Diploma Thesis

Taclets vs. Rewriting Logic - Relating Semantics of Java

by cand. inform.

Ralf Sasse

Responsible Supervisor: Prof. Dr. Peter H. Schmitt
Supervisor: Dr. Wolfgang Ahrendt, Andreas Roth

Institute for Logic, Complexity and Deduction Systems
Department of Computer Science
University of Karlsruhe

April 28, 2005

Statement

I hereby declare to have written this work independently and I did not use any other than the stated resources.

Karlsruhe, April 28, 2005

Ralf Sasse

Acknowledgments

Here I want to thank all those who kindly supported me during the work on this thesis. First of all, I want to thank Dr. Wolfgang Ahrendt for proposing the topic of this thesis as well as for his involved supervision of my work. I also want to thank Professor Peter Schmitt for making this thesis possible with supervision outside the University of Karlsruhe. Many thanks go to Andreas Roth for being my local supervisor and providing many helpful comments on my work and the implementation. I am also grateful to Steffen Schlager, Richard Bubel and Philipp Rümmer for their help on questions related to KeY and Java. I am also grateful to Professor Reiner Hähnle and Professor Peter Schmitt for making the stay in Gothenburg possible. Last but certainly not least, I want to thank my parents for their ongoing support of my studies.

Contents

1 Preliminaries **7**
1.1 KeY . . . 8
1.1.1 Taclets . . . 9
1.1.2 Code Transformation Taclets . . . 11
1.1.3 Propositional Logic Taclets . . . 13
1.1.4 Schema Variables and their Types . . . 13
1.2 Rewriting Logic . . . 14
1.2.1 Maude Language . . . 15
1.2.2 Programming Language Semantics in Maude . . . 17
1.2.3 Maude Java Semantics . . . 18
1.3 Goal of this Work . . . 23
1.4 Outline of this Work . . . 23

2 Correctness of Code Transformation Taclets **25**
2.1 Ideas for Taclet Correctness . . . 25
2.2 Validating Taclets with the MJS . . . 29
2.2.1 Technicalities on Maude w.r.t. Schematic Code . . . 30
2.2.2 Handling Schema Variables in General . . . 31
2.2.3 SV: Program Variable . . . 33
2.2.4 SV: Static Variable . . . 36
2.2.5 SV: Lefthandside . . . 38
2.2.6 SV: Simple Expression . . . 39
2.2.7 SV: Nonsimple Expression . . . 39
2.2.8 SV: Expression . . . 40
2.2.9 Equivalence of Configurations Modulo New Variables . 43
2.3 Restrictions on the Correctness Statement . . . 44
2.3.1 Missing Features . . . 45
2.3.2 Incomplete Features . . . 46

3 Lifting the Semantics **47**
3.1 Preparations . . . 47
3.1.1 Pausing the Execution . . . 47
3.1.2 Handling New Variables . . . 48

3.2 Lifting for Expressions 53
3.2.1 Rules for Applying a Side Effect Schematically 53
3.2.2 Helper Operator to Finish the Expression Evaluation . 60
3.2.3 Extended Conditional Values for Schematic Side Effects 61
3.2.4 Configuration Memory Changes by Side Effects 64
3.2.5 Delayed Configuration Memory Changes 66
3.2.6 Configuration Snapshots 69
3.2.7 Effects and Results of Expressions 70
3.2.8 Shortcomings of the Schematic Side Effect Handling . . 75

4 Changes to the Maude Java Semantics 77
4.1 Bug Fixes 77
4.1.1 Mistake in && and || w.r.t. the JLS 78
4.1.2 Array Creation Typo 79
4.1.3 Method Call Error 80
4.1.4 Bad Internal Handling which Stops the Execution . . . 83
4.1.5 Block Semantics 84
4.1.6 Missing Default Constructor Workaround 85
4.2 Extensions of the MJS - with JLS Defined Constructs 85
4.2.1 Type Check for Type Casts 85
4.2.2 Type Check for Assignments 87
4.3 Extensions of the MJS - with Taclet Language Constructs . . 90
4.3.1 Simultaneous Updates 90

5 Configuration Generation 93
5.1 Configuration Generation from Taclets by Java 93
5.2 Configuration Generation Maude Interface 94
5.2.1 Special Constants 94
5.2.2 Basics for Adding Data and Code to the Configuration 96
5.2.3 Adding Data to the Configuration 97
5.2.4 Adding Case Information 99
5.2.5 Examples of the Commands and their Results 99
5.3 Usage with Respect to Taclets 102

6 Propositional Logic Taclets in Rewriting Logic 104
6.1 Basics 104
6.2 Taclets Imitated 105
6.3 Propositional Logics Semantics 105
6.4 Validating Imitated Taclets with Help of the Semantics 106
6.5 Embedding Taclets into Rewriting Logic 108

6.6 Validating Embedded Taclets with Help of the Semantics . . . 109

7 Conclusions **110**

A Code for Examples **112**
A.1 Examples for Chapter 2.2 112

B Automated Use to Prove Sets of Taclets **119**
B.1 Technical Details 119

C Propositional Logic Taclets Maude Code **120**
C.1 Examples for Use of the Imitated Taclets 130
C.2 Propositional Logics Semantics Test Examples 131

1 Preliminaries

This work is about the problem of a formal validation of the rules of a programming language proof calculus. This is important to make sure that proofs given by the calculus will be correct. Otherwise for everything which is proven with the help of the calculus that proof only holds relative to the correctness of the calculus, meaning it could well be wrong. We attack this problem with a cross-validation of the rules with a semantics for the target language.

A cross-validation uses the fact that different formalizations have been done by different authors and thus contain different errors for the most part. Thus one can find the errors which are not the same in both formalizations and by eliminating these errors will end up with less total errors.

First, we need a semantics for the target programming language which in our work is *Java*. We decided to use a semantics which is given in *rewriting logic*. Then we can consider the calculus for a program logic whose rules we want to validate, which is the calculus for the *KeY* system. We validate the rules, implemented as so called *taclets*, of the KeY system with the help of the rewriting logic semantics for Java using *Maude*. Maude is an implementation of a rewriting logic system in which that Java semantics is already implemented.

The central problem we solved is that the rewriting logic semantics for Java is a semantics for *concrete* Java code while the Java code within the rules we want to validate is *schematic*. The main contribution of this work is the *lifting* of the semantics for Java to be able to cope with schematic Java code.

Our goal is to validate a large part of two kinds of rules:

1. *code transformation taclets* using the cross-validation with the rewriting logic semantics for Java in Maude,

2. *propositional logic taclets* using only Maude.

The rest of this chapter provides the necessary background information to understand our work and to sketch its context. This work has been done as part of the KeY project, which we will describe first. We describe the other used formalism, namely rewriting logic and the Maude language, second.

We focus on our approach for code transformation taclets in this work. There we find out that the relevant changes only happen in the schematic code. With the lifted Java semantics given in Maude, which is capable of executing schematic code, we can compare the two schematic code parts of a code transformation taclet. To automate that comparison, we implement a generation procedure which creates the generic starting states for such a comparison which is then executed by the semantics. Depending on the content of a taclet there are quite a lot of generic starting states necessary to show the taclet correct for all starting states and thus the automation is required.

In our other approach we implement a simple semantics for propositional logics in text-book style in Maude and then cross-validate that semantics with an imitation of the axiomatic taclets for propositional logics also implemented in Maude. This is possible because the axiomatic taclets for propositional logics implicitly specify a semantics for propositional logics, too.

1.1 KeY

The KeY project [ABB$^+$05], which is a joint work of the University of Karlsruhe, Chalmers University of Technology, Gothenburg, and the University of Koblenz, aims at integrating formal specification and verification of Java-Card programs into standard industry-used CASE tools to facilitate the use of formal methods in the software development community. Therefore the KeY project developed a sequent-based interactive proof system which gets its proof obligations from a CASE tool. As specification language it uses the Object Constraint Language (OCL) which ships with the widely used Unified Modeling Language (UML). OCL constraints are the basis for the proof obligation generation, e.g. to show that an implementation of a method really meets its specification. By now there is also a Java Modeling Language (JML) front end available which allows to give the constraints in JML in addition to giving them in OCL.

The proof obligations are given in a special logic, called JavaCard Dynamic Logic [Bec00, Bec01] (DL_J). For DL_J there are the two standard modalities box ([]) and diamond (<>). Programs will appear within these modalities in DL_J. Let p be a program and ϕ and ψ first order (or DL_J) formulas, then $< p > \phi$ expresses that p terminates in a state where ϕ holds and it is a formula in DL_J. Another example is the formula $\phi \rightarrow < p > \psi$ which states that when ϕ holds in a state then after the execution of p from that state the formula ψ will hold in the resulting state. This formula is also similar to the Hoare triple $\{\phi\}p\{\psi\}$.

In short DL_J is a program logic that can be seen as an extension of Hoare logic. The Dynamic Logic can also be viewed as a modal predicate logic with a modality for programs.

Example 1.1.1
A simple example rule, which has been simplified in its presentation in that there are no contexts for example, is

$$\frac{< \mathsf{x} = \mathsf{x} + 1; > \quad \phi}{< + + \mathsf{x}; > \quad \phi}$$

Pragmatically one has to read these rules bottom-up to get from complicated proof obligations to easier ones which can then hopefully be discharged. □

The calculus of the KeY system is a sequent calculus which is based on DL_J formulas and on both sides of a rule there are lists of DL_J formulas. The rules of the system are implemented as *taclets*. Taclets are, in the nomenclature of other (semi-)automated provers, basically light-weight tactics. They are used to describe and implement sequent calculi. We will devote Section 1.1.1 to taclets as they form the core of the prover and the goal of this work is to validate the code transformation taclets. We do that by cross-validating those taclets with the rewriting logic semantics for Java.

1.1.1 Taclets

Taclets represent rules which contain so-called schema variables (SV). Such a rule can be applied in infinitely many cases, namely those with a matching instantiation for its schema variables.

The main reference for taclets is the work reported in [BGH$^+$04]. Taclets are the rules of the calculus used within KeY. They are easy to master and provide good flexibility in designing a proof system. They substitute *meta languages* which are employed in other frameworks for interactive theorem proving.

We only present the parts of a taclet that are important for this work. These are the following parts for code transformation taclets:

- *find*: The find part is the part of the sequent which is changed by the taclet. There has to be a matching instantiation for the find part to match the part of the sequent which is to be changed.
- *replacewith*: The replacewith part replaces everything which has been matched by the find part and has to be instantiated accordingly.

So the formula the find part matched on is replaced by the replacewith part with the same instantiation when the taclet is applied. These rules have to be read bottom-up again as they expand that way.

Example 1.1.2
We will show three example taclets, the first one, with b a boolean schema variable, is:

```
not_left  {
  find  (! b ==>)
  replacewith(==> b)
  heuristics(alpha) };
```

This taclet replaces $!b$ on the left side of the sequent with b on the right side. In the usual way of writing rules this taclet would look like this:

$$\frac{\Gamma \text{ ==> } b, \Delta}{\Gamma, !b \text{ ==> } \Delta}$$

The second taclet, with #x a schema variable of type lefthandside, is:

```
preincrement {
  find        (#allmodal{{.. ++#x;  ...}}(post))
  replacewith (#allmodal{{.. #x =
                  (#typeof(#x))(#x+1);...}}(post))
  heuristics   (simplify_int)};
```

In this taclet there is a lot of syntactic sugar but the basic properties are that the find part is the code ++#x; and the replacewith part is made up of the code #x = (#typeof(#x))(#x+1) where the (#typeof(#x)) is a cast onto the type of the variable matched by #x.

The #allmodal states that this taclet can be applied with any modality. There are other elements of the taclet language which would only allow this to work with, for example, a box modality.

The #typeof is a *meta construct* in the taclet language which takes the type of its argument and puts it where #typeof was before. Meta constructs are instantiated when the taclet is applied and can also be a lot more complicated than just getting the type of its argument. There is, for example, a meta construct which puts the method body of a method call in its place.

The constructs .. and ... represent the context. The ... can be any code while .. is restricted to opening braces and try-statements, so that the code after .. is the first active statement.

The third taclet is using a new variable and in this taclet #nse is a schema variable of type nonsimple expression and #se is a SV of type simple expression and #lhs is a SV of type lefthandside.

```
compound_multiplication_1 {
  find (#allmodal{{.. #lhs = #nse * #se; ...}}(post))
  varcond (typeof(#nse) #v new)
  replacewith (#allmodal{{.. #typeof(#nse) #v = #nse;
                              #lhs = #v * #se; ...}}(post))
  heuristics (simplify_prog)
 displayname "multiplication"};
```

We want to focus on the varcond part which includes one *new variable*, namely #v. A new variable is a variable which is new w.r.t. the rest of the given program by its definition. We will take a closer look at new variables in Section 2.2.9 □

1.1.2 Code Transformation Taclets

Code transformation taclets (CTT) are a special case of taclets and they are crucial in this work as they form one subset of taclets we want to validate in Section 2. The other subset of taclets we look into in more detail are propositional logic taclets in Section 6

Definition 1.1.3 (Code Transformation Taclets)
Code transformation taclets contain a find part

$$find < \Pi > b$$

and exactly one replacewith part

$$replacewith < \Pi' > b$$

with Π and Π' pieces of code and b any formula. There can also be other parts as seen above which we have omitted here. □

A code transformation taclet is basically a sequent rule without branching where all changes happen within the modality, i.e. in the code. This means that a code transformation taclet can only ever change Java code to other Java code which is a fortunate restriction and will later allow us to work with them quite well.

If we take a look at the example 1.1.2 from above we can see that the second and third taclets are code transformation taclets because the only changes which happen there are in the code. The first taclet on the other hand is not a code transformation taclet as there are changes on the sequent level, apart from no code being there anyway.

Example 1.1.4
Let us take the second taclet from Ex. 1.1.2 to see an example of a taclet application. The taclet is:

```
preincrement {
  find        (#allmodal{{.. ++#x;  ...}}(post))
  replacewith (#allmodal{{.. #x =
                   (#typeof(#x))(#x+1);...}}(post))
  heuristics   (simplify_int)};
```

Given

```
++i; ++i;  (f)
```

where `i` is an integer variable and `f` is some formula which we do not want to spell out then the taclet can be applied with `#x` instantiated to `i` and `post` instantiated to `f`. In that case `#typeof(#x)` turns out to be `int` as `i` is an integer variable. Then the result of one application of the given taclet is:

```
i = (int) i+1; i++; (f)
```

□

So we have now seen that a taclet can change a dynamic logic sequent to another such sequent.

1.1.3 Propositional Logic Taclets

Propositional logic taclets are taclets which only contain sequents of formulas of propositional logic. In addition to the taclets we have already seen they have some extra parts, which are:

- *if*, this part has to appear in the sequent too if the taclet is to be applied. Instantiations which are necessary to match the if part have to allow the find part to be matched, too.
- *goaltemplate*, it consists of a replacewith part and an *add* part. The add part is optional and we will not have any add parts appear in this work. Thus each goaltemplate part reduces to a replacewith part here. There can be multiple goaltemplates per taclet which split the sequent into multiple goals.

We have already seen an example of a propositional logic taclet, which is the first example in Ex. 1.1.2. Another example is:

Example 1.1.5
This taclet finds an equivalence b <-> c on the right side of the sequent and replaces the sequent by two sequents, one in which b is on the left side of the sequent and c is on the right side and the other one has the variables in switched places.

```
equiv_right  { find (==> b <-> c)
                       replacewith(b ==> c);
                       replacewith(c ==> b)
                     heuristics(split ,beta) };
```

□

1.1.4 Schema Variables and their Types

As mentioned above, the taclets include so called schema variables. These schema variables are variables which can match code or terms, depending only on their type. A schema variable type is for example *program expression*, which is able to match on any Java expression. Another example is *program lefthandside*, which will only match on expressions which can appear on the lefthand side of an assignment. There are a few other types of schema variables and all of them are described, together with the consequences they have in Section 2.2.

An example of a schema variable can be found in the above Ex. 1.1.2 and is #x. In the above example #x is a schema variable of type program lefthand-side.

1.2 Rewriting Logic

We will first describe rewriting logic in short. This section's beginning has been taken from the rewriting logic road map paper [MOM02] with only slight modifications. In that paper you can find a more detailed explanation and references to a whole lot of work in the area as it includes a very large bibliography.

In *rewriting logic* the fundamental axioms are rewrite rules of the form $t \to t'$ where in general t and t' are expressions in a given language. You can read such a rewrite $t \to t'$ in two complementary ways, one logical and one computational:

- logically, the rewrite rule $t \to t'$ is interpreted as an inference rule, so that we can infer formulas of the form t' from formulas of the form t.
- computationally, the rewrite rule $t \to t'$ is interpreted as a local transition in a concurrent system, meaning t and t' describe patterns for parts of the distributed state of a system. The rule states how a local concurrent transition can take place in such a system, thereby changing the local state pattern from an instance of t to one of t'.

A rewrite theory is basically an instance of rewriting logic. Where rewriting logic proscribes the form of rules, the rewriting theory provides the concrete rules. A rewrite theory is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$ where (Σ, E) is the equational theory modulo which we rewrite, L is a set of labels and R is a set of labeled rules. Rewriting modulo equations means that all rewrites happen on equivalence classes which naturally have to be independent of the current representative. For now we assume that R consists of unconditional labeled rules but all this also extends easily to conditional rules that may even contain rewrites in their condition.

Example 1.2.1
Take a simple rewrite theory [MOM99] whose rewrite rules rewrite ground multisets, which are built out of some constants, by means of an associative and commutative multiset union operator, denoted e.g. by $\otimes$.

This rewrite theory has an obvious computational reading as a (place/transition) Petri net. There is also a logical reading for which you best see [MOM99].

In the case of a Petri net, Σ consists of the binary multiset union operator $\otimes$ and one constant for each place in the net. E consists of the associativity and commutativity equations for multiset union, L is the set of labels of the net's transitions and R is the set of transitions. Since we rewrite modulo the equations E we are actually rewriting equivalence classes of terms modulo E.

In the Petri net example that corresponds to the fact that each transition rewrites a part of the current multiset of places (graphically depicted as a "marking", with as many "tokens" in a place as its multiplicity) modulo the associativity and commutativity of multiset union. $\square$

So all relevant sentences of $\mathcal{R}$, provable or not, are sequents of the form $[t]_E \rightarrow [t']_E$ where t and t' are Σ-terms, maybe involving variables, and $[t]_E$ denotes the equivalence class of the term t modulo the equations E. The provable sentences are exactly those derivable by the 4 inference rules **Reflexivity**, **Congruence**, **Replacement** and **Transitivity** for unsorted unconditional rewriting logic. With unsorted unconditional rewriting logic we mean rewriting logic without conditional equations or rules and no sorts are used.

Again one can distinguish the computational and logical view. Computationally, the provable sequents describe all the complex concurrent transitions of the system axiomatized by $\mathcal{R}$. Logically, they describe all the possible complex deductions from one formula to another in the logic axiomatized by $\mathcal{R}$.

It is also worth to mention the fact that rewriting logic can be used as a logical and semantic framework as shown in the paper [MOM00].

1.2.1 Maude Language

The Maude language is based on rewriting logic and you can find detailed information about it in the Maude manual [CDE$^+$00]. Maude is the system that implements rewriting logic using the Maude language. In Maude you can give executable specifications of anything representable as a rewrite theory. Maude also allows a wide array of user-defined syntax which makes it particularly easy to write a semantics for a programming language. This semantics is then in itself an executable specification of the chosen language and can be used to execute code of that language in interpreter style.

Maude is also quite efficient and performed all the tasks in this work sufficiently fast. It is also simple enough that readers unfamiliar with Maude should be able to understand the basic code used within this work relating to the Maude semantics for Java.

To allow rewriting with the rules modulo equations in Maude the equations are required to be terminating and confluent and after each rule application equational simplification happens as far as possible.

We will now give a first simple example of a rewrite theory in Maude to get familiar with the syntax and also see more of a rewrite theory.

```
fmod EX-NAT is
  sort Nat .
  op zero : -> Nat .
  op s_ : Nat -> Nat .
  op _+_ : Nat Nat -> Nat .

  vars N M : Nat .
  eq zero + N = N .
  eq s N + M = s(N + M) .
endfm
```

The keywords fmod and endfm denote the start respectively end of a Maude module. EX−NAT is the module name and the is always follows after the name of the module. The keyword sort declares a new sort, which is like a type in programming languages. So here we declare the sort of Nat's, i.e. natural numbers. With the keyword op, we create an operator, or function, zero, in this case with 0 arguments so it is a constant and has its image in Nat. It represents the 0 element. Then we have the operator s_ which takes one argument of sort Nat (behind the colon and before the arrow are the arguments) and with the underscore after the s it means that you can write for example s zero instead of s(zero) which is always allowed for operators. It is similar for _+_, which uses a mix-fix notation, and takes two Nat arguments to result in a Nat. Then there are two variables declared with the keyword vars and they are given the type Nat. Now with all the declarations done we can give equations to work on the sort. They are given with the keyword eq and the first one allows us to reduce zero added to N to just N. The second takes the successor of N added to M and reduces it to N added to M and takes the successor of that. These two equations reduce any expression given with the operators provided here to a canonical form. It is obviously a terminating process for every given expression and it is also confluent as there is always only one of the two equations applicable. In a nutshell, we have developed a rewrite theory for the natural numbers given by zero and the successor function s, with the addition operator +. It does not matter that there is actually no rewrite rule given at all, this is still a rewrite theory but with an empty rule set.

1.2.2 Programming Language Semantics in Maude

To show that it is possible to specify the semantics of a programming language such a specification was developed for an imagined language quite similar to CaML. We mention this semantics because it is very similar in style to the Maude Java Semantics and it is also very well documented [MR04]. It also is much smaller so it allows a quicker start into the look and feel of such executable specifications of semantics with the help of Maude and rewriting logic if one is interested in that.

The semantics of a CaML-like language using rewriting logic and implemented in Maude is given in great detail and with a lot of explanations in [MR04]. This semantics of a CaML-like language is also an executable specification which can be used to execute CaML-like code, so one got an interpreter for free. It contains arithmetic and boolean expressions, conditional statements, higher-order functions, lists, let bindings, recursion with let rec, side effects via variable assignments, blocks and loops, exceptions, and concurrency via threads and synchronization.

The semantics for the CaML-like language uses *continuation passing style* (CPS), similar to the way CPS is used in CaML. Thus the actual code to be executed is within a continuation. Continuations can roughly be seen as an executable stack of statements which can be restored anytime. We do not have the complete state in the continuation but only the code with, possibly, some intermediate results. We always use this continuation and execute the code from it.

1.2.3 Maude Java Semantics

The Maude Java Semantics (MJS) is hinted on in [MR04, FCMR04, FMR04] and is an executable specification of the semantics of Java. It is built similar to the smaller size semantics of aa CaML-like language from above. The Maude Java Semantics is experimental only and does not cover all of Java. One of the main features which needs to be taken into account in this work is that it uses an explicit environment and memory model for Java. We will call the environment, memory and other state information altogether the "configuration" from now on. For more information on the MJS take a look at the paper about Java source code, [FCMR04], and the paper working with Java Virtual Machine (JVM) code, [FMR04]. They are both mainly concerned with working on a Java Formal Analyzer tool which needs the MJS as its core at least for source code verification. There is no real documentation of the MJS we could point you to and therefore we give some detail on the simpler CaML-style language in Section 1.2.2 to allow the reader to get a feeling for the MJS.

The MJS requires that the code which is to be executed with the MJS is changed a little bit compared to regular Java code, but the modifications are small and merely syntactic. An interpreter for Java code is obtained which is driven by the specification given in the MJS. We will use it to compare the results of two different code fragments later on, to show that they are equivalent in a certain sense. In the following we take a look at how a configuration of the MJS interpreter looks like. This is supposed to help understanding the modifications to the configuration which we introduce later on.

During the work we could compare the readability of the Java semantics given by the KeY taclets and the MJS. While single taclets are easier to understand, the execution of a more complex program with them is not that easy to imagine. On the other hand in the MJS there are no meaningful, small and easy to understand, parts but the whole has to be investigated and can then facilitate execution of programs. For somebody who does know neither taclets nor Maude, the semantics given by the taclets will be easier to comprehend as it can be viewed in smaller parts.

Configuration Structure

This part is designed to be useful as a later reference to understand what the rules and equations work on. It might be good to take a short look at it now to know what to expect and later see when returning here could be helpful.

As said above a configuration is the total state information, including the memory and environment as well as the code, of any point in the execution of a Java code fragment within the MJS.

In the following configuration you will see a lot of operators which are either from the original MJS or are added by us. The added parts are described in more detail, when they are really going to be used, see Sections 3 and 4.

So let us start looking into the MJS by first investigating the structure of configurations. In the following X:Sort, which is an ad-hoc declaration of a variable of sort Sort with the name X, is there so you can easily see what sorts the elements have to have where they appear. This is opposed to the way of first declaring every variable with a sort in the form var X : Sort . and then in the remainder of the code use the shorter X only. That is only useful when variables are used more than once which they are not in this configuration.

```
run(c(k(CODE:BlockStatements -> pause -> K:Continuation),
        e(E:Env),
        o(o(STATICTYPE:CType,DYNAMICTYPE:CType,OE:ObjEnv))),
    m(M:Store),
    n(I:Int),
    cl(LISTOFALLCLASSES:Classes),
    s(STATICENV:ObjEnv),
    out(O:Output),
    l(noLock),
    w(noLock),
    nextSnapshot(N:Int),
    snapshots(SNL:SnapshotList)
)
```

Now let us look at this step by step. First we see that everything is wrapped within a run (...) . The operator run is defined on a MyState, whose elements are StateAttributes, and makes the whole expression to be of type Output.

-
  ```
  c(k(CODE:BlockStatements -> pause -> K:Continuation),
    e(E:Env),
    o(o(STATICTYPE:CType,DYNAMICTYPE:CType,OE:ObjEnv))
  ```

)

This part is made up by a wrapping c (...) which takes a Context, which is made up of ContextItems, and the three inner lines:

- The first line is wrapped in a k (...) and is a ContextItem which itself takes a continuation. The continuation is CODE:BlockStatements −> pause −> K:Continuation where the code within CODE will get executed first, then there is a pause and afterwards there is a rest K of the Continuation. Intuitively this is the program which is run.

  This pause operator was invented by us to allow separating the execution of different code blocks. It allows us to stop after the code part of a taclet. This is relevant for us as we get code from taclets which we want to execute and after that code there can be quite a lot more unrelated code which we do not want to execute.

- The second line is wrapped in an e (...) which also is a ContextItem and which has an Env as argument. Env stands for environment. An environment basically maps variable names to locations and is realized as a multiset of such pairs. This maps all the variables which appear in the program to locations in the memory. At those locations their value can then be found.

- The third line is wrapped in an o (...) which is a ContextItem and takes an object as parameter. This object is the current object upon which the current computation, basically a method call, happens. The object is wrapped inside the inner o (...) and has three parts: its static type, its dynamic type, both of sort CType, and the ObjEnv which represents the object environment of this object. An object environment is actually similar to a normal environment with the difference that to each variable name-location pair there is a CType attached which is the class to which the attribute represented by this variable name belongs.

• ```
  m(M: Store),
  n(I : Int)
  ```

  Now there is the memory represented by the following where m(...) wraps a Store to make it a StateAttribute. A store is a multiset of location-value maps. This is what we will view as the memory of a configuration. Then n (...) wraps a single integer to make it a StateAttribute
  ```

too and this has the meaning that the wrapped integer is the next free location in the memory, as locations are written as `l(I:Int)`.

- `cl(LISTOFALLCLASSES: Classes)`

  There is also this where `cl (...)` wraps an element of `Classes`, which itself is a multiset made up of elements of sort `Class`, so it is a `StateAttribute`. In here all class definitions of all the classes which are used in a program run can be found. This is important for example for method calls so that the right method body can be found.

- `s(STATICENV: ObjEnv)`

  Then there is the static environment wrapped inside `s (...)` which is also an object environment but with a different meaning than above. This object environment includes all static attributes of all classes, again as a triple with the class type, the name (of the attribute) and the location. The wrapper also builds a `StateAttribute`.

- `out(O: Output)`

  This is the accumulated output which is wrapped inside `out (...)` and takes an `Output`. This is used in the very end of the computation to create the resulting output. The wrapper also builds a `StateAttribute`.

- `l(noLock), w(noLock)`

  There are also some locking mechanisms which are part of the configuration but we will not be using these as we only use sequential programs and therefore do not need any locks. The wrappers also builds `StateAttribute`s.

- `nextSnapshot(N: Int),`
  `snapshots(SNL: SnapshotList)`

  Additionally this two items are concerned with snapshots, which are introduced by us because we later on need them to identify when the execution of an expression with side effects has been started, see Section 3.2. One wrapper is `nextSnapshot (...)` which depends on an integer and is the identifying number of the next snapshot similar to `n (...)` for the next free memory element. `snapshots (...)`, which takes a `SnapshotList` and is the memory for all snapshots that are taken during the execution, is the other wrapper. The result of both of them is of type `StatementAttribute`.

Smaller subparts of the configuration

Now we investigate smaller parts of the configuration.

We should first look at the environments. They map variable names n to locations l in the form $[n, l]$. One single mapping is already an environment and concatenations of two environments are again an environment. Concatenation is performed by using juxtaposition, i.e. just a space between two environments.

Stores are very similar. They map locations l to values v in the form $[l, v]$.

So in order to get from a name to its value you need to do two steps in this memory modeling:

1. the variable name maps to a location in the environment,
2. that location maps to a value in the store.

Object environments are built up like environments. To make it an object environment you use the following: `(CT, Env)` where `CT` is any class type and `Env` is an environment, which by the definition of environments above can consist of multiple name to location mappings. This has the meaning that all the environment parts which belong to a class type are its attributes. Multiple object environments are again put together by juxtaposition.

Now it is also worthwhile to look at how an object is structured. In fact, calling it an object is a bit misleading as it really is more a rich object reference. The form is as follows: `o(ST, DT, OE)` where `ST` is the static type, `DT` is the dynamic type and `OE` is the object environment.

When this object reference gets assigned another object, the static type may not change according to the Java language specification but only the other two parts can differ. We had to do some modifications to the MJS to make the assignment of object references work correctly. In the original MJS the static type for an object reference was changed around at will which is wrong according to the JLS. When the static type has to be changed while evaluating an expression due to a type cast on the name representing the object reference then this is possible in intermediate states but does not change the object reference in memory, only the one on the continuation. The static type in the intermediate state is also only changed if it is a correct cast.

Note the overloading of the operator `o`. It is used to

1. create a StateAttribute when its argument is an object,
2. create an object when its arguments are the objects static type, dynamic type and object environment.

Thus, o(o (...)) appears quite often in configurations.

1.3 Goal of this Work

The goal of this work is to investigate the correctness of taclets with the help of Maude. We consider two areas:

1. code transformation taclets with the help of the MJS,
2. propositional logic taclets, using only Maude.

We want to make sure that the code transformations used in the taclets are exactly reflecting the Java Language Specification (JLS) [GJSB00]. Up until now the taclets have been written with the JLS on the developer's knees and hoping that no errors happen. Now we can compare the execution of the code in the MJS with the code transformation in taclets and by that should be able to find mistakes easily.

1.4 Outline of this Work

The rest of the work is organized as follows. In Chapter 2 we present our general ideas on how to validate code transformation taclets and then focus on the different kinds of schema variables and their possible instantiations which are relevant for that validation. Also the handling of new variables is pointed out. Restrictions of our approach are shown, too.

We expanded the MJS, which works on concrete Java, to allow for execution of the schematic Java used in code transformation taclets in Chapter 3. We focus on ways to handle schematic expressions for this. Our actual work on this was to lift the original MJS, a semantics for concrete Java code, to our current MJS which is a semantics capable to cope with schematic Java code. An interesting point of this is, for example, how to write into an unknown memory location to simulate an unknown side effect.

We show bug fixes that we did to the MJS and extensions of the MJS for handling constructs defined in the JLS but not before supported by the MJS. We also add some features to the MJS which are nice to have when working with the taclet language. See Chapter 4 for this.

To automate the proving process we extended KeY, using Java, so that KeY can provide an output usable by the MJS in Maude. We also created an interface on the Maude side in the MJS to facilitate this information exchange. This is all needed to create the configurations for the MJS, see Chapter 5.

All of the above focuses on code transformation taclets but we also want to validate the propositional logic taclets. We do that in Chapter 6 where we validate that small subset of the KeY taclets correct by imitating the taclets in Maude. We also try an embedding of propositional logic taclets for which an application mechanism for taclets is given in Maude but do not validate the taclets again this way.

We sum everything up and show what could be done as future work in the conclusion, Chapter 7.

2 Correctness of Code Transformation Taclets

In this chapter we will focus on why it is enough to show the equivalence of the two code sections of a code transformation taclet to validate it. Ideas for the way the semantics is denoted in this chapter came from the book [NN92].

2.1 Ideas for Taclet Correctness

As we have seen in the last chapter, a taclet application can change the Java code as well as the surrounding dynamic logic formula. We would like to show the correctness of the KeY taclets in this generality but for now we will have to restrict ourselves to code transformation taclets due to two reasons. The first is that working with the code transformation taclets is much easier than working with general taclets and we can use the Maude Java Semantics for the code transformation taclets exhaustively as only changes on the code level are made and so both states, before and after the taclet application, can be easily represented in Maude. The second reason for the restriction is merely the practical observation that a very large part of the taclets used in KeY are actually code transformation taclets. According to our counts there are roughly 350 taclets specific to Java code. About 140 of these 350 are code transformation taclets, i.e. 40%. We already proved 55 of those and a few more should be doable with minor technical enhancements. The difference between the amount of taclets we could prove and the number of code transformation taclets comes from the fact that in those taclets *meta constructs* of KeY are used which are not easily transferable to the MJS. Another factor for this are the restrictions which come from the MJS problems, i.e. mainly its missing features.

Apart from the fact that these code transformation taclets are a large part of the taclets they are also important to be validated because their correctness depends on the actual Java Semantics which one might easily misinterpret from the JLS. Checking this is normally not easy. Here the fact that there is a Java semantics for Maude helps tremendously because we check those taclets which have been developed with a certain interpretation of the Java semantics in mind against another interpretation implemented by different people and thus it is probable to find mistakes due to misinterpretations. This is a cross-validation of the KeY code transformation taclets and the Maude Java Semantics. We assume that we are working with syntactically correct Java code as otherwise that code is not meaningful and we could not properly treat it.

Just to remind you, code transformation taclets in general contain the following

$$find < \Pi > b \quad replacewith < \Pi' > b$$

according to Def. 1.1.3.

Example 2.1.1
We take up Ex. 1.1.4:

```
preincrement {
  find         (#allmodal{{.. ++#x;  ...}}(post))
  replacewith (#allmodal{{.. #x =
                    (#typeof(#x))(#x+1);...}}(post))
  heuristics   (simplify_int)};
```

In this case Π = ++#x;

and Π' = #x = (#typeof(#x)) (#x + 1);. □

Now we will use a big step *structural operational semantics* (SOS)-like semantics [NN92] with a transition relation "$\rightarrow$" which will do as many rewrites as necessary. This means that in case of a starting code segment given on the left-hand side and no code segment remaining on the right-hand side then as many rewrites as necessary are done to completely execute that code segment. This is not limited to SOS big step semantics in theory but is used in practice within the Maude Java semantics since there we will execute code until the end or until a "pause" is encountered.

To evaluate what Π and Π' do we execute both on the same state and look at the resulting state. State is used in the SOS sense. Then we can say that the two code segments are equivalent if they both induce the same changes on all states, i.e. both code segments have for any start state the same resulting state if they have a resulting state (not necessary one resulting state for all start states though). In such a case, the taclet replacing Π by Π' has no effect on the state after code execution, and thus the taclet is validated.

Calling the starting state s and using the above SOS-like semantics we can write the equivalence of the two code segments as the following, where the equation has to hold for all states s:

$$< \Pi, s > == < \Pi', s >$$

where we will take a closer look at the equality sign == in the next subsection. Now let us consider what happens when we execute the code Π in s:

$$< \Pi, s > \rightarrow < \epsilon, s' >$$

with a state s'.

Actually the taclet language [ABB$^+$05] allows some code segments to be in front of Π but Π has to consist of the first active statements, i.e. everything in front of it are opening braces or try-statements. Naturally there can be code segments after Π, say Ψ so the above state transition should be

$$< \Pi \;\; \Psi, s > \rightarrow < \Psi, s' >$$

when we only want the code of Π executed, but as the code in Π (respectively code in Π' if it appears before Ψ) can not change the code in Ψ and as Ψ is the same behind any code segment in front of it, i.e. behind Π as well as behind Π', we do not need to consider Ψ here, but consider its execution for later. To facilitate this in practice our new `pause` operator is used in the MJS implementation for such partial executions.

If we are given that for all states s

$$< \Pi, s > \rightarrow < \epsilon, s' >$$

holds, or, written in the usual way when there is no code left,

$$< \Pi, s > \rightarrow s'$$

holds, and we can then show that

$$< \Pi', s > \rightarrow s'$$

holds, we know that Π and Π' create the same final state for each common starting starting state and therefore the taclet is correct.

Now we can not do this for all possible states s per se but if we can find one or more generic states which together represent all of the possible states we could then show the above execution result on those generic starting states. By that we know that it holds in all states and thus we have proven the taclet's correctness.

Note, that taclets contain schematic code, as we have seen in Ex. 1.1.2. That schematic code needs to be instantiated which we will here in general do by substituting generic "skolem" constants for all schematic code elements.

To have the check

$$< \Pi, s > == < \Pi', s >$$

done automatically and the property checked by a tool we can employ Maude and its Maude Java Semantics (MJS). The generic starting states s mentioned above can then be configurations for the MJS which are comprised of generic "skolem" constants and after the code execution we compare the resulting states in both cases. For the code to be executable in the MJS it also has to be replaced by a concrete program started in a generic configuration $< \Pi_{sk}, s_{sk} >$, respectively $< \Pi'_{sk}, s_{sk} >$. How they are obtained is explained in the next section.

We would also like to note here that the configurations are not completely generic, but instead depend heavily on the code which is to be executed. This code puts some minimal requirements on the configuration, e.g. for all appearing variables there must be an environment and memory entry, similar for attributes, etc. Thus we need to generate fitting generic starting configurations. This dependance also motivates the definition of the equality "==". For details on this we refer to Section 2.2.

Concretized code, i.e. code which is generated from the schematic code as described below, suffices to draw conclusions about the schematic code because of the way it is constructed and can represent any possible code fitting the schema. An obvious issue which would generally be different in two different concrete instances of a schematic code part is the variable naming. On the other hand this is no problem as the variables actual names do not matter at all as long as that naming is consistent. To compare the result of two programs we can simply require the instantiation in both starting states to use the same names for the same variables or even perform a renaming after the program execution.

At least equally important is the treatment of expressions. Their side effect and result can be different in different instantiations. This is part of the problem which gets solved in Chapter 3 and makes all the effort there necessary and worthwhile. Details can be found there.

We now execute the Java code in the MJS, shown by using "$\xrightarrow{MJS}$" as the state transition instead of "$\longrightarrow$" from above. With

$$< \Pi_{sk}, s_{sk} > \xrightarrow{MJS} s'$$

and

$$< \Pi'_{sk}, s_{sk} > \xrightarrow{MJS} s''$$

the above task of showing the results of these two code executions to be equal is reduced to the need to show that $s' == s''$. Showing this for a generic starting state is the same as showing it for all possible states as stated above.

To summarize, we can conclude the correctness of the taclet if we can show the following: For all states if

$$< \Pi, s > \longrightarrow s'$$

holds, then

$$< \Pi', s > \rightarrow s'$$

holds. This is correct if we can prove $s' == s''$ from above and as we explained why the concrete programs and starting state can be used we have validated the taclet.

On a side note, one generic starting configuration will not be able to always cover all possibilities, but the sum of the cases which are developed below will do. That is why we have to take such a close look at the schema variables in the following parts. We will show the equality property from above for all possible cases of the generic starting configurations to prove correctness.

2.2 Validating Taclets with the Help of the Maude Java Semantics

As we have seen above we need a generic state and a concrete generic program to be executed. Concrete generic programs from the schematic code of the taclets need a lot of case distinctions depending on the contained schema variables. First of all we need to take a closer look at what we can do about the generic starting state, which also depends on the schema variables.

Code which is executed in the MJS is evaluated on a configuration and possibly changes the configuration during its run. The starting point to take a look at the Java code transformation taclets of KeY, with the help of the Maude Java Semantics, is a generic configuration to be used as the generic starting state. This generic configuration can be filled with skolem constants describing the state as general as possible, within the restrictions imposed by the considered code. The actual code will then be executed in the given configuration, where it is treated as being part of a method which was called on some generic object.

In order to get the resulting configurations s' and s'' from a generic starting configuration s we compute the result of the code from the *find part* of a taclet in the starting configuration s to get s'. We compute the result of the execution of the code from the *replacewith part* in the configuration s, where only the code is exchanged, and possibly new variables with their locations and values have been added, and which is otherwise unchanged, to get s''. If the resulting configurations are the same we can conclude that the two programs are equivalent and thus we can deduct the validity of the original taclet.

To be able to actually execute any of the schematic code from the taclets we need to transform it into generic code without schema variables but with *concrete generic variables*. We can first of all note that a different naming of variables (if it is consistent) does not change the results at all. With concrete generic variables we mean any Maude variables which are uninterpreted, like `I:Int`. It is concrete in being an `Int` Maude variable, yet generic because it is a placeholder for anything of sort `Int`.

2.2.1 Technicalities on Maude w.r.t. Schematic Code

We want to be able to put skolem constants into the configuration and make it generic. Therefore one needs to work on intermediate states which are created by the given `run(Classes Exp)` method from Maude when stopping the execution during the run. One can not use this `run` of the MJS to create a generic configuration as no variable memory elements and names can be given because everything put into this `run` has to be concrete and parse-able by Maude according to the MJS. However, the intermediate state can be manipulated as much as desired and therefore the configuration can be filled with generic variables, integers, objects, etc. There the operator `run (...)` also appears, but it is overloaded and has different argument types and a different number of arguments compared to the one above and there are no problems with generic elements.

An important change to the taclet code when translated to code for Maude is that variable declarations, of variables which have been declared as new in the varcond part, do not appear as declarations in the code. They are already integrated into the environment and memory of the Maude Java Semantics by our creation of that configuration. Otherwise there would be no way to ensure that these variables are new. Actually the generated code would not be correct Java if the integration of the declarations into the Maude configuration did not happen before. Because of that integration we can consider all these variables as having been declared before so the code is syntactically correct Java.

Type casts do not appear if they cast to a #typeof meta-construct due to problems in the translation of those in code. This is no problem as those type casts would only be relevant when throwing exceptions, which we can not handle anyway, see Section 2.3. There we exclude the corresponding cases from our consideration.

We will also not handle any taclets with a division / or modulo % inside because of problems of the Maude internal type handling of these. But these are not too interesting anyway as they appear only in a taclet when there is a similar taclet using $+$, $-$ and $*$ so the untreated operators / and % should be correct when the treated operators $+$, $-$ and $*$ are correct. It is possible that there could be some mistake with division by zero as it is a special case not relevant to the other operators.

There are also other minor syntactic changes necessary which we will not comment on in length. An example for this is that a + 1 has to be written as + #i(1).

2.2.2 Handling Schema Variables in General

We now define the translation of elements of schematic code, i.e. schema variables, to concrete generic code.

In the next subsections we will describe how schema variables from taclets can be translated and put into the starting configurations of the Maude Java Semantics to test those taclets. In general a schema variable has a type and a name. In the actual taclets the schema variables are declared with a type and name. Given a schema variable named #sv we will name it svName:Name. The :Name defines that it is of the type of all variables. Also for each variable named as above appearing in a taclet we will add [svName:Name, svLoc:Location] to a certain environment, see below for details of when to put it in which environment, and we will add [svLoc:Location, svVal:Value] to the memory part of the configuration.

Later on we will see the necessity for differentiating between values of an arbitrary type and values with a special type, like primitive types. It is necessary to already allow for that right here, so the svVal:Value can be replaced by one of the special subtypes of Value, like integer or String, if that is necessary for any of the operations that are to be performed on it. That is because for example the addition + is only defined on integers in the MJS. We will check whether generic Values are enough or not in Chapter 5. If it is not possible to use the generic type Value then all acceptable (by the check) subtypes have to be used and extra start configurations generated for every case but we will later look into that in detail.

With the generic locations we use it is quite possible that in a real execution some generic locations, i.e. two or more of them, are actually the same location. Now we have multiple, possibly different, values for that one location. One could think we need to decide which one to take as the real value but that is not the case. As we only compare two codes with each other then if both are equivalent that means that they correspond on all generic locations and thus it does not matter which value is the “real” one and which values have already been overwritten. That is because our results correspond on all locations and we are only concerned with the correctness of our approach. Thus we need not give a second thought about multiple locations falling together and being just one.

The motivation of the sequel is not to justify the design of the different types of schema variables. That means that we can not motivate all of these in greater detail but instead refer to [BGH$^+$04] and [ABB$^+$05]. So we are only listing the different schema variable types and state what they match on so we later know what cases we have to create for each schema variable. The creation of each case and how we do that was our work on the other hand.

2.2.3 SV: Program Variable

Schema variables of type *program variable* represent local variables and also attributes of the current object. Attributes of other objects can not be matched this way as they need a prefix with the object to which they belong.

The name of a variable here is derived as described above, so a program variable #sv would be called svName:Name. For all of the cases distinguished below the mapping [svName:Name, svLoc:Location] is always present in some environment. *Which* environment is taken differs for different cases and is specified below for each variant. Suppose the mapping is called ee, i.e. ee := [svName:Name, svLoc:Location]. Each case defines in which environment the ee mapping has to appear. A schema variable of type program variable can be instantiated in the following ways. They are all part of the following list:

- local variable,
- attribute of the current object without explicit this,
- attribute of the current object with explicit this,
- passive expression.

Passive expressions are only of interest when taking static initialization into account and they are a special intermediate construct of the taclet language showing that no more static initialization is necessary. Thus they are no real part of the Java language and can not be handled by our semantics subsequently.

In the case of the schema variable being a local variable, ee has to be part of the *local environment*, i.e. c(e(ee ENV), STATE) with ee added into the local environment and ENV is the local environment before the addition and STATE is the rest of the state attributes within c (...) . In the Maude Java code the #sv is then replaced by svName:Name.

If the schema variable is an attribute of the current object, then ee is added to the *environment of the current object* as part of the object environment. ee has the static type of the object, i.e. the environment of the current object looks like this with ee already inside: (CT, CT', OBJENV (CT , ee). Here CT is the static type of the object, CT' is the dynamic type of the object and OBJENV is the old object environment. The schema variable could also be an attribute of some superclass of type CT. That does not change anything during the execution because it is accessible either way. This, i.e. using CT directly, makes it easier to write down because otherwise one would instead have to give a hierarchy of super classes which is not bounded. One would have to try any conceivable length, which is every positive integer and which obviously makes this impossible to do in practice.

In the Maude Java code the #sv is replaced by svName:Name in the case with an implicit `this` and by `this . svName:Name` in case of an explicit `this`.

Whenever a SV of type program variable appears in a taclet, all above mentioned cases are possible and the correctness of the taclet has to be shown for each of them. Also, if there is more than one SV, every combination with all interpretations of other schema variables has to be taken into account in general.

Special short notation for examples

For the convenience of readers unexperienced with Maude we will use a special notation to make the configurations in the examples much easier to read. The full (real and running) configurations for the examples can be found in the appendix, Section A.1.

The structure of configurations has been described in subsection 1.2.3. For any of those wrapping operators W there, like m, e, k, c and so on, we will use the notation X $\in$ W for any (fitting) X to state that this X is an additional part of the (usually multiset) listing which is wrapped by W with the usual concatenation operator. So [L:Loc, V:Value] $\in$ m stands for m([L:Loc, V:Value] M:Store) where M:Store was inside m before so this actually only adds elements into the wrapping operators, it does not state its whole interior.

Example 2.2.1
We return to the second taclet from Ex. 1.1.2 with the SV name changed from #x to #lhs1 and the type of the SV #lhs1 is now program variable. We have Π = ++#lhs1; and Π' = #lhs1 = (#typeof(#lhs1)) (#lhs1 + 1); and the concrete code would look as follows together with the configuration, depending on which of the above mentioned cases we look at. The type casts are dropped as argued in Section 2.3.

The whole (executable) configurations for each part can be found in the appendix A.1.

1. If #lhs1 is a local variable, the programs are:

Π_{sk} = ++ lhs1Name:Name ;

Π'_{sk} = lhs1Name:Name = (lhs1Name:Name + #i(1)) ;

Here we compare the result of Π_{sk} with the result of Π'_{sk} when both are started in a configuration made up by the changes

- [lhs1Name:Name, lhs1Loc:Location] $\in$ e and
- [lhs1Loc:Location, int(lhs1Val:Int)] $\in$ m

and the respective program as the first code on the continuation.

2. If #lhs1 is an attribute of the current object without an explicit this, then again:

   Π_{sk} = ++ lhs1Name:Name ;

   Π'_{sk} = lhs1Name:Name = (lhs1Name:Name + #i(1)) ;

   Here we compare the results of Π_{sk} and Π'_{sk} started in the configuration with these changes:

   - [lhs1Name:Name, lhs1Loc:Location] $\in$ o which means that it is in the object environment of o, i.e. the current object.
   - [lhs1Loc:Location, int(lhs1Val:Int)] $\in$ m

   Compared to the above case the only difference is that here the first mapping is inside a different environment.

3. If #lhs1 is an attribute of the current object with an explicit this, then the only difference to the case above with an implicit this is that in front of all occurrences of lhs1Name:Name there will be a this . :

   Π_{sk} = ++ this . lhs1Name:Name ;

   Π'_{sk} = this . lhs1Name:Name = (this . lhs1Name:Name + #i(1)) ;

   Here we compare the results of the two programs started in the configuration with these changes again:

   - [lhs1Name:Name, lhs1Loc:Location] $\in$ o
   - [lhs1Loc:Location, int(lhs1Val:Int)] $\in$ m

   Compared to the case 1 above the difference is that here the first mapping is inside a different environment and there is a this . in front of every appearance of lhs1Name:Name. Compared to case 2 there is only one difference, which is that this . appears in front of every appearance of lhs1Name:Name.

□

2.2.4 SV: Static Variable

Schema variables of type *static variable* represent the static attributes of any object. They are of the form $T.a$, where T is a type reference, or $o.a$ where o is an object reference.

In each case below [svName:Name, svLoc:Location] is put in some environment. We call this pair ee. We consider it to be a mapping, i.e. ee := [svName:Name, svLoc:Location]. Each case defines in which environment ee has to be. A schema variable of type static variable can be instantiated in the following ways. They are all part of the following list:

1. $T.a_1.....a_n$ with $a_1, ..., a_n$ static attributes and T a type reference,
2. $o.a_1.....a_n$ with $a_1, ..., a_n$ static attributes and o an object reference.

Only one dereferencing is actually allowed by the taclet language definition, i.e. this will be typeref . svName:Name or obj . svName:Name.

In both cases a type is attached to ee to make it an ObjEnv, i.e. an object environment, which is then put into the set of static environments.

In the case 1 ee is put in an object environment with the type svCT:CType which is the type of the type reference. In real Java though it could actually be a static attribute of any superclass of that type, meaning it should be put into the object environment with that superclass' type, but this does not matter as the dereferencing leads to the same result with either type together with ee. The static attribute environment will then look like this: s((svCT:CType, ee) OBJENV) where OBJENV is the static attribute environment before adding this new static attribute. In the code this #sv is replaced by svCT:CType . svName:Name where the svCT:CType is a skolem constant representing the type of the type reference which we consider to have this attribute.

In the case 2 the ee is put in an object environment together with svCT:CType. In this case the #sv is replaced by svObjRef:Name . svName:Name where svObjRef:Name is actually a reference to the object reference o(svCT:CType, svDT:CType, svObjEnv:ObjEnv) where svCT:CType is the object reference's static type and is used in the static environment mentioned above. The dynamic type of the object is svDT:CType and the object environment for this object is svObjEnv:ObjEnv. Also this object is in the memory in a mapping of [svObjRefLoc:Location, o(svCT:CType, svDT:CType, svObjEnv:ObjEnv)] and the variable is mapped to that location in the local environment, [svObjRef:Name, svObjRefLoc:Location].

Example 2.2.2
We return to Ex. 2.2.1, but with #lhs1 of type static variable. Then $\Pi =$ ++#lhs1; and $\Pi' =$ #lhs1 = (#typeof(#lhs1))(#lhs1 + 1);. The concrete code can again be found in appendix A.1. Here the examples depend on which of the above mentioned cases we look at.

- If #lhs1 is a static attribute of the form $T.a_1.....a_n$:

  Π_{sk} = ++ lhs1CT:CType . lhs1Name:Name ;

  Π'_{sk} = lhs1CT:CType . lhs1Name:Name
  = (lhs1CT:CType . lhs1Name:Name + #i(1)) ;

  Here we compare the result of Π_{sk} with the result of Π'_{sk} when both are started in a configuration made up by the changes:

  - (lhs1CT:CType, [lhs1Name:Name, lhs1Loc:Location]) $\in$ s which means that it is in the static environment.
  - [lhs1Loc:Location, int(lhs1Val:Int)] $\in$ m

  and the respective program Π as the first piece of code on the continuation.

- If #lhs1 is a static attribute, with a program variable as the first element, followed by a sequence of attribute accesses. The program variable has to be an object reference therefore:

  Π_{sk} = ++ lhs1ObjRefName:Name . lhs1Name:Name ;

  Π'_{sk} = lhs1ObjRefName:Name . lhs1Name:Name
  = (lhs1ObjRefName:Name . lhs1Name:Name + #i(1)) ;

  Here we compare the results of the two programs started in the configuration with these changes:

  - (lhs1CT:CType, [lhs1Name:Name, lhs1Loc:Location]) $\in$ s,
  - [lhs1Loc:Location, int(lhs1Val:Int)] $\in$ m,

- [lhs1ObjRefName:Name, lhs1ObjRefLoc:Location] ∈ e,
- [lhs1ObjRefLoc:Location,
  o(lhs1CT:CType, lhs1DT:CType, lhs1ObjEnv:ObjEnv)] ∈ m

Compared to the first case above the difference is that here lhs1ObjRefName is added in the environment and the location it is mapped to is added to the memory with the fitting generic value from above and an object reference appears first instead of a type reference.

□

2.2.5 SV: Lefthandside

After we have seen what is done about *program variables* and *static variables* we can use them to define *lefthandsides.* A schema variable of type lefthandside simply subsumes by definition the program variable and static variable cases.

A lefthandside can either be a program variable or a static variable. In addition, in the case of a static variable it can have longer dereferencing chains which we do not model. So it could be `a . b . c` but we restrict this to length two as otherwise arbitrary length would have to be generated which is not possible. This is also enough because all greater length would just be dereferenced more often to return the same result, presuming that there are just more steps in between. Using more steps in between could only show whether the handling of multiple dereferencing is working correctly within the MJS, which it is in fact, but which is not what we want to examine. So what actually is happening boils down to exactly one attribute access in the static case.

Also static initialization could happen when treating schema variables of type lefthandside but that is something we can not do anything about because of the Maude Java Semantics. The MJS simply creates all static attributes at the beginning and so there is no static initialization left to do during runtime, which is a wrong model of Java. The static initialization has actually a lot of very subtle points where one needs to be careful in Java but we have to ignore that, see Section 2.3.

There is no need for an example as one can take the examples of program variables and static variables and all subparts in those are possible here.

2.2.6 SV: Simple Expression

Schema variables of type *simple expression* represent those expressions which do not have side effects. They are actually restricted a bit more as we detail below but the following description should be good enough for a first impression.

Schema variables of type simple expression can take multiple forms. Among those are all the possibilities of schema variables of type program variable which are handled as described there. So in total, the following can appear:

- all possibilities of program variables,
- (negated) literals, like 1, *true*, −1,
- instanceof.

The instanceof case is not described more precisely because it is not implemented in the Maude Java Semantics so we are not going to be able to use and check this at all (see Section 2.3).

Literals can take the following forms: a boolean value, represented by a skolem constant of type bool, an integer value, represented by a skolem constant of type int, a string value, represented by a skolem constant of type string and a float value, represented by a skolem constant of type float. In these cases no changes to the environments or memory are necessary and these values get the following names with which they are put into the code (for a SV named #sv): bool(svBool:Bool), int(svInt:Int), str(svStr:String), fl(svFl:Float). If we do not know or care which type such a literal has we put the generic value svVal:Value into the code.

All proper examples for simple expressions would include a nonsimple expression or a general expression so we delay giving an example for a simple expression until Section 2.2.8.

2.2.7 SV: Nonsimple Expression

A schema variable of type *nonsimple expression* is more complicated than a schema variable of type simple expression as it can have side-effects. The set of simple expressions and the set of nonsimple expressions are distinct. So here we can have any expression which is not covered by the simple expression case.

For nonsimple expressions we will simply work as with expressions. If we can show the necessary points for an expression it certainly holds for nonsimple expressions too, as they are a subset.

This will not lead to problems because of which we can not prove some taclets. This is because they were designed with full expressions in mind. The distinction between nonsimple expressions and simple expressions is only used to facilitate proof search in KeY. So we refer to Section 2.2.8 on expressions to see how these are handled.

2.2.8 SV: Expression

A schema variable of type *expression* can take the form of any conceivable Java expression. This subsumes the simple expression and nonsimple expression cases.

It gets more complicated if schema variables of type *expression* appear since they can have side effects. Such an expression is completely characterized by the side effects it generates together with the resulting value, if it terminates normally. Otherwise there will be some exception thrown and some side effects will still happen. As usual we have schematic expressions so we do not know what they do exactly, but we can write down a list of location and value pairs, using uninterpreted constants, as well as we can take such a constant for the resulting value. For a terminating expression this characterizes the expression completely. Special constructs have been added to Maude's Java Semantics to handle expressions this way and put the changes the side effects induce into all stores, by way of the *extended conditional values* (see Section 3.2.3) which are also implemented in the Maude Java Semantics. Those stores with *new locations* as location element are exempt from the side effect induced changes. An extended conditional value is basically a value which depends on the location where it was stored at the time of the side effect execution.

As we only ever compare two resulting configurations with each other there is no need to explicitly evaluate extended conditional values as long as they are the same in both configurations which resulted from the computation. You can find more details about the extra constructs in Chapter 4.

Termination is only an issue if we can not prove two code segments equal. If we can prove them equal then obviously all expressions have started in the same state (for each expression appearance in both codes) and thus if one does not terminate the other does not either and they are then equal as they have the same result, which is non-termination. If we can not prove two code segments equal they still might be equal in the special case that both do not terminate but for different reasons, which we do not care about as we need them to be equal in every case.

Please note that the side effects depend on the state in which the expression is evaluated. Another thing to note about this handling of side effects is, that all side effects of an expression will appear simultaneously, which is sufficient for sequential programs as we use them anyhow. On the contrary it would not be sufficient for concurrent execution of more than one thread as there would then be an uninterruptible assignment to multiple locations. This is no problem if one does not use the special construct for this feature which can not be in any normal Java program anyway and this command can be safely used in sequential programs as it could be replaced by a number of single assignments.

In the code a #sv of type expression is replaced by
eval(svEN:ExpressionName, RESULTTYPE) where RESULTTYPE is the type the expression returns and such that it can work with the other operators in the code, i.e. it passes a type check.

Example 2.2.3
We now reuse the third taclet from Ex. 1.1.2. With that we look at a taclet with a condition about a new variable varcond (typeof(#nse) #v new) which means that there is a new variable #v with the type that #nse has. Here, #nse is a nonsimple expression, #se is a simple expression and #lhs is of type lefthandside. See more about new variables below with the equivalence modulo new variables (Section 2.2.9) and in Chapter 4 where the technicalities are presented. The programs are
Π = #lhs = #nse $*$ #se ;
and
Π' = #v= #nse ; #lhs = #v $*$ #se ;

For a nonsimple expression, like #nse, we said that we use the same method as for expressions. Therefore we only get one new case for this, apart from all the possibilities from all other types. For simple expressions, like #se, we have to use all cases of program variables and (negated) literals. For sake of brevity of the example we will only use a literal and for the case of a program variable the sub case with a local variable. That gives us two cases, because 1 (number of new cases the #nse is split into) * 2 (number of cases we show for the #se) = 2. Now a literal could also have many different types which could substantially increase the amount of cases we get but as we are using a multiplication on the #se there are only two possibilities left for which it can be evaluated, which is int and float. As these cases are the same apart from changing the occurrences of int into float or fl we only show the case of an integer literal. The lefthandside schema variable #lhs also contributes all its possible cases (multiplicatively) to the number of cases to check, so here we only show it for a local variable. So now here are the two cases, first with the integer literal as #se, then with the local variable for it, also of type int. So in total we end up with a huge amount of cases which is: 5 (#lhs)* 6 (#nse) * 4(#se) = 120.

- First, we look at the case of #se being an integer literal, #nse returning an integer and #lhs being an integer local variable.

  Π_{sk} = lhsName:Name
  = eval(nseEN:ExpressionName, int−result) ∗ int (seInt : Int) ;

  Π'_{sk} = vName:Name = eval(nseEN:ExpressionName, int−result) ;
  lhsName:Name = vName:Name ∗ int(seInt:Int) ;

  We run those code sections in the configuration with:

  – [lhsName:Name, lhsLoc:Location] ∈ e,

  – [lhsLoc : Location , int (lhsVal : Int)] ∈ m.

- Second, we look at the case of #se being an integer local variable, #nse returning an integer and #lhs being an integer local variable.

  Π_{sk} = lhsName:Name
  = eval(nseEN:ExpressionName, int−result) ∗ seName:Name ;

  Π'_{sk} = vName:Name = eval(nseEN:ExpressionName, int−result) ;
  lhsName:Name = vName:Name ∗ seName:Name ;

  We run those code sections in the configuration with the following additions, where the difference to above is that additionally seName is in the local environment and its location is in the memory:

  – [lhsName:Name, lhsLoc:Location] ∈ e,

  – [lhsLoc : Location , int (lhsVal : Int)] ∈ m,

  – [seName:Name, seLoc:Location] ∈ e,

– [seLoc:Location, int(seVal:Int)] ∈ m.

□

2.2.9 Equivalence of Configurations Modulo New Variables

Simple example for motivation:

Example 2.2.4
Given two different states, where #v is a new variable:

1. #v = 4 , #x = 17
2. #x = 17

Then these two states are, when we take the new variable #v away, identical. Actually those state descriptions are more complicated in the real MJS but the principle stays the same. □

We define what equivalence modulo new variables means by first giving a more general definition:

Definition 2.2.5 (Equivalence modulo a set of variables)
We define $==_X$ in the following way as an equivalence: Two states s_1 and s_2 are **equivalent modulo the variables in the set X of variables**, i.e. $s_1 ==_X s_2$, if they are equal when we delete all variables in the set X. □

Deleting all variables in the set X means that wherever one of the variables which are part of the set of variables X appears in an environment, i.e. in that environment the variable is mapped to some location, this mapping is deleted. All locations encountered that way are memorized. For each of those memory locations, the memory location with its associated value or object reference inside is deleted, too.

After both states have been modified according to this description the resulting states have to be identical, i.e. equal with Maude's equality "==" for them to be equivalent modulo X.

Then we use that to define equivalence modulo new variables. New variables are a special sort in the MJS about which you will learn more in subsection 3.1.2.

Definition 2.2.6 (Equivalence modulo new variables)
Given two states, we define a set X which consists of all new variables ap-

pearing in any of the two states.

Then two states are **equivalent modulo the new variables** if the two states are equivalent modulo X as defined above. $\square$

The definition of equivalence modulo new variables is useful for comparing the resulting state of two code segments being executed when one was created from the other by a taclet with a varcond new var part in the taclet. Then this new variables only exists in one of the two states. It can also not be used again at any time after that code segment has been executed, so keeping the new variables and their values is pointless and they can be ignored after the code segment created by the taclet was executed. Thus it is sufficient to check whether two states are equivalent modulo the new variables which are declared in the taclet to see if the two code segments yield the same results. Especially it is not necessary to require the two states to be identical.

In practice there is a Maude operator created by us which can take care of this. It does not take a set of "new variables" however but instead the new variables and their locations have a special sort of their own and have to be put into the initial configuration accordingly. More on this can be found in the chapter about changes to the Maude Java Semantics. This in effect purges all new variables and the locations they have been mapped to, together with whatever values have been put into these locations, from the configuration.

Details on the generic configuration can be found in the preliminaries chapter in Section 1.2.3.

2.3 Restrictions on the Correctness Statement

The missing and incomplete features in this section are not allowed to appear in code we validate. This means that our approach is only correct when none of these features happen to be used by the validated code. So we have only shown the taclets for this subset of Java code which does not use any of the missing or incomplete features.

We now provide lists of the features which are problematic.

2.3.1 Missing Features

There are a couple of Java constructs which are not implemented within the MJS and so we cannot make use of them. All of them are given in the following list:

- throw
- try-catch-finally
- switch
- break
- continue
- conditional expression
- interfaces
- method overloading

As the MJS does not handle exceptions we need to think about how much that limits our correctness statement. Actually as the Java language elements throw, try, catch and finally are missing we are only restricted to taclets where none of these elements appear. In case some code throws an exception which is not caught and we show the taclet it appears in to be correct that is still valid. This is because in a case, where an expression which could potentially throw an exception appears, and our method says that it is correct, all such expressions are executed in exactly the same states for each code segment. Therefore if one expression throws an exception, the other will do so too and thus their resulting states are equivalent, too.

We have, in our extensions of the MJS, given a way for throwing some exceptions but it is limited. We simply allow critical expressions to throw exceptions without catching them. This has the advantage that we will not get any false positives, i.e. we will not show a taclet correct when it is not. This is because even in the case that both code sections throw the same exception they cannot be completely executed as the thrown exception stops the execution and therefore the comparison fails. It was also an easy feature to add, compared to a full exception handling mechanism at least.

As interfaces are missing we can only guarantee the correctness of taclets for code where no interfaces are used.

2.3.2 Incomplete Features

There are also some features which partially work but do not completely adhere to the proscriptions of the JLS.

- method calls in general, they do not pay attention to the differences of private and public and static methods,
- object creation, it needs the workaround for a default constructor given in Section 4.1.6,
- static initialization, it is completely done at the start which is wrong. The JLS requires static initialization to happen later and not for all classes at once.

Static initialization is not handled correctly by the MJS, so our correctness statements can only ever apply when no static initialization is necessary.

In addition, the meta-constructs, like #typeof, pose some problems. We have removed declarations which include #typeof from the code they appear in. Removing these declarations does not matter as such declarations exist for new variables only and new variables are integrated into the configuration beforehand by our configuration generation mechanism. This point thus boils down to the fact that the code from which these declarations are removed is by itself not syntactically correct, but it is syntactically correct in the states we execute it as those new variables are already internal to these states.

Termination does not put any limits on our approach as whenever we can prove something correct that means that all expressions are executed in the same states and thus if one does not terminate the other does not either and in the semantics of KeY two non-terminating pieces of code are equal.

The MJS also only supports integers and no other forms of numbers. Additionally the mathematical integers are used and not the Java integers. Even though floats are partially declared in the MJS there are no operations defined for them so we can not make use of them.

3 Lifting the Semantics

Here we describe all the extensions which are not immediately related to the Java Language Specification. These have been used by us to get easier access to all parts of the configuration and to get functionality needed to handle expressions for example. We also use these extensions to automatically handle the equivalence of two programs modulo new variables for instance.

In short, the need for this section arises from the fact that we must compare two pieces of code given only as schemas using a semantics for concrete code. So we have to lift the semantics to be able to compare those schematic pieces of code. This is important to keep in mind to see why all the little details which follow are necessary.

3.1 Preparations

In this section we will see a few preparatory constructs which will be used later and knowing them first should be helpful.

3.1.1 Pausing the Execution

There is one general extension which we put in for the purpose of easier access to the complete resulting configuration because we do not want to have only the output left after the execution. A program ending with stop does one extra rewrite to get an output from the last configuration and leaves everything else out. As we consider only parts of programs we might want to execute more code after the block we are currently interested in and for that reason we need to retain the whole configuration including the memory to make that possible.

Thus, we introduce a new operator, to be put into the continuation, named pause $->$.

```
op pause -> _ : Continuation -> Continuation .
```

If this is the first item on the continuation, then no further equations or rules are applicable inside the run (...) where pause −> ... appears. This is because it is a new operator for which the old semantics has no rules and we have not given any rules to work on this either. Therefore it does what its name suggests, i.e. pause the execution, until it is removed (by the user for example). Now one can take a look at the complete configuration including the memory. One could also continue with the rest of the code by simply removing the pause −> from the continuation. Therefore pause has multiple advantages, for our purposes, over stop.

3.1.2 Handling New Variables and Equivalence Modulo New Variables

New variables are a concept of the taclet language of KeY and their semantics is explained in more detail in Section 2.2.9. In short, they are given with respect to a certain code segment and are *new* to that segment, i.e. they do not appear anywhere inside the segment. In the Maude Java Semantics all variables are bound to some locations by environments and these new variables will be bound to *new locations*, i.e. locations which were not used before and also have a special sort in our Maude specification.

Example 3.1.1
A simple example using a new variable is the third example from Ex. 1.1.2, in which #lhs is a schema variable of type lefthandside, #nse is a SV of type nonsimple expression and #se is a SV of type simple expression.

```
compound_multiplication_1 {
  find (#allmodal{{.. #lhs=#nse * #se; ...}}(post))
  varcond (typeof(#nse) #v new)
  replacewith (#allmodal{{.. #typeof(#nse) #v=#nse;
                             #lhs=#v * #se; ...}}(post))
  heuristics (simplify_prog)
 displayname "multiplication"};
```

For now the interesting part is the varcond one, where #v is declared to be a new variable of the same type as #nse. In the *replacewith* part this new variable is used and because of the requirement of being *new* we know that it is not the same as any of the previously existing variables inside the nonsimple expression #nse and it is also not the same as #lhs or inside #se. This taclet facilitates splitting complex expressions into all their simple parts. □

To make Maude aware of the new variables, we first define sorts for new variables and new locations and put them in relation to already existing sorts:

```
sort TacletNewVarName .
subsort TacletNewVarName < Name .

sort TacletNewLocation .
subsort TacletNewLocation < Location .
```

This means that every usual operation defined on locations and variables, i.e. names, can be performed on elements of these new sorts. We can use the fact that they are members of a special subsort to define extra rules which take care of special properties of new variables and their locations. For example new variables, or rather their locations, are not affected by the side effects of the old code segment. This will be detailed in the next subsection.

Constructs to Describe Equality

Now we want to concentrate on using the handle we have on new variables to facilitate finding out whether two configurations are equivalent. There are a couple of constructs needed for this. We start with the operator which allows for comparing two program runs, where a program run is having that program code as part of a valid configuration, which are of sort Output in the Maude Java Semantics, to see whether they have the same resulting state. It waits until both computations have finished, recognizable by having pause as the first element on the continuation. Then it compares the two results with Maude's "==" equality operator which is automatically created for each and every module and checks whether the terms are equal modulo the equations and built-in axioms declared for that term.

```
var K : Continuation .
vars cnt cnt' : Context . vars state state' : MyState .
op compareResult : Output Output -> Bool .
eq compareResult(run(c(k(pause -> K), cnt), state) ,
                 run(c(k(pause -> K), cnt'), state') )
      = run(c(k(pause -> K), cnt), state)
        == run(c(k(pause -> K), cnt'), state') .
```

This does not suffice if new variables appear in one of the two results, which is the common case for us. An example for this is the taclet in the example above. We therefore remove all new variables from all the environments as well as the new locations from the stores, i.e. the memory. The first operators' defining equations will be given further down with a few extra comments. For now, think of removeNewVarsLocs as doing what the name suggests, i.e. removing the new variables and locations. Also removeNewVarsLocs removes itself when it has done all possible work on the term which is its argument. We can then define the second operator in terms of the first. Note that all removing action will be done before the removeNewVarsLocs operator removes itself. So only after both removeNewVarLocs are finished in their places in the equation below, it will be possible for compareResult to do its work according to its definition from above.

```
op removeNewVarsLocs : Output -> Output .
op compareResultsModNewVars : Output Output -> Bool .
eq compareResultsModNewVars
        (run(c(k(pause -> K), cnt), state) ,
         run(c(k(pause -> K), cnt'), state') )
   = compareResult
        (removeNewVarsLocs
           (run(c(k(pause -> K), cnt), state)),
         removeNewVarsLocs
           (run(c(k(pause -> K), cnt'), state'))) .
```

Removing new variables and locations

In the following we describe the removal process of new variables and locations in detail. There are four equations necessary. Equation 1 requires a new variable to be mapped to a new location inside the local environment and that new location to be mapped to any value in the memory. In such a case the equation removes both mappings. Equations 2 and 3 remove new variables from the *configuration snapshots* that have been taken during the run. Configuration snapshots are a way to save the current state of the memory and relevant environments for later reference in what state an expression has been executed (see Section 3.2.6). Equation 2 removes the new variable to new location mapping from the environment within the snapshot while equation 3 removes the new location to some value mapping from the memory of a snapshot. Removing these new variables is important as the value of the new variables might be different between two runs or a new variable might not even exist in one of the cases. It is safe to remove the new variables and be sure that their presence did not change anything about the evaluation of expressions because the expressions did not depend and did not write on those new variables for the simple reason that the new variables are new w.r.t. the old code, of which the expression is a part. Equation 4 finishes the process when the first three are not applicable any more, as can be seen by the [owise] tag.

Equation 1 is enough to remove all occurrences of new variables and locations from the configuration outside the snapshots because the new variables and locations can only appear together in this fashion as environment entry [Variable, Location] and memory entry [Location, Value] because they are only put into initial configurations by us. At the end, or pause, of a run one is always at the same method call level as at the beginning so the environment is retained and at best enlarged. The business about putting the new entries in there in this fashion is done by our method to create the initial configurations which only ever creates a new variable with a new location and puts it into the environment and memory in this way. Thus there can not be an unpaired memory or environment entry. There are other environments apart from the local environment but again no new variables can appear there as we do not put them there. For removing the new variables and their locations from the snapshots this does not necessarily hold because a snapshot could be taken within a method call, which could give a snapshot with quite a different environment in which possibly no new variable exists but the new locations are still in the memory of the snapshot. Therefore they have to be removed on their own and do not necessarily appear in such pairs.

```
var TNVN : TacletNewVarName . var TNL : TacletNewLocation .
vars CT CT' CT'' : CType . var V : Value . Var M : Store .
var Env : Env . var OE : ObjEnv .
```

1. equation:

   ```
   eq removeNewVarsLocs(run(c(e([TNVN, TNL] Env), cnt),
                              m([TNL, V] M), state))
        = removeNewVarsLocs (run(c(e(Env), cnt),
                                     m(M), state)) .
   ```

2. equation:

   ```
   eq removeNewVarsLocs(run(
          snapshots((snap(N), c(e([TNVN, TNL] Env), cnt),
                     state), SNL), state'))
   ```

```
          = removeNewVarsLocs(run(
              snapshots((snap(N), c(e(Env), cnt),
                          state), SNL), state')) .
```

3. equation:

```
eq removeNewVarsLocs(run(
     snapshots((snap(N), m([TNL, V] M),
                 state), SNL), state'))
     = removeNewVarsLocs(run(
         snapshots((snap(N), m(M),
                     state), SNL), state')) .
```

4. equation:

```
eq removeNewVarsLocs(run(state))
     = run(state) [owise] .
```

Now there is one point remaining which is worth mentioning and that is variable re-declarations. It could happen that by mistake somebody could maybe try to re-declare a new variable and would then create a normal variable with the same name instead. Then the entry for that normal variable in the environment would replace the original environment entry that included the new variable according to the semantics.

A new location can only be removed together with its new variable as you can see in the first equation above. This would mean that then the new location in the memory could not be removed any longer as there is no corresponding new variable anymore.

Luckily it can not happen that a re-declared variable overwrites a new variable. The reason is found in the way these new variables are introduced. They are only ever given as variableName:TacletNewVarName which is a format and sort any normally declared variable can never have. This is because those are of sort Qid, i.e. they have to be a quoted identifier and then they are of sort Name and not TacletNewVarName.

As we only look at syntactically correct Java code (see Section 2.1) there can not be any re-declarations (of local variables over local variables).

So in whole, if two starting configurations s and s' exist they can be checked for equivalence modulo new variables by typing the following in Maude, after only loading the extended Maude Java Semantics. Creating those starting configurations is another task which will be explained later in this work, in chapter 5 to be precise.

```
rew compareResultsModNewVars(s, s') .
```

The discussion of why removing the new variables and locations is acceptable can be found in the chapter about correctness, i.e. Chapter 2.

3.2 Lifting for Expressions

In this section we look at the handling of schematic variables of type expression and nonsimple expression in a concrete semantics and will end up with lifting the semantics to be able to work with schematic code.

3.2.1 Rules for Applying a Side Effect Schematically

In this subsection a lot of operators will be used which have not yet been declared and no defining equations have been given for them yet. Whenever such operators occur we describe informally what they are supposed to do in the accompanying text and their precise definitions will follow over the Sects. 3.2.2 through 3.2.7. So if you are eager to know what some operator really does you can just browse ahead and look that up before you return to the equation where it appears.

Schematic code with the possibility of having a side effect will be written as in the Maude rules of this subsection and its execution has to be done by a rule because it basically writes to the memory for the unknown side effect. Also at the moment of executing an expression the snapshot of the configuration is created and all the needed parts, i.e. the memory and the local environment as well as the current object, are copied there. The rules take eval out and leave another construct, the evalResultAfterCompleted, on the continuation which will only be executed after the side effect has had its effect on all memory elements and thus blocks subsequent code from interfering and it leaves the value given to the side effect on the continuation.

evalResultAfterCompleted can disappear with either noStore left to manipulate by the side effect change or with some change still necessary, but that change needs to be done on a store which is not concrete so that it has to be postponed and has to stay in the memory that way. That non-concrete store could not have been accessed before, because it does not contain concrete elements. Thus it is of no concern that now it cannot be accessed by the normal rules anymore because of the side effect memory change operator which wraps it and which is not yet applied and cannot be applied.

All of the rules below are (as a whole) responsible for the evaluation of expressions which could have side effects. Each covers a special case as described at each rule.

The side effects take place immediately. After use of any of these rules no one (i.e. no other thread) can get to any of the old values so this is a completely uninterruptible change on the memory with all cells affected at the same time. This is not what usually happens when an expression has more than one side effect, though for a sequential program this is a sufficient model!

Expression with a Result of Some Primitive Type

The following rule is applicable when the result of the expression is of any primitive type. The eval operator is the one which is put into the code during the initial configuration generation and it states the name of the expression which is evaluated as well as the type of its result. The rule then creates a generic value of that primitive type and puts it into the evalResultAfterCompleted operator (see Section 3.2.2) and it is just holding the value in place and waits until the change on the memory is done. Then it does the usual sideEffMemChange on the memory (see Section 3.2.4) which puts the change into every memory cell. It also takes a snapshot of the memory, the local environment, and the current object. So snapshots is a list of the already existing snapshots and the nextSnapshot is the index number of the snapshot which is taken next. The operator snap just holds the index number for the snapshot it belongs to (see Section 3.2.6 for these three operators). Also there is the effectof _ in _ operator which is the schematic skolem constant for the effect the given expression has when executed in the recorded snapshot (see Section 3.2.7) together with locs and vals which are operators that extract the list of locations which are changed and the list of values those locations are changed to from the former operator.

```
var EN : ExpressionName .
var PR : PrimitiveExpressionResultType .
var K : Continuation . var M : Store .
var SNL : SnapshotList . var Nn : Nat .
```

```
rl [SideEffect_primitive] :
  c(k(eval(EN, PR) -> K), cnt), m(M) , snapshots(SNL),
    nextSnapshot(Nn)
    => c(k(evalResultAfterCompleted
             (resultof EN in snap(Nn) GetsResType PR)
          -> K), cnt),
       m(sideEffMemChange
          ([locs(effectof EN in snap(Nn)) ;
            vals(effectof EN in snap(Nn))],
           M)
          ) ,
       snapshots(SNL , (snap(Nn), (c(cnt), m(M)))),
       nextSnapshot(Nn + 1) .
```

Expression with an Object Result with No Explicit Attribute

The next rule is used when the result type is an object type, CT, and it does not require an explicit attribute in the memory. Then the generic object is given as result value. Its dynamic type is the same as its static type. This is because any memory write with it will check whether the dynamic type is a subtype of the generic type which it can not decide for a generic dynamic type as we would want here. Of the new operators here obj−result (see Section 3.2.7) just states that the result is an object of this type as can be seen on the right-hand side of the rule. The RestObjEnv is just the schematic placeholder for the rest of the object environment of the given object (see also Section 3.2.7).

```
var CT : CType .

rl [SideEffect_obj_noExplicitAttribute] :
  c(k(eval(EN, obj-result(CT)) -> K), cnt),
    m(M) , snapshots(SNL),
    nextSnapshot(Nn)
    => c(k(evalResultAfterCompleted(
           o(CT, CT, RestObjEnv(EN, snap(Nn))
             ))
             -> K), cnt),
       m(sideEffMemChange
          ([locs(effectof EN in snap(Nn)) ;
            vals(effectof EN in snap(Nn))],
           M
          )
        ) ,
       snapshots(SNL , (snap(Nn),
         (c(cnt), m(M)))),
       nextSnapshot(Nn + 1) .
```

Expression with an Object Result with Explicit Attribute and a Former Side Effect Memory Write Remains

The following rule can be employed when the result of the expression is to be an object which has type CT and an attribute X which itself is of some primitive type. Also there has been an expression which left a side effect in the memory before. Then this creates an object as the rule above does but this object also explicitly has the given attribute in its environment mapped to a location. That location is mapped to some value in the memory next to the unconcrete store because it was a part of that before and is made explicit here. It is not a new creation! There we also see the sideEffMemChangeNotConcrete on the memory (see Section 3.2.4) and it allows to do the side effect change on what was before part of the unconcrete store. Also we use ResAttLoc and AttVal which are the schematic skolem constants for the location of the attribute of the resulting object and the value of that attribute (see Section 3.2.7). In addition there is the GetsResType operator, which is explained in the same subsection as the two above operators, and which guarantees that the value which is its first argument is given as the type which is its second argument.

```
var X : Name .

rl [SideEffect_obj_notFirstSE] :
  c(k(eval(EN, obj-result(CT, X, PR)) -> K), cnt),
    m(M sideEffMemChangeNotConcrete(SEL, M')) ,
    snapshots(SNL), nextSnapshot(Nn)
    => c(k(evalResultAfterCompleted(
           o(CT, CT, RestObjEnv(EN, snap(Nn))
             (CT, [X, ResAttLoc(EN, snap(Nn), X)])))
             -> K), cnt),
        m(sideEffMemChange
           ([locs(effectof EN in snap(Nn)) ;
             vals(effectof EN in snap(Nn))],
            M
```

```
              sideEffMemChangeNotConcrete(SEL,
                  M'
                  [ResAttLoc(EN,snap(Nn), X) ,
                   AttVal(EN, snap(Nn), X) GetsResType PR])
           )
          ) ,
         snapshots(SNL , (snap(Nn),
           (c(cnt),
            m(M sideEffMemChangeNotConcrete(SEL, M'))))),
         nextSnapshot(Nn + 1) .
```

Expression with an Object Result with Explicit Attribute as First Side Effect

The next rule is applicable when the result of the expression is to be an object which has type CT and an attribute X which itself is of some primitive type. Also there is no sideEffMemChangeNotConcrete left in the memory so this is the first side effect. We know that because OldSideEffInside is not true for this memory according to the condition. It is basically a predicate that checks whether there is a sideEffMemChangeNotConcrete(...) inside the store (see Section 3.2.5).

```
crl [SideEffect_obj_FirstSE] :
  c(k(eval(EN, obj-result(CT, X, PR)) -> K), cnt), m(M) ,
    snapshots(SNL), nextSnapshot(Nn)
    => c(k(evalResultAfterCompleted(
        o(CT, CT, RestObjEnv(EN, snap(Nn))
            (CT, [X, ResAttLoc(EN, snap(Nn), X)])))
            -> K), cnt),
       m(sideEffMemChange
           ([locs(effectof EN in snap(Nn)) ;
             vals(effectof EN in snap(Nn))],
            M  [ResAttLoc(EN,snap(Nn), X) ,
                  AttVal(EN, snap(Nn), X) GetsResType PR])
           ) ,
        snapshots(SNL , (snap(Nn), (c(cnt), m(M)))),
        nextSnapshot(Nn + 1)
  if OldSideEffInside(M) =/= true .
```

Expression with an Array Result with Explicit Member and a Former Side Effect Memory Write Remains

The following rule can be applied when the result is an array of some type and has an explicit member at the position of the integer I. There has been some side effect in the memory before. Then the result is an array of the given type with a generic ArrEnv except for the integer I which has a mapping to some generic location and at that location there is a generic value. This Location-Value pair is an argument of the sideEffMemChangeNotConcrete and next to the unconcrete memory. Here we have the ResArrLoc and ArrVal which are very similar to the above mentioned ResAttLoc and AttVal (see Section 3.2.7) and describe the location of the array element of the resulting array and the value of that. Also there is PrimResType used (see Section 3.2.7) and it just does a conversion of the type it gets to an element of the PrimitiveExpressionResultType which the GetsResType operator needs.

```
rl [SideEffect_arr_NotFirstSE] :
  c(k(eval(EN, arr-result(T, I)) -> K), cnt),
    m(M sideEffMemChangeNotConcrete(SEL, M'))  ,
    snapshots(SNL), nextSnapshot(Nn)
    => c(k(evalResultAfterCompleted(
         a(T, RestArrEnv(EN, snap(Nn))
            [I, ResArrLoc(EN, snap(Nn), I)]))
             -> K), cnt),
       m(sideEffMemChange
            ([locs(effectof EN in snap(Nn)) ;
              vals(effectof EN in snap(Nn))],
             M
              sideEffMemChangeNotConcrete(SEL, M'
                  [ResArrLoc(EN,snap(Nn), I) ,
                   ArrVal(EN, snap(Nn), I) GetsResType
                      PrimResType(T)])
            )
         ) ,
        snapshots(SNL , (snap(Nn), (c(cnt),
       m(M sideEffMemChangeNotConcrete(SEL, M'))))),
        nextSnapshot(Nn + 1) .
```

Expression with an Array Result with Explicit Member as First Side Effect

The next rule can be used when the result is some array and has an explicit member at the position of the integer I. There are no remainders of an earlier side effect in the memory. Then the result is an array of the given type with a generic ArrEnv except for the integer I which has a mapping to some generic location and at that location there is a generic value.

```
crl [SideEffect_arr_FirstSE] :
  c(k(eval(EN, arr-result(T, I)) -> K), cnt), m(M),
     snapshots(SNL), nextSnapshot(Nn)
     => c(k(evalResultAfterCompleted(
          a(T, RestArrEnv(EN, snap(Nn))
             [I, ResArrLoc(EN, snap(Nn), I)]))
              -> K), cnt),
        m(sideEffMemChange
             ([locs(effectof EN in snap(Nn)) ;
                vals(effectof EN in snap(Nn))],
               M  [ResArrLoc(EN,snap(Nn), I) ,
                    ArrVal(EN, snap(Nn), I) GetsResType
                      PrimResType(T)])
             ) ,
        snapshots(SNL , (snap(Nn), (c(cnt), m(M)))),
        nextSnapshot(Nn + 1)
    if OldSideEffInside(M) =/= true .
```

Expression with an Array Result with No Explicit Member

Finally the following rule is executed when the result is an array of some type and we do not need an explicit member of it.

```
rl [SideEffect_arr_noindex] :
  c(k(eval(EN, arr-result(T)) -> K), cnt),
    m(M) , snapshots(SNL),
     nextSnapshot(Nn)
     => c(k(evalResultAfterCompleted(
          a(T, RestArrEnv(EN, snap(Nn))))
             -> K), cnt),
        m(sideEffMemChange
             ([locs(effectof EN in snap(Nn)) ;
            vals(effectof EN in snap(Nn))],
           M
         )
       ) ,
     snapshots(SNL , (snap(Nn), (c(cnt), m(M)))),
     nextSnapshot(Nn + 1) .
```

3.2.2 Helper Operator to Finish the Expression Evaluation

As eval has been defined by the rules above we only now see the declaration of it which is that it takes an ExpressionName en and a ExpressionResultType rt with the following meaning: If eval(en, rt) is the first command on a continuation en is executed yielding a result of the type rt and as usual creating side effects. See Section 3.2.1.

The conditional equations for evalResultAfterCompleted are given below which can only work if the result value does not contain a GetResType anymore and there are no more memory changes, due to the side effect, left to be done. That can be seen by having noStore under the sideEffMemChange and in the other case already having the sideEffMemChange reduced to sideEffMemChangeNotConcrete. The UndefValType predicate is responsible for checking whether there is a GetResType left inside the value. It is explained in Section 3.2.5. It basically makes sure that the value is finally typed and does not have a type definition left inside.

```
sort ExpressionName .

op eval : ExpressionName ExpressionResultType -> StExp .
op evalResultAfterCompleted : Value -> StExp .

op sideEffMemChange : SideEffectList Store -> Store .
op sideEffMemChangeNotConcrete : SideEffectList Store
                                         -> Store .

var SEL : SideEffectList .

ceq c(k(evalResultAfterCompleted(V) -> K ), cnt),
   m(M sideEffMemChange(SEL, noStore))
     = c(k(V -> K), cnt),
       m(M)
   if UndefValType(V) =/= true  .
```

```
ceq c(k(evalResultAfterCompleted(V) -> K ), cnt),
    m(M sideEffMemChangeNotConcrete(SEL, M'))
      = c(k(V -> K), cnt),
        m(M sideEffMemChangeNotConcrete(SEL, M'))
     if UndefValType(V) =/= true .
```

The following operator is not directly related to expression evaluation but only to what can take the place of an expression. A value is a sub case of expression as well as of nonsimple expression. To be able to put that value into code it needs to be an expression and that is what resIsValue does in a fashion similar to the way #i does for integers and #b does for booleans.

```
op resIsValue : Value -> Exp .
eq k(resIsValue(V) -> K) = k(V -> K) .
```

3.2.3 Extended Conditional Values for Schematic Side Effects

In the KeY taclets we are facing schematic expressions of certain types. These can have side effects about which we do not know anything and they will return something of the type mentioned above, if they terminate and do not end with an exception. Modeling the effects of such an expression with side effects in Maude's explicit memory representation is described here. The reason why this is enough can be found in Chapter 2. First of all we can note that a normally terminating expression is completely characterized by the side effects it has on the configuration and the resulting value which all depend on the state of the configuration the expression is started in. Also note that the new variables introduced above can not be in any way affected by these side effects because they are by definition new w.r.t. the code given and thus there can be no reference to them in there and subsequently no change to them.

Example 3.2.1
Let us start with an easy example, say we want to execute the code:
`int a = 0; int b = 0; #v = #e;`
where #e is a schema variable of type expression, i.e. any possible expression could take its place. Modeling this with a side effect and a result one could replace #e by sideEffectAndResult(var1 $\rightarrow$ val1, resultval), where that should change the variable var1 to the value val1 and give resultval as its result. Now var1 could either be a or b and it could put any integer into those variables. That is actually not enough. Another possibility for #e is that it could be

an expression which changes a as well as b. So in general we do not know how many side effects there are so we use a list of side effects with no fixed length. This list also depends on the state of the configuration in which the expression is started. □

First we define a list of side effects which is made up of two lists, a list of locations and a list of values, with the meaning that each location from the location list gets assigned the corresponding value from the value list.

```
sort SideEffectList .
op [_;_] : LocationList ValueList -> SideEffectList .
```

The expressions we are concerned with are schematic, meaning we do not know their concretization in advance and then we also do not know the set of side effects they will introduce. To allow for this we decided to present extended conditional values, that is values which are set depending on where they were stored at the time of the expression execution. They can be used like any other values even if they look a bit complex at first sight. As we aim at comparing two program runs it is obvious that no further evaluation of the conditional values will be necessary. To allow for an easier presentation we omit some of the detail when facing specially typed values, like integers, booleans and strings but only present an operator for integers. This operator even has versions for CTypes and ObjEnvs. The first operator has the meaning that if the given location is equal to one of the elements of the location list the construct gets the value of the corresponding element of the value list, otherwise it keeps the original value, which is the last argument. The second operator is very similar except for the fact that it does not take a Value as fourth argument but an Int and what it creates is also an Int and not a Value. We refer to Section 3.2.4 why we need to do this.

```
op _in_??_::_      : Location LocationList ValueList Value
                     -> Value .

op _in_??_::_int   : Location LocationList ValueList Int
                     -> Int .
```

This following example motivates how the operator works. The operator is similar to a conditional expression with the difference that the boolean part is limited to the inclusion test of a location in a list of locations and that the result is either, on first glance, a list of values or a value. In case the list of values is selected the result is actually the value in that list at the position that the given location was found at in the location list. This example is just for motivating the operator and to show the way it should be viewed, it does not replace a rigorous definition. There will actually be no definition on how to evaluate this construct as that will not be necessary for our purposes.

Example 3.2.2
Let l1 and l2 be two distinct locations, l2 is then also a location list with just one element. Let v2 be a value list with as many elements as the location list, i.e. one element, which is the value v2, and let v1 be a value. Now the result of the above operator applied to some cases would be:

l1 in l2 ?? v2 :: v1 yields v1 because l1 is not an element of the list l2.

l2 in l2 ?? v2 :: v1 yields v2 because l2 is an element of the list l2. It is the first element and thus the result is the first element of the value list v2 which is the value v2.

Adding larger location lists l1 , l2 and l2 , l1 and a larger value list v11, v12 we can see that

l1 in l1 , l2 ?? v11, v12 :: v1 yields v11 because l1 is the first element of the location list and v11 is the first element of the value list,

l2 in l1 , l2 ?? v11, v12 :: v1 yields v12 because l2 is the second element of the location list and v12 is the second element of the value list,

l2 in l2 , l1 ?? v11, v12 :: v1 yields v11 because l2 is the first element of the location list and v11 is the first element of the value list.

With another distinct location l3 we can finally see that

l3 in l1 , l2 ?? v11, v12 :: v1 yields v1 because l3 is not an element of the location list. □

In case you wonder what this is useful for: As we have seen in example 3.2.1 one side effect, for which a simple conditional expression would have sufficed, is not enough to describe an expression in general, because it can have multiple side effects. To describe these multiple side effects we use the extended conditional value operator we just defined and do so with the following meaning:

Given a Location L0 at which the Value V0 is stored and we have a side effect which changes all elements of the LocationList “$\bar{l}$” to the element of the ValueList “$\overline{v}$” then at the Location L0 in the memory section of the configuration we would get the new Value “L0 in $\bar{l}$?? $\overline{v}$ :: V0”. This has the meaning that if L0 is an element of the list of locations then the resulting value is the value from the value list which corresponds to the location and otherwise, i.e. if L0 is not in that location list, the original value stays unchanged.

As we get a list of locations for a side effect, locs(effectof_in_), and a list of values, vals(effectof_in_), one can now see how this will be used when a SideEffectList has been created to show the effect that side effect had on each memory cell and it is used in subsection 3.2.7.

There is quite a good analogy for these extended conditional values, which is, that a simple conditional value is very similar to an if _ then _ else expression while these extended conditional values are much more like a case statement, where the original value is the base case which is used if no other match is found.

3.2.4 Side Effect Induced Changes in the Configuration Memory

Before we take a look at how exactly the sideEffMemChange works we want to consider another example with an important cue about that operator.

Example 3.2.3
Let us look at the following code:

```
#e[#e0] ++ ;
```

which is transformed to, with #v and #v0 new variables of fitting types,

```
#v = #e ; #v0 = #e0 ; #v[#v0] = #v[#v0] + 1 ;
```

That is the order in which the first statement is to be evaluated according to the KeY taclets. Now, such unknown side effects, like #e and #e0 can have, can not affect new variables by definition. If those side effects could affect new variables we would have the following effect: After #e is evaluated and stored in #v, having had side effects on all other variables, #e0 is evaluated and stored in #v0, and has side effects. These side effects could change #v if changing new variables would be allowed for unknown side effects. On the other hand in the original statement that could not happen, as #e is already evaluated when #e0 gets evaluated and there can be no changes on the value of #e by the execution of #e0. Therefore side effects, created by the generic side effect lists we use, must not be applicable to our new variables. □

The side effect will not change the store when the location is one of the new locations as argued above. If the location is a normal one, i.e. not a new one, its value is updated to be a conditional expression with the given meaning. If there are no concrete location and value pairs left in the memory and the rest is not the noStore, which is treated as described above, then it is wrapped in an extra construct showing that this is memory which is not concrete but still is possibly subject to the change by the side effect. Take a special look at the second equation, here the case of a primitive integer is treated so that afterwards the result is an integer again which is the only acceptable type-correct memory write anyway. There are similar equations in the implementation for the other primitive types (and more) which are not shown here. The third equation also has to check whether the value it is going to treat has a defined type, if that is not the case it has to wait for that to happen by the condition and then one of the other specially typed equations will be usable and used because that equation has the [owise] attribute.

```
var LocaL : LocationList . var Vl : ValueList .
var L : Location . var V : Value .
var I : Int .

eq sideEffMemChange([LocaL ; Vl], [TNL,V] M)
         = [TNL, V]
           sideEffMemChange([LocaL ; Vl], M) .

ceq sideEffMemChange([LocaL ; Vl], [L, int(I)] M)
         = [L, int(L in LocaL ?? Vl :: I int)]
           sideEffMemChange([LocaL ; Vl], M)
      if not L :: TacletNewLocation .

ceq sideEffMemChange([LocaL ; Vl], [L,V] M)
      = [L, L in LocaL ?? Vl :: V]
        sideEffMemChange([LocaL ; Vl], M)
      if not L :: TacletNewLocation
         and UndefValType(V) =/= true [owise] .

ceq sideEffMemChange([LocaL ; Vl], M)
      = sideEffMemChangeNotConcrete([LocaL ; Vl], M)
      if M =/= noStore [owise] .
```

The special treatment of the integer value, as well as the special treatment of all the other primitive types which are not shown here, is necessary for the following reasons. An int(I:Int) is of type Value but it is special within Value in that by the int() the semantics knows explicitly that this value is of sort Int and then the equations defining e.g. + can be applied which are not defined on Values but only on Ints for example.

All of the above changes can be safely performed by equations and do not need rewrite rules because they are, when viewed by the unextended semantics, happening at once. That is because everything which is not yet changed is caught under the sideEffMemChange operator where other memory access rules can not work. Also the evalResultAfterCompleted operator holds the current thread's execution until all memory changes are done. Even other threads can not interfere as the part of the memory which is not yet modified by the side effect is safely wrapped inside the changing operator. This is not a point of concern when working with sequential programs, which we are doing. In case of using multi-threading though this means that there is an uninterruptible, parallel change of a lot of data which is not something that is really possible. But because we are not concerned with multi-threading this is no problem as this special extra operator can not be part of a usual multi-threaded Java program and thus does not harm the capability of executing multi-threaded Java, provided one does not make use of this operator.

To sum up, the so-called new variables can not be affected by the side effects, which is clear, as the code with the side effect is given first and the new variable is selected such that it is new w.r.t. this code.

3.2.5 Delayed Configuration Memory Changes

As we have seen with the rules for applying eval on something in Section 3.2.1 it is possible, and necessary, to put Location-Value pairs under the sideEffMemChangeNotConcrete operator because these Location-Value pairs have been part of the unconcrete store until now but we need to access them so we have to take them out of the unconcrete store. Thus they stand next to the unconcrete store, as argument to the old change operator created by a side effect, if one exists. As follows there are the equations for the case in which the side effect is applied just as usual but delayed because the memory element was unconcrete when the side effect was originally being performed. The first six equations define that application for special types, the conditional equation for all other values. A little bit of help is necessary for the conditional equation in the form of the seventh equation which defines the UndefValType operator. That is a predicate on the kind level which checks whether the GetsResType has been evaluated already in case it was there. If that has not yet been evaluated the Value which includes it can not be pulled outside of the side effect construct in memory. The GetsResType is explained in 3.2.7 but it is appropriate to say a bit more about it now. See the example below the following defining equations for that.

```
---- 1
eq sideEffMemChangeNotConcrete([LocaL ; Vl],
                               [L, int(I)] M)
     = [L, int(L in LocaL ?? Vl :: I int)]
       sideEffMemChangeNotConcrete([LocaL ; Vl], M) .

---- 2
eq sideEffMemChangeNotConcrete([LocaL ; Vl],
                               [L, fl(F:Float)] M)
     = [L, fl(L in LocaL ?? Vl :: F:Float fl)]
       sideEffMemChangeNotConcrete([LocaL ; Vl], M) .

---- 3
eq sideEffMemChangeNotConcrete([LocaL ; Vl],
                               [L, str(S:String)] M)
     = [L, str(L in LocaL ?? Vl :: S:String str)]
       sideEffMemChangeNotConcrete([LocaL ; Vl], M) .

---- 4
eq sideEffMemChangeNotConcrete([LocaL ; Vl],
                               [L, bool(B:Bool)] M)
     = [L, bool(L in LocaL ?? Vl :: B:Bool bool)]
       sideEffMemChangeNotConcrete([LocaL ; Vl], M) .

---- 5
eq sideEffMemChangeNotConcrete([LocaL ; Vl],
                               [L, alloc(T:Type)] M)
     = [L, alloc(L in LocaL ?? Vl :: T:Type alloc)]
       sideEffMemChangeNotConcrete([LocaL ; Vl], M) .

---- 6
eq sideEffMemChangeNotConcrete([LocaL ; Vl],
                               [L, o(ST:CType,
                                     DT:CType,
                                     OE:ObjEnv )] M)
     = [L, o(ST:CType,
             L in LocaL ?? Vl :: DT:CType ctype,
             L in LocaL ?? Vl :: OE:ObjEnv objenv)]
             sideEffMemChangeNotConcrete([LocaL ; Vl],
                                                M) .

op UndefValType : Value -> [Bool] .

---- 7
eq UndefValType(V GetsResType PR) = true .

---- 8
ceq sideEffMemChangeNotConcrete([LocaL ; Vl], [L,V] M)
      = [L, L in LocaL ?? Vl :: V]
         sideEffMemChangeNotConcrete([LocaL ; Vl], M)
    if UndefValType(V) =/= true .
```

Before we get to the example there is a small thing we want to point out about equation 6 above which does a change on an object reference. The static type can not be changed so only the dynamic type and object environment are subject to change. The change in those is very similar,
L in LocaL ?? Vl :: DT:CType ctype and
L in LocaL ?? Vl :: OE:ObjEnv objenv, which looks like it would return the same Value V if L happens to be in the LocationList LocaL and once it is a CType and the other time it is an ObjEnv. In such a case when evaluating the construct the Value V would need to be a pair and the fitting part is taken in each case. This does not matter to us as we never evaluate these constructs.

Example 3.2.4
The GetsResType gets used when looking at the result of an expression to make sure it has the right type. If the result of an expression must be an integer the rules create the following:

```
resultof EN in snap(Nn) GetsResType int-result
```

where EN is the expression name and Nn is the integer index of the snapshot when the expression was executed. This is not yet of the required format which is int(I:Int) but that is what GetsResType assures. When it is executed the above gets changed to the following:

```
int(resultof EN in snap(Nn) int)
```

with resultof_in_int another operator defined in the later subsection 3.2.7 which just means that it is a result but has type Int instead of Value. □

Also we need the information whether there is a non concrete side effect construct left in a store so therefore we define the operator below. This predicate is true on a store when a side effect has already happened and its effect, i.e. a sideEffMemChangeNotConcrete is left in the memory surrounding some store which we consider to be not concrete.

```
op OldSideEffInside : Store -> [Bool] .
eq OldSideEffInside(M
    sideEffMemChangeNotConcrete(SEL, M')) = true .
```

3.2.6 Configuration Snapshots

The actual side effect list depends on the state in which the expression is executed. We use a *configuration snapshot* of a part of the configuration to define exactly when the expression was executed so that this can be compared for two different runs of the expression. For that we need an easy naming of a snapshot which do by just calling it snap(N) with N a natural number. There are two additional StateAttributes in the configuration now, snapshots is the list of snapshots already taken and nextSnapshot is the number of the next snapshot. The notation for lists of snapshots is as usual for lists in Maude and the final operator comprises a name and a state as a snapshot.

```
sort Snapshot .
sort SnapshotName .
sort SnapshotState .
sort SnapshotList .
subsort Snapshot < SnapshotList .

op snap : Nat -> SnapshotName .
op snapshots : SnapshotList -> StateAttribute .
op nextSnapshot : Nat -> StateAttribute .
op emptylist : -> SnapshotList .
op _,_ : SnapshotList SnapshotList
        -> SnapshotList [assoc comm id: emptylist] .
op (_,_) : SnapshotName MyState -> Snapshot .
```

In the MyState in this special case there will be the current memory, the local environment and the object on which the current continuation is executed. The snapshot is meant to hold all parts of the configuration upon which a side effect depends and which could possibly change. The above named three are all such parts. The memory is quite obvious among them, the local environment could be different if the expression is in one case started within a method call and in the other case outside of it. The same holds for the current object which could be changed in that way.

Then one could think that maybe the static attribute environment needs to be put into the snapshot but that is not the case as there is no static initialization in the MJS but all static attributes are created in the beginning. All other parts of the configurations are quite obviously not needed in the snapshot.

3.2.7 Effects and Results of Expressions

To model the list of side effects we say that an expression started in a certain configuration has the effect effectof_in_ as below. The locs and vals operators are used to create the lists of locations and values from the effect of the side effect so they can be used as described above with extended conditional values.

```
sort ExpressionEffect .

op effectof_in_ : ExpressionName SnapshotName
                        -> ExpressionEffect .
op locs : ExpressionEffect -> LocationList .
op vals : ExpressionEffect -> ValueList .
```

Before we go on with the result of such an expression we have to investigate some fundamental things first. Earlier we already looked at some intricate differences between Values and Ints and how they are related on the other hand (see Section 3.2.4). We repeat a bit of that here and go further then. An int(I:Int) is of type Value but it is special within Value in that by the int (...) the semantics knows explicitly that this value is of type int and then the equations defining e.g. + can be applied which are not defined on Values but only on int (...) s and certain other special types. This makes it necessary to differentiate between results of type Value and results of primitive types.

Also it would be much more work if one tried to integrate this differentiation into the rules given for eval (...) which were given in subsection 3.2.1. To avoid this some of the following constructs are used, most noticeably GetsResType. With them, we can just give one rule for all primitive cases of the evaluation of eval and can define the GetsResType to work locally. The only thing we have to be careful about then is that this has to be evaluated before the value of which it is a part can be otherwise used. This can be seen in Section 3.2.2, 3.2.4 and 3.2.5. There it is obvious that GetsResType is evaluated first because of the use of the predicate UndefValType.

Having considered the side effects of an expression we now have to consider the result of an expression. We will use a similar operator called resultof_in_ . Now here we have to again distinguish between the general case of any value when we do not care about exactly which type the value has and those cases where the value is of some specific primitive type, represented by the other four operators which are necessary after the considerations we presented above.

```
op resultof_in_ : ExpressionName SnapshotName -> Value .

op resultof_in_int     : ExpressionName SnapshotName
                         -> Int .
op resultof_in_float   : ExpressionName SnapshotName
                         -> Float .
op resultof_in_string  : ExpressionName SnapshotName
                         -> String .
op resultof_in_bool    : ExpressionName SnapshotName
                         -> Bool .
```

In order to get all these operators working together with the different types we need to do a few things for which we need the operators described below. The sorts ExpressionResultType and PrimitiveExpressionResultType with their subsort inclusion represent the types a result value can take. The operators are all constants used to identify the kind of type. A general resultof_in_ will be transformed to a typed TYPE(resultof_in_TYPE) with TYPE being replaced by the possibilities below when TYPE is primitive. The _GetsResType_ is doing that transformation using the equations below.

```
sort ExpressionResultType .
sort PrimitiveExpressionResultType .
subsort PrimitiveExpressionResultType
            < ExpressionResultType .

op int-result     : -> PrimitiveExpressionResultType .
op float-result   : -> PrimitiveExpressionResultType .
op string-result  : -> PrimitiveExpressionResultType .
op bool-result    : -> PrimitiveExpressionResultType .
op non-primitive-result :
                    -> PrimitiveExpressionResultType .
```

```
op _GetsResType_ : Value PrimitiveExpressionResultType
                   -> Value .

eq resultof EN in snap(Nn) GetsResType int-result =
     int(resultof EN in snap(Nn) int) .
eq resultof EN in snap(Nn) GetsResType float-result =
     fl(resultof EN in snap(Nn) float) .
eq resultof EN in snap(Nn) GetsResType string-result =
     str(resultof EN in snap(Nn) string) .
eq resultof EN in snap(Nn) GetsResType bool-result =
     bool(resultof EN in snap(Nn) bool) .
eq resultof EN in snap(Nn) GetsResType
     non-primitive-result = resultof EN in snap(Nn) .
```

The above types are not all possible ExpressionResultTypes, here are the other, more complex, ones. In the case that the result is an object we allow two ways for that:

1. The first way is for the operator to take a type and an attribute name with its associated type. Then the result is an object of that type and one has explicit access to the given attribute which holds a value of the given result type.
2. The second way is for the operator to just take a type for the object and therefore an object of that type is the result but it has no explicitly accessible attributes.

These two different creation methods are necessary because sometimes we can, by statically looking at the code, find out that the result object of an expression evaluation needs to have a certain attribute. On the other hand there are times where it does not need any attribute and then the object might not have an attribute at all so we can not just use the first operator but need to use the second in case there really exists no attribute for the given object. The second operator does not at all state that there is no attribute for the given class type but only that there is no explicitly accessible attribute given by the operator.

For arrays this is similar, the first case yields an array of the given type and the member at the position given by the integer is explicitly accessible while the second case creates an array of the type with no member being explicitly accessible.

```
op obj-result : CType Name PrimitiveExpressionResultType
```

```
                    -> ExpressionResultType .
op obj-result  : CType -> ExpressionResultType .

op arr-result  : Type Int -> ExpressionResultType .
op arr-result  : Type -> ExpressionResultType .
```

To process this information inside the configuration and keep it generic we need the following helpful operators:

- ResAttLoc which is the location where the attribute of the resulting object is stored,
- AttVal which is the value the attribute has,
- RestObjEnv which represents the rest of the environment of the object with all other (possibly many) attributes in there.

These get into a configuration during the application of the rules defining eval as you can see in Section 3.2.1:

```
op ResAttLoc   : ExpressionName SnapshotName Name
                 -> Location .
op AttVal      : ExpressionName SnapshotName Name
                 -> Value .
op RestObjEnv  : ExpressionName SnapshotName -> ObjEnv .
```

Similarly for arrays we have:

- ResArrLoc is the location of the array element which is explicitly accessible,
- ArrVal is the value of that element,
- RestArrEnv is the rest of the environment.

```
op ResArrLoc   : ExpressionName SnapshotName Int
                 -> Location .
op ArrVal      : ExpressionName SnapshotName Int -> Value .
op RestArrEnv  : ExpressionName SnapshotName -> ArrayEnv .
```

Similar to the resultof_in_TYPE operators above we have the following special operators for the value of the object's attribute. There are also the equations which do the type evaluation similar to above. Then the same happens for the value of the array's element also with the defining equations.

```
op AttValInt     : ExpressionName SnapshotName Name
                   -> Int .
op AttValFloat   : ExpressionName SnapshotName Name
                   -> Float .
op AttValString  : ExpressionName SnapshotName Name
                   -> String .
op AttValBool    : ExpressionName SnapshotName Name
                   -> Bool .

eq AttVal(EN, snap(Nn), X) GetsResType int-result =
     int(AttValInt(EN, snap(Nn), X)) .
eq AttVal(EN, snap(Nn), X) GetsResType float-result =
     fl(AttValFloat(EN, snap(Nn), X)) .
eq AttVal(EN, snap(Nn), X) GetsResType string-result =
     str(AttValString(EN, snap(Nn), X)) .
eq AttVal(EN, snap(Nn), X) GetsResType bool-result =
     bool(AttValBool(EN, snap(Nn), X)) .
eq AttVal(EN, snap(Nn), X) GetsResType
                               non-primitive-result =
     AttVal(EN, snap(Nn), X) .


op ArrValInt     : ExpressionName SnapshotName Int
                   -> Int .
op ArrValFloat   : ExpressionName SnapshotName Int
                   -> Float .
op ArrValString  : ExpressionName SnapshotName Int
                   -> String .
op ArrValBool    : ExpressionName SnapshotName Int
                   -> Bool .

eq ArrVal(EN, snap(Nn), I) GetsResType int-result =
     int(ArrValInt(EN, snap(Nn), I)) .
eq ArrVal(EN, snap(Nn), I) GetsResType float-result =
     fl(ArrValFloat(EN, snap(Nn), I)) .
eq ArrVal(EN, snap(Nn), I) GetsResType string-result =
     str(ArrValString(EN, snap(Nn), I)) .
eq ArrVal(EN, snap(Nn), I) GetsResType bool-result =
     bool(ArrValBool(EN, snap(Nn), I)) .
eq ArrVal(EN, snap(Nn), I) GetsResType
                               non-primitive-result =
     ArrVal(EN, snap(Nn), I) .
```

To decide which PrimitiveExpressionResultType's defining equation from the list above has to be used one needs a PrimitiveExpressionResultType while we only have a Type. Therefore we hereby present the operator PrimResType which extracts that from a normal Type and was used in Section 3.2.1 already.

```
op PrimResType : Type -> PrimitiveExpressionResultType .
var Ty : Type .
eq PrimResType(int)      = int-result .
eq PrimResType(float)    = float-result .
eq PrimResType(String)   = string-result .
eq PrimResType(boolean) = bool-result .
eq PrimResType(Ty)       = non-primitive-result [owise] .
```

3.2.8 Shortcomings of the Schematic Side Effect Handling

There are some problems remaining for the handling of schematic side effects which can not be easily remedied. To illustrate this we can first take a look at an example.

Example 3.2.5
Let us consider two expression which could have side effects, say E1 and E2. If the result of E1 is of an array type and the result of E2 is an integer we could look at the following code: E1 [E2]. Here we get into trouble because to evaluate the array element access we need the array returned from E1 to have an explicitly given element at the integer position which E2 returns. But E2 returns a resultof E2 in snap(X) int where X is some integer which we can not know beforehand. So there is no easy way to make sure that the array E1 returns has an explicit element at that position unless going to considerable length at the analysis of the code during the configuration creation. What happens if the array element is not explicitly there is that the execution simply stops and we therefore do not get a comparison between the two code segments. □

Actually as we know how a starting configuration is built up, we could count the number of expressions in the code which happen before E2, and therefore in the above example find out what integer E2 will return, which is resultof E2 in snap(initSnapCounter + Z) int where initSnapCounter is the uninterpreted constant from the configuration generation which represents the next number a snapshot gets when this execution started and Z is the number of expressions which trigger a snapshot being made before E2 is executed.

Finding that Z is not easy though and using the initSnapCounter seems dubious because it means we exploit our knowledge of the creation process a lot. That aside it would work in practice but it has not been implemented.

This problem is limited to array access for a simple reason. Trying to do something like this with, for example, an attribute access of an object, say E1 . E2, is not possible as in this case E2 would not be an acceptable expression but the whole term is just one expression with one result. As above we can use E1 . a where a is just any variable as then we statically know which attribute E1 needs to have. That it is ok to give the attribute to E1 explicitly is because the compiler would have to have checked that this is a valid access on the static type of E1 and thus this is ok. Even if it would be a non-acceptable access we can not do anything useful about it without exception handling.

In addition to that we are in trouble when using these expressions as booleans. If such an expression is the first element of a conjunction then the second element is only ever executed and evaluated if the expression is either true or "not yet decided". The expression, which returns such a generic value of type bool is always "not yet decided" though! At this point in the evaluation the semantics just sees that it is some generic value and can not match it with false directly so the second element will always be evaluated. As described in Section 4.1.1 that is not the way the boolean conjunction in Java should be evaluated. The right hand side element should only be evaluatend when the first boolean is true.

4 Changes to the Maude Java Semantics

The Maude Java Semantics is not complete and not completely correct in the features that are available and so we had to make some modifications to it. We also add some functionality. In this chapter we will detail what has been modified and why that was necessary. In general the Maude variables appearing in each section will be declared in the first Maude code part they appear in. They will not be repeated in the remainder of that section in most cases even though sometimes they might reappear. So if you find a Maude variable, recognizable by starting with a capital letter, say `X` and do not remember its type just search for `var X`, starting at the beginning of the section you encounter that `X` and you will easily find its declaration.

There are three different sorts of modifications to the Maude Java Semantics:

- There are bug fixes, including fixes to e.g. equations which lead to a deadlock. In that case Maude does not go on with the execution of the program even though it should be possible. There are also bugs in the sense that the execution's result was not meeting the specified result according to the Java language specification.
- There are extensions of the Maude Java Semantics which are necessary so that more features of Java code can be covered.
- There are extensions for the purpose of our work, to get the possibility to easily use some constructs from the taclet language.

4.1 Bug Fixes

In this section we will expose the bugs we have found and give a fix for them.

4.1.1 Mistake in && and || w.r.t. the Java Language Specification

According to the Java Language Specification [GJSB00] [15.22,15.23] for && and for || it is specified that these operators have to be evaluated "lazily", i.e. the right-hand side is only evaluated if the left-hand side is true (resp. false) in the && case (in the || case).

To reflect this the following two equations of the original semantics, for each of the operators, have been removed and instead there are four equations given for each case.

The old, "strict" and therefore faulty, defining equations of &&

```
vars E E' : Exp . var K : Continuation . vars B B' : Bool .
eq k((E && E') -> K)
      = k((E,E') -> && -> K) .
eq k((bool(B),bool(B')) -> && -> K)
      = k(bool(B and B') -> K) .
```

have been replaced by equations which evaluate the whole expression in a "lazy" way as the Java specification requires. The first equation puts the lefthand side expression of the conjunction to the front of the continuation for evaluation and "hides" the second expression behind the conjunction sign. In the case that the evaluation of the first expression yields false the second equation makes sure that the second expression is not evaluated and the result is false. The third equation evaluates the conjunction after both sides have been evaluated. The fourth and last equation only works if none of the others is applicable, i.e. it is not the case that both sides have been evaluated and also the lefthand side was not false. Then it swaps the second expression to the front of the continuation to be evaluated such that the third equation can finish the work afterwards when both sides are evaluated. The fourth equation can thus only be used at most once for every evaluation of &&.

```
eq k((E && E') -> K)
      = k(E -> && -> (E' -> K)) .
eq k(bool(false) -> && -> (E' -> K))
      = k(bool(false) -> K) .
eq k(bool(B)-> && -> (bool(B') -> K))
      = k(bool(B' and B) -> K) .
eq k(bool(B) -> &&-> (E' -> K))
      = k(E' -> && -> (bool(B) -> K)) [owise] .
```

The old, "strict" and therefore faulty, defining equations of ||

```
eq k((E || E') -> K)
      = k((E,E') -> || -> K) .
eq k((bool(B),bool(B')) -> || -> K)
      = k(bool(B or B') -> K) .
```

have been replaced by equations which evaluate the whole expression in a lazy way as required by the Java specification, here basically the same explanation as for the && case holds, except that true is in the second equation instead of false for obvious reasons.

```
  eq k((E || E') -> K)
       = k(E -> || -> (E' -> K)) .
  eq k(bool(true) -> || -> (E' -> K))
       = k(bool(true) -> K) .
  eq k(bool(B) -> || -> (bool(B') -> K))
       = k(bool(B' or B) -> K) .
  eq k(bool(B) -> || -> (E' -> K))
       = k(E' -> || -> (bool(B) -> K)) [owise] .
```

4.1.2 Array Creation Typo

Here the elements of an array of a primitive type are created. The second equation is wrong because instead of making an integer array element, as the left hand side suggests, it creates a float array element, i.e. instead of an int it creates a float. Also the first equation has the exact same lefthand side which is not acceptable. Because of the first equation it is safe to assume that this was a typo and that this should create a float array element like it does in the third equation. The third equation is given by us and replaces the second one in the actual semantics while the first one stays unchanged. The first and second equation have been part of the original semantics. The lines --- original and --- modified are here to show where each of the equations is from. They are comments as usually denoted by --- in Maude.

```
var K : Continuation .
eq k(noVal -> newArray(int) -> K) = k(int(0.0) -> K) .

--- original
eq k(noVal -> newArray(int) -> K) = k(fl(0.0) -> K) .
```

```
--- modified
eq k(noVal -> newArray(fl) -> K) = k(fl(0.0) -> K) .
```

4.1.3 Method Call Error

This is a quite severe problem which leads to non-executability of code and the computation stopping at a method call which should be executable. The problem arises as soon as a method of a super-class is called on an object of a sub-class.

Example 4.1.1
Let us look at a simple example: Two classes A and B are given, with class A having a method m and class A being the superclass of class B. Now we have an object o of type B. This should allow us to call the method m on the object o, that is do something like: `o . m()`. In the given semantics this might or might not work. That is because in this case a search is started which method m we are talking about here, even though in this case we have only one available method. That search looks like this: getMethod(B, m, A{...} B{...}), where the first B is the type of the object on which the method is called, the m is the method name and the set A{...} B{...} is the set of all existing classes, i.e. their specifications, which are searched for the method.

Please note that the getMethod operator yields a multiset of possible candidate methods out of which later on the right one is picked but the mistake happens here already. The if SuperOf(A, B, B{...}) getM(m, A{...}) below has to be read like this: In the case that the predicate is true the result of the getM operator is part of the multiset of possible methods otherwise the line yields the empty element. The second equality below then arises from a refinement of the second line of the intermediate state of the multiset and has to be read accordingly, like the whole second execution order further down. Between two lines which are not separated by a = imagine a multiset union, which we have denoted, as often done in Maude, by juxtaposition, i.e. just using a space between elements.

What happens now is that there are two different orders in which this can be evaluated, first, and correct, is this one:

```
getMethod(B, m, A{...} B{...})

  = if SuperOf(A, B, B{...}) getM(m, A{...})
    getMethod(B, m, B{...})
  = if SuperOf(A, B, B{...}) getM(m, A{...})
    if SuperOf(B, B, ) getM(m, B{...})
```

The SuperOf is always evaluated with respect to its third argument, the list of classes. Also the list of classes is reduced in every step here, to keep track of what has been visited and what has not yet been visited. There is information lost on the way and the evaluation of SuperOf can get wrong w.r.t. the original set of classes. Above the first SuperOf is true, because in the class B in the third argument we will have the information that B extends A. The second SuperOf is trivially true.

Second, and wrong, is this execution.

```
getMethod(B, m, A{...} B{...})

  = if SuperOf(B, B, A{...}) getM(m, B{...})
    getMethod(B, m, A{...})

  = if SuperOf(B, B, A{...}) getM(m, B{...})
    if SuperOf(A, B, ) getM(m, A{...})
```

Above the first SuperOf expression is trivially true but the second one is false, because A is no superclass of B if we do not know anything about the classes which is the case here as we only have the empty set of classes remaining. That is wrong, because A is really a superclass of B. Thus we do not find the method in this case. This is due to the loss of information to keep track what we have done already.

An easy remedy for this is to keep a second set of the classes, which does not get changed and is used in creating the checks whether a class is a superclass of another one.

This problem is especially severe as these two execution paths show that the equational specification is non-confluent! □

On a more detailed level we will now show what changes we made to the semantics to get this to work correctly.

As suggested before we will need two sets of the class list and for that reason the old operator, given first, is replaced by a new one which takes an extra Classes argument and is given second below. We will use these sets of classes as mentioned before. The first one will be consumed during the search to make sure everything is checked exactly once as it happens in the old semantics. The second set will be given as argument to all the checks for super classes so all available information is there and it will not be changed at all.

```
op GetMethodList : CType MName Classes
                   -> MethodList .
```

gets replaced by

```
op GetMethodList : CType MName Classes Classes
                   -> MethodList .
```

The second equation replaces the first, which makes sure that the initial call to GetMethodList gets the Classes list twice.

```
var CT : CType . var mn : MName . var Cl : Classes .
eq GetMethods(CT, mn, Cl) =
      Compact(GetMethodList(CT, mn, Cl), Cl) .

eq GetMethods(CT, mn, Cl) =
      Compact(GetMethodList(CT, mn, Cl, Cl), Cl) .
```

We will restrict ourselves here to the base case and one of the four possible ways a class can be given, the others work the same way. Here we have those from the old semantics, where after applying the second equation the size of the set of classes in the SuperOf operator decreases with every use:

```
var md : Modifier . var Xc : Qid .
var sp : Supers . var cb : ClassBody .

eq GetMethodList(CT, mn, noClass) = none .

eq GetMethodList(CT, mn, ((md Class Xc sp cb) Cl)) =
     (if SuperOf(#c(Xc), CT, Cl)
      then GetMethodList(CT, mn, #c(Xc), cb)
      else none fi), GetMethodList(CT, mn, Cl) .
```

On the contrary in the changed semantics this does not happen. The set of classes used to create the checks about being a superclass is always the unchanged version in the last argument of the operator. Thus it is additional ballast to be carried through all the appearances of the operator but does its actual work only in the SuperOf operator where it is used instead of the smaller set of classes which is worked through. In the changed semantics the above two equations are replaced by these two, using the same variables as above.

```
eq GetMethodList(CT, mn, noClass , Cl') = none .

eq GetMethodList(CT, mn, ((md Class Xc sp cb) Cl), Cl') =
      (if SuperOf(#c(Xc), CT, Cl')
       then GetMethodList(CT, mn, #c(Xc), cb)
       else none fi), GetMethodList(CT, mn, Cl, Cl') .
```

4.1.4 Bad Internal Handling which Stops the Execution

This is a fairly technical point because it requires precise knowledge about other internal operators which have not been exposed here but it deserves to be mentioned nonetheless.

The first rule is the faulty one from the original specification and the second one is its replacement. First, all variables are declared as usual.

```
var L : Location . var E : Exp . var K : Continuation .
var V : Value . var cnt : Context . var M : Store .

--- original
rl c(k(L -> += (E) -> K), cnt), m([L,V] M)
      => c(k([E | V] -> + ->
          (set&fetch(L) -> K)), cnt), m([L,V] M) .

--- modified
rl c(k(L -> += (E) -> K), cnt), m([L,V] M)
      => c(k(V -> [E | noVal] -> + ->
          (set&fetch(L) -> K)), cnt), m([L,V] M) .
```

This replacement is necessary because the only equations given that can handle [_|_] have a value V as the first element in the continuation, i.e. V -> [El|Vl], with El being an expression list and Vl a value list. There is no equation which can handle a continuation beginning with [E | V] which the faulty rule creates and thus stops the computation. Simply putting the V -> in front and taking it out of the value list fixes the problem of stopping the execution and it sends the V to the right position, i.e. into the value list, during future equation applications.

Similar changes are necessary and have been done for the operators −=, ∗=, /= and %=.

4.1.5 Block Semantics

For a block of Java code there are a few things to note. First, everything which is visible when entering the block is visible inside and can be changed. All variables declared within the block are on the other hand only visible inside the block and not anymore after leaving the block. This is something the compiler checks at compile time so at runtime this should not be a problem, except in some special cases. It actually can get to be a problem with blocks within methods. Let us look at an example for this:

Example 4.1.2
Having any Java class with the attribute `int i;` declared and a method `m()` which does the following: `{ {int i; i = 0; } i = 1; }` the following happens. According to the Java language specification the last assignment of `i` to `1` should change the attribute `i` of the class. In the MJS because of the handling of blocks this assignment changes the `i` locally declared within the block, which is wrong. □

Another interesting point is also that the authors of the MJS have obviously thought about this as in the code we find the following:

```
var bs : BlockStatements . var K : Continuation .
var Env : Env .

---   eq k({bs} -> K), e(Env) = k(bs -> Env -> K), e(Env) .

      eq k({bs} -> K), e(Env) = k(bs -> K), e(Env) .
```

The first equation is commented out so this one is not really used but it is the one which is correct because it restores the local environment from before the block at the end of the block again. The second is not correct as we have seen in the example above. So we simply changed the comments from the first equation to the second equation, meaning that now the first equation is in the MJS and the second is not anymore.

4.1.6 Missing Default Constructor Workaround

There is no default constructor implemented so for every class which is given in the set of classes for a program run one has to manually define the default constructor and also call the constructor of the superclass first. This can be done in the following way:

```
Class 'B extends #c('A)
            { 'B () { #m('A) () ; ... } }
```

Where the `'B ()` is the default constructor of class 'B and the first thing it does is a call to the constructor of the superclass 'A.

4.2 Extensions of the Maude Java Semantics - with JLS Defined Constructs

There are several features which could be added to the Maude Java Semantics. Here we describe what we have added. The selection we took was mainly influenced by what we thought was necessary for the program transformations we intend to prove.

4.2.1 Type Check for Type Casts

We are only concerned with the correctness of run-time type casts on objects. Mistakes about primitive type casts are in Java caught by the compiler.

In the original MJS type casts were simply done, without a check whether they are allowed or not. This can be seen in the first equation. The second equation replaces the first and checks whether the new static type for the reference is a super-type of the dynamic type of the object. To do that it needs the set of classes `Cl`. Only in the case that the type check is positive the cast is allowed and the execution in the MJS will proceed as in the first equation. Otherwise an exception is generated. There is no exception handling so this is not very useful but at least it stops the computation and we can not get a false positive this way, i.e. no two pieces of code will be considered equivalent wrongly, as it could have happened without the type check.

```
vars CT CT' CT'' : CType . var oEnv : ObjEnv .
```

```
var K : Continuation . var Cl : Classes .
--- original
eq k(o(CT, CT', oEnv) -> {CT''} -> K) =
     k(o(CT'', CT', oEnv) -> K) .

--- modified
eq c(k(o(CT, CT', oEnv) -> {CT''} -> K), cnt), cl(Cl) =
        c(k(if SuperOf(CT'', CT', Cl)
          then o(CT'', CT', oEnv) -> K
          else
          throw ClassCastException(CT'', o(CT, CT', oEnv)) ;
          -> K fi),
          cnt), cl(Cl) .
```

Now one can still only cast class types and there are two primitive type casts. But those class type casts now check type correctness and raise exceptions in case of error. Only ints can be cast to floats and vice versa floats to ints for the primitive types in the original semantics. These type casts are done by using native Maude operators which do this so it might or might not do the cast exactly as Java would. This is not a problem for us as we only concern ourselves with comparing two configurations and then in both of those this would happen in the same way which means that in Java it would be done the same way both times, too. In addition we are not using any actual integers or floats with which this could be a problem but only typed skolem constants. Also in case of overflow in Java this would not happen in the MJS because it uses the mathematical integers which are internal to Maude, so that is not modeled truthfully.

We added the possibility to cast a primitive type onto itself with the following equations which actually have an implicit type check:

```
var I : Int . var f : Float . var str : String .
var B : Bool .

eq k(int(I) -> {int} -> K) = k(int(I) -> K) .
eq k(fl(f) -> {float} -> K) = k(fl(f) -> K) .
eq k(str(str) -> {String} -> K) = k(str(str) -> K) .
eq k(bool(B) -> {boolean} -> K) = k(bool(B) -> K) .
```

4.2.2 Type Check for Assignments

In the original semantics assignments have been executed without a type check and they also did not respect the static type of a reference.

Example 4.2.1
After creating an object reference with the declaration A a; with A a class and A being a superclass of B one would expect that the code a = new B(); creates an object of type B which the reference a points to. Also a should still have static type A and thus attributes of type B should not be accessible without a prior type cast. Actually in the implementation the assignment a = new B() removes any knowledge about a being of type A and it will be treated as being of type B. This has been fixed by the extensions and changes below. □

We added two new operators for this type checking. The first one introduces a run-time type check and the second one does the actual change after the type check was found to be ok. Also this change then respects the static type in contrast to the original. The first argument of typecheck is the value from the memory, which also contains its type, respectively its static type if it is a reference. We want to compare it with the type of the second value which is the one we want to assign to that location. The operator change−checked just works as change originally did, except for not writing directly over the memory location but using the originally provided, but not yet utilized, ReplaceWith construct, which replaces the dynamic type and the object environment, in the case of an object reference, but leaves the static type as it is.

```
op typecheck (_,_,_) -> _ :
      Value Value Location Continuation -> Continuation .
op change-checked (_,_) -> _ :
      Value Location Continuation -> Continuation .
```

There needs to be only one addition to the ReplaceWith construct, so it can work on generic values too if a certain value is not defined more precisely by the original equations, seen by the [owise] attribute.

```
eq ReplaceWith(V, V') = V [owise] .
```

The first rule below is the rule which governed memory changes in the original semantics and is the root of all problems. It does not check the types of V and V' and overwrites V' completely so any knowledge about the static type of V', if it was an object reference, is lost. It is replaced by the second rule, which even though we only want the static type of V needs to be a rule and no equation. That is because we want what is inside V to be fully equationally simplified before taking it out again, which is guaranteed when we use a rule.

```
vars V V' : Value . var L : Location .
var K : Continuation . var M : Store . var cnt : Context .
---- original
rl c(k(change(V, L) -> K), cnt), m([L, V'] M) =>
      c(k(K), cnt), m([L, V] M) .

---- modified
rl c(k(change(V, L) -> K), cnt), m([L, V'] M) =>
      c(k(typecheck(V', V, L) -> K), cnt), m([L, V'] M) .
```

Below follows the rule that does the actual memory write, like change did in the original version. The difference is that this is only invoked after the type check has been done, which is explained further down. Note that it leaves the way how one object reference is replaced by another, actually any value replaced by another, to the definition of ReplaceWith. The ReplaceWith construct keeps the static type of the reference as it should be and also works correctly for primitives as well as arrays, which also have a static type like object references.

```
rl c(k(change-checked(V, L) -> K), cnt), m([L, V'] M)
      => c(k(K), cnt), m([L, ReplaceWith(V', V)] M) .
```

For “objects” the type check works as defined by the equation below. The dynamic type of the “object” reference that is assigned to the location has to be a subtype of the static type of the “object” reference at that location. In the notation used below CT is the static type of the object at the location and CT''' is the dynamic type of the object that shall be written into the location. We are writing “object” because what is called object in this semantics is actually not an object but more like a “rich reference” to the object. SuperOf is another operator provided by the original semantics. It checks whether the first argument is a super class type (or the same class type) of the second argument w.r.t the given set of classes inside the third argument. If the SuperOf check yields true we actually do the change but if it is not true then a ClassCastException is thrown. Even though there is no exception handling we know when looking at a final configuration, which this creates, or also a complete trace, what happened. This could not be executed further anyway because we would not know what the exception handler would do and thus the results we get might differ from the result the Java Language Specification requires, which is not acceptable.

```
vars CT CT' CT'' CT''' : CType . var Cl : Classes .
eq c(k(typecheck(o(CT, CT', oEnv),
                o(CT'', CT''', oEnv'),
                L) -> K), cnt), cl(Cl) =
    c(k(if SuperOf(CT, CT''', Cl)
        then change-checked(o(CT'', CT''', oEnv'), L) -> K
        else throw
            ClassCastException(o(CT, CT', oEnv),
                              o(CT'', CT''', oEnv'),
                              L) ; -> K
        fi),
      cnt), cl(Cl) .
```

Arrays require a separate and special treatment for the type check too, as in this case we can have also Type[] as types, where Type is any type. This can also happen in multiple layers, i.e. Type [][][] . We can use the Assignable check, which takes care of the possibility of having [] in the type, possibly more than once, too. Inside it uses SuperOf as above and Assignable is also provided by the original semantics.

```
var T : Type . vars aEnv aEnv' : ArrayEnv .
eq c(k(typecheck(a(T, aEnv), a(T', aEnv'), L) -> K), cnt),
   cl(Cl) =
    c(k(if Assignable(T, T', Cl)
        then change-checked(a(T', aEnv'), L) -> K
        else throw ClassCastException(a(T, aEnv),
                                      a(T', aEnv'), L) ;
            -> K
        fi), cnt), cl(Cl) .
```

For all other cases, i.e. primitive types, we do not type check at run-time because it is the compiler's job to make sure that nothing bad happens here.

```
eq c(k(typecheck(V', V, L) -> K), cnt), cl(Cl) =
    c(k(change-checked(V, L) -> K ), cnt),
    cl(Cl) [owise] .
```

A problem related to this treatment, in particular to the handling of the owise case where we assume to work with primitive types, is that no type check is performed if not both sides, the value at the location and the value which is to be written there, are either objects or arrays. Thus one could write something like this: int a; a = new A() and it would be executed as if a would have been declared with type A. In these cases however it is the compiler's job to catch the problems before. Thus we need to make sure that everything is checked by the compiler first.

There is a minor mistake in the implementation of Assignable which is that it does not take into account the special treatment required by type Object. It can be assigned any higher-order array than it is itself. We therefore added one equation to the definition of Assignable before its owise equation:

```
var TI TI' :  Types . var CI : Classes .
eq Assignable((Object , TI), (T' [] , TI'), CI)
      = Assignable((Object , TI), (T' , TI'), CI) .
```

4.3 Extensions of the Maude Java Semantics - with Taclet Language Constructs

In this section we will take a look at special things we added to the MJS to be able to easily integrate functionality of the schematic Java in the taclet language.

4.3.1 Simultaneous Updates

Simultaneous Updates are another special construct used in KeY. They are basically assignments but have a few restrictions on them. These are that no expression appearing on any side of an update can have side effects. With the "simultaneous" one wants to motivate that all right hand sides of the updates are evaluated first, that is in the state before any of the assignments is really completed, i.e. written to memory. After all right hand sides have been evaluated they are assigned to their respective left hand sides in order of appearance. That means especially that if a left hand side appears twice as left hand side only the latest assignment to it will be effective. Here we show a way to enter simultaneous updates into an initial configuration and how it will be evaluated.

First we declare the sorts Update and UpdateList, with their subsort inclusion, as well as the operator which creates an update. The unit () symbol is the identity for sets of update lists.

```
sort Update .
op _:=_; : Exp Exp -> Update .
sort UpdateList .
subsort Update < UpdateList .
op `(`) : -> UpdateList .
op __ : UpdateList UpdateList
        -> UpdateList [assoc id: ()] .
```

The simUpd... operator takes the updates in a way the KeY user would expect it to work. The simUpdList... operator is an internal one into which the updates get put in the format listleft | listright with listleft the list of left hand sides and listright the list of right hand sides, where the i-th left hand side element belongs to the i-th right hand side element.

```
op simUpd_ -> _ : UpdateList Continuation -> Continuation .
op simUpdList_ | _ -> _ : Exps Exps Continuation
                                -> Continuation .
```

The following three equations create simUpdList(El | El') -> noLoc from a simUpd(e1 := e1' ; e2 := e2 ' ; ...) where the order of e1, e2 ,... in the El list and e1 ', e2 ',... in the El' list stays the same as in the original. An empty initial update list is not allowed because then there is no need to put an update construct there anyway!

```
var E E' : Exp . var El El' : Exps . var UD : UpdateList .
var LocaL : LocationList . var L : Location .

eq k(simUpd((E := E' ;) UD) -> simUpdList El | El' -> K) =
      k(simUpd(UD) -> simUpdList El, E | El', E' -> K) .
eq k(simUpd((E := E' ;) UD) -> K) =
      k(simUpd(UD) -> simUpdList E | E' -> noLoc -> K)
      [owise] .
eq k(simUpd(()) -> K ) = k(K) .
```

Now we need to evaluate the lefthand sides to their locations and keep those locations in reserve without assigning anything yet. In the first equation below, the first expression of the left hand side list is put to the front and is evaluated to a location by the existing rules. The evaluated location is then put behind the simUpdList... in a location list by the second equation below. Note that only the left set of expressions, the left hand sides, gets evaluated this way and is transformed into locations.

When all location expressions are removed from the operator and the operator is the first thing on the continuation with the () as its first argument, i.e. all locations are evaluated and in a list behind the operator, the third equation creates a continuation which evaluates all the expressions in the second expression list into a value list which is the first item on the continuation then. The second argument is the list of locations which cannot be seen here as that is subsumed in the K variable. The continuation, a value list followed by a location list, leads to the assignment of all the values to their corresponding locations. For details on how all this works internal to the full original semantics see [FCMR04].

```
eq k(simUpdList E, El | El' -> K) =
      k(loc(E) -> simUpdList El | El' -> K) .
eq k(L -> simUpdList El | El' -> LocaL -> K) =
      k(simUpdList El | El' -> LocaL, L -> K) .

eq k(simUpdList () | E, El -> K) =
      k(E -> [El | noVal] -> K) .
```

5 Configuration Generation

In this chapter we describe how we automatize the creation of the starting configurations for the MJS from a taclet. We will do the configuration generation in a two step process. The first part of that process extracts the required information from the taclets and it is in Java. It will format its output in a way such that the second part can create the real configuration, using Maude. We will first state a few things about what the Java implemented part does. Then we describe what we did in Maude to see what kind of interface specifications the Java implemented part has to meet and generally see how such a configuration can be generated quite easily. At the end we will see how the two parts work together to create all the necessary configurations for a taclet.

5.1 Configuration Generation from Taclets by Java

Starting with a code transformation taclet we create all possible combinations of the sub cases of its schema variables. For each of these complete sub cases a separate string along the lines of Section 5.2 is created. This string is the starting point to generate a configuration. It is then given to Maude where the actual configuration is created and executed. This happens for each case. All this works not only for a single taclet but for sets of taclets. It is also very fast as e.g. for 55 taclets the part which creates the string set, written in Java, takes about 5 seconds and the final generation together with the execution in Maude takes about 35 seconds for those 55 taclets.

This string creation including sub case building is done as a part of the KeY prover code, using a lot of the work already done there. It is especially important to name the parsing of the taclets which we did not need to re-do. We will not give any details on this part of the generation as it is straightforward Java programming.

The output of this is a file containing a string for each combination of the types of schema variables, where the string is as described in Section 5.3 and you can see Section 2.2.2 for the combinations of types of schema variables. Those strings are required for each taclet in the set of taclets which are to be validated. This file is then loaded by the modified extended MJS. For the interface points it offers for the generated text see Section 5.2.

5.2 Configuration Generation Maude Interface

This interface is actually added to the Maude Java Semantics and it is not implemented as a separate module. This facilitates the run of the actual program after the configuration has been created. Also a lot of the definitions of the MJS are already used during the configuration generation. With the Maude implementation given below we not only provide the interface for the Java written part but actually do a lot of internal work. The Java implemented part does not integrate everything into the configuration but only makes a list of what has to be entered. This Maude based implementation then takes care of this task and integrates everything into a real configuration. The Java implemented part of the generation creates a string which is read in by Maude as a usual rewriting command. So this string is in the format the Maude module requires and below we define what "helper functions" are available.

5.2.1 Special Constants

For an initial configuration we need a few skolem constants which can be interpreted as exactly that, which is there, for every "real" configuration. Those skolem constants will not be used as usual variables and constants but will remain unchanged, even though some of them can be overwritten and thus disappear. Now let us get on with the actual implementation.

This is a special sort with one operator which takes the place where later on the actual code is put. The second operator is just the standard way of getting any element of the sort, i.e. the first operator, into a continuation.

```
var CODE : BlockStatements . var AL : AddList .
sort InitialCode .
op initCodePlaceHolder : -> InitialCode .
op _ -> _ : InitialCode Continuation -> Continuation .
```

The first operator below marks where the initial code ends, which is created by the Java part later on, so it can be easily read from the string the Java implemented part has produced. The second operator is the constant representing the basic starting configuration upon which we expand with the help of the add operator later on.

```
op endOfInitCode : -> Continuation .

op basicInitConfiguration : -> Output .
```

To be able to define the basicInitConfiguration from above equationally we need quite a few “Maude Constants” which from our point of view are skolem constants. They all start with init for easier recognition in the resulting states.

These constants create “junk” [CDE+00] for some of the sorts but that is ok, we do want these constants to remain in our computation. Now we do not protect these sorts any more which were, in part, imported with the protecting attribute, but that is just a minor point of fixing declarations in the sub-modules.

The names of these constants are quite self-explanatory especially when one looks-up in the definition of the starting configuration where they are put so there is not much to explain here.

```
op initRemainderOfCode   : -> Continuation .
op initLocalEnv          : -> Env .
op initStaticClassType   : -> CType .
op initDynamicClassType  : -> CType .
op initObjEnv            : -> ObjEnv .
op initMemCounter        : -> Nat .
op initMemory            : -> Store .
op initSetOfClasses      : -> Classes .
op initStaticAttEnv      : -> ObjEnv .
op initSnapCounter       : -> Nat .
op initSnapList          : -> SnapshotList .
op initOutput            : -> Output .
```

This equation defines the basic initial configuration with the help of the above given constants. On this configuration we work with the add operator. Just a word on the noLock’s in there. We do not need locks as we only work with sequential programs. Therefore these do not get some “initialConstant” value but really noLock.

```
eq basicInitConfiguration
      = run(c(k(initCodePlaceHolder -> pause
                -> initRemainderOfCode)
               ,
              e(initLocalEnv),
               o( o(initStaticClassType,
                     initDynamicClassType,
                     initObjEnv
                    )
                 )),
            n(initMemCounter),
            m(initMemory),
            l(noLock),
            w(noLock),
            cl(initSetOfClasses),
            s(initStaticAttEnv),
            nextSnapshot(initSnapCounter),
            snapshots(initSnapList),
            out(initOutput)
            ) .
```

5.2.2 Basics for Adding Data and Adding the Code to the Configuration

We will arrange the addition of all data in small parts, i.e. one add... at a time. We will then have a list of such items which need to be added. These items will thus form a list, called AddList which is built as defined below. The single items will be AddListElements.

The last operator builds up the starting configuration (of sort output) required by the taclet by basically holding the configuration together. It takes a current configuration and adds all elements of the AddList to the configuration and in the end takes the code from the continuation and puts it into the configuration so the execution can proceed.

```
sort AddList .
sort AddListElement .
subsort AddListElement < AddList .
op emptyAddList : -> AddList .
op __ : AddList AddList
        -> AddList [assoc id: emptyAddList] .
```

```
op add : Output AddList Continuation -> Output .
```

This equation replaces the place holder code with the real initial code when no further items need to be added to the configuration and it allows the actual Java semantics to start. The initCodePlaceHolder stops all rules of the MJS from working before this happened. Also this equation removes the wrapping add operator at the same time.

```
eq add(run(c(k(initCodePlaceHolder -> pause -> K),
             cnt), state
          ),
        emptyAddList,
        CODE -> endOfInitCode)
      = run(c(k(CODE -> pause -> K), cnt), state) .
```

5.2.3 Adding Data to the Configuration

Now we look at the operators with their defining equations which add things into the configuration:

The first operator adds a Location-Value pair to the memory. It puts the given location and value into the memory as one Store.

The second operator adds a Name-Location pair to the local environment, i.e. a mapping of a variable name to a location. It puts the given name and location into the local environment as one Env.

The third operator adds a Name-Location pair to the environment of the current object. There it is seen as an attribute of the class type of the current object's static type.

```
op addToMemory : Location Value -> AddListElement .
ceq add(run(m(M), state),
        addToMemory(L, V) AL,
        K)
      = add(run(m(M [L,V]), state),
            AL, K)
            if not L :: TacletNewLocation .

op addToLocalEnv : Name Location -> AddListElement .
ceq add(run(c(e(Env), cnt), state),
         addToLocalEnv(X, L) AL,
         K)
       = add(run(c(e(Env [X, L]), cnt), state), AL, K)
       if (not X :: TacletNewVarName)
         and (not L :: TacletNewLocation) .

op addToCurrentObjEnv : Name Location -> AddListElement .
ceq add(run(c(o(o(CT, CT', oEnv)) ,cnt), state),
          addToCurrentObjEnv(X, L) AL,
          K)
        = add(run(c(o(o(CT, CT', oEnv (CT, [X, L]))),
                    cnt), state),
              AL, K)
        if (not X :: TacletNewVarName)
          and (not L :: TacletNewLocation) .
```

This puts the name and location together in an Env and together with the given CType they form an ObjEnv which is put into the environment of static attributes, where all static attributes of all class types are.

```
op addToStaticEnv : CType Name Location -> AddListElement .
ceq add(run(s(oEnv), state),
          addToStaticEnv(CT, X, L) AL,
          K)
        = add(run(s(oEnv (CT, [X, L])), state),
              AL, K)
        if (not X :: TacletNewVarName)
          and (not L :: TacletNewLocation) .
```

This adds a mapping of the given new variable to the given new location into the local environment and the mapping of the new location to the given value into the memory.

```
op addNewToLocalEnvAndMem :
       TacletNewVarName TacletNewLocation Value
       -> AddListElement .
eq add(run(c(e(Env), cnt), m(M), state),
         addNewToLocalEnvAndMem(TNVN, TNL, V) AL,
         K)
       = add(run(c(e(Env [TNVN, TNL]), cnt), m(M [TNL, V]),
                 state), AL, K) .
```

5.2.4 Adding Case Information

To keep track of the current case when there are a lot of taclets with a lot of sub cases for each taclet we introduce a new operator, caseInfo, which takes the actual code to be rewritten and wraps it together with two integers. The first is the taclet number and the second is the case number. With these one can look into the generated test file and find the taclet and case which one is interested in easily.

```
op caseInfo : Int Int Bool -> Bool .

eq caseInfo(N1:Int , N2:Int , B:Bool) = B:Bool .
```

This wrapper is of no relevance to the actual result which is of type Bool and that is allowed for by the one equation which is given. This result is the result of the comparison of two program runs on configurations which each have sort Output.

5.2.5 Examples of the Commands and their Results

Here we will take a look at a few examples of how to use these add commands and see what the resulting configurations look like.

Example 5.2.1
This first example will take a look at the two most basic commands addToMemory and addToLocalEnv. First you see the command to create a starting configuration with two variables, both of them defined in the local environment, and their respective values are put into the memory. No real code is used because we just want to see where each of the add part goes which we can best do if the execution stops because of not having real code.

```
rew add(basicInitConfiguration ,
        addToLocalEnv(X1:Name, L1:Location)
        addToMemory(L1:Location , V1:Value)
        addToLocalEnv(X2:Name, L2:Location)
        addToMemory(L2:Location , V2:Value)
        ,
        NoRealCodeToExecute:BlockStatements
          -> endOfInitCode) .
```

This results in the following configuration:

```
run(c(k(NoRealCodeToExecute:BlockStatements -> pause ->
        initRemainderOfCode),
      e(initLocalEnv [X1:Name,L1:Location]
        [X2:Name, L2:Location]),
      o(o(initStaticClassType , initDynamicClassType ,
          initObjEnv))),
    n(initMemCounter),
    m(initMemory [L1:Location ,V1:Value]
      [L2:Location , V2:Value]),
    l(noLock),w(noLock),cl(initSetOfClasses),
    s(initStaticAttEnv),out(initOutput),
    snapshots(initSnapList),nextSnapshot(initSnapCounter))
```

Continuing with this example we can use some real code. To keep it simple we just add the two variables which we now define to have values of type integer.

```
rew add(basicInitConfiguration ,
        addToLocalEnv(X1:Name, L1:Location)
        addToMemory(L1:Location , int(V1:Int))
        addToLocalEnv(X2:Name, L2:Location)
        addToMemory(L2:Location , int(V2:Int))
        ,
        (X1:Name = X1:Name + X2:Name ;) -> endOfInitCode) .
```

Next is the intermediate state of the execution where the configuration has been completely created but the execution within the actual MJS has not yet started.

```
run(c(k((X1:Name = X1:Name + X2:Name ;) -> pause ->
         initRemainderOfCode),
      e(initLocalEnv [X1:Name,L1:Location]
        [X2:Name,L2:Location]),
      o(o(initStaticClassType , initDynamicClassType ,
          initObjEnv))),
    n(initMemCounter),
    m(initMemory [L1:Location ,int(V1:Int)]
      [L2:Location ,int(V2:Int)]),
    l(noLock),w(noLock),cl(initSetOfClasses),
    s(initStaticAttEnv),out(initOutput),
    snapshots(initSnapList),nextSnapshot(initSnapCounter))
```

And this is the final result of the execution of the generated initial configuration.

```
run(c(k(pause -> initRemainderOfCode),
      e(initLocalEnv [X1:Name,L1:Location]
         [X2:Name,L2:Location]),
      o(o(initStaticClassType,initDynamicClassType,
          initObjEnv))),
    n(initMemCounter),
    m(initMemory [L1:Location,int(V1:Int + V2:Int)]
       [L2:Location,int(V2:Int)]),
    l(noLock),w(noLock),cl(initSetOfClasses),
    s(initStaticAttEnv),out(initOutput),
    snapshots(initSnapList),nextSnapshot(initSnapCounter))
```

□

After this simple example we can now see all of the available commands in action on one starting configuration in the next example.

Example 5.2.2

This is the example code which generates a starting configuration and uses all available commands to show precisely where each part is put.

```
rew add(basicInitConfiguration,
        addToLocalEnv(X1:Name, L1:Location)
        addToMemory(L1:Location, V1:Value)
        addToCurrentObjEnv(X2:Name, L2:Location)
        addToMemory(L2:Location, V2:Value)
        addToStaticEnv(ST:CType, X3:Name, L3:Location)
        addToMemory(L3:Location, V3:Value)
        addNewToLocalEnvAndMem
          (X4:TacletNewVarName, L4:TacletNewLocation,
                                V4:Value)
        ,
        NoRealCodeToExecute:BlockStatements
          -> endOfInitCode) .
```

The execution results in this initial configuration. In practice the NoRealCodeToExecute:BlockStatements would be replaced by some actual code and the execution could start right away.

```
run(c(k(NoRealCodeToExecute:BlockStatements -> pause ->
        initRemainderOfCode),
      e(initLocalEnv
         [X4:TacletNewVarName,L4:TacletNewLocation]
         [X1:Name,L1:Location]),
      o(o(initStaticClassType,initDynamicClassType,
            initObjEnv
            initStaticClassType,[X2:Name,L2:Location]))),
   n(initMemCounter),
   m(initMemory  [L4:TacletNewLocation,V4:Value]
      [L1:Location,V1:Value]  [L2:Location,V2:Value]
      [L3:Location,V3:Value]),
   l(noLock),w(noLock),cl(initSetOfClasses),
   s(initStaticAttEnv
      ST:CType,[X3:Name,L3:Location]),
   out(initOutput),
   snapshots(initSnapList),nextSnapshot(initSnapCounter))
```

□

5.3 Usage with Respect to Taclets

All of the above is used in the following way by the Java implemented part with respect to a taclet. First of all, the code sections are extracted from the find and replacewith parts. Then the variables are extracted from those and all possible combinations are created, but let us focus on just any one of those for now. For that fully decided (in the sense that for each schema variable we know the case) case we then reformat the code mentioned above for use with the MJS in the current case. The modified code part for the find part gets called code1 while the one for the replacewith part gets called code2. We get a list of things to add, call it add−a−lot, consisting of the AddListElements, from that. The mentioned combinations result from the possible different forms and types a schematic variable can take. From the varcond part of the taclet we find out which new variables are necessary, here we instantiate the new variables according to the original variables. Let us call that list of new variables we need to add add−new−vars and it is in the AddListElements format. Then for this one combination the rewrite which has to be true for the taclet to be possibly true is the following:

```
rew compareResultsModNewVars(
      add(basicInitConfiguration , add−a−lot , code1),
      add(basicInitConfiguration , add−a−lot add−new−vars ,
            code2)) .
```

To really validate the taclet this rewrite has to yield true for all possible combinations, i.e. for all cases.

6 Propositional Logic Taclets in Rewriting Logic

A different approach to validate taclets w.r.t. Maude is the one shown in this section. It turned out that only a very limited kind of taclet could be treated this way.

Part of the notation, and implementation, was inspired by the way the paper [CM00] treated linear logic, but the main point of that paper, using reflection in the maude language, was not brought to bear here. All of the modules mentioned in this chapter can be found in full in the Appendix C.

6.1 Basics

The idea here is to validate taclets for propositional logic by means of a semantics for propositional logic where both are implemented in rewriting logic, i.e. in Maude. For details on propositional logic taclets see Section 1.1.3.

First of all we have a module PROPO which includes truth values as well as atoms and the usual connectives of conjunction, disjunction, negation, implication and equivalence. This is the syntax of a propositional logic. This module gets extended by PROPO-SEQUENT to a propositional logic with sequents. Here we get the definition of sequents and how they can be built up but again this is only the syntax. The PROPO-SEQUENT module gets extended in both approaches mentioned above, i.e. when giving an imitation of the propositional taclets of KeY in rewriting logic that is built upon PROPO-SEQUENT's syntax as well as the semantics for propositional logic which is also based on that module.

6.2 Taclets Imitated

We have an imitation of taclets in the module PROPO-SEQUENT-TACLETS-IMITATION where KeY taclets have been translated to rewriting logic equations by hand but following a method with which they can be systematically created and thus could be automatically generated. Taclets which only have a *find*, *replacewith* and *if* part and only work on the top level, i.e. with a sequent (==> respectively |−) in each part, the translation works like this:

```
find (x ==> y) if (a ==> b) replacewith (r ==> t)
```

gives rise to the equation

```
eq  a , x , RestL |− b , y , RestR
              = a , r , RestL |− b , t , RestR .
```

Both rests, i.e. RestL and RestR, are variables of the corresponding type which is FormulaMultiSet. If one of the other variables is not included in the taclet, i.e. one or more of x, y, a, b, r, t are missing they can simply be left out in all their positions in the rewriting logic equation. In case that the whole *if* part is left out a and b can be dropped completely.

Taclets with find(b), where no sequent ==> exists, lead to problems. This is the case for example with the "replace_known_left" taclet. But taclets of that form are not part of the axiomatic propositional logic taclet set and will not be looked into here.

Now with PROPO-SEQUENT-TACLETS-IMITATION we have a calculus for propositional logic which returns closedgoal if it is given a universally valid formula in propositional logic. Example inputs for this can be found in the appendix C.1.

6.3 Propositional Logics Semantics

We also have a semantics for propositional logics given in PROPO-SEQUENTS-SEMANTICS which works on formulas but can take sequents as input and translates them to formulas according to the equivalence of the sequent

$$\phi_1, \phi_2, ..., \phi_n \vdash \psi_1, \psi_2, ..., \psi_m$$

and the formula

$$\wedge_{i=1}^{n}\phi_i \rightarrow \vee_{i=1}^{m}\psi_i$$

The evaluation of a propositional logic formula requires a mapping of each atom to a truth value and then uses the usual evaluation by the interpretation function which applies the mapping. The final truth value is computed according to the usual truth table which is also given in the semantics. There is a special operator TorF which rewrites to either true or false by a rewrite rule and therefore by mapping each atom to TorF all possible combinations of variable to truth value mappings can be achieved. To really check the result of all of them and not just an arbitrary one we use a search for final states with Maude's built-in search command. If the only possible result is true then the formula which is evaluated is universally valid. If false is a final state then the formula is not universally valid and one gets a variable mapping which makes the formula false by looking at the path which yields the final state false. Examples for this can be found in appendix C.2.

6.4 Validating Imitated Taclets with Help of the Semantics

As stated in the beginning the goal is to validate the imitated taclets in rewriting logic with help of the semantics. As we have the taclet imitation and the semantics we can look at how to do that. Given a Sequent A on which a taclet T can be applied call the result of the taclet application A'. By applying the taclet we mean using the equation which specifies the taclet. Now we can use the given semantics to find the value of the Sequent A for every atom to truth value assignment as well as for the sequent A'. For the taclet to be correct the semantics has to correspond on the final value of A and A' for all assignments. We do not use any example sequent on which T can be applied but a generic one with skolem constants so it could represent just any sequent on which T is applicable.

Therefore we can create a pair of formulas, or sequents which can be translated to formulas, (A, A') with one atom to truth value mapping M, namely the one where all atoms get mapped to TorF, for both formulas and then look at all created final pairs with the help of a search by which from the rewriting of all TorF constants all possible combinations of true and false are created. The resulting pairs can be (true, true), (true, false), (false, true) and (false, false). By using the equivalence $\leftrightarrow$ on the elements of the pair one gets true if both formulas agree and false otherwise. So it boils down to $eval(A, M) \leftrightarrow eval(A', M)$. If the only result is true that means that A and A' agree for every assignment and thus the taclet which transformed A to A' is validated. This is done by PROPO-SEQUENT-SEMANTICS-PROOFTACLETS.

So we face the question of how to give a generic sequent A upon which a taclet can be used to get A' and then check the equivalence of A and A' under all possible assignments M. We propose that it is adequate to assume all parts of the sequent are atoms and then by checking all assignments M all possible combinations of truth values for the generic sequent will be checked. One could worry about what happens if there are some elements, i.e. formulas, which depend on each other but that is just a special (sub-)case of checking all independent combinations. Therefore it suffices to consider the elements as atoms and check all atom mappings to proof the taclet correct.

As an example for this let us look at the "not_left" taclet which allows to put a negated formula on the left-hand side of the sequent to the right-hand side of the sequent but in positive form:

$$\neg A, Ar \vdash Br \ = \ Ar \vdash A, Br$$

Here Ar and Br are multisets of Propo, i.e. basically multisets of formulas. In the compilation of the sequent to a formula all of the elements of Ar (Br), which are Propos, are connected conjunctively (disjunctively). We can assume that that has happened before, so both are just one Propo and not a multiset of many Propos. To evaluate the equivalence of the sequent before the taclet application and the sequent after the taclet application it suffices to take Ar and Br as atoms with Ar, Br and A being independent, that is different, atoms. If they are dependent on each other then the atom assignment (still thinking of them as atoms) would be limited in what combinations are possible but nothing more. Thus showing the formula true for the independent case proves it for the special case of dependent formulas too. Thinking of the possibly interdependent formulas as atoms is acceptable as all the formulas are put together into one conjunctive (disjunctive) formula anyway and that can only be true or false in any case.

Basically all possible combinations of truth values for all elements (atoms!) get created and even if there are dependencies in the instance of the sequent which is looked at, all combinations which are allowed by the restriction because of the dependencies, will be checked anyway.

6.5 Embedding Taclets into Rewriting Logic

In contrast to the imitated taclets in the earlier part now we would like to embed taclets into rewriting logic in the sense that they are not defined by a rewriting logic equation or rule but that there is an actual operator which takes a taclet very close to the usual form it is given in and creates a configuration (of sort StateAttribute) from it. Then an application mechanism is needed. All that is defined in PROPO-SEQUENT-TACLETS-EMBEDDING. Right now the operator representing a taclet looks like this:

```
op taclet : LState -> StateAttribute .
```

By this, one defines a taclet as a state. In this state there can be its name, a sequent as the find part and multiple goaltemplates which include replacewith and add parts in their state because the replacewith and add parts always appear as pairs inside goaltemplates. There can be more than one of those pairs per taclet. There can also be another sequent representing the if part. This representation is easily extensible as one can simply add another ingredient to the state multiset.

A taclet contains schema variables which have to correspond to certain formulas from the sequent on which the taclet is applied. These schema variables can not be Maude variables but have to be constants of a sort SchemaVariable. These constants have to be mapped to the formulas of the sequent which is done by the function map. It maps an element of sort SchemaVariable to a formula (-multiset). One has to check if the set of mappings resulting at a taclet application is acceptable and this is done by the function ok−mapset and its helper functions. It checks that each SchemaVariable is only mapped to at most one formula. It could not be mapped to anything at all for this test but then the mapping would not be complete and the execution would simply stop. So every schema variable which is mapped to a formula is mapped to exactly one formula. Then we also need a function that gets the formula which a schema variable has been mapped to in a map set. Equipped with all these things it is possible to write down a set of (conditional) equations which can apply a given taclet to a given sequent if it is applicable. It is not capable of finding a fitting taclet out of a set of taclets to be applied to a sequent chosen out of a multiset of sequents.

At the moment this is also severely limited as only simple SchemaVariables can be part of the taclets and no terms over SchemaVariables are possible. Those terms are a necessary addition to be able to check all propositional logic taclets but adding them in, modifying the mapping to work on them and so forth is a lot of technical work for a very small gain, which is to have the taclets mechanically checked again with one step less done by hand.

6.6 Validating Embedded Taclets with Help of the Semantics

In a very similar way to the section about the validation of imitated taclets it is possible to validate the embedded taclets too. Only two taclets can actually be validated with the restriction to simple schema variables without connectives as we have it now.

The code for the tests of the taclets requires the module
PROPO-SEQUENT-TACLETS-EMBEDDING-SEMANTICS-PROOF.

This works by the above schema again, i.e. a pair of a sequent A and the application of the taclet tc which is to be checked on the sequent A get compared under the same atom assignment. So we get the pair (A , apply(tc , A)) and check that under all atom assignments M the two elements agree as seen above.

7 Conclusions

In this work we have shown ways to validate KeY code transformation taclets with the help of the Maude Java Semantics (MJS) and propositional logic taclets with the help of Maude.

Our main focus was on the code transformation taclets. There are about 140 code transformation taclets out of the 350 Java code related taclets, i.e. 40%. We already proved 55 of the code transformation taclets, see Section 2.1. A few more taclets should be provable with just minor technical enhancements. The difference between the amount of code transformation taclets we could prove and the total number of code transformation taclets comes from the fact that in these taclets *meta constructs* of the taclet language of KeY are used which are not easily transferable to the MJS. Another, even bigger, factor are the restrictions which come from the limitations of the MJS.

The majority of our work was done to lift a semantics for *concrete* Java code to a semantics for *schematic* Java code. This was necessary because we compare the two code segments of code transformation taclets which are not composed of ordinary Java code but of schematic Java code. This posed a number of questions and difficulties which were solved during our work.

Our approach for code transformation taclets is extensible to cover all code transformation taclets without meta constructs when all elements of Java are covered correctly by the MJS. With enough effort some of the meta constructs could possibly be translated, too. The number of taclets which we cannot handle without being able to work with all meta constructs is roughly 20. Of the remaining 120 taclets we already proved 55 and could prove another 20 when division and modulo computations on variables, i.e. non-ground terms, works. Out of the other 45 taclets about 25 require the MJS to be completed to cover the respectively used Java language constructs and the remaining 20 are problematic because of boolean decisions, like `if`.

The paper [MR04] introduces a rewriting logic semantics for a CaML-like language. It was important to us for understanding the principles of rewriting logic semantics, using continuations, for programming languages. It was thus a very good starting point to understand the MJS, for which no documentation exists as far as we know. In this paper the MJS was also mentioned as a semantics for Java but no comments on its quality and completeness were made. We then found out that there are quite some problems with the MJS, i.e. errors, and that the MJS does not cover all of the Java language.

The Maude Java Semantics availability was of paramount importance for this work. Even though we were restricted by its errors and its incompleteness it was of tremendous help. To use it properly a lot of work on our part was necessary because the mistakes, and their remedies, were far from obvious in most cases. Some of the errors we found could be corrected while others were so deeply rooted that in the limited time frame of this work we could not mend them. We also added some features to the MJS which we deemed necessary for our work.

In the other approach, the one looking at propositional logic taclets, we were able to prove all axiomatic propositional logic taclets given in the KeY system.

As future work we thus propose some technical enhancements of our method and the completion of the MJS. With that done one could go ahead and investigate the problems we have with decisions on boolean variables. Whenever an `if`, or a similar construct which needs to know whether the boolean value, which is its argument, is true or false and a generic constant which represents its value is not enough, appears, it seems like it is necessary to split the execution in two parts and then look at the two reduced problems.

Proving taclets which include meta constructs is an interesting open question, too. It might be possible to exploit the fact that the meta constructs are implemented as Java code but it was not done yet. A necessary requirement is the completeness of the MJS though.

We see this work as a contribution to the field of (semi)-automated provers and their correctness. We specifically did a step towards validating the KeY system with this approach by validating part of the rules underlying the KeY system's prover .

A Code for Examples

A.1 Examples for Chapter 2.2

The three code sections below are related to Ex. 2.2.1.

- If #lhs1 is a local variable, the whole (executable) configuration is:

```
rew compareResult(
run(
  c(k((++ lhs1Name:Name ; )
       -> pause -> RoP:Continuation),
       e([lhs1Name:Name, lhs1Loc:Location]
         RestOfLocalEnvironment:Env),
       o(o(#c(ct:Qid), #c(ct':Qid), objEnv:ObjEnv))),
  m([lhs1Loc:Location , int(lhs1Val:Int)]
    RestOfMemory:Store),
  n(mC:Nat), cl(setOfClasses:Classes),
  s(RestOfStaticAttributeEnvironments:ObjEnv),
  out(anyOutput:Output), l(noLock), w(noLock))
,
run(
  c(k((lhs1Name:Name = (lhs1Name:Name + #i(1)) ;)
       -> pause -> RoP:Continuation),
       e([lhs1Name:Name, lhs1Loc:Location]
         RestOfLocalEnvironment:Env),
       o(o(#c(ct:Qid), #c(ct':Qid), objEnv:ObjEnv))),
  m([lhs1Loc:Location , int(lhs1Val:Int)]
    RestOfMemory:Store),
  n(mC:Nat), cl(setOfClasses:Classes),
  s(RestOfStaticAttributeEnvironments:ObjEnv),
  out(anyOutput:Output), l(noLock), w(noLock))
) .
```

- If #lhs1 is an attribute of the current object without an explicit this, the whole (executable) configuration is:

```
rew compareResult(
run(
  c(k((++ lhs1Name:Name ; )
       -> pause -> RoP:Continuation),
      e(RestOfLocalEnvironment:Env),
      o(o(#c(ct:Qid), #c(ct':Qid), objEnv:ObjEnv
        (#c(ct:Qid),
           [lhs1Name:Name, lhs1Loc:Location]) ))),
  m([lhs1Loc:Location , int(lhs1Val:Int)]
    RestOfMemory:Store),
  n(mC:Nat), cl(setOfClasses:Classes),
  s(RestOfStaticAttributeEnvironments:ObjEnv),
  out(anyOutput:Output), l(noLock), w(noLock))
,
run(
  c(k((lhs1Name:Name =
          (lhs1Name:Name + #i(1)) ;)
       -> pause -> RoP:Continuation),
      e(RestOfLocalEnvironment:Env),
      o(o(#c(ct:Qid), #c(ct':Qid), objEnv:ObjEnv
        (#c(ct:Qid),
           [lhs1Name:Name, lhs1Loc:Location]) ))),
  m([lhs1Loc:Location , int(lhs1Val:Int)]
    RestOfMemory:Store),
  n(mC:Nat), cl(setOfClasses:Classes),
  s(RestOfStaticAttributeEnvironments:ObjEnv),
  out(anyOutput:Output), l(noLock), w(noLock))
) .
```

- If #lhs1 is an attribute of the current object with an explicit this, then the only difference to the case above without an explicit this is that in front of all occurrences of lhs1Name:Name there will be a this . . The whole (executable) configuration is then:

```
rew compareResult(
run(
  c(k((++ this . lhs1Name:Name ; )
       -> pause -> RoP:Continuation),
      e(RestOfLocalEnvironment:Env),
      o(o(#c(ct:Qid), #c(ct':Qid), objEnv:ObjEnv
        (#c(ct:Qid),
                [lhs1Name:Name, lhs1Loc:Location]) ))),
    m([lhs1Loc:Location, int(lhs1Val:Int)]
      RestOfMemory:Store),
    n(mC:Nat), cl(setOfClasses:Classes),
    s(RestOfStaticAttributeEnvironments:ObjEnv),
    out(anyOutput:Output), l(noLock), w(noLock))
  ,
  run(
    c(k((this . lhs1Name:Name =
              {int} (this . lhs1Name:Name + #i(1)) ;)
          -> pause -> RoP:Continuation),
        e(RestOfLocalEnvironment:Env),
        o(o(#c(ct:Qid), #c(ct':Qid), objEnv:ObjEnv
          (#c(ct:Qid),
            [lhs1Name:Name, lhs1Loc:Location]) ))),
    m([lhs1Loc:Location, int(lhs1Val:Int)]
      RestOfMemory:Store),
    n(mC:Nat), cl(setOfClasses:Classes),
    s(RestOfStaticAttributeEnvironments:ObjEnv),
    out(anyOutput:Output), l(noLock), w(noLock))
  ) .
```

Now to the two code sections related to Ex. 2.2.2:

- If #lhs1 is a static attribute with a type reference followed by a sequence of static attribute accesses, the whole (executable) configuration is:

```
rew compareResult(
run(
  c(k((++ lhs1CT:CType . lhs1Name:Name ; )
        -> pause -> RoP:Continuation),
      e(RestOfLocalEnvironment:Env),
      o(o(#c(ct:Qid), #c(ct':Qid), objEnv:ObjEnv))),
  m([lhs1Loc:Location, int(lhs1Val:Int)]
    RestOfMemory:Store),
  n(mC:Nat), cl(setOfClasses:Classes),
  s((lhs1CT:CType, [lhs1Name:Name, lhs1Loc:Location])
    RestOfStaticAttributeEnvironments:ObjEnv),
  out(anyOutput:Output), l(noLock), w(noLock))
,
run(
  c(k((lhs1CT:CType . lhs1Name:Name =
        {int} (lhs1CT:CType . lhs1Name:Name + #i(1)) ;)
        -> pause -> RoP:Continuation),
      e(RestOfLocalEnvironment:Env),
      o(o(#c(ct:Qid), #c(ct':Qid), objEnv:ObjEnv))),
  m([lhs1Loc:Location, int(lhs1Val:Int)]
    RestOfMemory:Store),
  n(mC:Nat), cl(setOfClasses:Classes),
  s((lhs1CT:CType, [lhs1Name:Name, lhs1Loc:Location])
    RestOfStaticAttributeEnvironments:ObjEnv),
  out(anyOutput:Output), l(noLock), w(noLock))
) .
```

- If #lhs1 is a static attribute, with a program variable as the first element, followed by a sequence of attribute accesses. The difference to the case above is that there is an extra object reference in the local environment and that reference is added to the memory too. Also in the code that object reference's name replaces the type reference from above. The whole (executable) configuration is then:

```
rew compareResult(
run(
  c(k((++ lhs1ObjRef:Name . lhs1Name:Name ; )
        -> pause -> RoP:Continuation),
      e([lhs1ObjRef:Name, lhs1ObjRefLoc:Location]
        RestOfLocalEnvironment:Env),
      o(o(#c(ct:Qid), #c(ct':Qid), objEnv:ObjEnv))),
  m([lhs1Loc:Location, int(lhs1Val:Int)]
    [lhs1ObjRefLoc:Location,
      o(lhs1CT:CType, lhs1DT:CType,lhs1ObjEnv:ObjEnv)]
    RestOfMemory:Store),
  n(mC:Nat), cl(setOfClasses:Classes),
  s((lhs1CT:CType, [lhs1Name:Name, lhs1Loc:Location])
    RestOfStaticAttributeEnvironments:ObjEnv),
  out(anyOutput:Output), l(noLock), w(noLock))
,
run(
  c(k((lhs1ObjRef:Name . lhs1Name:Name = {int}
        (lhs1ObjRef:Name . lhs1Name:Name + #i(1)) ;)
        -> pause -> RoP:Continuation),
      e([lhs1ObjRef:Name, lhs1ObjRefLoc:Location]
        RestOfLocalEnvironment:Env),
      o(o(#c(ct:Qid), #c(ct':Qid), objEnv:ObjEnv))),
  m([lhs1Loc:Location, int(lhs1Val:Int)]
        [lhs1ObjRefLoc:Location,
          o(lhs1CT:CType, lhs1DT:CType,lhs1ObjEnv:ObjEnv)]
        RestOfMemory:Store),
      n(mC:Nat), cl(setOfClasses:Classes),
      s((lhs1CT:CType, [lhs1Name:Name, lhs1Loc:Location])
        RestOfStaticAttributeEnvironments:ObjEnv),
      out(anyOutput:Output), l(noLock), w(noLock))
    ) .
```

Following are the two code sections related to Ex. 2.2.3:

- First, we look at the case of #se being an integer literal, #nse returning an integer and #lhs being an integer local variable. The whole (executable) configuration is then:

```
rew compareResultsModNewVars(
run(
  c(k((lhsName:Name =
          eval(nseEN:ExpressionName, int-result)
          * #i(seInt:Int) ;)
        -> pause -> RoP:Continuation),
      e([lhsName:Name, lhsLoc:Location]
        RestOfLocalEnvironment:Env),
      o(o(#c(ct:Qid), #c(ct':Qid), objEnv:ObjEnv))),
  m([lhsLoc:Location, int(lhsVal:Int)]
    RestOfMemory:Store),
  n(mC:Nat), cl(setOfClasses:Classes),
  s(RestOfStaticAttributeEnvironments:ObjEnv),
  out(anyOutput:Output), l(noLock), w(noLock),
  snapshots(SnapList:SnapshotList),
  nextSnapshot(numOfSnaps:Nat))
,
run(
  c(k((vNewVar:TacletNewVarName =
          eval(nseEN:ExpressionName, int-result) ;
        lhsName:Name =
          vNewVar:TacletNewVarName
          * #i(seInt:Int) ;)
        -> pause -> RoP:Continuation),
      e([lhsName:Name, lhsLoc:Location]
        [vNewVar:TacletNewVarName,
           vNewLoc:TacletNewLocation]
        RestOfLocalEnvironment:Env),
        o(o(#c(ct:Qid), #c(ct':Qid), objEnv:ObjEnv))),
    m([lhsLoc:Location , int(lhsVal:Int)]
      [vNewLoc:TacletNewLocation , int(vNewInt:Int)]
      RestOfMemory:Store),
    n(mC:Nat), cl(setOfClasses:Classes),
    s(RestOfStaticAttributeEnvironments:ObjEnv),
    out(anyOutput:Output), l(noLock), w(noLock),
    snapshots(SnapList:SnapshotList),
    nextSnapshot(numOfSnaps:Nat))
  ) .
```

- Second, we look at the case of #se being an integer local variable, #nse returning an integer and #lhs being an integer local variable, the whole (executable) configuration is:

```
rew compareResultsModNewVars(
run(
  c(k((lhsName:Name =
          eval(nseEN:ExpressionName , int-result)
          * seName:Name ;)
       -> pause -> RoP:Continuation),
      e([lhsName:Name, lhsLoc:Location]
        [seName:Name, seLoc:Location]
        RestOfLocalEnvironment:Env),
      o(o(#c(ct:Qid), #c(ct':Qid), objEnv:ObjEnv))),
  m([lhsLoc:Location , int(lhsVal:Int)]
    [seLoc:Location , int(seVal:Int)]
    RestOfMemory:Store),
  n(mC:Nat), cl(setOfClasses:Classes),
  s(RestOfStaticAttributeEnvironments:ObjEnv),
  out(anyOutput:Output), l(noLock), w(noLock),
  snapshots(SnapList:SnapshotList),
  nextSnapshot(numOfSnaps:Nat))
,
run(
  c(k((vNewVar:TacletNewVarName =
          eval(nseEN:ExpressionName , int-result) ;
       lhsName:Name =
          vNewVar:TacletNewVarName
          * seName:Name ;)
       -> pause -> RoP:Continuation),
      e([lhsName:Name, lhsLoc:Location]
        [seName:Name, seLoc:Location]
```

```
          [vNewVar:TacletNewVarName,
            vNewLoc:TacletNewLocation]
          RestOfLocalEnvironment:Env),
        o(o(#c(ct:Qid), #c(ct':Qid), objEnv:ObjEnv))),
    m([lhsLoc:Location, int(lhsVal:Int)]
      [seLoc:Location, int(seVal:Int)]
      [vNewLoc:TacletNewLocation, int(vNewInt:Int)]
      RestOfMemory:Store),
    n(mC:Nat), cl(setOfClasses:Classes),
    s(RestOfStaticAttributeEnvironments:ObjEnv),
    out(anyOutput:Output), l(noLock), w(noLock),
    snapshots(SnapList:SnapshotList),
    nextSnapshot(numOfSnaps:Nat))
  ) .
```

B Automated Use to Prove Sets of Taclets

B.1 Technical Details

The execution of the class “CheckPrgTransfSoundness” [see the KeY implementation] with a “.key” file which includes the taclets which are to be validated creates an intermediate text file. This intermediate text file has to be executed by Maude, after loading the extended MJS, with the output written to a result text file. Now open the resulting file with a text editor of your choice. To find out whether all taclets are correct you now have to search for certain result strings. First you can search for “result Bool: false”, if you find that the corresponding taclet is not correct. If you do not find that string you need to search for “result [Bool]” and if you find an instance of that you have a case where there will be the resulting configuration from which the MJS cannot continue any more and so there probably is some problem with what this work supports and what your taclet needs. The “[Bool]” represents the kind of expressions of sort “Bool” which means that they are those which could not be reduced to the ground terms, true or false. If you can find none of the two things above then it is safe to assume that all results will be “result Bool: true” and thus all your taclets are correct.

C Propositional Logic Taclets Maude Code

We now present the Maude modules which were used for the implementation of the validation of propositional logic taclets. Not all comments in here are very helpful, some of them were meant as reminders during the implementation.

```
fmod ATOM is
  sort Atom .
  ops a b c d e : -> Atom . --- number of atoms is arbitrary!
endfm

mod PROPO is
  including ATOM .
  including QID .

  sort Propo .
  subsort Qid < Atom < Propo .

  sort TruthValue .
  subsort TruthValue < Propo .
  ops t f : -> TruthValue .

  op !_ : Propo -> Propo [prec 30] .
  op _&_ : Propo Propo -> Propo [assoc comm  prec 32] .
  op _|_ : Propo Propo -> Propo [assoc comm  prec 33] .
  op _->_ : Propo Propo -> Propo [prec 35] .
  op _<->_ : Propo Propo -> Propo [assoc comm prec 37] .
endm

mod PROPO-SEQUENT is
  including PROPO .
  sort FormulaMultiSet .
  subsort Propo < FormulaMultiSet .
  op nil : -> FormulaMultiSet .
  op _,_ : FormulaMultiSet FormulaMultiSet
          -> FormulaMultiSet [assoc comm id: nil prec 40] .
  --- in KeY this is a duplicate-free list, list structure
  --- is not necessary here as there is no user-interaction
  --- sadly it is not really a "set" as duplicate entries
  --- are possible, we would like to get rid of them with
  --- idempotency as below but that is a problem:
  --- ceq Ar , Ar = Ar if (Ar =/= nil) . <-- this is not
  --- possible because matching modulo assoc and idempotency
  ---  is not currently available

  sort Sequent .
  op closedgoal : -> Sequent .
  op _|-_ : FormulaMultiSet FormulaMultiSet -> Sequent [prec 45] .

  sort OpenGoals .
  subsort Sequent < OpenGoals .
  op __ : OpenGoals OpenGoals
          -> OpenGoals [assoc comm id: closedgoal prec 50] .
  --- eq closedgoal closedgoal = closedgoal .
endm

mod PROPO-SEQUENT-SEMANTICS is
  including PROPO-SEQUENT .

  sort VariableToTruthVal .
  op _mappedto_ : Atom TruthValue -> VariableToTruthVal .

  sort VariableMap .
  subsort VariableToTruthVal  < VariableMap .
  op emptyvarmap : -> VariableMap .
  op __ : VariableMap VariableMap
          -> VariableMap [assoc comm id: emptyvarmap] .

  op evalFormula : Propo VariableMap -> TruthValue .

  --- evaluation of formulas over true and false, i.e. on truth values
  --- the equations below can also be seen as the truth table
  eq ! t = f .
  eq ! f = t .

  eq t & t = t .
  eq t & f = f .
  eq f & t = f .
  eq f & f = f .

  eq t | t = t .
  eq t | f = t .
  eq f | t = t .
  eq f | f = f .

  eq t -> t = t .
  eq t -> f = f .
  eq f -> t = t .
  eq f -> f = t .

  eq t <-> t = t .
  eq t <-> f = f .
  eq f <-> t = f .
  eq f <-> f = t .

  --- this is the interpretation function which, when given a formula
  --- and a atom-to-truthvalue mapping takes the formula apart and
  --- replaces the atoms by their truthvalues, afterwards the truth
  --- table above takes over
  vars A B C : Propo .
  vars M N : VariableMap .
  var X : Atom .
  var V : TruthValue .

  eq evalFormula(! A , M) = ! evalFormula(A , M) .
  eq evalFormula(A & B , M) = evalFormula(A , M) & evalFormula(B , M) .
  eq evalFormula(A | B , M) = evalFormula(A , M) | evalFormula(B , M) .
  eq evalFormula(A -> B , M) = evalFormula(A , M) -> evalFormula(B , M) .
  eq evalFormula(A <-> B , M) = evalFormula(A , M) <-> evalFormula(B , M) .

  eq evalFormula(X , M (X mappedto V)) = V .
  eq evalFormula(V , M) = V .


  --- this makes sure each atom is mapped to one and the same truth
  --- value everywhere! here you can give atoms mapped to TorF, that
  --- is this atom can be either true or false. we cannot allow the
  --- rewrites "AssignTrue" and "AssignFalse" to happen later as then
  --- an atom "a" might in one place be set to true and false in
  --- another. this happens in this extra wrapper functions.
  --- For this there is a third truth value added to the constants.
  --- This is what all atoms are mapped to initially.
  op evForm : Propo VariableMap -> TruthValue .
  op fixAllAtomAssignments : Propo VariableMap VariableMap -> TruthValue .
  op TorF : -> TruthValue .

  eq evForm(A , M)
    = fixAllAtomAssignments(A , M , emptyvarmap) .

  eq fixAllAtomAssignments(A , M (X mappedto t) , N)
    = fixAllAtomAssignments(A , M , N (X mappedto t)) .
  eq fixAllAtomAssignments(A , M (X mappedto f) , N)
    = fixAllAtomAssignments(A , M , N (X mappedto f)) .
  rl [AssignTrue] : TorF => t .
  rl [AssignFalse] : TorF => f .
  eq fixAllAtomAssignments(A , emptyvarmap , N) = evalFormula(A , N) .


  --- translation of Sequents to formulas.
  op FormulaFromSequent : Sequent -> Propo .

  vars F G H I : FormulaMultiSet .

  ceq FormulaFromSequent(F , G , H |- I)
      = FormulaFromSequent( F & G , H |- I)
      if (F =/= nil) and (G =/= nil) .
  ceq FormulaFromSequent(I |- F , G , H)
      = FormulaFromSequent( I |- F | G , H)
      if (F =/= nil) and (G =/= nil) .
  eq FormulaFromSequent( nil |- A) = A .
  eq FormulaFromSequent( A |- nil) = ! A .
  eq FormulaFromSequent( A |- B) = A -> B .
  eq FormulaFromSequent(closedgoal) = t .
endm

mod PROPO-SEQUENT-SEMANTICS-PROOFTACLETS is
  including PROPO-SEQUENT-SEMANTICS .

  ---sort TruthValuePair .
  ---op _;_ : TruthValue TruthValue -> TruthValuePair .

  op evFormPair : Propo Propo VariableMap -> TruthValue .
  op fixAllAtomAssignmentsForPairs : Propo Propo VariableMap VariableMap
                                     -> TruthValue .

  vars A B C : Propo .
  vars M N : VariableMap .
  var X : Atom .

  eq evFormPair(A , B , M)
     = fixAllAtomAssignmentsForPairs(A , B , M , emptyvarmap) .

  eq fixAllAtomAssignmentsForPairs(A , B , M (X mappedto t) , N)
     = fixAllAtomAssignmentsForPairs(A , B , M , N (X mappedto t)) .
  eq fixAllAtomAssignmentsForPairs(A , B , M (X mappedto f) , N)
     = fixAllAtomAssignmentsForPairs(A , B , M , N (X mappedto f)) .

  eq fixAllAtomAssignmentsForPairs(A , B , emptyvarmap , N)
     = evalFormula(A , N) <-> evalFormula(B , N) .

endm

mod PROPO-SEQUENT-TACLETS-IMITATION is
  including PROPO-SEQUENT .

  vars A B C : Propo .
  vars Ar Br : FormulaMultiSet .

  ---- taclets below, line numbers refer to key/proof/rules/propRule.key

  --- be careful about the rules and reading top/down...

  --- closegoal line: 25
  eq     A , Ar |- A , Br
     =-----------------
          closedgoal .

  ---close_by_false   { find (false ==>) close goal heuristics(closure) };
  eq     f , Ar |- Br
     =-------------
         closedgoal .

  ---close_by_true    { find (==> true) close goal heuristics(closure) };
  eq     Ar |- t , Br
     =-------------
         closedgoal .

  --- true_left line: 39
  eq     t , Ar |- Br
     =-------------
         Ar |- Br .

  --- false_right line: 40
  eq     Ar |- f , Br
     =-------------
          Ar |- Br .

  --- not_left line: 39
  eq     ! A , Ar |- Br
     =---------------
        Ar |- A , Br .

  --- not_right line: 40
  eq     Ar |- ! B , Br
     =---------------
        B , Ar |- Br .
  --- imp_left line: 45
  eq          A -> B , Ar |- Br
     =----------------------------------
       (B , Ar |- Br)   (Ar |- A , Br) .

  --- imp_right line: 49
  eq    Ar |- A -> B , Br
     =------------------
       A , Ar |- B , Br .

  --- and_left line: 52
  eq    A & B , Ar |- Br
     =-----------------
       A , B , Ar |- Br .

  --- and_right line: 53
  eq          Ar |- A & B , Br
     =---------------------------------
       (Ar |- A , Br)  (Ar |- B , Br) .

  --- or_left line: 55
  eq          A | B , Ar |- Br
     =---------------------------------
       (A , Ar |- Br)  (B , Ar |- Br) .

  --- or_right line: 57
  eq    Ar |- A | B , Br
     =-----------------
       Ar |- A , B , Br .

  --- equiv_left line: 63
  eq            A <-> B , Ar |- Br
     =-----------------------------------------
       (A , B , Ar |- Br) ( Ar |- A , B , Br) .

  --- equiv_right line: 72
  eq            Ar |- A <-> B , Br
     =----------------------------------------
       (A , Ar |- B , Br) (B , Ar |- A , Br) .
endm

mod PROPO-SEQUENT-TACLETS-EMBEDDING is
  including PROPO-SEQUENT .

  sort TCName .
  subsort Qid < TCName .

  sort StateAttribute .
  sort LState .
  subsort StateAttribute < LState .
  op empty : -> LState .
  op _||_ : LState LState -> LState [assoc comm id: empty] .

  op taclet : LState -> StateAttribute .

  op tacname : TCName -> StateAttribute .
  op find : Sequent -> StateAttribute .

  op goaltemplate : LState -> StateAttribute .
op rplcw : Sequent -> StateAttribute .
op add : Sequent -> StateAttribute .

op if : Sequent -> StateAttribute .

--- This application is meant to apply one FITTING taclet to a single
--- given sequent. This is only the application, it does not decide
--- whether a taclet is applicable though it will not allow unfitting
--- taclets to have any effect, but that ends in a non-rewritable term
--- as then a "apply-tc(Taclet , Sequent)" term will stay as is.

op apply-tc : LState Sequent -> OpenGoals .

var N : TCName .
vars X Y A B Rr Ra Tr Ta RestL RestR : FormulaMultiSet .
vars S E : Sequent .
var RestState RestState2 RestState3 : LState .

--- SchemaVariableTERM needs to be added and the mapping needs to
--- be accordingly generalized!
--- otherwise only a very limited amount of taclets can be checked
--- and used
sort SchemaVariable .
subsort SchemaVariable < TruthValue . --- < Propo

--- variables of sort SchemaVariable
vars SVX SVY SVRr SVTr SVRa SVTa SVA SVB : SchemaVariable .

--- constant of sort SchemaVariable, characterizing the "empty",
--- or non-existing, SV.
op mtsv : -> SchemaVariable .


sort Map .
sort MapSet .
subsort Map < MapSet .

op empty-map : -> Map .
op map : SchemaVariable FormulaMultiSet -> Map .

op __ : MapSet MapSet -> MapSet [assoc comm id: empty-map prec 60] .

op ok-mapset : MapSet -> Bool .
op ok-mapset-help : Map MapSet -> Bool .
op ok-map-pair : Map Map -> Bool .

vars SV1 SV2 : SchemaVariable .
vars Mp Mp2 : Map .
var MSet : MapSet .

eq ok-mapset( Mp ) = true .
eq ok-mapset( Mp MSet ) = ok-mapset-help( Mp , MSet )
                          and ok-mapset(MSet) .

eq ok-mapset-help( Mp , empty-map) = true .
eq ok-mapset-help( Mp , Mp2 MSet)
   = ok-map-pair(Mp , Mp2) and ok-mapset-help(Mp , MSet) .

eq ok-map-pair( map(SV1 , A) , map(SV1 , A) ) = true .
ceq ok-map-pair( map(SV1 , A) , map(SV1 , B) ) = false if A =/= B .
ceq ok-map-pair( map(SV1 , A) , map(SV2 , B) ) = true if SV1 =/= SV2 .

--- This is giving the formula that the given schemavariable was mapped to
--- in the given mapset! If there is nothing SV1 was mapped to this will
--- return map-1(SV1 , empty-map) and can't be rewritten any more (see
--- the 2 equations below).
op map-1 : SchemaVariable MapSet -> FormulaMultiSet .

eq map-1(SV1 , map(SV1 , A) MSet) = A .
ceq map-1(SV1 , map(SV2 , A) MSet) = map-1(SV1 , MSet) if SV1 =/= SV2 .

--- taclet application general case ; the second part of the condition
--- is there because if A and B are equal to nil there is no find part
--- basically!
ceq apply-tc( taclet( find(SVX |- SVY) || goaltemplate(rplcw(SVRr |- SVTr)
                      || add(SVRa |- SVTa) ) || if(SVA |- SVB)
                      || RestState )
           , (X , A , RestL |- Y , B , RestR) )
      = (map-1(SVRr ,  map(SVX , X) map(SVY , Y) map(SVA , A)
                      map (SVB , B) map(mtsv , nil)) ,
         map-1(SVRa ,  map(SVX , X) map(SVY , Y) map(SVA , A)
                      map (SVB , B) map(mtsv , nil))
         , A , RestL |-
         map-1(SVTr ,  map(SVX , X) map(SVY , Y) map(SVA , A)
                      map (SVB , B) map(mtsv , nil)) ,
         map-1(SVTa ,  map(SVX , X) map(SVY , Y) map(SVA , A)
                      map (SVB , B) map(mtsv , nil))
         , B , RestR)
        apply-tc( taclet( find(SVX |- SVY) || if(SVA |- SVB) || RestState )
                , (X , A , RestL |- Y , B , RestR) )
      if ok-mapset( map(SVX , X) map(SVY , Y) map(SVA , A) map (SVB , B)
                   map(mtsv , nil) ) and (A =/= nil or B =/= nil) .

--- taclet application in case a goal is closed ; the second part of the
--- condition is there because if A and B are equal to nil there is no
--- find part basically!
ceq apply-tc( taclet( find(SVX |- SVY)
                     ||  goaltemplate( rplcw( closedgoal )
                                       || RestState2 )
                     || if(SVA |- SVB) || RestState )
             , (X , A , RestL |- Y , B , RestR) )
      = (closedgoal)
        apply-tc( taclet( find(SVX |- SVY) || if(SVA |- SVB) || RestState )
                 , (X , A , RestL |- Y , B , RestR) )
      if ok-mapset( map(SVX , X) map(SVY , Y) map(SVA , A) map (SVB , B)
                   map(mtsv , nil)) and (A =/= nil or B =/= nil) .

--- taclet application in case all replacewith/add pairs have been
--- used up, i.e. no goaltemplates are left over!
ceq apply-tc( taclet( RestState ) , (X , A , RestL |- Y , B , RestR) )
      = closedgoal
      if taclet-has-goaltemplate( taclet( RestState) ) =/= true .

--- necessary for the case with no goaltemplates left over
--- (construction inspired by page 71 of the maude manual)
op taclet-has-goaltemplate : [LState] -> [Bool] .
eq taclet-has-goaltemplate( taclet( goaltemplate(RestState2)
                                    || RestState ) )
    = true .
```

```
--- constants of sort SchemaVariable to be used as Schema"Variables"
ops SA SB SC SD : -> SchemaVariable .


--- taclets given as constant operators below
-------------------------------------------------
--- // closing goals
op tc-close-goal : -> LState .
eq tc-close-goal =
      taclet( tacname('close-goal) || find(mtsv |- SB)
             || goaltemplate( rplcw(closedgoal) ) || if(SB |- mtsv) ) .
---close_goal       { if (b ==>) find (==> b) close goal
---                        heuristics(closure) };

op tc-close-goal-antec : -> LState .
eq tc-close-goal-antec =
      taclet( tacname('close-goal-antec) || find(SB |- mtsv)
             || goaltemplate( rplcw(closedgoal) ) || if(mtsv |- SB) ) .
--- close_goal_antec { if (==> b) find (b ==>) close goal };


--- to be applicable with the above application mechanism terms
--- (better schemavariableterms) have to be allowed where now only
--- schemavariables may stand, see also line 33 of this module.
--- this holds for all taclets below!!!
*****************************************************************

op tc-close-goal-by-false : -> LState .
eq tc-close-goal-by-false =
      taclet( tacname('close-goal-by-false) || find(f |- mtsv)
             || goaltemplate( rplcw(closedgoal) ) || if(mtsv |- mtsv) ) .
--- close_by_false   { find (false ==>) close goal heuristics(closure) };


op tc-close-goal-by-true : -> LState .
eq tc-close-goal-by-true =
      taclet( tacname('close-goal-by-true) || find(mtsv |- t)
             || goaltemplate( rplcw(closedgoal) ) || if(mtsv |- mtsv) ) .
--- close_by_true    { find (==> true) close goal heuristics(closure) };

--- // junctor rules

op tc-true-left : -> LState .
eq tc-true-left =
      taclet( tacname('true-left) || find(t |- mtsv)
             || goaltemplate( rplcw(mtsv |- mtsv) || add(mtsv |- mtsv) )
             || if(mtsv |- mtsv) ) .
--- true_left    { find (true ==>) replacewith(==>)
---                heuristics(concrete) };

op tc-false-right : -> LState .
eq tc-false-right =
      taclet( tacname('false-right) || find(mtsv |- f)
             || goaltemplate( rplcw(mtsv |- mtsv) || add(mtsv |- mtsv) )
             || if(mtsv |- mtsv) ) .
--- false_right  { find (==> false) replacewith(==>)
---                heuristics(concrete) };

op tc-not-left : -> LState .
```

```
eq tc-not-left =
      taclet( tacname('not-left ) || find(! SB |- mtsv)
              || goaltemplate( rplcw(mtsv |- SB) || add( mtsv |- mtsv) )
              || if(mtsv |- mtsv) ) .
--- not_left  { find  (! b ==>) replacewith(==> b)  heuristics(alpha) };

op tc-not-right : -> LState .
eq tc-not-right =
      taclet( tacname('not-right ) || find(mtsv |- ! SB)
              || goaltemplate( rplcw(SB |- mtsv) || add(mtsv |- mtsv) )
              || if(mtsv |- mtsv) ) .
--- not_right { find (==> ! b)  replacewith(b ==>) heuristics(alpha) };

--- PROBLEM: 2 replacewith parts!
op tc-imp-left : -> LState .
eq tc-imp-left =
      taclet( tacname('imp-left ) || find(SB -> SC |- mtsv)
              || goaltemplate( rplcw(mtsv |- SB) || add(mtsv |- mtsv) )
              || goaltemplate( rplcw(SC |- mtsv) || add(mtsv |- mtsv) )
              || if(mtsv |- mtsv) ) .
--- imp_left  { find  (b -> c ==>)
---             replacewith(==> b);
---             replacewith(c ==>)
---           heuristics(split,beta) };

op tc-imp-right : -> LState .
eq tc-imp-right =
      taclet( tacname('imp-right ) || find(mtsv |- SB -> SC)
              || goaltemplate( rplcw(SB |- SC) || add(mtsv |- mtsv) )
              || if(mtsv |- mtsv) ) .
---imp_right { find (==> b -> c)  replacewith(b ==> c)
---             heuristics(alpha) };

op tc-and-left : -> LState .
eq tc-and-left =
      taclet( tacname('and-left ) || find(SB &  SC |- mtsv)
              || goaltemplate( rplcw(SB , SC |- mtsv)
                                || add(mtsv |- mtsv) )
              || if(mtsv |- mtsv) ) .
--- and_left  { find  (b & c ==>) replacewith(b, c ==>)
---               heuristics(alpha) };

--- PROBLEM: 2 replacewith parts!
op tc-and-right : -> LState .
eq tc-and-right =
      taclet( tacname('and-right ) || find(mtsv |- SB &  SC)
              || goaltemplate( rplcw(mtsv |- SB) || add(mtsv |- mtsv) )
              || goaltemplate( rplcw(mtsv |- SC) || add(mtsv |- mtsv) )
              || if(mtsv |- mtsv) ) .
--- and_right { find (==> b & c)  replacewith(==> b);  replacewith(==> c)
---                 heuristics(split,beta) };

--- PROBLEM: 2 replacewith parts!
op tc-or-left : -> LState .
eq tc-or-left =
      taclet( tacname('or-left ) || find(SB |  SC |- mtsv)
              || goaltemplate( rplcw(SB |- mtsv) || add(mtsv |- mtsv) )
              || goaltemplate( rplcw(SC |- mtsv) || add(mtsv |- mtsv) )
              || if(mtsv |- mtsv) ) .
--- or_left   { find  (b | c ==>) replacewith(b ==>); replacewith(c ==>)
```

```
  ---               heuristics(split,beta) };

  op tc-or-right : -> LState .
  eq tc-or-right =
        taclet( tacname('or-right) || find(mtsv |- SB |  SC)
               || goaltemplate( rplcw(mtsv |- SB , SC)
                                || add(mtsv |- mtsv) )
               || if(mtsv |- mtsv) ) .
  --- or_right  { find (==> b | c) replacewith(==> b, c)
  ---             heuristics(alpha) };


  op tc-equiv-left : -> LState .
  eq tc-equiv-left =
        taclet( tacname('equiv-left) || find(SB <-> SC |- mtsv)
               || goaltemplate( rplcw(SB , SC |- mtsv)
                                || add(mtsv |- mtsv) )
               || goaltemplate( rplcw(mtsv |- SB , SC)
                                || add(mtsv |- mtsv) )
               || if(mtsv |- mtsv) ) .
  --- equiv_left   { find (b <-> c ==>)
  ---                 replacewith(b, c ==>);
  ---                 replacewith(==> b, c)
  ---               heuristics(split,beta) };

  op tc-equiv-right : -> LState .
  eq tc-equiv-right =
        taclet( tacname('equiv-right) || find(mtsv |- SB <-> SC)
               || goaltemplate( rplcw(SB |- SC) || add(mtsv |- mtsv) )
               || goaltemplate( rplcw(SC |- SB) || add(mtsv |- mtsv) )
               || if(mtsv |- mtsv) ) .
  --- equiv_right  { find (==> b <-> c)
  ---                 replacewith(b ==> c);
  ---                 replacewith(c ==> b)
  ---               heuristics(split,beta) };
endm


mod PROPO-SEQUENT-TACLETS-EMBEDDING-SEMANTICS-PROOF is
  including PROPO-SEQUENT-SEMANTICS .
  including PROPO-SEQUENT-TACLETS-EMBEDDING .

  ---sort TruthValuePair .
  ---op _;_ : TruthValue TruthValue -> TruthValuePair .

  op evFormPair : Propo Propo VariableMap -> TruthValue .
  op fixAllAtomAssignmentsForPairs : Propo Propo VariableMap VariableMap
                                         -> TruthValue .

  vars A B C : Propo .
  vars M N : VariableMap .
  var X : Atom .


  eq evFormPair(A , B , M)
     = fixAllAtomAssignmentsForPairs(A , B , M , emptyvarmap) .

  eq fixAllAtomAssignmentsForPairs(A , B , M (X mappedto t) , N)
     = fixAllAtomAssignmentsForPairs(A , B , M , N (X mappedto t)) .
  eq fixAllAtomAssignmentsForPairs(A , B , M (X mappedto f) , N)
```

```
      = fixAllAtomAssignmentsForPairs(A , B , M , N (X mappedto f)) .

   eq fixAllAtomAssignmentsForPairs(A , B , emptyvarmap , N)
      = evalFormula(A , N) <-> evalFormula(B , N) .
endm
```

C.1 Examples for Use of the Imitated Taclets

Here we have the examples for the subsection 6.2.

This first example is a check whether the combination of all possibilities of disjunction of 4 variables (and their negations) which are in turn connected by conjunction each, and the whole is afterwards negated, always returns true, or here returns closedgoal, which it does. Then the same happens in the 3 variable case and the 2 variable case.

```
4 Vars:

rew nil
|-
!(  (a | b | c | d)
  & (a | b | c | ! d)
  & (a | b | ! c | d)
  & (a | b | ! c | ! d)
  & (a | ! b | c | d)
  & (a | ! b | c | ! d)
  & (a | ! b | ! c | d)
  & (a | ! b | ! c | ! d)
  & (! a | b | c | d)
  & (! a | b | c | ! d)
  & (! a | b | ! c | d)
  & (! a | b | ! c | ! d)
  & (! a | ! b | c | d)
  & (! a | ! b | c | ! d)
  & (! a | ! b | ! c | d)
  & (! a | ! b | ! c | ! d)) .


3 Vars:

rew nil
|-
```

```
!(   (a | b | c )
   & (a | b | ! c )
   & (a | ! b | c )
   & (a | ! b | ! c )
   & (! a | b | c )
   & (! a | b | ! c )
   & (! a | ! b | c )
   & (! a | ! b | ! c )) .


2 Vars:

rew nil
|-
!(   (a | b)
   & (a | ! b)
   & (! a | b)
   & (! a | ! b)) .
```

There are also more examples which we do not mention here.

C.2 Propositional Logics Semantics Test Examples

These are examples which test the semantics of propositional logic taclets given in Maude above.

```
--- result is true
search evForm( FormulaFromSequent(nil
|-
!(   (a | b | c | d)
   & (a | b | c | ! d)
   & (a | b | ! c | d)
   & (a | b | ! c | ! d)
   & (a | ! b | c | d)
   & (a | ! b | c | ! d)
   & (a | ! b | ! c | d)
   & (a | ! b | ! c | ! d)
   & (! a | b | c | d)
   & (! a | b | c | ! d)
   & (! a | b | ! c | d)
   & (! a | b | ! c | ! d)
```

```
  & (! a | ! b | c | d)
  & (! a | ! b | c | ! d)
  & (! a | ! b | ! c | d)
  & (! a | ! b | ! c | ! d))) , (a mappedto TorF)
       (b mappedto TorF) (c mappedto TorF)
       (d mappedto TorF) )
=>! X:TruthValue .


---- result is true, false;
---- this is not universally valid, the negation (!) of
---- the last atom "d" is missing
search evForm( FormulaFromSequent(nil
|-
!(  (a | b | c | d)
  & (a | b | c | ! d)
  & (a | b | ! c | d)
  & (a | b | ! c | ! d)
  & (a | ! b | c | d)
  & (a | ! b | c | ! d)
  & (a | ! b | ! c | d)
  & (a | ! b | ! c | ! d)
  & (! a | b | c | d)
  & (! a | b | c | ! d)
  & (! a | b | ! c | d)
  & (! a | b | ! c | ! d)
  & (! a | ! b | c | d)
  & (! a | ! b | c | ! d)
  & (! a | ! b | ! c | d)
  & (! a | ! b | ! c |  d))) , (a mappedto TorF)
       (b mappedto TorF) (c mappedto TorF)
       (d mappedto TorF) )
=>! X:TruthValue .
```

Bibliography

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

[Bec00] Bernhard Beckert. A dynamic logic for java card. In *Proceedings, 2nd ECOOP Workshop on Formal Techniques for Java Programs, Cannes, France*, pages 111–119, 2000.

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

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

[CDE$^+$00] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *A Maude Tutorial.* SRI International, 2000. Available from `http://maude.cs.uiuc.edu/papers/`.

[CM00] Manuel Clavel and José Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.

[FCMR04] Azadeh Farzan, Feng Chen, José Meseguer, and Grigore Rosu. Formal analysis of java programs in javafan. In Rajeev Alur and Doron Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 501–505. Springer, 2004.

[FMR04] Azadeh Farzan, José Meseguer, and Grigore Rosu. Formal jvm code analysis in javafan. In Charles Rattray, Savi Maharaj, and Carron Shankland, editors, *AMAST*, volume 3116 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2004.

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

[MOM99] N. Martí-Oliet and J. Meseguer. Action and change in rewriting logic. In R. Pareschi and B. Fronhofer, editors, *Dynamic Worlds: From the Frame Problem to Knowledge Management*, volume 12 of *Applied Logic Series*, pages 1–53. Kluwer Academic Publishers, 1999.

[MOM00] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.

[MOM02] Narciso Martí-Oliet and José Meseguer. Rewriting logic: roadmap and bibliography. *Theor. Comput. Sci.*, 285(2):121–154, 2002.

[MR04] José Meseguer and Grigore Rosu. Rewriting logic semantics: From language specifications to formal analysis tools. In *Proceedings of the IJCAR'04, Cork, Ireland*, volume 3097, pages 1–44. Springer-Verlag LNCS, July 2004.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction.* Wiley, 1992.