Static Universe Type Inference

Nathalie Kellenberger

Master Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

http://sct.inf.ethz.ch/

September 30, 2005

Supervised by:
Dipl.-Ing. Werner M. Dietl
Prof. Dr. Peter Müller

Software Component Technology Group
inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

The Universe type system gives programmers the possibility to specify ownership information in Java source code and enforces a stricter runtime behavior that eases reasoning about software. These additional type modifiers have to be provided by the programmer. Adding these modifiers to an already existing large program can be a big effort. We developed an architecture that infers the Universe type annotations by static code analysis. As a proof of concept the architecture was implemented and successfully applied to real Java programs. This report covers every aspect of the information gathering and processing needed to perform this task.

Contents

Chapter 1

Introduction

1.1 Motivation

Each reference in a program allows a write access to the appropriate object. If a reference to a whole data-structure can be gained, the built structure can be destroyed. The Java programming language covers objects only by visibility, which still allows to gain a write access over a reference returned by a method. The Universe type system [1] can be applied to Java programs to protect compact designed structures.

The goal of this project is to infer the Universe types of a given program by statically analyzing all type relations with Prolog.

1.2 Introduction to the Universe Type System

1.2.1 Aliasing

Whenever we have more than one reference, that points to an object, we are talking about aliasing. All references in a program have the same rights on the object they point to. If aliasing is not controlled, it is hard to reason about the behavior of a program.

Representation Exposure

Representation Exposure is the main problem, which occurs with aliasing. A client can gain access to data-structures or values of a program and is therefore allowed to change and destroy them. The representation of an object or a whole structure is not fully encapsulated, but leaks.

```
class Person {
        Address adr;

        public void set(Address a) {
                this.adr = a;
        }

        public Address get() {
                return this.adr;
        }
}

//in another class
Person p = new Person();
p.set(new Address());          // anyone can change the address of a person
Address a = p.get();           // anyone can get a reference to the address field
```

In the example anyone can change the address of a person. If we are administrating a database of addresses, it cannot be allowed, that a client can change any address in the database.

We can also get a reference to the internally stored address object of a person with a call to the `get()` method. This reference is package private and no client should be allowed to use this reference to change the address of an arbitrary person. Clients should only read addresses of other persons but not change them.

1.2.2 Universe Type System

The Universe type system was created to prevent aliasing. It divides the object store in so called universes. Each object can only belong to exactly one universe and the universes themselves are hierarchical structured. Objects can be in the same universe or in the inner universe of another object and therefore belong to this object.

The Universe type system controls references across universe boundaries. There is no write access allowed over a reference, which points into another universe. Write access is only allowed to objects of the same universe or to an universe, which is owned by the object. The Universe type system guarantees the following invariant [1]:

If object X holds a direct reference to object Y then either

1. X and Y belong to the same universe, or
2. X is the owner of Y, or
3. the reference is read-only.

The Universe type system realizes this invariant with the following three type modifiers:

1. *peer*
   This modifier is used to denote, that references are peer to each other, meaning they point to objects, which lie in the same context.
2. *rep*
   This modifier is used to create a new universe, which belongs to another object. The object itself lies in this inner context and cannot be changed from outside but only over the owner of the universe.
3. *readonly*
   This modifier designates references, which point to an arbitrary context. It is only possible to read from another universe than the own. A *readonly* reference can never change the object it points to.

To prevent our address database from changes of clients, we would have to apply the Universe type system in the following way:

```
class Person {
        rep Address adr;

        public void set(rep Address a) {
                this.adr = a;
        }

        public pure rep Address get() {
                return this.adr;
        }
}

//in another class
peer Person p = new peer Person();
p.set(new rep Address());                // not possible anymore
readonly Address a = p.get();            // can only read but not write
```

In the example we can change the address of a person anymore, because a person owns its address and only the person itself is allowed to change it. The `get()` method will now always return a *readonly* reference, when not invoked on the *this* object. Therefore a person is only allowed to read the address of another person.

```
rep <:  [p] * par(set)       =>       rep <:  peer * rep <:  readonly
[p] * ret(get) <:  [a]       =>       peer * rep <:  readonly
```

The combination of the Universe type of the target with the Universe type of the formal parameter of the method `set` results in *readonly*. The Universe type of the argument is *rep* and a subtype of *readonly*. The method call of the `set` method is not legal, because we are not allowed to invoke a non-pure method with *rep* parameters on any other target than the *this* object.
The combination of the Universe type of the target with the return type of the `get()` method also results in *readonly*. We assign the return type of the method call to a *readonly* reference. Like this the subtype relation of the assignment is fulfilled.

The type rules of the Universe type system are showed in detail later, when discussing their modelling in the Prolog system.

Pure Methods

Another construct of the Universe type system are *pure* methods. *Pure* methods are not allowed to change any existing objects and can only call other *pure* methods. Therefore it is safe to invoke a *pure* method on a *readonly* reference, because the purity of the method assures, that the object will not be changed by the method. It would be too restrictive to forbid the calling of any method on *readonly* references.

An example of a *pure* method is the `Object.equals(Object o)` method, which only compares two objects.

1.3 Prolog Introduction

In this master thesis the programming language Prolog and the SWI Prolog system is used. For those who do not know about logic programming, we give a short introduction here.

1.3.1 Logic Programming

A Prolog program is composed of facts and rules, which model the problem, one wants to solve and build a Prolog database. By asking a question to this database, one can get none, one ore several solutions.

The facts in a prolog program describe the basic set of the formulated problem. What we want to get in a solution, has to be written as a constant in a fact. A fact is the last step in a prolog program.

Prolog rules describe relations and are mostly recursive. Rules try to apply themselves as long as a fact can be found, which gives the solution of the rule.

To run a Prolog program, one has to ask a question to the created database. The Prolog system searches as long in the database and applies rules and facts until a solution is found or all applying facts and rules are visited. If a solution is found, one can enforce the system to backtrack until all possible solutions are found.

To answer a Prolog query, the Prolog system uses unification, which is a kind of pattern matching. All Prolog variables in a query return a constant of the created database in a solution. While the Prolog system looks for a solution, a Prolog variable is initialized with a constant and keeps

this value as long as this solution is rejected by the system, because not all parts of the query can be fulfilled.

The order of the Prolog facts and rules is important because the Prolog system always tries out the first fact or rule.

1.3.2 Prolog Syntax

Constants

Constants in facts are part of the solution set. Every variable returns a constant in a solution. Constants can be: Names starting with a small letter, strings or numbers. Strings in Prolog are enclosed by apostrophes.

```
alfred, 'Have fun', 8
```

Variables

Prolog variables are used in queries and rules and must start with a capital letter:

```
Variable, Var1, N, Give_me_a_solution
```

Logical Operators

In Prolog the following logical operators are often used:

logical operator	meaning
,	logical conjunction
;	logical disjunction
:-	logical implication
not	negation
==	equality
\==	inequality
A -> B;C	if A then B else C

Facts

Facts are the smallest part of a Prolog program and are used as a fact of the design or to end recursion:

```
father(alfred, ben).          alfred is the father of ben
father(alfred, mary).
father(john, anna).
father(arthur, alfred).
```

The name of the fact is `father` and the Prolog system saves the fact as a binary `father/2` fact, meaning it has two parameters.

Rules

A rule is used to model a relation and has a head and a body. In the body of the rule one can call the same or other rules, facts, write to files or execute arithmetical expressions.

```
grandfather(X, Y) :- father(X, Z), father(Z, Y).
```

Queries

A query can consist of one or more calls to rules or facts and contains Prolog variables and constants.

Query	*Answer of the system*
`?- father(alfred, ben).`	`Yes`
`?- father(ben, alfred).`	`No`
`?- father(X, mary).`	`X = alfred`
`?- grandfather(X, Y).`	`X = arthur Y = alfred`
`?- father(X, Z), father(Z, Y).`	`X = arthur Y = alfred`

With the semicolon one enforces the system to backtrack:

```
?- father(alfred, X).          X = ben ; X = mary.
```

Arithmetic Expressions

Arithmetic expressions are also possible in Prolog. Prolog does not use the assignment operator for arithmetic expressions, but the keyword ”is”:

```
N is 5*3
```

File Handling

Prolog rules can write to files as a side effect, for example to store a solution. But one has to be careful with file writing in rules. If the Prolog system reaches a rule and writes part of a solution to a file, it can happen, that the rule is not fulfilled for the whole query and the system backtracks and returns another solution. But the false solution has been already written to the file.

Lists

Lists are a very useful data-structure in Prolog especially for the combination of results in recursion. The list itself is defined as a recursive data-structure:

```
List -> []
List -> [Element | List]
```

The recursive definition of the list is one possibility to display a list and traverse it in a recursive rule.

`[a, b, c, d]`	can be written as	`[a, b, c, d | [ ]]` or
		`[a, b, c | [d]]` or
		`[a, b | [c, d]]` or
		`[a | [b, c, d ]]`

If one wants to traverse a list, one detaches the first element of the list and applies the rule to the tail of the list. For example to check if an element is contained in a list:

```
member(X, [X|List).
member(X, [Element|List]) :- member(X, List).
```

The list is traversed as long as the first element of the list is not the element we are looking for.

1.3.3 Backtracking

We show how backtracking in Prolog works with the following example query to the database presented before.

```
?- father(X, Y), grandfather(X, Z).

father(alfred, ben).                X = alfred, Y = ben.
grandfather(alfred, Z) :- father(alfred, A), father(A, Z).
father(alfred, ben).                A = ben.
father(ben, Z).                     no.
father(alfred, mary).               A = mary.
father(mary, Z).                    no.

father(alfred, mary).               X = alfred, Y = mary.
grandfather(alfred, Z) :- father(alfred, A), father(A, Z).
father(alfred, ben).                A = ben.
father(ben, Z).                     no.
father(alfred, mary).               A = mary.
father(mary, Z).                    no.

father(john, anna).                 X = john, Y = anna.
grandfather(john, Z) :- father(alfred, A), father(A, Z).
father(john, anna).                 A = anna.
father(anna, Z).                    no.

father(arthur, alfred).             X = arthur, Y = alfred.
grandfather(arthur, Z) :- father(alfred, A), father(A, Z).
father(arthur, alfred).             A = alfred.
father(alfred, ben).                Z = ben.

;
father(alfred, mary).               Z = mary.
```

The Prolog system tries to fulfill the first part of the query and checks, if the found solution also works for the second part. The first fact of the database is used as a solution and the grandfather rule is called with this solution. The first solution is no possible solution, because ben and mary do not have children in our database. The second tried solution leads to the same problem. The last solution does finally work out and leads to the two possible solutions.

The order of the facts and rules in the database is important, because the first rule or fact is always tried out first for a solution. If the fact `father(arthur, alfred)` would have been written on the top in the database, all possible solutions would have been found in the first step and no backtracking would have been needed.

A query is always asked top down and therefore the order of the facts to call in a query is also important, because backtracking starts the other way around. When we change our query to `?- grandfather(X, Z), father(X, Y).` we would also avoid some backtracking, because the second part of the query is implicitly part of the grandfather rule and therefore always fulfilled if the grandfather rule is fulfilled.

```
?- grandfather(X, Z), father(X, Y).

grandfather(X, Z) :- father(X, A), father(A, Z).
father(alfred, ben).                X = alfred, A = ben.
father(ben, Z).                     no.
father(alfred, mary).               X = alfred, A = mary.
father(mary, Z).                    no.

grandfather(X, Z) :- father(X, A), father(A, Z).
```

```
father(john, anna).                 X = john, A = anna.
father(anna, Z).                    no.

grandfather(X, Z) :- father(X, A), father(A, Z).
father(arthur, alfred).             X = arthur, A = alfred.
father(alfred, ben).                Z = ben.
father(arthur, Y).                  Y = alfred.
```

1.3.4 Limits of the Prolog Language

Some things, which are normal to use in other programming languages are not suitable for Prolog. A global counter for example could be implemented, but one has to cheat the Prolog system to enforce that. Accustomed expressions like the following cannot be used in Prolog:

`n = n+1;` in Prolog `N is N+1` does not work

Once a Prolog variable was initialized with a value, it sticks to it and cannot be assigned another one. A simple counter - not a global one! - can be implemented by using recursion:

```
count(0).
count(N) :- count(M), N is M + 1.
```

If we enforce the system to backtrack, the counter moves on:

```
?- count(X).       X = 0 ; X = 1 ; X = 2 ; X = 3 ; X = 4
```

In the Prolog language one has to use rules instead of methods or functions. But a rule has no return type. If one needs to return a value, this is just another parameter of the rule, which will be instantiated with the solution of the rule. Because of this the composition of rules is not possible.

Prolog rules are not transitive. If a transitive behavior is wished, the user has to implement each relation by himself.

```
smaller(x, y).
smaller(y, z).
```

When asking the query `?- smaller(X, Y).` the Prolog system would only reply with

```
X = x, Y = y ; X = y, Y = z
```

For the third solution can be found by the Prolog system, it has to be given as an additional fact by the programmer:

```
smaller(x, z).
```

1.3.5 Prolog System

The choice of the Prolog system gives us some advantages as well as some disadvantages and we have to consider them both in our design.

The great advantage of the Prolog system is, that it can find all solutions of a given problem. One can formulate a given problem into Prolog rules and the system applies them as long as one solution - or none, if there exists none - is found. By enforcing the system to backtrack, one can get all possible solutions. A Prolog program always terminates and so the user knows, if there is one, several or none solution to his stated problem. This is the reason, why the Prolog system has been chosen for our solution.

But we have to deal with some disadvantages of the system as well. Prolog variable and file names have to be in a certain form, that they will be accepted by the system. The grouping of rules cannot be arbitrarily and the order of them might become important.

Because of the way the Prolog system works, some design steps are already given. All the used rules; like the rules of the Universe type system and the type relations of a given Java program, have to be represented by Prolog rules. Rules of the same type - meaning rules with the same name and number of parameters - have to be in the same file and we have to have a look at the order of them. The Prolog system always tries out the first rule of a list of rules to find a solution. This gives some of the structure of our Prolog program.

1.4 Goal

The goal of this project is to develop and implement a modular architecture, which annotates a regular Java program with the Universe type system statically. This is called static type inference. The implemented architecture should be able to extract all necessary type relations by analyzing the syntax tree of each class of a program. These type relations should be written in a way, the Prolog program of the architecture can use them to find all possible annotation solutions. By applying heuristics to a specific set of found solutions, we should be able to display the approximative best solution. The incorporation of the annotation into the original Java program is not part of this project. This is done in the semester project of Marco Meyer [10].

It is also not part of this project to be able to determine *pure* methods. It is task of the user to specify, if a method has side effects or not and cannot be detected by static inference.

1.5 Outline

First we discuss the developed architecture in chapter 2. In chapter 3 we show some implementation details and how the static inference tool is used. In the last chapter we present the results of this project.

Chapter 2

Architecture

In this chapter we describe the modular architecture, that we developed to infer Universe types. The architecture is composed of three main parts:

1. Constraints Generator
2. Solver (with heuristics)
3. Solution Writer

All type relations of a given program are statically extracted out of the syntax tree and written as constraints. These can be used by a solver to infer the needed Universe types. Some heuristics have to be applied, to find the approximative best annotation solution in reasonable time. This solution is then written in an XML format, which can be used to annotate the original program.

For this project, we chose the Prolog system as the solver system. All details are discussed with regards to the Prolog system. Though the implementation of our design allows to use another system for the solver.

First we give an overview over the single parts of the architecture in 2.1 and show the functionality on a small example in 2.2. We discuss the designs, which were rejected in 2.3. At the end of this chapter in 2.4 we discuss our developed architecture in detail.

2.1 Overview of the Parts of the Architecture

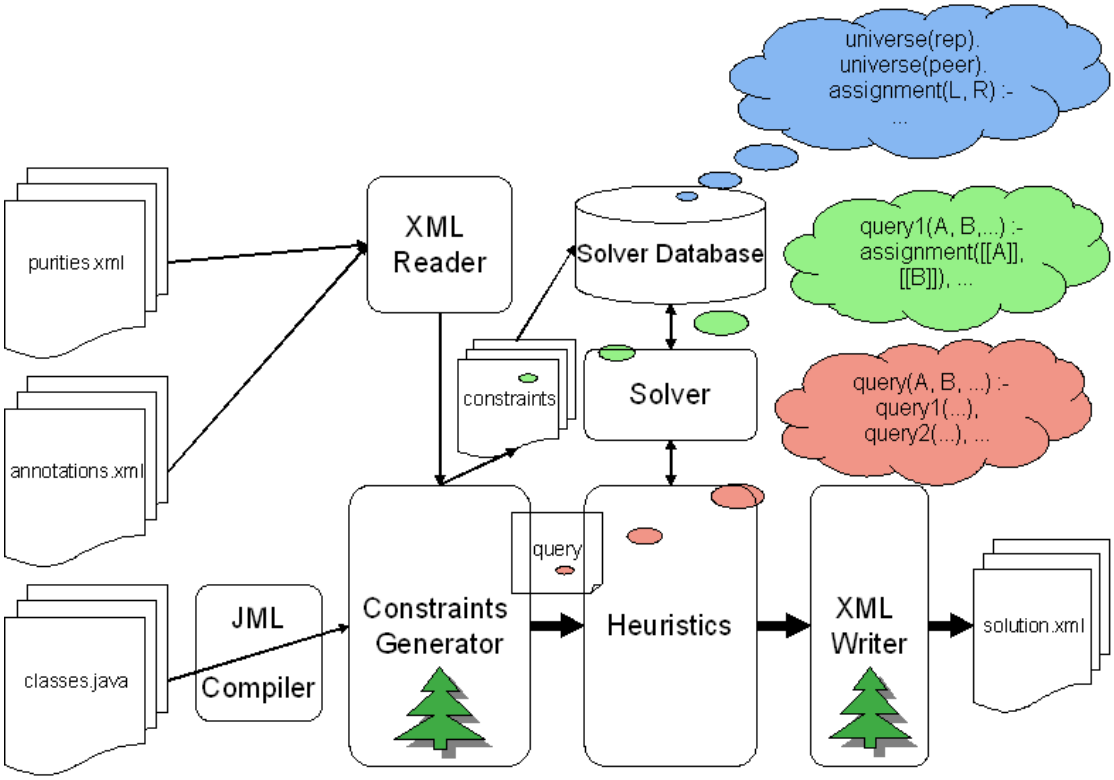

Figure 2.1: The steps of the architecture

We have the following predefined inputs:

- all classes of a Java program
- purities of all methods
- already annotated variables

Variables can be annotated directly in the original program or the annotation can be given in a separate XML file like the purities of the methods. These inputs suffice to determine all possible annotation solutions for a given program.

2.1.1 Generation of Constraints out of the Syntax Tree

We want to annotate a given program with the Universe type system. Therefore we have to extract all type relations of the original program. The Universe type system is a conservative extension of the Java type system. We still have the same type relations of expressions as usual, for example, that the right-hand side of an assignment needs to be subtype or equal to the left-hand side.

All type relations can be gained by traversing the syntax trees of all classes. We use the JML compiler to compile and type-check the given Java source and parse all used libraries. The JML compiler also grant us access to the syntax tree of a class. We need a way, to write down all type relations, so that they can be used for type inference. The most common way is to use a kind of abstract syntax tree representation. But we do not have to model each part of the syntax tree. We only design the relevant parts, which have type restrictions. All type relations of the syntax tree are then written down in the form of the designed constraints.

We designed the constraints in a way, they can be used by the Prolog system. But the design should still be usable for other solver systems. There is a Prolog rule for each expression written in a separate Prolog program. All these rules apply Universe type rules to Java expressions and build the Universe type rule database. A call to a rule is generated for each expression of the syntax tree as a constraint. By stringing all generated constraints together, we get a query to the Prolog database of Universe type rules. The query is fulfilled if and only if we receive a possible annotation for all variables as the solution. Otherwise it is not fulfilled and the query fails.

The generation step of the architecture is written in Java and takes several XML and Java files as an input and has several Prolog files as an output. We generate a query rule for each class, which asks all generated constraints to the database. The query head contains all variables and the body contains all generated constraints. The queries of all classes are combined in a global query rule, which calls all queries of all classes. Like this we can reuse generated constraints of a single class, when annotating the class in another context. When we ask the global query to the Prolog system, the Prolog database gives us all possible solutions to annotate the original Java source.

2.1.2 Finding Solutions considering Heuristics

The solver system is used to infer the Universe types of a program. We ask the global query to the solver and get in return a set of all possible annotation solutions. The query to the system is simply the head of the global query rule, which does nothing else than call all generated constraints.

Normally the query to the Prolog system would compute a great amount of solutions or in the case of partly annotated programs maybe not even a single one. To reduce the number of the solutions to exactly one, we apply several heuristics to all possible solutions or a subset of found solutions. We traverse a list of solutions and apply a heuristic rule to each. The solution, which fulfills all heuristics, will be returned at the end.

For the consideration of partly annotated programs and to get a greater amount of solutions, we have to track down type conflicts in expressions and resolve them by type casts. This can be done by allowing less restrictive type relations within the solver database.

This part of the architecture is done in Prolog and in Java. We need a Java plug-in to call a Prolog query within Java code. The Prolog database of the Universe type rules is written in a separate file and loaded with the generated constraints into the Prolog system. All together form a Prolog database, which can evaluate the global query. The Java plug-in allows to iterate over all possible annotation solutions and to apply heuristics to each of them.

2.1.3 Generation of the XML Output

After the call of our query, we get one possible annotation solution by applying heuristics to a subset of solutions found by the Prolog system. The solution is then written to a separate XML file by traversing all syntax trees again. When traversing all syntax trees again, we have all the information we need, to write down the annotation of all variables to the XML output. During this second traversal of the syntax trees we can also resolve conflicts found by the solver. We check each expression again for conflicting types and write down an additional cast if necessary.

The user can choose, if he wants to generate one XML solution file for all compiled classes or for each class a separate one. One solution XML file for each class could be useful by reusing this solution in a query for a partly annotated program.

2.1.4 Benefits

The approach of our solution is to strictly separate the constraints generator, the type inference and the solution writing part of the design. All fixed actions like the traversal of the syntax tree and the reading and writing of XML files are written in Java. Once written they should not

need much of a change afterwards. The abstract syntax tree representation of Java expressions in Prolog does not change and so the generation step stays the same.

The benefit of our solution is, that later changes to the Universe type system can be applied easily. If we do not change the design of the constraints, we still can change the database, which is applied to the constraints and apply more restrictions, less or even new ones. The Prolog system can also be exchanged by another solver system. It is also simple to apply other heuristics or add some more extensions, because these parts of the architecture are fully configurable by the user.

2.2 Introductory Example

We want to illustrate the single steps of our architecture on a simple example before going into detail.

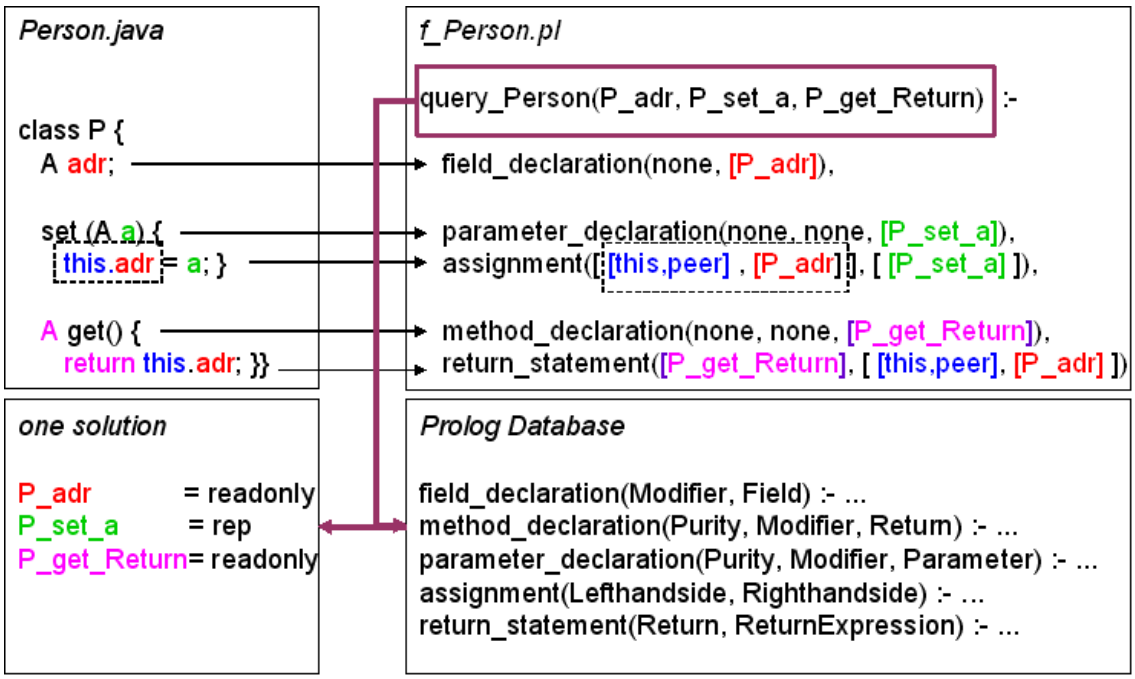

Figure 2.2: Steps of the architecture for a simple example

We get a simple input Java program with just one single class we want to annotate:

```
class Person {
        Address adr;

        void set(Address a) {
                this.adr = a;
        }

        Address get() {
                return this.adr;
        }
}
```

First of all we traverse the syntax tree of the class and look at every single expression. We generate a call to a related Prolog rule for every expression, which looks like an abstract view of the expression itself. The following expressions will generate a call to a rule:

- `Address adr;`
  field declaration of a non-static field
  `field_declaration(none,[Person_adr])`

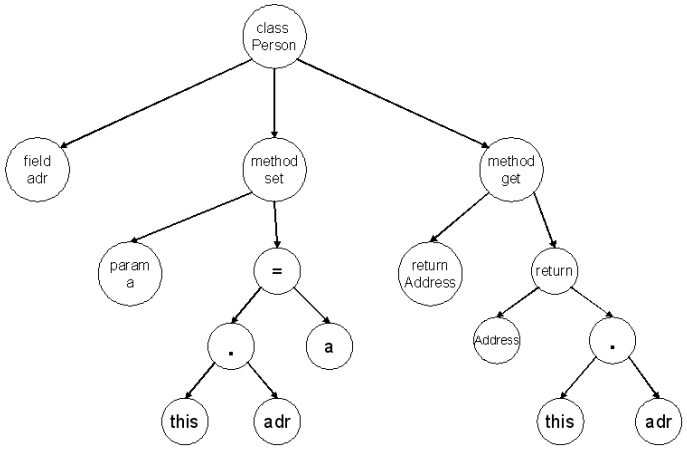

Figure 2.3: Syntax tree of the simple example

- `public void set(Address a)`
  parameter declaration of a non-pure and non-static method
  `parameter_declaration(none,none,[Person_set_a])`

- `this.adr = a;`
  assignment with a field access on the left-hand side
  `assignment([[this],[Person_adr]],[[Person_set_a]])`
  the field access is done within the assignment rule and is displayed in the head of the rule as a list of the appended variables
  `[this], [Person_adr]`

- `public Address get()`
  method declaration of a non-pure and non-static method
  `method_declaration(none,none,[Person_get_Return])`
  the method declaration rule is only used for the declaration of the return type

- `return this.adr;`
  return statement with a field access
  `return_statement([Person_get_Return],[[this],[Person_adr]])`

If we add up all these calls to the rules of the Prolog database, we can build a query rule. The query head contains all variables of the query body.

```
query_Person(Person_adr, Person_set_a, Person_get_Return) :-
       field_declaration(none,[Person_adr]),
       parameter_declaration(none,none,[Person_set_a]),
       assignment([[this],[Person_adr]],[[Person_set_a]]),
       method_declaration(none,none,[Person_get_Return]),
       return_statement([Person_get_Return],[[this],[Person_adr]]).
```

The head of the query rule is used to execute the query. If we ask the query head to the Prolog database, we will get a first solution for all the variables in the query head:

```
?- query_Person(Person_adr, Person_set_a, Person_get_Return).
```

And the system replies with:

```
Person_adr        = readonly
Person_set_a      = rep
Person_get_Return = readonly
```

2.3 Rejected Designs

The following designs were rejected during the design step of the project. We present them shortly in this section to show the development of the architecture and to emphasize once more the benefits of our design.

2.3.1 Type relations as facts

One approach was to design the constraints in a way they contain all the necessary information of an expression of the syntax tree. The constraint would be written as Prolog fact to a file. A separate Prolog program, which contains the Universe type rules, would then use these facts and produce a solution. The facts would contain all variables from the original Java program as constants.

First it seemed a good approach to just write down all the type relations as facts and handle them later by an additional Prolog program. Later it did not seem to us like the Prolog way of handling a problem. Prolog only needs to evaluate the type relations given by a program. Prolog does not need to care for names or types of variables. These are only needed for writing the solution in the XML format. We just feed the Prolog system with an abstract form of all type relations, which gives us the solution. Therefore it seems better to model the Universe type rules in Prolog and query the system with the type relations out of the syntax tree. We generate no facts but a query to the type relations, which should give the correct answer as a solution.

In the introductory example this approach might have generated Prolog facts like the following:

```
field(address, name_adr, type_Address).
method(name_set, returntype(void, type_void),
      [parameter(address, name_a, type_Address)]).
assignment(field_access(this, name_adr), name_a).
method(name_get, returntype(address, type_Address), []).
return(field_access(this, name_adr)).
```

2.3.2 No separate Prolog Program

As one of the first approaches we generated one rule for each expression of the syntax tree. All rule heads for a class were then connected to a bigger solution rule, the solution query for one class. We also had to connect all solution query heads to a global query. The global query returns the solution for all variables from all classes. The single rules, which are generated for each expression, apply directly all used type rules for the expression; like combination and sub-typing, which are the only predefined rules.

The great benefit of the separate Prolog program is, that we can later change the type rules of the Universe type system and the generation out of the syntax tree does not change. By using a separate Prolog program we do not generate the type relation for an expression, but generate a call to a predefined rule, which can be changed anytime. It is even possible to write a whole new Prolog program for all constraints.

In the introductory example this approach might have generated Prolog rules like the following:

```
constraint1(P_adr, P_set_a)                    :- combination(this, P_adr, Result),
                                                  subtype(P_set_a, Result).
constraint2(P_adr, P_get_Return)               :- combination(this, P_adr, Result),
                                                  subtype(Result, P_get_Return).
constraintPerson(P_adr, P_set_a, P_get_Return) :- constraint1(P_adr, P_set_a),
                                                  constraint2(P_adr, P_get_Return).
```

2.3.3 Prolog used as more than a Solver

From the architectural point of the design a first approach was to decouple Prolog from Java. The front of our inference tool would then be written and executed in Java and the tail in Prolog. We would read in the annotations of already annotated classes and the purities of all methods in Java. This information would have been used while generating constraints out of the syntax tree. Then the Java part of the tool is over.

The Prolog part of the program can be called from the Prolog system and will directly apply heuristics and write the solution in the XML format. The applying of heuristics and the writing of the solution in the XML format is possible in Prolog, but not very nice. Prolog has the possibility to write to files, but we would have to generate some extra information out of the syntax tree and generate more complex queries.

The Java programming language is much more qualified for reading and writing XML, and we can save all necessary information in Java data-structures like hash-tables. Therefore we decided to integrate the Prolog part of the tool into the Java part. We use a Java interface to the Prolog system and call the Prolog query within the Java code. The flow still remains linear, but the two parts are connected together. They still can be used independently.

2.3.4 One Query for all Expressions

If we write just one big query, which contains all calls to Prolog rules, we do not have to connect the generated queries to a global one. All the same we decided to generate a query for each class. Like this we do not have to generate the query for a class again, if it already has been generated. We do have to connect the queries of all compiled classes to a global query, but we might reuse queries of already compiled classes.

2.4 Detailed Description of the Architecture

2.4.1 Constraints Generator

In this section we want to explain more precisely, how the constraints are generated out of the syntax tree. We show for each expression, what constraints will be generated and how they look like. With the aid of some examples we hope to visualize the generation step of the architecture.

For the conversion of Java variable names into Prolog variable names in the examples in this section, we use the following naming conventions:

`A a;`	generates the prolog variable name	`[V_a]`
`A[] a;`	generates the prolog variable name	`[V_a, V_a_El]`

Variable ”a” of type ”A” is in the class ”V”. We do not distinguish between fields and local variables, so that the names in the examples do not get too long.

We encapsulate each Prolog variable in a list, because we have two different representations of simple references and array references. An array reference needs two Universe types. With the list representation for all references, we do not have to write each Prolog rule in the database twice: once for references and once for arrays.

The following short names are used additionally:

- `MPurity`
  acts as a placeholder for the purity of the called method in the examples and is *pure* or *none*
- `[MRet]`
  Prolog variable name for the method return type in the examples

- `[MPar]`
  Prolog variable name for a formal parameter of a method in the examples

To annotate a given Java program, we have to extract all type relations from the syntax tree. We have to store each type relation in a constraint, that is used in the type inference step. For human debuggers a constraint should still resemble the original expression. The design of the constraints must be as general, that it is still useful, when all other steps of the architecture change.

Each designed constraint is used as the head of a Prolog rule. When generating constraints, we actually build calls to the appropriate Prolog rules. The design of the constraints needs to be fixed, so the generation step of the architecture needs no changes in the future. That is why we explain the generation step first and the Prolog database afterwards.

We chose as design of the constraints an abstract syntax tree representation. We do not use all existing expressions of a program, but only those, who are useful for the type inference; meaning those, that restrict types of variables.

Therefore we design a constraint for each of the following expressions:

- field declaration
- local variable declaration
- method declaration
- object creation
- return statement
- method call
- assignment
- equality and relational operators
- cast
- static call

Other expressions - like conditionals or loops for example - are not needed, because they do not give additional type information but only use the mentioned expressions.

Simple Reference

For every reference in a Java program a Prolog variable is generated. The Prolog variable contains the Universe type of the appropriate reference.

`c` is reference generates `[V_c]`

Array references and the *this* reference are a bit special and will be discussed next.

Array Reference

Arrays are a bit special, because they need two Universe types: one for the array reference and one for the array elements - if they are references too. We generate for every array two Prolog variables: one for the array reference and one for the array elements respectively.

`c` is array reference generates `[V_c, V_c_El]`

Arrays, which contain primitive types, are handled like simple references and only the array reference is generated:

`c` is array reference generates `[V_c]`

this Reference

Whenever we reach a *this* object, we generate the following:

`this` generates `[this]`

If later a *this* reference can be of type *readonly* again, we just generate `[this, readonly]` when necessary and would have to change some of the other rules, which are used - like the combination rule for example.

The usage of *super* generates also `[this]`, because there is no difference between *super* and *this* in the Universe type system.

Null and Primitive Types

The *null* reference generates the Prolog constant `[null]`.

If we hit a primitive type we write the Prolog constant `[primitive]`.

Field and Array Access

```
field_access_right(FieldList, ResultingType)
field_access_left(FieldList, ResultingType)
```

`FieldList` is the list of all Prolog variables of the field access and `ResultingType` is the resulting Universe type of the field access. We need to distinguish between field accesses on the right-hand and on the left-hand side of an assignment. Field accesses on the left-hand side of an assignment are updating field accesses and more restricted than those on the right-hand side. The field access rule is normally only used implicitly within other predefined rules of the Prolog database as for example the `assignment` rule. We do generate the field access only as a list of the Prolog variables in the call to this rule then. But we need to generate a call to the field access rule if we are in a primitive assignment with field accesses in the operands of an addition for example.

For a chain of field accesses we produce a list of Prolog variables.

a.b.c.d generates `[V_a], [V_b], [V_c], [V_d]`

`a.f = ...`	`f` simple reference	generates	`[V_a], [V_f]`
`a.f = ...`	`f` primitive type	generates	`[V_a], [primitive]`
`a.f = ...`	`f` array reference	generates	`[V_a], [V_f, V_f_El]`
`this.f = ...`	`f` simple reference	generates	`[this], [V_f]`

An array access is exactly done in the same way as a field access to a simple reference:

`a[8] = ...` generates `[V_a], [V_a_El]`

An array access to an array, which contains primitive types and no references, is done the following way:

`a[8] = ...` generates `[V_a], [primitive]`

The access to the `length` field of the array class is ignored, because it is a primitive field and always allowed to call. We will therefore generate nothing for the following expression:

```
a.length
```

Field Declaration

```
field_declaration(FieldModifier, [Field])
field_declaration(FieldModifier, [FieldArray, FieldArrayElement])
```

`FieldModifier` is *static* or *none*. `Field` represents the Prolog variable for a field.

For every reference field declaration we generate a call to the `field_declaration` rule, where we generate a Prolog variable for the reference.

```
static C c;       generates       field_declaration(static, [V_c]),
C c;              generates       field_declaration(none, [V_c]),
C[] c;            generates       field_declaration(none, [V_c, V_c_El]),
```

We call the `field_declaration` rule with the identifier `static` instead of `none` for static fields.

Local Variable Declaration

```
local_variable_declaration(MethodModifier, [Local])
local_variable_declaration(MethodModifier, [LocalArray, LocalArrayElement])
```

`MethodModifier` is *static* or *none* and represents the modifier of the method. `Local` represents the Prolog variable for a local variable.

For every variable declaration we generate a call to the `local_variable_declaration` rule, where we generate a Prolog variable for the reference and tell the rule, if the variable is declared in a static method or not.

Assume for this example that we are in a static method:

```
C c;      generates       local_variable_declaration(static, [V_c]),
C[] c;    generates       local_variable_declaration(static, [V_c, V_c_El]),
```

Assume for the next example that we are not in a static method:

```
C c;      generates       local_variable_declaration(none, [V_c]),
C[] c;    generates       local_variable_declaration(none, [V_c, V_c_El]),
```

Method and Parameter Declaration

```
method_declaration(MethodPurity, MethodModifier, [ReturnType])
method_declaration(MethodPurity, MethodModifier,
      [ReturnTypeArray, ReturnTypeArrayElement])
```

`MethodPurity` is *pure* or *none* and `MethodModifier` is *static*, *public* or *none*. `ReturnType` represents the Prolog variable for the return type of a method.

Method declarations of methods with no reference return type only use the `parameter_declaration` rule. If the method has not even parameters, no rule at all is used for the method declaration.

```
parameter_declaration(MethodPurity, MethodModifier, [Parameter])
parameter_declaration(MethodPurity, MethodModifier,
      [ParameterArray, ParameterArrayElement])
```

`MethodPurity` is *pure* or *none* and `MethodModifier` is *static*, *constructor*, *public* or *none*. `Parameter` represents the Prolog variable for a parameter.

We generate a call to the `method_declaration` rule for a method signature, if the method has a reference return type and a call to the `parameter_declaration` rule for each of the parameters of a method:

```
/*@ pure @*/ static Node foo(Node n)    method_declaration(pure, static, [MRet]),
                                        parameter_declaration(pure, static, [V_n]),

public /*@ pure @*/ Node foo(Node[] n)  method_declaration(pure, public, [MRet]),
                                        parameter_declaration(pure, public,
                                              [V_n, V_n_El]),

public /*@ pure @*/ V(Node n, Object e) parameter_declaration(pure, constructor,
                                              [V_n]),
                                        parameter_declaration(pure, constructor,
                                              [V_e]),

public static Node foo(Node n)          method_declaration(none, static, [MRet]),
                                        parameter_declaration(none, static, [V_n]),

Node foo(Node n)                        method_declaration(none, none, [MRet]),
                                        parameter_declaration(none, none, [V_n]),

public V(Object e)                      parameter_declaration(none, constructor,
                                              [V_e]),
```

We have to tell the `method_declaration` and the `parameter_declaration` rule if we are in a pure, static or public method or a constructor.

The name of the method is not needed in the Prolog query, although it is generated while traversing the syntax tree. If we hit a method declaration, we generate the Prolog variable for the method name and use it to look up the given purity of a method in a hashtable. The method name is implicitly used in the query within the Prolog variable names for parameters and local variables.

Object Creation

```
new([Variable])
```

`Variable` represents the Prolog variable for the newly created object.

```
new([Array,Array_El])
```

`Array` represents the Prolog variable for the newly created array reference and `Array_El` represents the Prolog variable for the array elements.

For every object creation we produce a Prolog variable and a call to the `new` rule with this variable.

```
new Node()      generates      new([V_New1]),
new Node[5]     generates      new([V_New1, V_New1_El]),
```

Return Statement

```
return_statement(ReturnType, ExpressionList)
```

`ReturnType` represents the Prolog variable for the return type and can be a simple or an array reference. `ExpressionList` contains all Prolog variables from the return statement.

The return type of the method is set by a call to the `return_statement` rule for each return statement in a method. We ignore the return statements if a method returns a primitive type.

```
return a;       generates    return_statement([MRet], [[V_a]]),
return a.b.c;   generates    return_statement([MRet], [[V_a], [V_b], [V_c]]),
```

```
return a[0];      generates    return_statement([MRet], [[V_a], [V_a_El]]),
return a;         a is array   return_statement([MRet, MRet_El], [[V_a, V_a_El]]),
```

Method Call

```
method_call_parameters_left(FieldList, ParList, FormalParList, MethodPurity)
method_call_parameters_right(FieldList, ParList, FormalParList, MethodPurity)
```

These rule heads are designed for a method call on the left-hand and on the right-hand side of an assignment. `FieldList` contains the target, `ParList` contains all arguments, `FormalParList` contains all formal parameters and `MethodPurity` represents the method purity of the called method - it can be *pure* or *none*.

Each method call generates a call to the `method_call_parameters_right` or `method_call_parameters_left` rule, depending on where the method call appears. If the method call is on the right-hand side of an assignment or not in an assignment at all, we call the first one. Only when the method call is on the left-hand side of an assignment, we call the second one.

```
a.b.m(x, y);  generates    method_call_parameters_right([[V_a], [V_b]],
                                [[[V_x]],[[V_y]]],
                                [[MPar1], [MPar2]], MPurity),
```

It is possible, that field accesses appear in the argument of a method call. Field accesses are represented as a list of the involved Prolog variables. This converts the single argument in a list and turns the whole list of arguments, which is given to the rule, into a nested list.

```
b.m(a.f);     generates    method_call_parameters_right([[V_b]],
                                [[[V_a], [V_f]]],
                                [[MPar]], MPurity),
```

The formal return type of a method call is always used within an assignment or a field access. We generate the Prolog variable for the formal return type and integrate it in the appropriate list of the assignment or field access. We will present some examples for this case next.

Assignment

For each assignment we generate a call to the `assignment` rule:

```
assignment(LeftList, RightList)
```

`LeftList` contains all variables of the left-hand side of an assignment and `RightList` all variables on the right-hand side.

The left-hand side of the assignment is given as a list and the right-hand side as well. In the simplest case these lists consist of one element. This is in an assignment of two references. As soon as we have field or array accesses or method calls, we will have a longer list of Prolog variables for the assignment.

There exist the following kinds of assignments:

```
a = b;                  assignment([[V_a]],[[V_b]]),

C a = new C();          new([V_New1]),
                        assignment([[V_a]],[[V_New1]]),

a.f = b.g.h;            assignment([[V_a],[V_f]],[[V_b],[V_g],[V_h]]),

a = this.f;             assignment([[V_a]], [[this],[V_f]]),
```

```
a = this.m(e);            assignment([[V_a]], [[this],[MRet]]),
                          method_call_parameters_right([[this]],[[[V_e]]],
                                [[MPar]],MPurity),

a.f.m(e).g = b.m(e).h;    assignment([[V_a],[V_f],[MRet],[V_g]],
                          [[V_b],[MRet],[V_h]]),
                          method_call_parameters_left([[V_a],[V_f]],[[[V_e]]],
                                [[MPar]],MPurity),
                          method_call_parameters_right([[V_b]],[[[V_e]]],
                                [[MPar]],MPurity),

b = (new A()).m(e);       new([V_New1]),
                          assignment([[V_b]], [[V_New1],[MRet]]),
                          method_call_parameters_right([[V_New1]],[[[V_e]]],
                                [[MPar]],MPurity),

C[] a = new C[5];         new([V_New1,V_New1_El]),
                          assignment([[V_a,V_a_El]], [[V_New1,V_New1_El]]),

a[0] = b[4];              assignment([[V_a],[V_a_El]], [[V_b],[V_b_El]]),

a[0].f = b[1].m(e).g;     assignment([[V_a],[V_a_El],[V_f]],
                                [[V_b],[V_b_El],[MRet],[V_g]]),
                          method_call_parameters_right([[V_b], [V_b_El]],[[[V_e]]],
                                [[MPar]],MPurity),
```

Assignment with Primitive Types

Assignments with primitive types are handled a little bit differently. If we have an expression with primitive types, we might have method calls, field and array accesses as well. But additionally there might be other operations like division, multiplication, addition and more.

We show the generation of primitive assignments on the following example:

```
a[0].f = b[1].m(e).g + this.f;

assignment([[V_a],[V_a_El],[primitive]], [[primitive]]),
field_access_right([[V_b], [V_b_El], [MRet], [primitive]], [primitive]),
field_access_right([[this], [primitive]], [primitive]),
method_call_parameters_right([[V_b], [V_b_El]],[[[V_e]]], [[MPar]], MPurity),
```

For the left-hand side of the assignment we generate the normal call to the `assignment` rule, but ignore the right-hand side of the assignment and generate instead a simple primitive type. For each part of the right-hand side of the assignment we generate a call to the `field_access_right` rule, where the type of the field access is always primitive. For the primitive field, which is accessed, no Prolog variable name is written, but also the constant `primitive`.

Equality and Relational Operators

```
boolean_expression(LeftList, RightList)
```

This rule is used for boolean expressions like equality or relational operators. `LeftList` and `RightList` are the same as in the assignment rule.

For every equality or relational operator, which contains references, we have a call to the `boolean_expression` rule:

```
boolean_expression(LeftList,RightList)
```

The generation is done analogous to the generation of assignments.

```
n == m                  generates       boolean_expression([[V_n]],[[V_m]]),
n.next != m.next;       generates       boolean_expression([[V_n],[V_next]],
                                              [[V_m],[V_next]]),
```

Cast

```
cast_left(MethodModifier, CastType, Expression)
cast_right(MethodModifier, CastType, Expression)
```

These rules are used for a cast of an expression. `CastType` is the Universe type of the cast and `Expression` contains all variables of the expression to cast. `MethodModifier` is *static* or *none* and represents the modifier of the method.

Assume for this example that we are in a static method:

```
d = (C) b;              generates       cast_right(static, [V_C1], [[V_b]]),
                                        assignment([[V_d]], [[V_C1]]),
```

Assume for the next example that we are in a non-static method:

```
d = (C[]) b.f.g;        generates       cast_right(none, [V_C2, V_C2_El],
                                              [[V_b], [V_f], [V_g, V_g_El]]),
                                        assignment([[V_d, V_d_El]],
                                              [[V_C2, V_C2_El]]),
(C) d = (C) ((B) b).f;  generates       cast_left(none, [V_C3], [[V_d]]),
                                        cast_right(none, [V_B1], [[V_b]]),
                                        cast_right(none, [V_C4], [[V_B1], [V_f]]),
                                        assignment([[V_C3]], [[V_C4]]),
```

The first modifier of the rule designates, if the cast is in a static or non-static method.

Static Target

```
static_target(MethodModifier, [Type])
```

`MethodModifier` is *static* or *none* and represents the modifier of the method. `Type` represents the Prolog variable for a type of a static method call or field access.

For each type of a static target we generate a call to the `static_target` rule, where we generate a Prolog variable for the type of the static call and tell the rule if the static call is within a static method or not.

Assume for this example that we are in a static method:

```
c = T.f;            generates       static_target(static, [V_T1]),
                                    assignment([[V_c]], [[V_T1], [V_f]]),
```

Assume for the next example that we are in a non-static method:

```
c = T.m(a);         generates       static_target(none, [V_T2]),
                                    assignment([[V_c]], [[V_T2], [MRet]]),
                                    method_call_right([[V_T2]], [[[V_a]]],
                                    [[MPar]], none),
```

The first modifier of the `static_target` rule designates if the static call is in a static or non-static method.

Loops and Conditional

With all loops like for- and while-loops and conditionals like if statements we are proceeding the same way: we do not generate an additional constraint, but all assignment, equality and relational operator parts of the statement.

```
if(a.elem < b.elem)     generates     boolean_expression([[V_a], [V_elem]],
{                                           [[V_b], [V_elem]]),
     a.elem = b.elem;                 assignment([[V_a], [V_elem]],
}                                           [[V_b], [V_elem]]),
```

Composed Statements

We have the possibility to combine several expressions together:

```
Iter i = list1.first, Iter j = list2.first;
```

We just generate all rule calls for the single expressions within the composed statement.

```
local_variable_declaration(none, [V_i]),
assignment([[V_i]], [[V_list1], [V_first]]),
local_variable_declaration(none, [V_j]),
assignment([[V_j]], [[V_list2], [V_first]]),
```

Composed Assignments

In Java we have the possibility to compose assignments. We split up a composed assignment like this is done in the syntax tree and handle each assignment on its own, using the other side of the composed assignment we have done before.

```
a = b = c;        generates        assignment([[V_b]], [[V_c]]),
                                   assignment([[V_a]], [[V_b]]),

a = b.f = c;      generates        assignment([[V_b], [V_f]], [[V_c]]),
                                   assignment([[V_a]], [[V_b], [V_f]]),
```

Exceptions

The Exception object in a catch- or throw-clause is set to *readonly* by default, because an exception can be thrown in any context and the exception object is normally given to an arbitrary context. Exception objects do not have to be annotated and no call to a Prolog rule is generated for a throws, throw or catch statement. When an exception object is used in an expression, we just generate `[readonly]` instead of a Prolog variable for the exception object.

Strings

`Strings` are a special case of references in the Java language, because they can be initialized without an invocation of `new`:

```
String s = "Hello World";
```

In the Universe type system, `Strings` are always *readonly* and all methods of the `String` class are *pure*. Therefore we generate for each `String` the constant `[readonly]`. For `String` arrays the array elements are always *readonly*, but the array reference will generate a Prolog variable name like a normal array reference does.

In JDK 1.5 wrapper classes like `Integer` can also be initialized with directly assigning a value. It will be likely that all wrapper classes will behave the same like the `String` class in the future and will be always *readonly* with all class methods set to *pure*. Therefore we might generate more *readonly* constants for instances of wrapper classes in the future. By now all wrapper classes except for the `String` class are handled like normal references.

The query rule

We generate a query rule for each class of a program to annotate. The head of the query contains all variables used in the body. The body of the query rule is built by stringing all the generated constraints of a class together. When asking the query head to the Prolog system, all constraints in the query body are called. We want to remind you, that a constraint denotes a call to an existing Prolog rule of the Prolog database. The query rule is just used to combine all calls of one class.

We talk about the order of the rule calls in the query body later, when discussing heuristics and performance in section 3.2.

Method Purity

The purity of a method, *pure* or *none*, is given in an external XML file. When reading in the XML file in Java, we generate a Prolog variable for the purity of the method - have a look at section 2.4.3 - and save the method purity in a hashtable, where the Prolog variable is the key of the hashtable entry and the purity becomes the value. The generated Prolog variable name is unique and therefore used as the key.
Whenever we reach a method call or declaration while traversing the syntax tree, we generate the Prolog variable name for the called or declared method and look up the method's purity in the hashtable. The purity is directly written to the generated method call or declaration rule and the Prolog variable for the method is only used for looking up the purity in the hashtable. The variable itself is not used in the Prolog query.

Overridden Methods

The Universe type annotations of parameters and of the return type in overridden methods have to be the same as in the superclass. For parameters and the return type of overridden methods we generate not an own Prolog variable name, but the Prolog variable name of the superclass or interface. Like this we use the same annotation as in the superclass or interface, if the superclass or interface has to be annotated too. Otherwise it would be best to give the annotation of the superclass or interface as direct input.

2.4.2 Solver Database

In this section we go into more detail of the Prolog design and do not discuss just the head of the rules, but look a bit deeper in the body of a predefined rule. We describe how the body of the presented rules could be implemented; meaning what conditions should be fulfilled and present another few rules, which are used to model the Universe type system or help implement a part of a rule.

Simple Universe Type

Universe type rule

The Universe type system uses three modifiers: *rep*, *readonly* and *peer*.

- *rep* is used to indicate, that an object is in the inner context of another object, that it is owned by another object. A *rep* field for example is owned by the instance of the class the field is declared in. *rep* objects can only be changed by their owner and are therefore protected by their owner.
- The *peer* type denotes, that objects are in the same context. All references to objects in the same context are allowed to change their objects.
- The *readonly* modifier was designed to cross context boundaries. It is not allowed to change an object of another context, but still possible to read it.

Realization in Prolog

All used types are modeled as Prolog constants and become a possible solution for a question to the Prolog database. We created two kind of types: Universe types and normal types. But only the simple universe types, which are described next, are possible solutions of a generated query. The other types are used to model the Java type system, so every possible expression, which uses references, will work.

All type facts are only used within other Prolog rules and never generated out of the syntax tree. They are only used to model the Universe type system.

```
universe([rep]).
universe([readonly]).
universe([peer]).
```

Array Universe Type

Universe type rule

Array references are annotated with two Universe modifiers: one for the array reference and one for the array elements. Array elements can never be of type *rep*. Array elements of type *rep* would be owned by the array reference and could never be changed.

Realization in Prolog

```
universe([rep, peer]).
universe([rep, readonly]).
universe([readonly, readonly]).
universe([readonly, peer]).
universe([peer, readonly]).
universe([peer, peer]).
```

All Universe types are modeled as facts with the name `universe`. Arrays need two Universe types, if they contain references and the second type must not be *rep*.

this Reference

Universe type rule

The actual instance of a class is allowed to change its own representation. It is allowed to update all of its fields and to invoke all of its own methods.

- `v = this.f;`
  `[this] * [f] <: [v]`
- `this.f = e;`
  `[e] <: [this] * [f]`
- `v = this.m(e);`
  `[e] <: [this] * par(m), [this] * res(m) <: [v]`

The *this* object is always of type *peer*.

Realization in Prolog

```
type([this]).
```

The identifier `this` is used to indicate, that we are looking at a *this* object. We need this identifier, because there is a special combination rule for a field access over the *this* object - have a look at the discussion of the combination rule. The *this* reference can be only of type *peer* right now.

Types

Universe type rule

All Universe modifiers denote an extension to the type of the annotated reference.

```
S <: T <=> rep S <: rep T                    S and T are Java types
S <: T <=> peer S <: peer T
S <: T <=> readonly S <: readonly T
```

Realization in Prolog

```
type(X) :- universe(X).
type([primitive]).
type([null]).
```

Every Universe type is also a simple type. This is needed, because expressions like assignments can mix these two types, if we have for example a field access to a primitive type or assign the *null* reference to a reference.

These are also the two reasons, why other types like the Universe types are modeled at all. Primitive types are needed, because we can have an updating field access to a primitive type, where we have to assure, that the accessing object is not *readonly*.

The *null* constant is used, because we have to check assignments with *null* for the same reason:

```
a = null;
a.f = null;
```

The first assignment need not to be checked, but in the second assignment, we have a field access and so the accessing object must not be *readonly*. Therefore we check all assignments with references in it.

Like this the constant `null` and `primitive` are possible solutions of a Prolog query. They will not occur, because we call additional Prolog rules for references, which request all Prolog variables used in the query to be of a Universe type.

Subtypes

Universe type rule

```
rep <:  readonly
peer <:  readonly
null <:  rep
null <:  peer
```

The *readonly* Universe type is a super type of the other two. Like this we can cover any object of any context with a *readonly* reference and recover it only with a type cast.

Realization in Prolog

```
subtype_notequal([rep], [readonly]).
subtype_notequal([peer], [readonly]).
```

The array reference needs a special treatment, because we need to save two annotations for one reference. Therefore we connect the two corresponding Prolog variables in a Prolog list. There are similar subtype rules for array references like for example the following:

```
subtype_notequal([rep, readonly], [readonly, readonly]).
```

The *null* object is a subtype of all universe types. We have to add this subtype relation, because *null* is a reference and used in assignments, equalities and relational operators with references we have to annotate.

```
subtype_notequal([null], X) :- universe(X).
subtype_notequal([null], [this]).
```

Universe type `X` is subtype or equal to `Y`. This rule is used in the `assignment`, `boolean_expression`, `return_statement` and the `method_call_parameters_left` rule.

```
subtype_equal([X], [Y]) :- subtype_notequal(X, Y).
subtype_equal(X, X) :- type(X).
```

The *this* reference is of Universe type *peer* and has therefore to be equal to the simple type *peer* and is also a subtype of *readonly*. Because we designed the Universe type for *this* in a special way, we have to add the following relations:

```
subtype_notequal([this], [readonly]).
subtype_equal([this],[peer]).
subtype_equal([peer], [this]).
```

Combination

Universe type rule

The type combination of two Universe types follows the following rules:

*	peer	rep	readonly	boolean	int	null
peer	peer	readonly	readonly	boolean	int	null
rep	rep	readonly	readonly	boolean	int	null
readonly	readonly	readonly	readonly	boolean	int	null

The type combination is used in field accesses and method calls.

Realization in Prolog

`combination([X], [Y], [Z])` for example `combination([rep], [peer], [rep]).`

```
combination([X], [primitive], [primitive]) :- universe([X]).
combination([X], [A, A_El], [XA, A_El]) :- combination([X], [A], [XA]).
```

Universe type `X` combined with Universe type `Y` is of Universe type `Z`. This rule is used by the `field_access_right` and `field_access_left` rule. The combination of a Universe type with a primitive type is always a primitive type again. The combination of a simple Universe type with an array Universe type is an array Universe type again, where the combination is only between the references. The array elements keep their Universe type. Universe Type `XA` is the resulting type of the combination of `X` and `A`.

Normally the combination of *peer* and *rep* is *readonly*, but if the field access occurs over the *this* reference, it is *rep* again. This is the reason, why we created an own Universe type for the *this* reference. It is the only way to distinguish between a normal combination and the one with the *this* reference.

```
combination([this], [rep], [rep]).
```

This applies also for array references:

```
combination([this], [rep, peer], [rep, peer]).
combination([this], [rep, readonly], [rep, readonly]).
```

Field and Array Accesses

Universe type rule

Each field access is a combination of the types of the target and the field. Updating field accesses on the left-hand side of an assignment are only allowed on *rep* or *peer* references, because a *readonly* reference cannot change the representation of the object it points to. A *rep* field can only be changed by its owner. No updating field access on a *rep* field is allowed over another reference than *this*.

- `v = w.f;`
  `[w] * [f] <: [v]`
  `[f] =` *rep* `=> [w] =` *readonly*
- `v.f = e;`
  `[e] <: [v] * [f]`

`[v] != readonly, [f] != rep`

Realization in Prolog

```
field_access_right([H], Res) :- type(H), Res = H.
field_access_right([H1, H2|T], Res) :-
      type(H1), type(H2),
      combination(H1, H2, HRes),
      field_access_right([HRes|T], Res).
```

This rule is used for field and array accesses on the right-hand side of an assignment, or if they are not in an assignment at all. The rule executes a sequence of combinations - meaning recursive calls to the `combination` rule.
It is called in the `assignment`, `method_call_parameters_right`, `method_call_parameters_left`, `return_statement` and `split_right` rule.

```
field_access_left([H], Res) :- type(H), Res = H.
field_access_left([H1, H2|T], Res) :-
      type(H1), H1 \== [this], H1 \== [readonly],
      type(H2), H2 \== [rep],
      combination(H1, H2, HRes),
      field_access_left([HRes|T], Res).
field_access_left([[this], H2|T], Res) :-
      type(H2),
      combination([this], H2, HRes),
      field_access_left([HRes|T], Res).
```

This rule does the same on the left-hand side of an assignment, where a field or array access over a *readonly* reference is not allowed.

Field Declaration

Universe type rule

A *static* field is accessible for all instances of a class and cannot be owned by an object. Therefore *Static* fields cannot become *rep* in the Universe type system.

Realization in Prolog

```
field_declaration(static, [Field]) :- universe(Field), Field \== rep.
field_declaration(static, [FieldArray, FieldArrayEl]) :-
      universe([FieldArray, FieldArrayEl]), FieldArray \== rep.
field_declaration(none, Field) :- universe(Field).
```

Assigns a Universe type to a field, whereby static fields must not be of type *rep*.

Local Variable Declaration

Universe type rule

Static methods are not invoked on an object. Therefore they cannot be owned by an object and cannot contain any *rep* object.

Realization in Prolog

```
local_variable_declaration(static, [Local]) :- universe([Local]), Local \== rep.
local_variable_declaration(static, [LocalArray, LocalArrayEl]) :- ...
local_variable_declaration(none, Local) :- ...
```

This rule assigns a Universe type to a local variable, whereby local variables of static methods must not be of type *rep*.

Method and Parameter Declaration

Universe type rule

- *Pure* methods are not allowed to change the representation of the target. All parameters have to be *readonly*. The return type can be of any Universe type, because we are still allowed to return a newly created object by a *pure* method.
- Public methods are not liked to contain *rep* parameters. They are written to get information from another context.

Realization in Prolog

We distinguish between pure, static and public methods. Static methods must not have return types of type *rep*. The return type of a pure method can be of all of the three types. We could for example create a new object and return it. But new objects can only be created by *pure* constructors.

```
method_declaration(pure, static, [ReturnType]) :-
      universe([ReturnType]), ReturnType \== rep.
method_declaration(pure, public, [ReturnType]) :-
      universe([ReturnType]), ReturnType \== rep.
method_declaration(pure, none, [ReturnType]) :-
      universe([ReturnType]).
method_declaration(none, static, [ReturnType]) :- ...
method_declaration(none, public, [ReturnType]) :- ...
method_declaration(none, none, [ReturnType]) :- ...
```

There are similar rules for array return types of public or static methods.

```
method_declaration(MethodPurity, MethodModifier,
     [ReturnTypeArray, ReturnTypeArrayEl]) :- ...
```

If a method has no return type or returns a primitive type, we do not use this rule and use only the `parameter_declaration` rule.

Parameters of *pure* methods can only be readonly and parameters of static or public methods must not be *rep*, therefore we have to distinguish parameters of pure, static or public methods.

```
parameter_declaration(pure, _, [Parameter]) :- Parameter = readonly.
parameter_declaration(pure, _, [ParameterArray, ParameterArrayEl]) :-
      ParameterArray = readonly, universe([ParameterArray, ParameterArrayEl]).
parameter_declaration(none, static, [Parameter]) :- ...
parameter_declaration(none, public, [Parameter]) :- ...
```

```
method_parameter_declaration(none, none, Parameter) :- universe(Parameter).
```

The rules for parameters of static or public methods are the same as the rules for return types. There are similar rules for array parameters of static or public methods.

```
parameter_declaration(MethodPurity, MethodModifier,
      [ParameterArray, ParameterArrayEl]) :- ...
```

Object Creation

Universe type rule

`v = new TS();` `TS <: [v], TS != readonly`

Whenever we create a new object, we must specify in which context the object has to be allocated. It is not allowed to create an object of type *readonly*, otherwise we would never be allowed to write on this object.

Realization in Prolog

We wrote a rule for the creation of references, because newly created references cannot be of type *readonly*.

```
new(static, [Variable]) :-
      universe([Variable]), Variable \== readonly, Variable \== rep.
new(none, [Variable]) :-
      universe([Variable]), Variable \== readonly.
```

The creation of arrays needs a similar rule.

```
new(static, [Array,Array_El]) :-
      universe([Array, ArrayEl]), Array \== readonly, Array \== rep.
new(none, [Array,Array_El]) :-
      universe([Array, ArrayEl]), Array \== readonly.
```

Return Statement

Universe type rule

`[return statement] <: [return type]`

The resulting type of the return statement has to be subtype or equal to the return type.

Realization in Prolog

For return statements the `return_statement` rule is used. It evaluates `ExpressionList`, which is a subtype of the return type of the method. `ReturnType` can be a simple Universe type or an array Universe type.

```
return_statement(ReturnType, ExpressionList) :-
      field_access_right(ExpressionList, ExpressionType),
      subtype_equal(ExpressionType, ReturnType).
```

Because we are only generating constraints for reference return types, a return expression can only be composed of a single reference, a field or array access, a method call or a combination of the last three. These are always represented as a list in our design and therefore the return expression is always a list too, but maybe with only one element.

Method Call

Universe type rule

```
v = w.m(e);
[e] <:  [w] * par(m), [w] * res(m) <:  [v]
```

We have to look at the resulting type of the combination with the target.

- m is not a *pure* method
  `par(m) != rep, res(m) != rep, [w] != readonly`
- m is a *pure* method
  `res(m) = rep => [w] = readonly`
- m invoked on *this*
  no restrictions

Realization in Prolog

```
method_call_parameters_left(FieldList, ParList, FormalParList, none) :-
      field_access_left(FieldList, FieldType),
      split_left(ParList, ParameterList),
      parameters(FieldType, ParameterList, MethParList, none).
method_call_parameters_left(FieldList, ParList, FormalParList, pure) :-
      field_access_left(FieldList, FieldType),
      FieldType \== [readonly],
      split_left(ParList, ParameterList),
      parameters(FieldType, ParameterList, MethParList, pure).
```

The rule for method calls on the right-hand side of an assignment works analogous.

```
method_call_parameters_right(FieldList, ParList, FormalParList, none) :- ...
method_call_parameters_right(FieldList, ParList, FormalParList, pure) :- ...
```

These two rules combine the type of the parameters for a method call on the left-hand side or the right-hand side of an assignment. The type of the target given in `FieldList` is combined with the type of each formal parameter. The type of the argument has to be a subtype of the result of the combination.

The `method_call_parameters_left` and `method_call_parameters_right` rules are the other two rules, which require the purity of a method beside the `method_declaration` and `parameter_declaration` rules. They need to know the purity of the called method, because on *readonly* references it is only allowed to call *pure* methods.

These rules only build the universe types for the parameters of a called method. The return type is implicitly built within an assignment (look at section 2.4.1). If the method call is not in an assignment, the method has no return type or it is not used. In either case we just ignore the return type.

The two rules for a method call use several internal rules to fulfill the conditions stated above:

- `split_left(ArgumentList, ResultList)`
  This rule splits up a list of arguments of a method call on the left-hand side of an assignment. The rule checks, if each argument is again a list, means the given argument is a field access. The rule builds the combination of the types and appends it to the resulting argument list. `ResultList` is a single list, whereby `ArgumentList` is a nested list. The rule is called by the rule `method_call_parameters_left`.

- `split_right(ArgumentList, FormalParameterList, ResultList)`
  This rule does the same thing as the `split_left` rule but within a method call on the right-hand side of an assignment. It is called within the `method_call_parameters_right` rule.

- `parameters(FieldType, ParList, FormalParList, MethodPurity)`
  This rule builds up the Universe types of the arguments. `FieldType` is the Universe type of the target. The rule is called by the `method_call_parameters_left` and `method_call_parameters_right` rule.

Assignment

Universe type rule

```
v = w;        [w] <:  [v]
```

The right-hand side of an assignment is subtype or equal to the left-hand side.

Realization in Prolog

```
assignment(LeftList, RightList) :-
      field_access_left(LeftList, LeftRes),
      field_access_right(RightList, RightRes),
      subtype_equal(RightRes, LeftRes).
```

Field and array accesses on the left-hand side are written in `LeftList` and field and array accesses on the right-hand side in `RightList`. The rule executes the field and array accesses on both sides and assigns the resulting subtype of the right-hand side to the left-hand side.

Equality and Relational Operators

Universe type rule

```
v == w;        [w] <:  [v] or [v] <:  [w]
```

One side has to be subtype or equal to the other side. It does not matter, on which side the subtype is written.

Realization in Prolog

```
boolean_expression(LeftList, RightList) :-
      field_access_left(LeftList, LeftRes),
      field_access_right(RightList, RightRes),
      subtype_equal(RightRes, LeftRes).
boolean_expression(LeftList, RightList) :-
      field_access_left(LeftList, LeftRes),
      field_access_right(RightList, RightRes),
```

```
      subtype_notequal(LeftRes, RightRes).
```

This rule is used for boolean expressions like equality or relational operators and works quite similar like the assignment rule. But the resulting type of the right-hand side can be a subtype of the left-hand side and the other way around.

Cast

Universe type rule

```
v = (TS) e;        TS <:  [v], TS <:  [e]
```

The type of a cast must be subtype or equal to the resulting type of the expression to cast.

Realization in Prolog

In static methods the usage of *rep* is not allowed and therefore also no cast to *rep* is allowed. We denote for each cast, if it is used in a static or non-static method.

```
cast_left(static, [CastType], Expression) :-
      universe([CastType]), CastType \== rep,
      field_access_left(Expression, ExpressionType),
      subtype_notequal([CastType], ExpressionType).
cast_left(none, [CastType], Expression) :- ...
cast_right(static, [CastType], Expression) :- ...
cast_right(none, [CastType], Expression) :- ...
```

We can have a cast of an expression on the left-hand or the right-hand side of an assignment. Field and array accesses of the cast expression are written in `Expression` as a list. `CastType` is either the same as, a subtype or supertype of the resulting type of `Expression`, because we can cast an expression. The three cases are tried out in exactly this order.

There exists similar rules for array cast types.

```
cast_left(none, [CastType, CastTypeEl], Expression) :- ...
cast_right(static, [CastType, CastTypeEl], Expression) :- ...
```

Static Target

Universe type rule

A static target can be of all of the three Universe types.

Realization in Prolog

```
static_target(static, [Type]) :- universe([Type]), Type \== rep.
static_call_target(none, [Type]) :- universe([Type]).
```

The Prolog variable for static calls is only used in expressions, but it needs an additional rule call to assure, that in static methods no static target becomes ever *rep*.

2.4.3 Prolog Variable Naming

Each Prolog variable represents a Universe type. If we call a Prolog rule, we receive for each Prolog variable the Universe type, for which the rule is satisfied.

The Prolog rules are used to design the type relations in a program. Each variable of the original Java program is mapped to a Prolog variable with a unique name. We use some naming rules for variables to attain this. But there are some limits set by the Prolog system itself. The following rules must be fulfilled:

- names of files must start with a small letter
- names of variables must start with a capital letter
- names of constants must start with a small letter
- names of rules must start with a small letter

There are always ways to get a name unique, but names should not get too long because of clearness. So the names have to be:

- unique
- resembling the original variable name
- as short as possible

Naming of Variables

Variable names are not global, they are only unique in their scope of visibility. In Prolog this scope is different. Variables are only visible in one single rule, so you can reuse the same variable name for as much rules as you like. But how can we draw the dependencies we need? If we always name the same variable in a constraint rule the same way, this will have no effect in Prolog. In Prolog it is a different variable in every rule, although it has the same name. But the naming might be a good idea for ourselves and in the automatic generation of constraints. If we use the same name for a variable in every constraint, we can reuse this name in the finding of the solution - the head of a query.

Now we will have a look at the generation of variable names and how to make them unique. The uniqueness of a name can be achieved if we add each scope name to the single variable name. This results in the following rules for the naming of variables:

```
Variable    := Prefix Binder Packagename Binder Classname Binder ClassTail.
Binder      := '_'.
Prefix      := 'L'.
Packagename := name of the package, replace '.' with Binder.
Classname   := name of the class.
ClassTail   := Fieldname | Newname | Typename | Castname
            | Methodname [Binder MethodTail].
Fieldname   := name of the field [Arrayname].
Methodname  := name of the method.
MethodTail  := "Return" [Arrayname] | "FormalParameter" Binder Number [Arrayname]
            | Typename | Castname | Varname.
Varname     := "Local" Binder Localname Number [Arrayname] | Newname.
Localname   := name of the local variable.
Newname     := "New" Number [Arrayname].
Typename    := "Type" Number.
Castname    := "Cast" Number [Arrayname].
Arrayname   := Binder "El".
Number      := 1 | 2 | 3 | ...
```

The naming of arrays is a bit special. When declaring an array, we have to give two Universe types: one for the array reference and one for the array elements. Because of this we always need two variables for each array.

Now we have to look at the limits the Prolog system sets us. As we mentioned, variable names have to start with a capital letter. Normally class names in Java also start with a capital letter, because this is like an implicit naming rule under programmers. But - and this is important - they do not need to! It would not be nice just to assume, that every programmer does so. There are two solutions for this problem: One can turn the first letter in a capital letter for every class or one can add another identifier to the beginning of the variable name. We add the Prefix 'L' to all our names, so all variable names always start with a capital letter.

We get the following kinds of variable names with this rules:

- **L_Packagename_Classname_Fieldname**
  variables for class fields
- **L_Packagename_Classname_Fieldname_El**
  variables for elements of array class fields
- **L_Packagename_Classname_Methodname**
  variables for the methods purity which can only be *pure* or *none*
- **L_Packagename_Classname_Methodname_Return**
  variables for the return type of methods
- **L_Packagename_Classname_Methodname_Return_El**
  variables for the related array elements of the return type of methods
- **L_Packagename_Classname_Methodname_FormalParameter1**
  variables for parameters of a method
- **L_Packagename_Classname_Methodname_FormalParameter1_El**
  variables for the related array elements of the array reference parameter
- **L_Packagename_Classname_Methodname_Type1**
  variables for types in static method calls or static field accesses
- **L_Packagename_Classname_Methodname_Cast1**
  variables for cast expressions
- **L_Packagename_Classname_Methodname_Cast1_El**
  variables for the related array elements of cast expressions
- **L_Packagename_Classname_Methodname_Local_Varname1**
  variables for local variables of a method
- **L_Packagename_Classname_Methodname_Local_Varname1_El**
  variables for local array elements
- **L_Packagename_Classname_Methodname_New1**
  variables for the type before a new expression when a local object is created
- **L_Packagename_Classname_Methodname_New1_El**
  additional variables for the type before a new expression when a local array is created

Conflicts in Naming

Method Overloading

The naming conflict for variables is solved by the naming rules above. But this could still generated the same name for two different variables because of method overloading. It is allowed

in Java to have two or more methods with the same name; they just have to differ from each other by the number or type of the parameters. To avoid the naming conflicts given by overloading, we can add the differentiating factor of the single methods. The drawback of this is of course, that the names get long again. We just could number the single method names, but this would decrease the clarity of the names. So we decided for the longer ones:

```
Methodname   := Returntype Binder MName TypeOfParams.
MName        := name of the method.
TypeOfParams := {Binder Type}.
Type         := type of the parameter (package and class name).
```

We always have to apply these longer names, because method overloading could occur in subclasses and we do not know if there exists method overloading in a subclass when we compile a class.

Inner Classes

Another conflict with Naming may occur when using nested classes. We just add all class names to the Prolog variable name to get a unique name again:

```
Variable     := Prefix Binder Packagename Binder Classes Binder ClassTail.
Classes      := Classname {Binder Classname}
Classname    := name of the class.
```

For variables of a nested class the Prolog names just contain two or more class names.

Uniqueness

The generation of Prolog variable names for each variable leads to the following problem:

Variables with the same name but in different blocks generate the same Prolog variable name.

We can avoid this by adding a block counter to every generated Prolog name for local variables. But then we would have to change the XML annotation format. We must be able to generate a Prolog variable name for variables, which are annotated in an XML input file.
Another possibility is to change the definition of the index of a variable. In the syntax tree the index of the two `iterator` variables in the example is the same, because they are in blocks of the same level.

```
public String foo(Vector v, Vector u) {
        for( Iterator iterator = v.iterator(); iterator.hasNext(); ) {
                Object o = iterator.next();
                //do something with variable o
        }
        for( Iterator iterator = u.iterator(); iterator.hasNext(); ) {
                iterator.next().remove();
        }
}
```

The two `iterator` variables in the example generate the same Prolog variable name, although they are two independent variables. This will give the generated Prolog variable more restrictions, which is not correct.

Programmers should always use a new name for each variable, to avoid this problem. If you use unique names in your program, unique Prolog variable names will always be generated. Later this case may be handled by more detailed XML support.

2.4.4 Conflicts

In simple programs without *pure* methods, there is always a solution without conflicts. In the worst case, there is just one solution and all variables are mapped to *peer*. The search for solutions does terminate and has at least one solution. Unfortunately mapping all variables to *peer* is like using no annotations at all.

As soon as we are using *pure* methods and partly annotated programs, we have to look at conflicts, because then it can happen, that an assignment or method call will not work anymore.

There are two possibilities to trace Universe type conflicts of the original source:

1. The user adds casts to all points in the original source, where a conflict may occur
2. The type inference part traces conflicts automatically

If the user adds no casts to the original source and the type inference part traces no conflicts, there are only those annotation solutions found, which work out for all type relations. The found annotation solution is always compilable and runnable.

The user can control conflicts by adding casts to the original source. We have three possibilities to annotate a cast, which was inserted by the user:

1. cast to the resulting type of the expression to cast
   This is like we would not add a cast at all.
2. cast to a super type of the expression to cast
3. cast to a subtype of the expression to cast

If the user adds a cast to an expression, he allows more possibilities to annotate this expression. Like this the user is concerned about possible Universe type conflicts in his program. But we also want to automate this step. We always get better solutions, if the user adds some hints into his program code, by adding some annotations, that are desired and adding some casts, where conflicts should be allowed. But we must also allow the static inference tool to trace conflicts automatically. This might be useful to users, who want to analyze a program, they have not written themselves or to compare annotations found automatically with annotations found manually.

A conflict occurs for example in one of the following situations:

```
/*@ pure @*/ void foo(T b) {
    /*@ rep @*/ T a;
    a = b;
}
```

`b` gets *readonly*, because parameters of *pure* methods have to be *readonly*
`a` is already annotated by the user with *rep*
try to assign *readonly* to *rep* => **conflict**

```
/*@ readonly @*/ T a;
T b = a;
b.m();
```

`a` is annotated by the user with *readonly*
`b` gets *readonly*, because `a` is *readonly*
if method `m` is not *pure* => **conflict**

A cast can be allowed in these situations, but we do not know, if it will actually work at runtime. The compiler allows a cast in such situations and prints a warning, that the cast may fail at runtime. There is a runtime error message, if the cast does not work.

Partly annotated Programs

A program, which uses external classes, has three possibilities to use them in the type inference step:

1. *Annotate the external classes together with the program*
   We annotate all classes - the classes of the program and the external classes - with our architecture. We generate constraints for the program and the external classes and call all queries within the global query. Like this all variables are annotated by our architecture, also the variables of the external classes.

2. *Use the already found annotation solution for the external classes*
   In the second case, we read in the found solution of the external classes and use them as fixed values for the generation of the constraints of the program to annotate. All annotations are stored in a hashtable with their respective Prolog variable name.
   Whenever we reach a variable of the external class in the traversal of the syntax tree, we look up the already given annotation of this variable and write it to the current constraint. The Prolog variable is then only used for looking up the annotation and not used anymore in the constraints. No query is generated for the external classes.

3. *Use a part of the found annotation solution for the external classes*
   In this case we proceed in the same way as in the second case, but we generate a query of each of the external classes. The queries of the external classes contain some constraints with no variables, but only fixed annotation values. Like this the already given annotation solution is verified once more.

The user also has the possibility to add annotations directly into the code of his program. During the traversal of the syntax tree, when generating the constraints, the inserted annotations are visited and also entered into the hashtable with the appropriate Prolog name for the variable. The Prolog variable name is then also not used in the constraints, but directly the given annotation.

In the following example the user annotated variable `o` with *readonly*, because he only wants to allow a read access over this reference. The annotation of the variable is stored in the hashtable and directly written to the generated `assignment` and `local_variable_declaration` constraints.

`/*@ readonly @*/ Object o = a.b;`	the user annotated variable `o`
`assignment([[readonly]], [[V_a], [V_b]]),`	no Prolog variable name is used for `o` in the generated constraints
`local_variable_declaration(none, [readonly]),`	the declaration rule is needed, because the user might have annotated wrongly

Additional temporary Universe Types

Conflicts are integrated in the Prolog part and can be solved by an additional cast. We implement conflicts in an additional Prolog program. If our query does not return any solution, we have to apply casts to get a solution. To achieve this, we extend our Prolog type facts by two new type facts:

```
universe([conflict_rep]).
universe([conflict_peer]).
```

These new types correspond to the Universe type *readonly*, but will be handled by the Prolog program like *rep* and *peer*. Updating field accesses and method calls of non-pure methods are allowed on them.

The constant `conflict_rep` in the annotation solution found by the Prolog system is used to show, that we need a cast to *rep* somewhere. We have to traverse the syntax tree again and check each expression, where a variable was annotated by Prolog with one of the additional conflict types. Whenever we reach a variable, which was annotated by the Prolog system with `conflict_rep` or `conflict_peer`, we write the annotation *readonly* to the output for this variable. We write an additional cast to the annotation output for each expression the variable is used in if necessary.

The new type `conflict_rep` is equal to *rep* and *readonly*, `conflict_peer` to *peer* and *readonly*. Whenever we assign `conflict_rep` to *rep* and `conflict_peer` to *peer*, we are doing a cast. The

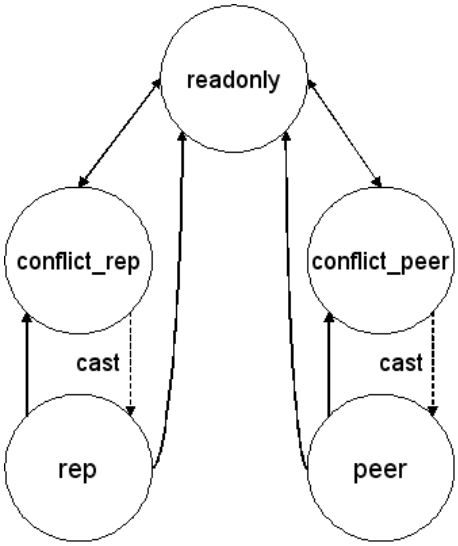

Figure 2.4: Type relations of the new types

type `conflict_rep` behaves like *rep* in the combination and `conflict_peer` like *peer*. Like this all assignments, field and array accesses and method calls can still work.

```
subtype_equal([conflict_rep], [rep]).
subtype_equal([conflict_rep], [readonly]).
subtype_equal([conflict_peer], [readonly]).
subtype_equal([conflict_peer], [peer]).
subtype_equal([conflict_peer], [this]).
subtype_equal([rep], [conflict_rep]).
subtype_equal([readonly], [conflict_rep]).
subtype_equal([readonly], [conflict_peer]).
subtype_equal([peer], [conflict_peer]).
subtype_equal([this], [conflict_peer]).
```

The combination rules do not change much. A combination with *readonly* still results in *readonly*. If the result of a combination would normally become *rep* or *peer*, it will still do so.

```
combination([conflict_rep], [rep], [readonly]).
combination([conflict_peer], [rep], [readonly]).
combination([conflict_rep], [readonly], [readonly]).
combination([conflict_peer], [readonly], [readonly]).
combination([conflict_rep], [peer], [rep]).
combination([conflict_peer], [peer], [peer]).
combination([conflict_rep], [conflict_rep], [readonly]).
combination([conflict_rep], [conflict_peer], [rep]).
combination([conflict_peer], [conflict_rep], [readonly]).
combination([conflict_peer], [conflict_peer], [peer]).
combination([conflict_rep], [primitive], [primitive]).
combination([conflict_peer], [primitive], [primitive]).
```

In order that fields can also become `conflict_rep` or `conflict_peer` the following rules were added to the Prolog program.

```
combination([rep], [conflict_rep], [readonly]).
combination([rep], [conflict_peer], [rep]).
combination([readonly], [conflict_rep], [readonly]).
combination([readonly], [conflict_peer], [readonly]).
combination([peer], [conflict_rep], [readonly]).
```

```
combination([peer], [conflict_peer], [peer]).
combination([this], [conflict_rep], [rep]).
combination([this], [conflict_peer], [peer]).
```

Because of the arrangement of these facts in the file they are saved, the solution `conflict_rep` or `conflict_peer` is only used, when all other universe types do not work out. The Prolog system tries out these types as a solution before backtracking. When a call to a Prolog rule does not work out for any of the Universe types, the Prolog system tries out `conflict_rep` and `conflict_peer` and returns it as one possible solution for the respective Prolog variable. If we force the Prolog system to backtrack, another solution with none of these two types can be found, if there is one. The facts with the constant `conflict_rep` and `conflict_peer` have to be the last in the chain of the other facts, so they will be the last possible solution to try before backtracking.

We have to add similar `universe`, `subtype_equal` and `combination` rules for array types to become `conflict_rep` or `conflict_peer`. Have a look at section B.1 in the appendix to see the additional rules for array types.

There are some more restrictions, which have to be added to some rules of the Prolog program:

1. parameters of pure methods can become `conflict_rep` or `conflict_peer`.

2. variables of static methods cannot be of type `conflict_rep`, because we are not allowed to cast a variable to *rep* in a static method.

3. There is no need for types in a static target to get `conflict_rep` or `conflict_peer`, because we would never cast a static target. Casts and newly created objects cannot get `conflict_rep` or `conflict_peer` for the same reason.

We show on the following example, how conflicts are traced and inserted. Variable `a` is already annotated by the user with *rep* and field `f` with *peer*. The two parameters have to become *readonly* in the annotation solution written to the output, because they are parameters of a *pure* method.

```
/*@ pure @*/ void foo(T b, T c) {          b and c get readonly
    /*@ rep @*/ T a;                       a is already annotated by the user with rep
    a = b;                                 try to assign readonly to rep => conflict
    a = b.f;                               f is annotated by the user with type peer
    b = c; }                               c is readonly
```

In this example, the Prolog system tries variable `b` to become *readonly*, because it is the parameter of a *pure* method. This will not work out, because of the first assignment to the *rep* variable `a`. We are not allowed to assign a variable of type *readonly* to a variable of type *rep*. Therefore the only annotation solution the Prolog system finds for variable `b` is `conflict_rep` and we need to add a cast to the first assignment in the method. Let us look at the fully annotated example:

```
/*@ pure @*/ void foo(/*@ readonly @*/ T b, /*@ readonly @*/ T c) {
    /*@ rep @*/ T a;
    a = (/*@ rep @*/ T) b;
    a = ((/*@ rep @*/ T) b).f;
    b = c; }
```

In the second assignment we also need a cast, because we have a variable of type `conflict_rep` used in a combination on the right-hand side and the resulting type of the left-hand side is *rep*. In the last assignment no cast is needed, because we are assigning a variable of type *readonly* to a variable of type `conflict_peer`.

We show at which point a cast has to be inserted on the following examples:

Code	Annotation found by Prolog	Annotation written to the output	Annotation inserted in the code
`a = b;`	`V_a=peer, V_b=conflict_peer`	`[a]=peer, [b]=readonly;`	`a = (/*@ peer @*/ T) b;`
`a = b.f;`	`V_a=peer, V_b=conflict_peer, V_f=peer`	`[a]=peer, [b]=readonly, [f]=peer`	`a = ((/*@ peer @*/ T) b).f;`
`c = d.e;`	`V_c=rep, V_d=rep, V_e=conflict_peer`	`[c]=rep, [d]=rep, [e]=readonly`	`c = (/*@ rep @*/ T) d.e;`
`a.h = b.h;`	`V_a=peer, V_b=conflict_peer, V_h=readonly`	`[a]=peer, [b]=readonly, [h]=readonly`	`a.h = b.h;`
`a.h = b;`	`V_a=peer, V_b=conflict_peer, V_h=readonly`	`[a]=peer, [b]=readonly, [h]=readonly`	`a.h = b;`

Variable `b` needs to be casted in the first two assignments to *peer*, because it is assigned to *peer*. Whenever a field is annotated with `conflict_rep` or `conflict_peer` by Prolog, we have to write a cast to the resulting type of the combination as this is done for the field `f`.
In the last two examples we do not have a conflict, because the right-hand side is already a subtype of the left-hand side and we need not to cast variable `b` this time.

Drawback

The given annotations in a partly annotated program are definite values in the Prolog part. Therefore we cannot find casts of already annotated variables. If a variable is annotated as *readonly*, it will stay readonly in the whole Prolog program and can never be changed into `conflict_rep` or `conflict_peer`.

Let us assume for the next example, that the variable `annotated` is annotated by the user with *readonly* and that the variable `free` is annotated by the static inference tool with *peer*, because it needs to call a non-pure method somewhere.

```
readonly A annotated = new peer A();
peer A free = new peer A();
free = free.changeObject();
free = annotated;
```

Although the last assignment is legal, if we add a cast to `annotated`, the Prolog program will not find it. For the Prolog program the variable `annotated` is always *readonly* and can never get `conflict_peer`, which would indicate a cast.

But the user can add a cast by himself to the `annotation` variable in this expression and then the solver will find a solution with the type *peer* of the cast in this expression.

```
free = (A) annotated;
```

If we want to allow a conflict on an already annotated variable, we have to mark the conflict with a cast in an expression by hand. The static inference tool cannot find out a conflict for an already annotated variable yet.

Resolution of the temporary Types

The user can decide, if he wants the static inference tool to trace conflicts automatically by selecting the Prolog database, which implements the two additional conflict types. By traversing

the syntax tree again we can resolve each conflict, the Prolog system finds and write an additional cast to the annotation output.

We designed a new cast tag for our XML annotation format to insert an additional cast:

```
<addcast line="8" type="Object" modifier="rep" identifier="element" index="0"/>
8:  ...  = ...  (/*@ rep @*/ Object) element ...;
```

We have to add a cast to the expression on line 8, where we cast the first occurrence of variable `element` to the type `/*@ rep @*/ Object`. The index designates the occurrence of the variable in the expression. If there is more than one usage of the variable in an expression, we can figure out with the index of the variable, which occurrence of the variable needs a cast.

After receiving a solution with conflicts, we have to traverse the syntax tree again and check all expressions, which contain variables of type `conflict_rep` and `conflict_peer`. For each of the following expressions we have to check if a cast is necessary and if so, write one to the XML solution file.

1. **variable initialization**
   `Node n = list.first.getNewNode().next;`
   When we declare a *rep* or *peer* variable, it is possible, that the initialization is a conflict, because one of the variables used within is of type `conflict_rep` or `conflict_peer`.

2. **assignment**
   `n.prev = list.first.getNewNode().next;`

   - *Updating field access on the left-hand side*
     If we have an updating field access over a variable of type `conflict_rep` or `conflict_peer`, we also need a cast to have a valid annotation. This is the case if any object, which accesses a field on the left-hand side of an assignment is of one of the conflict types. The last field of the chain of field accesses is allowed to be *readonly*, `conflict_rep` or `conflict_peer`.
   - *right-hand side is super type of left-hand side*
     Whenever the right-hand side of an assignment contains a variable of a conflict type, the resulting type of the right-hand side would become *readonly*. If the left-hand side of the assignment is of type *rep* or *peer*, we need to cast each variable of type `conflict_rep` or `conflict_peer` in the right-hand side.

3. **return statement**
   `return list.first.getNewNode().next;`
   A return statement is like a variable initialization. If the return type is neither *readonly*, `conflict_rep` nor `conflict_peer` we need to cast each `conflict_rep` and `conflict_peer` variable in the return statement.

4. **method call**
   `list.first.setNode(n);`

   - *target object*
     The target of a non-*pure* method is not allowed to be *readonly*, because a *readonly* reference can only access *pure* methods. Therefore we must cast each `conflict_rep` or `conflict_peer` reference in the target of a call to a non-*pure* method.
   - *formal parameter*
     The type of the argument has to be a subtype of the combination of the type of the target and the formal parameter. If the combination of the target and the formal parameter results in type *peer* or *rep*, we must cast an argument of type `conflict_rep` or `conflict_peer`.
     When the argument and the formal parameter are `conflict_rep` or `conflict_peer`, there is no problem, because the formal parameter will become *readonly* and a combination with the type of the target and *readonly* results in *readonly* again.
     `conflict_rep <:  [target] * conflict_rep`

5. **constructor call in object creation**
   `Node n = new Node(list.first);`
   We have to check all `conflict_rep` or `conflict_peer` objects of the argument, if the combination of the Universe type of the newly created object and the Universe type of the formal parameter is *rep* or *peer*.

6. **cast**
   `Element e = (Element) list.first.getElement((Element) n.elem);`
   The expression of a cast cannot contain any variables of type `conflict_rep` or `conflict_peer`. The cast already casts the expression to a type, that is useful for the expression the cast is in. Therefore we do not need to look at cast expressions.

7. **instanceof**
   `n.elem instanceof Elem`
   We do not do anything for the instanceof expression. Therefore we always look at the instance of the static type of a variable or expression.

To get the resulting type of each part of an expression, we have to combine the Universe types of the single parts of the expression as we have done in the Prolog part. We combine the Universe types of each part of the following expressions:

- **field access**
  `[object] * [field]`
- **array access**
  `[array reference] * [array element]`
- **method call**
  `[target] * [formal parameter]`
  `[target] * [return type]`
- **constructor call in object creation**
  `[object creation type] * [formal parameter]`

For each expression we store all variable and method names in a map with their respective occurrence in the expression. Whenever we reach a variable of type `conflict_rep` or `conflict_peer`, we check, if an additional cast is needed for the expression and write one to the output. We save the actual Universe type of the subpart of the expression, we are in and write a cast to the combination of this type with the type of the variable.

We show how this works on the following two examples. The first example is an assignment with a chain of field accesses on the right-hand side. The type relations of the Universe types of all variables is shown next. We assumed a possible annotation solution found by the Prolog system for the assignment, where variable `a` and `b` become *rep*, `c` and `d` are `conflict_peer` and variable `e` is *peer*. Whenever we have a combination with `conflict_rep` or `conflict_peer`, we need to write a cast to the resulting type of the combination. Therefore we have to cast the field access `b.c` and `b.c.d` to *rep* in the first example. These two casts are denoted with an additional cast tag for the two variables `c` and `d`. At the end the original expression with the inserted casts is shown.

1.
```
a = b.c.d.e;
[a] :> [b] * [c] * [d] * [e]
rep :> rep * conflict_peer * conflict_peer * peer
rep :> rep * [c] * [d] * [e]
rep :> rep * conflict_peer * [d] * [e]
<addcast line="0" type="A" modifier="rep" variable="c" index="0"/>
rep :> rep * [d] * [e]
rep :> rep * conflict_peer * [e]
<addcast line="0" type="A" modifier="rep" variable="d" index="0"/>
rep :> rep * [e]
rep :> rep * peer
a = ((/*@ rep @*/ A) ((/*@ rep @*/ A) b.c).d).e;
```

2.
```
b.m(c.d);
[b] * [formal parameter of m] :> [c] * [d]
rep * peer :> conflict_rep * peer
rep :> conflict_rep * [d]
<addcast line="1" type="A" modifier="rep" variable="c" index="0"/>
rep :> rep * peer
b.m(((/*@ rep @*/ A) c).d);
```

In the second example, we violate the subtype relation with the type of the argument and need to cast variable `c` to *rep*.

Universe types of formal parameters and return types of methods, which are not part of the program to annotate, must be given with the annotation input to the static inference tool. We cannot determine the Universe types of methods of external classes. They can only be used by the actual program. Therefore they must be given with the annotation input and cannot become `conflict_rep` or `conflict_peer`, because they denote fixed values to the Prolog part of the tool.

2.4.5 Optimizations

As we explained in the Prolog introduction in 1.3, the order of rules is important. The right order of the rules in a query may save a lot of backtracking and is therefore a gain in performance. The more backtracking is needed to find a possible solution, the more time the search for a solution needs. We have two points in the static inference tool, where we have to decide about the order of rules.

1. **Order of the rule calls in a query body**
   Different order of the rule calls changes the order of the possible solutions and possibly saves backtracking. The Prolog system evaluates rule calls in top down order. Therefore the first rule call keeps its first initialization as long as possible, because this is the point reached last in backtracking.

2. **Order of several queries asked together**
   The order of the single queries called within the global query body may also change the order of the possible solutions and save backtracking.

Order of the Rule Calls

The order of the rule calls in the body of a Prolog rule is important. The Prolog system evaluates the rule calls top down. If the last rule call cannot be fulfilled, the system backtracks as long, as a solution is found or all solutions are tried. Therefore it is useful to start with more restrictive rule calls first. A bad order of the rule calls is mainly a loss in performance. With reordering rules, we do not necessarily get better solutions first.

The following example shows the constraints, which were generated for the introductory example in section 2.2. We provide two possible orders of the generated constraints.

```
field_declaration(none, [P_adr]),
parameter_declaration(none, public,
  [P_set_a]),
assignment([[this], [P_adr]],
  [[P_set_a]]),
method_declaration(none, none,
  [P_get_Return]),
return_statement([[P_get_Return],
  [[this], [P_adr]]).
```

```
assignment([[this], [P_adr]],
  [[P_set_a]]),
return_statement([[P_get_Return],
  [[this], [P_adr]]).
field_declaration(none, [P_adr]),
parameter_declaration(none, public,
  [P_set_a]),
method_declaration(none, none,
  [P_get_Return]),
```

The left order is just the way the syntax tree is visited. But the declaration rules mean no real type restrictions for the variables. Therefore the right order of the rules is better, because we execute the expression rules first. Expression rules do apply the Universe type system to the used variables. The more expressions we have with all declared variables, the better annotations we can get. The declaration rules only tell us, if the found solution is valid in this special declaration environment. For example in static methods no usage of *rep* variables is allowed.

We have to consider, that the first rule call is the last, which is reached while backtracking. For example we can have a list of field declaration rule calls in our query. The system will try to keep the first field *rep* as long as possible. It will first set all other fields to *peer* until it tries out a solution with the first field *readonly* or even *peer* and all other fields *rep* or *readonly*. Therefore the order of the rule calls can also change the order of the possible solutions.

Order of the Queries

The solution found for the first query, initializes all variables of the first query with a possible value. These values are used in all other queries as fixed values for the same variables. The Prolog system tries out all possible solutions for all other variables, which are not used in the first query. If no variation does succeed, the solution of the first query is rejected and the system checks if the next found solution for the first query works out for all others.

The order of the single queries in the global query body is therefore also important. If we start for example with all small classes first and the big main class is asked at the end, it will most surely not be fulfilled with the first allocation and the Prolog system has to backtrack all small queries several times.

In general it is a good approach, to start with classes, which restrict the used variables of the program more than the other classes. Look also at dependencies of the single classes and start rather with big classes than with small.

Have a look at the examples in section 4.1 to see, which order was preferred in some examples. In the examples we usually ordered the queries of the single classes by their usage. The query of a class, which uses another class, is called before the query of the used class.

2.4.6 Heuristics

Not all annotation solutions, which are possible, are of interest for the user. But the solver of our architecture can find all possible solutions. We have to find the approximative best solution of all possible solutions.

A solution, which maps all variables to *peer* is only interesting, when it is the only solution the system finds. Then we would like to understand, why this is the only solution and what we have to change in the program, that there are other solutions. Normally we are not interested in solutions, which set all objects into the same context, this is like not adding any annotations at all.

The user has some preferences, how the best annotation for a problem should look like. The first solution the solver returns, need not fulfill all these preferences.

We have to decide at how many solutions we look at most to find the best solution and which properties must be fulfilled for a best solution. The maximum number of solutions we want to look at, depends on the size of the program, the computer we use and on the time we want to wait for a solution.

1. *Check a set of solutions*
   It can be time consuming to look at all possible solutions the Prolog system finds to find one solution, which fulfills all preferences of the user.

2. *Estimate a solution*
   We decide, what properties the best solution must fulfill and decide for each solution of a set, if it is among the best ones.

We need to look at at least two possible solutions - and of course more than two if we like to - and compare them. We have to compare a set of solutions, picking the one, which fulfills the conditions we like to have; like that we want to get solutions with more *rep* and *readonly* than peer. This could be done in Prolog but is not very nice. We would have to force the Prolog system to give us only a set of all possible solutions. It is easy in Prolog to get all possible solutions, but this would slow down our system a lot and might not even succeed.

The Java interface to Prolog can get an iterator with the query we ask to the Prolog system and the next element is just the next solution the system would return. We get a number of solutions and visit them within our Java code, to pick the best solution of this set, because the first solution need not be the best one.

```
while(have not visited allowed maximum number of solutions) {
       solution = get next possible solution;
       if(solution fulfills all wished properties) {
              return solution;
       }
}
```

Check a set of solutions

There are the following possible approaches to visit a set of solutions:

1. check only one solution at once
2. check more than one solution at once
3. check all possible solutions at once
4. check at most `n` solutions
5. check at most all possible solutions

To find a good solution of all possible solutions, we have to check all solutions. But this can be a great loss in performance for big programs or might not even succeed in time. We can decide at how many solutions to look at in one step, because we may want to pick a good solution out of three given solutions or even more. We have to give a limit of the maximum number of solutions to visit, because we look as long at more solutions as the given properties are not fulfilled, which designate the best solution.

If we have only one property, which has to be fulfilled, this is done very simple:

```
while(have not visited allowed maximum number of solutions) {
       solution list = get list of possible solutions in this step;
       if(solutions of list fulfill wished property) {
              return best solution of the list;
       }
}
```

It is possible to have several properties, which characterize the best solution. Each property allows us, to pick the best solution of a set. But they do not need to agree about the best solution. We check more solutions as long as they do not agree about the best one. But if we already visited all solutions, we pick the best found solution of the most important property.

```
while(have not visited allowed maximum number of solutions) {
       solution list = get list of possible solutions in this step;
       for(all properties) {
```

```
            best solution i = best solution of the list for property i;
      }
      if(all properties found the same best solution) {
            return best solution;
      }
}
for(all properties) {
      if(property has found best solution and has highest priority) {
            return best solution of property i;
      }
}
```

Estimate a solution

The best solution is designated by the following properties:

1. least number of *peer*
2. least number of conflicts
3. fields are wished to be *rep*
4. parameters and return types are best to be *readonly*

Mapping all variables to *peer* is nothing else, than not using Universe types at all, because all objects lie in the same context. We want to annotate a variable with *peer* only, if we absolutely wish or need to.

It is also possible, that all variables have a conflict somewhere. This is not very useful either. We allow conflicts, because with them, it is more likely, that the used variables can get *rep* or *readonly*. Conflicts should only be allowed, when a variable has to be mapped to *peer*, if we do not allow a cast at that point.

It is more important, that a field is of type *rep*, than that a parameter is not of type *peer*. Fields designate the attributes of a class and a class should usually own its attributes. Like this they are protected by the class and can only be changed by accessing the class. There are some exceptions, when a field is wished to be *peer*: For example if we design a data-structure like a list item, that should belong to a list. All list items are then owned by the list, but have to be in the same context. Therefore all attributes of the list item are mapped to *peer*.

This shows us, that some properties are more important than others. We need to know, which property is the most important. We show in the next example, how different solutions can be evaluated as the best one by different properties. We show two possible solutions of the introductory example in section 2.2.

```
class P {
  /*@ rep @*/ A adr;
  void set(/*@ readonly @*/ A a) {
    this.adr = (/*@ rep @*/ A) a;
  }
  /*@ readonly @*/ A get() {
    return this.adr; }
  }
}
```

```
class P {
  /*@ readonly @*/ A adr;
  void set(/*@ readonly @*/ A a) {
    this.adr = a;
  }
  /*@ readonly @*/ A get() }
    return this.adr; }
  }
}
```

The first heuristic would estimate both solutions as equal, because they contain no *peer* at all. The second heuristic would choose the right solution as the better one, because it contains no conflict. The third heuristic would prefer the left solution, because the field is mapped to *rep* and for the fourth heuristic both solutions are equal again. Nevertheless the left solution is the better one, because the right solution does not tell us anything. All objects are *readonly* and if we later

add some more methods, this might not even work anymore.

Chapter 3

Implementation

In this chapter we show some implementation details of our developed architecture. In section 3.1 we discuss the main implementation of the architecture and in 3.2 we show what solver databases and solver optimisations we provide and discuss two implemented heuristics. The configuration of the architecture is explained in 3.3 and the usage in 3.4. In section 3.5 we discuss some of the tools we used to implement and use the architecture.

3.1 Static Inference Tool

The static inference tool has one main package, the `ch.ethz.inf.sct.static_typinfer` package, which contains the main Java part of the tool. The class diagram of the main package is shown in figure 3.1. We give a short overview over all classes now and then discuss each class in detail.

The main package contains the following classes:

Main
The Main class connects all parts of the static inference tool and is used to start the tool.

Configuration
This class reads in the configuration for the static inference tool given in the config.xml file.

Name Generator
The Name Generator is used to generate Prolog variable names for each Java method and variable.

Annotation Reader
This class reads in the purities and annotations given by the user.

Constraints Generator
The Constraints Generator visists the syntax tree of a class and generates a Prolog query for it.

Inquirer
This interface is used to get all solutions of a query from the solver.

JPL Prolog Query
The Prolog Query class asks a Prolog query to the SWI Prolog system.

Annotation Writer
This class writes the found annotation solution into a given XML format.

Helper Class
The Helper Class contains some useful methods.

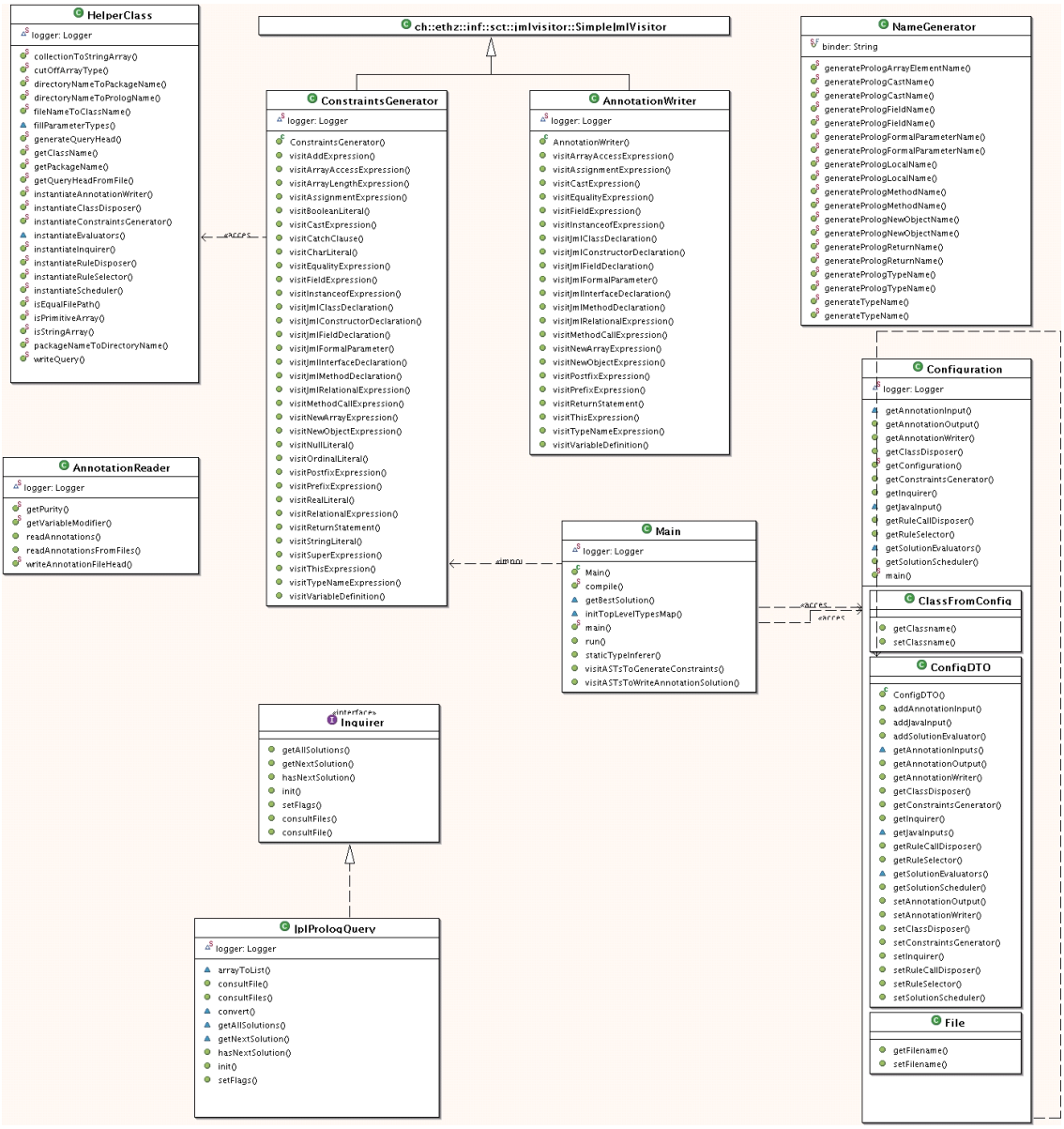

Figure 3.1: Class diagram for the main package

3.1.1 Main

The Main class puts all parts of the static inference tool together. It is also the starting point of the tool, containing the *main* method.

The Main class calls the other parts of the tool in the following order:

1. Annotation Reader
2. Constraints Generator
3. Solution Scheduler and Evaluator
4. Annotation Writer

The annotation reader parses and stores the given purities of all methods and the annotations of already annotated variables. This information is used in the generation of the constraints.

The generated constraints are ordered during the generation step with one of the orders discussed in section 3.2. Each generated constraint will be inserted in the chosen order during the

first traversal of the syntax tree.

When we are finished with the traversal of all syntax trees with the constraints generator, the global query rule is generated, which calls all generated constraints. We save each query head, we generated for a class, for it will be used again in the global query body. The heads are inserted in the global query body with a chosen order explained in section 3.2. If there is only one single class to annotate, no global query is generated. We directly use the head of the query generated for the class to ask to the Prolog system.

The solution scheduler and evaluator are discussed in section 3.2 and they may be called several times until the wished solution is found.

The annotation writer traverses each syntax tree of the original program again to resolve type conflicts and writes the annotation solution to the output.

3.1.2 Configuration

The Configuration class parses the config.xml file, which is used to configure the static inference tool. The class is used several times to get the necessary information about input and output of the tool. This class is implemented with the singleton pattern, so we can get a needed attribute of the configuration at any point in the static inference tool. Please have a look at section 3.3, where the configuration file is explained in detail.

3.1.3 Name Generator

This class contains methods to build a Prolog variable name for each of the following constructs:

- fields
- methods
- return type of a method
- formal parameters
- local variables
- newly created objects
- array elements
- static target
- cast types

Have a look at section 2.4.3 for the naming rules.

The Prolog variable name for methods is only built to look up the purity of a method. These names are only generated and used in the Java part of the tool and never in the Prolog part. Because of the uniqueness of the Prolog variable names, we can use them to store the purity of the methods in a hashtable. We are already building the Prolog variable names for methods in the constraints generator, because they are used for the Prolog variable names of local variables.

3.1.4 Annotation Reader

The Annotation Reader reads in already given annotations and the purity of all methods. This is done by a method, which reads in the XML input file. The information of the XML file is read and for each variable and method a Prolog variable name is generated. Each Prolog variable name is saved in a hashtable as the key with the appropriate purity or annotation as the value. The uniqueness of our Prolog variable names allows us to use a hashtable for storing the annotation of each variable and the purity of each method. With a hashtable we can quickly get the purity of a method.

3.1.5 Constraints Generator

The Constraints Generator class is derived from the `SimpleJmlVisitor`. The `SimpleJmlVisitor` is used to traverse the syntax tree of a Java program.

While traversing the syntax tree of a Java class, we generate the Prolog rule calls as described in section 2.4.1. We discuss the implementation in detail next.

Deepest Node Problem

When we traverse the syntax tree, we reach the deepest node at some point. In this deepest node we actually know where we are, but we cannot reconstruct, where we just came from.

In the local variable and the formal parameter node, we have to know, if we started in a variable declaration, an assignment, a field access, a method call or in another expression. Therefore we save the path in the syntax tree within flags. Basically it is enough to save, if we are in an expression or in a declaration. In an expression we generate the Prolog variable name, in a declaration the whole rule call of the declaration rule, when reaching a local variable or formal parameter node.

Traversal Order

In each expression there might be some method calls or newly created objects, which require to generate an additional call to a Prolog rule. We have to generate these rule calls before generating the rule call of the expression itself. Whenever reaching an expression, we first generate all additional rule calls within the expression. This is done very easily by declaring, that we are not in the generation of an expression now. Then all rule calls for newly created objects and method calls are created. After this the same part of the syntax tree is traversed again, but this time we specifiy, that we are in an expression now. Like this we have to traverse some parts of the syntax tree twice.

Visiting some parts of the syntax tree twice is a loss in performance. We discuss on the next example, why we have to do this:

```
temp.m(var.p(a).field).f = t;
```

We generate the following rule calls for this line of code:

assignment([[V_temp], [MRet], [V_f]], [[V_t]]),
method_call_parameters_left([[V_temp]],**[[[V_var], [PRet], [V_field]]]**, [[MPar]], none),
method_call_parameters_left([[V_var]], [[[V_a]]], [[PPar]], none),

The bold parts have to be generated simultaneously. When visiting variable `var` we have to generate the Prolog variable name and use it in the argument of the first generated method rule call and in the target of the second. Because of the deepest node problem we do not know, where we just came from and that we have to append the generated Prolog variable to more than one constraint. Each generated constraint has to be global, because we must be able to add another Prolog variable name to the constraint in each node we visit. With arbitrary nested method calls we cannot determine anymore, to which constraint we have to add a generated Prolog variable

name.

Information Storage

During the traversal some other useful information is saved within flags, for example in what kind of method we are:

- constructor
- static
- public

This information is used in the generation of the declaration statements.

For the left-hand side of an assignment there is always a more restrictive rule than for the right-hand side. Therefore we need to store on which side of the assignment we are. Other useful information to store is: the classname of the current class and the current method name.

All generated Prolog variable names are saved in a hashtable, because they build the solution set and are used in the query head of the class. We are using a hashtable, because we insert a Prolog variable name, whenever we create one. Like this the same Prolog variable name might be inserted into the hashtable more than once. The Prolog variable name is the key of the hashtable and it is unique. If we insert the same name twice, the old entry in the hashtable is just overwritten with the excactly same entry. The Prolog variable name of a variable does not change and neither does the kind of the variable.

The value of the hashtable is the kind of the variable:

- ”field”: for fields
- ”return”: for return types
- ”parameter”: for parameters
- ”local”: for local variables
- ”new”: for newly created objects
- ”type”: for static targets
- ”cast”: for cast types

Method Declarations

For each method declaration or call we have to check if the method is an overridden one. For overridden methods we have to generate the same Prolog variable names for the parameters and the return type as in the superclass or interface. We check for every method if there exists the same method in one of the superclasses or in one of the interfaces. The method is an overridden one, if it has the same method name and the same type and number of parameters as the method of the superclass or interface. We always generate the Prolog variable name for the method in the highest superclass or interface. This is done, because overridden methods must have the same signature with the same Universe types as the original method.

Object Creation

The creation of new objects is a bit special. When first traversing the expression, where one or more new objects are created, we generate the rule call and the Prolog variable name for the newly created object. Each generated Prolog variable name for a new object is saved on a temporary stack. When traversing the same part of the syntax tree for the second time, we access the names on the stack and use them in the rule call for the expression. This works because a newly created object is limited to one expression. The last generated object is used next in the generation step. Please have a look at the next example for a better understanding:

```
class Node {
       private Node(Object o) { ...  }

       public Node newNode() {
              Node n = new Node(new Object());
       }
}
```

The one line of code in the method `newNode()` generates the following rule calls:

```
local_variable_declaration(none, [V_n]),
new(none, [V_New2]),
new(none, [V_New1]),
method_call_parameters_right([[V_New1]], [[[V_New2]]], [[MPar]], none),
assignment([[V_n]], [[V_New1]]),
```

References

In each variable node, we have to check, if the variable is a reference or not. If it is not, we just generate the constant `primitive` as explained in 2.4.1. If the variable is a reference, we have to generate the Prolog variable name for it. If the variable is already annotated by the user, we directly generate the annotation. For Strings we always generate *readonly*.

Arrays

For arrays we normally have to generate the Prolog variable name for the array elements. Whenever we reach an array reference, we have to check the following cases:

- array of primitive type
- array declaration
- expression with array access

The Prolog view of an array access and an array reference assignment differs. Please have a look at section 2.4.1. In an assignment of one array reference to another, where the arrays are of primitive type, we do not generate anything for the array elements. But if we are in an expression of an array access of a primitive array, we do generate the constant `primitive` for the array elements.

Query Rules Order

To build up the query of a class, we save each generation of a Prolog rule call in a string. When the rule call is completed, we add the string to a linked list, which saves all rule calls. The linked list is very practical, because we can add elements at the beginning, the end or in the middle of the list. Therefore we order the rule calls during the traversal of the syntax tree. We do not need to reorder them after the traversal. The order of the rule calls is done by an implementation of the interface explained in section 3.2.

3.1.6 Inquirer

The Prolog Inquirer is an interface to the solver system. It is an accessor to all solutions of a query and provides the following methods:

void init(String query);	initializes the Inquirer with the query string to ask
boolean consultFile(String fileName);	loads one file into the solver system
boolean consultFiles(List<String> fileNames);	loads a list of files into the solver

void setFlags();	configures the system of the solver
boolean hasMoreSolutions();	to check if there is another solution
List<Map<String,String> > getAllSolutions();	get all possible solutions of a query
Map<String,String> getNextSolution();	get the next solution

The `hasMoreSolutions()` method already gets the next solutions, which will be returned by a call to the `getNextSolution()` method. Therefore the `hasMoreSolutions()` method needs not to return a fast answer.

The last one is the same as when we are in the Prolog system and type a semicolon after the actual solution the system shows. With these methods it is possible to get an iterator over all possible solutions of a query.

A solution is always presented as a map of strings. The keys are the Prolog variable names of the query head and the values are the solutions of the respective Prolog variables, i.e. Universe types.

3.1.7 JPL Prolog Query

This class implements the `Inquirer` interface. It uses the JPL interface from SWI Prolog. JPL is a Java interface to Prolog that allows to call a Prolog query from Java code. JPL requires SWI Prolog 3.1.0 or later and provides several classes to call Prolog from Java. We cover the functionality of this interface in our JPL Prolog Query class for one could later use another interface than the JPL interface or even another Prolog system as the SWI Prolog system.

3.1.8 Annotation Writer

The Annotation Writer class is also derived from the `SimpleJmlVisitor` and writes one annotation solution for a class to a predefined annotation XML format.

Traversal

The benefits of traversing all syntax trees again are:

- the right order of all variables
- all information about the variables is available: name, type, line number, index and so on
- tracing of additional casts

Another possibility is of course to save all information during the first traversal of the syntax tree. But if we want to allow conflicts, we have to retrace them and write them to the XML output file. The user needs to know, in which line a conflict actually occurs. This can only be done in traversing the syntax tree again and checking each expression for occurring conflicts. We write down additional casts for occurring conflicts as explained in the conflicts section in 2.4.4.

Variable Declarations

We mainly look at the declaration of variables, because we write an XML tag for every declared variable. When allowing conflicts we also have to check each expression for additional casts.

For every reference, we have to get its annotation. This can be a part of the solution of the Prolog query or an already given annotation by the user. If we get the type `conflict_rep` or `conflict_peer` as solution for a reference, we simply write *readonly* as the modifier. We also have to check for each reference, if it is an array reference of a non-primitive type. We just generate the array element name for each reference and check, if there is an annotation. If there is one, we know, that the reference is an array of a non-primitive type.

Method Declarations

We have to check for each parameter and return type of a method again, if they are of an overridden method. First we generate the Prolog variable names for the method of the actual class. If we do not find a solution in the hash-tables, we generate the Prolog variable names for the method of the superclass and the interfaces and so on. As long as we do not get any solution, it was not the right Prolog variable name. This is done for all super-classes and interfaces of a class.

3.1.9 Helper Class

This class contains some helpful methods like the following:

- split off class name from package name
- split off the package name from a fully qualified type name
- convert directory name into package name
- convert package name into directory name
- checks if an array type is of a primitive type
- generation of the head of a query with all Prolog variable names
- writing a whole query to a Prolog file
- read query head from a file
- determine kind of variable by the Prolog variable name
- if a string is contained in another string

The Constraints Generator gives us all Prolog rule calls of the query body of a class in a linked list and all Prolog variables of the query head in a hashtable. With this information, we are able to write down each Prolog query for each class in a separate file. If the file already exists, because the query has been generated in another context before, we are not traversing the syntax tree again. We read in the query head and all Prolog variables of the query head from the already existing file.

3.2 Solver and Heuristics

All interfaces and classes of this part belong to the `ch.ethz.inf.sct.static_typinfer.heuristic` package. The class diagram of all classes in this package is shown in figure 3.2. All of the following interfaces are implemented to give the user the possibility to experiment with the static inference tool. In the future we need more tests to figure out how the static inference tool is configured best.

3.2.1 Solver Database Selection

The Prolog part of the static inference tool is a Prolog program, which models the type relations of the Universe type system. We implemented two possible Prolog databases one can use. The first one is the Prolog database to model the Universe type rules in Prolog and the second one contains the additional type rules to trace conflicts automatically. Developers may also write and use new Prolog programs. We need a new Prolog program, when the Universe type system changes or developers want to apply a whole new type system to Java programs.

If the Prolog part is written within more than one Prolog file, we can still load all files at once into the Prolog system. So we allow only one class to be used for the following interface.

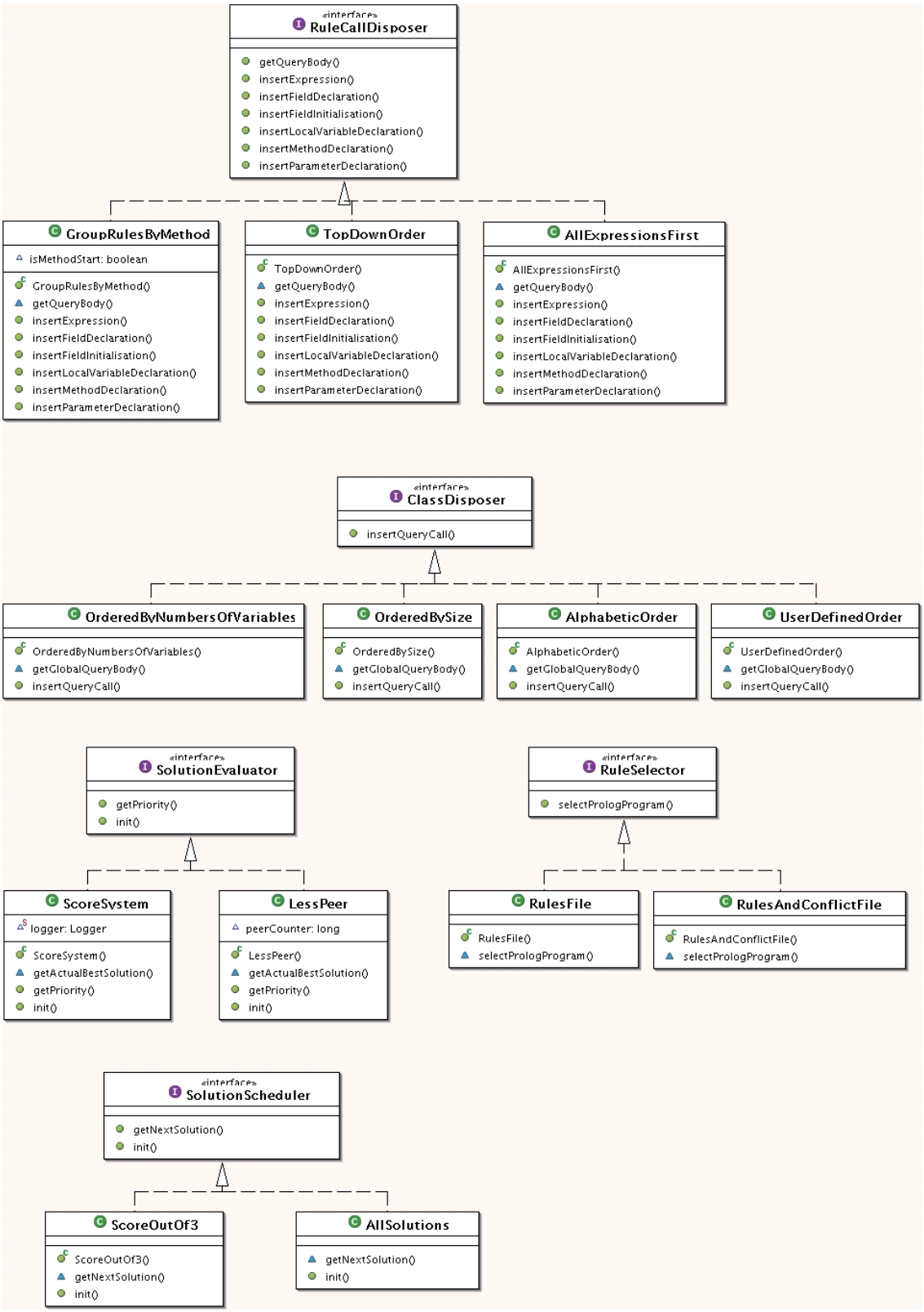

Figure 3.2: Class diagram for the heuristic package

Rule Selector Interface

This interface is called in the main class before we ask the global query to the Prolog system.

The `RuleSelector` interface is kept simple:

`List<String> selectPrologProgram();`
All file names of all parts of a Prolog program are returned in a list and will be loaded in the Prolog system together with all generated Prolog files in the main class before asking the global query. All files of a Prolog program have to be in the current directory and the name must be given without the ".pl" ending of the Prolog file name.

We provide two implementations of this interface:

Usage of the Universe Prolog Program

We return only the `rules.pl` file. The `rules.pl` file is the Prolog part of the static inference tool and our provided implementation of the Universe type system in Prolog.

Additional use of the Conflict Prolog Program

We return the `rules.pl` file and the `conflict.pl` file to the Prolog system. The `conflict.pl` file implements the two new types `conflict_rep` and `conflict_peer`, which are used for allowing conflicts. It overrides several rules from the `rules.pl` file. There will be several warnings printed out by the Prolog system, when the `conflict.pl` file is loaded. The Prolog system prints a warning for each Prolog rule of the `rules` database that is overwritten by the same rule in the `conflict` database. All warnings can be ignored.

The order of the two files is important. The `rules.pl` file has to be loaded into the Prolog system before the `conflict.pl` file. Otherwise the `rules.pl` file would overwrite the rules of the `conflict.pl` file and the `conflict.pl` file would be useless.

3.2.2 Optimisations

We provide two interfaces for the two optimizations discussed in section 2.4.5. The first one is used to order the rule calls in the query of a single class and the second one is used to order the query calls in the global query body. We provide some implementations of each of the interfaces.

Rule Call Disposer Interface

The user can only specify one class to order the rule calls in a query. It is sufficient to order the rule calls of all classes in the same way.

This interface is called during the generation of the constraints and contains the following methods:

`void insertFieldDeclaration(String fieldDeclaration);`
inserts the generated constraint of a field declaration at the right position in the query body

`void insertFieldInitialisation(String expression);`
inserts the generated constraint of a field initialization at the right position in the query body

`void insertExpression(String expression);`
inserts the generated constraint of an expression at the right position in the query body

`void insertMethodDeclaration(String declaration);`
inserts the generated constraint of a method declaration at the right position in the query body

`void insertParameterDeclaration(String declaration);`
inserts the generated constraint of a parameter declaration at the right position in the query body

`void insertLocalVariableDeclaration(String declaration);`
inserts the generated constraint of a local variable declaration at the right position in the query body

`List<String> getQueryBody();`
returns the whole query body in a list

We implemented three classes with possible realizations of this interface:

Top Down Order

This class orders the rule calls in the way the syntax tree is visited. It contains just one single list and appends each rule call at the end of the list.

All Expressions First

Declarations tell us, which solutions are allowed for a variable, but expressions do actually restrict the set of possible solutions for a variable. The usage of a variable shows the relations a variable is used in and leads to the correct annotation. When starting with all expression rule calls first, we can avoid some backtracking.

This class contains two lists:

- one for all expressions
- and one for all declarations

The two lists are connected to one list in the `getQueryBody()` method. We start with the calls to all expression rules from all methods and ask at the end all declaration rule calls from all methods.

```
Order                    := [FieldExpressions] {Expressions} Declarations.
Fieldexpressions         := all generated expressions for field initializations.
Expressions              := all generated expressions for a method.
Declarations             := [Fielddeclarations] {[Methoddeclaration]
                            [Parameterdeclarations] [Localvariabledeclarations]}.
Methoddeclaration        := generated method declaration for one method.
Parameterdeclarations    := all generated parameter declarations for one method.
Localvariabledeclarations := all generated local variable declarations for one method.
Fielddeclarations        := all generated field declarations.
```

Group Rules by Method

Local variables are only used within the method body and fully restricted there. If we group all rule calls of a method together, at least the rule calls of all local variables are grouped together. When a solution found for a local variable does not fit the declaration or the usage in other expressions, the Prolog system will only backtrack locally. Of course we have to ask the expression rule calls of the variable before the declaration rule call, because they are more restrictive. Field initializations and declarations are asked at the very end, because they may be used in all methods of a class.

This class contains five lists:

- one for all expressions of a method
- one for all declarations of a method
- one for all field initializations
- one for all field declarations
- and one for the whole query body

We add the first two lists to the last one, whenever we reach another declaration part of a new method. Whenever we register the first call to the `insertMethodDeclaration` or `insertParameterDeclaration` method after some calls to the `insertExpression` method, we know, that we enter a new method. If a method has no reference return type and no reference parameters we cannot notice, that we are entering a new method and the declarations are appended at the end or after the next method with a reference return type or parameter.
The field initialization and declaration lists are added to the end of the last list, before it is returned in the `getQueryBody()` method.

```
Order                       := {Expressions Declarations}
                               [Fieldexpressions] [Fielddeclarations].
Expressions                 := all generated expressions for a method
Declarations                := [Methoddeclaration] [Parameterdeclarations]
                               [Localvariabledeclarations].
Methoddeclaration           := generated method declaration for one method.
Parameterdeclarations       := all generated parameter declarations for one method.
Localvariabledeclarations   := all generated local variable declarations for one method.
Fieldexpressions            := all generated expressions for field initializations.
Fielddeclarations           := all generated field declarations.
```

Class Disposer Interface

This interface is used to order the query calls in the global query body. As we have only one global query, it is sufficient to order the query calls in the global query body with one given order. We call this interface in the main class after the generation of all constraints, while generating the global query.

The `ClassDisposer` interface contains only two methods:

```
void insertQueryCall(Map<String,String> allPrologVariables,
     Map<String,String> alreadyAnnotatedVariables, String[] javaFiles,
     String javaFileName, String queryCall);
```

The first method inserts one query call in the global query body. All given Parameters can be used to determine the place of the query call.
`allPrologVariables`: contains all Prolog variable names of all references, which are not annotated by the user, for a whole program.
`alreadyAnnotatedVariables`: contains all Prolog variable names of already annotated variables
`javaFiles`: contains all file names of the Java files of the program
`javaFileName`: name of the Java file to insert generated query in the global query body
`queryCall`: query to insert in the global query body

```
List<String> getGlobalQueryBody();
```

The second method returns the list with the global query body. The single query calls are ordered in the specified way.

We implemented the following four approaches, which use this interface:

Alphabetic Order

A simple first approach is to order the classes alphabetically, which is a kind of random order. We order the query calls by the given Java file name alphabetically.

Ordered by Size

A more specific approach to order the query calls in the global query is, to look at the size of each class file. The biggest file is placed at the beginning of the global query body. Like this bigger files are less backtracked than small files. We use the Java file name to get the appropriate file and determine its size.

Ordered by Numbers of Variables

This approach is quite similar to the last one. But we order the query calls after the number of variables to annotate. Variables already annotated by the user are not counted. Therefore we only look at the size of a query head. The query head with the most Prolog variables is asked first, so it will be always visited last while the system backtracks.

User defined Order

Another approach is to let the user order the queries directly. The order is taken from the configuration file and the single queries are asked in the same way the Java files of the related classes are ordered in the configuration file. Like this the user can order the files in the order of their dependencies. If he does arrange the files arbitrarily, we get a random order. The order of the Java files is given in the `javaFiles` array.

3.2.3 Heuristics

We provide two interfaces, which allow the user to experiment with several heuristics. The first one is only to determine how many solutions should be consulted to find an approximative best solution and the second one decides for each set of solutions if it already contains a good one.

Solution Scheduler Interface

This interface is used to decide, at how many solutions we look at in a single step and at how many in total. We allow only one instance of this interface, because the two factors, we can set, suffice.

`void init(Inquirer i);`
We initialize the solution scheduler with the interface to the solver system within this method. The inquirer interface is an accessor to all solutions of a query.

`List< Hashtable<String,String> > getNextSolutions();`
This method returns a list of the next solutions we like to look at. It returns a list of solutions as long as we have not looked at the maximum allowed number of solutions. Otherwise it returns *null*. We can look at only one solution at a time or at more than one.

We apply the interface in the main class after the generation of the constraints and implemented the following possibilities for it:

Score out of 3

We always get three solutions at a time and do this at most 10 times. Like this we look at at most 30 solutions.

All Solutions

We get all possible solutions of a query at once. The `getNextSolutions()` method of this implementation can only be called once and returns all solutions, the Prolog system finds.

Solution Evaluator Interface

The solution scheduler gives us a set of possible solutions, and we have to figure out, which one is the solution that we would like to have. The solution evaluator stores the best solution of all solutions visited so far. The best solution can depend on more than one property. Therefore we allow multiple instances of this interface.

The implementation of the evaluator interface is also called in the main class together with the solution scheduler.

```
init(Map<String,String> allPrologVariables,
     Map<String,String> alreadyAnnotatedVariables);
```

For some implementations of this interface we might need the kind of the Prolog variables and the annotations of already annotated variables to estimate a solution. The first map contains all Prolog variable names of a program with the kind of the variable saved as value. The second hashtable stores already given annotations with the related Prolog variable name as the key.

```
Map<String,String> getBestSolutionOfAllVisited(
     List< Map<String,String> > solutions);
```

Each solution evaluator estimates a solution by one property, the best solution must fulfill. This method returns the best solution found, of all so far visited solutions, which fulfills the given property. The solution evaluator picks the best solution out of the given list and compares it to the currently stored best one. This method does not return the best solution of the list given as parameter, but of all so far visited solutions.
If no solution fulfilled the property of the solution evaluator so far, this method returns *null*. But the currently best solution found is nevertheless stored within the solution evaluator. Although it does not fulfill the given property.

```
Map<String,String> getActualBestSolution();
```

If we have visited all solutions and the property of the solution evaluator is still not fulfilled, we can get the internally stored best solution with this method.

```
int getPriority();
```

Returns the priority of a solution evaluator. The solution evaluator with the highest priority is the most important. The priority is a fixed value in the implementation. We allow any integer value as priority value. Please look at the priorities of already implemented solution evaluators if you implement a new one. If your new solution evaluator is more important than all others, choose a high value, but maybe not the highest possible integer. There might be a more important property in the future. If there is more than one solution evaluator with the same highest priority, the one, which is declared first in the configuration file will always succeed.

Less Peer

A simple property is that we want to get more *rep* and *readonly* in our solution than *peer*. We count the occurrences of *peer* in each solution and keep the one with the least *peer* references.

This solution evaluator has priority 0, because it is not very important.

Score System

Another property is, that we rather want fields to be *rep* than local variables. We can use a kind of a score system, where a *rep* field counts more than a *readonly* field or a *rep* local variable. The solution with the highest score wins and is returned.

We assume a score for each Universe type depending on the kind of variable:

Universe type	**fields**	**parameters**	**local variables**	**return types**
rep	2^3* number of variables*	2^1	2^2	2^1
readonly	2^2* number of variables*	2^3	2^3	2^3
peer	0	2^2	2^1	2^2
conflict	2^1* number of variables*	0	0	0

One field scores as much as the sum of all variables to annotate, because it is more important to have a *rep* or *readonly* field than a *rep* or *readonly* parameter, local variable or return type.

To determine if we already have found a good solution, we use a threshold score, which is half of the maximum possible score:

threshold = (
    2^3* number of variables * number of fields*)
    + (2^3* number of parameters*)
    + (2^3* number of return types*)
    + (2^3* number of local variables*)
) / 2.0;

We only return a solution, if we have already found a solution with a score, which is better than the threshold. We save the solution with the highest score within the class.

This solution evaluator has priority 10, because it refines the property of the other solution evaluator.

3.3 Configuration

The user has to configure the static inference tool with the configuration file and the annotation files. In the configuration file, the user has to specify the following parts of the static inference tool:

- Java input
- annotation input
- annotation output
- solver system
- heuristics

Like this all parts of the static inference tool are exchangeable.

The Java input files are simply the files the user likes to annotate by the tool. The file names have to be given as fully qualified type names, meaning with preceding package name.

In the annotation input the purity of every method of each class, the user wishes to annotate, must be specified. The purity cannot be determined by the static inference tool and has to be given by the user. All parameters of each method are required, because the static inference tool needs the parameter types for the generation of the Prolog name for a method. The return type

and the local variables are not needed. The XML input files have to be in the right form, which is declared in the annotations.xsd file. The XML format for the annotations input and output and the one for the configuration file are written in the annotations.xsd and the configuration.xsd file. Both files are stored on the CD, which contains the whole architecture.

The user also has the possibility to specify some predefined annotations in one or more XML file. The tool then tries to find a solution, which satisfies these annotations. Another possibility for the user to insert annotations, is to directly declare them in the Java file at the variable declaration point.

The user can specify more than one annotation input file. He can for example give one file containing all purities and one file with additional annotations or a file with annotations and purities for each class.

It is possible to get for every Java class a XML annotation output file. This is useful, if the user wants to store the annotation of single classes or wants to reuse the annotation of a class for another inference step. The user has only two possibilities:

- one output file
- one XML output file for each analyzed class

In the first case all classes are written to one file with the name given in the configuration file. In the second case the user does not specify a filename in the configuration file and each output file gets the same name as the appropriate Java class.

The solver system has to be specified with an implementation of the inquirer interface.

As a last point the user needs to configure, which heuristics he wants to apply to get the best solution. For each interface discussed in section 3.2 the user must specify the class name of exactly one implementation. Only for the `SolutionEvaluator` interface there can be more than one implementation given.

3.4 Usage

The static inference tool is used in the simple following way:

```
run
```

Just run the `run` shell script in your console. For the run shell script to work, the JPL environment shell `env.sh` is needed in the working directory as well. The `env.sh` script sets some path variables for JPL. It can be found in your SWI Prolog installation in the examples order: `file:/usr/lib/pl-5.4.7/doc/packages/examples/jpl/java`. The run shell script simply runs the main class.

The `compileAndRun` shell compiles the code, applies the retroweaver tool to it and runs the main class. Developers, who add new classes or change existing classes, should use this shell.

The CLASSPATH must contain all libraries, which are needed by the tool:

- commons-beanutils.jar
- commons-collections-3.1.jar
- commons-digester-1.7.jar
- commons-logging-1.0.4.jar
- log4j-1.2.11.jar

- jsr173_api.jar
- xbean.jar
- annotations.jar
- jpl.jar
- antlr-2.7.5.jar
- java-getopt-1.0.11.jar
- junit.jar

They are saved in the `lib` directory of the tool. If you install a newer SWI Prolog version, feel free to overwrite the jpl.jar file in the `lib` directory. The jpl.jar file is given within your SWI Prolog installation. The annotations.jar file must first be generated out of the annotations.xsd file, which is done by the `scomp` tool:

```
scomp -out annotations.jar annotations.xsd
```

This tool comes together with the XMLBeans library.

The configuration file for your shell should look the following way:

```
export JUTS_ROOT=/home/MJ/
export MJ_RELATIVE_ROOT=../../../MJ/
export JML_ROOT=/home/JML2
export JML_CLASSROOT=/home/JML2/
export JAVA_HOME=/usr/java/j2sdk1.4.2_08
export JAVAC=$JAVA_HOME/bin/javac
export JAVA=$JAVA_HOME/bin/java
export JAR=$JAVA_HOME/bin/jar
export PLBASE=/usr/lib/pl-5.4.7
export PLLIBDIR=$PLBASE/lib/i686/libjpl
export LD_LIBRARY_PATH=$PLLIBDIR
export TOOLS14=$JAVA_HOME/lib/tools.jar
export HOME=/home/static-inference/lib


export CLASSPATH=${HOME}/antlr-2.7.5.jar:${TOOLS14}:
                 ${HOME}/java-getopt-1.0.11.jar:${HOME}/junit.jar:
                 ${JML_ROOT}:${JML_ROOT}/specs:
                 ${JUTS_ROOT}:${HOME}/jpl.jar:${JAVA_HOME}/lib:
                 ${HOME}/commons-beanutils.jar:
                 ${HOME}/commons-collections-3.1.jar:
                 ${HOME}/commons-digester-1.7.jar:
                 ${HOME}/commons-logging-1.0.4.jar:
                 ${HOME}/log4j-1.2.11.jar:
                 ${HOME}/jsr173_api.jar:
                 ${HOME}/annotations.jar:${HOME}/xbean.jar:.


export PLSL_CLASSROOT=${JUTS_ROOT}
export MJ_CLASSROOT=${JUTS_ROOT}
export JPATHSEP=:
export JFILESEP=/
export JTEMP=/tmp/
```

The example is used for a bash shell under Linux Red Hat. Please replace `/home/` with your home directory, where you saved the Multijava and JML environment.

The config.xml file cannot change its name and has to be placed in the working directory from where the tool is started. This is the same directory where the run shell is.

All generated query files are stored in the working directory, because the static inference tool and therefore the Prolog system is started from the working directory. If there is already stored a query file for a class in the working directory, the syntax tree of the class is not traversed and no new query is generated. If one has changed the class and needs to rebuild the query for it, one has to delete or remove the old query file first. Otherwise the file will not be newly built. This is actually useful for the case if the user wants to use the already compiled class in another context and the query file needs not be built again.

3.5 Used Tools

We used the following tools for the implementation of our tool.

3.5.1 Reading and Writing XML

To read and write XML at several points in the Java part, we use the `org.apache.xmlbeans` package. This package can generate classes for each XML tag out of a *.xsd* file. The classname is always built the following way:

`Classname =` *name of the tag starting with a capital letter* `+ "Document"`

Each class has a method to add a nested tag and so it is easy to build up the hierarchical structure of the XML file.

3.5.2 Retroweaver Tool

We used the SDK 1.5 to implement the Java part of the inference tool, because we want to use the generic features of the new Java version. The visitors of the static inference tool have some dependencies on JML and Multijava, which requires SDK 1.4.2. We used the Retroweaver tool to transform compiled SDK 1.5. code in SDK 1.4.2 executable code.

The Retroweaver tool is used to convert SDK 1.5 runnable code into SDK 1.4.2 runnable code. SDK 1.5 has many new features, which a developer wants to use. But many environments still run only under SDK 1.4.2. Retroweaver is an open source tool to solve this problem.
The tool has a GUI or can be started on the command line. One has to declare the directory, where the SDK 1.5 compiled classes lie and the whole classpath. Unfortunately the class path cannot be determined by the Retroweaver tool. We have to declare all dependencies inclusive two additional ones:

- rt.jar of the Java SDK 1.4.2
- retroweaver-rt.jar of the Retroweaver tool

The call to the retroweaver tool over the command line then looks the following way:

```
jdk1.5.0_02/bin/java -cp release/retroweaver-ex.jar:lib/bcel-5.1.jar:
                        lib/Regex.jar:lib/jace.jar:lib/asm-1.5.1.jar:
                        /usr/java/j2sdk1.4.2_08/jre/lib/rt.jar:
                        /home/retroweaver/release/retroweaver-rt.jar:
                        /home/static-inference/src:
                        /usr/java/j2sdk1.4.2_08/lib/tools.jar:
                        /home/MJ:/home/JML2:/home/JML2/specs:
                        /home/static-inference/lib/annotations.jar:
                        /home/static-inference/lib/ant.jar:
```

```
                        /home/static-inference/lib/antlr-2.7.5.jar:
                        /home/static-inference/lib/java-getopt-1.0.11.jar:
                        /home/static-inference/lib/junit.jar:
                        /home/static-inference/lib/jpl.jar:
                        /home/static-inference/lib/commons-beanutils.jar:
                        /home/static-inference/lib/commons-collections-3.1.jar:
                        /home/static-inference/lib/commons-digester-1.7.jar:
                        /home/static-inference/lib/commons-logging-1.0.4.jar:
                        /home/static-inference/lib/log4j-1.2.11.jar:
                        /home/static-inference/lib/jsr173_api.jar:
                        /home/static-inference/lib/xbean.jar
com.rc.retroweaver.Weaver
-source /home/static-inference/src/ch/ethz/inf/sct/static_typinfer/
```

Chapter 4

Results and Conclusion

In this chapter we discuss the functionality of our static inference tool. In section 4.1 we extensively tested the tool with some examples and discuss the annotation results. In section 4.2 we discuss the complexity of the static type inference tool. Further we compare our results to the runtime inference [9] in section 4.3 and discuss the future work, which has to be done in section 4.4.

4.1 Examples

We show the functionality of our tool at some small examples and one medium example in this section. We always present the solution with and without allowing the tool to trace conflicts.

The small examples always pick the best solution of all possible solutions if we are not allowing the tool to trace conflicts automatically. When looking at solutions with conflicts, we always look at most 30 solutions. When allowing additional casts, there are much too many solutions to look at them all.

We always order the rule calls of a single query with the `GroupRulesByMethod` strategy and the query calls in the global query body by a user defined order. We specify the user defined order in each example. We normally order the classes by their dependencies.

4.1.1 Iterator

We first look at the iterator example from [2]. Iterators need a direct reference to the internal structure of a collection, because they are allowed to read and write objects of a collection. But with a direct reference to the structure of the collection, they can also destroy that structure. In this implementation the iterator stores a reference to a list and can modify objects through the `set` method of the list.

We extended the example by the `addElement(Object e)` method in the `LinkedList` class and the `getNext()` method in the `Iter` class. Like this more dependencies between the variables are drawn. The method `equals(Object e)` overwrites the method from the `Object` class. We declare the method as *pure* and the parameter as *readonly* in the annotation input.

The class `LinkedList` is used in the class `Iter` and the class `Node` in the class `LinkedList`. This gives us the following order of the query calls in the global query body:

1. Iter
2. LinkedList
3. Node

All generated constraints are given in the appendix in A.1, where we also show, what query we ask to the Prolog system for this example and what solution is returned first.

Solution without Conflicts

```
public class Node {
        public readonly Node prev;
        public peer Node next;
        public readonly Object elem;
}

public class LinkedList {
        peer Node first;

        void set(peer Node np, readonly Object e) {
                peer Node n = np;
                n.elem = e;
        }

        public void addElement(readonly Object e) {
                peer Node n = new peer Node();
                n.elem = e;
                n.next = first;
                first = n;
                first.next.prev = first;
        }

        public boolean equals(readonly Object l) {
                if (!(l instanceof LinkedList))
                return false;
                readonly Node f1 = first;
                readonly Node f2 = ((readonly LinkedList)l).first;
                while (f1 != null && f2 != null && f1.elem == f2.elem) {
                        f1 = f1.next;
                        f2 = f2.next;
                }
                return f1 == null && f2 == null;
        }
}

public class Iter {
        peer LinkedList list;
        peer Node pos;

        public Iter(peer LinkedList l) {
                list = l;
                pos = l.first;
        }

        public void setValue(readonly Object e) {
                list.set(pos, e);
        }

        public void getNext() {
                pos = pos.next;
        }
}
```

Without allowing conflicts, we get a solution, which sets all list nodes in the same context as the list. The list does not own the list nodes as in [2]. The elements of the nodes can be in an arbitrary context and the iterator is also in the same context as the list and the nodes. This is as expected.

The best solution found by our architecture does not achieve that the list nodes belong to the list. Like this the iterator has the possibility to change the list nodes without accessing them over the list and is therefore able to destroy the structure of the list, because the `pos` field of the iterator is mapped to *peer*.

The iterator calls the non-*pure* method `set` over the `list` pointer and therefore this pointer has to be *peer*. The position into the list, the iterator stores in the `pos` pointer, is an argument of the same method call and can therefore not become *readonly*, because the combination of the target and the formal parameter is *peer*.

The parameter of the `set` method in the linked list must be *peer*, because it is assigned to a local variable, which is used to update a field. The local variable cannot be *readonly* because of the field update and therefore the parameter can neither be *readonly* as no casts are allowed. As both fields of the iterator get *peer*, the `first` pointer of the list cannot become *rep* or *readonly* anymore, because it is accessed over the `list` pointer in the constructor of the iterator and the combination is assigned to the position. All other variables are annotated as expected.

The static inference tool looked at all possible 138 solutions and chose the first as the best one, because of the less peer heuristic. The threshold of the score system is not reached and therefore, this heuristic is not considered. Although the second solution gets the most points within the score system. The threshold of the score system is not reached, because we have too many variables mapped to *peer*. The single parts of the tool were done in the following times:

Step of the Tool	*Time needed*
parse classes	21.329 s
read annotations	1.486 s
generate constraints	0.165 s
type inference	1.177 s
write annotation solution	1.054 s

As we can see in the table, the parsing of all classes needs the most time. The JML compiler needs to parse each class to annotate and all used library classes. This cannot be avoided, because we need the type informations of all used classes, otherwise we are not able to check subclasses and interfaces for overridden methods.

If we change the order to the following one, the type inference part needs more time:

- Node
- LinkedList
- Iter

Step of the Tool	*Time needed*
type inference	3.176 s

The query is much faster, when asking the query of the iterator first. Most of the possibilities are eliminated in the iterator class.

Solution with Conflicts

```
class Node {
        readonly Node prev;
        peer Node next;
        readonly Object elem;
}

public class LinkedList {
        rep Node first ;
```

```
        void set(readonly Node np, readonly Object e) {
                readonly Node n = np;
                (rep Node n).elem = e;
        }

        public void addElement(readonly Object e) {
                rep Node n = new rep Node();
                 ...
        }

        public boolean equals(readonly Object l) {
                  ...
                rep Node f1 = first ;
                readonly Node f2 = ((readonly LinkedList)l ). first ;
                 ...
        }
}

public class Iter {
        peer LinkedList list ;
        readonly Node pos;

        public Iter (peer LinkedList l ) { ... }

        public void setValue(readonly Object e ) { ... }

        public void getNext () { ... }
}
```

This time the list owns its nodes and therefore only the list has write access to its nodes. The iterator is in the same context as the linked list and has only read access to nodes. But the iterator can manipulate nodes over the `set` method of the list.

When enabling conflicts we get a solution, which differs only in the following annotations to the one found in [2]:

1. `prev` is not *peer*
   All node objects are in the same context and their references should therefore be annotated with *peer*. Our architecture does not annotate the reference to a previous node with *peer*, because it is never used in an updating field access like `next`.

2. The local variable `n` is casted to *rep* in the `LinkedList.set` method instead of the formal parameter. This is no real difference, because both casts have the same result: We can only change the value of a node over the list, because the nodes are owned by the list.

3. The local variable `f1` is not *readonly*. In a *pure* method we do like all variables to be *readonly*, because like this, we can easy see, that no object can ever be changed. Our architecture annotates `f1` with *rep*, because we always try out *rep* first.

By adding one single cast, we get a much better solution than without. This example shows, why additional casts are useful. The cast in the `set` method of the list allows the parameter to become *readonly*, which turns the position stored in the iterator into *readonly*. Only a *readonly* position allows the `first` pointer of the list to become *rep*.

The static inference tool looked at 30 possible solutions to pick the best one. The first solution was taken, because it has less peer, although the seventh solution scores the most. But the threshold is not reached. If we do not use the less peer heuristic, we get the solution, which annotates the local variable `f1` of the `LinkedList.equals` method with *readonly* as desired. The following times were needed for the single steps when conflicts and both heuristics were enabled:

Step of the Tool	*Time needed*
parse classes	21.236 s
read annotations	1.488 s
generate constraints	0.148 s
type inference	0.490 s
write annotation solution	0.883 s

4.1.2 Producer Consumer

This example was also taken from [2]. The producer and the consumer share a common buffer. The producer puts products in the buffer and the consumer gets them from the buffer. The consumer does not modify the buffer, but only reads from it. The consumer and producer are synchronized by storing a reference to each other.

Both classes use each other. The consumer constricts the variables of the producer by accessing them. Therefore we ask the query of the consumer first.

Solution without Conflicts

```
public class Producer {

        public rep readonly Product[] buf;
        public int n;
        public readonly Consumer con;

        public Producer() {
                buf = new rep readonly Product[10];
        }

        public void produce(readonly Product p) {
                buf[n] = p;
                n = (n+1) % buf.length;
        }
}

public class Consumer {

        public readonly readonly Product[] buf;
        public int n;
        public readonly Producer pro;

        public Consumer(peer Producer p) {
                buf = p.buf;
                pro = p;
                n = buf.length−1;
                p.con = this;
        }

        public readonly Product consume() {
                n = (n+1) % buf.length;
                return (readonly Product) buf[n];
        }
}
```

We get nearly the same solution as in [2]. We have the following differences:

1. the reference to the consumer within the producer is *readonly* and not *peer*
2. the parameter of the method `produce()` is *readonly* and not *peer*

3. the reference to the producer in the consumer is *readonly* and not *peer*
4. the return type of the method `consume` is *readonly* and not *peer*

If we additionally analyze another class, which uses the producer and consumer classes in the intended way, we might get the same solution as in [2]. There are no calls of the method `Consumer.consume` in this program and therefore it is enough to annotate the return type with *readonly*.

Our solution nevertheless puts the producer in the same context as the consumer, which can be seen at the parameter of the constructor of the consumer. The given producer, which is stored within the class is *peer* and therefore in the same context. The buffer is owned by the producer and the consumer can only access the buffer over the producer. The consumer can although point to the buffer, because he does not write anything to it, but only reads from it. With a *readonly* reference the consumer is allowed to point into the buffer's context, which is owned by the producer.

The static inference tool looked at all possible 9984 solutions. The first was taken, because it has less peer and the highest score in the score system.

Step of the Tool	*Time needed*
parse classes	21.177 s
read annotations	1.379 s
generate constraints	0.098 s
type inference	22.463 s
write annotation solution	2.861 s

The type inference needs so much time, because we are looking at many solutions, which is not even necessary, because the first one is already the best one.

Solution with Conflicts

When enabling conflicts for this example, the same solution as before is returned as the best one. There are no conflicts needed in this example.

Step of the Tool	*Time needed*
parse classes	21.338 s
read annotations	1.465 s
generate constraints	0.115 s
type inference	0.303 s
write annotation solution	0.903 s

This time the type inference is much faster, because we are only looking at the first solution. Both heuristics agree on the first solution and therefore no more solutions are checked. On this example we see, that it is sometimes a waste of time to look at all possibles solutions.

4.1.3 Tree

The following example is a larger example to test the static inference tool. It uses interfaces and inheritance and has the following additional dependencies, which are not seen in the class diagram:

- The `SortedTreeNode` class is used in all tree iterators and the sorted tree.
- The `Statistics` interface and all classes, which implement this interface are only used in the `Test` class.
- All tree iterator classes are used in the sorted tree.
- All classes are used in the `Test` class.

These dependencies give us a good order of the single queries in the global query rule. We always query a class after we query for the using class:

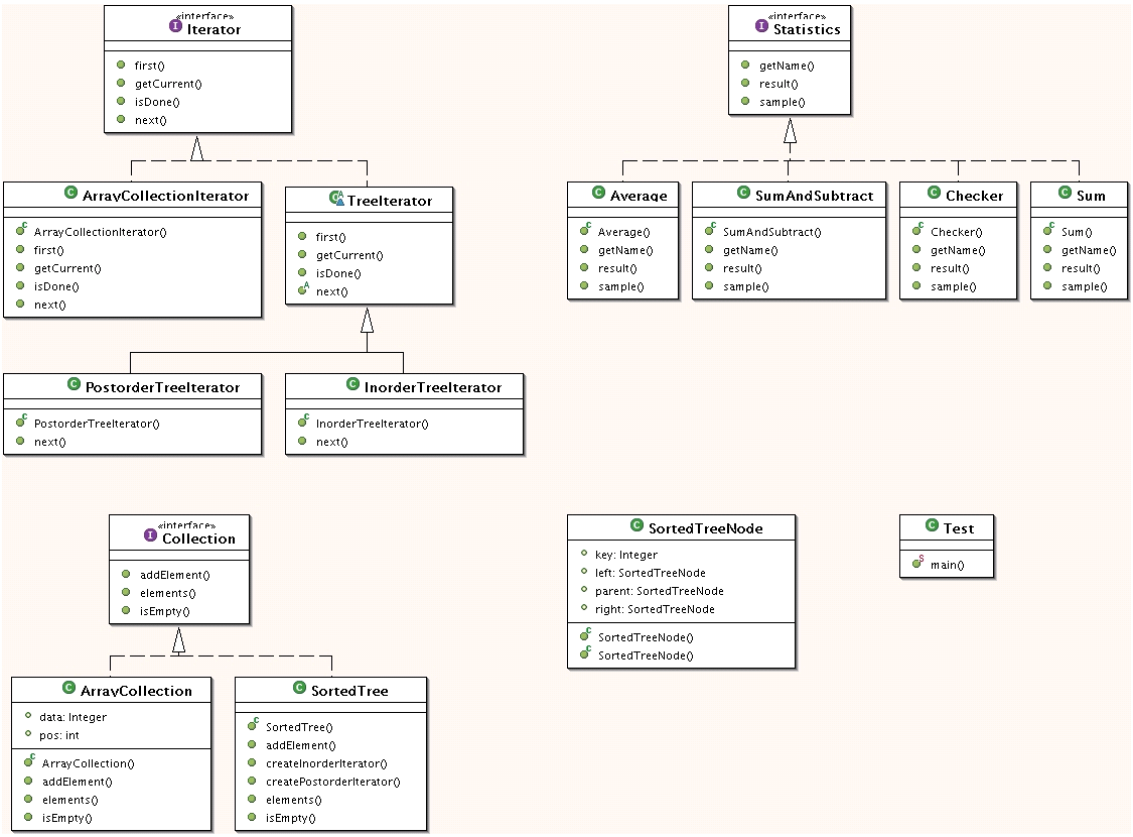

Figure 4.1: Class diagram for the tree example

1. Test
2. SortedTree
3. InorderTreeIterator
4. PostorderTreeIterator
5. TreeIterator
6. SortedTreeNode
7. Iterator
8. Collection
9. ArrayCollectionIterator
10. ArrayCollection
11. Checker
12. Average
13. SumAndSubtract
14. Sum
15. Statistics

The static inference tool returns the expected solution. A tree owns its nodes and is in the same context as the tree iterator. The tree iterator can only read the nodes of a tree. The elements of the tree nodes are in another context. The array collection also owns its elements, which are

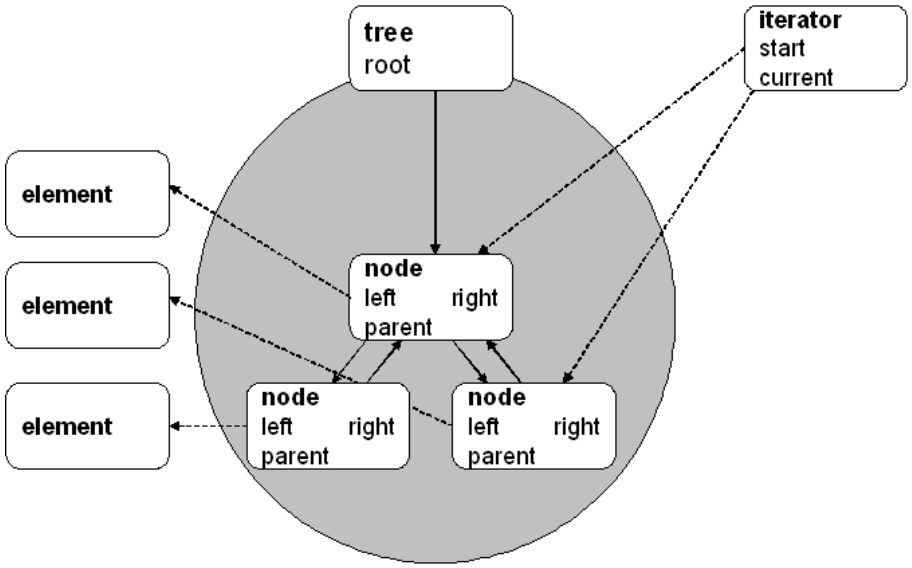

Figure 4.2: Contexts of the tree example

in an arbitrary context and the array collection iterator can only read the elements. The fully annotated source code is given in the appendix in A.2.

The static inference tool looked at 3 solutions and picked the first solution as the best one. Both heuristics agreed in the first step, that the first solution is the best one of the 3 drawn solutions.

Step of the Tool	*Time needed*
parse classes	25.965 s
read annotations	1.486 s
generate constraints	0.358 s
type inference	28.125 s
write annotation solution	1.559 s

We do not look at the solution, which can be found when enabling conflicts, because we already found the solution we expected and the next examples will show, that we cannot find a better solution with enabling conflicts then.

4.1.4 Linked List

The linked list example is taken from [9] and implements a singly linked list, where the list items create new items to store new objects. The method `Object.equals(Object)` is declared *pure* with a *readonly* parameter as before and the method `ListItem.contains(Object)` is also declared *pure* in the annotation input.

We order the query calls in the global query in the following user defined way, because the items are used within the list and the list is used by the user.

1. LinkedListUser
2. LinkedList
3. ListItem

Solution without Conflicts

```
public class ListItem {
        readonly Object stored;
        peer ListItem next;
```

```
        public String name;

        ListItem(readonly Object toStore){
                stored = toStore;
                next = null;
        }

        public peer ListItem getNextItem(){
                return next;
        }

        public void insert(readonly Object toStore){
                if(next == null){
                        next = new peer ListItem(toStore);
                        next.name = "item";
                }else{
                        next.insert(toStore);
                }
        }

        public readonly Object remove(int index){
                if(next == null){
                        return null;
                }else if(index == 0){
                        readonly Object ret = next.stored;
                        next = next.next;
                        return ret;
                }else{
                        return next.remove(index-1);
                }
        }

        public readonly Object remove(readonly Object o){
                if(next == null){
                        return null;
                }else if(o.equals(next.stored)){
                        readonly Object ret = next.stored;
                        next = next.getNextItem();
                        return ret;
                }else{
                        return next.remove(o);
                }
        }

        public pure boolean contains(readonly Object o){
                if(o.equals(stored)){
                        return true;
                }else{
                        if(next == null){
                                return false;
                        }else{
                                return next.contains(o);
                        }
                }
        }
}

public class LinkedList {
        rep ListItem head;
        int size;

        public LinkedList(){
                head = null;
```

```
            size = 0;
        }

        public int size(){
            return size;
        }

        public void insert(readonly Object o){
            if(head == null){
                head = new rep ListItem(o);
            }else{
                head.insert(o);
            }
            size++;
        }

        public boolean contains(readonly Object o){
            if(head != null){
                return head.contains(o);
            }else{
                return false;
            }
        }

        public readonly Object remove(int index){
            if(head == null || index < 0){
                return null;
            }else if(index == 0){
                readonly Object ret = head.stored;
                head = head.next;
                return ret;
            }else{
                return head.remove(index-1);
            }
        }

        public readonly Object remove(readonly Object o){
            if(head == null){
                return null;
            }else if(o.equals(head.stored)){
                readonly Object ret = head.stored;
                head = head.next;
                size--;
                return ret;
            }else{
                readonly Object ret = head.remove(o);
                if(ret != null){
                    size--;
                }
                return ret;
            }

        }
}

public class LinkedListUser {
        public static void main(readonly readonly String[] args) {
            peer LinkedList list = new peer LinkedList();
            for(int i=0; i<1000; i++){
                if(i%2 == 0){
                    list.insert(new peer A());
                }else{
                    list.insert(new peer B());
```

```
                }
            }
            for(int i=0; i<400; i++){
                if(i%10 != 0){
                    if(i%21 == 0){
                        for(int j=0; j<(i%10); j++){
                            list.remove(i);
                        }
                    }
                }else{
                    list.insert(new peer A());
                }
            }
        }
}
```

The solution is as expected for a linked list and the same as found in [9]. The next pointers of the items are all in the same context. The elements are in another context and the list owns its nodes. All parameters of the methods `insert`, `remove`, `contains` of the `ListItem` and the `LinkedList` class are mapped to *readonly*. The `ListItem.getNextItem` method returns *peer*, so the `ListItem.remove` method can operate on an item. The `ListItem.getNextItem` method is called in the `ListItem.remove` method over the next pointer of the item and assigns it to the same next pointer again.

```
next = next.getNextItem();
```

As the called method is not *pure*, the next pointer needs to be *peer* and because of the assignment to the *peer* next pointer, the return type of the method must also be *peer*.

The static inference tool looked at all possible 718 solutions and returned the second as the best one. The second solution has the highest score within the score system, although the first solution has less peer than the second one. The type inference step needed more time than in the other list example, but less than the parsing of all used libraries.

Step of the Tool	*Time needed*
parse classes	22.595 s
read annotations	1.483 s
generate constraints	0.212 s
type inference	14.742 s
write annotation solution	1.737 s

Solution with Conflicts

```
public class ListItem {
    ...

    public readonly ListItem getNextItem(){
        return next;
    }

    ...

    public peer Object remove(int index){
        ...
    }

    public readonly Object remove(readonly Object o){
        if(next == null){
            return null;
        }else if(o.equals(next.stored)){
            readonly Object ret = next.stored;
```

```
                    next = (peer ListItem ) next.getNextItem();
                    return ret;
                }
                ...
        }

        ...
}

public class LinkedList {
        ...

        public rep Object remove(int index){
                if(head == null || index < 0){
                    return null;
                }else if(index == 0){
                    readonly Object ret = head.stored;
                    head = head.next;
                    return ret;
                }else{
                    return head.remove(index-1);
                }
        }

        ...
}
```

The solution with conflicts differs in the following annotations from the one without:

1. The `ListItem.getNextItem` method returns now *readonly*. But when the method is called in the `ListItem.remove(Object)` method, the actual return type is casted to *peer*. This cast is not useful, because it is as we would not use a cast at all and just set the return type of the `ListItem.getNext` method to *peer* like before.

2. The `ListItem.remove(int)` method returns now *peer*. This is the price that has to be payed that the `LinkedList.remove(int)` method returns *rep*. This solution scores less than the solution without conflicts.

3. The `LinkedList.remove(int)` method returns now *rep*

For this example it is not useful to allow conflicts, because we get not the solution we wished. But as we have seen, we already get the wished solution without conflicts and therefore we do not need to look at conflicts at all, because we do not want to allow any additional casts in this program.

The static inference tool looked at 3 solutions and picked the second one, which has less peer and a higher score than the first one.

Step of the Tool	*Time needed*
parse classes	22.618 s
read annotations	1.482 s
generate constraints	0.196 s
type inference	2.654 s
write annotation solution	0.930 s

4.1.5 Lottery

The lottery example is also taken from [9] to compare the results of the static and the runtime inference. A person obtains a lottery ticket and chooses a number. When the lottery is drawn, the person has to check the winning number.

The methods `Lottery.getWinner`, `Person.check`, `WinningNumber.isWinner` and `Object.equals` are denoted as *pure* methods in the annotation input.

The query of the classes are ordered by their dependencies:

1. LotteryGame
2. Person
3. Lottery
4. WinningNumber
5. LotteryTicket

Solution without Conflicts

```
public class LotteryTicket {
        int chosenNumber;
}

public class WinningNumber {
        int number;

        public pure boolean isWinner(readonly LotteryTicket ticket){
                if(ticket.chosenNumber == number){
                        return true;
                }else{
                        return false;
                }
        }
}

public class Lottery {
        rep WinningNumber winner;
        public peer LotteryTicket getNewTicket(){
                return new peer LotteryTicket();
        }
        public void draw(){
                winner = new rep WinningNumber();
                winner.number = 123456;
        }
        public pure rep WinningNumber getWinner(){
                return winner;
        }
}

public class Person {
        peer LotteryTicket ticket;
        readonly Lottery lottery;
        public void play(peer Lottery lottery){
                this.lottery = lottery;
                ticket = lottery.getNewTicket();
                ticket.chosenNumber = 123456;
        }

        public pure boolean check(){
                return lottery.getWinner().isWinner(ticket);
        }
}

public class LotteryGame {

        public static void main(readonly readonly String[] args) {
```

```
            peer Lottery  lottery  = new peer Lottery();
            peer Person person = new peer Person();
            person.play( lottery );
             lottery .draw();
            if (person.check()){
                    System.out. println ("I've won!");
            }else{
                    System.out. println ("I've lost !" );
            }

        }

}
```

Our architecture finds excactly the same annotation solution as with runtime inference in [9].

The static inference tool looked at 16 solutions and picked the first one, which has the least number of peer, although the second solution scores higher. But the threshold is not reached for this example.

Step of the Tool	*Time needed*
parse classes	22.767 s
read annotations	1.530 s
generate constraints	0.105 s
type inference	0.256 s
write annotation solution	0.948 s

Solution with Conflicts

This solution differs in the following annotations:

1. The `winner` field of the lottery class is now *readonly*, but casted to *rep* in the `Lottery.draw()` and the `Lottery.getWinner()` method. The `Lottery.getWinner()` method still returns *rep*.
2. The `Lottery.getNewTicket()` method now returns *readonly*, but the call of the method is casted to *peer*.

Both casts should be omitted. The solution found without allowing conflicts is much better. There is no need in allowing conflicts, because this program does not need any conflicts and we can only get worse solutions with less type restrictions. Whenever we allow conflicts, we actually need less type restrictions to get a solution. This is unnecessary for this program.

The static inference tool looked at 30 solutions and picked the 28th one, which has less peer, although the fourth has the highest score. But the threshold is not reached and therefore the solution with the least number of peer returned.

Step of the Tool	*Time needed*
parse classes	23.663 s
read annotations	1.520 s
generate constraints	0.104 s
type inference	0.401 s
write annotation solution	0.904 s

If we do not use the less peer heuristic, the fourth solution is picked as the best one and this is nearly the same solution as without conflicts. No additional cast has to be inserted. The only difference is, that the method `Lottery.getWinner()` returns *readonly*, which is possible, because we only call a *pure* method on the return type of the call to the `Lottery.getWinner()` method. This is a case, where the less peer heuristic is not helpful and should not be used.

4.1.6 Golf Driver

This is the last example analyzed in [9]. A car has an engine, which can be started and stopped. A golf is also a car, which can be driven. The golf driver describes how the golf is driven. The golf class is derived from the car class and uses the golf driver class. This gives us the following user defined order of the query calls in the global query rule:

1. Golf
2. GolfDriver
3. Car
4. Engine

Solution without Conflicts

```
public class Car {
        rep Engine engine;
        public Car(){
                engine = new rep Engine();
                engine.isRunning = false;
        }
        public void start (){
                engine.isRunning = true;
        }
        public void stop(){
                engine.isRunning = false;
        }
}

public class GolfDriver {
        peer Golf golf;
        public void driveGolf (){
                golf.enter(this);
                golf.start ();
                golf.stop ();
        }
        public static void main(readonly readonly String [] args){
                peer Golf mycar = new peer Golf();
                peer GolfDriver me = new peer GolfDriver();
                me.golf = mycar;
                me.driveGolf ();
        }
}

public class Golf extends Car {
        readonly GolfDriver driver;
        public void enter(readonly GolfDriver driver){
                this.driver = driver;
        }
}
```

We get the same solution as in [9] except for the annotations of the golf, where both variables were annotated with *peer* in [9]. This is an example, where the solution found by runtime solution is better than the one found with our architecture. The car and its driver should be in the same context for they can interact with each other. As we never access the `golf` field over the `driver` in the `Golf` class, we do not annotate the `driver` field with *peer*.

The static inference tool looked at all possible 12 solutions and picked the first one, which has less peer and the highest score, although the threshold is not reached.

Step of the Tool	*Time needed*
parse classes	22.601 s
read annotations	1.533 s
generate constraints	0.120 s
type inference	0.178 s
write annotation solution	0.832 s

Solution with Conflicts

The solution found with allowing conflicts is worse than the one before. Again this program does not need any conflicts and we cannot get a better solution with less type restrictions. The engine of the car is now *readonly* and casted in each updating field access, which is not useful. The local variable `me` in the `main` method is also *readonly* now and needs to be casted for each method invocation.

The static inference tool looked at 30 solutions and picked the 28th one, which has less peer, although the 10th solution has the highest score. But the threshold is not reached.

Step of the Tool	*Time needed*
parse classes	22.664 s
read annotations	1.538 s
generate constraints	0.096 s
type inference	0.249 s
write annotation solution	0.939 s

If we omit the less peer heuristic, we get the 10th solution returned by the score heuristic. This solution does not annotate the local variable `me` of the main method with *readonly*, but does also annotate the engine of the car with *readonly* and cast it whenever it is used. We do not get the same solution than without using conflicts when not using the less peer heuristic.

4.1.7 Conclusion

The static inference tool worked for the small and for one medium example and is fast enough. In all the examples it was not much slower than the parsing of all libraries by the JML compiler. All examples except for the medium one were even much faster than the parsing. The right order of the query calls in the global query body can help to gain performance.

Conflicts should only be enabled, when the solution found without conflicts is not the expected one or if the programmer knows, that the design will must probably need conflicts to work out. Otherwise conflicts should be omitted, because they allow less restrictive type relations and lead to larger solution sets. The annotation solution found will then most probably insert a cast, where no one is needed as we have seen in the examples.

If all classes of a program are used in the intended way, better annotation solutions can be found. Therefore it is very important to annotate only classes with the static inference tool, which are fully implemented and use all their variables in all allowed ways.

4.2 Complexity

We want to show the complexity of the static type inference in this section. The two parts, which influence the complexity are:

1. the solver and
2. the heuristics

The solver is the important factor if we have to detect that there is no possible solution and the heuristics determine the complexity if all possibilities work out for a program.

There are 3 possible Universe types to annotate each variable in a program. If a program has `n` variables, there are 3^n possibilities to try out for a program until the solver notices that there is no possible solution.

$$\boxed{\text{No annotation possible} = O(3^n)} \tag{4.1}$$

If a program is as less restrictive that all Universe annotations are possible solutions for all variables, the set of all possible solutions grows to 3^n, which is just the number of all possibilities the solver can check as explained before. Each heuristic looks at all `n` variables of a solution. If we have to check `n` variables for all 3^n possible solutions, we get the following complexity for this worst case:

$$\boxed{\text{All annotations possible} = O(3^n * n)} \tag{4.2}$$

The two worst cases should be omitted, because they mean a loss in performance:

1. detect that there is no solution
2. find best solution if all annotations are possible

The first case is not as easy to circumvent. Normally there is an annotation solution for each program, as long as we are not using partly annotated programs and be careful with exceptions and pure methods. Exception objects in a throw- or catch-clause are always *readonly* by default and are not allowed to call a non-pure method.

The second case can be circumvented if the source is fully implemented and uses all its variables in the appropriate manner. If a user really wants the annotation of a program, which has no real type restrictions, there is no need to apply heuristics and the first solution returned by the solver should already be the best one.

4.3 Comparison to Runtime Inference

The benefits of the static inference compared to the runtime inference project of [9] are the following:

1. *Annotate larger subset*
   With static inference we are able to annotate all variables of a program, also those, whose annotation cannot be determined easily by runtime inference:
   - local variables
   - arrays
   - static fields
   - static methods
2. *Partially annotated programs*
   The user can directly annotate variables in the program or give the annotations in an additional XML input file. The given annotations are inserted easily in the generated constraints.
3. *Dereferencing chains*
   The Prolog program applies the Universe combination rule to a chain of field accesses. The appropriate Prolog `field_access_left` rule assures, that the write access after a dereferencing chain is legal.
4. *Time Consumption*
   In our examples the static type inference can be much faster than the runtime inference. This can be seen on the linked list example from [9], which needed over 20 minutes in the runtime inference, where the static type inference needed less than a minute. The runtime inference needs the most time to build up the needed object graph. The inference step is also fast enough.

Code coverage and input values are very important to get a good solution with runtime inference. The user must provide a good usage of the program he wants to annotate for the best solution can be found. To find a good solution with static type inference, it is more important that the program to annotate is fully implemented with all allowed possibilities and uses all declared variables in the intended way.

The runtime inference always finds the solution with the deepest nested universes, but our architecture may not return the best possible solution. We cannot assure that the expected solutions are returned first by the Prolog system and that the heuristics always grab the solution we wish. But we have the possibility to exchange the solver and experiment with a whole new solver system, which may have more influence on the order of possible solutions.

Another problem with our solution is, that we cannot guarantee, that all casts of a program work out. Annotations found for casts the user inserted in the original source and for additional casts we insert because of occurring conflicts may fail at runtime. As long as we are not using conflicts and the user inserts no casts in the original source the annotation found is always compilable and runnable.

In the future it is best to combine the static and the runtime inference to find a good annotation solution. The runtime inference should be used to infer the annotations of fields, parameters and return types. These annotations are then used as input for the static inference, which will annotate the method bodies. Like this the benefits of both designs can be used.

4.4 Future Work

4.4.1 Inner Classes

The Universe type rules for inner classes are not yet implemented in the Prolog database. Inner classes are parsed and we also generate constraints for them, but we do not apply the additional Universe type rules for inner classes. The Prolog database has to be extended in the future for inner classes are also handled correctly and has to be adapted to all changes of the Universe type system, which take place after this project.

4.4.2 Testing

In the future we need to test all possible configurations of the static inference tool to find the best one. We need to experiment with the already implemented heuristics and optimizations and possibly develop new ones.

We have to check, which optimisation strategy is best for larger programs:

- Does the arrangement of the rules in a query gain much performance?
- How are the query calls in the global query body ordered best?

We provided some solutions for these two questions, but further tests are needed.

We also need to experiment with more heuristics to find out, how we can get the best solution automatically. The interaction of the heuristics also needs further testing. As we have seen in the examples, the score system sometimes finds a better solution, but has no influence, because the threshold is not fulfilled. The less peer heuristic has to be carefully used when allowing conflicts, because when a solution has less *peer* than it will probably add more casts.

4.4.3 User Interaction

Another improvement is to allow the user to interact with the architecture while it is running. The user should be able to look at solutions found by the architecture and has to decide, if the

solution found is already the expected one. If this is not the case, the user can get another solution. We should be able to change heuristics at runtime if we do not get the expected solution with a heuristic. Further the user should be allowed to insert annotations during the static type inference to see how the solution changes and if there is still a solution found at all.

As is proposed in the work of Jonathan Aldrich, we can also allow the solver to ask for an annotation if a certain period of unsuccessful searches is elapsed. This would require an extended solver program, which is not just a database to model the Universe type system, but also observes its own runtime behavior.

The work of Jonathan Aldrich [4] is about type inference for object oriented languages. It shows a possible implementation of a constraints solver in Prolog, which resembles our Prolog database approach. The work talks about the problems with Prolog as a solver as for example that the recursive descent techniques to solve constraints can be a loss in performance. It also proposes the usage of heuristics to approximate a good solution. There is also a user interaction with type inference proposed, where the type inference is paused after a certain period of unsuccessful searches and the user is asked to add an annotation.

4.5 Conclusion

We showed with the results presented in this section, that the goal of the project was achieved. We developed and implemented a modular architecture, which is able to infer the Universe types of all used variables correctly. If there is an annotation solution, our architecture can find one in reasonable short time.

Our implemented architecture is modular and can be configured with additional heuristics and Prolog optimizations. To grant also a short runtime for the type inference of large programs further testing with Prolog optimizations is needed. It is also necessary to experiment with more heuristics and find a way to assure that the expected solution is found with a heuristic.

It is also possible to exchange the Prolog database and to apply another type system to the generated constraints of a given Java source or to implement a new database if the Universe type system changes. In the future it is also a good idea to experiment with another solver system than the Prolog system.

Our architecture will be used together with the runtime inference algorithm of [9] to generate feasible annotations for Java programs. The annotation solutions found by automatic type inference will further be compared to manual annotations for Java patterns.

Bibliography

[1] Peter Müller and A. Poetzsch-Heffter: Universes: A Type System for Alias and Dependency Control. *Technical Report, Fernuniversität Hagen*, 2001.

[2] Werner Dietl and Peter Müller: Universes: Statically-checkable ownership for JML. *Journal of Object Technology*, 2005.

[3] Dietl, W. and Müller, P.: Exceptions in Ownership Type Systems, *Formal Techniques for Java-like Programs*, 2004.

[4] Jonathan Aldrich: Incremental Type Inference for Software Engineering. *University of Washington*, December 1997.

[5] Peter Müller: Modular Specification and Verification of Object-Oriented Programs, volume 2262 of *Lecture Notes in Computer Science*, Springer Verlag, 2002.

[6] Jan Wielemaker: SWI-Prolog Reference Manual, 5.4.3 ed. http://www.swi-prolog.org/.

[7] Jakarta Project. XMLBeans. Available from http://xmlbeans.apache.org/.

[8] Retroweaver. Available from http://retroweaver.sourceforge.net/.

[9] Frank Lyner: Runtime Universe Type Inference, http://www.sct.inf.ethz.ch/projects/student_docs/Frank_Lyner/, 2005. Master Thesis.

[10] Marco Meyer: Interaction with ownership graphs, http://www.sct.inf.ethz.ch/projects/student_docs/Marco_Meyer/, 2005, Semester Project.

Appendix A

Examples

A.1 Iterator Example

In this example, the identifier 'L' as well as the packagename in the variable names are omitted for the original variable names can still be recognised. The longer names for methods are not used either for the same reason. The solution is just the first solution, the Prolog system finds.

The following files would be generated for the class Node, LinkedList and Iter:

A.1.1 query_Node.pl

```
query_Node([Node_next, Node_elem, Node_prev]) :-
        field_declaration (none, [Node_elem]),
        field_declaration (none, [Node_next]),
        field_declaration (none, [Node_prev]).
```

A.1.2 query_LinkedList.pl

```
query_LinkedList ([Node_elem, Node_prev, LinkedList_equals_Local_f1 ,
                   LinkedList_set_FormalParameter1 , LinkedList_set_FormalParameter0 ,
                   LinkedList_set_Local_n , LinkedList_addElement_FormalParameter0,
                   Node_next, LinkedList_addElement_Local_n , LinkedList_equals_Cast0 ,
                   LinkedList_equals_Local_f2 , LinkedList_first ,
                   LinkedList_addElement_New0]) :-

      assignment ([[ LinkedList_set_Local_n ]],
                  [[ LinkedList_set_FormalParameter0 ]]),
      assignment ([[ LinkedList_set_Local_n ], [ Node_elem]],
                  [[ LinkedList_set_FormalParameter1 ]]),
      parameter_declaration (none, none, [ LinkedList_set_FormalParameter0 ]),
      parameter_declaration (none, none, [ LinkedList_set_FormalParameter1 ]),
      locavariable_declaration (none, [ LinkedList_set_Local_n ]),
      new(none, [LinkedList_addElement_New0]),
      assignment ([[ LinkedList_addElement_Local_n ]],
                  [[ LinkedList_addElement_New0]]),
      assignment ([[ LinkedList_addElement_Local_n ], [ Node_elem]],
                  [[ LinkedList_addElement_FormalParameter0]]),
      assignment ([[ LinkedList_addElement_Local_n ], [ Node_next]],
                  [[ this ], [ LinkedList_first ]]),
      assignment ([[ this ], [ LinkedList_first ]],
                  [[ LinkedList_addElement_Local_n ]]),
      assignment ([[ this ], [ LinkedList_first ], [ Node_next], [Node_prev]],
                  [[ this ], [ LinkedList_first ]]),
      assignment ([[ LinkedList_equals_Local_f1 ]], [[ this ], [ LinkedList_first ]]),
      cast_right (none, [ LinkedList_equals_Cast0 ], [[ readonly ]]),
      assignment ([[ LinkedList_equals_Local_f2 ]],
                  [[ LinkedList_equals_Cast0 ], [ LinkedList_first ]]),
      boolean_expression ([[ LinkedList_equals_Local_f1 ]], [[ null ]]),
      boolean_expression ([[ LinkedList_equals_Local_f2 ]], [[ null ]]),
      boolean_expression ([[ LinkedList_equals_Local_f1 ], [ Node_elem]],
                  [[ LinkedList_equals_Local_f2 ], [ Node_elem]]),
      assignment ([[ LinkedList_equals_Local_f1 ]],
                  [[ LinkedList_equals_Local_f1 ], [ Node_next]]),
      assignment ([[ LinkedList_equals_Local_f2 ]],
                  [[ LinkedList_equals_Local_f2 ], [ Node_next]]),
```

```
boolean_expression ([[ LinkedList_equals_Local_f1 ]], [[ null ]]),
boolean_expression ([[ LinkedList_equals_Local_f2 ]], [[ null ]]),
parameter_declaration (none, public,
        [LinkedList_addElement_FormalParameter0]),
locavariable_declaration (none, [LinkedList_addElement_Local_n]),
locavariable_declaration (none, [ LinkedList_equals_Local_f1 ]),
locavariable_declaration (none, [ LinkedList_equals_Local_f2 ]),
field_declaration (none, [ LinkedList_first ]).
```

A.1.3 query_Iter.pl

```
query_Iter ([Node_next, LinkedList_first , Iter_pos , Iter_list ,
            LinkedList_set_FormalParameter1,
            Iter_setValue_FormalParameter0,
            LinkedList_set_FormalParameter0,
            Iter_Iter_FormalParameter0 ]) :-

    assignment([[ this ], [ Iter_list ]], [[ Iter_Iter_FormalParameter0 ]]),
    assignment([[ this ], [ Iter_pos ]],
            [[ Iter_Iter_FormalParameter0 ], [ LinkedList_first ]]),
    parameter_declaration (none, constructor , [ Iter_Iter_FormalParameter0 ]),
    method_calparameters_right ([[ this ], [ Iter_list ]],
            [[[ this ], [ Iter_pos ]], [[ Iter_setValue_FormalParameter0 ]]],
            [[ LinkedList_set_FormalParameter0 ],
                [ LinkedList_set_FormalParameter1 ]],
            none),
    assignment([[ this ], [ Iter_pos ]], [[ this ], [ Iter_pos ], [ Node_next]]),
    parameter_declaration (none, public , [ Iter_setValue_FormalParameter0 ]),
    field_declaration (none, [ Iter_list ]),
    field_declaration (none, [ Iter_pos ]).
```

A.1.4 query.pl

```
query([Node_elem, Node_prev, LinkedList_equals_Local_f1 ,
            LinkedList_set_FormalParameter1,
            LinkedList_addElement_FormalParameter0, LinkedList_set_Local_n ,
            LinkedList_set_FormalParameter0,
            Iter_setValue_FormalParameter0 , Node_next,
            LinkedList_addElement_Local_n , LinkedList_equals_Cast0 , Iter_pos ,
            LinkedList_equals_Local_f2 , LinkedList_first , Iter_list ,
            Iter_Iter_FormalParameter0 , LinkedList_addElement_New0]) :-

    query_Iter ([Node_next, LinkedList_first , Iter_pos , Iter_list ,
            LinkedList_set_FormalParameter1 , Iter_setValue_FormalParameter0 ,
            LinkedList_set_FormalParameter0 , Iter_Iter_FormalParameter0 ]),
    query_LinkedList ([Node_elem, Node_prev,
            LinkedList_equals_Local_f1 , LinkedList_set_FormalParameter1 ,
            LinkedList_set_FormalParameter0 , LinkedList_set_Local_n ,
            LinkedList_addElement_FormalParameter0,
            Node_next, LinkedList_addElement_Local_n,
            LinkedList_equals_Cast0 ,
            LinkedList_equals_Local_f2 , LinkedList_first ,
            LinkedList_addElement_New0]),
    query_Node([Node_next, Node_elem, Node_prev]).
```

A.1.5 Output by the Prolog System

The call of the query to the Prolog System looks like this:

```
?- query([Node_elem, Node_prev, LinkedList_equals_Local_f1 ,
            LinkedList_set_FormalParameter1,
            LinkedList_addElement_FormalParameter0, LinkedList_set_Local_n ,
            LinkedList_set_FormalParameter0,
            Iter_setValue_FormalParameter0 , Node_next,
            LinkedList_addElement_Local_n , LinkedList_equals_Cast0 , Iter_pos ,
            LinkedList_equals_Local_f2 , LinkedList_first , Iter_list ,
            Iter_Iter_FormalParameter0 , LinkedList_addElement_New0]).
```

And the Prolog system answers with the following solution:

```
Node_elem = readonly
Node_prev = readonly
LinkedList_equals_Local_f1 = readonly
```

```
LinkedList_set_FormalParameter1 = readonly
LinkedList_addElement_FormalParameter0 = readonly
 LinkedList_set_Local_n  = peer
LinkedList_set_FormalParameter0 = peer
Iter_setValue_FormalParameter0 = readonly
Node_next = peer
LinkedList_addElement_Local_n = peer
 LinkedList_equals_Cast0  = rep
 Iter_pos  = peer
 LinkedList_equals_f2  = rep
  LinkedList_first  = peer
  Iter_list  = peer
 Iter_Iter_FormalParameter0  = peer
LinkedList_addElement_New0 = peer
```

A.2 Tree Iterator Example

A.2.1 Statistics

```
interface  Statistics  {
        public void sample(readonly Integer  i );

        public readonly Object  result ();

        public  String  getName();
}
```

A.2.2 Average

```
public  class  Average implements Statistics {

        private  int  sum;
        private  int  counter;

        public  Average() {
                sum = 0;
                counter = 0;
        }

        public void sample(readonly Integer  i ) {
                sum += i.intValue();
                counter++;
        }

        public readonly Object  result () {
                return new rep Double(sum / (double) counter);
        }

        public  String  getName() {
                return "Average";
        }
}
```

A.2.3 Sum

```
public  class  Sum implements Statistics {

        private  int  sum;

        public  Sum() {
                sum = 0;
        }

        public void sample(readonly Integer  i ) {
                sum += i.intValue();
        }

        public readonly Object  result () {
                return new rep Integer(sum);
        }

        public  String  getName() {
```

```
            return "Sum";
        }
}
```

A.2.4 SumAndSubtract

```
public class SumAndSubtract implements Statistics {

        private int sum;
        private int operator;

        public SumAndSubtract() {
                sum = 0;
                operator = 1;
        }

        public void sample(readonly Integer i) {
                sum += operator * i.intValue();
                operator -= operator;
        }

        public readonly Object result () {
                return new rep Integer(sum);
        }

        public String getName() {
                return "SumAndSubtract";
        }
}
```

A.2.5 Checker

```
public class Checker implements Statistics {

        private boolean found;
        private readonly Integer element;

        public Checker(readonly Integer val) {
                element = val;
                found = false;
        }

        public void sample(readonly Integer i) {
                if (i.equals(element)) {
                        found = true;
                }
        }

        public readonly Object result () {
                return new rep Boolean(found);
        }

        public String getName() {
                return "Checker";
        }
}
```

A.2.6 Collection

```
interface Collection {
        public peer Iterator elements();

        public void addElement(readonly Integer i);

        public boolean isEmpty();
}
```

A.2.7 Iterator

```
import java.util.NoSuchElementException;
```

```
interface Iterator {

        public void first ();

        public void next() throws NoSuchElementException;

        public readonly Integer getCurrent();

        public boolean isDone();
}
```

A.2.8 ArrayCollection

```
public class ArrayCollection implements Collection {

        public rep readonly Integer [] data;
        public int pos;
        private static final int SIZE = 64;

        public ArrayCollection () {
                data = new rep readonly Integer[SIZE];
                pos = 0;
        }

        public void addElement(readonly Integer i) {
                if (pos >= data.length) {
                        rep readonly Integer [] oldData = data;
                        data = new rep readonly Integer[data.length*2];
                        for(int j = 0; j<oldData.length; j++) {
                                data[j] = oldData[j];
                        }
                        data[pos++] = i;
                }
        }

        public boolean isEmpty() {
                return (pos == 0);
        }

        public peer Iterator elements() {
                return new peer ArrayCollectionIterator (this);
        }
}
```

A.2.9 ArrayIterator

```
import java.util.NoSuchElementException;

public class ArrayCollectionIterator implements Iterator {
        private int pos;
        private readonly ArrayCollection coll;

        public ArrayCollectionIterator (readonly ArrayCollection coll) {
                this.coll = coll;
        }

        public void first () {
                pos = 0;
        }

        public void next() throws NoSuchElementException {
                if (pos < coll.data.length -1) {
                        pos++;
                }
                else {
                        throw new NoSuchElementException(
                                " ArrayCollectionIterator ");
                }
        }

        public readonly Integer getCurrent() {
                return coll.data[pos];
        }
```

```
        public boolean isDone() {
                return (pos >= coll.pos ) || ( pos >= coll.data.length );
        }
}
```

A.2.10 SortedTreeNode

```
public class SortedTreeNode {

        public readonly Integer key;

        public peer SortedTreeNode left;

        public peer SortedTreeNode right;

        public readonly SortedTreeNode parent;

        public SortedTreeNode(readonly Integer i) {
                key = i;
                this.parent = null;
        }

        public SortedTreeNode(readonly Integer i, readonly SortedTreeNode parent) {
                key = i;
                this.parent = parent;
        }
}
```

A.2.11 SortedTree

```
public class SortedTree implements Collection {
        protected rep SortedTreeNode rootNode;

        public SortedTree() {
                rootNode = null;
        }

public boolean isEmpty() {
        return (rootNode == null);
}

public void addElement(readonly Integer i) {
        if (rootNode == null) {
                rootNode = new rep SortedTreeNode(i);
        }
        else if (i.intValue() <=rootNode.key.intValue()) {
                insertInLeftSubtree (i, rootNode);
        }
        else {
                insertInRightSubtree (i, rootNode);
        }
}

public peer Iterator elements() {
        return createInorderIterator ();
}

public rep Iterator createPostorderIterator () {
        return new rep PostorderTreeIterator(rootNode);
}

public peer Iterator createInorderIterator () {
        return new peer InorderTreeIterator(rootNode);
}

private void insertInLeftSubtree (readonly Integer i,
                rep SortedTreeNode node) {
        rep SortedTreeNode leftChild = node.left;
        if ( leftChild == null) {
                rep SortedTreeNode newnode =
                        new rep SortedTreeNode(i, node);
                node.left = newnode;
        }
        else if (i.intValue() <= leftChild.key.intValue()) {
```

```
                insertInLeftSubtree (i, leftChild);
            }
            else {
                insertInRightSubtree (i, leftChild);
            }
        }

        private void insertInRightSubtree (readonly Integer i,
                rep SortedTreeNode node) {
            rep SortedTreeNode rightChild = node.right;
            if (rightChild == null) {
                rep SortedTreeNode newnode =
                        new rep SortedTreeNode(i, node);
                node.right = newnode;
            }
            else if (i.intValue() <= rightChild.key.intValue()) {
                insertInLeftSubtree (i, rightChild);
            }
            else {
                insertInRightSubtree (i, rightChild);
            }
        }
}
```

A.2.12 TreeIterator

```
import java.util.NoSuchElementException;

abstract class TreeIterator implements Iterator {

    protected readonly SortedTreeNode currentNode;

    protected readonly SortedTreeNode startNode;

    public void first () {
        currentNode = startNode;
    }

    abstract public void next() throws NoSuchElementException;

    public readonly Integer getCurrent() {
        return currentNode.key;
    }

    public boolean isDone() {
        return (currentNode == null);
    }
}
```

A.2.13 InorderTreeIterator

```
import java.util.NoSuchElementException;

public class InorderTreeIterator extends TreeIterator {

    public InorderTreeIterator (readonly SortedTreeNode rootNode) {
        startNode = rootNode;
        if (rootNode != null) {
            while(startNode.left != null) {
                startNode = startNode.left;
            }
        }
        currentNode = startNode;
    }

    public void next() throws NoSuchElementException {
        if (currentNode == null) {
            throw new NoSuchElementException("InorderTreeIterator");
        }
        else if (currentNode.right != null) {
            currentNode = currentNode.right;
            while(currentNode.left != null) {
                currentNode = currentNode.left;
            }
```

```
                }
                else {
                        while((currentNode.parent != null)
                                && (currentNode.parent.left != currentNode)) {

                                currentNode = currentNode.parent;
                        }
                        currentNode = currentNode.parent;
                }
        }
}
```

A.2.14 PostorderTreeIterator

```
import java.util.NoSuchElementException;

public class PostorderTreeIterator extends TreeIterator {

        public PostorderTreeIterator (readonly SortedTreeNode rootNode) {
                startNode = rootNode;
                if (startNode != null) {
                        while((startNode.left != null)
                                || (startNode.right != null)) {
                                while(startNode.right != null) {
                                        startNode = startNode.right;
                                }
                                if (startNode.left != null) {
                                        startNode = startNode.left;
                                }
                        }
                }
                currentNode = startNode;
        }

        public void next() throws NoSuchElementException {
                if (currentNode == null) {
                        throw new NoSuchElementException("PostorderTreeIterator");
                }
                if (currentNode.parent == null) {
                        currentNode = currentNode.parent;
                        return;
                }
                if (currentNode.parent.right == currentNode) {
                        if (currentNode.parent.left != null) {
                                currentNode = currentNode.parent.left;
                        }
                        else {
                                currentNode = currentNode.parent;
                                return;
                        }
                        while((currentNode.left != null)
                                || (currentNode.right != null)) {
                                while(currentNode.right != null) {
                                        currentNode = currentNode.right;
                                }
                                if (currentNode.left != null) {
                                        currentNode = currentNode.left;
                                }
                        }
                }
                else {
                        currentNode = currentNode.parent;
                }
        }
}
```

A.2.15 Test

```
import java.util.Random;

public class Test {

        public static void main(readonly readonly String[] argsv) {
                peer Statistics s = null;
```

```
        if (argsv.length < 1) {
                return;
        }
        else if (argsv[0].equals("average")) {
                s = new peer Average();
        }
        else if (argsv[0].equals("sumsub")) {
                s = new peer SumAndSubtract();
        }
        else if (argsv[0].equals("search")) {
                s = new peer Checker(new peer Integer(
                        readonly Integer.parseInt(argsv[1])));
        }
        else {
                return;
        }
        peer Test.measure(100, s);
        measure(10000, s);
        measure(1000000, s);
}

private static void measure(int elements, peer Statistics s) {
        int i;
        long start, end;

        peer ArrayCollection array = new peer ArrayCollection();

        fill(array, elements);

        apply(array, s);

        peer SortedTree tree = new peer SortedTree();

        fill(tree, elements);

        apply(tree, s);
}

private static void apply(peer Collection coll, peer Statistics s) {
        peer Iterator iter = coll.elements();
        while(!iter.isDone()) {
                s.sample(iter.getCurrent());
                iter.next();
        }
}

private static void fill(peer Collection coll, int elements) {
        peer Random random = new peer Random();
        random.setSeed(42);
        for(int i = 0; i<elements; i++) {
                coll.addElement(new peer Integer(random.nextInt() % 100));
        }
}
}
```

Appendix B

Prolog Program

B.1 rules.pl


```
% File rules.pl

% types

% simple universe types

universe([rep]).
universe([readonly]).
universe([peer]).

% all array universe types
% only for arrays which contain a reference type

universe([rep, readonly]).
universe([rep, peer]).
universe([readonly, readonly]).
universe([readonly, peer]).
universe([peer, readonly]).
universe([peer, peer]).

% all types

type(X) :- universe(X).
type([primitive]).
type([null]).

% special type for 'this' objects

type([this]).

% universe subtypes

subtype_notequal([rep], [readonly]).
subtype_notequal([peer], [readonly]).

subtype_notequal([null], X) :- universe(X).
subtype_notequal([null], [this]).

subtype_notequal([rep, readonly], [readonly, readonly]).
subtype_notequal([rep, peer], [rep, readonly]).
subtype_notequal([rep, peer], [readonly, readonly]).
subtype_notequal([rep, peer], [readonly, peer]).
subtype_notequal([readonly, peer], [readonly, readonly]).
subtype_notequal([peer, readonly], [readonly, readonly]).
subtype_notequal([peer, peer], [readonly, readonly]).
subtype_notequal([peer, peer], [readonly, peer]).
subtype_notequal([peer, peer], [peer, readonly]).

subtype_notequal([this], [readonly]).


% variable subtypes
% x = y;
% [y] <: [x];

subtype_equal(X, Y) :- subtype_notequal(X, Y).
```

```
subtype_equal(X, X) :- type(X).
subtype_equal([ this ],[ peer ]).
subtype_equal([peer ], [ this ]).


% combinations

combination([rep ], [ rep ], [ readonly ]).
combination([rep ], [ readonly ], [ readonly ]).
combination([rep ], [ peer ], [ rep ]).

combination([readonly ], [ rep ], [ readonly ]).
combination([readonly ], [ readonly ], [ readonly ]).
combination([readonly ], [ peer ], [ readonly ]).

combination([peer ], [ rep ], [ readonly ]).
combination([peer ], [ readonly ], [ readonly ]).
combination([peer ], [ peer ], [ peer ]).

combination([X], [Y, Z], [U, Z]) :- combination([X], [Y], [U]).

combination([ this ], [ rep ], [ rep ]).
combination([ this ], [ readonly ], [ readonly ]).
combination([ this ], [ peer ], [ peer ]).

combination([ this ], [Y, Z], [U, Z]) :- combination([ this ], [Y], [U]).

combination([rep ], [ primitive ], [ primitive ]).
combination([readonly ], [ primitive ], [ primitive ]).
combination([peer ], [ primitive ], [ primitive ]).
combination([ this ], [ primitive ], [ primitive ]).

combination([X,Y], [ primitive ], [ primitive ]) :- universe([X,Y]).


% field declaration

% static reference fields are always readonly by now

field_declaration ( static , [ Field ]) :- universe([ Field ]),
        Field \== rep.
field_declaration ( static , [ FieldArray , FieldArrayEl ]) :-
        universe([ FieldArray , FieldArrayEl ]), FieldArray \== rep.
field_declaration (none, Field ) :- universe( Field ).

% heuristic
% if a field is tried to be peer and the preceding field
% is still rep, try first to make the preceding field
% readonly and the field rep again


% local variable declaration

% local variables in static methods must not be of type rep

local_variable_declaration ( static , [Local]) :- universe([Local ]), Local \== rep,
        Local \== conflict_rep .
local_variable_declaration ( static , [LocalArray , LocalEl]) :-
        universe([LocalArray , LocalEl ]), LocalArray \== rep,
        LocalArray \== conflict_rep .
local_variable_declaration (none, Local) :- universe(Local).


static_target ( static , [Type]) :- universe([Type]), Type \== rep,
        Type \== conflict_rep, Type \== conflict_peer.
static_target (none, [Type]) :- universe([Type]),
        Type \== conflict_rep, Type \== conflict_peer.


% methods

parameter_declaration(pure, _, [Parameter]) :-
        Parameter = readonly ; Parameter = conflict_rep ;
        Parameter = conflict_peer .
parameter_declaration(pure, _, [ParameterArray, ParameterArrayEl]) :-
        (ParameterArray = readonly ;
```

```
        ParameterArray = conflict_rep  ; ParameterArray = conflict_peer ),
        universe ([ParameterArray, ParameterArrayEl]).
parameter_declaration (none, constructor ,[Parameter]) :-
        universe ([Parameter]), Parameter \== rep,
        Parameter \== conflict_rep.
parameter_declaration (none, constructor ,[ParameterArray, ParameterArrayEl]) :-
        universe ([ParameterArray, ParameterArrayEl]), ParameterArray \== rep,
        ParameterArray \== conflict_rep.
parameter_declaration (none, static ,[Parameter]) :-
        universe ([Parameter]), Parameter \== rep,
        Parameter \== conflict_rep.
parameter_declaration (none, static ,[ParameterArray, ParameterArrayEl]) :-
        universe ([ParameterArray, ParameterArrayEl]),
        ParameterArray \== rep,
        ParameterArray \== conflict_rep.
parameter_declaration (none, public ,[Parameter]) :-
        universe ([Parameter]), Parameter \== rep.
parameter_declaration (none, public ,[ParameterArray, ParameterArrayEl]) :-
        universe ([ParameterArray, ParameterArrayEl]),
        ParameterArray \== rep.
parameter_declaration (none, none,Parameter) :-
        universe (Parameter).


% method declaration

method_declaration(pure, static , [ReturnType]) :-
        universe ([ReturnType]), ReturnType \== rep,
        ReturnType \== conflict_rep.
method_declaration(pure, static , [ReturnArray, ReturnArrayEl]) :-
        universe ([ReturnArray, ReturnArrayEl]), ReturnArray \== rep,
        ReturnArray \== conflict_rep.
method_declaration(pure, public , ReturnType) :-
        universe (ReturnType).
method_declaration(pure, none, ReturnType) :-
        universe (ReturnType).
method_declaration(none, static , [ReturnType]) :-
        universe ([ReturnType]), ReturnType \== rep,
        ReturnType \== conflict_rep.
method_declaration(none, static , [ReturnArray, ReturnArrayEl]) :-
        universe ([ReturnArray, ReturnArrayEl]),
        ReturnArray \== rep, ReturnArrayEl \== rep,
        ReturnArray \== conflict_rep , ReturnArrayEl \== conflict_rep .
method_declaration(none, public , ReturnType) :-
        universe (ReturnType).
method_declaration(none, none, ReturnType) :-
        universe (ReturnType).


% object creation
% v = new TS();
% TS <: [v]

new(static , [ Variable ]) :- universe ([ Variable ]),
        Variable \== readonly, Variable \== rep,
        Variable \== conflict_rep , Variable \== conflict_peer .
new(none, [Variable ]) :- universe ([ Variable ]), Variable \== readonly,
        Variable \== conflict_rep , Variable \== conflict_peer .


% array creation
% V[] v = new V[5];

new(static , [Array, ArrayEl]) :- universe ([Array, ArrayEl]),
        Array \== readonly, Array \== rep,
        Array \== conflict_rep , Array \== conflict_peer ,
        ArrayEl \== conflict_peer .
new(none, [Array, ArrayEl]) :- universe ([Array, ArrayEl]),
        Array \== readonly, Array \== conflict_rep ,
        Array \== conflict_peer , ArrayEl \== conflict_peer .


% field accesses right-hand side

% v = w.f.g.h;
% [w]*[f]*[g]*[h] <: [v]
```

```
% including method calls
% v = w.m(e);           field_access_right ([W, MReturn], Res).
% v = w.f.g.m(e);       field_access_right ([W, F, G, MReturn], Res).
% v = w.f.g.m(e).h;     field_access_right ([W, F, G, MReturn, H], Res).

field_access_right ([H], Res) :- type(H), Res = H.
field_access_right ([H1, H2|T], Res) :-
        type(H1),
        type(H2),
        combination(H1, H2, HRes),
         field_access_right ([HRes|T], Res).


% field accesses left-hand side
% including method calls

% v.f.g.h = w;
% [w] <: [v]*[f]*[g]*[h]

field_access_left ([H], Res) :- type(H), Res = H.
field_access_left ([H1, H2|T], Res) :-
        type(H1), H1 \== [this], H1 \== [readonly],
        type(H2), H2 \== [rep], H2 \== [conflict_rep],
        combination(H1, H2, HRes),
         field_access_left ([HRes|T], Res).
field_access_left ([H1, [Array, ArrayEl]|T], Res) :-
        type(H1), H1 \== [this], H1 \== [readonly],
        universe([Array, ArrayEl]), Array \== rep, Array \== conflict_rep,
        combination(H1, [Array, ArrayEl], [ResArray, ResArrayEl]),
         field_access_left ([[ResArray, ResArrayEl]|T], Res).
field_access_left ([[this], H2|T], Res) :-
        type(H2),
        combination([this], H2, HRes),
         field_access_left ([HRes|T], Res).


cast_left (static, [CastType], Expression) :-
         field_access_left (Expression, ExpressionType),
        [CastType] = ExpressionType,
        universe([CastType]),
        CastType \== rep,
        CastType \== conflict_rep, CastType \== conflict_peer.
cast_left (static, [CastType], Expression) :-
         field_access_left (Expression, ExpressionType),
        subtype_notequal([CastType], ExpressionType),
        universe([CastType]),
        CastType \== rep,
        CastType \== conflict_rep, CastType \== conflict_peer.
cast_left (static, [CastType], Expression) :-
         field_access_left (Expression, ExpressionType),
        subtype_notequal(ExpressionType, [CastType]),
        universe([CastType]),
        CastType \== rep,
        CastType \== conflict_rep, CastType \== conflict_peer.
cast_left (static, [CastType, CastEl], Expression) :-
         field_access_left (Expression, ExpressionType),
        [CastType, CastEl] = ExpressionType,
        universe([CastType, CastEl]),
        CastType \== rep,
        CastType \== conflict_rep, CastType \== conflict_peer,
        CastEl \== conflict_peer.
cast_left (static, [CastType, CastEl], Expression) :-
         field_access_left (Expression, ExpressionType),
        subtype_notequal([CastType, CastEl], ExpressionType),
        universe([CastType, CastEl]),
        CastType \== rep,
        CastType \== conflict_rep, CastType \== conflict_peer,
        CastEl \== conflict_peer.
cast_left (static, [CastType, CastEl], Expression) :-
         field_access_left (Expression, ExpressionType),
        subtype_notequal(ExpressionType, [CastType, CastEl]),
        universe([CastType, CastEl]),
        CastType \== rep,
        CastType \== conflict_rep, CastType \== conflict_peer,
        CastEl \== conflict_peer.
```

```
cast_left (none, [CastType], Expression) :-
        field_access_left (Expression, ExpressionType),
        [CastType] = ExpressionType,
        universe([CastType]),
        CastType \== conflict_rep, CastType \== conflict_peer.
cast_left (none, [CastType], Expression) :-
        field_access_left (Expression, ExpressionType),
        subtype_notequal([CastType], ExpressionType),
        universe([CastType]),
        CastType \== conflict_rep, CastType \== conflict_peer.
cast_left (none, [CastType], Expression) :-
        field_access_left (Expression, ExpressionType),
        subtype_notequal(ExpressionType, [CastType]),
        universe([CastType]),
        CastType \== conflict_rep, CastType \== conflict_peer.
cast_left (none, [CastType, CastEl], Expression) :-
        field_access_left (Expression, ExpressionType),
        [CastType, CastEl] = ExpressionType,
        universe([CastType, CastEl]),
        CastType \== conflict_rep, CastType \== conflict_peer,
        CastEl \== conflict_peer.
cast_left (none, [CastType, CastEl], Expression) :-
        field_access_left (Expression, ExpressionType),
        subtype_notequal([CastType, CastEl], ExpressionType),
        universe([CastType, CastEl]),
        CastType \== conflict_rep, CastType \== conflict_peer,
        CastEl \== conflict_peer.
cast_left (none, [CastType, CastEl], Expression) :-
        field_access_left (Expression, ExpressionType),
        subtype_notequal(ExpressionType, [CastType, CastEl]),
        universe([CastType, CastEl]),
        CastType \== conflict_rep, CastType \== conflict_peer,
        CastEl \== conflict_peer.

cast_right (static, [CastType], Expression) :-
        field_access_right (Expression, ExpressionType),
        [CastType] = ExpressionType,
        universe([CastType]),
        CastType \== rep,
        CastType \== conflict_rep, CastType \== conflict_peer.
cast_right (static, [CastType], Expression) :-
        field_access_right (Expression, ExpressionType),
        subtype_notequal([CastType], ExpressionType),
        universe([CastType]),
        CastType \== rep,
        CastType \== conflict_rep, CastType \== conflict_peer.
cast_right (static, [CastType], Expression) :-
        field_access_right (Expression, ExpressionType),
        subtype_notequal(ExpressionType, [CastType]),
        universe([CastType]),
        CastType \== rep,
        CastType \== conflict_rep, CastType \== conflict_peer.
cast_right (static, [CastType, CastEl], Expression) :-
        field_access_right (Expression, ExpressionType),
        [CastType, CastEl] = ExpressionType,
        universe([CastType, CastEl]),
        CastType \== rep,
        CastType \== conflict_rep, CastType \== conflict_peer,
        CastEl \== conflict_peer.
cast_right (static, [CastType, CastEl], Expression) :-
        field_access_right (Expression, ExpressionType),
        subtype_notequal([CastType, CastEl], ExpressionType),
        universe([CastType, CastEl]),
        CastType \== rep,
        CastType \== conflict_rep, CastType \== conflict_peer,
        CastEl \== conflict_peer.
cast_right (static, [CastType, CastEl], Expression) :-
        field_access_right (Expression, ExpressionType),
        subtype_notequal(ExpressionType, [CastType, CastEl]),
        universe([CastType, CastEl]),
        CastType \== rep,
        CastType \== conflict_rep, CastType \== conflict_peer,
        CastEl \== conflict_peer.

cast_right (none, [CastType], Expression) :-
```

```
        field_access_right (Expression, ExpressionType),
        [CastType] = ExpressionType,
        universe([CastType]),
        CastType \== conflict_rep, CastType \== conflict_peer.
cast_right(none, [CastType], Expression) :-
        field_access_right (Expression, ExpressionType),
        subtype_notequal([CastType], ExpressionType),
        universe([CastType]),
        CastType \== conflict_rep, CastType \== conflict_peer.
cast_right(none, [CastType], Expression) :-
        field_access_right (Expression, ExpressionType),
        subtype_notequal(ExpressionType, [CastType]),
        universe([CastType]),
        CastType \== conflict_rep, CastType \== conflict_peer.
cast_right(none, [CastType, CastEl], Expression) :-
        field_access_right (Expression, ExpressionType),
        [CastType, CastEl] = ExpressionType,
        universe([CastType, CastEl]),
        CastType \== conflict_rep, CastType \== conflict_peer,
        CastEl \== conflict_peer.
cast_right(none, [CastType, CastEl], Expression) :-
        field_access_right (Expression, ExpressionType),
        subtype_notequal([CastType, CastEl], ExpressionType),
        universe([CastType, CastEl]),
        CastType \== conflict_rep, CastType \== conflict_peer,
        CastEl \== conflict_peer.
cast_right(none, [CastType, CastEl], Expression) :-
        field_access_right (Expression, ExpressionType),
        subtype_notequal(ExpressionType, [CastType, CastEl]),
        universe([CastType, CastEl]),
        CastType \== conflict_rep, CastType \== conflict_peer,
        CastEl \== conflict_peer.


% assignment

assignment(LeftList, RightList) :-
        field_access_left (LeftList, LeftRes),
        field_access_right (RightList, RightRes),
        subtype_equal(RightRes, LeftRes).

% conflict: if the assignment is not possible
% try to make the right-hand side readonly and have
% a conflict on the right-hand side, which can be solved
% by a type cast to the type of the left-hand side


% boolean expressions like equality or inequality
% for example n.next == m.next

boolean_expression(LeftList, RightList) :-
        field_access_right (LeftList, LeftRes),
        field_access_right (RightList, RightRes),
        subtype_equal(RightRes, LeftRes).

boolean_expression(LeftList, RightList) :-
        field_access_right (LeftList, LeftRes),
        field_access_right (RightList, RightRes),
        subtype_notequal(LeftRes, RightRes).


% return statement in a method

return_statement(ReturnType, ExpressionList) :-
        field_access_right (ExpressionList, ExpressionType),
        subtype_equal(ExpressionType, ReturnType).


% method invocation

% splits up parameterlist and looks for every parameter, if it is again a list
% means field accesses
% builds the remaining type
% v = b.m(a.f);
% method_call_parameters_right([B], [[A, F]], [MPar], MMod).
% split_right([[A, F]], Result).
```

```
split_left   ([], []).
split_left  ([P|Ps],  Result) :-
          split_left  (Ps, PRes),
          field_access_left  (P, PType), Result = [PType|PRes].

split_right   ([], []).
split_right  ([P|Ps],  Result) :-
          split_right  (Ps, PRes),
          field_access_right  (P, PType), Result = [PType|PRes].


% parameters

% method call on an object
% v = w.m(e);

parameters ([], [ Par| ParList ], [ MPar|MParList], _) :-
          subtype_equal(Par, MPar), MPar \== [rep],
          parameters ([],  ParList, MParList, none).

parameters(_  , [], [],   _).

parameters([this ], [ Par| ParList ], [ MPar|MParList], _) :-
          combination([this ],  MPar, X),
          subtype_equal(Par, X),
          parameters([this ],  ParList, MParList, _).

parameters(FieldType, [ Par| ParList ], [ MPar|MParList], none) :-
          FieldType \== [this],
          combination(FieldType,  MPar, X),
          subtype_equal(Par, X), MPar \== [rep],
          parameters(FieldType,  ParList, MParList, none).

parameters(FieldType, [[ PArr, PArrEl]| ParList ],
                    [[ParArray,  ParArrayEl]|MParList], none) :-
          FieldType \== [this],
          universe ([ParArray,  ParArrayEl]),
          combination(FieldType, [ParArray,  ParArrayEl ],  X),
          subtype_equal([PArr,  PArrEl ],  X), ParArray \== rep,
          parameters(FieldType,  ParList,  MParList, none).

parameters(FieldType, [ Par| ParList ], [ MPar|MParList], pure) :-
          FieldType \== [this],
          combination(FieldType,  MPar, X),
          subtype_equal(Par,  X),
          parameters(FieldType,  ParList,  MParList, pure).


% method call on an object  on the  left-hand side on an assignment
% v = w.m(e);

% MPurity has to be given  directly  and not through a  variable .
% call  the  predicate  with the constant  pure  or  none

method_call_parameters_left  ([],   ParList,  MethParList, _) :-
          split_right  ( ParList,  ParameterList ),
          parameters ([],  ParameterList,  MethParList, _).

method_call_parameters_left ( FieldList ,  ParList,  MethParList, pure) :-
          field_access_right  ( FieldList ,  FieldType ),
          split_right  ( ParList,  ParameterList ),
          parameters(FieldType,  ParameterList,  MethParList, pure).

method_call_parameters_left ( FieldList ,  ParList,  MethParList, none) :-
          field_access_left  ( FieldList ,  FieldType ),
          FieldType \== [readonly],
          split_left  ( ParList,  ParameterList ),
          parameters(FieldType,  ParameterList,  MethParList, none).


% method call on an object  on the  right-hand side on an assignment
% v = w.m(e);

method_call_parameters_right  ([],   ParList,  MethParList, _) :-
          split_right  ( ParList,  ParameterList ),
```

```
        parameters ([],  ParameterList , MethParList, _ ).

method_call_parameters_right ( FieldList , ParList , MethParList, pure) :-
          field_access_right  ( FieldList , FieldType ),
          split_right  ( ParList , ParameterList ),
         parameters(FieldType , ParameterList , MethParList, pure ).

method_call_parameters_right ( FieldList , ParList , MethParList, none) :-
          field_access_right  ( FieldList , FieldType ),
         FieldType \== [readonly],
          split_right  ( ParList , ParameterList ),
         parameters(FieldType , ParameterList , MethParList, none ).
```

B.2 conflict.pl

```
% types

% simple universe types

universe ([rep ]).
universe ([readonly ]).
universe ([peer ]).
universe ([ conflict_rep  ]).
universe ([ conflict_peer  ]).

% all array universe types
% only for arrays which contain a reference type

universe ([rep , readonly ]).
universe ([rep , peer ]).
universe ([readonly , readonly ]).
universe ([readonly , peer ]).
universe ([peer , readonly ]).
universe ([peer , peer ]).

universe ([rep ,  conflict_peer  ]).
universe ([readonly ,  conflict_peer  ]).
universe ([peer ,  conflict_peer  ]).
universe ([ conflict_rep  , readonly ]).
universe ([ conflict_rep  , peer ]).
universe ([ conflict_rep  ,  conflict_peer  ]).
universe ([ conflict_peer  , readonly ]).
universe ([ conflict_peer  , peer ]).
universe ([ conflict_peer  ,  conflict_peer  ]).


% variable subtypes
% x = y;
% [y] <: [x];

subtype_equal(X, Y) :- subtype_notequal(X, Y).
subtype_equal(X, X) :- type(X).
subtype_equal ([ this ],[ peer ]).
subtype_equal ([peer ], [ this ]).

subtype_equal ([ conflict_rep  ], [ rep ]).
subtype_equal ([ conflict_rep  ], [ readonly ]).
subtype_equal ([ conflict_peer  ], [ readonly ]).
subtype_equal ([ conflict_peer  ], [ peer ]).
subtype_equal ([ conflict_peer  ], [ this ]).
subtype_equal ([rep ], [  conflict_rep  ]).
subtype_equal ([readonly ], [  conflict_rep  ]).
subtype_equal ([readonly ], [  conflict_peer  ]).
subtype_equal ([peer ], [  conflict_peer  ]).
subtype_equal ([this ], [  conflict_peer  ]).

subtype_equal ([rep ,  conflict_peer  ], [ rep , readonly ]).
subtype_equal ([rep ,  conflict_peer  ], [ rep , peer ]).
subtype_equal ([rep ,  conflict_peer  ], [  conflict_rep  , readonly ]).
subtype_equal ([rep ,  conflict_peer  ], [  conflict_rep  , peer ]).
subtype_equal ([rep ,  conflict_peer  ], [  conflict_rep  ,  conflict_peer  ]).
subtype_equal ([readonly ,  conflict_peer  ], [ readonly , readonly ]).
subtype_equal ([readonly ,  conflict_peer  ], [ readonly , peer ]).
subtype_equal ([readonly ,  conflict_peer  ], [  conflict_rep  , readonly ]).
```

```
subtype_equal([readonly, conflict_peer], [conflict_rep, peer]).
subtype_equal([readonly, conflict_peer], [conflict_peer, readonly]).
subtype_equal([readonly, conflict_peer], [conflict_peer, peer]).
subtype_equal([readonly, conflict_peer], [conflict_rep, conflict_peer]).
subtype_equal([readonly, conflict_peer], [conflict_peer, conflict_peer]).
subtype_equal([peer, conflict_peer], [peer, readonly]).
subtype_equal([peer, conflict_peer], [peer, peer]).
subtype_equal([peer, conflict_peer], [conflict_peer, readonly]).
subtype_equal([peer, conflict_peer], [conflict_peer, peer]).
subtype_equal([peer, conflict_peer], [conflict_peer, conflict_peer]).
subtype_equal([conflict_rep, readonly], [rep, readonly]).
subtype_equal([conflict_rep, readonly], [readonly, readonly]).
subtype_equal([conflict_rep, readonly], [rep, conflict_peer]).
subtype_equal([conflict_rep, readonly], [readonly, conflict_peer]).
subtype_equal([conflict_rep, readonly], [conflict_rep, conflict_peer]).
subtype_equal([conflict_rep, peer], [rep, peer]).
subtype_equal([conflict_rep, peer], [rep, conflict_peer]).
subtype_equal([conflict_rep, peer], [readonly, peer]).
subtype_equal([conflict_rep, peer], [readonly, conflict_peer]).
subtype_equal([conflict_rep, peer], [conflict_rep, conflict_peer]).
subtype_equal([conflict_peer, readonly], [peer, readonly]).
subtype_equal([conflict_peer, readonly], [readonly, readonly]).
subtype_equal([conflict_peer, readonly], [readonly, conflict_peer]).
subtype_equal([conflict_peer, readonly], [conflict_peer, conflict_peer]).
subtype_equal([conflict_peer, peer], [peer, peer]).
subtype_equal([conflict_peer, peer], [readonly, peer]).
subtype_equal([conflict_peer, peer], [conflict_peer, peer]).
subtype_equal([conflict_peer, peer], [peer, conflict_peer]).
subtype_equal([conflict_peer, peer], [readonly, conflict_peer]).
subtype_equal([conflict_peer, peer], [conflict_peer, conflict_peer]).
subtype_equal([conflict_rep, conflict_peer], [rep, peer]).
subtype_equal([conflict_rep, conflict_peer], [rep, readonly]).
subtype_equal([conflict_rep, conflict_peer], [readonly, peer]).
subtype_equal([conflict_rep, conflict_peer], [readonly, readonly]).
subtype_equal([conflict_rep, conflict_peer], [conflict_rep, readonly]).
subtype_equal([conflict_rep, conflict_peer], [readonly, conflict_peer]).
subtype_equal([conflict_rep, conflict_peer], [conflict_rep, peer]).
subtype_equal([conflict_rep, conflict_peer], [rep, conflict_peer]).
subtype_equal([conflict_peer, conflict_peer], [peer, peer]).
subtype_equal([conflict_peer, conflict_peer], [readonly, peer]).
subtype_equal([conflict_peer, conflict_peer], [peer, readonly]).
subtype_equal([conflict_peer, conflict_peer], [readonly, readonly]).
subtype_equal([conflict_peer, conflict_peer], [conflict_peer, peer]).
subtype_equal([conflict_peer, conflict_peer], [peer, conflict_peer]).
subtype_equal([conflict_peer, conflict_peer], [readonly, conflict_peer]).

subtype_equal([rep, readonly], [rep, conflict_peer]).
subtype_equal([rep, readonly], [conflict_rep, readonly]).
subtype_equal([rep, readonly], [conflict_rep, conflict_peer]).
subtype_equal([rep, peer], [rep, conflict_peer]).
subtype_equal([rep, peer], [conflict_rep, peer]).
subtype_equal([rep, peer], [conflict_rep, conflict_peer]).
subtype_equal([readonly, readonly], [readonly, conflict_peer]).
subtype_equal([readonly, readonly], [conflict_rep, readonly]).
subtype_equal([readonly, readonly], [conflict_peer, readonly]).
subtype_equal([readonly, readonly], [conflict_rep, conflict_peer]).
subtype_equal([readonly, readonly], [conflict_peer, conflict_peer]).
subtype_equal([readonly, peer], [readonly, conflict_peer]).
subtype_equal([readonly, peer], [conflict_rep, peer]).
subtype_equal([readonly, peer], [conflict_peer, peer]).
subtype_equal([readonly, peer], [conflict_rep, conflict_peer]).
subtype_equal([readonly, peer], [conflict_peer, conflict_peer]).
subtype_equal([peer, readonly], [peer, conflict_peer]).
subtype_equal([peer, readonly], [conflict_peer, readonly]).
subtype_equal([peer, readonly], [conflict_peer, conflict_peer]).
subtype_equal([peer, peer], [peer, conflict_peer]).
subtype_equal([peer, peer], [conflict_peer, peer]).
subtype_equal([peer, peer], [conflict_peer, conflict_peer]).


% combinations

combination([rep], [rep], [readonly]).
combination([rep], [readonly], [readonly]).
combination([rep], [peer], [rep]).
```

```
combination([readonly ], [ rep ], [ readonly ]).
combination([readonly ], [ readonly ], [ readonly ]).
combination([readonly ], [ peer ], [ readonly ]).

combination([peer ], [ rep ], [ readonly ]).
combination([peer ], [ readonly ], [ readonly ]).
combination([peer ], [ peer ], [ peer ]).

combination([X], [Y, Z], [U, Z]) :- combination([X], [Y], [U]).

combination([this ], [ rep ], [ rep ]).
combination([this ], [ readonly ], [ readonly ]).
combination([this ], [ peer ], [ peer ]).

combination([this ], [Y, Z], [U, Z]) :- combination([this ], [Y], [U]).

combination([rep ], [ primitive ], [ primitive ]).
combination([readonly ], [ primitive ], [ primitive ]).
combination([peer ], [ primitive ], [ primitive ]).
combination([this ], [ primitive ], [ primitive ]).

combination([X,Y], [ primitive ], [ primitive ]) :- universe ([X,Y]).
```

```
combination([rep ], [ conflict_rep ], [ readonly ]).
combination([rep ], [ conflict_peer ], [ rep ]).
combination([readonly ], [ conflict_rep ], [ readonly ]).
combination([readonly ], [ conflict_peer ], [ readonly ]).
combination([peer ], [ conflict_rep ], [ readonly ]).
combination([peer ], [ conflict_peer ], [ peer ]).

combination([ conflict_rep ], [ rep ], [ readonly ]).
combination([ conflict_peer ], [ rep ], [ readonly ]).
combination([ conflict_rep ], [ readonly ], [ readonly ]).
combination([ conflict_peer ], [ readonly ], [ readonly ]).
combination([ conflict_rep ], [ peer ], [ rep ]).
combination([ conflict_peer ], [ peer ], [ peer ]).

combination([ conflict_rep ], [ conflict_rep ], [ readonly ]).
combination([ conflict_rep ], [ conflict_peer ], [ rep ]).
combination([ conflict_peer ], [ conflict_rep ], [ readonly ]).
combination([ conflict_peer ], [ conflict_peer ], [ peer ]).

combination([ conflict_rep ], [ primitive ], [ primitive ]).
combination([ conflict_peer ], [ primitive ], [ primitive ]).
combination([this ], [ conflict_rep ], [ rep ]).
combination([this ], [ conflict_peer ], [ peer ]).
```