deps(t_1 \doteq t_2) &= deps(t_1) \,\dot\cup\, deps(t_2)
\end{aligned}$$

EXAMPLE 23:

Some results of applying the *deps* function on a term:

- $deps(\texttt{a} + \texttt{b}) = \texttt{a}^{dep} \,\dot\cup\, \texttt{b}^{dep}$
- $deps(if(\texttt{h} > 4)then(3)else(\texttt{k})) = \texttt{h}^{dep} \,\dot\cup\, \texttt{k}^{dep}$

The second difference with $val_{M,s,\beta}$ is located in the handling of conditional and loop statements. The function $val'_{M,s,\beta}$ adds $deps(g)$ to $\mathtt{x}^{dep}$ only if `x` is changed in the body of the conditional or while loop, to cover the implicit flow of information.

Example 24:

To prove that the non-interference property holds for program (vii) from Example 22, i.e. that no information is leaked from the High variable `h` to the Low variable `l`, we have to show that the sequent

$$\mathtt{h}^{dep} = \{\mathtt{h}\}, \mathtt{l}^{dep} = \{\mathtt{l}\} \Longrightarrow [\mathtt{if\ (h>0)\ \{h=l;\ l=h\}}](\mathtt{l}^{dep} \dot{\subseteq} \{\mathtt{l}\})$$

is valid. The formulas placed in the antecedent denote that we assume the initial value of a program variable, i.e. before execution of the program, depends only on itself. The post-condition after the program denotes that the final value of `l` at most depends on the initial value of `l`.

The semantics indeed do not add `h` as a dependency, since the value of `l` is left unchanged.

3.2.2. Extensions to Calculus Rules

In accordance with the extensions made in Definition 20 we update the calculus rules. Only the semantics of programs have changed from the original dynamic logic semantics. Hence the rules not dealing with symbolic execution of programs, e.g. the weakenUpdate rule, are left unchanged. For the assignment rule, we can almost directly translate the semantics into a calculus rule:

$$\text{assignment}^{\text{dep}}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}\{\mathtt{x} := t \parallel \mathtt{x}^{dep} := deps(t)\}[\ldots]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathtt{x\ =\ }t\mathtt{;\ \ }\ldots]\varphi, \Delta}$$

For conditional statements, the new semantics adds the dependencies from the condition to the tracked dependencies of a program variable, only if its value has been changed at the end of a branch. We translate this concept by adding a special shaped update $\mathcal{V}$ into the premises of the rule:

$$\text{ifElse}^{\text{dep}}\ \frac{\begin{array}{c}\Gamma, \{\mathcal{U}\}g,\ \{\mathcal{U}\}(\bar{\mathtt{y}} \doteq \bar{y}^{pre}) \Longrightarrow \{\mathcal{U}\}[\mathtt{p1}]\{\mathcal{V}\}[\ldots]\varphi, \Delta \\ \Gamma, \{\mathcal{U}\}\,!\,g,\ \{\mathcal{U}\}(\bar{\mathtt{y}} \doteq \bar{y}^{pre}) \Longrightarrow \{\mathcal{U}\}[\mathtt{p2}]\{\mathcal{V}\}[\ldots]\varphi, \Delta\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathtt{if\ \ (}g\mathtt{)\ \ \{p1\}\ else\ \{p2\};\ \ }\ldots]\varphi, \Delta}$$

where

- $\bar{\mathtt{y}} = (\mathtt{y}_1, \mathtt{y}_1^{dep}, \ldots, \mathtt{y}_n, \mathtt{y}_n^{dep})$ is a list of all program variables occurring in g, `p1` or `p2`, together with the corresponding dependency variables,
- $\bar{y}^{pre} = (y_1^{pre}, y_1^{predep} \ldots, y_n^{pre}, y_n^{predep})$ is a list of fresh constant symbols of the same length as $\bar{\mathtt{y}}$, and
- $\mathcal{V}$ is the update

$$\mathtt{y}_1^{dep} := if(\mathtt{y}_1 \doteq y_1^{pre})then(\mathtt{y}_1^{dep})else\big(\mathtt{y}_1^{dep} \,\dot{\cup}\, \{\bar{\mathtt{y}} := \bar{y}^{pre}\}deps(g)\big)$$

$$\| \dots \|$$
$$\mathtt{y}_n^{dep} := \mathit{if}(\mathtt{y}_n \doteq y_n^{pre})\,\mathit{then}(\mathtt{y}_n^{dep})\,\mathit{else}\big(\mathtt{y}_n^{dep} \dot\cup \{\bar{\mathtt{y}} := \bar{y}^{pre}\}\,\mathit{deps}(g)\big)$$

Here the fresh constant symbols $\bar{y}^{pre}$ store the pre-state values of the program variables $\bar{\mathtt{y}}$ before the symbolic execution of the conditional statement. The update $\mathcal{V}$ checks after execution of each branch whether a program variable has changed, by comparing it with its value in the pre-state. If so, the set $deps(g)$ is added to its dependencies (evaluated in the pre-state) otherwise the program variables affecting g are not added as a dependency of that program variable.

The same idea can be applied to the loopUnwind and invariantUpdate rules (where $\bar{\mathtt{y}}$, $\bar{y}^{pre}$ and $\mathcal{V}$ are defined as above):

$$\mathsf{loopUnwind}^{\mathrm{dep}}\quad \frac{\begin{array}{l}\Gamma, \{\mathcal{U}\}g,\ \{\mathcal{U}\}(\bar{\mathtt{y}} \doteq \bar{y}^{pre}) \Longrightarrow \{\mathcal{U}\}[\mathtt{if}\ (g)\ \{\mathtt{p}\}]\{\mathcal{V}\}[\mathtt{while}\ (t)\ \{\mathtt{p}\};\ \ldots]\varphi, \Delta \\ \Gamma, \{\mathcal{U}\}\,!\,g \Longrightarrow \{\mathcal{U}\}[\ldots]\varphi, \Delta\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathtt{while}\ (g)\ \{\mathtt{p}\};\ \ldots]\varphi, \Delta}$$

$$\mathsf{invariantUpdate}^{\mathrm{dep}}\quad \frac{\begin{array}{l}\Gamma, \{\mathcal{U}\}(\bar{\mathtt{x}} \doteq \bar{c}) \Longrightarrow \exists \bar{\gamma}.\{\mathcal{U}'\}(\bar{\mathtt{x}} \doteq \bar{c}), \Delta \\ \Gamma, \{\mathcal{U}'\}g,\ \{\mathcal{U}'\}(\bar{\mathtt{y}} \doteq \bar{y}^{pre}),\ \{\mathcal{U}'\}[\mathtt{if}\ (g)\ \{\mathtt{p}\}]\{\mathcal{V}\}(\bar{\mathtt{x}} \doteq \bar{c}) \Longrightarrow \exists \bar{\gamma}.\{\mathcal{U}'\}(\bar{\mathtt{x}} \doteq \bar{c}), \Delta \\ \Gamma, \{\mathcal{U}'\}\,!\,g \Longrightarrow \{\mathcal{U}'\}[\ldots]\varphi, \Delta\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathtt{while}\ (g)\ \{\mathtt{p}\};\ \ldots]\varphi, \Delta}$$

The execution of the body of the while loop is wrapped in the condition of the while loop, both in the loopUnwind and the invariantUpdate rule. This is done in order to make sure that the implicit dependencies are tracked correctly.

Furthermore, to apply abstraction in the dependency-aware version of the calculus two abstract domains have to be defined (as adopted from Bubel et al. (2009)):

1. The abstract domain $\mathcal{A}_{val}$ for the value abstraction of *normal* program variables that carry values. The choice of $\mathcal{A}_{val}$ depends on the application context. An example is the sign domain for integers.

2. The abstract domain $\mathcal{A}_{dep}$ for the value abstraction of the *dependency* program variables. Again, the suitable choice depends on the application context. In an information-flow security context, a natural choice for $\mathcal{A}_{dep}$ is suggested by the security lattice. An example of such a lattice can be found in Figure 3.1.

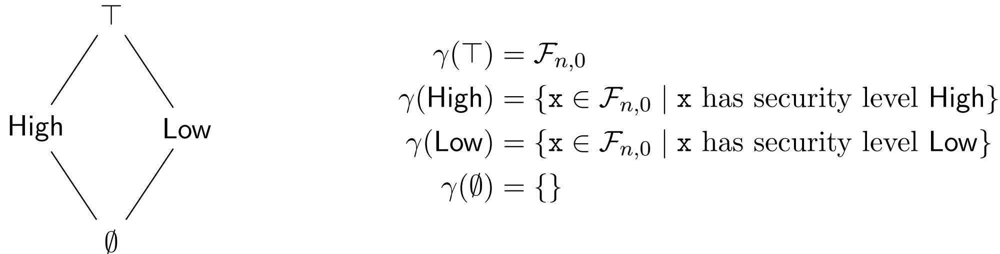

Figure 3.1.: Abstract domain lattice for program variables, based on a setting in which the security level of a program variable can be either High or Low

The proof search strategy remains nearly unchanged from the standard version as defined in Section 2.2.3. When computing an abstraction, the abstract domain $\mathcal{A}_{val}$ is used for normal program variables $\mathtt{x} \in \mathcal{F}_{n,0}$ and $\mathcal{A}_{dep}$ for dependency program variables $\mathtt{x}^{dep} \in \mathcal{F}^{dep}_{n,0}$.

The second premise of the invariantUpdate rule has been changed to ensure correct tracking of implicit dependencies. As a result, we compute the weakened update in the join procedure (Section 2.2.3) no longer from the body `p` of the while loop, but from the program **`if`**`(g){p}`, so that we correctly include the implicit dependencies in the search for a fixed point.

EXAMPLE 25:

We prove the non-interference of the program `p` :

```
l = 3;
while(h > 0) {
  l = l + h;
  h = h - 1
};
if (l > 0) {
  l = 4
}
```

The non-interference is formulated in the sequent:

$$\mathtt{l}^{dep} \doteq \mathtt{\{l\}}, \mathtt{h}^{dep} \doteq \mathtt{\{h\}} \Longrightarrow [\mathtt{p}](\mathtt{l}^{dep} \dot{\subseteq} \mathtt{\{l\}})$$

We symbolically execute the first assignment:

$$\Gamma \Longrightarrow \{\mathtt{l} := 3 \,\|\, \mathtt{l}^{dep} := \mathtt{\{\}}\}[\mathtt{while(h>0)\{l=l+h;h=h-1\}if(l>0)\{l=4\}}]\varphi$$

The automatic search for a fixed point computes as an invariant of this while loop the update $\mathcal{U}'$:

$$\mathtt{l} := \gamma_{pos,0} \,\|\, \mathtt{l}^{dep} := \gamma_{\mathsf{High},0} \,\|\, \mathtt{h} := \gamma_{\top,0} \,\|\, \mathtt{h}^{dep} := \mathtt{\{h\}}$$

We apply the $\mathsf{invariantUpdate}^{\mathsf{dep}}$ rule using $\mathcal{U}'$ as the weakening update, or: invariant update. We focus on the third branch resulting from this application:

$$\Gamma \Longrightarrow \{\mathcal{U}'\}[\mathtt{if(l>0)\{l=4\}}]\varphi$$

Application of the $\mathsf{ifElse}^{\mathsf{dep}}$ rule results in two branches. We know that the condition $\mathtt{l} > 0$ is always true, so we can close the else-branch immediately. We continue with the then-branch, which gives us after symbolic execution of the assignment:

$$\Gamma, \gamma_{pos,0} > 0 \Longrightarrow \{\mathcal{U}'\}\{\mathtt{l} := 4 \,\|\, \mathtt{l}^{dep} := \mathtt{\{\}}\}\varphi$$

Applying these updates on the formula φ results in:

$$\Gamma, \gamma_{pos,0} > 0 \Longrightarrow \mathtt{\{\}} \dot{\subseteq} \mathtt{\{l\}}$$

Which is obviously valid.

3.2.3. Limitations

Although the approach described in the previous sections correctly over-approximates the dependencies of a program variable, this over-approximations is, as the name suggests, imprecise. That is, for programs such as (vi) or (iv) from Example 22 the set of program variables tracked in $\mathtt{l}^{dep}$ is larger than the real set of dependencies, resulting in the inability to prove the non-interference property.

In this section we identify the causes of this incompleteness by running through those example programs. We start with program (vi): `l = h; l = l - h;`. The sequent to prove for non-interference is:

$$\mathtt{h}^{dep} \doteq \{\mathtt{h}\}, \mathtt{l}^{dep} \doteq \{\mathtt{l}\} \Longrightarrow [\mathtt{l=h;l=l-h}](\mathtt{l}^{dep} \dot{\subseteq} \{\mathtt{l}\})$$

We collapse the antecedent into Γ, the non-interference formula to φ and symbolically execute the first assignment:

$$\Gamma \Longrightarrow \{\mathtt{l} := \mathtt{h} \parallel \mathtt{l}^{dep} := \mathtt{h}^{dep}\}[\mathtt{l=l-h}]\varphi$$

The current state transition described by the update is correct: the value of `l` is depending on the initial value of `h`. We now symbolically execute the second assignment and parallelize the two (omitting the elementary updates overruled by the 'last-one wins' semantics as usual):

$$\Gamma \Longrightarrow \{\mathtt{l} := \mathtt{h} - \mathtt{h} \parallel \mathtt{l}^{dep} := \mathtt{h}^{dep}\}\varphi$$

At this stage the state transition described by the update is still correct, however over-approximates the dependencies of `l`. We identify that this is caused by the fact that in the $\mathsf{assignment}^{\mathsf{dep}}$ rule the program variable $\mathtt{l}^{dep}$ is updated to a term solely based on the *syntactical* term assigned to `l`. As a result, the dependencies to which $\mathtt{l}^{dep}$ gets updated have *no value sensitivity.* Therefore the fact that the term to which `l` is updated always evaluates to 0, independent of the value of `h`, goes unnoticed for the update to $\mathtt{l}^{dep}$.

We continue with the example program (iv), which was **if** `(h > 0) {l = 2}` **else** `{l = 2}`. Again, we place this in a sequent for proving non-interference:

$$\mathtt{h}^{dep} \doteq \{\mathtt{h}\}, \mathtt{l}^{dep} \doteq \{\mathtt{l}\} \Longrightarrow [\mathtt{if(h>0)\{l=2\}else\{l=2\}}](\mathtt{l}^{dep} \dot{\subseteq} \{\mathtt{l}\})$$

We symbolically execute this conditional statement, resulting in the following two premises (note that $\bar{\mathtt{y}} = \{\mathtt{l}, \mathtt{l}^{dep}\}$):

$$\begin{aligned}
&\Gamma, \mathtt{h} > 0, \mathtt{l} \doteq \mathtt{l}^{pre}, \mathtt{l}^{dep} \doteq \mathtt{l}^{predep} \Longrightarrow \\
&\quad [\mathtt{l=2}]\{\mathtt{l}^{dep} := \mathit{if}(\mathtt{l} \doteq \mathtt{l}^{pre})\mathit{then}(\mathtt{l}^{dep})\mathit{else}(\mathtt{l}^{dep} \dot{\cup} \{\mathtt{l} := \mathtt{l}^{pre} \parallel \mathtt{l}^{dep} := \mathtt{l}^{predep}\}\mathit{deps}(g))\}\varphi \quad (1)\\
&\Gamma, !(\mathtt{h} > 0), \mathtt{l} \doteq \mathtt{l}^{pre}, \mathtt{l}^{dep} \doteq \mathtt{l}^{predep} \Longrightarrow \\
&\quad [\mathtt{l=2}]\{\mathtt{l}^{dep} := \mathit{if}(\mathtt{l} \doteq \mathtt{l}^{pre})\mathit{then}(\mathtt{l}^{dep})\mathit{else}(\mathtt{l}^{dep} \dot{\cup} \{\mathtt{l} := \mathtt{l}^{pre} \parallel \mathtt{l}^{dep} := \mathtt{l}^{predep}\}\mathit{deps}(g))\}\varphi \quad (2)
\end{aligned}$$

These two sequents become, after symbolic execution of the assignment, parallelization and application of updates, and rewriting them using Γ:

$$\begin{aligned}
\Gamma, \mathtt{h} > 0, \mathtt{l} \doteq \mathtt{l}^{pre}, \mathtt{l}^{dep} \doteq \mathtt{l}^{predep} &\Longrightarrow \mathit{if}(2 \doteq \mathtt{l})\mathit{then}(\{\})\mathit{else}(\{\mathtt{h}\}) \dot{\subseteq} \{\mathtt{l}\} \quad (1)\\
\Gamma, !(\mathtt{h} > 0), \mathtt{l} \doteq \mathtt{l}^{pre}, \mathtt{l}^{dep} \doteq \mathtt{l}^{predep} &\Longrightarrow \mathit{if}(2 \doteq \mathtt{l})\mathit{then}(\{\})\mathit{else}(\{\mathtt{h}\}) \dot{\subseteq} \{\mathtt{l}\} \quad (2)
\end{aligned}$$

We cannot show that the condition $2 \doteq \mathtt{l}$ holds, therefore a cut has to be performed on whether this condition holds or not. As a result, we cannot prove above sequents. The reason that again we have an over-approximation of the actual dependencies of `l` is caused by the split into two

sequents as a result of the $\mathsf{ifElse}^{\mathsf{dep}}$ rule. Since we cannot formulate a calculus rule combining two sequents, it is impossible to tell that each branch results in the same update to `l`. This can therefore be classified as a combination of a lack of value sensitivity and *control flow sensitivity*.

In the next section we improve this extension with a more precise approximation of the dependencies by addressing these two issues.

3.3. Improving Value and Control-Flow Sensitivity

Section 3.2.3 shows that the original approach is limited in its handling of value sensitivity within terms, such as: `y - y` always becomes 0, and has a lack of control-flow sensitivity (though arguably in combination with value sensitivity), such as: in **if**(g)`{x=1}`**else**`{x=1}`, `x` always becomes 1. In this section we suggest an extension based on the one in Section 3.2 that addresses these issues.

To improve the value sensitivity, $\mathtt{x}^{dep}$ no longer directly represents the approximated dependencies of `x`, but a Herbrand universe-like version (Herbrand, 1930) of the term t to which `x` is updated (called a term expression). For example if `x` is updated to $\mathtt{x} - \mathtt{x}$, the variable $\mathtt{x}^{dep}$ is (more or less) updated to $\mathtt{x}_c -_c \mathtt{x}_c$. In the semantics we evaluate term expressions that are equivalent to the same equivalence class. This allows us to create rewrite rules on term expressions that evaluate to the same equivalence class, enabling value sensitivity.

To improve the control flow sensitivity we delay the split based on the condition of a conditional statement to *after* the symbolic execution of both branches, by making use of temporal variables and the conditional term constructor $\mathit{if}(\varphi)\mathit{then}(t)\mathit{else}(t)$. As a result we have a variable's value from both branches present in the same sequent, allowing for calculus or rewrite rules that increase the control flow sensitivity.

3.3.1. Extensions to Signature, Syntax and Semantics

The sets of rigid functions is extended with a copy of the current rigid functions, non-rigid functions and predicates (denoted by the subscript $_c$). One could think of these elements as the ground terms for a Herbrand universe, however they are not used as the interpretation for the original symbols. Instead, they are used as a constant copy of these symbols (hence the subscript $_c$) for updates to dependency variables. We call these symbols *term expressions* and give them the type `TExp`.

Definition 21 (Signature with dependencies)**.** *Given a dynamic logic signature* $\Sigma = (\mathcal{F}, \mathcal{P}, \mathcal{V})$ *from Definition 5, the* dependency extended *version of* Σ *is a signature* $\Sigma^{dep} = (\mathcal{S}, \mathcal{F}^{dep}, \mathcal{P}^{dep}, \mathcal{V})$*. The new sets are defined as follows:*

- $\mathcal{S}$ *is a set of sorts to which terms can be assigned, and contains at least the sorts* `LocSet` *and* `TExp`*,*
- $\mathcal{F}_r^{dep} = \mathcal{F}_r \cup \{\mathtt{\{\}}, \dot{\cup}\} \cup \{\mathtt{\{x\}} \mid \mathtt{x} \in \mathcal{F}_{n,0}\} \cup \{\mathcal{W}\} \cup \mathcal{F}_{\mathtt{TExp}} \cup P_{\mathtt{TExp}}$ *where all symbols are as in Definition 19,* $\mathcal{W}$ *is a rigid function of arity 2 and* $\mathcal{F}_{\mathtt{TExp}} = \{f_c \mid f \in \mathcal{F}\}$*. Given that* f *has arity* n*,* f_c *has type* $\mathtt{TExp}^n \rightarrow \mathtt{TExp}$*. Note that we add constant* f_c *functions for* both *rigid and non-rigid functions. Similar* $P_{\mathtt{TExp}} = \{p_c \mid p \in P\}$*. Given that* p *has arity* n*,* p_c *has type* $\mathtt{TExp}^n \rightarrow \mathtt{TExp}$*. Note specifically that the symbols in the set* $P_{\mathtt{TExp}}$ *are rigid function symbols as well and* not *predicate symbols.*
- $\mathcal{F}_{n,0}^{dep} = \mathcal{F}_{n,0} \cup \{\mathtt{x}^{dep} \mid \mathtt{x} \in \mathcal{F}_{n,0}\}$.

- $\mathcal{P}^{dep} = \mathcal{P} \cup \{\dot{\subseteq}\} \cup \{\dot{\subseteq}_{dep}\}$, *where* $\dot{\subseteq}$ *is as in Definition 19,* $\dot{\subseteq}_{dep}$ *has arity 2 and type* $\texttt{TExp} \times \texttt{LocSet}$.

We adopt the additional requirements from Definition 19.

The $\mathcal{W}$ function introduced here is mainly an aid that is needed for tracking implicit dependencies resulting from while loop conditions. The $\dot{\subseteq}_{dep}$ predicate is similar to the $\dot{\subseteq}$ predicate. It is introduced since the $\texttt{x}^{dep}$ variables are no longer evaluated to a `LocSet` but to a `TExp`. The $\dot{\subseteq}_{dep}$ allows us to still compare the explicitly tracked dependencies with defined location sets, e.g. $\texttt{x}^{dep}\dot{\subseteq}_{dep}\{\texttt{x}\}$.

EXAMPLE 26:

In our example a subset of the functions $\mathcal{F}$ is:

$$\{+ : \texttt{int} \times \texttt{int} \to \texttt{int}, - : \texttt{int} \times \texttt{int} \to \texttt{int}, 0 : \texttt{int}, 1 : \texttt{int}, \texttt{x} : \texttt{int}\}$$

The set $\mathcal{F}_{\texttt{TExp}}$ (and thus also $\mathcal{F}'$) therefore has the subset:

$$\{+_c : \texttt{TExp} \times \texttt{TExp} \to \texttt{TExp}, -_c : \texttt{TExp} \times \texttt{TExp} \to \texttt{TExp}, 0_c : \texttt{TExp}, 1_c : \texttt{TExp}, \texttt{x}_c : \texttt{TExp}\}$$

Similar, a subset of the predicates $\mathcal{P}$ is:

$$\{\geq : \texttt{int} \times \texttt{int}, \leq : \texttt{int} \times \texttt{int}\}$$

Therefore the set $P_{\texttt{TExp}}$ has the subset:

$$\{\geq_c : \texttt{TExp} \times \texttt{TExp}, \leq_c : \texttt{TExp} \times \texttt{TExp}\}$$

Definition 22 (Syntax with term expressions)**.** *We extend the syntax of terms and add a syntax for term and formula expressions using only the newly added symbols. In the following* $f_c \in \mathcal{F}_{\texttt{TExp}}$ *and* $p_c \in P_{\texttt{TExp}}$*, we list here only the changes to the syntax definition given in Definition 6.*

$$\begin{aligned}
t &::= \cdots \mid t_{\texttt{Exp}} \mid \mathit{while}(\varphi, t)\\
t_{\texttt{Exp}} &::= f_c(t_{\texttt{Exp}}, \ldots, t_{\texttt{Exp}}) \mid \mathit{if}_c(\varphi_{\texttt{Exp}})\mathit{then}_c(t_{\texttt{Exp}})\mathit{else}_c(t_{\texttt{Exp}}) \mid \mathit{while}_c(\varphi_{\texttt{Exp}}, t_{\texttt{Exp}})\\
\varphi_{\texttt{Exp}} &::= \mathrm{true}_c \mid \mathrm{false}_c \mid p_c(t_{\texttt{Exp}}, \ldots, t_{\texttt{Exp}}) \mid \varphi_{\texttt{Exp}} \,\&_c\, \varphi_{\texttt{Exp}} \mid (\varphi_{\texttt{Exp}} \mid_c \varphi_{\texttt{Exp}}) \mid\\
&\quad\;\; \varphi_{\texttt{Exp}} \mathrel{-\!\!>_c} \varphi_{\texttt{Exp}} \mid\, !_c\varphi_{\texttt{Exp}} \mid t_{\texttt{Exp}} \doteq_c t_{\texttt{Exp}}
\end{aligned}$$

Note that the constructor $f_c(t_{\texttt{Exp}}, \ldots, t_{\texttt{Exp}})$ is superfluous since f_c is also in $\mathcal{F}'_r$ and thus the same term can be constructed with the constructor $f(t, \ldots, t)$ that already exists for terms (Definition 6). It is only added for the sake of Definition 23.

The $\texttt{x}^{dep}$ variables are updated to term expressions (`TExp`) and no longer to a set of program variables (`LocSet`). As a result we need a different way of 'storing' implicit dependencies, which for conditional statements is provided with $\mathit{if}_c(\varphi_{\texttt{Exp}})\mathit{then}_c(t_{\texttt{Exp}})\mathit{else}_c(t_{\texttt{Exp}})$ and for while loops with $\mathit{while}_c(\varphi_{\texttt{Exp}}, t_{\texttt{Exp}})$.

Since we want to be able to *unwrap* this constructor with the function defined below, we add the constructor $while(\varphi, t)$ for regular terms as well. This is just an aid to ensure that all dependencies, both implicit and explicit, are present in an unwrapped term. It is by no means intended as a fix point operator or μ-recursive function. The semantic interpretation of this term is the function $\mathcal{W}$ that is purposely under-specified to ensure that no two $while(\varphi, t)$ terms can be proven to evaluate to the same value except if they have exactly the same arguments.

There is no need to copy other constructors (similar for the copied formula constructors) since they are not allowed to appear in programs (Definition 6) and we therefore have no use for them.

The $if_c(\varphi_{\mathtt{Exp}})then_c(t_{\mathtt{Exp}})else_c(t_{\mathtt{Exp}})$ constructor is also important for establishing control flow sensitivity, as is shown in Section 3.3.3.

Definition 23 (Ground TExp terms and formulas)**.** *The set* $T^0_{\mathtt{TExp}} = \bigcup t_{\mathtt{TExp}}$*, i.e. the set of all terms that can be constructed from the* $\mathcal{F}_{\mathtt{TExp}}$ *function symbols and the syntax from Definition 22. Similar we create the set* $P^0_{\mathtt{TExp}} = \bigcup \varphi_{\mathtt{TExp}}$ *as the set of all ground formulas for term expressions.*

Example 27:

Some elements of $T^0_{\mathtt{TExp}}$ include:

- $3_c +_c \mathtt{x}_c$
- $while_c(\mathtt{h}_c >_c 0_c, 7_c -_c \mathtt{y}_c)$
- $\mathtt{z}_c +_c if_c(\text{true}_c)then_c(\mathtt{z}_c)else_c(0_c)$

Similar, some elements of $P^0_{\mathtt{TExp}}$ are:

- $d_c \dot{=}_c \mathtt{y}_c$ (d a constant)
- $3_c >_c 2_c$
- $(7_c \,/_c\, \mathtt{x}_c \dot{=}_c 0_c) \,\&_c\, \text{false}_c$

Note that this definition is similar to the construction of a Herbrand universe.

We define a function $unwrap : T^0_{\mathtt{Exp}} \to Term$ that returns a term in the logic based on the provided term expression. We overload the function to handle formulas as well, $unwrap : P^0_{\mathtt{Exp}} \to For$:

$$\begin{aligned}
unwrap(f_c(t_{\mathtt{Exp},1}, \ldots, t_{\mathtt{Exp},n})) &= f(unwrap(t_{\mathtt{Exp},1}), \ldots, unwrap(t_{\mathtt{Exp},n})) \\
unwrap(if_c(\varphi_{\mathtt{Exp}})then_c(t_{\mathtt{Exp},1}) &= if(unwrap(\varphi_{\mathtt{Exp}})) \\
else_c(t_{\mathtt{Exp},2})) &\quad then(unwrap(t_{\mathtt{Exp},1}))else(unwrap(t_{\mathtt{Exp},2})) \\
unwrap(while_c(\varphi_{\mathtt{Exp}}, t_{\mathtt{Exp}})) &= while(unwrap(\varphi_{\mathtt{Exp}}), unwrap(t_{\mathtt{Exp}})) \\
unwrap(\text{true}_c) &= \text{true} \\
unwrap(\text{false}_c) &= \text{false} \\
unwrap(p_c(t_{\mathtt{Exp},1}, \ldots, t_{\mathtt{Exp},n})) &= p(unwrap(t_{\mathtt{Exp},1}), \ldots, unwrap(t_{\mathtt{Exp},n})) \\
unwrap(\varphi_{\mathtt{Exp},1} *_c \varphi_{\mathtt{Exp},2}) &= unwrap(\varphi_{\mathtt{Exp},1}) * unwrap(\varphi_{\mathtt{Exp},2}) \quad \text{where } * \in \{\&, |, \text{->}\}
\end{aligned}$$

$$unwrap(!_c\varphi_{\texttt{Exp}}) = !\, unwrap(\varphi_{\texttt{Exp}})$$
$$unwrap(t_{\texttt{Exp},1} \doteq_c t_{\texttt{Exp},2}) = unwrap(t_{\texttt{Exp},1}) \doteq unwrap(t_{\texttt{Exp},2})$$

EXAMPLE 28:

Some examples of the operation of the rather simple *unwrap* function:

- $unwrap(3_c +_c \texttt{x}_c) = 3 + \texttt{x}$
- $unwrap(8_c \geq_c \texttt{y}_c) = 8 \geq \texttt{y}$

We use this *unwrap* function in the definition of equivalence for term and formula expressions. Two of these expressions are equivalent if the value of their unwrapped terms or formulas are equal under any first-order structure, state or logic variable assignment.

Definition 24 (Equivalence of term and formula expressions)**.** *Two term expressions $t_{\texttt{Exp}}, v_{\texttt{Exp}} \in T^0_{\texttt{TExp}}$ are said to be equivalent (denoted by $t_{\texttt{Exp}} \sim v_{\texttt{Exp}}$) if and only if for all first-order structure $M = (\mathsf{D}, I)$, states s and logic variable assignments β : $val_{M,s,\beta}(unwrap(t_{\texttt{Exp}})) = val_{M,s,\beta}(unwrap(v_{\texttt{Exp}}))$.*

Similar two formula expressions $\varphi_{\texttt{Exp}}, \phi_{\texttt{Exp}} \in P^0_{\texttt{TExp}}$ are equivalent ($\varphi_{\texttt{Exp}} \sim \phi_{\texttt{Exp}}$) if and only if for all first-order structures $M = (\mathsf{D}, I)$, states s and logic variable assignments β : $val_{M,s,\beta}(unwrap(\varphi_{\texttt{Exp}})) = val_{M,s,\beta}(unwrap(\phi_{\texttt{Exp}}))$.

EXAMPLE 29:

Recall that with Definition 12 we made the assumption that the interpretation I has a fixed, standard interpretation for mathematical operators $(+, -, /, *, \leq, \geq, 0, 1, 2, \dots)$. This allows to give the following examples of equivalent term expressions:

- $\texttt{x}_c +_c 0_c \sim \texttt{x}_c$
- $(\texttt{a}_c \,/_c\, 1_c) +_c 4_c \sim \texttt{a}_c +_c 4_c$
- $3_c \geq_c 0_c \sim \text{true}_c$
- $\text{true}_c \,\&_c\, \text{false}_c \sim \text{false}_c$

We now combine term expressions that are equivalent according to Definition 24 in the same *equivalence class*. We introduce these classes so that we can make term expressions evaluate to their corresponding class (Definition 28). This allows us to create rewrite rules between term expressions of the same class, in that way enabling value sensitivity. Similar for formula expressions.

Definition 25 (Equivalence class of term or formula expression)**.** *We define the set $T^0_{\texttt{TExp}/\simeq}$ of equivalence classes on term expressions. An equivalence class $[a] \in T^0_{\texttt{TExp}/\simeq}$ is the set $\{x \in T^0_{\texttt{TExp}} \mid x \sim a\}$. Similar we define the set $P^0_{\texttt{TExp}/\simeq}$ of equivalence classes on formula expressions. An equivalence class $[\varphi] \in P^0_{\texttt{TExp}/\simeq}$ is the set $\{\psi \in P^0_{\texttt{TExp}} \mid \psi \sim \varphi\}$.*

EXAMPLE 30:

The following notations describe the same equivalence class:

- $[\mathtt{y}_c -_c \mathtt{y}_c] = [\mathtt{x}_c -_c \mathtt{x}_c] = [0_c *_c \mathtt{z}_c] = [0_c]$
- $[\mathtt{a}_c *_c (\mathtt{b}_c /_c \mathtt{c}_c)] = [(\mathtt{a}_c /_c \mathtt{c}_c) *_c \mathtt{b}_c]$
- $[\mathtt{x}_c +_c 1_c >_c \mathtt{y}_c] = [\mathtt{x}_c \geq_c \mathtt{y}_c]$
- $[\mathtt{x}_c *_c 1_c >_c \mathtt{x}_c] = [\mathtt{y}_c >_c \mathtt{y}_c] = [\mathrm{false}_c]$

In Definition 18 the dependencies of a program variable `x` under a program `p` are defined. We here define the dependencies of a term t.

Definition 26 (Term dependencies). *The dependencies of a term t form the smallest set $\mathcal{D}(t) \subseteq \mathcal{F}_{n,0}$ of program variables such that the following holds for all first-order structure $M = (\mathsf{D}, I)$ and all logic variable assignments β: if $s_1, s_2 \in \mathcal{S}$ are such that we have $s_1(\mathtt{y}) = s_2(\mathtt{y})$ for all $\mathtt{y} \in \mathcal{D}(t)$, then $val_{M,s_1,\beta}(t) = val_{M,s_2,\beta}(t)$.*

We introduce a helper function *getLocs* that, when provided a term expression or formula expression, returns the set of all the program variables present in the unwrapped version of that term. That is:

$$getLocs(t_{\mathtt{Exp}}) = getVars(unwrap(t_{\mathtt{Exp}}))$$
$$getLocs(\varphi_{\mathtt{Exp}}) = getVars(unwrap(\varphi_{\mathtt{Exp}}))$$

where *getVars* returns all the symbols present in the as argument provided term, that are a member of $\mathcal{F}_{n,0}$. If a term expression is provided to *getLocs* that cannot be unwrapped to a term that is part of the original signature (e.g. because of special symbols introduced for term expressions), the *getLocs* function returns the set of all program variables ($= \mathcal{F}_{n,0}$).

We introduce a choice function $\mathcal{C}$ that is used in the interpretation and evaluation of term (and formula) expressions.

Definition 27 (Choice function). *The function $\mathcal{C} : T^0_{\mathtt{TExp}/\simeq} \to T^0_{\mathtt{TExp}}$ selects a representative r from the provided equivalence class such that getLocs(r) evaluates to the smallest* `LocSet` *of the provided class.*

Note that by the axiom of choice (Zermelo, 1904) such a function always exists.

In the rest of this section, and for the calculus rules introduced in Sections 3.3.2 and 3.3.3, we need the following two lemmas.

Lemma 2. *For any term $t \in [r], [r] \in T^0_{\mathtt{TExp}/\simeq}$, getLocs$(t)$ is a superset of the actual dependencies (Definition 26) of every term unwrap$(t'), t' \in [r]$.*

Proof. We first show that all unwrapped term expressions in $[r]$ have exactly the same dependency set. This is easily proven by contradiction. Assume that there are two term expressions $t, t' \in [r]$ such that $\mathcal{D}(unwrap(t)) \neq \mathcal{D}(unwrap(t'))$. This implies from the definition of term dependencies (Definition 26) that there must be at least one program variable,

say $\mathtt{y}$, for which holds $\mathtt{y} \in \mathcal{D}(\mathit{unwrap}(t)), \mathtt{y} \notin \mathcal{D}(\mathit{unwrap}(t'))$ - or the other way around, but without loss of generality we assume the current case. This means that there must be a pair of states s_1, s_2 which coincide on all program variables except for $\mathtt{y}$, where $s_1(\mathtt{y}) \neq s_2(\mathtt{y})$ and $\mathit{val}_{I,s_1,\beta}(\mathit{unwrap}(t)) \neq \mathit{val}_{I,s_2,\beta}(\mathit{unwrap}(t))$. The evaluation of $\mathit{unwrap}(t')$ on the other hand is identical in both states since this term is not affected by $\mathtt{y}$ by assumption. Thus $\mathit{val}_{I,s_1,\beta}(\mathit{unwrap}(t)) \neq \mathit{val}_{I,s_1,\beta}(\mathit{unwrap}(t'))$ or $\mathit{val}_{I,s_2,\beta}(\mathit{unwrap}(t)) \neq \mathit{val}_{I,s_2,\beta}(\mathit{unwrap}(t'))$. However, this contradicts the requirement for t and t' to be in the same equivalence class (Definition 25). Hence all unwrapped terms in $[r]$ must have exactly the same dependencies.

Note that for term dependencies, a program variable has to be present *in* that term in order to be a dependency. Considering the evaluation of terms (Definition 8), a program variable has no effect unless it is present in that term. As a result the set $\mathit{getLocs}(t), t \in [r]$ contains at least the dependencies of $\mathit{unwrap}(t)$. Combining this with the above proved property that all other unwrapped terms of the same equivalence class have the same dependencies, we have proven this lemma. $\square$

Lemma 3. *For all* $t \in [r], [r] \in T^0_{\mathtt{TExp}/\simeq}$, $\mathcal{D}(\mathit{unwrap}(t)) = \mathit{getLocs}(\mathit{unwrap}(\mathcal{C}([r])))$. *The set of program variables present in the unwrapped term returned by the choice function, is exactly the minimal dependency set (Definition 26) of all term expressions in the corresponding equivalence class.*

Proof. The definition of the choice function gives us that for all terms $t' \in [r]$ independent of $t = \mathcal{C}([r])$, $\mathit{getLocs}(t') \supseteq \mathit{getLocs}(t)$. We show that $\mathit{getLocs}(t)$ is indeed equal to the real dependencies of $\mathit{unwrap}(t)$. From Lemma 2 we already obtain that $\mathit{getLocs}(t)$ is at least a superset of the real dependencies. Therefore if we show that the real dependencies cannot be a real subset of this set, i.e. if we show that $\mathcal{D}(\mathit{unwrap}(t)) \not\subset \mathit{getLocs}(t)$, we have proven the lemma.

We show this by contradiction. Assume that $\mathcal{D}(\mathit{unwrap}(t)) \subset \mathit{getLocs}(t)$. This means that t contains at least one program variable, say $\mathtt{y}$, that has no influence on the value of $\mathit{unwrap}(t)$. Which implies that there exists a term $t' = t[\mathtt{y}_c/c_c]$ where $\mathtt{y}$ is replaced with any constant term expression c_c (e.g. $0_c, 4_c, \dots$), and $\mathit{val}_{M,s,\beta}(\mathit{unwrap}(t')) = \mathit{val}_{M,s,\beta}(\mathit{unwrap}(t))$ for any first-order structure M, state s and logic variable assignment β. Hence t' is an element of $[r]$. However, we can now deduce that $\mathit{getLocs}(t) = \mathit{getLocs}(t') \cup \{\mathtt{y}\}$, thus that $\mathit{getLocs}(t') \subset \mathit{getLocs}(t)$, which contradicts our definition that the choice function $\mathcal{C}$ returns a term expression that when unwrapped contains the least number of program variables. Hence, $\mathcal{D}(\mathit{unwrap}(t)) \not\subset \mathit{getLocs}(t)$ which is what we had to show. $\square$

We evaluate both term and formula expressions to their equivalence class. This allows us to introduce rewrite rules that rewrite between terms and formulas of the same equivalence class. The function $\mathcal{W}$ is required to have a different interpretation on every input. Since the $\mathit{while}(\varphi, t)$ is evaluated as $\mathcal{W}(\varphi, t)$, this ensures that no two *while* terms evaluate to the same equivalence class. No sound rewrite rules can thus be created for *while* terms. This ensures that the implicit dependencies remain present in φ, which is the only reason for which the *while*-constructor is used.

Definition 28 (Domain and interpretation with term expressions)**.** *We only list the interpretation of the symbols not already defined in Definition 19. The concrete domain for term expressions* $\mathsf{D}_{\mathtt{TExp}}$ *is the set* $T^0_{\mathtt{TExp}/\simeq}$*, for location sets* $\mathsf{D}_{\mathtt{LocSet}}$ *it is the powerset of all program variables* $\wp(\mathcal{F}_{n,0})$*. The interpretation I maps the new rigid functions from Definition 21, i.e.*

$f_c \in \mathcal{F}_{\texttt{TExp}} \cup P_{\texttt{TExp}}$ *of arity* n *to a function* $I(f_c) : \left(\mathsf{D}_{\texttt{TExp}}\right)^n \rightarrow \mathsf{D}_{\texttt{TExp}}$. *We only consider interpretations* I *where for all function symbols* f_c*:*

$$\forall e_1, \ldots, e_n \in \mathsf{D}_{\texttt{TExp}} : I(f_c)(e_1, \ldots, e_n) = [f_c(\mathcal{C}(e_1), \ldots, \mathcal{C}(e_n))]$$

Where $\mathcal{C} : T^0_{\texttt{TExp}/\simeq} \rightarrow T^0_{\texttt{TExp}}$ *(thus* $\mathsf{D}_{\texttt{TExp}} \rightarrow T^0_{\texttt{TExp}}$*) is the choice function from Definition 27.*

We require the following property on the interpretation of $\mathcal{W}$*:*

$$\forall p_1, p_2, v_1, v_2.(p_1 \neq p_2 \text{ or } v_1 \neq v_2) : I(\mathcal{W})(p_1, v_1) \neq I(\mathcal{W})(p_2, v_2)$$

The interpretation of the $\dot{\subseteq}_{dep}$ *predicate is:*

$$\forall e \in \mathsf{D}_{\texttt{TExp}}, l \in \mathsf{D}_{\texttt{LocSet}} : I(\dot{\subseteq}_{dep})(e, l) = getLocs(\mathcal{C}(e)) \subseteq l$$

In the definition of $\dot{\subseteq}_{dep}$ we use the property that $\mathcal{C}$ returns the actual (minimal) dependency set of all terms in the provided equivalence class (Lemma 3). This implies that the *getLocs* function applied on the term expression selected by $\mathcal{C}$ always returns the actual dependencies for all terms in that class.

Opposed to that, no matter the representative term selected by $\mathcal{C}$, the interpretation of f_c always gives the same equivalence class. Hence the additional property that $\mathcal{C}$ always returns the term expression with the smallest `LocSet` is not relevant for this particular use.

The interpretation requirement on $\mathcal{W}$ to which the *while* constructor is evaluated, ensures that no two terms containing this constructor are in the same equivalence class, unless the arguments in $while(\varphi, t)$ are exactly the same. This is, as stated before, necessary to include the implicit dependencies of a while-statement.

For the definition of semantics, note that term and formula expressions are entirely built from $\mathcal{F}_{\texttt{TExp}}$ and $P_{\texttt{TExp}}$ (see Definition 23). Therefore their evaluation is already defined with Definition 28, and we can conclude that all $t_{\texttt{Exp}} \in T^0_{\texttt{Exp}}$, $\varphi_{\texttt{Exp}} \in P^0_{\texttt{Exp}}, L \in \texttt{LocSet}$ have the following property:

$$\begin{aligned} val_{M,s,\beta}(t_{\texttt{Exp}}) &= [t_{\texttt{Exp}}] \\ val_{M,s,\beta}(\varphi_{\texttt{Exp}}) &= [\varphi_{\texttt{Exp}}] \\ val'_{M,s,\beta}(t_{\texttt{Exp}} \dot{\subseteq}_{dep} L) &= tt \text{ iff } getLocs(\mathcal{C}(val_{M,s,\beta}(t_{\texttt{Exp}}))) \subseteq val_{M,s,\beta}(L) \end{aligned}$$

Definition 29 (Program Semantics with Dependencies)**.** *We only list the difference in semantics with the one from the dynamic logic defined in Definition 8.*

$$\begin{aligned} val_{M,s,\beta}(while(\varphi, t)) &= \mathcal{W}(val_{M,s,\beta}(\varphi), val_{M,s,\beta}(t)) \\ val'_{M,s,\beta}(\texttt{x = } t) &= \left\{val_{M,s,\beta}\left(\texttt{x} := t \parallel \texttt{x}^{dep} := deps'(t)\right)\right\} \\ val'_{M,s,\beta}(\texttt{p1;p2}) &= \left\{val'_{I,s',\beta}(\texttt{p2}) \mid s' \in val'_{M,s,\beta}(\texttt{p1})\right\} \\ val'_{M,s,\beta}(\textbf{if}(g)\{\texttt{p1}\}\ \textbf{else}\ \{\texttt{p2}\}) &= \begin{cases} S'_1 & \text{if } val_{M,s,\beta}(g) = tt \\ S'_2 & \text{otherwise} \end{cases} \\ &\quad \text{where } S_1 = val'_{M,s,\beta}(\texttt{p1}), S_2 = val'_{M,s,\beta}(\texttt{p2}), \\ &\quad S'_i = \emptyset \text{ iff } S_i = \emptyset, \text{otherwise } S'_i = \{s'_i\} \text{ where} \\ &\quad s'_i(\texttt{x}) = \begin{cases} s_i(\texttt{x}) & \text{if } \texttt{x} \in \mathcal{F}_{n,0} \\ [if_c(\mathcal{C}(val_{M,s,\beta}(deps'(g)))) & \\ then_c(\mathcal{C}(s_1(\texttt{x}))) & \\ else_c(\mathcal{C}(s_2(\texttt{x})))] & \text{otherwise } (\texttt{x} = \texttt{y}^{dep}) \end{cases} \end{aligned}$$

$$val'_{M,s,\beta}(\texttt{while (}g\texttt{) \{p\}}) = \begin{cases} \bigcup_{s'_1 \in S'_1} val'_{I,s'_1,\beta}(\texttt{while(}g\texttt{) \{p\}}) & \text{if } val_{M,s,\beta}(g) = tt \\ \{s\} & \text{otherwise} \end{cases}$$

$$\text{where } S_1 = val'_{M,s,\beta}(\texttt{p}), S'_1 = \emptyset \text{ iff } S_1 = \emptyset,$$

$$\text{otherwise } S'_1 = \{s'_1\} \text{ where}$$

$$s'_1(\texttt{x}) = \begin{cases} s_1(\texttt{x}) & \text{if } \texttt{x} \in \mathcal{F}_{n,0} \text{ or} \\ & \texttt{x} = \texttt{y}^{dep} \text{ and } s_1(\texttt{y}) = s(\texttt{y}) \\ [while_c(& \\ \mathcal{C}(val_{M,s,\beta}(deps'(g))), & \\ \mathcal{C}(s_1(\texttt{x})))] & \text{otherwise} \end{cases}$$

We here make use of the $deps'$ function that, different from the $deps$ function, returns a term or formula expression on a provided term or formula instead of a `LocSet`. We define this function as:

$$\begin{aligned} deps'(f(t_1,\ldots,t_n)) &= f_c(deps'(t_1),\ldots,deps'(t_n)) \\ deps'(\texttt{x}) &= \texttt{x}_c \\ deps'(if(\varphi)then(t_1)else(t_2)) &= if_c(deps'(\varphi))then_c(deps'(t_1))else_c(deps'(t_2)) \\ deps'(a) &= a_c \qquad \text{where } a \in \{\text{true}, \text{false}\} \\ deps'(p(t_1,\ldots,t_n)) &= p_c(deps'(t_1),\ldots,deps'(t_n)) \\ deps'(\varphi_1 * \varphi_2) &= deps'(\varphi_1) *_c deps'(\varphi_2) \qquad \text{where } * \in \{\&, |, \texttt{->}\} \\ deps'(!\,\varphi) &= !_c deps'(\varphi) \\ deps'(t_1 \doteq t_2) &= deps'(t_1) \doteq_c deps'(t_2) \end{aligned}$$

This is the inverse of the $unwrap$ function, that is $unwrap(deps'(t)) = t$. In the evaluation of the conditional statement we no longer make a difference in the semantics whether a program variable has the same value after the execution of the branch as before. Instead, we return the equivalence class of a conditional term expression. The control flow sensitivity benefits from the rewrite rules within this equivalence class, as is described in Section 3.3.3.

Theorem 1 (Dependency correctness)**.** *For every program* p*, program variable* x *and* `LocSet` L*, if* $\bar{\texttt{y}}^{dep} \doteq \bar{\texttt{y}}_c$ `->` $[\texttt{p}]\texttt{x}^{dep} \dot{\subseteq}_{dep} L$ *is valid, then* $\mathcal{D}(\texttt{x},\texttt{p}) \subseteq I(L)$ *holds. In which* $\bar{\texttt{y}}^{dep}$ *are the dependency variables associated with* $\bar{\texttt{y}}$*,* $\bar{\texttt{y}}_c$ *the constant term expressions for the non-rigid function symbols* $\bar{\texttt{y}}$*, and* $\bar{\texttt{y}}$ *is a list of all program variables present in program* p*.*

Proof. (*sketch*) We do not consider interpretations where $\bar{\texttt{y}}^{dep} \neq \bar{\texttt{y}}_c$ since then the statement holds trivially.

If $val_{M,s,\beta}(\texttt{p}) = \emptyset$ for any first-order structure M, state s and logic variable assignment β the conclusion of this theorem automatically holds, so we assume that $val_{M,s,\beta}(\texttt{p}) = \{s_1\}$.

We can rewrite the theorem above to the statement "if $getLocs(\mathcal{C}(val_{I,s_1,\beta}(\texttt{x}^{dep}))) \subseteq val_{I,s_1,\beta}(L)$ holds under the assumption that $val_{M,s,\beta}(\texttt{y}^{dep}) = val_{M,s,\beta}(\texttt{y}_c)$ for all y in $\bar{\texttt{y}}$, then $\mathcal{D}(\texttt{x},\texttt{p}) \subseteq I(L)$ holds". Since $val_{M,s,\beta}(L) = I(L)$, we have to show that $\mathcal{D}(\texttt{x},\texttt{p}) \subseteq getLocs(\mathcal{C}(val_{I,s_1,\beta}(\texttt{x}^{dep})))$ which by transitivity of $\subseteq$ closes the proof.

To prove this we show that for each program evaluation step, $\texttt{x}^{dep}$ is updated to an equivalence class $[r]$ such that $getLocs(\mathcal{C}([r]))$ contains at least all the dependencies of x on that point in the program evaluation. The evaluations of interest are those of assignment, conditional and loop statements.

The semantics for assignments gives us that the term to which x is updated is a member of the equivalence class to which $\mathtt{x}^{dep}$ is updated.

By induction we have for the conditional statement that both the *then*-branch $\mathtt{p}_1$ and the *else*-branch $\mathtt{p}_2$ over-approximate the dependencies of changed variables. Each variable $\mathtt{x}^{dep}$ is updated to something similar to $[\mathit{if}_c(\varphi)\mathit{then}_c(t_1)\mathit{else}_c(t_2)]$, where φ, t_1 and t_2 are terms containing at least the dependencies of the guard, *then*- and *else*-branch result respectively. Hence, the *getLocs* function (Page 71) returns on this equivalence class a set of program variables which is a superset of the actual dependencies.

For the loop statement we have by induction that the dependencies are over-approximated when the while loop's guard condition evaluates to *tt*. In case the condition evaluates to *ff*, the dependencies of each variable whose value is changed in the while loop, is updated. It is updated to something similar to $[\mathit{while}_c(\varphi, t)]$, where φ and t are terms containing at least the dependencies of the guard and loop's body respectively. Again the *getLocs* function returns on this equivalence class a set of program variables which is a superset of the actual dependencies.

These three statements may be combined with the $\mathtt{p}_1;\mathtt{p}_2$ constructor, where $\mathtt{p}_2$ is evaluated in the state resulting from $\mathtt{p}_1$, thus correctly preserving the dependencies. This implies that $val_{I,s_1,\beta}(\mathtt{x}^{dep}) = [r]$ and the term $\mathcal{C}([r])$ contains at least all dependencies of x under p, which is what we had to show. □

EXAMPLE 31:

As an example we evaluate the formula $\mathtt{x}_c -_c \mathtt{x}_c \dot{\subseteq}_{dep} \mathtt{\{\}}$:

$$\begin{aligned} val_{M,s,\beta}(\mathtt{x}_c -_c \mathtt{x}_c \dot{\subseteq}_{dep} \mathtt{\{\}}) = \; & tt \text{ iff } \mathit{getLocs}(\mathcal{C}(val_{M,s,\beta}(\mathtt{x}_c -_c \mathtt{x}_c))) \subseteq val_{M,s,\beta}(\mathtt{\{\}}) \\ & tt \text{ iff } \mathit{getLocs}(\mathcal{C}([\mathtt{x}_c -_c \mathtt{x}_c])) \subseteq \{\} \\ & tt \text{ iff } \mathit{getLocs}(0_c) \subseteq \{\} \\ & tt \text{ iff } \{\} \subseteq \{\} \\ & tt \end{aligned}$$

3.3.2. Value Sensitivity

In the definitions of syntax and semantics in the previous section, variables of the kind $\mathtt{x}^{dep}$ are updated to a term of type `TExp`. Since we prefer to reason on non-interference using location sets, i.e. terms of type `LocSet` instead of the equivalence class to which $\mathtt{x}^{dep}$ evaluates, we introduce the following calculus rule:

$$\mathsf{expressionToLocSet}\ \frac{\Gamma \Longrightarrow \mathit{getLocs}(t)\dot{\subseteq}L, \Delta}{\Gamma \Longrightarrow t\dot{\subseteq}_{dep}L, \Delta}$$

Note that both formulas are almost equal; except that $\dot{\subseteq}_{dep}$ uses the smallest set of dependencies of equivalence class $[t]$ (using the choice function $\mathcal{C}$), while *getLocs* returns a set that is bigger or equal to that set. Therefore if the premise holds, then so does the conclusion of the rule.

The $\mathsf{assignment}^{\mathsf{dep}}$ rule from Section 3.2.2 is left unchanged, except that we now use the deps' function instead of *deps*:

$$\textsf{assignment}^{\textsf{dep}} \; \frac{\Gamma \Longrightarrow \{\mathcal{U}\}\{\mathtt{x} := t \,\|\, \mathtt{x}^{dep} := deps'(\mathtt{t})\}[\ldots]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathtt{x}\ \mathtt{=}\ t\mathtt{;}\ \ldots]\varphi, \Delta}$$

To achieve value sensitivity note that a term expression $t_{\mathtt{Exp}}$ is evaluated to an equivalence class. This implies that we can freely rewrite between terms that are equivalent according to Definition 24, without altering the value of those terms. A large number of term rewriting rules for regular terms can be adopted for term expressions. Here we list a number of them[1]:

$$\begin{aligned}
t_{\mathtt{Exp}} +_c v_{\mathtt{Exp}} &\leadsto v_{\mathtt{Exp}} +_c t_{\mathtt{Exp}} \\
t_{\mathtt{Exp}} *_c v_{\mathtt{Exp}} &\leadsto v_{\mathtt{Exp}} *_c t_{\mathtt{Exp}} \\
t_{\mathtt{Exp}} -_c t_{\mathtt{Exp}} &\leadsto 0_c \\
1_c *_c t_{\mathtt{Exp}} &\leadsto t_{\mathtt{Exp}} \\
0_c *_c t_{\mathtt{Exp}} &\leadsto 0_c \\
i_c +_c j_c &\leadsto k_c \quad \text{where } i, j, k \in \mathbb{Z}, k = i + j \\
t_c *_c (v_c \,/_c\, w_c) &\leadsto (t_c *_c v_c) \,/_c\, w_c
\end{aligned}$$

There is no need to convert all rules for arithmetic. Certain rules can be left out for simplification of the proving process, which needs to be balanced against the resulting loss of completeness.

Lemma 2 gives us that the variables present in the unwrapped version of these terms is always a superset of the actual dependencies. These rewrite rules now give us the value sensitivity we seek.

EXAMPLE 32:

We start with the simple program `l = h - h` in the following sequent:

$$\mathtt{l}^{dep} \doteq \mathtt{l}_c, \mathtt{h}^{dep} \doteq \mathtt{h}_c \Longrightarrow [\mathtt{l=h-h}](\mathtt{l}^{dep} \,\dot{\subseteq}_{dep}\, \mathtt{\{l\}})$$

We perform the $\textsf{assignment}^{\textsf{dep}}$ rule and obtain:

$$\Gamma \Longrightarrow \{\mathtt{l} := \mathtt{h} - \mathtt{h} \,\|\, \mathtt{l}^{dep} := \mathtt{h}^{dep} -_c \mathtt{h}^{dep}\}(\mathtt{l}^{dep} \,\dot{\subseteq}_{dep}\, \mathtt{\{l\}})$$

We can apply this update and use the equivalence relations in Γ to obtain:

$$\Gamma \Longrightarrow \mathtt{h}_c -_c \mathtt{h}_c \,\dot{\subseteq}_{dep}\, \mathtt{\{l\}}$$

If we now apply the expressionToLocSet rule, we get $\Gamma \Longrightarrow \mathtt{\{h\}} \,\dot{\subseteq}\, \mathtt{\{l\}}$, which we cannot prove. Instead, we first use the rewrite rule $t_{\mathtt{Exp}} -_c t_{\mathtt{Exp}} \leadsto 0_c$, to replace the term expression with another expression from the same equivalence class, giving us:

$$\Gamma \Longrightarrow 0_c \,\dot{\subseteq}_{dep}\, \mathtt{\{l\}}$$

Now we apply the expressionToLocSet rule, and get:

$$\Gamma \Longrightarrow \mathtt{\{\}} \,\dot{\subseteq}\, \mathtt{\{l\}}$$

[1] The rewrite rule $t_{\mathtt{Exp}} \,/_c\, t_{\mathtt{Exp}} \leadsto 1_c$ is not a sound rule, since these terms do not belong to the same equivalence class in a state s with $s(\mathtt{x}) = 0$.

which holds trivially.

EXAMPLE 33:

Consider `p` to be the program `l = h * (0 / 5)` in the sequent:

$$\mathtt{l}^{dep} \doteq \mathtt{l}_c, \mathtt{h}^{dep} \doteq \mathtt{h}_c \Longrightarrow [\mathtt{p}](\mathtt{l}^{dep} \dot{\subseteq}_{dep} \mathtt{\{l\}})$$

We apply the $\mathsf{assignment}^{\mathsf{dep}}$ rule and obtain:

$$\Gamma \Longrightarrow \{\mathtt{l} := \mathtt{h} * (0/5) \parallel \mathtt{l}^{dep} := \mathtt{h}^{dep} *_c (0_c \,/_c\, 5_c)\}(\mathtt{l}^{dep} \dot{\subseteq}_{dep} \mathtt{\{l\}})$$

Applying this update and rewriting equalities from Γ gives us:

$$\Gamma \Longrightarrow \mathtt{h}_c *_c (0_c \,/_c\, 5_c) \dot{\subseteq}_{dep} \mathtt{\{l\}}$$

We now use the rewrite rule $t_c *_c (v_c \,/_c\, w_c) \rightsquigarrow (t_c *_c v_c) \,/_c\, w_c$:

$$\Gamma \Longrightarrow (\mathtt{h}_c *_c 0_c) \,/_c\, 5_c \dot{\subseteq}_{dep} \mathtt{\{l\}}$$

The rewrite rule $t_{\mathtt{Exp}} *_c v_{\mathtt{Exp}} \rightsquigarrow v_{\mathtt{Exp}} *_c t_{\mathtt{Exp}}$:

$$\Gamma \Longrightarrow (0_c *_c \mathtt{h}_c) \,/_c\, 5_c \dot{\subseteq}_{dep} \mathtt{\{l\}}$$

And $0_c *_c t_{\mathtt{Exp}} \rightsquigarrow 0_c$:

$$\Gamma \Longrightarrow 0_c \,/_c\, 5_c \dot{\subseteq}_{dep} \mathtt{\{l\}}$$

We could reduce this term expression further, but we can also directly apply the expressionToLocSet rule and obtain:

$$\Gamma \Longrightarrow \mathtt{\{\}} \dot{\subseteq} \mathtt{\{l\}}$$

which holds trivially.

3.3.3. Control Flow Sensitivity

The problem in the approach outlined in 3.2 with respect to flow sensitivity is that the proof tree is split into two separate branches as soon as an conditional statement is symbolically executed. In the rule below we postpone this branching which allows us to have a better control flow sensitivity. Note that in the following rule the succedent of the premise is a formula that contains multiple programs placed in box operators.

$$\mathsf{ifElse}^{\mathsf{dep}} \; \frac{\Gamma \Longrightarrow \{\mathcal{U}\}\{\bar{\mathtt{x}}^{pre} := \bar{\mathtt{x}}\}[\mathtt{p1;\ } \bar{\mathtt{x}}^{t} \mathtt{\ =\ } \bar{\mathtt{x}}]\{\bar{\mathtt{x}} := \bar{\mathtt{x}}^{pre}\}[\mathtt{p2;\ } \bar{\mathtt{x}}^{e} \mathtt{\ =\ } \bar{\mathtt{x}}]\{\bar{\mathtt{x}} := \bar{\mathtt{x}}^{pre}\}\{\mathcal{V}\}[\ldots]\varphi.\Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathtt{if(}g\mathtt{)\{p1\}else\{p2\};\ }\ldots]\varphi, \Delta}$$

where

- $\bar{\mathtt{x}} = (\mathtt{x}_1, \mathtt{x}_1^{dep}, \ldots, \mathtt{x}_n, \mathtt{x}_n^{dep})$ are all variables changed in `p1` or `p2` with their corresponding dependencies.

- $\bar{\mathtt{x}}^{pre} = (\mathtt{x}_1^{pre}, \mathtt{x}_1^{predep}, \ldots, \mathtt{x}_n^{pre}, \mathtt{x}_n^{predep})$ a list of fresh variables, the same length as $\bar{\mathtt{x}}$.
- $\bar{\mathtt{x}}^t = (\mathtt{x}_1^t, \ldots, \mathtt{x}_n^t)$ and $\bar{\mathtt{x}}^e = (\mathtt{x}_1^e, \ldots, \mathtt{x}_n^e)$ are fresh variables, half the length of $\bar{\mathtt{x}}$.
- Update $\mathcal{V}$ is the collection of parallel updates for all $\mathtt{x}_i \in \bar{\mathtt{x}}$:

$$\mathtt{x}_i := \mathit{if}(g)\mathit{then}(\mathtt{x}_i^t)\mathit{else}(\mathtt{x}_i^e) \parallel$$
$$\mathtt{x}_i{}^{dep} := \mathit{if}_c(\mathit{deps}'(g))\mathit{then}_c(\mathtt{x}_i^{t\,dep})\mathit{else}_c(\mathtt{x}_i^{t\,dep})$$

- Vector notation is used in the changes to the program as well. $\bar{\mathtt{x}}^t$ `=` $\bar{\mathtt{x}}$ is short for $\mathtt{x}_1^t$ `=` $\mathtt{x}_1$`;` `...;` $\mathtt{x}_n^t$ `=` $\mathtt{x}_n$, similar for other vector notations.

The idea behind this rule is as follows. When encountering an conditional statement we first store the current value of all variables $\bar{\mathtt{x}}$ in the variables $\bar{\mathtt{x}}^{pre}$, thus $\mathtt{x}^{pre}$ holding the pre-state of each variable `x` with respect to the conditional statement.

We execute the then-part of the statement and store the outcome for each variable `x` in $\mathtt{x}^t$. This implies that the corresponding term expressions (due to the $\mathsf{assignment}^{\mathsf{dep}}$ rule) are stored in each $\mathtt{x}^{t\,dep}$. All variables $\bar{\mathtt{x}}$ are then reset to their initial value just before the symbolic execution of the conditional statement, and the else-part is executed. The outcome of this part (thus independently from the then-part of the branch) for each variable `x` is stored in $\mathtt{x}^e$.

We then again reset the values of variables $\bar{\mathtt{x}}$ to their value in the pre-state, ensuring that g and $\mathit{deps}'(g)$ are evaluated the same as in the state before symbolic execution of the conditional statement. The update $\mathcal{V}$ represents the real state transition caused by the conditional statement. Note that due to the last-one-wins semantics for updates, $\mathcal{V}$ overwrites all the effects caused by symbolic execution of both the branches with respect to the variables in $\bar{\mathtt{x}}$. For each variable `x`, we know that depending on the condition g its value is now either $\mathtt{x}^t$ or $\mathtt{x}^e$. The dependency variable of each `x` is updated to the corresponding term expression.

Similar to term expressions, formula expressions are evaluated to their equivalence class. We can therefore also freely rewrite between formulas that are equivalent according to Definition 24, just as we did with term expressions in Section 3.3.2. Some of the rewrite rules include:

$$i_c >_c 0_c \rightsquigarrow \mathrm{true}_c \quad \text{where } i \in \mathbb{Z}, i > 0$$
$$t_{\mathtt{Exp}} +_c i_c \geq_c t_{\mathtt{Exp}} \rightsquigarrow \mathrm{true}_c \quad \text{where } i \in \mathbb{Z}, i \geq 0$$
$$t_{\mathtt{Exp}} /_c v_{\mathtt{Exp}} \doteq_c w_{\mathtt{Exp}} \rightsquigarrow t_{\mathtt{Exp}} \doteq_c v_{\mathtt{Exp}} *_c w_{\mathtt{Exp}}$$

To obtain control flow sensitivity we add some more rewrite rules between equivalent term expressions related to the conditional terms:

$$\mathit{if}_c(\varphi_{\mathtt{Exp}})\mathit{then}_c(t_{\mathtt{Exp}})\mathit{else}_c(t_{\mathtt{Exp}}) \rightsquigarrow t_{\mathtt{Exp}}$$
$$\mathit{if}_c(\mathrm{true}_c)\mathit{then}_c(t_{\mathtt{Exp}})\mathit{else}_c(v_{\mathtt{Exp}}) \rightsquigarrow t_{\mathtt{Exp}}$$
$$\mathit{if}_c(\mathrm{false}_c)\mathit{then}_c(t_{\mathtt{Exp}})\mathit{else}_c(v_{\mathtt{Exp}}) \rightsquigarrow v_{\mathtt{Exp}}$$

Example 34:

We use program (iv) from Example 22 in the following sequent:

$$\mathtt{h}^{dep} \doteq \mathtt{h}_c, \mathtt{l}^{dep} \doteq \mathtt{l}_c \Longrightarrow [\mathtt{if(h>0)\{l=2\}else\{l=2\}}]\mathtt{l}^{dep} \dot{\subseteq}_{dep} \mathtt{\{l\}}$$

Applying the $\mathsf{ifElse^{dep}}$ rule gives us (note that $\bar{\mathtt{x}} = \{\mathtt{l}, \mathtt{l}^{dep}\}$):

$$\begin{aligned}\Gamma \Longrightarrow\ & \{\mathtt{l}^{pre} := \mathtt{l} \parallel \mathtt{l}^{predep} := \mathtt{l}^{dep}\}[\mathtt{l=2;l}^{t}\mathtt{=l}]\\ & \{\mathtt{l} := \mathtt{l}^{pre} \parallel \mathtt{l}^{dep} := \mathtt{l}^{predep}\}[\mathtt{l=2;l}^{e}\mathtt{=l}]\\ & \{\mathtt{l} := \mathtt{l}^{pre} \parallel \mathtt{l}^{dep} := \mathtt{l}^{predep}\}\{\mathcal{V}\}\varphi\end{aligned}$$

Symbolic execution of the first program box and parallelization[2] gives us:

$$\begin{aligned}\Gamma \Longrightarrow\ & \{\mathtt{l}^{pre} := \mathtt{l} \parallel \mathtt{l}^{predep} := \mathtt{l}^{dep} \parallel \mathtt{l} := 2 \parallel \mathtt{l}^{dep} := 2_c \parallel \mathtt{l}^{t} := 2 \parallel \mathtt{l}^{t\,dep} := 2_c\}\\ & \{\mathtt{l} := \mathtt{l}^{pre} \parallel \mathtt{l}^{dep} := \mathtt{l}^{predep}\}[\mathtt{l=2;l}^{e}\mathtt{=l}]\\ & \{\mathtt{l} := \mathtt{l}^{pre} \parallel \mathtt{l}^{dep} := \mathtt{l}^{predep}\}\{\mathcal{V}\}\varphi\end{aligned}$$

We also parallelize the two updates at the start of the succedent, giving:

$$\begin{aligned}\Gamma \Longrightarrow\ & \{\mathtt{l}^{pre} := \mathtt{l} \parallel \mathtt{l}^{predep} := \mathtt{l}^{dep} \parallel \mathtt{l} := \mathtt{l} \parallel \mathtt{l}^{dep} := \mathtt{l}^{dep} \parallel \mathtt{l}^{t} := 2 \parallel \mathtt{l}^{t\,dep} := 2_c\}\\ & [\mathtt{l=2;l}^{e}\mathtt{=l}]\\ & \{\mathtt{l} := \mathtt{l}^{pre} \parallel \mathtt{l}^{dep} := \mathtt{l}^{predep}\}\{\mathcal{V}\}\varphi\end{aligned}$$

Symbolic execution of the second program box and parallelization with the preceding update, gives us:

$$\begin{aligned}\Gamma \Longrightarrow\ & \{\mathtt{l}^{pre} := \mathtt{l} \parallel \mathtt{l}^{predep} := \mathtt{l}^{dep} \parallel \mathtt{l} := 2 \parallel \mathtt{l}^{dep} := 2_c \parallel\\ & \quad \mathtt{l}^{t} := 2 \parallel \mathtt{l}^{t\,dep} := 2_c \parallel \mathtt{l}^{e} := 2 \parallel \mathtt{l}^{e\,dep} := 2_c\}\\ & \{\mathtt{l} := \mathtt{l}^{pre} \parallel \mathtt{l}^{dep} := \mathtt{l}^{predep}\}\{\mathcal{V}\}\varphi\end{aligned}$$

Parallelizing the two updates before $\mathcal{V}$ and replacing $\mathcal{V}$ with its definition, now gives us:

$$\begin{aligned}\Gamma \Longrightarrow\ & \{\mathtt{l}^{pre} := \mathtt{l} \parallel \mathtt{l}^{predep} := \mathtt{l}^{dep} \parallel \mathtt{l} := \mathtt{l} \parallel \mathtt{l}^{dep} := \mathtt{l}^{dep} \parallel\\ & \quad \mathtt{l}^{t} := 2 \parallel \mathtt{l}^{t\,dep} := 2_c \parallel \mathtt{l}^{e} := 2 \parallel \mathtt{l}^{e\,dep} := 2_c\}\\ & \{\mathtt{l} := \mathit{if}(\mathtt{h} > 0)\mathit{then}(\mathtt{l}^{t})\mathit{else}(\mathtt{l}^{e}) \parallel\\ & \quad \mathtt{l}^{dep} := \mathit{if}_c(\mathtt{h}^{dep} >_c 0_c)\mathit{then}_c(\mathtt{l}^{t\,dep})\mathit{else}_c(\mathtt{l}^{e\,dep})\}\varphi\end{aligned}$$

And with parallelization we obtain:

$$\begin{aligned}\Gamma \Longrightarrow\ & \{\mathtt{l}^{pre} := \mathtt{l} \parallel \mathtt{l}^{predep} := \mathtt{l}^{dep} \parallel \mathtt{l} := \mathtt{l} \parallel \mathtt{l}^{dep} := \mathtt{l}^{dep} \parallel\\ & \quad \mathtt{l}^{t} := 2 \parallel \mathtt{l}^{t\,dep} := 2_c \parallel \mathtt{l}^{e} := 2 \parallel \mathtt{l}^{e\,dep} := 2_c \parallel\\ & \quad \mathtt{l} := \mathit{if}(\mathtt{h} > 0)\mathit{then}(2)\mathit{else}(2) \parallel\\ & \quad \mathtt{l}^{dep} := \mathit{if}_c(\mathtt{h}^{dep} >_c 0_c)\mathit{then}_c(2_c)\mathit{else}_c(2_c)\}\varphi\end{aligned}$$

We apply this update on φ and get using the equalities in Γ:

$$\Gamma \Longrightarrow \mathit{if}_c(\mathtt{h}_c >_c 0_c)\mathit{then}_c(2_c)\mathit{else}_c(2_c) \mathrel{\dot{\subseteq}_{dep}} \{\mathtt{l}\}$$

[2]Note that for a shorter notation we omit elementary updates that are later overruled by the last-one wins semantics

We can now use the rewrite rule $if_c(\varphi_{\texttt{Exp}})then_c(t_{\texttt{Exp}})else_c(t_{\texttt{Exp}}) \rightsquigarrow t_{\texttt{Exp}}$:

$$\Gamma \Longrightarrow 2_c \mathrel{\dot{\subseteq}}_{dep} \texttt{\{l\}}$$

Followed by the expressionToLocSet rule:

$$\Gamma \Longrightarrow \texttt{\{\}} \mathrel{\dot{\subseteq}} \texttt{\{l\}}$$

Which holds.

Similar, we update the loopUnwind and invariantUpdate rules from the original extension for dependencies to:

$$\text{loopUnwind}\ \frac{\begin{array}{l}\Gamma, \{\mathcal{U}\}g, \{\mathcal{U}\}(\bar{x} \doteq \bar{x}^{pre}) \Longrightarrow \{\mathcal{U}\}[\texttt{p}]\{\mathcal{V}\}[\texttt{while (}t\texttt{) \{p\}; }\ldots]\varphi, \Delta \\ \Gamma, \{\mathcal{U}\}\,!\,g \Longrightarrow \{\mathcal{U}\}[\ldots]\varphi, \Delta\end{array}}{\Gamma, \Longrightarrow \{\mathcal{U}\}[\texttt{while (}t\texttt{) \{p\}; }\ldots]\varphi, \Delta}$$

$$\text{invariantUpdate}^{\text{dep}}\ \frac{\begin{array}{l}\Gamma, \{\mathcal{U}\}(\bar{\texttt{x}} \doteq \bar{c}) \Longrightarrow \exists \bar{\gamma}.\{\mathcal{U}'\}(\bar{\texttt{x}} \doteq \bar{c}), \Delta \\ \Gamma, \{\mathcal{U}'\}g,\ \{\mathcal{U}'\}(\bar{\texttt{x}} \doteq \bar{x}^{pre}),\ \{\mathcal{U}'\}[\texttt{p}]\{\mathcal{V}\}(\bar{\texttt{x}} \doteq \bar{c}) \Longrightarrow \exists \bar{\gamma}.\{\mathcal{U}'\}(\bar{\texttt{x}} \doteq \bar{c}), \Delta \\ \Gamma, \{\mathcal{U}'\}\,!\,g \Longrightarrow \{\mathcal{U}'\}[\ldots]\varphi, \Delta\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\texttt{while (}g\texttt{) \{p\}; }\ldots]\varphi, \Delta}$$

where $\mathcal{V}$ is the update:

$$\texttt{x}_i^{dep} := deps'(if(\texttt{x}_i \doteq \texttt{x}_i^{pre})then(\texttt{x}_i)else(while(g, \texttt{x}_i))$$

for all $\texttt{x}_i \in \bar{\texttt{x}}$.

Again we need to change some parts in the join procedure. In the automatic search for a fixed point, we have to join in the abstract lattice not using $\texttt{x}^{dep}$ but using $getLocs(\texttt{x}^{dep})$. The abstract element returned (e.g. High) is used in the term to which $\texttt{x}^{dep}$ is updated in the joined update, however $\texttt{x}^{dep}$ is not updated to (e.g.) $\gamma_{\textsf{High},z}$ but to $\gamma_{\textsf{High},z_c}$. Since these are standard non-rigid functions, the *unwrap* function returns the normal term $\gamma_{a,z}$ when supplied a term expressions γ_{a,z_c}. When this unwrapped $\gamma_{a,z}$ term is supplied to the *getVars* function, it returns the set a. E.g.:

$$getLocs(\gamma_{\textsf{High},z_c}) = getVars(\gamma_{\textsf{High},z}) = \textsf{High}$$

where High is the set of all program variables that have the security label High.

EXAMPLE 35:

We now show how to prove the non-interference property for program (viii) from Example 22.

$$\texttt{l}^{dep} \doteq \texttt{l}_c, \texttt{h}^{dep} \doteq \texttt{h}_c \Longrightarrow [\texttt{if(h-h+l>0)\{l=0\}else\{l=2\}}](\texttt{l}^{dep} \mathrel{\dot{\subseteq}}_{dep} \texttt{\{l\}})$$

Symbolic execution of the conditional statement, gives us:

$$\begin{aligned}\Gamma \Longrightarrow\ & \{\texttt{l}^{pre} := \texttt{l} \parallel \texttt{l}^{predep} := \texttt{l}^{dep}\}[\texttt{l=0;l}^{t}\texttt{=l}] \\ & \{\texttt{l} := \texttt{l}^{pre} \parallel \texttt{l}^{dep} := \texttt{l}^{predep}\}[\texttt{l=2;l}^{e}\texttt{=l}] \\ & \{\texttt{l} := \texttt{l}^{pre} \parallel \texttt{l}^{dep} := \texttt{l}^{predep}\}\{\mathcal{V}\}(\texttt{l}^{dep} \mathrel{\dot{\subseteq}}_{dep} \texttt{\{l\}})\end{aligned}$$

Update parallelization (and unfolding of $\mathcal{V}$) gives us:

$$\Gamma \Longrightarrow \{\mathtt{l}^{pre} := \mathtt{l} \parallel \mathtt{l}^{predep} := \mathtt{l}^{dep} \parallel \mathtt{l}^{t} := 0 \parallel \mathtt{l}^{tdep} := 0_c \parallel \mathtt{l}^{e} := 2 \parallel \mathtt{l}^{edep} := 2_c \parallel$$
$$\mathtt{l} := \mathit{if}(\mathtt{h} - \mathtt{h} + \mathtt{l})\mathit{then}(0)\mathit{else}(2) \parallel \mathtt{l}^{dep} := \mathit{if}_c(\mathtt{h}^{dep} -_c \mathtt{h}^{dep} +_c \mathtt{l}^{dep})\mathit{then}_c(0_c)\mathit{else}_c(2_c)\}$$
$$(\mathtt{l}^{dep} \dot{\subseteq}_{dep} \{\mathtt{l}\})$$

We apply this update on the predicate and get:

$$\mathtt{l}^{dep} \doteq \mathtt{l}_c, \mathtt{h}^{dep} \doteq \mathtt{h}_c \Longrightarrow \mathit{if}_c(\mathtt{h}^{dep} -_c \mathtt{h}^{dep} +_c \mathtt{l}^{dep})\mathit{then}_c(0_c)\mathit{else}_c(2_c) \dot{\subseteq}_{dep} \{\mathtt{l}\}$$

From Section 3.3.2 have the rewrite rule $t_{\mathtt{Exp}} -_c t_{\mathtt{Exp}} \rightsquigarrow 0_c$ which we apply here with $t_{\mathtt{Exp}} = \mathtt{h}^{dep}$:

$$\mathtt{l}^{dep} \doteq \mathtt{l}_c, \mathtt{h}^{dep} \doteq \mathtt{h}_c \Longrightarrow \mathit{if}_c(0_c +_c \mathtt{l}^{dep})\mathit{then}_c(0_c)\mathit{else}_c(2_c) \dot{\subseteq}_{dep} \{\mathtt{l}\}$$

And using the equalities from Γ:

$$\mathtt{l}^{dep} \doteq \mathtt{l}_c, \mathtt{h}^{dep} \doteq \mathtt{h}_c \Longrightarrow \mathit{if}_c(0_c +_c \mathtt{l}_c)\mathit{then}_c(0_c)\mathit{else}_c(2_c) \dot{\subseteq}_{dep} \{\mathtt{l}\}$$

Further rewrite rules could not reduce the number of variables in the term, so we apply the expressionToLocSet rule and obtain:

$$\mathtt{l}^{dep} \doteq \mathtt{l}_c, \mathtt{h}^{dep} \doteq \mathtt{h}_c \Longrightarrow \{\mathtt{l}\} \dot{\subseteq} \{\mathtt{l}\}$$

Which holds.

3.4. Other Approaches

During the research for the value sensitive information flow analysis as described in Section 3.3, a number of other approaches had been considered as well. Despite the problems that they presented we describe two of them in this section because the underlying concepts may be used in further extensions to this framework. We describe the approaches without going into much detail and point out why these approaches were abandoned.

3.4.1. Dependency Tracking with Labels

In one of the earlier approaches we introduced the notion of *labels* to the logic. This allowed us to label a variable (or, in one of the variations: whole terms). In a normal setting, a program variable `x` is updated to a term t representing the value of `x`. With the addition of labels we keep track of which dependencies occurring in $\mathtt{x}^{dep}$ were added because of which sub-term in t.

For example, the assignment `x = y` results in the update $\mathtt{x} := [l_0]\mathtt{y}$, where l_0 is a fresh label. Assuming that at the initial state before program execution each program variable depends only on itself, e.g. $\mathtt{y}^{dep} \doteq \{[]\mathtt{y}\}$ (where $[]$ denotes an empty list of labels), we update the dependencies of `x` simultaneously: $\mathtt{x}^{dep} := [l_0]\mathtt{y}^{dep}$. This becomes, using information from the antecedent: $\mathtt{x}^{dep} := \{[l_0]\mathtt{y}\}$. Several different approaches to introduce the labeled dependencies were considered as well, but we do not discuss them here. After the symbolic execution of a second assignment, e.g. `x = x - y + w`, we get after some simplification:

$$\{\mathtt{x} := [l_0]\mathtt{y} \parallel \mathtt{x}^{dep} := \{[l_0]\mathtt{y}\} \parallel \mathtt{x} := [l_0]\mathtt{y} - [l_2]\mathtt{y} + [l_3]\mathtt{w} \parallel \mathtt{x}^{dep} := \{[l_0, l_1, l_2]\mathtt{y}, [l_3]\mathtt{w}\}\}$$

To achieve value sensitivity, several rewrite rules are introduced. First of all we allow for the removal of labels from dependencies of a program variable, if that label is not present in the term to which the program variable in question is updated. The argumentation behind this step is that a label indicates the introduction of a dependency. Thus, if a label is not present in a term, then the dependency it introduced is somehow gone. Such a rewrite rule would allow us to remove the label l_1 in our example, that disappeared during the parallelization step. Again, different approaches were tried in which the label l_1 would have remained present in the term, but this gave rise to the same problems as are discussed later in this section. Omitting the elementary updates having no effect due to the last-one wins semantics, we thus get in our example:

$$\{\mathtt{x} := [l_0]\mathtt{y} - [l_2]\mathtt{y} + [l_3]\mathtt{w} \parallel \mathtt{x}^{dep} := \mathtt{\{}[l_0, l_2]\mathtt{y}, [l_3]\mathtt{w}\mathtt{\}}\}$$

The rule that really enables the value-sensitivity part of this approach, is one that allows for the removal of labels from a variable in a term, if the value of this term is the same regardless the value of that variable. After all, if variable has no influence on the term, it is not a dependency of that term. In our example, we can use this rule to remove the labels from variable y:

$$\{\mathtt{x} := []\mathtt{y} - []\mathtt{y} + [l_3]\mathtt{w} \parallel \mathtt{x}^{dep} := \mathtt{\{}[l_0, l_2]\mathtt{y}, [l_3]\mathtt{w}\mathtt{\}}\}$$

We can now again use the rule to remove labels from the dependency variables no longer present in the term to which x is updated:

$$\{\mathtt{x} := []\mathtt{y} - []\mathtt{y} + [l_3]\mathtt{w} \parallel \mathtt{x}^{dep} := \mathtt{\{}[]\mathtt{y}, [l_3]\mathtt{w}\mathtt{\}}\}$$

As a final rewrite rule, we now note that if a program variable dependency has no labels, apparently all introductions of this dependency have been removed, allowing us to remove that dependency entirely. This can in our example be used to remove the dependency on y:

$$\{\mathtt{x} := []\mathtt{y} - []\mathtt{y} + [l_3]\mathtt{w} \parallel \mathtt{x}^{dep} := \mathtt{\{}[l_3]\mathtt{w}\mathtt{\}}\}$$

The exact definition of those rewrite rules differed between various approaches but the conceptual idea described here remained the same.

Problems arose on different parts in the formalization of this approach, of which the most common ones we list here:

- When introducing the conditional or loop statement, we notice that the implicit dependencies possibly arising from the associated condition are not added to the term to which a variable (e.g. x) is updated. The update to x is only concerned with the explicit dependencies. Therefore, if we add a labeled implicit dependency variable to $\mathtt{x}^{dep}$, the rewrite rules allow for immediate removal of this dependency since its labels do not occur in the update to x. As a solution to that issue, approached were tried to label whole terms with the label of implicit dependencies. This complicated definition but was not a problem that prevented a successful formalization.

- A bigger issue formed the formal definition of the semantics of labels, labeled terms and labeled dependencies. The evaluation of a labeled term should become a tuple of both the label-value and the actual value, and functions need to be defined as operating on this tuples instead of only values. A different option is to include, similar to a logic variable assignment β or state s, a label assignment function to the evaluation of terms and formulas. Again, this is not a problem that prevents successful definition of the concept of labels, but complicated the definition and indirectly the other issues as well.

- An obstacle that was not overcome was a result of the fact that the value of program variable x is now in relation with the program variable x^{dep}. The preservation of this relation and the underlying assumptions on this relation that make the rewrite rules possible is of a great importance. To correctly preserve this relation several approaches were taken, differing in e.g. the preservation or replacement of labels in function applications, update parallelization etc. Examples of elements in which we were not consistently able to preserve the relation between program variable and dependency set, were in splits on conditionals and, though similar, in the *cut*-rule[3].
- As a result of the above complications, the definitions of the actual rewrite rules for value-sensitivity become more restricted and in that way achieve less value-sensitivity as was hoped for with the original concept.

3.4.2. Separation of Implicit and Explicit Dependencies

In a different approach we started with the remark that the dependencies of a program variable x can be split into two sets, namely the program variables that have an *explicit* influence on the value of x, and those that have an *implicit* influence (via conditions in conditional or loop statements) on the value of x. We therefore remove the x^{dep} variable from our extension and instead replace it by two additional program variables for each existing program variable, namely x^{exp} and x^{imp}. The calculus rules are updated consequently, such that we get for example from the program `if(y>0){x=w-w}else{x=z}` the update:

$$\mathtt{x} := \mathit{if}(\mathtt{y} > 0)\mathit{then}(\mathtt{w} - \mathtt{w})\mathit{else}(\mathtt{z}) \,\|\, \mathtt{x}^{exp} := \{\mathtt{w},\mathtt{z}\} \,\|\, \mathtt{x}^{imp} := \{\mathtt{y}\} \,\|$$

We now provide a rule to remove explicit dependencies if they do not affect the value of term to which x is updated. For example, if we perform a split on the condition $y > 0$ we get two branches. In the first branch we have the update:

$$\mathtt{x} := \mathtt{w} - \mathtt{w} \,\|\, \mathtt{x}^{exp} := \{\mathtt{w},\mathtt{z}\} \,\|\, \mathtt{x}^{imp} := \{\mathtt{y}\} \,\|$$

In which both the explicit dependencies of w and z can be removed from x^{exp}. In the other branch we have the update:

$$\mathtt{x} := \mathtt{z} \,\|\, \mathtt{x}^{exp} := \{\mathtt{w},\mathtt{z}\} \,\|\, \mathtt{x}^{imp} := \{\mathtt{y}\} \,\|$$

In which only the dependency of w can be removed. The implicit dependencies however are separated and can thus not be removed, which fixes one of the problems faced with the label-based approach of the previous section.

On the other hand we do in this approach still have the problem that there is a relation between x and x^{exp} that needs to be preserved, giving rise to the same problems as faced in the label-approach. It was attempted to combine both approaches but since the essence of the problem is shared by both of them, this did not lead to a successful framework.

[3]The *cut*-rule is a rule that splits a proof tree into two branches. Each branch has a copy of the sequent on which this rule is applied, however for the same formula ψ one branch includes this formula in the antecedent, while the other branch in the succedent. The soundness of this rule is based on the fact that a formula can only hold or not hold. Therefore if one can close both the two branches, one has proven the original sequent as well.

4. Conclusions and Future Research

The automatic abstraction of while loops as was introduced in (Bubel et al., 2009) is successfully lifted to an approach that can handle objects and fields as well. We extended the dynamic logic's signature, syntax and semantics to include these objects and fields. Consequently calculus rules, abstract domains and the automatic proof search are updated as well to be applicable to these new definitions.

The resulting approach is able to find an invariant update for a while loop automatically, however the abstraction becomes imprecise when proving properties concerning the shapes that arise from the inclusion of objects, such as lists and trees. Especially when these shapes are altered within a program difficulties arise. A better suited approach to prove those kind of properties can be found in shape analysis, however we aimed at integrating the inclusion of objects consistently with the original approach on invariant derivation, which differs radically from the shape analysis technique.

A suggestion for improvement is done to imitate some of the capabilities of shape analysis by adding explicit support for one particular shape, namely a list iterator. However, the limits of extending the abstraction mechanism with concepts copied from shape analysis are met, when fields responsible for the shape are changed within a program. In future research it would be interesting to see if a stronger link with shape analysis can be obtained, or if it possible to integrate shape analysis and the abstract lattice approach used in this thesis in one logical framework.

We also improved on the information flow analysis using dependencies, introduced in (Bubel et al., 2009) as well. By tracking the dependencies of a variable as a (Herbrand) term expression instead, we are able to achieve both value and control flow sensitivity to a far larger extend than the original approach. It is even suspected that the semantical definition of the explicitly tracked dependencies equals exactly the set of actual dependencies, or at least comes very close. This is only a suspicion based mainly on the observation that no counter-examples could be found. For future research we would like to investigate this further and prove this hypothesis if possible.

Another interesting subject for future research is to create a more generic logical framework for dependency tracking. For cryptographic purposes, one could for example consider a framework for tracking dependencies under a certain assumption. Such a framework would be able to prove that in the program `p : x = g^y`, where `g` is the generator of a cyclic group, the variable `y` is not a dependency of `x` under the Discrete Logarithm assumption.

Combining the information flow analysis and the object extension as they are introduced in this thesis remains an open task. During the development of the logical framework for information flow as it has been presented in this thesis a number of different approaches have been tested. The concepts underlying these approach may be profited from in later extension to the framework, including the integration with an object-oriented language.

We hope to keep on integrating and improving these logical frameworks in the future, and eventually implement them in the formal verification tool KeY.

Bibliography

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

Clark Barrett and Cesare Tinelli. CVC3. In *Proceedings of the* 19th *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, 2007.

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

Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96. IEEE Computer Society, 2001.

Niklas Broberg and David Sands. Flow Locks – Towards a Core Calculus for Dynamic Flow Policies. In *ESOP'06: The 15th European Symposium On Programming*, pages 180–196. Springer-Verlag, 2006.

Richard Bubel, Reiner Hähnle, and Benjamin Weiß. *Abstract Interpretation of Symbolic Execution with Explicit State Updates*, pages 247–277. Springer-Verlag, 2009.

Pierre Castéran and Yves Bertot. *Interactive theorem proving and program development. Coq'Art: The Calculus of inductive constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252. ACM, 1977.

Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19:236–243, 1976.

Dorothy E. Denning. *Cryptography and data security.* Addison-Wesley Longman Publishing Co., Inc., 1982.

David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52:365–473, 2005.

Bruno Dutertre and Leonardo de Moura. The Yices SMT solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, 2006.

Melvin C. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer-Verlag, 1990.

Gerhard Gentzen. Untersuchungen über das logische Schließen I. *Mathematische Zeitschrift*, (39):176, 1934.

Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik*, 38:173–198, 1931.

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

Jacques Herbrand. *Recherches sur la Theorie de la Demonstration.* PhD thesis, University of Paris, 1930.

Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Advances in Cryptology - CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 789–789. Springer, 1999.

Saul A. Kripke. Semantical Considerations on Modal Logic. *Acta Philosophica Fennica*, 16: 83–94, 1963.

Thanh-Ha Le, Cécile Canovas, and Jessy Clédière. An overview of side channel analysis attacks. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, ASIACCS '08, pages 33–43. ACM, 2008.

Benjamin Niedermann. Proofs of Termination by means of Term Rewriting in KeY. Master's thesis, Chalmers University of Technology and Karlsruhe Institute of Technology, 2011.

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

Sam Owre, John Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *Automated Deduction-CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.

Philipp Rümmer. Sequential, Parallel, and Quantified Updates of First-Order Structures. In *Logic for Programming, Artificial Intelligence and Reasoning*, volume 4246 of *LNCS*, pages 422–436. Springer, 2006.

Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. In *Proceedings of the 18th IEEE Workshop on Computer Security Foundations '05*, pages 255–269, 2005.

Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24:217–298, 2002.

Michel Sintzoff. Calculating properties of programs by valuations on specific models. In *Proceedings of ACM conference on Proving assertions about programs*, pages 203–207. ACM, 1972.

Ernst Zermelo. Beweis, dass jede menge wohlgeordnet werden kann. *Mathematische Annalen*, 59:514–516, 1904.

Appendices

A. Rewrite Rules

A.1. Update Rewriting Rules

A rewrite rule $a \rightsquigarrow b$ is applicable to any occurrence of a within a sequent, and applying it means to replace that occurrence of a with b.

$$
\begin{array}{rcl}
\{\mathcal{U}\}\{\mathtt{x}_1 := t_1 \parallel \ldots \parallel \mathtt{x}_n := t_n\} & \rightsquigarrow & \{\mathcal{U} \parallel \mathtt{x}_1 := \{\mathcal{U}\}t_1 \parallel \ldots \parallel \mathtt{x}_n := \{\mathcal{U}\}t_n\} \\
\{\mathcal{U}\}f(t_1, \ldots, t_n) & \rightsquigarrow & f(\{\mathcal{U}\}t_1, \ldots, \{\mathcal{U}\}t_n) \\
\{\mathtt{x}_1 := t_1 \parallel \ldots \parallel \mathtt{x}_n := t_n\}\mathtt{x} & \rightsquigarrow & \left\{ \begin{array}{ll} \mathtt{x} & \text{if } \mathtt{x} \notin \{\mathtt{x}_1, \ldots, \mathtt{x}_n\} \\ t_k & \text{if } \mathtt{x} = \mathtt{x}_k \text{ and } \mathtt{x} \notin \{\mathtt{x}_{k+1}, \ldots, \mathtt{x}_n\} \end{array} \right. \\
\{\mathcal{U}\}a & \rightsquigarrow & a \qquad \text{where } a \in \mathcal{V} \cup \{\text{true}, \text{false}\} \\
\{\mathcal{U}\}\mathit{if}(\varphi)\mathit{then}(t_1)\mathit{else}(t_2) & \rightsquigarrow & \mathit{if}(\{\mathcal{U}\}\varphi)\mathit{then}(\{\mathcal{U}\}t_1)\mathit{else}(\{\mathcal{U}\}t_2) \\
\{\mathcal{U}\}p(t_1, \ldots, t_n) & \rightsquigarrow & p(\{\mathcal{U}\}t_1, \ldots, \{\mathcal{U}\}t_n) \\
\{\mathcal{U}\}(\varphi_1 * \varphi_2) & \rightsquigarrow & \{\mathcal{U}\}\varphi_1 * \{\mathcal{U}\}\varphi_2 \qquad \text{where } * \in \{\&, |, \texttt{->}\} \\
\{\mathcal{U}\}\,!\,\varphi & \rightsquigarrow & !\{\mathcal{U}\}\varphi \\
\{\mathcal{U}\}\mathcal{Q}y.\varphi & \rightsquigarrow & \mathcal{Q}y.\{\mathcal{U}\}\varphi \qquad \text{where } \mathcal{Q} \in \{\forall, \exists\}, y \notin \text{free}(\mathcal{U}) \\
\{\mathcal{U}\}(t_1 \doteq t_2) & \rightsquigarrow & \{\mathcal{U}\}t_1 \doteq \{\mathcal{U}\}t_2
\end{array}
$$

A.2. Update Rewriting Rules with Objects

The rewrite rules remain the same as in A.1, except for the application of updates on fields (as adopted from (Beckert et al., 2007, Chapter 3)).

Let $\mathcal{U} = \forall x.\varphi_1(x) \texttt{->} \mathit{loc}_1[x] := t_1[x] \parallel \ldots \parallel \forall x.\varphi_n(x) \texttt{->} \mathit{loc}_n[x] := t_n[x]$

$$
\{\mathcal{U}\}\mathit{loc}.b \rightsquigarrow \left\{ \begin{array}{ll} t_k[y'] & \text{if } \exists y.\big((\{\mathcal{U}\}\mathit{loc}) = \mathit{loc}_k[y] \text{ and } \varphi_k(y) \text{ and } \mathit{loc} \notin \{\mathit{loc}_{k+1}[y], \ldots \mathit{loc}_n[y]\}\big) \\ (\{\mathcal{U}\}\mathit{loc}).b & \text{otherwise} \end{array} \right.
$$

Where y' is the “smallest” substitute for the quantified variable x in the update such that $\varphi_k(y')$ holds (preserving the *least-one wins* semantics). The instantiation of y' is out of the scope of this thesis, for more information the reader is referred to (Rümmer, 2006).

B. Proofs

B.1. Lemma: Soundness of weakenUpdate

For convenience we repeat the weakenUpdate rule here:

$$\text{weakenUpdate}\ \frac{\begin{array}{l}\Gamma, \{\mathcal{U}\}\overline{\forall} x.(\overline{loc}[x] \doteq \bar{c}(x)), \Xi \Longrightarrow \overline{\forall} x.\chi_{\bar{a}}(\bar{y}(x)) \ \&\ \{\mathcal{U}'\}[\bar{\gamma}/\bar{y}]\overline{\forall} x.(\overline{loc}[x] \doteq \bar{c}(x)), \Delta \\ \Gamma \Longrightarrow \{\mathcal{U}'\}\varphi, \Delta\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}\varphi, \Delta}$$

Proof. We assume the premises of the rule to be valid. Hence for all first-order structures $M = (\mathsf{D}, I)$, states s, and logic variable assignments β we assume the following two statements hold:

$$val_{M,s,\beta}(\Gamma, \{\mathcal{U}\}\overline{\forall} x.(\overline{loc}[x] \doteq \bar{c}(x)), \Xi \Longrightarrow \overline{\forall} x.\chi_{\bar{a}}(\bar{y}(x)) \ \&\ \{\mathcal{U}'\}[\bar{\gamma}/\bar{y}]\overline{\forall} x.(\overline{loc}[x] \doteq \bar{c}(x)), \Delta) = tt \tag{B.1}$$

$$val_{M,s,\beta}(\Gamma \Longrightarrow \{\mathcal{U}'\}\varphi, \Delta) = tt \tag{B.2}$$

We need to show that for an arbitrary first-order structure $M_0 = (\mathsf{D}, I_0)$, state s_0 and logic variable assignment β_0 the statement $val_{M_0,s_0,\beta_0}(\Gamma \Longrightarrow \{\mathcal{U}\}\varphi, \Delta) = tt$ holds. Without the following assumption we would be done immediately, so we include it:

$$val_{M_0,s_0,\beta_0}(\bigwedge(\Gamma \cup {!}\,\Delta)) = tt \tag{B.3}$$

We thus need to prove that $val_{M_0,s_0,\beta_0}(\{\mathcal{U}\}\varphi) = tt$. We take $s_1 = val_{M_0,s_0,\beta_0}(\mathcal{U})$, we thus need to prove $val_{M_0,s_1,\beta_0}(\varphi) = tt$. Taking $M_0' = (\mathsf{D}, I_0')$ with $I_0' = I_0$ except that $I_0'(\bar{c}) = s_1(F(\overline{loc}))$, F as in the definition of semantics (Definition 13).

$$F(loc) = \left\{\begin{matrix}\mathtt{x} & \text{if } loc = \mathtt{x}\ (loc \in \mathcal{F}_{n,0}) \\ (b, val_{M,s,\beta}(loc')) & \text{otherwise, } loc = loc'.b\end{matrix}\right\}.$$

Thus we have

$$val_{M_0',s_0,\beta_0}(\{\mathcal{U}\}\overline{\forall} x.(\overline{loc}[x] \doteq \bar{c}(x))) = tt \tag{B.4}$$

Since I_0' only differs from I_0 in the interpretation of $\bar{c}$, which are not present in neither Γ nor Δ, we also have that:

$$val_{M_0',s_0,\beta_0}(\bigwedge(\Gamma \cup {!}\,\Delta)) = tt \tag{B.5}$$

All formulas $\xi_i \in \Xi$ are *required* (see the definition of the weakenUpdate rule in Section 2.2.4) to be of the form $\forall x.y_i(x) \doteq r[x]$, with y_i fresh and not occurring anywhere else in Ξ. Therefore,

the formulas in Ξ can never contradict themselves and we can conclude, combining (B.5), (B.4) and the first premise (B.1):

$$val_{M_0',s_0,\beta_0}(\overline{\forall}x.\chi_{\bar{a}}(\bar{y}(x)) \;\&\; \{\mathcal{U}'\}[\bar{\gamma}/\bar{y}]\overline{\forall}x.(\overline{loc}[x] \doteq \bar{c}(x))) = tt \tag{B.6}$$

We now define interpretation $M_0'' = (\mathsf{D}, I_0'')$ with $I_0'' = I_0'$ except for $I_0''(\bar{\gamma}) = I_0'(\bar{y})$ which allows us to remove the substitution (and the & connective) to obtain:

$$val_{M_0'',s_0,\beta_0}\big(\{\mathcal{U}'\}\overline{\forall}x.(\overline{loc}[x] \doteq \bar{c}(x))\big) = tt \tag{B.7}$$

Let $s_1' = val_{M_0'',s_0,\beta_0}(\mathcal{U}')$, i.e., s_1' is the state reached by starting in s_0 and executing $\mathcal{U}'$ with the interpretation I_0''. Equation (B.7) is equivalent to

$$val_{M_0'',s_1',\beta_0}(\overline{\forall}x.(\overline{loc}[x] \doteq \bar{c}(x))) = tt \tag{B.8}$$

This implies that $I_0''(\bar{c}) = s_1'(F(\overline{loc}))$ and since we know that $I_0''(\bar{c}) = s_1(F(\overline{loc}))$ we also have that $s_1(F(\overline{loc})) = s_1'(F(\overline{loc}))$. So s_1 and s_1' are identical on all locations potentially changed by $\mathcal{U}$ or $\mathcal{U}'$. Both are derived from s_0 with these updates, so we can conclude that $s_1 = s_1'$, which gives us:

$$val_{M_0'',s_0,\beta_0}(\mathcal{U}') = s_1 \tag{B.9}$$

Let $M_1 = (\mathsf{D}, I_1)$ with I_1 be the interpretation identical to I_0'' except that the $I_1(\bar{c}) = I_0(\bar{c})$. Since the function symbols $\bar{c}$ do not occur in $\mathcal{U}'$ we can get from (B.9):

$$val_{M_1,s_0,\beta_0}(\mathcal{U}') = s_1 \tag{B.10}$$

Note that we would have got the same interpretation I_1 if we had defined it as identical to I_0 except with $I_1(\bar{\gamma}) = I_0''(\bar{\gamma})$. Since the $\bar{\gamma}$ symbols do not occur in Γ, Ξ or Δ we know from (B.3) that

$$val_{M_1,s_0,\beta_0}\big(\bigwedge(\Gamma \cup \Xi \cup !\,\Delta)\big) = tt \tag{B.11}$$

Combining (B.2) with (B.11) we get

$$val_{M_1,s_0,\beta_0}\big(\{\mathcal{U}'\}\varphi\big) = tt \tag{B.12}$$

With (B.10), this implies

$$val_{M_1,s_1,\beta_0}(\varphi) = tt \tag{B.13}$$

Since the symbols $\bar{\gamma}$ do not occur in φ, and since M_1 is otherwise identical to M_0, we get

$$val_{M_0,s_1,\beta_0}(\varphi) = tt \tag{B.14}$$

which is what we had to show. $\square$

B.2. Lemma: Soundness of invariantUpdate

The invariantUpdate rule from the Section 2.2.4:

invariantUpdate

$$\frac{\begin{array}{l}\Gamma, \{\mathcal{U}\}\overline{\forall} x.(\overline{loc}[x] \doteq \bar{c}(x)), \Xi \Longrightarrow \forall x.\chi_{\bar{a}}(\bar{y}(x)) \;\&\; \{\mathcal{U}'\}[\bar{\gamma}/\bar{y}]\overline{\forall} x.(\overline{loc}[x] \doteq \bar{c}(x)), \Delta \\ \Gamma, \{\mathcal{U}'\}g, \{\mathcal{U}'\}[\mathtt{p}]\overline{\forall} x.(\overline{loc}[x] \doteq \bar{c}(x)), \Xi \Longrightarrow \overline{\forall} x.\chi_{\bar{a}}(\bar{y}(x)) \;\&\; \{\mathcal{U}'\}[\bar{\gamma}/\bar{y}]\overline{\forall} x.(\overline{loc}[x] \doteq \bar{c}(x)), \Delta \\ \Gamma, \{\mathcal{U}'\}\,!\,g \Longrightarrow \{\mathcal{U}'\}[\ldots]\varphi, \Delta\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathtt{while\ (}g\mathtt{)\ \{p\};\ }\ldots]\varphi, \Delta}$$

Proof. We assume the premises to be valid, so we assume that for all first-order structures M, states s, logic variable assignments β:

$$val_{M,s,\beta}(\Gamma, \{\mathcal{U}\}\psi, \Xi \Longrightarrow \overline{\forall} x.\chi_{\bar{a}}(\bar{y}(x)) \;\&\; \{\mathcal{U}'\}[\bar{\gamma}/\bar{y}]\psi, \Delta) = tt \quad \text{(B.15)}$$

$$val_{M,s,\beta}(\Gamma, \{\mathcal{U}'\}g, \{\mathcal{U}'\}[\mathtt{p}]\psi, \Xi \Longrightarrow \overline{\forall} x.\chi_{\bar{a}}(\bar{y}(x)) \;\&\; \{\mathcal{U}'\}[\bar{\gamma}/\bar{y}]\psi, \Delta) = tt \quad \text{(B.16)}$$

$$val_{M,s,\beta}(\Gamma.\{\mathcal{U}'\}\,!\,g \Longrightarrow \{\mathcal{U}'\}[\ldots]\varphi, \Delta) = tt \quad \text{(B.17)}$$

Where $\psi = \overline{\forall} x.(\overline{loc}[x] \doteq \bar{c}(x))$ is used as an abbreviation to save space. We need to show that for an arbitrary first-order structure $M_0 = (\mathsf{D}, I_0)$, state s and logic variable assignment β the statement $val_{M_0,s_0,\beta_0}(\Gamma \Longrightarrow \{\mathcal{U}\}[\mathtt{while\ (}g\mathtt{)\ \{p\};\ }\ldots]\varphi, \Delta) = tt$ holds. As with the proof for the weakenUpdate rule, without the following assumption we would be done immediately, hence we include it:

$$val_{M_0,s_0,\beta_0}(\bigwedge(\Gamma \cup !\,\Delta)) = tt \quad \text{(B.18)}$$

Thus our statement to prove becomes $val_{M_0,s_0,\beta_0}(\{\mathcal{U}\}[\mathtt{while\ (}g\mathtt{)\ \{p\};\ }\ldots]\varphi) = tt$. Let $s_1 = val_{M_0,s_0,\beta_0}(\mathcal{U})$, the state reached after executing update $\mathcal{U}$ in s_0. If the loop does not terminate starting in s_1, the proof goal holds trivially (note that we have used [p] for partial correctness). Hence we assume that the loop terminates. Taking into consideration the semantics of the while loop, we know that there has to be a finite sequence of states $s_1, \ldots s_k$ such that:

$$val_{M_0,s_i,\beta_0}(\mathtt{p}) = \{s_{i+1}\} \qquad i \in \{1, \ldots, k-1\} \quad \text{(B.19)}$$

$$val_{M_0,s_i,\beta_0}(g) = tt \qquad i \in \{1, \ldots, k-1\} \quad \text{(B.20)}$$

$$val_{M_0,s_k,\beta_0}(g) = f\!f \quad \text{(B.21)}$$

Which makes our goal to prove that the statement $val_{M_0,s_k,\beta_0}([\ldots]\varphi) = tt$ holds.

To do so we first prove by induction that for every state s_i ($i \in \{1, \ldots k\}$), we can find an interpretation I_i of the symbols $\bar{\gamma}$ such that applying update $\mathcal{U}'$ to the initial state s_0 with this interpretation, we directly obtain state s_i.

So we proof by induction: for all $i \in \{1, \ldots k\}$, there exists an I_i identical to I_0 except for the interpretation of $\bar{\gamma}$ such that $val_{M_i,s_0,\beta_0}(\mathcal{U}') = s_i$ holds, where $M_i = (\mathsf{D}, I_i)$.

- *Base case* ($i = 1$).
  Note that the first premise of the weakenUpdate rule is the same as the first premise of the invariantUpdate rule. Similar to that proof, we can construct an interpretation I_1 with the requested properties, see (B.10) in Appendix B.1.

- *Induction step* ($i \in \{2, \ldots, k\}$)
  The induction hypothesis gives us:

$$val_{M_{i-1},s_0,\beta_0}(\mathcal{U}') = s_{i-1} \tag{B.22}$$

with $M_{i-1} = (\mathsf{D}, I_{i-1})$ and I_{i-1} identical to I_0 except for the interpretation of the symbols $\bar{\gamma}$. We define $M'_{i-1} = (\mathsf{D}, I'_{i-1})$ with I'_{i-1} identical to I_{i-1} except that $I'_{i-1}(\bar{c}) = s_i(F(l\bar{o}c))$. Since the $\bar{c}$ do not occur in $\mathtt{p}$, we obtain from (B.19):

$$val_{M'_{i-1},s_{i-1},\beta_0}([\mathtt{p}]\overline{\forall}x.(\overline{loc}[x] \doteq \bar{c}(x))) = tt \tag{B.23}$$

Since both $\bar{\gamma}$ and $\bar{c}$ do not occur in either Γ or Δ, and I'_{i-1} is otherwise identical to I_0, we get from (B.18):

$$val_{M'_{i-1},s_0,\beta_0}(\bigwedge(\Gamma \cup !\,\Delta)) = tt \tag{B.24}$$

Since the $\bar{c}$ also do not occur in $\mathcal{U}'$ and I'_{i-1} is otherwise identical to I_{i-1}, we get from (B.22):

$$val_{M'_{i-1},s_0,\beta_0}(\mathcal{U}') = s_{i-1} \tag{B.25}$$

Combining this with (B.23) we obtain:

$$val_{M'_{i-1},s_0,\beta_0}(\{\mathcal{U}'\}[\mathtt{p}]\overline{\forall}x.(\overline{loc}[x] \doteq \bar{c}(x))) = tt \tag{B.26}$$

Since neither $\bar{\gamma}$ nor $\bar{c}$ appear in g, we can combine (B.25) with (B.20) and obtain:

$$val_{M'_{i-1},s_0,\beta_0}(\{\mathcal{U}'\}g) = tt \tag{B.27}$$

If we now combine (B.24),(B.25),(B.26) on the premise (B.16) (again using the notion that formulas in Ξ never contradict themselves as in B.1), we get:

$$val_{M'_{i-1},s_0,\beta_0}(\overline{\forall}x.\chi_{\bar{a}}(\bar{y}(x)) \ \&\ \{\mathcal{U}'\}[\bar{\gamma}/\bar{y}]\overline{\forall}x.(\overline{loc}[x] \doteq \bar{c}(x))) = tt \tag{B.28}$$

We now take $M''_{i-1} = (\mathsf{D}, I''_{i-1})$ with I''_{i-1} identical to I'_{i-1} but with $I''_{i-1}(\bar{\gamma}) = I'_{i-1}(\bar{y})$ which allows us to remove the substitution (and the & connective) to obtain:

$$val_{M''_{i-1},s_0,\beta_0}(\{\mathcal{U}'\}\overline{\forall}x.(\overline{loc}[x] \doteq \bar{c}(x))) \tag{B.29}$$

Let $s'_i = val_{M''_{i-1},s_0,\beta_0}(\mathcal{U}')$, we have that the previous equation is equivalent to:

$$val_{M''_{i-1},s'_i,\beta_0}(\overline{\forall}x.(\overline{loc}[x] \doteq \bar{c}(x))) \tag{B.30}$$

From this we know that $s'_i(l\bar{o}c) = I''_{i-1}(\bar{c})$. Since we defined $I''_{i-1}(\bar{c}) = s_i(l\bar{o}c)$, we know that $s'_i = s_i$. Given our definition of s'_i, we obtain:

$$val_{M''_{i-1},s_0,\beta_0}(\mathcal{U}') = s_i \tag{B.31}$$

We now take $M_i = (\mathsf{D}, I_i)$ with I_i identical to I''_{i-1} except that $I_i(\bar{c}) = I_{i-1}(\bar{c})$. Since $\bar{c}$ does not occur in $\mathcal{U}'$, we can rewrite the above equation to:

$$val_{M_i,s_0,\beta_0}(\mathcal{U}') = s_i \tag{B.32}$$

So now we have the interpretation I_i being equal to I_0 except for the interpretation of $\bar{\gamma}$ and for which the desired property holds. This ends the induction.

Having proven our induction, we are interested in the particular case that $i = k$:

$$val_{M_k,s_0,\beta_0}(\mathcal{U}') = s_k \tag{B.33}$$

Since the $\bar{\gamma}$ symbols do not occur in g, we can combine this with (B.21) and obtain:

$$val_{M_k,s_0,\beta_0}(\{\mathcal{U}'\}g) = ff \tag{B.34}$$

And since these $\bar{\gamma}$ also do not occur in Γ nor Δ, we get using (B.18):

$$val_{M_k,s_0,\beta_0}(\bigwedge(\Gamma \cup !\,\Delta)) = tt \tag{B.35}$$

If we combine these last two equations with the third premise of the invariantUpdate rule (B.17) we get:

$$val_{M_k,s_0,\beta_0}(\{\mathcal{U}'\}[\ldots]\varphi) = tt \tag{B.36}$$

With the knowledge on s_k (B.33) we can replace this with

$$val_{M_k,s_k,\beta_0}([\ldots]\varphi) = tt \tag{B.37}$$

Finally, since the $\bar{\gamma}$ symbols do not occur in $[\ldots]\varphi$ and I_k is otherwise identical to I_0, we get:

$$val_{M_0,s_k,\beta_0}([\ldots]\varphi) = tt \tag{B.38}$$

which is what we had to show. □