Static Universe Type Inference using a SAT-Solver

Matthias Niklaus

Master Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

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

Dec. 2005 - May 2006

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 allows to restrict the possible aliasing in object-oriented programs and thereby enables static reasoning about individual components. Compared to other ownership type systems, the Universe type system is lightweight, but annotating existing software is still a considerable effort. To ease the effort of annotation, static inference of Universe modifiers from Java source code is an interesting problem. In this work, we investigated the use of pseudo-boolean optimizers to find Universe annotations. To show the benefit of the new backend we implemented a new static Universe type inferer. The results reveal that also heavy annotated programs can be typed correctly. This report presents the architecture, implementation, and results of the new tool.

Contents

0.1 Abbreviations

Abbreviations used in this report:

- AST - Abstract Syntax Tree
- CNF - Conjunctive Normal Form
- DNF - Disjunctive Normal Form
- JML - Java Modelling Language
- PB - The Pseudo Boolean Problem
- PB-S - An arbitrary Pseudo Boolean Solver
- PBS-tool - A concrete Pseudo Boolean Solver the PBS (sometines only referred to as PBS)
- SAT - The Boolean Satisfiability Problem
- SUTI1 - Static Universe Type Inference 1 (former project by Natalie Kellenberger [3])
- SUTI2 - Static Universe Type Inference 2 (the current project)
- UTI - Universe Type Inferer

Chapter 1

Introduction

1.1 Motivation

Object oriented programming strongly benefits from distributing references among differnet components to get access to the referenced objects without copying. This aliasing unfortunately may lead to a modification of object structures by components which are not meant to change these structures. The Universe type system [1] introduces a mechanism to allow aliasing without the potential danger of a modification of the aliased objects.
A former master thesis [3] to infer the Universe types of a given Java program was written by Nathalie Kellenberger.

The goal of this master thesis was to replace the Prolog backend of the first static Universe type inferer tool with a SAT-solver. We hope to be able to have a better influence on the order of the found solutions. So that we do not have to consider too many unsuitable solutions until we get one which satisfies our expectations.

1.2 Short Introduction to the Universe Type System

The Universe type system provides a mechanism for aliasing and dependency control. All objects are placed in different hierarchical contexts. At most one owner is assigned to every object. All objects with the same owner form a context. The Universe type system ensures that writable references are allowed only between objects which are in the same context and from the owner to its owned objects. Any other reference must be read-only.

To express this ownership structure the Universe type system provides three type modifiers (*peer*, *rep*, and *readonly*).

- *peer* denotes references to objects in the same universe.
- *rep* denotes references to owned objects.
- *readonly* is used for arbitrary references.

The Universe annotations are not used for primitive types since they cannot be aliased.

1.3 The SUTI1 Tool

In the SUTI1 project a static Universe type inferer was introduced. This inferer was implemented with a Prolog backend. The inferer parses the Java source files and generates a Prolog abstraction of the relevant parts. The Prolog system then is used to get multiple Universe type annotations for the provided program. As a next step, the retrieved annotations are rated and the best one is chosen.

1.4 SAT Introduction

The boolean satisfiability problem *SAT* is a well known decision problem in computer science. An instance of the problem is a boolean expression written using only AND, OR, NOT, literals, and parentheses. The question is: given the expression, is there some assignment of TRUE and FALSE values to the variables that will make the entire expression true? [13]

$$(rain \rightarrow clouds) \leftrightarrow \neg(rain \wedge \neg clouds) \tag{1.1}$$

The formula 1.1 is an example for such a boolean expression. The left subexpression of this equivalence is an implication. To satisfy this implication *clouds* must be set to TRUE if *rain* is TRUE. This sub formula seems quite logic. I cannot remember some rain without clouds.

1.4.1 Conversions

There are many tools for solving the SAT problem. These tools require that the boolean expression is in a specific form. For many tools, the expression has to be in the conjunctive normal form *CNF*. The CNF is a conjunction of clauses. Each clause is a disjunction of possibly negated literals. If S_i is a disjunction of literals then $\bigwedge_{i=1}^{n}$ is called a system S in CNF with clauses S_i.

$$\begin{aligned} (\neg rain \vee clouds \vee rain) \quad & \wedge \\ (\neg rain \vee clouds\neg \vee clouds) \quad & \wedge \\ (\neg rain \vee clouds \vee rain) \quad & \wedge \\ (\neg rain \vee clouds\neg \vee clouds) \quad & \end{aligned} \tag{1.2}$$

The formula 1.2 shows the expression 1.1 converted in a CNF. Each line shows a clause. Only the literals are negated.
Every boolean expression can be converted in a CNF with equivalent transformation. A transformation is equivalent iff the transformed expression evaluates always to the same boolean value as the original expression. The equivalence is easy to see if both expressions are written in a truth table. Then all entries must be equal. In the next sections we will look at some used equivalent transformations.

Implication

An implication as in the left subexpression of formula 1.1 means that when the left handside of the implication is true then also the right handside of the implication must be true to make the whole implication true. If the left side is false then it does not matter whether the right side of

the implication is true or false.
The implication in the formula 1.1 says, that if it *rains* then it must have *clouds*.

An implication can be converted straight forward to a clause in a CNF. The following rule shows how the implication can be written as a disjunction.

$$P \rightarrow Q \equiv \neg P \vee Q$$

The correctness of this transformation can be seen in the truth table. The truth table for this conversion has in the columns for the original and the converted expression the same entries.

P	Q	$P \rightarrow Q$	$\neg P$	$\neg P \vee Q$
0	0	1	1	1
0	1	1	1	1
1	0	0	0	0
1	1	1	0	1

With applying this rule, we can remove the implication in the formula 1.1 and replace it with a disjunction.

$$\begin{aligned} (rain \rightarrow clouds) &\Rightarrow (\neg rain \vee clouds) \\ (rain \rightarrow clouds) \leftrightarrow \neg(rain \wedge \neg clouds) &\Rightarrow (\neg rain \vee clouds) \leftrightarrow \neg(rain \wedge \neg clouds) \end{aligned} \tag{1.3}$$

In the resulting formula 1.3 the left side of the equivalence is a negated sub-expression. How thes negation can be removed is shown in the next section.

De Morgan

The laws of De Morgan say how we can pull a negation ($\neg$) which is outside a parenthesised expression into the parenthesis. This allows us to remove the negations in front of a parenthesised expression. So in the resulting expression the negations only appear before a literal.

$$\neg(P \vee Q) \equiv \neg P \wedge \neg Q$$

The De Morgan rule for a negated disjunction. The rule can be verified with the truth table below.

P	Q	$\neg(P \vee Q)$	$\neg P$	$\neg Q$	$\neg P \wedge \neg Q$
0	0	1	1	1	1
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	0

$$\neg(P \wedge Q) \equiv \neg P \vee \neg Q$$

The De Morgan rule for a negated conjunction. The rule can also be verified with the truth table below.

P	Q	$\neg(P \wedge Q)$	$\neg P$	$\neg Q$	$\neg P \vee \neg Q$
0	0	1	1	1	1
0	1	1	1	0	1
1	0	1	0	1	1
1	1	0	0	0	0

When we apply the law of De Morgan to the right side of the equivalence $\neg(rain \wedge \neg clouds)$ in the boolean expression 1.3, we can remove the negation. To remove the negation of this conjunction, we must replace the conjunction with a disjunction and negate the literals.

$$\begin{aligned} \neg(rain \wedge \neg clouds) &\Rightarrow (\neg rain \vee \neg\neg clouds) \\ (\neg rain \vee clouds) \leftrightarrow \neg(rain \wedge \neg clouds) &\Rightarrow (\neg rain \vee clouds) \leftrightarrow (\neg rain \vee \neg\neg clouds) \end{aligned} \tag{1.4}$$

With the help of the laws of De Morgan we could remove the negation in front of the whole right side in the equivalence. But by this removal we introduced a double negation. Let us remove this double negation.

Double Negation

A double negation is the same as no negation. And therefore the following rule holds.

$$\neg\neg P \equiv P$$

The negation conversion can be verified in the small truth table below.

P	$\neg P$	$\neg\neg P$
0	1	0
1	0	1

With this rule we reduce the double negation in the resulting formula 1.4.

$$\begin{aligned} (\neg rain \vee \neg\neg clouds) &\Rightarrow (\neg rain \vee clouds) \\ (\neg rain \vee clouds) \leftrightarrow (\neg rain \vee \neg\neg clouds) &\Rightarrow (\neg rain \vee clouds) \leftrightarrow (\neg rain \vee clouds) \end{aligned} \tag{1.5}$$

The resulting formula 1.5 now consists of a equivalence of two disjunctions.

Equivalence

An equivalence is true if and only if the right and left side evaluates to the same boolean value. Therefore we may split up an equivalence into two implications. One left to right and one right to left. These two implications must both be true, to satisfy the equivalence. The implications then can be converted to disjunctions.

$$P \leftrightarrow Q \equiv (P \rightarrow Q) \wedge (Q \rightarrow P)$$

To see the correctness of this conversion again a truth table is provided.

P	Q	$P \leftrightarrow Q$	$P \rightarrow Q$	$Q \rightarrow P$	$(P \rightarrow Q) \wedge (Q \rightarrow P)$
0	0	1	1	1	1
0	1	0	1	0	0
1	0	0	0	1	0
1	1	1	1	1	1

Now we replace the equivalence in the formula 1.5 by two implications.

$$\begin{aligned}&(\neg rain \vee clouds) \leftrightarrow (\neg rain \vee clouds) \Rightarrow \\ &\quad ((\neg rain \vee clouds) \rightarrow (\neg rain \vee clouds)) \wedge \\ &\quad ((\neg rain \vee clouds) \rightarrow (\neg rain \vee clouds))\end{aligned} \tag{1.6}$$

The resulting formula 1.6 now is a conjunction of two implications. We saw above that an implication can be transformed into a disjunction. Then we have a mixed expression consisting only of conjunctions and disjunctions. With the help of the distributive law can convert the formula into the CNF. But this step increases the size of the formula. Converting a DNF into a CNF need exponential time in the size of the starting formula.

It is not really useful to still convert this formula. It consist already out of four equal parts. And it is obvious that both implication always evaluates to true. And the whole formula is a tautology. But however the formula converted in the CNF was shown before. The formula 1.2 is the result.

Truth Table

We have seen the truth table before. There is a rule how you can write a expression in CNF given the truth table for this expression. You have to consider each row in the table which evaluates to false. Then you must make a disjunction over all literals with negated assignment for each row. The disjunctions you have to conjunct.

P	Q	$P \vee Q$
0	0	0
0	1	1
1	0	1
1	1	1

The truth table of a simple disjunction. Here only the first row evaluates to false. P and Q are both assigned with false therefore $P \vee Q$ is equivalent to this truth table.

P	Q	$P \wedge Q$
0	0	0
0	1	0
1	0	0
1	1	1

The truth table of a conjunction. Here only the last row evaluates to true. We have to consider all first three rows. The expression $(P \vee Q) \wedge (P \vee \neg Q) \wedge (\neg P \vee Q)$ is equivalent to this truth table.

You can see that this does not lead to the shortest expression, but it is a good way to write a CNF given the boolean function as a truth table.

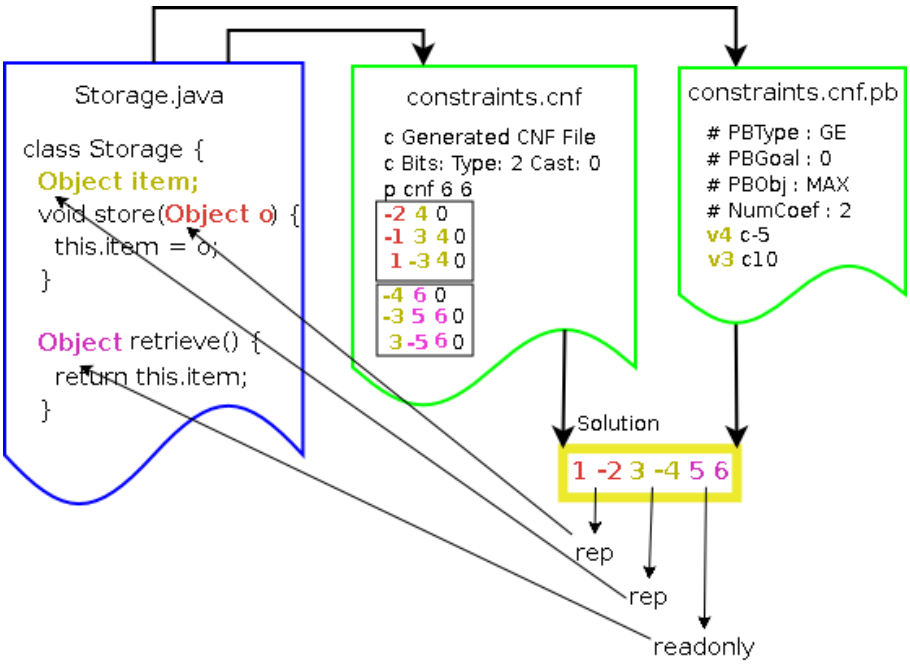

Figure 1.1: A toy example

1.4.2 Complexity

The SAT problem is *NP*-complete. Which must be considered later in the discussion about runtime of the new backend.

But also the conversions may use exponential time in size of the expression to convert. Especially converting an expression in disjunctive normal form to a CNF needs exponential time. The SAT problem on a DNF is not a difficult problem, because a DNF is satisfiable if, one conjunction is satisfiable. That means converting an expression from CNF to DNF is similar to solve the SAT problem for this expression.

So it is important that we build the rules representing the Universe type system from the beginning in the conjunctive normal form.

Instead of using equivalence transformation we could also use only satisfiability equivalence transformations. With these transformations one can generate more efficient a CNF.

1.5 A Toy Example with a PBS Backend

Before we go into the details here is a small example how the types of a Java program are infered with the PBS backend.
PBS is a type of SAT-solver with some optimisation features. The details of PBS are discussed in section 3.2.1.

In principle we are doing the same thing as was done in the former solution [3]. We read the source code of a Java program and extract the Universe type restrictions which must be satisfied by a correct Universe typing. We translate the type restrictions in a boolean formula which is in CNF and write it to a file. In addition we generate a second file which contains a description

of the preferred solution. These two files are the input of the PBS-tool. If the system is not overconstrained the tool returns a model for the CNF. This model is optimal in relation to the description of the preferred solution.

A concrete example is shown in figure 1.1. We have a single Java class `Storage`. The `Storage` class implements a very simple container. It has the ability to store an item of `Object`. To store and retrieve the item there are appropriate methods. The *Storage.java* file is shown in the left-most and blue document on figure 1.1. There are three Universe annotations to be infered for the `Storage` class.

To infer these three annotation the Universe type constraints are extracted from th Java code. Then the *constraints.cnf* file is generated. In this file, the boolean encoding of the type constraints are written. The right most file is the *constraints.cnf.pb* file, which contains some coefficients to weight the bits and the goal the solver has to satisfy.

For each annotation we have chosen one color. The places in the Java program, the literals in the *.cnf file, the weighting in the *cnf.pb file and the literals in the result are always colored the same way. So one can see where the informations for one annotation are stored.

The pseudo boolean solver the tries to find an optimal solution. As result the solver returns a bunch of signed integers representing the found model for the provided constraints. The infered Universe types are encoded in this model.

To get the infered annotations the obtained model must be decoded.

Chapter 2

Requirements

In this chapter we formulate the requirements our architecture should meet. There are obvious requirements like the static correctness of the annotations. This means, that a program which is annotated with infered Universe types, must be accepted by a Universe type aware compiler. But other requirements are not so easy to formulate. The tool should produce a reasonable annotation and provide a mechanism to specify what a reasonable solution is.

2.1 Completeness and Efficiency

Every Java program can be annotated with the Universe type system. Therefore the tool must be able to find an annotation for every program. With partly annotated programs, it may be necessary to introduce some casts to find a valid annotation. We must allow the necessary casts to find a solution. But the annotated program should also be runnable and not try to perform invalid casts.
On the other hand the number of possible solutions strongly increases when casts are allowed. So the task to find an optimal solution gets hard.

2.1.1 The Necessity of Casts

The previous static inference tool developed in [3] already introduced casts. To minimize the number of possible and to reject a lot of corrupt solutions a reference was only allowed to cast to *rep* or *peer* but never to both of them. This restriction is quite strong and in my opinion too strong. With this restriction it is not possible to annotate the following partly annotated and very common piece of code.

```
public class Storage {
  Object stored;
  public void store(Object o){
    this.stored = o;
  }
  public Object retrieve(){
    return stored;
  }
}
```

```
public class RepClient {
  /*@ rep @*/ Object field;
  /*@ peer @*/ Storage sto;
  public RepClient() {
     field  = new Object();
    sto  = new Storage();
  }
  public void doSomething() {
    sto.store( field );
     field  = sto.retrieve ();
  }
}

class  PeerClient {
  /*@ peer @*/ Object field;
  /*@ peer @*/ Storage sto;
  public PeerClient(){
     field  = new MyObject();
    sto  = new Storage();
  }
  public void doSomething(){
    sto.store( field );
     field  = sto.retrieve ();
  }
}
```

The code shows a container class which is used by two clients. One client uses the container to store *rep* objects the other client stores *peer* objects. A container class is often used in two different contexts. A sensible solution is to annotate all fields and parameters in the *Storage* class as *readonly*. Without generics this is a very common situation. And the tool must be able to handle such a situation. The tool presented in SUTI1 [3] is not able to handle this kind of situations.

As mentioned above the tool developed in SUTI1 is not able to annotate this code. The prolog implementation would answer *no*. The system is over constraint.

We want be able to handle such a situation. Therefore we must allow that one reference can be casted to *peer* and *rep*.

Perhaps it is possible to bypass all such situation with the generics mechanism. But it still would be a beg restriction to the programmer to prohibit all such situations.

2.1.2 Reducing the Number of Potential Casts

If all kinds of casts are allowed the feasible domain is very large. To reduce the number of feasible solutions we only want to allow casts at positions in the code where they may be necessary. These positions are:

- The target of a method call.
- The argument of a method call.
- The right side of an assignment statement.

- The target of an update statement.
  target of non pure method call
  target of field assignment statement
- Casts inserted by the user.

So we do not insert multiple casts in an access chain. The casts are inserted always before a reference is constraint. Such a constraint may require a cast.

It is absolutely enough to allow these cast, so that we can find an annotation for all partly annotated programs. To introduce more places for potential casts would be disadvantageous for the tools performance and solutions quality.
It is important to say that all introduced casts may fail at runtime. Therefore each cast is objectionable. The tool as well must avert to introduce casts which will fail for sure at runtime.

2.2 Solution Quality

A well annotated program is a program where the Universe annotation describes a deep object structure. The annotation should statically provide as much information as possible. Moreover it should not fail at runtime. To get a feeling for the word deep we will give some examples.

Every unannotated program may be annotated with all *peer*. This means the same as no annotation at all. All objects are in the same universe and we gained nothing. For getting such a solution we could conserve our labour.

On the other hand the whole program may be annotated with *readonly* and we introduce a cast when necessary. With such an annotation we also have quite no information we can use to verify our program and the probability of a cast runtime failure is very high.

An other possible annotation is to annotate quite all fields as *rep* and to declare the interfaces as *readonly*. Without casts this would be the most desirable solution, because the Universe type system declares a very deep object structure and all information is statically available. Such an annotated program is optimal for static formal verifications.
But often such an annotation requires a lot of casts. Casts insert runtime checks. And therefore we lose static information and gain uncertainty. Even worse are casts which will always fail at runtime. Such a solution can not be called valid.

What kind of solution do we want now?
A solution which represents a deep object structure but uses as few casts as possible. So it is not possible to easily give the answer what is an overall best solution criteria. But we can extract attributes of a solution and try to say for each attribute how much we prefer it.

2.2.1 Attributes of the Universe Type Annotation that We Want to Specify in Our Preferences

To determine the quality of the solutions we extract different attributes of it and give each of these attribute a weight. In this section we define the attributes of a solution which we want to weight. The actual default weight is determined later in a refinement process. The user of the tool should get an intuitive interface to adjust these weights for his purpose.

- Type of a field {rep, peer, readonly}

- Type of a local variable {rep, peer, readonly}
- Type of a parameter {rep, peer, readonly}
- Type of a return value {rep, peer, readonly}
- Acceptance of a cast {allow, forbid}

When we specify a weight for each of the listed attributes, then a value can be calculated. And the solution with the highest value should be the most appropriate solution for each solution.

Chapter 3

Architecture

The architecture we have chosen for the SUTI1 project is described in this chapter. SUTI1 is split up in two parts. The client part and the inferer part. Between these two parts is a clearly defined interface, which allows the independent development of both parts. Further it allows to use the inferer for different clients and it also allows to implement different inferer.

- Client
  This part provides the Java program structure to the inferer. It also handles the whole user interaction and provides the results in the preferred form.
- Inferer
  The inferer takes all provided program information into account and infers an optimal solution. The Universe annotation can be queried by the client.

As in the former SUTI1project [3] we statically extract all necessary information out of a syntax tree. All this information is the provided by the **U**niverse **T**ype **I**nference UTI interface to the inferer. The solution infered is then written to the specified xml output file.

The section 3.1 roughly explains the architecture and shows all components. The motivation for the decoupling of the inferer part is given in section 3.2. The inferer interface called *UTI* is described in section 3.3. Then the architecture of the client side and an implementation of the UTI interface is explained.

3.1 Overview

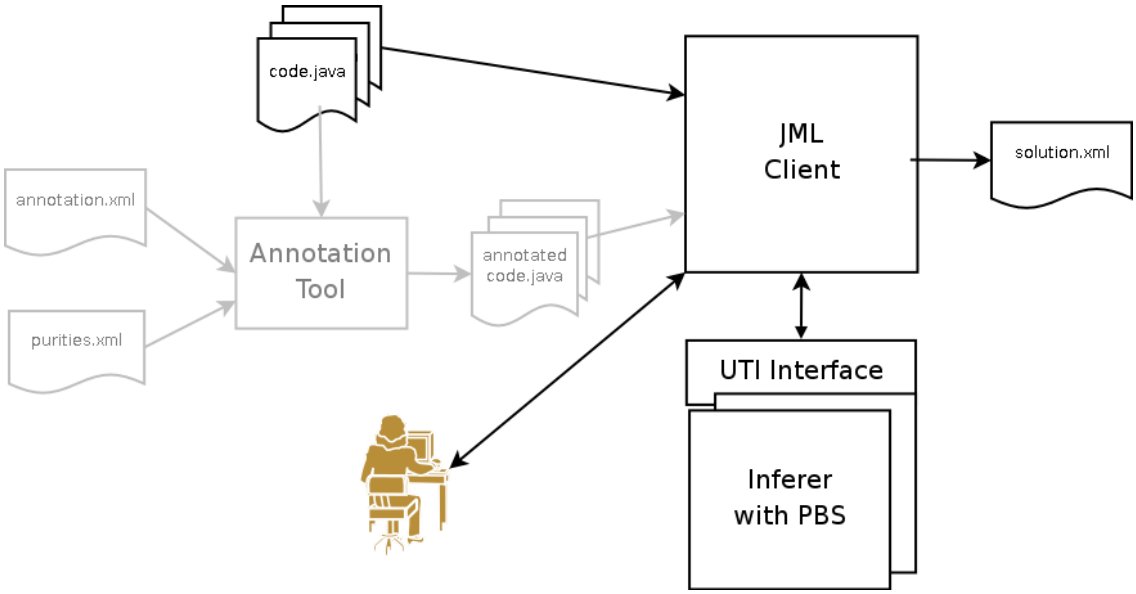

Figure 3.1: The tools architecture

The diagram shows how the tool is used. The grey parts are not implemented.

It was intended to use the Annotation tool [12] to reprocess the provided Java program. The annotation tool would insert the additional annotations given as xml files into the Java program code. The annotated Java program then is the actual input to the JML client.

At the moment the tool considers only the provided Java code. The code is parsed, with the annotations which are already in the code. And at the end of the process the new optimal annotation is written as xml to a specified file.

3.1.1 The JML Client

The concrete architecture of a UTI client using the JML compiler is explained in section 3.4. Here in this section we only describe what tasks the client fulfils.

The JML client provides the interface for user interaction. It parses the specified partial annotated Java program and provides the necessary program structure to the inferer implementing the UTI interface. After the whole program structure is provided to the inferer, then the client asks the inferer for the *best* solution.

The solution quality depends on the inferer. The client then may show the solution to a developer. The developer checks the solutions overall quality. This developer should especially check inserted casts. Principally it is possible that inserted casts fail during runtime. The tool makes no control flow analysis to get enough information about the program to ensure that the casts will work.

After checking the solution, the client may provide some corrections to the inferer. There are several possible ways to change the solution for the client.

- Forbid a particular cast

- Fix a particular cast to a predetermined Universe type
- Fix a type annotation to a predetermined Universe type
- Prevent a type annotation of a predetermined Universe type
- Change the purity of a particular method
- Change the solution preferences

After providing the changes the client must ask the inferer for a new optimal solution. It is not necessary to again provide the whole program structure. So it is not necessary to parse again all the Java program files. Then the developer may again check the solution. This check, restrict or relax process may be done as many times as the developer wants to change something in the solution.

Once the solution quality is good enough the client may write the solution to a xml file to store it persistent or annotate the program.

3.1.2 Universe Type Inferer Implements UTI Interface

The inferer must implement the UTI interface described in section 3.3. So there may be several inferer and at the initialization time the clients may freely choose which inferer they want use. The inferer is a passive component and is used by the client. It provides an interface to consume the program structure which is necessary to deduce the Universe type annotations.

All the magic stuff is done by the inferer. The inferer can be configured with a key value map so that it meets the clients needs.

The inferer consumes the program structure. It builds out of the program structure a constraint system representing all Universe type system constraints. A model satisfying this constraint system is a Universe annotation for the whole consumed Java program. The inferer then applies some heuristics to find an optimal model.

An inferer may be implemented with different constraint solving algorithms or tools. It is possible to implement an inferer which uses the Prolog backend from SUTI1 [3]. More possible tools to use by an inferer are discussed in section 3.2.1.

We implemented an inferer described in section 3.5 designed to use the *PBS* from [14]. PBS is an Incremental Pseudo-Boolean Backtrack Search SAT Solver and Optimizer. The PBS tool has the big advantage in comparison with the Prolog system, that it is an optimization tool and provides a quite good mechanism to specify what a good solution is. More about the PBS tool is explained in section 3.2.1.

3.1.3 The Best Solution

As described in the section above it is the inferers task to find the best solution. In section 2.2 we discussed the properties which are relevant for the solution quality. Each implementation of an inferer treats the Universes type system differently and implements different heuristics to find the optimal solution.

In the former work SUTI1[3] it was possible for the user to choose a certain heuristic out of a finite set of heuristics. Each of the heuristics implemented a different algorithm to evaluate the

solution's quality. So it is important that the user understands each of the heuristics to be able to choose the best one for his needs.

With multiple implementations of different inferer the user would need to know quite many different heuristics, to be able to choose the one which fulfils his needs. It is already enough to know which inferer fits best. Also a client software wants to be able to provide a standard interface to specify the prefered properties of the infered Universe annotations. Our intention is to elaborate an abstraction of all possible heuristics for all possible inferers.

A general description of the Universe annotation properties is the most intuitive way how a user wants to specify his wishes. The idea is that the user may specify for each kind of type occurrence a relation between the three possible Universe type modifiers *peer*, *rep*, and *readonly*. Moreover it is possible to specify a relation between the different type occurrences. So for example the inferer should give a higher priority to the member field annotation relation than to the parameter annotation relation. And at the end the user may specify how many casts he wants to allow to get the specified relation among the annotations.

It can not be guaranteed that every inferer with the same description of the preferred Universe annotations deduces the same result. It is the idea that the inferer should behave similarly. But more important is that all inferer may be configured via the same interface. It is very important that every inferer shows his good default configuration to the user, so that he only has to adapt the configuration to his needs when he is not pleased with the deduced solution.

The exact description of the solution description interface is given in section 3.3.

3.1.4 The Advantages

With the introduction of the UTI interface we got a strict and flexible partition in a client and an inferer part. So the clients may concentrate on the user interface and use a freely chosen compiler frontend. This allows an easy integration of static Universe type inference in already existent tools. On the other hand is the UTI interface so flexible that most static Universe type inference algorithms may be implemented as an inferer. Algorithms which want to make control flow analysis are not supported by the UTI interface. Such algorithms needs a close interaction with a Java syntax tree and are better implemented individually for each tool.

The general facility to describe the preferred solution properties allows the usage of a wide range of different inferer implementing specific elaborated algorithms. All these different inferer are tuned the same way. To not restrict the inferer there is moreover the possibility of providing inferer specific *key value* pairs for additional configuration.

3.2 Choosing an Appropriate Problem

To solve the Universe infering problem we want to map it to a common problem the computer science or use a dedicated programming language. In section 2.2 we stated that we want to describe the propertries of our solution. The found solution should match this property specification as close as possible. So the *optimal* solution matches the annotation specification.

The former project SUTI1[3] mapped the Universe infering problem to Prolog. The Prolog system then was used to get a specified number of solutions. Then there were different weighting algorithms to weight these solutions. And the solution with the highest weight was chosen. The Prolog system was only used to get a limited set of valid solutions. All the heuristics were implemented manually in multiple Java algorithms.

When the system is not much constraint or it is allowed to introduce casts then the number of possible solution is huge. To get a chance to find a quite optimal solution in a huge amount of possible solutions the limited set of solutions which are considered must be quite big. This then results in a bad runtime. So we want to put the solution optimality into the problem representing the Universe infering problem. This gives us the advantage that we don't have to consider many not optimal solutions. And moreover we want to use dedicated and highly optimized algorithms to solve the complex optimization problem. This we can accomplish by choosing standard optimization problems in computer science.

3.2.1 The Problem Solvers

In this section we discuss some problems we might map the Universe infering problem to and the tools which are available to solve these problems. Also we discuss how well the problem suits the mapping of the Universe infering problem.

Constraint and Constraint Logic Programming (CP and CLP)

In the constraint programming paradigm relations between variables are specified in form of constraints and not the steps to solve the problem are provided. The first constraint programming paradigm was introduced in Prolog and is called constraint logic programming. In [3] such a system was used. But there was no possibility to get an optimized solution. There are numbers of popular constraint logic languages which are most based on Prolog. But these are proprietary. And were not closer considered.

Beside the constraint logic programming, which bases on logic host languages, there is constraint programming implemented in various other host languages. For example the free *Choco* [18] implementation. More about Choco later in this section.

The constraint programming paradigm fits good for the Universe infering problem because it allows to specify the constraints we are extracting out of the Java program in a natural way. The drawback is again that constraint programming is not an optimization programming technique. Although there are some CP implementations which provide basically optimization facility. The Choco library does provide such an optimization function.

Choco Choco is a constraint programming system. Choco is a Java library for constraint satisfaction problems (CSP) and constraint programming (CP). Choco is freely available.

A CSP is defined as a triplet (X,D,C) such as:

1. Variables: $X = \{X_1, X_2, ..., X_n\}$ is the set of variables of the problem.
2. Domain: D is a function which associates to each variable X_i its domain $D(X_i)$, i.e., $\{peer, rep, readonly\}$.
3. Constraints: $C = \{C_1, C_2, ..., C_k\}$ is the set of constraints. Each constraint C_j is a relation between a subset of variables which restricts the domain of each one.

To model the Universe infering problem we associate to each variable a Universe annotation position. The domain of all variables would be the possible state in which an annotation may be. In a simple system this would be *peer, rep* and *readonly*. And the constraints can be used to map the type relations.
Here is a example how to model the subtype relation of two variables in Choco.

Listing 3.1: Subtype relation in Java for Choco

```
// rep = 0, peer = 1, readonly = 2
//subtype a <: b no casts
// the problem instance
AbstractProblem uip = new Problem();
IntEnumVar a = pb.makeEnumIntVar("a",0,2);
IntEnumVar b = pb.makeEnumIntVar("b",0,2);

// subtype constraint must be created out of aritmetic expressions
pb.post(pb.subType(a,b));

// heuristic we prefer b rep.
int [] weight = {0, -1};
// objective function
c = pb.post(pb.scalar(weight, new IntVar[]{a,b}));
//start optimization
pb.maximise(c,false);
```

In Choco all the constraints must be build up out of arithmetic expressions. This does not really fit well for the Universe infering problem. The objective function is also an arithmetic expression. In the example we have chosen to weight each variable. But this is also not really very practical.

SAT-Solvers

In the project description we talked about replacing the Prolog backend in SUTI1 [3] with a SAT backend. The hope was to get earlier an appropriate solution. So that we don't have to consider so many solutions with the heuristics. Moreover there was a chance for a second improvement. When the partly annotated programs were overconstraint, the Prolog solution only says *no*. We hoped that with a SAT backend we would get an answer like:
The annotation for field "x" is overconstraint. Check the already fixed annotations or change the program structure and start a new inference.

The boolean satisfaction problem is a decision problem. And the tools just tries to find a model for the provided boolean expression. It is completely tool dependent in which order the possible solutions are reported. Some tools not even report a solution they only say that the expression is satisfiable. This is the same as Prolog does.

Some SAT-solvers report the conflicting literal or clause when the boolean expression is not satisfiable. This conflicting literal or clause can be used to generate a hint for the user to change some predefined settings. This is an advantage over Prolog, but when allowing casts this will be a very rare situation.

There are two main categories of SAT algorithms.

- Davis-Putnam-Logemann-Loveland DPLL (deterministic)

  Chaff

  GRASP

  BerkMin

- Stochastic Local Search algorithms SLS (non-deterministic)

  WalkSAT

  GSAT

  HSAT

These are all high performance algorithms. There are many implementations available and most of them understand the DIMACS [17] CNF file format. So coding the Universe infering problem to the DIMACS CNF file format would provide a variety of tools to solve it. But unfortunately these tools would not retrieve much information. Because when we are infering the Universe type annotation and allow to inserts cast we already know that there exists a solution. So the result of the SAT-solver telling us *yes* there is a solution is worth nothing. When the solver returns us a model for the Universe infering problem then we have the same as we already had with Prolog, perhaps it is faster with a SAT solver. But the Prolog input is much more readable.

An ordinary SAT-Solver seams not to bring advantages over the old Prolog backend.

COMET

COMET [20] is a object oriented language that supports a constraint-based architecture for local search. COMET has a very sophisticated three layer design and can model stochastic local search (SLS) algorithms for solving complicated constraints satisfaction and optimization problems.

COMET provides quite all features we need to solve and optimize the constraints of the Universe infering problem. But using COMET requires to write our own SLS algorithm. Writing the SLS is not what we want. We are interested in a tool with a generic optimization algorithm into which we can easily encode the Universe infering problem.

The COMET system provides all the feature needed to relatively fast develop a SLS algorithm for a specific problem. But the effort to write a specific optimization algorithm seams to be too big.

UBCSAT Algorithms for SAT and MAX-SAT

UBCSAT is an implementation and experimentation environment for Stochastic Local Search (SLS) algorithms for SAT and MAX-SAT [15]. UBCSAT provides implementations of numerous well known SLS algorithms for SAT and MAX-SAT. Furthermore it provides many reporting and statistical features to analyze the algorithms. New algorithms may be implemented in UBCSAT efficient and straightforward.
UBCSAT is freely available at satlib.org [16].

As we stated in section 3.2 we are interested in an optimization problem. Therefore the SAT capabilities are not so interesting for us. We are interested in the MAX-SAT part of the UBCSAT tool. The MAX-SAT problem is a modification of the SAT problem. In the MAX-SAT problem the boolean formula is mostly over constrained and not satisfiable. Then a weight is associated with each clause in the boolean formula. The goal is now to maximize the sum of the weight of the satisfiable clauses.

Listing 3.2 shows the coding of a subtype relation between two types in the *WCNF*-file format. The WCNF file format is a straight forward extension of the DIMACS CNF file format [17]. The first number in each clause is the weight of the clause.

Listing 3.2: Subtype relation in wcnf format for MAX-SAT

```
1  c Test wcnf file
2  c 2 Bits per Type
3  c 1. bit: Type true -> rep; false -> peer
4  c 2. bit: Readonly bit true -> readonly
5  c subtype a <: b may introduce cast (?)a <: b
6  c a = {1,2}
7  c b = {3,4}
8  c subtype
9  c
10 p wcnf 4 17
11 c b[rep] -> (a[rep] or a[readonly])
12 10 -3 4 1 2 0
13 c b[peer] -> (a[peer] or a[readonly])
14 10 3 4 -1 2 0
15 c
16 c We prefer no cast
17 c b[!readonly] -> a[!readonly]
18 1 4 -2 0
19 c
20 c b is a field, we prefer b[rep]
21 1 -4 0
22 1 3 0
23 %
24 0
```

An advantage of encoding the Universe infering problem as a MAX-SAT problem is the modular tightening of the type system. As a core one can write down constraints which must be satisfied for a correct Universe infering problem model. That means when all the constraints in the core are satisfied the solution is correct. And now multiple layers of additional constraints may be built around this core. Each layer may add further constraints which lead to a more preferred solution. The more clauses around the core are fulfilled the better the solution is.
This modular tightening allows to add more and more relations we would like to be fulfilled.

The drawback of the MAX-SAT encoding is that it is not so easy to guarantee that the core constraints will all be satisfied. It is no problem to check if there are satisfied, but we want to force the tool to satisfy them. A solution were the core constraints are not satisfied is unusable. An idea to force the tool to fulfil the core is weighting the core constraint so high, that we can say that there is no solution with a bigger sum than a solution with all core constraints satisfied. This can be achieved with weighting each core constraint higher than the sum of all non-core constraits. When we have a big Universe infering problem this leads to huge weights for the core constraints and could make some problems.

There are other problems supporting strong and weak constraints which would better fit. The strong constraints must be satisfied and the weak constraints should be satisfied. But only dividing the constraints into strong and weak ones will not really provide a good tool to solve the Universe infering problem accurately.

PBS Incremental Pseudo-Boolean Backtrack Search SAT Solver and Optimizer

The pseudo boolean solver (PBS-tool) solves instances of 0-1 Integer Linear Programming (0-1 ILP) problems. These 0-1 ILP problems are boolean optimization problems. Traditionally these problems were handled as instances of generic ILP problems. 0-1 ILP problems call for the minimization or maximization the linear objective function $c^T x$ subject to a set of m linear constraints $Ax \leq b$ where $b, c \in \mathbb{Z}^n$, $A \in \mathbb{Z}^m \times \mathbb{Z}^n$ and $x \in \{0,1\}^n$. These constraints are commonly refered to as *pseudo-Boolean* (PB) inequalities. These PB constraints are a natural extention of the CNF constraints generally handled by SAT solvers.

The PBS-tool is an extended SAT-solver. It handels the basic CNF expressions and an additional file containing PB constraints and the objective function. Because the PBS-tool is based on a SAT solver it is better to provide as much constraints in CNF as possible. The constraints in the PB form are handled not so fast than the CNF ones.

The 0-1 ILP problem is a subtype of the general ILP problem. But the complexity of the 0-1 ILP still remains NP-hard. So it is important that we have a fast solver to handle a large instance of the Universe infering problem.

There exist also some other pseudo boolean solvers (PB-S). Some of them support the same input file format as the PBS-tool. An other format which is widely supported by PB-S is the OPB format. More information over the OPB format and other solvers is availabel at the Pseudo Boolean Evaluation 2005 web site [22].

As shown in the introductory example we use this tool for our implementation. We encode the whole typesystem in a CNF and use the PB feature only for the objective function to maximize by.

3.3 UTI Interface

The Static Universe Type Inference 1 project [3] introduced a Prolog abstraction of the whole Java program. This abstraction allowed the change of the Universe type system property without adapting the `ConstraintGenerator`.

In this project we introduce a new interface between the Java-client side and the Universe inferer side - he Universe Type Inferer (UTI) interface. This interface was already mentionend before and is an abstraction of a static Universe Type Inferer. In this section we show the details of this abstraction and the motivation.

3.3.1 Why Do We Need a New Abstraction?

Translating the Prolog representation to CNF seems not to be easier than writing a new `ConstraintGenerator` and generating the CNF from the JML syntax tree. So the Prolog abstraction seems only to be the appropriate choice when all the constraint solving is done in Prolog. But to support different backends using different technologies a more general and more comfortable interface seems to fit better.

The former project SUTI1 [3] made a simplification of the possible casts. This simplification was already discussed before in section 2.1.1. The Prolog Java abstraction is built on top of this simplification. This leads to a problem when loosening this simplification to allow the inferer to introduce both casts for a variable, so that a variable may be casted once to peer and once to rep.

Listing 3.3: Cast the same variable twice to different types

```
1  // class declaration
2  class MyClass {
3     ...
4    public peer Object foo(peer Object o){
5       ...
6    }
7    public void bar(){
8      MyClass mc = new MyClass();
9      rep Object res;
10     peer Object par;
11     res = mc.foo(par); //can this be typed?
12   }
13 }
```

Listing 3.3 shows a Java code segment were a problem may occur. This is a very synthetic example but it may also appear in common programs.

The partially annotated program shows a situation where a non-pure method `foo` is called on a unannotated local variable. The methods formal parameter and the return value are both peer types. The return value of `foo` is assigned to a `rep` variable. The actual parameter of the method call `foo` is of type `peer`. Is such a situation still typeable? In the Universe type system this situation is not allowed and therefore the situation is not typeable.
Let us have a look at the Prolog abstraction.

Listing 3.4: The prolog abstraction

```
query_MyClass([MyClass_foo_FormalParameter0, MyClass_foo_Return,
               MyClass_bar_Local_mc,
               MyClass_bar_Local_par,
               MyClass_bar_Local_res]) :-
    ...
    method_callparameters_right([[MyClass_bar_Local_mc]],
                [[[ MyClass_bar_Local_par]]],
                [[ MyClass_foo_FormalParameter0]],
                none),
    assignment([[MyClass_bar_Local_res]], [[ MyClass_bar_Local_mc], [MyClass_foo_Return]]),
```

The Prolog abstraction used in SUTI1 decouples the method call into two separate parts. The method call part handles the parameters and the purity of the method. The return type of the method is handled somewhere else. In this example the return type is handled by the assignment rule.

The infering tool from SUTI1correctly recognizes that this situation is not typable. But can we also support a less restricted casting behavior with this abstraction?

We now assume that the rule database or another inferer using this abstraction allows the casting of a `readonly` typed variable to `peer` and `rep`. The query generated for this program has only one not annotated reference. The reference on which the method `foo` is called. Solving the Prolog abstraction from listing 3.4 with such a relaxed cast behavior might give us the answer from listing 3.5.

Listing 3.5: Solve the query

```
?- query_MyClass([peer, peer,
                  MyClass_bar_Local_mc,
                  peer,
                  rep]).


MyClass_foo_FormalParameter0 = peer
MyClass_foo_Return = peer
MyClass_bar_Local_mc = readonly
MyClass_bar_Local_par = peer
MyClass_bar_Local_res = rep
```

The found annotation for the local variable `mc` in method `bar` is `readonly`. It is surprising that there exists a solution, as we said before there is no possibility to type this partially annotated program with the UTS. But both rules can be satisfied since they can be considered independently. The `method_call` rule is satisfied if `MyClass_bar_Local_mc` is casted to `peer`. To satisfy the `assignment` rule `MyClass_bar_Local_mc` must be casted to `rep`. This is possible since there is no information available, that the `mc` field must be casted in both rules with the same cast.

We think that the requirement of a general cast is legitimate and therefore the Prolog abstraction does not provide sufficient support for a more general inferer. It seems to be not such a big effort to extend the Prolog abstraction to support the more general cast policy. But extending the Prolog abstraction does not solve the fact, that the Prolog abstraction does not suit well to implement other not Prolog based inferer.

Therefore we decided to introduce the UTI interface to hopefully provide a enough abstract and comfortable interface between the Java program and the inferer to support an easy implementation of different infering backends. And more we hope that the UTI interface provides a good enough abstraction of the Universe inferer so that these inferer also can be used with different Java frontends.

3.3.2 UTI Overview

The Universe type inference (UTI) interface is an abstraction of a static Universe type inferer. The figure 3.2 gives an overview of the interface. The UTI interface describes how the inferer consumes the Java program on which it has to infer the Universe types. Also it provides some mechanism to interact with the inferer. Intermediate solutions can be reviewed and some hints may be told to the inferer. The UTI interface consists of the following components:

- `UtiController`
  The Controller builds the main interface between the client and the inferer. The controller is in the same Universe as the inferer. So it is possible to change the settings over the controller. The controller gives access to the concrete instances of all components.
- `UtiConstraintBuilder`
  The Constraint Builder defines the interface for providing the Java program structure. The inferer then builds up a constraint system.
- `UtiVariable`
  To each place in the Java program where a Universe modifier must be infered a `UtiVariable`

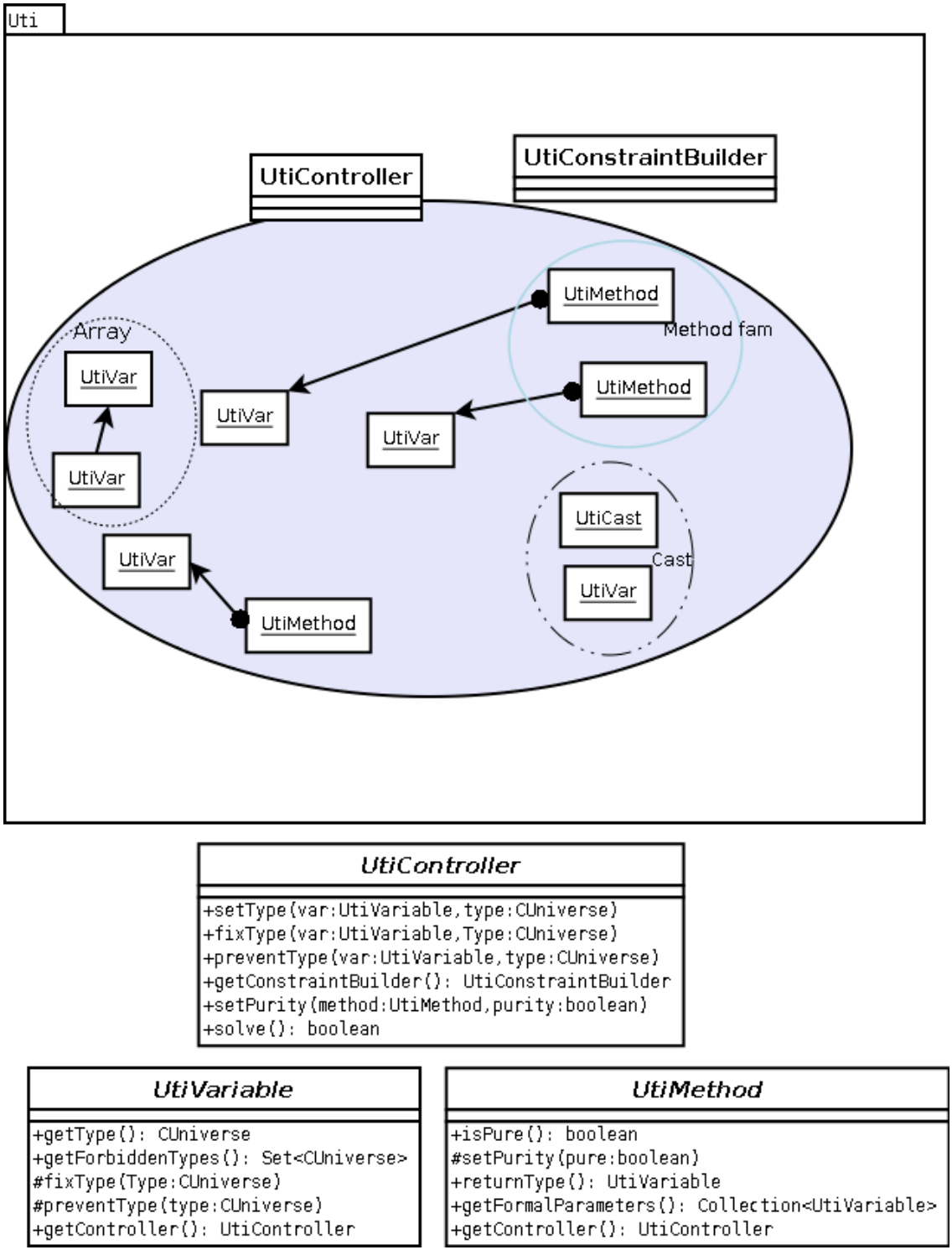

Figure 3.2: Overview of the UTI interface

is associated. This variable is a representant for this annotation.
The `UtiSingleVar` is an implementation of the `UtiVariable` for four non-variable cases.

- `UtiCast`
  Depending on the implementation of the `UtiConstraintBuilder` possible casts are inserted. For each possible cast a `UtiCast` is provided as representant.
- `UtiMethod`
  For each method there is a `UtiMethod` as representant. This representant may be used to change method purities and to form families of overridden methods.
- `UtiSolutionDescription`
  The Solution Description is used to provide a general description of the solution. This description then is translated by the implementation to a specific heuristic used to find the optimal solution.
- Some small helper classes
  Additionally there are some helper classes used by the other components.

3.3.3 The UtiController

The `UtiController` is the main component of the inferer. The Controller is designed as an abstract base class. It provides a static factory method `getController` to create an instance of a concrete controller. The class of the controller to instantiate is given as an argument. With the second parameter (which is a key value hash map) inferer specific options may be provided. These options depend on the actual inferer and are interpreted by the actual instance.

Building the Components

Once a concrete controller is created it provides a number of further factory methods. With these further factory methods most other components must be instantiated. This is done this way to allow each implementation of an inferer to use a specific implementation of the other components, so that they provide the used internal functionality. An other reason is that the controller is the owner of most of the other components. the other components are only provided to query some information from and to build up the program structure with, therefore the controller must create them.

Providing Writable Access

The design tries to obey the Universe type system. The `UtiController` and `UtiConstraintBuilder` are *peer* components. So these are the only components where the client may provide information to. The `UtiConstraintBuilder` is treated in section 3.3.7. This concludes that the controller must provide all necessary methods to the client to which it must be able to write. So the controller has some methods to set the properties of the `UtiVariable`, `UtiCast` and `UtiMethod`.

Usage

First we mention some additional necessary methods. Next to the already mentioned facilities it also has methods to set and retrieve the solution description. This description is used by the inferer to adjust the used heuristic when the Universe annotations are infered. And the last important method is the `solve` method which triggers the infering process.

The controller can be used the following way. First instantiate a controller representing the inferer. Then get the constraint builder. The constraint builder then is used to provide the program structure of the Java program on which the UTS annotation has to be infered. During building up these constraints the controller is used to create all the used variables, methods, and cast representants. After finishing building the constraints a first UTS annotation may be deduced. The fully annotated program then can be reviewed by a developer. After reviewing, several possible options are available.

- Accepting the solution.

  Nothing left to do for the inferer. Go on, and try to infer an other program.

- Do not accept the solution.

  Provide a modified solution description to the inferer.

  Change some annotations to a predefined type as hint.

  Prevent some annotations to get a defined type as hint.

  Disallow some casts or fix to a defined type.

  Change some method's purity.

If the solution is not satisfying several from the above mentioned actions can be combined. Then another annotation can be deduced by again calling the solve function. Some inferer may also find other annotations without changing anything. But this is implementation specific. This can be repeated till the solution is satisfying.

For a detailed description of the interface we refer to the Javadoc documentation and to the source code. An implementation is discussed later in section 3.5.

3.3.4 The UtiVariable

The `UtiVariable` represents what we are interested in, a single Universe type annotation. For each Universe type annotation which must be infered a `UtiVariable` object is created and associated with that Universe type annotation. The variable is owned by the controller. So it is read-only for the client. But it is still possible to query the UTI variable for the represented Universe type. Which is either *peer*, *rep*, or *readonly*.

The `UtiVariable` provides methods to change the properties. Since all instances are owned by the controller these methods cannot be used directly when the Universe type system is applied due to the fact that they are not pure. Therefore, the controller offers methods to set these properties. The variable offers a method to obtain the controller. Since the variables depend of the inferer instance they are only usable with exactly the instance which created them.

Arrays

To support arrays the `UtiVariable` instances can be combined. An array is represented with two variables. One variable represents the component annotation and the other variable represents the array annotation. The component variable is a normal variable. To create an array variable the controller offers a special factory method. This factory method needs as parameter a variable which represents the component type. The client needs not to distinguish between arrays with a primitive or reference component type. In figure 3.2 such an array representation is shown.

To represent the array only the array variable must be specified. If an array component access should be represented. This is done in an access chain by specifying the array variable followed by the component variable. Concrete examples are shown in section 3.3.7.

At the moment the UTS represents all array with one or two annotations. This seems to be comfortable for the programmer and adequate. If the future shows that for each nesting an annotation would be appropriate, this would still be supported by this mechanism.

The Universe Type Property

The UTI controller offers some methods to change the properties of the UTI variables. The following properties may be changed.

- The fixed Universe type.
  The Universe type represented by a variable may be fixed to a specified type. This fixing reduces the solution domain. This mechanism is used to support already partially annotated programs. And it is also used to change intermediate solutions.
- The prevented Universe types.
  Preventing a particular Universe type for an annotation my favorably augment the quality of the next deduced solution.

Fixing and preventing can be made undone by the `unfix` method. Preventing two different types has the same result than fixing to a definite type.

Degenerated UtiVariables

All the relations which affect the Universe annotations are expressed as relations among `UtiVariables`. To be able to express all these relations we introduced some degenerated types of UTI variables.

- Variables without Universe type.

  Variable representing the current current object (this, super).

  Variable representing a primitive type.

  Variable representing `null`.
- Variable with permanently fixed Universe types.

  Variable permanently representing *readonly*, used for `Strings` and all boxing types in Java 5.

We use these degenerated UTI variables to enable a simple straight forward Java program structure consumption mechanism. So a client must not worry to much how different Java constructs are preprocessed. It must only provide all requested information.

The UTI interface provides a simple implementation for all these variables in the class `UTISingleVar`. The name SingleVar indicates that these variables are thought to be used as single instance only. So the controller creates for each type one instance and returns for each request the corresponding instance.

3.3.5 The UtiCast

The `UtiCast` instances are representants for casts in the Java program. All casts which might be necessary to type a program are generated during the constraint building phase. A cast instance always is related to a `UtiVariable` instance. This UTI variable represents the Universe type to which the cast casts to. In figure 3.2 this relation is shown.

The number of inserted casts depends of the actual constraint builder instance. It is possible that no cast at all is inserted, when the inferer does not support casts. On the other hand the inferer may introduce a lot of possible casts. This is the case because the inferer must insert a `UtiCast` by the constraint building process for each cast it may use during the infering process.

The `UtiCast` has two properties the client may change between multiple infering processes. Each cast can be allowed or disallowed. When a cast is disallowed the inferer is not allowed to use this cast to find a valid Universe annotation. This property is introduced to forbid casts the Universe inferer may introduce and will fail at runtime. So disallowing all casts the inferer creates during the constraint building phase leads to the same solution as when the inferer would create no cast at all. And the second property it the type to cast to. A cast can be fixed or prevented to a specified type. This can be done via the related UTI variable. To change these properties the client must use the appropriate methods of the UTI controller.

After a solution was infered the UTI casts can be queried if they are actually used in the current annotation.

3.3.6 The UtiMethod

The `UtiMethod` instances are representants for the methods in the infered Java program. During the constraint building phase to each Java method an UTI method is associated. These methods are used to express implementation and overwriting relations among methods and for setting the methods purity. A method family is shown in figure 3.2. After an infering process there are no new information for the client in the methods. So they are only used to provide information to the inferer.

In Java it is possible to overload methods. The UTS does not allow an overloading where the difference is only in the Universe types. Due to this overloading restrictions all overwritten Java methods must remain overwritten methods after an annotation with the UTS. That concludes that overwritten methods must obtain the same Universe type annotations. To ensure this restriction we build up families of overwritten methods. These families of methods are then assigned with the same Universe types.

Methods overwriting an existing method may not become less pure than the overwritten method. The infering tool does not ensure this purity property, because the purity is information only provided by the client and not deduced by the inferer. So to ensure a correct UTS annotation, the purity information must be provided correctly by the client.

3.3.7 The UtiConstraintBuilder

The `UtiConstraintBuilder` (UtiCB) is an interface defining how the Java program structure must be provided to the inferer. This interface replaces the Prolog abstraction from SUTI1 [3]. Instead of providing the whole program structure at once the UtiCB interface is a builder. A client then provides to this builder the structure bit by bit and the chosen implementation of UtiCB may build the desired constraints as they are used to infere the Universe types.

The UtiCB is designed so that it should be straight forward to use it together with an abstract syntax tree from an arbitrary Java compiler. It provides a natural way to feed the program structure so it should also be easy useable in many other situations.

The UtiCB has two types of methods. One type of method is used to provide scope information. The other type of methods is used to provide statements which are relevant for the inferer.

- Scope Information
  This methods are used to reflect the scope hierarchy in a Java program. Differend scopes may be nested as it is in Java.
  - startConstraintBuilding
    Used to initialize and reset the constraint builder. Must be called before the program structure is provided.
  - endConstraintBuilding
    This method is called after all information is provided. Then the constraints for the program are generated and the next time the controllers `solve` method is called the Universe types for the new program are infered.
  - typeDeclarationStart
    Starts the declaration of a new type. This may be a class or an interface. The end of the type declaration is indicated with a closeScope method call.
  - methodDeclarationStart
    Starts the declaration of a new method. The end of the method declaration is indicated with a closeScope method call.
  - constructorDeclarationStart
    Starts the declaration of a new constructor. The end of the constructor declaration is indicated with a closeScope method call.
  - catchClauseStart
    Starts a new catch clause. The end of the catch clause is indicated with a closeScope method call.
  - scopeOpen
    Opens a new scope. This can be used by field initializers and for object initializers.
  - scopeClose
    Any opened scope must be closed with a call of this method.
- Statements
  With these methods different statements and declarations may be provided to the constraint builder.
  - fieldDeclaration
    Declare a field. This can be done in a type declaration scope. When the field has an initializer then the openScope and assignment methods can be used to provide this information.
  - variableDeclaration
    Declare a variable. This can not be done in a type declaration scope. For the initializer no new scope must be opened.
  - staticTarget
    Declare a static target. Static methods are normally called by a type. This type is called static target.
  - returnStatement
    This method is used to provide a return expression in a method scope.

- assignment
  All assignments are provided with this method.
- booleanExpr
  A comparison between two references restricts their Universe types. To reflect this restriction the booleanExpr method must be called for `==` and `!=` relations between references.
- methodCall
  With this method, method calls can be provided. Static methods should as target get a static target.
- constructArray
  Every creation of an array must be provided with this method. This are new array expressions and array initializers.
- objectCreation
  The creation of arbitrary objects is provided with this method.
- explicitConstructorCall
  This method builds the constraints for the explicit constructor calls.
- cast
  Already introduced casts are provided to the constraint builder with this method.
- instanceOf
  The instanceOf expression is provided with this method.

How is a Java Program Provided to the Constraint Builder?

Here we describe how a Java program is provided to the constraint builder. For all relevant parts in a Java program we show the necessary methods to call.
In section 3.6 a more detailed example is shown. In figure 3.11 is shown how a program structure is provided to the inferer.

Start and End of the Constraint Building for a Program

Before building the constraints the constraint builder must be prepared for building constraints for a new program.

```
void startConstraintsBuilding();
```

At the end when all necessary information is provided the constraint builder is informed to finish building the constraints and providing them to the conroller.

```
void endConstraintBuilding();
```

References, Null and Primitive Types

For each reference a `UtiVariable` must be created. Then every time a reference occurs in an expression this UTI variable must be used for the reference. For `String` and boxing type references a readonly `UtiVariable` must be used. This is motivated by the following properties. A `String` reference is defined as read-only in the UTS. In Java 5 the boxing conversion allows to automatically create instances of wrapper classes. If we state, that the instance is created in the Universe the static Universe type refers to, then these instances may be created in an arbitrary Universe. To prevent this, one must forbid the boxing conversion on read-only references. This leads to the

solution to set these boxing types to read-only as well. This brings no restrictions, because the wrapper classes are all immutable.

`null` must always be mapped to the null `UtiVariable`. The `this` reference is also mapped to a special `UtiVariable`, the this UTI variable.

Field and Array Access Chains

Field and array access chains are often used in expressions. When an access chain occurs in an expression it is represented as a `Java.util.List<UtiVariable>`. The UTI variables must have the same order in the list as the corresponding references in the program.
For an array in the access chain it is enough to insert the UTI variable representing the array because this UTI variable is composed with the element type and knows that it represents an array. Expressing an array access is done the same way as a field access. The UTI variable representing the array followed by the UTI variable representing the component type are inserted into the list.
It is important that such chains always represents correctly typed Java code.
In figure 3.11 two access chains are shown in the example.

Type Declarations

Each interface and class introduces a new type. To declare the fields and methods declared by them we open a scope with the method:

```
void typeDeclarationStart(MemberInfo info, UtiVariable encl);
```

Most information used for a type declaration is provided with the `info` parameter.
The `encl` parameter is only used by inner classes. At the moment the reference to the enclosing instance by inner classes is treated as read-only. In many cases it might be useful to have a peer relation between an instance of an inner class and the enclosing instance. The enclosing instance never can be in a rep relation to the inner instance because the enclosing instance must be created first. So at the moment it is necessary to provide a UTI variable fixed to *readonly* as the `encl` parameter. In the future a normal unfixed UTI variable might be used.

After providing all information we close the type scope with calling the method:

```
void scopeClose();
```

Method and Constructor Declarations

For each method and constructor a `UtiMethod` is created the first time it occurs. That may be in a method call expression, declaration or object creation expression. The formal parameters and return type are already needed by the creation of the representant. The purity of the method can already be provided by the creation of the representant or set at any time later. A constructor is defined with the method:

```
void constructorDeclarationStart(UtiMethod constructor);
```

This opens a scope to provide all information about the constructor's implementation. To define a method there is the method:

```
void methodDeclarationStart(UtiMethod method, List<UtiMethod> overriddenMethods);
```

This method also opens a new scope. The parameter `overriddenMethods` is the list of methods overridden and implemented by this method. This information is used to ensure that the methods keep all the same signature.

When all necessary statements of the method or constructor are provided the scope must be closed with:

```
void scopeClose();
```

Catch Clauses

Each catch clause has one formal parameter, the caught exception. This formal parameter is treated by the UTS as read-only. So a special readonly UTI variable must be provided.

```
void catchClauseStart(UtiVariable formalParam);
```

At the end of the catch clause the opened scope has to be closed.

```
void scopeClose();
```

Field Declarations

Fields can be declared in the scope of a type declaration. By the field declaration the UTI variable representing the field is linked with the necessary information. For example if the field is a static field. This information is provided with the `info` parameter.

```
void fieldDeclaration(MemberInfo info, UtiVariable field);
```

If the field has an initializer it must be provided in a special scope.
`void scopeOpen(boolean isStatic);`
By static fields a static scope must be opened.
Then the information for the initializer can be provided and at the end the scope must be closed again.

```
void scopeClose();
```

Variable Declarations

A variable can be declared in all scopes without a type declaration scope. For an initializer no new scope must be opened. There is no need to specify if the variable is static or not, because this is implicit provided by the scope. Variables declared in a static scope are static therefore the UTS restricts the possible types to *readonly* and *peer*.

```
void variableDeclaration(NameInfo info, UtiVariable var);
```

Assignments

Assignments must be provided to the constraint builder. When the left and right side of an assignment statement only consists of field access chains then the two lists representing these access chains are provided as parameter of the assignment method.

```
UtiCBReturnValue assignment(List<UtiVariable> left, List<UtiVariable> right);
```

The return value of this method contains an auxiliary UTI variable representing the type of the provided assignment statement. This can be used to build up composed assignment statements. Moreover it contains possibly by the constraint builder introduced casts.

If the expressions of the left or right side do not only consist of access chains then first these expressions must be processed, before the assignment method is called. The type of an expression is returned by the method used to provide the expression to the UtiCB. This expression type is then used to build up complex expressions.

For example the right side of a composed assignment is again an assignment. The right side of the second assignment consists only of a access chain. Before calling the assignment method for the first assignment expression the right side must be processed. To process the right side, the assignment method can be called. The parameters are the lists representing the access chains for the left and right side. Then the method for the first assignment can be called. The parameters are a list representing the access chain for the left side and a list containing the expression type received by the processing of the second assignment.

Equality and Not-Equality

The `==` and `!=` operators introduce also a restriction to the Universe types of the compared references. The types have to be convertable.

```
void booleanExpr(List<UtiVariable> left, List<UtiVariable> right);
```

The parameters are the same as by the assignment method. The return type is `void`. This is the case because it makes no sense to insert any casts and the expression type is always `boolean`, which is a primitive type without a Universe type modifier.

Method Calls

By a method call the method, the target and the actual parameters must be provided.

```
UtiCBReturnValue methodCall(UtiMethod method, List<UtiVariable> target,
  List<List<UtiVariable>> actualParam);
```

As by the assignment, first all subexpressions for the target and the parameters must be processed. Then the method call method can be invoked.

The `UtiCBReturnValue` returned by this method may contain the expression type of the processed method call. Moreover it may contain a cast for the target and a cast for each actual parameter. With the expression type a more complex expression can be build up.

Java allows to call static methods on references and on types. The UTS allows for each static method call a type annotation to say in which context the method is called. So we recommend to refactor the Java program so that static methods are always called explicitly on a type.
If static methods are called on references, the reference's Universe is used to execute the static method. A reference on which a static method is called cannot become read-only. If static methods are called on string or other read-only references then no annotation for the program can be found because the system will be overconstraint.

Static targets can be declared with the following method.

```
void staticTarget(NameInfo info, UtiVariable type);
```

Creation of Objects

In Java objects may be created on several ways.

- Array Creation:
  When an array is created the following method must be called.

  ```
  UtiCBReturnValue arrayCreation(UtiVariable type,
        List<List<UtiVariable>> initializer);
  ```

  - Array Initializer at declaration:
    This is the situation when an array reference is declared and initialized with an array initializer. The problem by this array creation is that it is not clear in which Universe the array is created. We state that the array is created in the Universe the declared reference points to. But this then forbids the initialization of read-only arrays, because the Universe to create the array is not defined.

    As the `type` parameter the UTI variable representing the declared reference is provided. For the `initializer` a lists of expressions is provided. The expressions are list of UTI variables. The expressions must again be processed before this method is invoked. If a nested array is initialized the expressions may again be array initializer. Then first the nested initializer must be provided. The nested array has then the same type as the toplevel array.
  - Array Creation Expression with facultative initializer:
    An array may also be created with an explicit new array expression. When no initializer is combined with the new expression then the `initializer` parameter is set to `null`. The array size expression must also be processed. If it contains again an array initializer then it must be provided the same way as above.

  The `UtiCBReturnValue` contains next to the expression type the possible inserted casts for the expressions in the array initializers.

- Class Instance Creation
  Class instances basically can be created implicit or explicit.

  - Explicit creation:
    By an explicit class instance creation it has to be specified in which Universe the instance is created.

    * Class instance creation expression.
      For most classes and all user defined classes an instance can only be created by a class instance creation expression. All non inner classes are created with an new expression consisting of a general constructor call. For this situations the `objectCreation` method can be used the same way as for a normal method call. The `type` parameter is set to the UTI variable representing the created type. The other additional parameters can be set to `null`.
      For inner classes the two additional parameter must also be used. With the `encl` parameter the UTI variable representing the relation to the enclosing instance must be provided. This is in the current state always a UTI variable which is fixed to read-only. The expression specifying the enclosing instance must also be processed and the expression type must the be provided by `thisExpr` parameter.

      ```
      UtiCBReturnValue objectCreation(UtiMethod constructor,
            UtiVariable type, List<List<UtiVariable>> actualParams,
            List<UtiVariable> thisExpr, UtiVariable encl);
      ```

      The return value contains a facultative cast for the `thisExpr` and a list of casts for the actual parameters. The expression type may be used to represent complex expressions.

- Implicit creation:
  By implicit instance creation we do not make any UTS annotations. Here are only `String` and boxing type instances created. These types are all set permanently to readonly.
  We do not invoke any method for the corresponding code fragments.
  * Loading of a class or interface that contains a String literal.
  * Execution of an operation that causes boxing conversion.
  * Execution of a string concatenation operator.

Explicit Constructor Calls

It is a common scenario to explicit call a constructor with the `super` or `this` key word. For an explicit constructor call the `explicitConstructorCall` must be called. The parameters next to the missing `type` parameter are set the same way as by a class instance creation expression. No `type` parameter is used, because the constructor is called on the current instance.

```
UtiCBReturnValue explicitConstructorCall(UtiMethod constructor,
    List<List<UtiVariable>> actualParams, List<UtiVariable> thisExpr,
    UtiVariable encl);
```

Also the return value has the same contents as by a object creation expression.

Casts and InstanceOf

Already introduced casts must also be handled by the inferer. By a cast the expression which is to cast must be processed first. Then the UTI variable representing the type to cast to and the expression is provided by following method:

```
UtiCBReturnValue cast(UtiVariable type, List<UtiVariable> expr);
```

The return value contains the expression type, which can be used to represent complex expressions. If the expression type is not fixed to `readonly` then also a UTI cast representing this cast is returned.

InstanceOf expressions are used to get the actual type of an object. The gained information is often used together with a conditional cast. There is no possibility for the constraint builder to link the `instanceOf` and a corresponding cast together, because it does not make control flow analysis. Therefore the inferer will in most cases annotate the `instanceOf` type with the static type of the corresponding expression.

```
void instanceOf(UtiVariable type, List<UtiVariable> expr);
```

If a client wants to set equal the type for an `instanceOf` operator and a cast it may use the same UTI variable to represent their type.

How to Handle All Other Statements

Generally speaking in all other statements the contained expressions must independently be processed according to the rules stated above. Expressions only containing primitive types can be ignored, as long the UTS does not consider them. But do not ignore an assignment of primitive types when it includes a field update, because the fields target must be writeable and that is considered by the UTS.

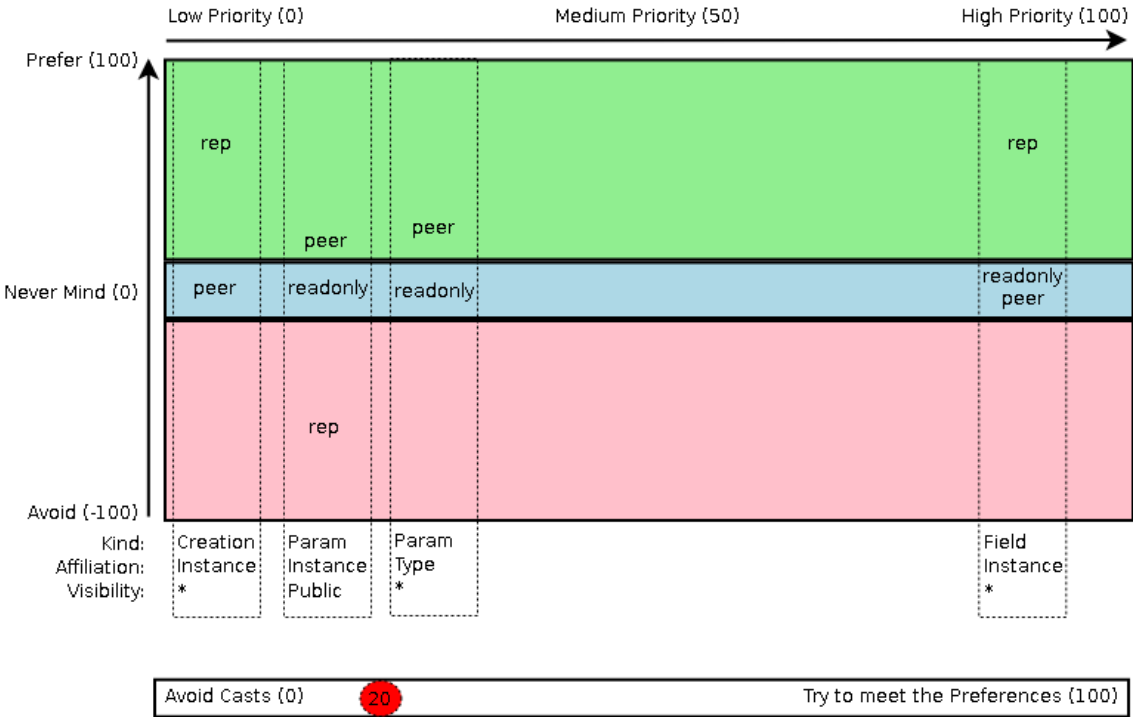

Figure 3.3: Describe the Universe annotations.

3.3.8 The Solution Description

The `UtiSolutionDescription` class represents a description of several properties of a Java program's object structure related to the Universe type system. The inferer then tries to annotate the program in such a way that the solution meets the requirements specified by such a solution description.

In the next section we describe how the solution description must be understand. It represents a way to state what the preferences are. Each inferer slightly differently may interpret these preferences. So we have a mechanism to deal with different inferer.

Interpretation of the Description

Two distinct properties can be specified.

- Annotation Preference
- Cast Acceptance

The annotation preference allows for each kind of Universe type annotation to specify what the preferences are. This means that one can state a relation between the allowed Universe type modifiers. The modifiers preference values must be placed in a finite range. The range goes from minus boundary to plus boundary. The boundary is set to 100. By default each preference is set to zero. Then setting a preference of a negative value means that we do not like that this kind of annotation is set to the modifier with this negative value. In figure 3.3 on the right side we can see the preferences for a Universe type modifier of an instance field. The *peer* and *readonly* modifier are left to zero. The *rep* modifier is set approximately to the value 60. This means that the field

should be annotated with *rep*, to gain a deep object structure. If it is not possible to annotate a field with *rep* then it does not matter if it is *readonly* or *peer*.
On the left side three more preferences are shown. The more right a preference is positioned on the figure the higher is the priority to fulfil this preference. This priority is also a value form 0 to boundary.

The different Universe type annotations are distinguished by three properties. Each of these three properties has a finite set of values.

- Kind: Specifies the situation where the annotation occurs.
  - Field: The declaration of a field.
  - Local: The declaration of a local variable. Here the affiliation is related to the scope in which the variable is defined.
  - Parameter: The decalration of a parameter. The affiliation is related to the method.
  - Return type: The return type. The affiliation is related to the method.
  - Creation: The explicit creation of new instances. The affiliation again is related to the scope where this expression occurs. For the creation only the *peer* and *rep* modifier can be related.
- Affiliation: States if the item or the current scope belongs to a instance or to a type.
  - Type: This are static fields or items defined in a static context. For all preferences the *rep* modifier is not allowed.
  - Instance: All non static situations.
- Visibility: States which access modifier is used for the item or the scope.
  - Public
  - Private
  - Protected
  - Standard access

For each kind of annotation we specify a relation between the different possible modifiers. The bigger the distance between the modifiers the more we prefer or avoid one modifier comparing to the other modifier. Therefore we state that it seams reasonable to always set at least one modifier to the value zero. This is also done this way in figure 3.3. It is possible that a much stronger relation with a lower priority receives more attention than the one with the higher priority by the inferer.

The cast acceptance is a simple second property. With the cast acceptance one can state how much casts are accepted to reach the prefered solution. The cast acceptance is a value between zero and boundary. In figure 3.3 the cast acceptance is shown at the bottom. It is set to the value 20 which means, that not so many casts should be allowed to get the prefered solutions.

Dealing with Different Inferers

Inferer may be implemented with completely different constraint solvers and optimization tools. They also may use different heuristics. It is the task of each inferer to map the general solution description to the internal mechanism. This mapping should be as intuitive as possible.

There will never be one solution description which will return the same result on all inferers but the results should be similar. To make the start easier for the user. Each inferer should at least provide one solution description which leads to general reasonable solutions. With this as start point a customization could lead to a fast success.

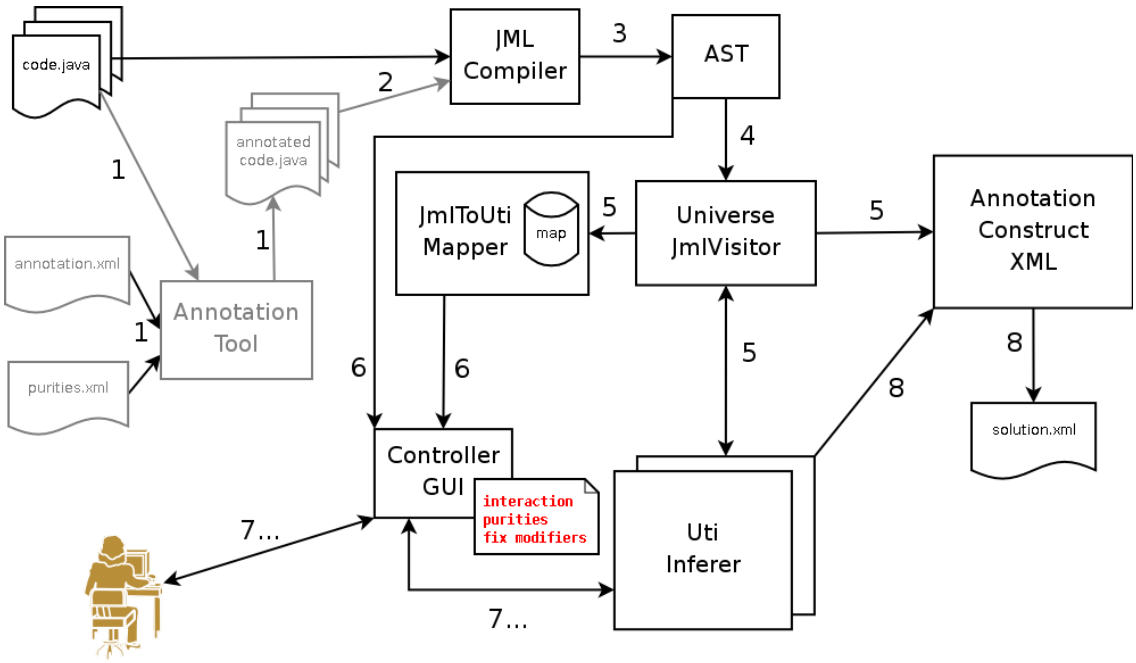

Figure 3.4: The architecture of the JML side,the UTI client.

3.4 JML Client

The JML side of the tool. The main function of the JML client is to translate the Java program into calls to the `UTIConstraintBuilder` interface and invoke the inference process.

3.4.1 Overview

The figure 3.4 shows the components of the JML side. The numbers next to the arrows indicate the order how of the information flows through the tool.

As input we have the Java source files of the program to annotate, some facultative xml files providing some Universe preannotation and an xml config file.

As output we generate again xml files containing the Universe annotations for the provided Java program.

Data Flow and Components

The provided Java source files are annotated with the preannotation given as input. To preannotate we use the annotation tool [12] elaborated by a former student project. Then the preannotated sources are parsed with the JML compiler. From the JML compiler we get the abstract syntax tree (AST).

The AST then is used to get access to the program structure. This can be used by future graphical user interface implementations. At the moment we use the AST in the UniverseJmlVisitor. This visitor translates the program structure provided by the AST to the necessary `UtiConstraintBuilder` interface method calls. At the same time the UniverseJmlVisitor also

builds up the annotation construct. The UniverseJmlVisitor uses the JmlToUtiMapper to map the UTI variables and methods to the corresponding JML objects from the AST.

The JmlToUtiMapper is also intended to be used later by components who need a mapping between the inferers representants and the AST objects. As an example a GUI could use the program information from the JML AST and get the corresponding UTI representants via the JmlToUtiMapper. It is also imaginable to use the mapper to update the AST with the infered Universe annotation for the Java program and then type check the annotated AST again to verify the found solution.

The AnnotationConstruct represents the whole information used to generate the annotation xml files. As the Universe modifiers are not known at the building time the UTI representants are stored instead. So this structure can then also be used to get access to the UTI representants by describing them with the same information as stored in the xml annotations files. This Structure is for example used by a textual user interface to interact with the inferer.

After all information to the UTI inferer is provided, the infering process may be started. In a non interactive run this is done by the controller. The controller then also lets the annotation construct generate the xml files and write them out.
In an interactive run the user may now interact with the infereing process. For this purpose we provide a textual user interface (TUI). With this TUI the user may invoke several infering processes and change the annotations. Intermediate results can be written to an annotation xml file or the code can be annotated and viewed in an text editor.

We tried to use as many already existing components as possible in the architecture. So we planed from the beginning to use the annotation tool instead of handling the annotation xml files directly. This was motivated by avoiding to implement the same functionality twice. Furthermore the xml format was not yet chiseled in stone so we only have to maintain one tool. To parse the Java source files we use the JML compiler as it was the case in the previous project [3].

3.4.2 Universe JML Visitor

The Universe JML Visitor is comparable with the Constraint Builder from the SUTI1 project [3]. But to translate the Java program to a Prolog file it builds up the constraints used by the inferer using the `UtiConstraintBuilder` interface. The mapping from the Java program to the builder method calls was explained in the previous chapter. And it is very similar to the mapping in the previous work. Therefore we only consider some differences to the previous work and important things.

Auxiliary Variables to Build Complex Expressions

The UTI interface introduced auxiliary variables to build up complex expressions. This means that when providing an expression the Universe JML visitor gets an uti variable representing the Universe type of the provided expression. So this UTI variable can then be used to comfortably build up complex expressions.

For example we look at the composed assignment `a = b = c;`. The assignment operator is right associative and therefore the operands group right to left `a = (b = c);`. So first the most right assignment expression `(b = c);` is processed. The result type of this assignment expression is the type of `b` after the capture conversion. For non generic types this is the identity. The UTI interface method consuming the assignment returns a UTI variable representing this type. Next the left assignment `(a = (b = c));` must be provided. This can be done using the UTI variable obtained by processing the first assignment.

This mechanism can be used straight forward for most composed statements. This leads to a very comfortable transition from the JML AST to the `UtiConstraintBuilder` interface.

Boxing Types and Strings

The `String` class is treated as a special type in the Universe type system. Strings are always typed as *readonly*. This is motivated by the immutable nature of the `String` class and enforced by the uncontrolled creation of `String` instances.

If the UTS is treated the same why as the Java types are, then this restriction about `String` types does not encounter a restriction in the expression power of the type system. But in the current JML implementation the *rep* type is handled more restricted as one would expect.

A *rep* reference is stated as read-only when accessing from another instance than the owner of the object. This allows no modification over the reference for other objects than the owner. It is also not possible to assign another object to the reference for not owner objects. This is the case because it is not possible to provide an assignment convertable reference, or expression. But one can assign `null` to the reference. `null` is assignable to all references, because null is a suptype of each reference type and therefore assignment compatible. *The direct supertypes of the null type are all reference types other than the null type itself.* [4]

This holds as long as the `null` type does not get a Universe modifier. If the `null` expression must be in the same Universe as the reference type, then the whole story gets a little bit different. And I would guess that it can be arguable to say that it is not allowed to nullify a rep reference through an access chain.

The current JML implementation does not allow nullifying a *rep* reference. Therefore the restriction of the `String` type introduces some loss of expression power. With this restriction it is no longer possible to protect a `String` reference on object granularity from being nullified by another object than the owner.

If nullifying would be allowed by the Universe type system for *rep* references accessed over an accessing chain, then we could type the types supported by the boxing conversion permanently as *readonly*without loss of the expression power. This is the case because all types supported by the boxing conversion are immutable as the `String` class is.

Permanently typing the types as *readonly* can be understood as creating them in the global Universe or in a special read-only Universe. This makes only sense for immutable types. The special positioning of these types is motivated by the uncontrolled or better unannotated creation of instances of these types. To handle them explicitly by the Universe type system would require a huge amount of unnecessary annotations as these types are not mutable. Therefore we handle handle the following types as read-only.

- String
- Boolean
- Byte
- Character
- Short
- Integer
- Long
- Float
- Double

Array Initializers

Array initializers may occur at two different situations in a Java program.

- In an array declaration.
  Object[] oa = {**new** Bar(1), **new** Bar(2)};
- In an array creation expression.
  oa = **new** Object[] {**new** Bar(1), **new** Bar(2)};

There are two things we have to consider when handling the array initializers. How do we handle the nested nature of array initializers and which type do we use to type the array initializer.

How to type the Initializer The initializer in the context of a new array expression can simply be typed with the type specified by the new expressions. There is no reason to make something else.
oa = **new rep peer** Object[] {**new** Bar(1), **new** Bar(2)};
The array initializer must be typed as **rep peer**. This works the same way as for any normal reference type.

For initializers in an array declaration context we might run in some ambiguity situations.
rep peer Object[] oa = {**new** Bar(1), **new** Bar(2)};
This declaration declares a **rep peer** Object[]. The array created by the initializer must also be of Universe type **rep peer** to be assignment compatible with the declared reference. Also the specification of the Java programming language says that the initializer creates an instance of the declared type. This includes that the created array is of type Object[] and not of type Bar[]. So to be consequent we also have to say that the created instance is of type **rep peer** Object[]. This represents a seamless integration of the Universe type system into the Java type system.

But there is a problem with defining the initializers this way. Let us have a look at the following situation.
readonly readonly Object[] = {"Hello", "Universe", "Where␣are␣you"};
Now a *readonly* instance must be created. This is not possible because we do not know in which Universe we have to create the instance. Therefore we must forbid the array initializers for *readonly* arrays. This enforces that *readonly* array declarations are initialized by an new array expression instead of an array initializer.

Here it is worth to note that this restriction has some impact to the possible found solutions by the inferer. Some array references meant to be *readonly* are now not allowed to be *readonly*, which might have strange effects.

Handling of Nested Arrays The UTS uses two modifiers for reference array types. This is independent from the array's dimension. Therefore the type for each nesting stage is the same. When processing nested array initializers we respect that with ensuring the same type for each array initializer.

Field Hiding and Shadowed Items

In Java it is allowed to hide a field with a field of an arbitrary type. Therefore we will not introduce a more restricted behavior. In the previous work hidden fields were restricted to have the same Universe type as the hidden one. We see no motivation for such a restriction and handle it the way it is described by the Java language specification [4].

Shadowed items must not be considered specially by the Universe type system. Shadowing is name specific and therefore handled as usual. The problem of overloading and overwriting methods is considered in 3.3.

The motivation for this is the goal to seamless integrate the UTS in the Java type system so that developers don't have to reason about two different behaviors.

Enclosing Instances of Inner Member Classes

The enclosing instances of inner classes are handled using a *readonly* reference at the time . In the UTI interface the reference to the enclosing instance can be explicitly specified by the client. This explicit specification was introduced to support a more liberal handling of the enclosing instance in the future.

The Universe JML visitor provides all information needed to build up the access chains for enclosing instance fields. But fixes the UTI variables to *readonly*. So it is very easy to support more liberal behavior. Fixing these references to *rep* would lead to an overconstraint system, because *rep* references are not allowed for enclosing instances.

The arbitration how a inner class is related to the enclosing instance is done when the classes are designed and will not change when used in another context. Therefore it seems to be sufficient to allow a specification about how to access the enclosing instance for the class declaration. So all instances of the class have the same modifier to their enclosing instance. It seems not to be necessary to allow a specification for the enclosing modifier for each instance. Further more it might be really bad. The specification in the class declaration could be achieved by a JML declaration. A specification like: /∗@enclosing peer@∗/ or /∗@enclosing readonly@∗/ might be imaginable in the class declaration head.

We do not say that this is necessary because it is possible to cast the reference to enclosing instances, but it would allow to type the program more precisely and provide more static information to the verifiertools.

To ensure that the enclosing instance is of the specified type, the creation of inner classes must be handled more complete. The primary expression by a qualified class instance creation expression must result in an assignment compatible type to the specified type. If the inner class is created by an unqualified class instance creation expression, then type of the reference to the inner most common enclosing instance must be assignment compatible to the specified type.

We have no chance to simply insert a cast if we have unqualified class instance creation expressions. So this situation must be avoided.

3.4.3 Annotation Construct

The annotation construct was introduced to show how the UTI variables can be used to get access to the Universe modifiers of the annotated types. The UTI interface provided the ability to infer the type annotations in one step. So it was no longer necessary to traverse the AST a second time, to deduce the actual modifiers from the result received from the Prolog backend.

But we still had the task to write the annotations at the end to the xml file. Writing the data to the xml file would again require to traverse the AST to collect all the necessary information next to the modifiers used in the file. So we decided to make it different. We introduced the annotation construct, which is a structure containing all the information used to generate the xml file. And for the modifiers we reference the UTI variables. This structure is then build up by the Universe JML visitor in the same run when building the constraints.

As xml is a widely used format there exist many tools and components to work with it. Such tools are able to read in the xml schema files and generate Java code to write and read in xml files obeying the provided schema. With the generated code it is easy to build up a structure representing the contents of the xml file as our implementation does. So it seems that the work invested in implementing the annotation construct was a bad investment.

The problem with the generated code was, that it does not work together with the UTI variables and that it is not yet clear at constraint building time where casts must be introduced. The work around for this two limitations would not be much simpler than the implementation of the annotation construct.

Later in the project phase we extended the annotation construct's interface and used it also as to provide the information for to textual user interface.

3.5 UTI Implementation with a PB Solver

Here we explain a UTI implementation using the PBS-tool. The pseudo boolean solvers are motivated in section 3.2.1. In this part the actual inferer is implemented. This inferer can be handled through the UTI interface and is the main part infering of the tool.

3.5.1 Overview

In figure 3.5 the different components of the inferer are shown. The solid arrows indicate how the data flows between the components. The architecture consists of the components known from the UTI interface. Additionally there are components to represent the internal constraints and the PB solver.

The inferer generates a `VarConstraint` for each UTI variable in the constraint building phase. This constraint stores how the variable is related to other variables. All constraints generated during the building phase are stored in the `PbsConstraints` component. This component is also the owner of all var constraints. Therefore the var constraint can only be modified over this component.

After the constraint building phase, the solving procedure may be invoked. To solve the constraints system a PB solver is used. The PB-S requires input files in a specific file format. To generate the required files the basic constraints must be provided to the boolean representation.This component is responsible for the boolean encoding. The boolean representation uses then an boolean writer which writes the encoded system to the files in the required format.

After the generation of the required files, the PB-S is invoked to solve the constraint and deduce an optimal solution. If the constraint system is not over constraint it writes out a solution. Then the controller must provide the found boolean solution to the boolean representation component, because the controller does not know, how the Universe type system was encoded. The boolean representation then decodes the solution.

The internal constraints generated by the building process are no longer modified after the building process. So the internal constraints must be aware of the additional properties which are not fixed. These properties are the method purity, disallowed casts, and fixed or prevented types. When the inferer generates the input files for the solver these changeable properties are considered. So each time the solve method is invoked, and some of these soft properties have been changed the files must be generated again to represent the new situation.

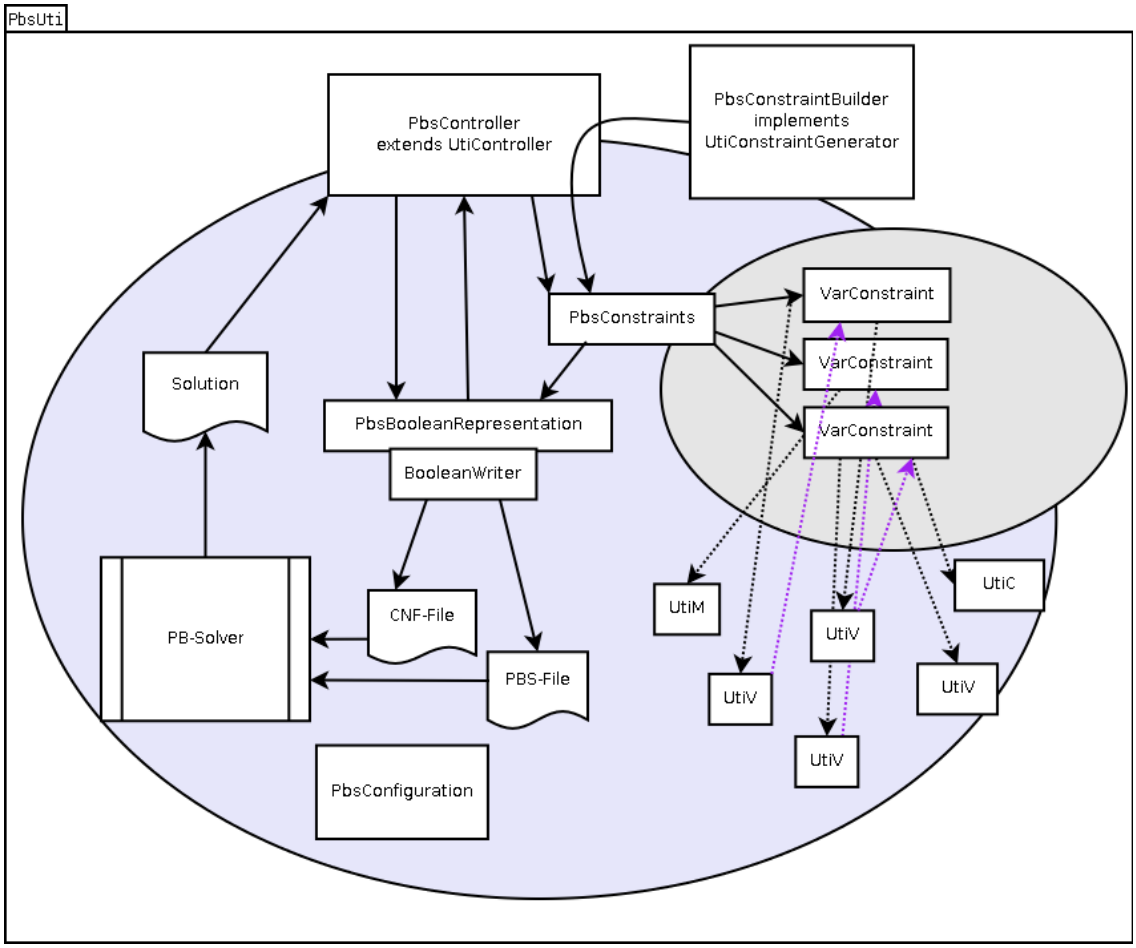

Figure 3.5: The architecture of the UTI implementation with PBS.

3.5.2 Exchangeable Components

The inferer consists generally of five components, where three of them are exchangeable.

- The controller: Together with the PB-S this component builds the basis and is not meant to exchange.
- The constraints: The constraints are also central. They are not meant to exchange.
- The constraint builder: Can be exchanged to interpret the UTS differently, or to implement another cast policy.
- The boolean representation: For each boolean representation a dedicated implementation can be used.
- The boolean writer: To write the boolean encoding in a specific file format different writers can be used.

The boolean representation will be the component which is exchanged the most. We provide with our work multiple different implementations of the boolean representation. The constraint builder might also be interesting to exchange. A constraint system might be build up with no support for casts. So the amount of used variables is much smaller.

3.5.3 The Constraints

All the information used to infer the Universe type system annotations is expressed with five kinds of constraints.

- Definition
- Subtype
- Convertable
- Cast
- Combination

The system built up with these five constraints is then transformed into the boolean CNF formula, before it is solved. In other words the boolean representation must only know these five constraints. This leads to a fast and simple implementation of different boolean representations.

In the next sections we explain the semantics of each of the five constraints. How a program is mapped to this constraints is shown later.

Definition

The definition constraint can be used to restrict the possible types for a variable. But it is also intended to generate some basic clauses used in the boolean representation for each type. So this constraint must be supplied for each variable at least once to the CNF generator even if the variable is not restricted.

The cast policy *allowBothCast* for the defined variable can also be specified in the definition constraint. The cast policy is not used by all implemented boolean representations. It is also not

used to specify if a variable is allowed to cast or not. For this the cast constraint is used. With the cast policy, one can specify if the defined variable may be casted to *rep* and *peer*. Actually the extended conflict boolean representations is the only representation using this feature. This does not surprise as the extended conflict boolean representation is the only one supporting such casts.

Subtype

The subtype constraint consists of two variables. It expresses a subtype relation among the two variables.

```
String subtype(UVariable sub, UVariable sup);
```

- sub <: sup

The clauses generated for the subtype constraint must ensure that `sub` is subtype of `sup`. It must not be a proper subtype. If `sup` is an array type then also `sub` must be an array. It is not necessary the other way round, because array is subtype of object. If `sup` is an array the subtype relation must hold for the array and the element type.

Convertable

The convertable constraint also consists of two variables. It expresses that the two variables must be convertable to each other.

```
String convertable(UVariable var1, UVariable var2);
```

- var1 <: var2 or
- var1 :> var2

Two types are convertable if it is possible to cast one type to the other, using a casting conversion. If both variables are arrays then the arrays must be convertable.

Cast

The cast constraint allows to insert a cast. The cast constraint consists of two variables and a cast representant. The type to cast to, the target type which is to cast and the cast representant. The cast representant is used as identifier for this cast. If the cast is used in the found solution, then the boolean representation must mark this cast as used.

Arrays can be involved on multiple ways in a cast. If only the `type` is an array then we cast an object to an array. In this situation we have no information over the element type. The element type must be handled in the boolean representation to conform to the used internal type system's cast behavior. If only the target is an array, then we want to assign the array to an object reference. Then only the array type must be considered. If two arrays are involved, then the array must be handled completely.

```
String cast(UVariable type, Cast cast, UVariable target);
```

The generated clauses must ensure the following properties:

- type = target or
- type <> target and cast is used

This means, that the *target* must have the same type as the *type* or the cast must be marked as used. We must allow upcast to be able to handle cast's which may be inserted by a developer.

Combination

The combination constraint consists of three variables. It expresses the Universe type combination.

```
String combination(UVariable var1, UVariable var2, UVariable var3);
```

- var1 * var2 = var3

The clauses generated for the combination constraint must ensure that `var3` represents the combination of `var1 * var2`.

3.5.4 The Heuristic

The heuristic is part of the boolean representation. We use a pseudo boolean solver to find an optimal solution for the generated boolean encoding. The optimal solution is described by an objective function. The coefficients for the objective function are generated during the boolean encoding out of the provided solution description.

Because the objective function is a linear pseudo boolean function we have some limitations. The coefficients of the objective functions are signed integers. Where the sign is interpreted as a boolean not. The absolute value of the coefficients is then multiplied with the facultative negated boolean value of the corresponding literal. The literal has the value zero if false and one if true.

For example we encode the three Universe types with two boolean values. We have to chose a fix representation. With two bits four combinations are possible. For *peer* and *rep* we use two combinations. For the read-only type we must decide if we want use two bit patterns or only one and forbid one pattern. To express our preferences we have two coefficients we can specify. With these two coefficients we cannot express all preferences. Because there can always be a preference we must express like: Coef[Type3] = Coef[Type1] + Coef[Type2], where the Coef[?] must always be a positive integer. So it is not possible to express all our preferences. It depends on the hamming distance between the different type encodings which preference we have to express like that.

It gets more complicated if we want to encode more states with three bits. Then states get weighted that we don't want to be weighted and the effect on the whole system is not predictable. To solve this problem one can use one bit for each state to weight. But this needs more literals to encode the same problem. We will introduce different encodings and in the testing section we will see if there are significant differences.

3.5.5 Internal Universe Type System

In this section we describe the different internal type systems we used to infer the Universe types. For each type system there may be several boolean representations implemented. It is not always

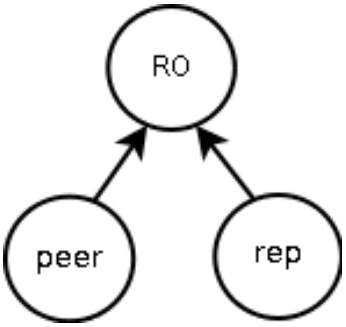

Figure 3.6: Universe type hierarchy.

clear if a compact representation is better than a larger one. In a short representation it is sometimes difficult to estimate how the different weights influence each other. In a longer representation one can introduce a bit for each property and then the weights may be calculated straight forward. But we need more bits to represent the program.

We introduce three internal type systems. The first one is the static Universe type system as it is. The second one is a conflict type system similar to one that was already used by SUTI1[3]. The third type system is an extended version of the conflict type system. The extended version allows to support general casts.

Static Universe Types

The static Universe types system consists of the three types *peer*, *rep* and *readonly*. In figure 3.6 the type hierarchy is shown. It does not support any casts.

Semantics This type system has exactly the Universe type system semantics. The subtype relation is shown on figure 3.6. The type combinator is shown in the table below.

Type combinator:

*	peer	rep	readonly
peer	peer	readonly	readonly
rep	rep	readonly	readonly
readonly	readonly	readonly	readonly

- Subtype: The *readonly*type is supertype of the other two types. Each type is supertype of itself. The *peer*and *rep*types are not related.
- Convertable: Two types are convertable if they are not *peer* and *rep*. This is ensured using a convertable constraint.
- Cast: Casts are not allowed. For a cast constraint both types must be equal and the cast is never used.
- Combination: The combination constraint ensures, that the three types behave according to the combinator table.

Encoding To encode this internal type system we have basically two possibilities. We can encode the three states compactly or use one different bit for each type. The compact encoding uses two bits per type and the other uses three bits per type. As we do not support any casts with this internal representation, we also need no bits for the casts. We have no information to encode, as we know there is no cast used.

Here we decided to show both encodings. First the compact one and then the one with three bits and the possibility to express all preferences. For each constraint we show which clauses must be generated to ensure their properties. At the end we show how the preferences from the solution description are expressed.

The Two Bit Encoding To encode the types with two bits we use a type bit (tBit) and a read-only bit (roBit). If the roBit is set then the type represented is *readonly*. If the roBit is not set, then the type bit *tBit* indicates if it is *peer* or *rep*. To reduce the feasible region we set the type bit to false if it is read-only.

Type encoding with two bits:

Type	tBit	roBit
peer	false	false
rep	true	false
readonly	false	true

The five constraints:
On the left side we show in a higher logic what properties must hold and on the right hand we show the necessary clauses which must be generated to ensure this properties.

- Definition $def(T1)$
  We forbid invalid bit combinations.

  $T1[roBit] \rightarrow \neg T1[tBit]$: $(\neg T1[roBit] \vee \neg T1[tBit])$

  Disallow forbidden types:

  Not *peer*: $(T1[tBit] \vee T1[roBit])$
  Not *rep*: $(\neg T1[tBit] \vee T1[roBit])$
  Not *readonly*: $(T1[tBit] \vee \neg T1[roBit])$

  The clause used to disallow *readonly* could be simplified to $(\neg T1[roBit])$. But writing it the long way, makes it easier to implement.

- Subtype $T1 <: T2$
  We use three implications to express the subtype relation. If $T2$ is an array we generate the rules for both variables.

  $T1[readonly] \rightarrow T2[readonly]$: $(\neg T1[roBit] \vee T2[roBit])$ $\wedge$
  $T2[peer] \rightarrow T1[peer]$: $(\neg T2[tBit] \vee T2[roBit] \vee T1[tBit])$ $\wedge$
  $(\neg T2[tBit] \vee T2[roBit] \vee \neg T1[roBit])$ $\wedge$
  $T2[rep] \rightarrow T1[rep]$: $(T2[tBit] \vee T2[roBit] \vee \neg T1[tBit])$ $\wedge$
  $(T2[tBit] \vee T2[roBit] \vee \neg T1[roBit])$

- Convertable $T1 <:> T2$
  Two normal types are convertable when they are not peer and rep.

$\neg(T1[peer] \wedge T2[rep])$: $(T1[tBit] \vee T1[roBit] \vee \neg T2[tBit] \vee T2[roBit])$ $\wedge$
$\neg(T1[rep] \wedge T2[peer])$: $(\neg T1[tBit] \vee T1[roBit] \vee T2[tBit] \vee T2[roBit])$

For the array convertable property we need some more restrictions. Here we do not show the rule on the bit level. The $T1E[?]$ stands for array element type part of $T1$.

$\neg(T1[peer] \wedge T1E[readonly] \wedge T2[readonly] \wedge T2E[peer])$ $\wedge$
$\neg(T2[peer] \wedge T2E[readonly] \wedge T1[readonly] \wedge T1E[peer])$ $\wedge$
$\neg(T1[rep] \wedge T2[readonly] \wedge T2E[peer])$ $\wedge$
$\neg(T2[rep] \wedge T1[readonly] \wedge T1E[peer])$ $\wedge$

- Cast $(T1)T2$
  Both types must be equal. This also counts for arrays. If we cast an object to an array we set the array element type to *readonly*.

$T1[peer] \rightarrow T2[peer]$: $(T1[tBit] \vee T1[roBit] \vee \neg T2[tBit])$ $\wedge$
$(T1[tBit] \vee T1[roBit] \vee \neg T2[roBit])$ $\wedge$
$T1[rep] \rightarrow T2[rep]$: $(\neg T1[tBit] \vee T1[roBit] \vee T2[tBit])$ $\wedge$
$(\neg T1[tBit] \vee T1[roBit] \vee \neg T2[roBit])$ $\wedge$
$T1[readonly] \rightarrow T2[readonly]$: $(T1[tBit] \vee \neg T1[roBit] \vee \neg T2[tBit])$ $\wedge$
$(T1[tBit] \vee \neg T1[roBit] \vee T2[roBit])$

- Combination $T1 * T2 = T3$

$T1[readonly] \rightarrow T3[readonly]$: $(T1[tBit] \vee \neg T1[roBit] \vee \neg T3[tBit])$ $\wedge$
$(T1[tBit] \vee \neg T1[roBit] \vee T3[roBit])$ $\wedge$
$T2[readonly] \rightarrow T3[readonly]$: $(T2[tBit] \vee \neg T2[roBit] \vee \neg T3[tBit])$ $\wedge$
$(T2[tBit] \vee \neg T2[roBit] \vee T3[roBit])$ $\wedge$
$T2[rep] \rightarrow T3[readonly]$: $(\neg T2[tBit] \vee T2[roBit] \vee \neg T3[tBit])$ $\wedge$
$(\neg T2[tBit] \vee T2[roBit] \vee T3[roBit])$ $\wedge$
$T1[peer] \wedge T2[peer] \rightarrow T3[peer]$: $(T1[tBit] \vee T1[roBit] \vee T2[tBit]) \vee T2[roBit] \vee \neg T3[tBit]$ $\wedge$
$(T1[tBit] \vee T1[roBit] \vee T2[tBit]) \vee T2[roBit] \vee \neg T3[roBit]$ $\wedge$
$T1[rep] \wedge T2[peer] \rightarrow T3[rep]$: $(\neg T1[tBit] \vee T1[roBit] \vee T2[tBit]) \vee T2[roBit] \vee T3[tBit]$ $\wedge$
$(\neg T1[tBit] \vee T1[roBit] \vee T2[tBit]) \vee T2[roBit] \vee \neg T3[roBit]$ $\wedge$

Expressing the solution preference:
The determinated weighting for each type must be assigned to the corresponding bits. There is no way to map all weightings exactly to the two bits.

The Three Bit Encoding To encode the types with three bits we use for each type a bit and state that only one bit might be true at a time. For *peer* (pBit), for *rep* (rBit) and for *readonly* (roBit). Even if we have now three bits the feasible region is not bigger than before.

Type encoding with three bits:

Type	tBit	rBit	roBit
peer	true	false	false
rep	false	true	false
readonly	false	false	true

The five constraints:
On the left side we show in a higher logic which properties must hold and on the right hand we

then show the necessary clauses which must be generated to ensure this properties. In contrast to the two bit encoding we use here the fact, that always only one bit can be true. Arrays are treated the same way as before.

- Definition $def(T1)$
  We forbid invalid bit combinations.

$$
\begin{array}{lll}
T1[pBit] \rightarrow \neg T1[rBit] \wedge \neg T1[roBit]: & (\neg T1[pBit] \vee \neg T1[rBit]) & \wedge \\
 & (\neg T1[pBit] \vee \neg T1[roBit]) & \wedge \\
T1[rBit] \rightarrow \neg T1[pBit] \wedge \neg T1[roBit]: & (\neg T1[rBit] \vee \neg T1[pBit]) & \wedge \\
 & (\neg T1[rBit] \vee \neg T1[roBit]) & \wedge \\
T1[roBit] \rightarrow \neg T1[pBit] \wedge \neg T1[rBit]: & (\neg T1[roBit] \vee \neg T1[pBit]) & \wedge \\
 & (\neg T1[roBit] \vee \neg T1[rBit]) &
\end{array}
$$

  This ensures that only one bit can be set at a time. Expressing this as a pseudo boolean formula would be much simpler. $1 * T1[pBit]\ 1 * T1[rBit]\ 1 * T1[roBit] = 1$. The sum of all bits must be 1, that ensures exactly what we need.
  Disallow forbidden types:

$$
\begin{array}{ll}
\text{Not } peer: & \neg T1[pBit] \\
\text{Not } rep: & \neg T1[rBit] \\
\text{Not } readonly: & \neg T1[roBit]
\end{array}
$$

  Here we only use the bit necessary to specify the rule.

- Subtype $T1 <: T2$
  We use three implications to express the subtype relation.

$$
\begin{array}{lll}
T1[readonly] \rightarrow T2[readonly]: & (\neg T1[roBit] \vee T2[roBit]) & \wedge \\
T2[peer] \rightarrow T1[peer]: & (\neg T1[pBit] \vee T2[pBit]) & \wedge \\
T2[rep] \rightarrow T1[rep]: & (\neg T1[rBit] \vee T2[rBit]) &
\end{array}
$$

- Convertable $T1 <:> T2$
  Two types are convertable if they are not peer and rep.

$$
\begin{array}{lll}
\neg(T1[peer] \wedge T2[rep]): & (\neg T1[pBit] \vee \neg T2[pBit]) & \wedge \\
\neg(T1[rep] \wedge T2[peer]): & (\neg T1[rBit] \vee \neg T2[rBit]) &
\end{array}
$$

  Here we have the same extension for arrays as by the two bit encoding.

$$
\begin{array}{ll}
\neg(T1[peer] \wedge T1E[readonly] \wedge T2[readonly] \wedge T2E[peer]) & \wedge \\
\neg(T2[peer] \wedge T2E[readonly] \wedge T1[readonly] \wedge T1E[peer]) & \wedge \\
\neg(T1[rep] \wedge T2[readonly] \wedge T2E[peer]) & \wedge \\
\neg(T2[rep] \wedge T1[readonly] \wedge T1E[peer]) & \wedge
\end{array}
$$

- Cast $(T1)T2$
  Both types must be equal.

$$
\begin{array}{lll}
T1[peer] \rightarrow T2[peer]: & (\neg T1[pBit] \vee T2[pBit]) & \wedge \\
T1[rep] \rightarrow T2[rep]: & (\neg T1[rBit] \vee T2[rBit]) & \wedge \\
T1[readonly] \rightarrow T2[readonly]: & (\neg T1[roBit] \vee T2[roBit]) &
\end{array}
$$

- Combination $T1 * T2 = T3$

$$
\begin{array}{lll}
T1[readonly] \rightarrow T3[readonly]: & (\neg T1[roBit] \vee T3[roBit]) & \wedge \\
T2[readonly] \rightarrow T3[readonly]: & (\neg T2[roBit] \vee T3[roBit]) & \wedge \\
T2[rep] \rightarrow T3[readonly]: & (\neg T2[rBit] \vee T3[roBit]) & \wedge \\
T1[peer] \wedge T2[peer] \rightarrow T3[peer]: & (\neg T1[pBit] \vee \neg T2[pBit] \vee T3[pBit]) & \wedge \\
T1[rep] \wedge T2[peer] \rightarrow T3[rep]: & (\neg T1[rBit] \vee \neg T2[pBit] \vee T3[rBit]) &
\end{array}
$$

Expressing the solution preference:
Since we have for each type a bit we can weight, expressing the preferences is straight forward. The preprocessed values can be directly assigned to the corresponding bit. There are no weights interacting with other states.

The rules for the three bit encoding are much simpler than the rules for the two bit encoding. The problem to solve is the same. So we tend to say that a three bit encoding is advanced over the two bit encoding since it is much easier to handle.

Static Universe Types with Safe Casts

The type system introduced before can not handle all situation, where we can statically show that everything goes well, but casts are needed. An extension was already motivated by SUTI1. This internal type system is intended to handle sparse annotated programs and tries to deal with the read-only parameter of pure methods. This type system can only ensure that the casts will always work, if all code can be processed. But even if not all code can be processed, we have still a better chance to get a working annotation.

In contrast to the conflict type system introduced in SUTI1 this system here is much more restricted. It is only a way to treat read-only types as writable types. The internal type system we present afterwards is more similar to the system we know from SUTI1.

The class shown in listing 3.6 is not typable with the static Universe type system without casts. The pure method `goNLinksFurther` must have read-only parameters, because the method is pure. Therefore the return type also is read-only. Since the return type is read-only the method `insertAT` can not modify the obtained link. And now we have an over constraint system. A programmer easily sees that the return value of the `goNLinksFurther` can be casted to a writable type without going into a risk. We want to create the whole link structure in one Universe. This is achieved with a *peer*type for the `next` field in this class. Then we know that we always stay in the same Universe. And if we supply a *peer*reference to the `goNLinksFurther` method we know that the result still must be in the *peer*context.

With an extended internal type system we can handle such situations without introducing dangerous casts. We only inserts casts which will work at runtime with the above mentoned restriction. To achieve this we make three read-only types.

- *NC*: Not Castable
  This is a not castable read-only type.
- *RoP*: Read-only Peer
  This is a read-only type but we know that it refers in the current context.
- *RoR*: Read-only Rep
  This is a read-only type but we know that it refers in the context owned by this.

Listing 3.6: Not typeable with without casts.

```
public class Chain {
  Object stored;
  Chain next;
  public /*@ pure @*/ Chain getNextLink(){
    return next;
  }
  public /*@ pure @*/ Chain goNLinksFurther(Chain link, int n){
    for(int i = 0; i < n; i++) {
      if(link != null) {
        link = link.getNextLink();
      }
    }
    return link;
  }
  public void insertAt(Object toStore, int i){
    Chain link = goNLinksFurther(this, i);
    if(link != null) {
      link.stored = toStore;
    }
  }
}
```

Semantics Actually the *RoP* and *RoR* types are *peer* and *rep* types which are only called *readonly*. Figure 3.7 shows the type hierarchy of this type system. The dashed lines show the allowed casts.

Type combinator:

*	peer	rep	ro-peer	ro-rep	NC
peer	peer	NC	ro-peer	NC	NC
rep	rep	NC	ro-rep	NC	NC
ro-peer	ro-peer	NC	ro-peer	NC	NC
ro-rep	ro-rep	NC	ro-rep	NC	NC
NC	NC	NC	NC	NC	NC

- Subtype: The *NC* is supertype all other types. Each type is supertype of itself. The peer and rep types are subtypes of the related read-only type.
- Convertable: Two types are convertable when they are not *peer* and *rep*. This must be ensured by a convertable constraint. The convertable constraint could be formulated stronger. But this is ok, since it ensures correct type annotation for the UTS.
- Cast: Two different casts are allowed. *RoP* to *peer* and *RoR* to *rep*. If a cast is used this must be marked. When no cast is used, the same type for both sides must be ensured.
- Combination: The combination constraint ensures, that the three types behave according to the combinator table.

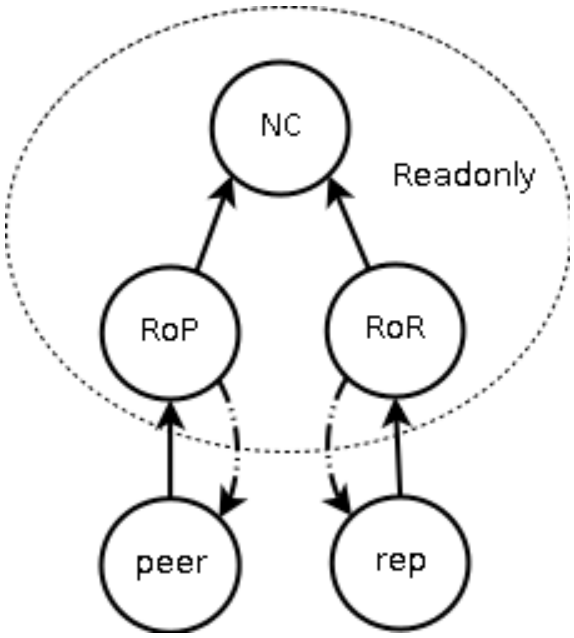

Figure 3.7: A type hierarchy with save casts.

Encoding This internal type system has five types. So we need at least three bits to encode each type. We can also use one bit for each state so that we need five bits. With such a five bit encoding the rules are quite simple and we can express the preferences correct. Since we are not interested to weight intermediate variables and the additional types of the internal type system we can also use a hybrid encoding.

Here we will introduce the 3 + 2 bit hybrid encoding. The properties of this encoding are:

- Ability of precise preferences expression.
- Support for four read-only and four writable types.
- Uses only three bits for intermediate variables.

These properties can be achieved with the following trick. We use a compact three bit encoding where we fix one bit to the read-only bit. The read-only bit is used to switch between the four read-only and writable types. The other two bits of the compact encoding can be used to encode all necessary states of the internal type system. The precise weighting of the properties we get with two additional weighting bits for non intermediate variables. These two weighting bits are only used by the definition and for the objective function. The peer weighting bit is always set true if the compact encoding encodes the type peer and the rep weighting bit is set true if the compact encoding encodes the type rep.

With such a representation we could save some bits and keep the full control over our preferences. The drawback is that we must maintain two different encoding lengths which requires a more complicated infrastructure. If the saved bits are worth the overhead must be considered in big test cases.

Next to the bits used for the types we also need one bit for the casts.

We now show the different type encodings. And then we show for each constraint in a higher level logic how to ensure their properties. The translation to the clauses is done the same way as

before and not shown here. To handle the arrays, a straight forward extension can be used, which we don't show here. At the end we show how the preferences from the solution description are expressed.

The encoding with three bits consists of a type bit (tBit), a read-only bit (roBit) and a cast bit (cBit). The cast bit is set to true if it is possible to cast the read-only type to the corresponding writable type. The read-only bit is set to true if it is a readonly type. And the type bit is used to distinguish between *peer* and *rep*. For the casts we simply use one bit which is set to true if used. All bit combinations which are not specified are forbidden to keep the feasible region small.

Type encoding with three bits:

Type	tBit	roBit	cBit
peer	false	false	false
rep	true	false	false
ro-peer	false	true	true
ro-rep	true	true	true
NC	false	true	false

The encoding with five bits is straight forward. We use for each type the corresponding bit. Here we disallow most combinations. The cast is also represented with one bit. With this encoding we can express the preferences correctly.

Type encoding with five bits:

Type	pBit	rBit	ropBit	rorBit	ncBit
peer	true	false	false	false	false
rep	false	true	false	false	false
ro-peer	false	false	true	false	false
ro-rep	false	false	false	true	false
NC	false	false	false	false	true

The 3 + 2 bit encoding uses the 3 bit encoding introduced above. Additionally we introduce the weight peer bit (wpBit) and the weight rep bit (wrBit). These additional bits are only used for variables we want to weight. We can leave them away, because the whole internal type system is expressed with the first three bits. With this hybrid encoding we can save bits and get a precise weighting capability. The cast is expressed the same way as before and the not specified bit combinations are prevented to keep the feasible region small.

Type encoding with 3 + 2 bits:

Type	tBit	roBit	cBit	wpBit	wrBit
peer	false	false	false	true	false
rep	true	false	false	false	true
ro-peer	false	true	true	false	false
ro-rep	true	true	true	false	false
NC	false	true	false	false	false

The Encoding of the Five Constraints

- Definition $def(T1)$
  We forbid invalid bit combinations for each encoding. For the 3 + 2 bit encoding this includes the binding of the weighting bits to the corresponding types.

  Disallow forbidden types:

Not *peer*:	$\neg T1[peer]$
Not *rep*:	$\neg T1[rep] \wedge \neg T1[ro - rep]$
Not *readonly*:	$\neg T1[nc] \wedge \neg T1[ro - peer] \wedge \neg T1[ro - rep]$

This high level logic formulas are reformed and optimized for each implementation if necessary. The *RoR* type must also be forbidden when *rep* is forbidden.

- Subtype $T1 <: T2$
  The subtype relation is now a bit more complicated.

$$
\begin{array}{ll}
T1[NC] \rightarrow T2[NC] & \wedge \\
T2[peer] \rightarrow T1[peer] & \wedge \\
T2[rep] \rightarrow T1[rep] & \wedge \\
T2[ro - peer] \rightarrow \neg T1[NC] & \wedge \\
T2[ro - peer] \rightarrow \neg T1[ro - rep] & \wedge \\
T2[ro - peer] \rightarrow \neg T1[rep] & \wedge \\
T2[ro - rep] \rightarrow \neg T1[NC] & \wedge \\
T2[ro - rep] \rightarrow \neg T1[ro - peer] & \wedge \\
T2[ro - rep] \rightarrow \neg T1[peer] &
\end{array}
$$

- Convertable $T1 <:> T2$
  In the UTS are two types are convertable if they are not peer and rep. Is this restricted enough for that type system to ensure that the program will work? We should restrict it even more with applying the Java definition of convertable to our type system. If the convertable rule is only used in boolean expressions, it might be ok. Let us go the save way.

$$
\begin{array}{ll}
\neg(T1 <: T2) & \vee \\
\neg(T1 > T2) &
\end{array}
$$

- Cast $(T1)T2$ *is used* C
  It is allowed to cast the read-only types to the corresponding writable types. In all other cases the types must stay equal and the cast is not used.

$$
\begin{array}{ll}
T2[NC] \rightarrow T1[NC] & \wedge \\
 & \\
T2[peer] \rightarrow \neg T1[rep] & \wedge \\
T2[peer] \rightarrow \neg T1[ro - rep] & \wedge \\
T2[peer] \rightarrow \neg T1[nc] & \wedge \\
(T2[peer] \wedge \neg C[bit]) \rightarrow \neg T1[ro - peer] & \wedge \\
 & \\
T2[rep] \rightarrow \neg T1[peer] & \wedge \\
T2[rep] \rightarrow \neg T1[ro - peer] & \wedge \\
T2[rep] \rightarrow \neg T1[nc] & \wedge \\
(T2[rep] \wedge \neg C[bit]) \rightarrow \neg T1[ro - rep] & \wedge \\
 & \\
T2[ro - peer] \rightarrow \neg T1[rep] & \wedge \\
T2[ro - peer] \rightarrow \neg T1[ro - rep] & \wedge \\
T2[ro - peer] \rightarrow \neg T1[nc] & \wedge \\
(T2[ro - peer] \wedge \neg C[bit]) \rightarrow \neg T1[peer] & \wedge \\
 & \\
T2[ro - rep] \rightarrow \neg T1[peer] & \wedge \\
T2[ro - rep] \rightarrow \neg T1[ro - peer] & \wedge \\
T2[ro - rep] \rightarrow \neg T1[nc] & \wedge \\
(T2[ro - rep] \wedge \neg C[bit]) \rightarrow \neg T1[rep] &
\end{array}
$$

- Combination $T1 * T2 = T3$

$$
\begin{array}{ll}
T1[NC] \rightarrow T3[NC] & \wedge \\
T2[NC] \rightarrow T3[NC] & \wedge \\
 & \\
T2[rep] \rightarrow T3[NC] & \wedge \\
T2[ro-rep] \rightarrow T3[NC] & \wedge \\
 & \\
T1[peer] \wedge T2[peer] \rightarrow T3[peer] & \wedge \\
T1[peer] \wedge T2[ro-peer] \rightarrow T3[ro-peer] & \wedge \\
T1[rep] \wedge T2[peer] \rightarrow T3[rep] & \wedge \\
T1[rep] \wedge T2[ro-peer] \rightarrow T3[ro-rep] & \wedge \\
 & \\
T1[ro-peer] \wedge T2[peer] \rightarrow T3[ro-peer] & \wedge \\
T1[ro-peer] \wedge T2[ro-peer] \rightarrow T3[ro-peer] & \wedge \\
T1[ro-rep] \wedge T2[peer] \rightarrow T3[ro-rep] & \wedge \\
T1[ro-rep] \wedge T2[ro-peer] \rightarrow T3[ro-rep] & \wedge
\end{array}
$$

Expressing the solution preference For the five bit and the 3 + 2 bit encodings we can just assign the necessary values to the corresponding bits. In the compact three bit representation we must try to find a bit weighting which represents the preferences best. This must be done with some testing and is considered more in the implementation.

Universe with Conflict Types

This internal Universe type representation also consists of five different states. A very similar representation was introduced by SUTI1 [3]. This representation is a kind of degenerated type system. The subtype relation does not introduce a partial order among the different types. This anomaly is introduced to prevent bad casts. Therefore we tend to speak here of an internal representation instead of an internal type system. To support our needs we had to change the behavior of this system a little. With this representation less restriced casts are supported, but also the probability of cast a failure is higher.

In this representation the read-only type is split up in three different states. In contrast to the conflict representation introduced by [3] we renamed the *readonly* state to *not castable*. This was done to clearly show the differences between the Universe type *readonly* and the internal splitting up of this type in three states. The three read-only states are listed below:

- *NC*: Not castable
  This state can not be casted to *peer* or *rep*. It is a super "type" to all other types.

- *CP*: Conflict peer
  This state can be casted to *peer*. It is a super "type" to *peer*. And it is assignment compatible to *NC*.

- *CR*: Conflict rep
  This state can be casted to *rep*. It is a super "type" to *rep*. And it is assignment compatible to *NC*.

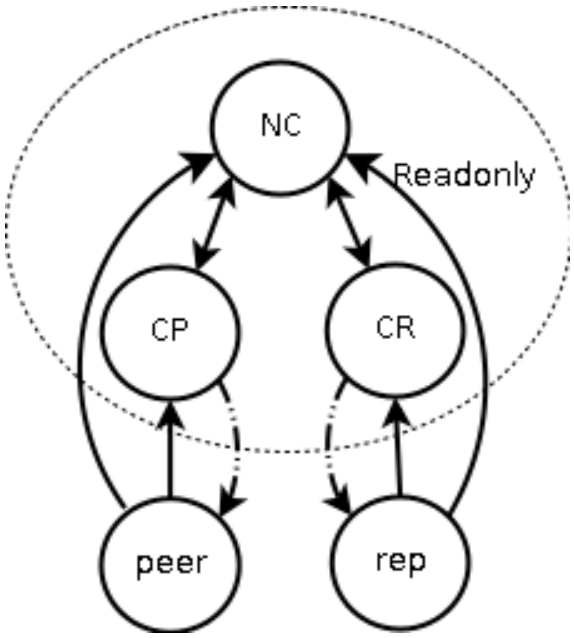

Figure 3.8: Type hierarchy with two conflict states as used by SUTI1.

Semantics This internal representation implements the Universe type system and allows some casts. The type hierarchy is shown in figure 3.8. The solid arrows show the assignment compatibility relation. The dashed arrows show the possible casts.

The idea of this representation is to allow for a reference to be typed as *readonly* and then cast and assigned by either *peer* or *rep* but not both of them. This restriction eliminates many possible casts which will fail at runtime.

The type combinator introduced by [3] for this type system is not useful for us, because we have to insert the casts at predefined positions and don't want to traverse the AST a second time. Therefore we do not cast down in the combinator, but keep the information ready for an introduced cast.

Type combinator:

*	peer	rep	Cpeer	CRep	NC
peer	peer	NC	CPeer	NC	NC
rep	rep	NC	CRep	NC	NC
CPeer	CPeer	NC	CPeer	NC	NC
CRep	CRep	NC	CRep	NC	NC
NC	NC	NC	NC	NC	NC

The combinator is quite the same as the original one. Any combination with *NC* results in *NC* as the original *readonly*. Any combination containing a conflict type result in the conflict version of the original result type when it not results in *NC*. The rest stays the same.

As the other parts of the inferer do not know what internal type or states we use, there are no external rules to change. But to be complete and ensure a correct handling we interpret restrictions in the range of possible types for an annotation as follows.

- *peer* not allowed:
  Only the simple *peer* type is not allowed. It might still be possible to cast a reference annotated with such a type to *peer*.
- *rep* not allowed:
  The simple *rep* and the *CRep* type are not allowed.
- *readonly* not allowed:
  All three internal states for *readonly* are prevented.

It is not necessary to make more restricted prevention of the conflict types as in [3]. This is the case, because we do not automatically insert casts in the combination function. All inserted casts are checked and therefore the allowed types for casts can be specified directly. But to handle the assignments correctly we also forbid the *CRep* if *rep* is forbidden.

Because of the assignment compatibility among the readonly types, we lose information about assignments. This loss of information may introduce a solution which will fail at runtime, but this compatibility is used to type more programs.

The changes of this type system in contrast to [3] did not change the capability of typeable programs.

Encoding This internal representation has also five states. Therefore we can encode it with 3 bits, 3 + 2 bits and 5 bits. And one additional bit for the casts.

We now show the different type encodings. And then we show for each constraint in a higher level logic how to ensure their properties. The translation to the clauses is done the same way as before and not shown here.

The encoding with three bits is achieved the same way as above. Only the interpretation is different. With this encoding we have problems to express the preferences.

Type encoding with three bits:

Type	tBit	roBit	cBit
peer	false	false	false
rep	true	false	false
CPeer	false	true	true
CRep	true	true	true
NC	false	true	false

The encoding with five bits is straight forward. We use for each type the corresponding bit. With this encoding we can express the preferences correctly.

Type encoding with five bits:

Type	pBit	rBit	cpBit	crBit	ncBit
peer	true	false	false	false	false
rep	false	true	false	false	false
CPeer	false	false	true	false	false
CRep	false	false	false	true	false
NC	false	false	false	false	true

The 3 + 2 bit encoding uses the 3 bit encoding introduced above. And the additional weight bits. The additional bits are only used for variables we want to weight. Here we get again the hybrid properties.

Type encoding with 3 + 2 bits:

Type	tBit	roBit	cBit	wpBit	wrBit
peer	false	false	false	true	false
rep	true	false	false	false	true
CPeer	false	true	true	false	false
CRep	true	true	true	false	false
NC	false	true	false	false	false

The Encoding of the Five Constraints

- Definition $def(T1)$
  We forbid invalid bit combinations for each encoding. By the 3 + 2 bit encoding this includes the binding of the weighting bits to the corresponding types.

  Disallow forbidden types:

  Not *peer*: $\neg T1[peer]$
  Not *rep*: $\neg T1[rep] \wedge \neg T1[CRep]$
  Not *readonly*: $\neg T1[nc] \wedge \neg T1[CPeer] \wedge \neg T1[CRep]$

  This high level logic formulas are transformed if necessary.

- Subtype $T1 <: T2$
  The subtype relation is a little bit more unrestricted.

  $T2[peer] \rightarrow T1[peer]$ $\wedge$
  $T2[rep] \rightarrow T1[rep]$ $\wedge$
  $T2[CPeer] \rightarrow \neg T1[CRep]$ $\wedge$
  $T2[CPeer] \rightarrow \neg T1[rep]$ $\wedge$
  $T2[CRep] \rightarrow \neg T1[CPeer]$ $\wedge$
  $T2[CRep] \rightarrow \neg T1[peer]$

- Convertable $T1 <:> T2$
  Here we use again twice the subtype rule.

  $\neg(T1 <: T2)$ $\vee$
  $\neg(T1 > T2)$

- Cast $(T1)T2$ *is used* C
  It is allowed to cast the conflict-types to the corresponding writable types. The same as before.

  $T2[NC] \rightarrow T1[NC]$ $\wedge$

  $T2[peer] \rightarrow \neg T1[rep]$ $\wedge$
  $T2[peer] \rightarrow \neg T1[cRep]$ $\wedge$
  $T2[peer] \rightarrow \neg T1[nc]$ $\wedge$
  $(T2[peer] \wedge \neg C[bit]) \rightarrow \neg T1[cPeer]$ $\wedge$

  $T2[rep] \rightarrow \neg T1[peer]$ $\wedge$
  $T2[rep] \rightarrow \neg T1[cPeer]$ $\wedge$
  $T2[rep] \rightarrow \neg T1[nc]$ $\wedge$
  $(T2[rep] \wedge \neg C[bit]) \rightarrow \neg T1[cRep]$ $\wedge$

$$
\begin{array}{ll}
T2[cPeer] \rightarrow \neg T1[rep] & \wedge \\
T2[cPeer] \rightarrow \neg T1[cRep] & \wedge \\
T2[cPeer] \rightarrow \neg T1[nc] & \wedge \\
(T2[cPeer] \wedge \neg C[bit]) \rightarrow \neg T1[peer] & \wedge \\
 & \\
T2[cRep] \rightarrow \neg T1[peer] & \wedge \\
T2[cRep] \rightarrow \neg T1[cPeer] & \wedge \\
T2[cRep] \rightarrow \neg T1[nc] & \wedge \\
(T2[cRep] \wedge \neg C[bit]) \rightarrow \neg T1[rep] &
\end{array}
$$

- Combination $T1 * T2 = T3$

$$
\begin{array}{ll}
T1[NC] \rightarrow T3[NC] & \wedge \\
T2[NC] \rightarrow T3[NC] & \wedge \\
 & \\
T2[rep] \rightarrow T3[NC] & \wedge \\
T2[CRep] \rightarrow T3[NC] & \wedge \\
 & \\
T1[peer] \wedge T2[peer] \rightarrow T3[peer] & \wedge \\
T1[peer] \wedge T2[CPeer] \rightarrow T3[CPeer] & \wedge \\
T1[rep] \wedge T2[peer] \rightarrow T3[rep] & \wedge \\
T1[rep] \wedge T2[CPeer] \rightarrow T3[CRep] & \wedge \\
 & \\
T1[CPeer] \wedge T2[peer] \rightarrow T3[CPeer] & \wedge \\
T1[CPeer] \wedge T2[CPeer] \rightarrow T3[CPeer] & \wedge \\
T1[CRep] \wedge T2[peer] \rightarrow T3[CRep] & \wedge \\
T1[CRep] \wedge T2[CPeer] \rightarrow T3[CRep] & \wedge
\end{array}
$$

Expressing the solution preference For the five bit and the 3 + 2 bit encodings we can just assign the necessary values to the corresponding bits. In the hybrid encoding also the constraits for the weighting bits must be generated if the bits are weighted.

Extended Conflicts

In section 2.1 we gave an example which is not typeable with the internal type systems seen till now. With the extended conflicts internal type system we introduce a system, which allows to type all programs, as long as they are typeable and will not fail for sure at runtime. It is not a problem, to be able to type all programs. If we would allow to cast from *readonly* to *peer* and *rep* in the static Universe type system, it would also be possible to type all programs. But with such a strategy we would get almost for sure an annotation which will never work at runtime, because the introduced casts will fail with a high propability.

With the extended conflicts internal representation we try to introduce a system capable to handle all preannotated programs without the insertion of too many bad casts. Bad casts are casts which will fail at runtime for sure or too often.

To achieve this goal we extended the conflict internal representation with an additional internal state. With this step, we try to keep as much information as possible. We partition the not casteable type *NC* into two new states.

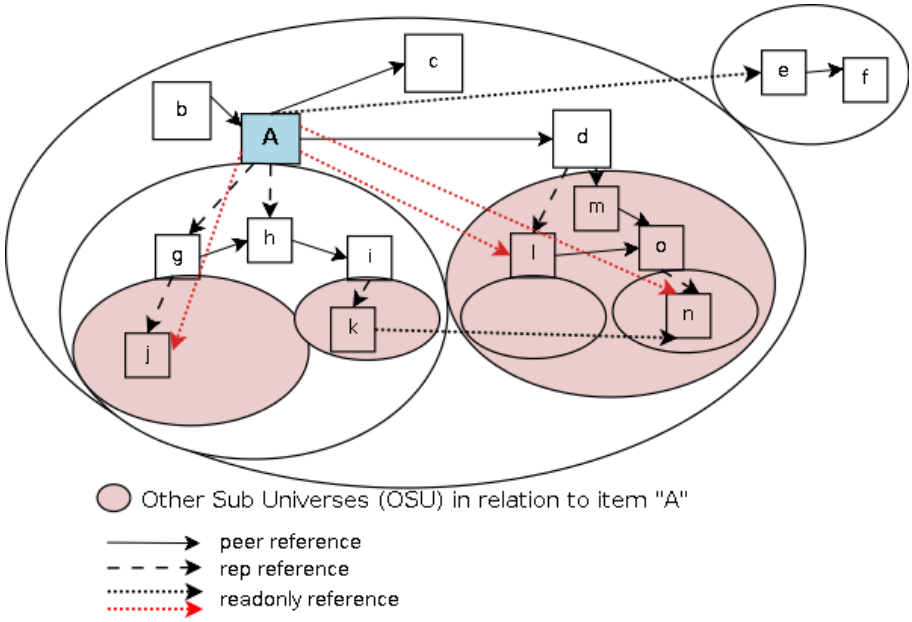

Figure 3.9: The *OSU* other sub Universe type.

- *CC*: Conflict Conflict
  The conflict conflict type is a *readonly* type which is castable to *peer* and *rep*. When this type is used we have no information over what is covered by it.
- *OSU*: Other Sub Universe
  The other sub Universe state specifies a type which cannot be casted for sure. A reference with that type references an object which is in a Universe where we cannot get a writable reference to. See figure 3.9. For objects typed with this type we have the knowledge that they can not be writable. We have widened the definition for *OSU*. We only want to ensure that not castable references are typed with *OSU*. But the *OSU* state is also supertype of all other states.

Semantics This internal representation is extended by the *OSU* state. With this extension we will be able to type all programs and reduce the danger of infering an unusable type annotation.

The *OSU* type semantic is visualized in 3.9. The *OSU* type stands for references which point into Universes which are owned by objects of our representation or Universes which are owned by peer objects of us and all sub Universes of the two before mentioned Universes. For this type we know that we cannot cast it to a writable type.

Rejected extended conflicts type interpretation As described the *OSU* state enforces that the contents must not be castable to a writable type. This requirement leads to the following properties.

One non *OSU* *readonly* reference is enough to lose all the information, because they might refer to a Universe we have write permission. And the information stored in a *OSU* type is, that we know it is not castable.

Type combinator:

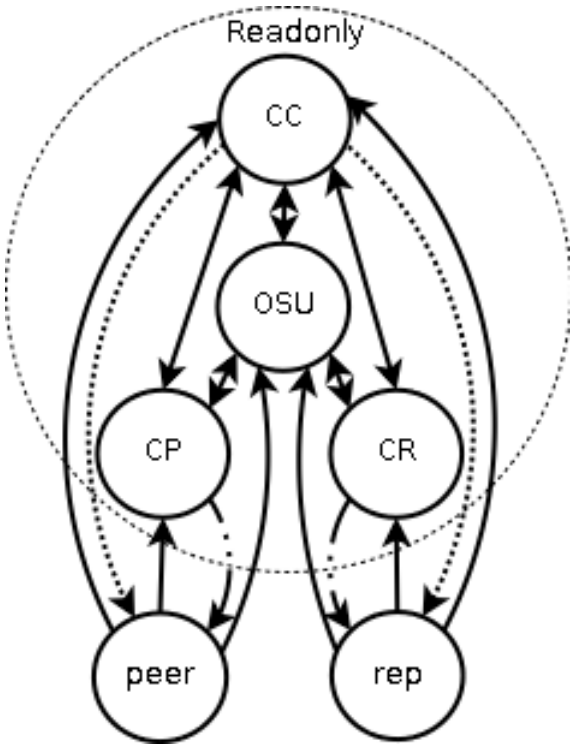

Figure 3.10: Universe type hierarchy with support for all casts.

*	peer	rep	Cpeer	CRep	CC	OSU
peer	peer	OSU	CPeer	CC	CC	CRep
rep	rep	OSU	CRep	CC	CC	OSU
CPeer	CPeer	CC	CPeer	CC	CC	CRep
CRep	CRep	CC	CRep	CC	CC	CC
CC	CC	CC	CC	CC	CC	CC
OSU	OSU	OSU	CC	CC	CC	OSU

This combinator leads to many *CC* states, where we have no information about, where the reference points to. And the *OSU* state can only be created via a combination. It must be a subtype of the *CC* state since we have more information than in the *CC* state. Therefore it is not possible to assign something else to the *OSU* type.

But since we say that we cast a conflict-type only to the corresponding type we can say that $CPeer * CRep = OSU$. This is the case because $peer * rep = OSU$ and we say, that we cannot cast the conflict types to something else. So we know that the type is not castable to a writable type.

Furthur more we want to be able to use the *OSU* type to express all types we cannot cast and want not to cast. For this we need to be able to assign all other types to the *OSU* state. With an additional state we could also handle this wish. The additional state will not bring other advantages.

So we state now, that we don't want to cast the *OSU* type references to **another Universe or sub Universe**. And also that the *OSU* state must be used for references we know that they refer to another sub Universe.

Usable extended conflicts type interpretation Figure 3.10 shows how the types of the extended conflicts internal type system may be casted. The dashed arrows show the allowed casts. As an addition to the conflict type system we now also allow to cast the *CC* type to either *peer* or *rep*. This casts are drawn with the small dashed arrows. We want to distinguish between these two kinds of casts. The small dashed casts are much more error prone than the other ones and should be used more rarely than the casts known from the conflict type system.

The assignment relation among the different types is shown with the solid arrows. By this internal type system we still lose more information with assignments. Again most of the read-only types are assignment compatible among themself. Only the *CP* and *CR* types are not compatible read-only types. This incompatibility should help avoiding the insertion of invalid casts.

The following type combinator function shows how the *OSU* type is combined with the other types. With the new interpretation we only combine to *CC* when already one *CC* occurs in the combination. So we ensure that we do not introduce much more casts than before.

Type combinator:

*	peer	rep	Cpeer	CRep	CC	OSU
peer	peer	OSU	CPeer	OSU	CC	CRep
rep	rep	OSU	CRep	OSU	CC	OSU
CPeer	CPeer	OSU	CPeer	OSU	CC	CRep
CRep	CRep	OSU	CRep	OSU	CC	OSU
CC	CC	CC	CC	CC	CC	CC
OSU	OSU	OSU	OSU	OSU	CC	OSU

Encoding This internal representation has six states. Therefore we can encode it with 3 bits, 3 + 2 bits and 6 bits. The six bit encoding seems to be quite large and perhaps one can see a difference in the run time. For the casts we want to always use two bits. We want to distinguish between the two different types of casts we can introduce. The *CC* cast must be penalized more than the normal conflict cast.

We now show the different type encodings. And then we show for each constraint in a higher level logic how to ensure their properties. The translation to the clauses is done the same way as before and not shown here. At the end we show some differences in expressing the solution description preferences.

The compact encoding with three bits and the 3 + 2 bit encoding. The first three bits represent the compact three bit encoding. The two last bits are used for the 3 + 2 bit encoding. This type system has four read-only states and therefore we used all available read-only states of the 3 + 2 bit encoding.

Type encoding with three bits and the 2 additional weighting bits :

Type	tBit	roBit	cBit	wpBit	wrBit
peer	false	false	false	true	false
rep	true	false	false	false	true
CPeer	false	true	true	false	false
CRep	true	true	true	false	false
CC	true	true	false	false	false
OSU	false	true	false	false	false

The encoding with six bits is straight forward but large. We use for each type the corresponding bit. With this encoding we can express the preferences correctly.

Type encoding with six bits:

Type	pBit	rBit	cpBit	crBit	ncBit	
peer	true	false	false	false	false	false
rep	false	true	false	false	false	false
CPeer	false	false	true	false	false	false
CRep	false	false	false	true	false	false
CC	false	false	false	false	true	false
OSU	false	false	false	false	false	true

The Encoding of the Five Constraints

- Definition $def(T1)$
  We forbid invalid bit combinations for each encoding. Additionally we use the cast policy for this type system. If *allowBothCasts* is set to false we forbid the CC state.

  Disallow forbidden types:

  Not *peer*: $\neg T1[peer]$
  Not *rep*: $\neg T1[rep] \wedge \neg T1[CRep]$
  Not *readonly*: $\neg T1[CC]\neg T1[OSU] \wedge \neg T1[CPeer] \wedge \neg T1[CRep]$

  This high level logic formulas are transformed and optimized for each implementation if necessary.

- Subtype $T1 <: T2$
  The subtype relation is again a little bit less restricted.

  $T2[peer] \rightarrow T1[peer] \quad \wedge$
  $T2[rep] \rightarrow T1[rep] \quad \wedge$
  $T2[CPeer] \rightarrow \neg T1[CRep] \quad \wedge$
  $T2[CPeer] \rightarrow \neg T1[rep] \quad \wedge$
  $T2[CRep] \rightarrow \neg T1[CPeer] \quad \wedge$
  $T2[CRep] \rightarrow \neg T1[peer]$

- Convertable $T1 <:> T2$
  We use the same convertable rule as always.

  $\neg(T1 <: T2) \quad \vee$
  $\neg(T1 > T2)$

- Cast $(T1)T2$ *is used* C
  It is allowed to cast the conflict-types to the corresponding writable types. And the CC type can be casted to both writable types. The corresponding cast bit must be set. In the second last line we have a disjunction of conjunctions on the right side of the implication. This construct can be implemented with one additional bit.

  $T2[NC] \rightarrow T1[NC] \quad \wedge$
  $T2[peer] \rightarrow \neg T1[rep] \quad \wedge$
  $T2[peer] \rightarrow \neg T1[cRep] \quad \wedge$
  $T2[peer] \rightarrow \neg T1[OSU] \quad \wedge$
  $T2[peer] \rightarrow \neg T1[CC] \quad \wedge$
  $(T2[peer] \wedge \neg C[bit]) \rightarrow \neg T1[cPeer] \quad \wedge$

$$
\begin{array}{l}
T2[rep] \rightarrow \neg T1[peer] \quad \wedge \\
T2[rep] \rightarrow \neg T1[cPeer] \quad \wedge \\
T2[rep] \rightarrow \neg T1[OSU] \quad \wedge \\
T2[rep] \rightarrow \neg T1[CC] \quad \wedge \\
(T2[rep] \wedge \neg C[cBit]) \rightarrow \neg T1[cRep] \quad \wedge \\
\\
T2[cPeer] \rightarrow \neg T1[rep] \quad \wedge \\
T2[cPeer] \rightarrow \neg T1[cRep] \quad \wedge \\
T2[cPeer] \rightarrow \neg T1[OSU] \quad \wedge \\
T2[cPeer] \rightarrow \neg T1[CC] \quad \wedge \\
(T2[cPeer] \wedge \neg C[cBit]) \rightarrow \neg T1[peer] \quad \wedge \\
\\
T2[cRep] \rightarrow \neg T1[peer] \quad \wedge \\
T2[cRep] \rightarrow \neg T1[cPeer] \quad \wedge \\
T2[cRep] \rightarrow \neg T1[OSU] \quad \wedge \\
T2[cRep] \rightarrow \neg T1[CC] \quad \wedge \\
(T2[cRep] \wedge \neg C[cBit]) \rightarrow \neg T1[rep] \quad \wedge \\
\\
T2[CC] \rightarrow \neg T1[CPeer] \quad \wedge \\
T2[CC] \rightarrow \neg T1[cRep] \quad \wedge \\
T2[CC] \rightarrow \neg T1[OSU] \quad \wedge \\
(T2[CC] \wedge \neg C[ccBit]) \rightarrow \neg T1[peer] \quad \wedge \\
(T2[CC] \wedge \neg C[ccBit]) \rightarrow \neg T1[rep]
\end{array}
$$

- Combination $T1 * T2 = T3$
  Here the whole combinator table could be written down as implications. We do not show it here.

Expressing the solution preference For the six bit and the 3 + 2 bit encodings we can just assign the necessary values to the corresponding bits. For the conflict conflict cast bit we can use a fixed factor to multiply the normal cast bit value to get a extra penalization for the conflict conflict casts.

3.6 Putting it All Together: An Example.

Now we have shown all the details of the architecture. To get a better understanding how all the components work together, we consider a small example. In the example we infer the Universe type annotations for a simple class Storage.

Storage.java

```
class Storage {
  Object item;
  void store(Object o){
    this.item = o;
  }
  public Object retrieve(){
    return this.item;
  }
}
```

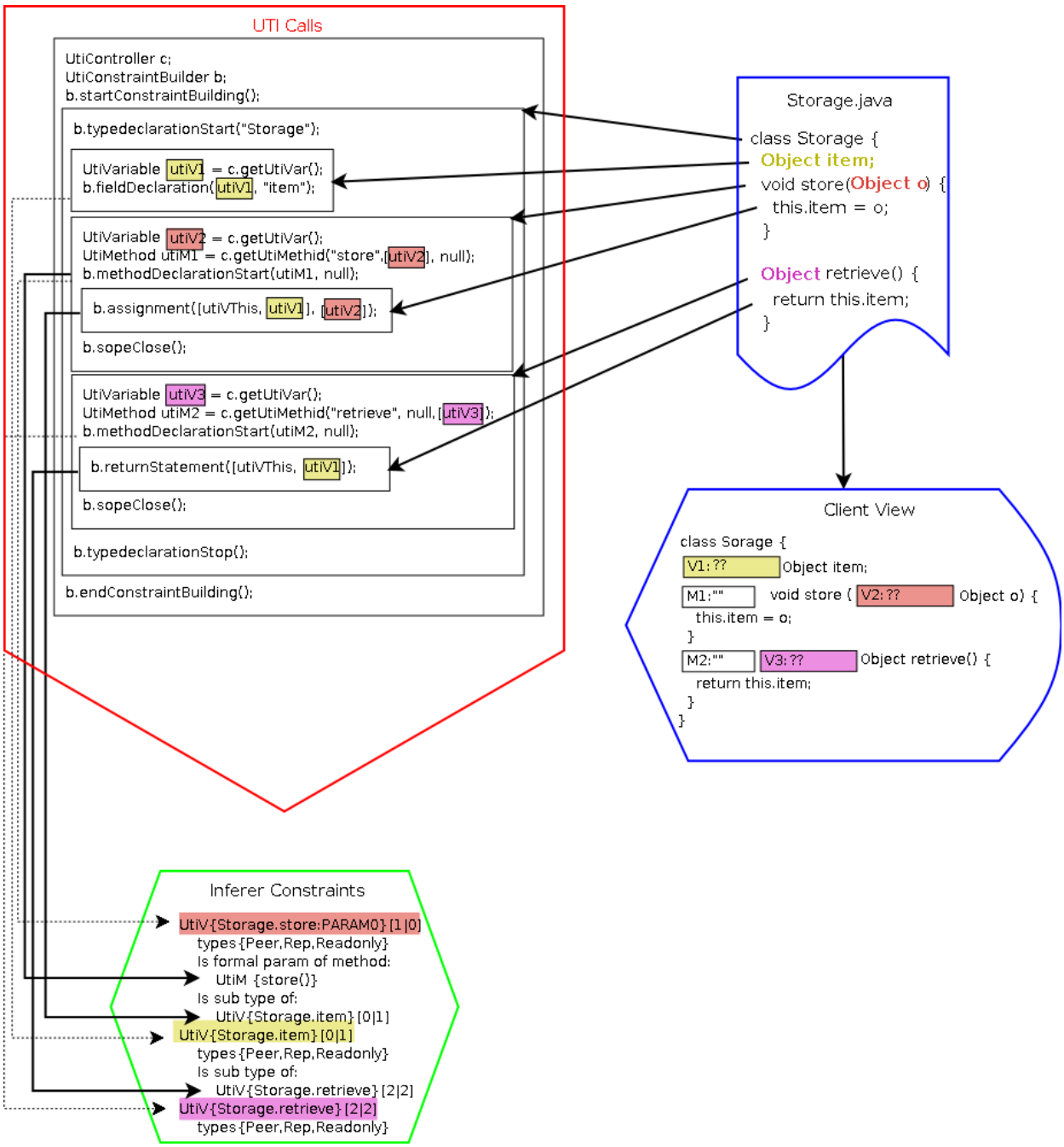

Figure 3.11: Providing the program structure.

First the Java source code is parsed and typechecked with the JML compiler. The built up abstract syntax tree (AST) is then visited by the UniverseJmlVisitor. This visitor then calls the necessary UTI methods to build up the constraints used to infer the Universe types and also builds up any user constructs to display or store the infered solution. In the example we have a *Client View*. On figure 3.11 are four components. On the right the source file is displayed. Below a *Client View* is represented. On the left top the UTI method calls are visualized and below the internal constraints. In this example there are three annotations to infer. For each annotation we use a dedicated color to highlight the relations and dependencies.

First a UtiController and UtiConstraintBuilder must be retrieved. Then the Universe JML visitor can start processing the AST. At first the visitor tells the builder that it wants to start building up constraints. Then the visitor reaches the class declaration of the *Storage* class in the AST. For this declaration it invokes a typedeclarationStart method. With this method a type scope is opened in the builder. Then the visitor processes the fields of this class. It creates a UTI variable and maps it to the *item* field. After the creation the field declaration method is called and the UTI variable is given as argument. Next to the UTI calls it also inserts the UTI variable in the clients view. The builder inserts a constraint into the constraints for the declared field.

After the fields, the methods are processed. For the *store* method the UTI variable for the argument is created and then a UTI method for the method. Then the visitor invokes a methodDeclarationStart method which opens a new scope for the method. The visitor also inserts the UTI method and the representative for the parameter into the view. The builder creates a constraint for the parameter and stores the relation between the UTI method and the UTI variable. Then the method body can be processed. The assignment triggers an assignment method of the builder. Two lists are provided as parameters representing the field access on the left hand side and the expression on the right side. The builder generates a subtype constraint for the parameter out of this information. The parameter *o* must be a subtype of the *item* field. Then at the end of the method the visitor invokes a scopeClose method to inform the builder that the method declaration is processed.

The rest of the class declaration is processed the same way. At the end of the class declaration the class scope is also closed. When no other classes must be visited then the constraint building process is finished with calling the endConstraintBuilding method. The builder then finishes building up the constraints and provides the constraints to the controller to solve them. The visitor also finishes inserting the method and type representants in the view. Since we use a constraint builder which does not insert any casts, there are no casts to insert in the view.

The next step is solving the constraints which is shown on figure 3.12. The solve method must be invoked on the UTI controller. Then the configured internal type system and the corresponding boolean representation is used to generate the boolean encoding used by the pseudo boolean solver. In the example the static Universe type system with the compact encoding is used. So each type is encoded with two bits. In this encoding the definition constraint is not used since two bit combinations are valid for the read-only type. For the two subtype constraints the corresponding clauses are written into the *constraints.cnf* file. Since the method *store* is not pure we do not generate anything for the formal parameter relation. The objective function is specified in the *constraints.cnf.pb* file. Only two weights are specified for the *item* field.

After the generation of the two files the pseudo boolean solver is used to solve the specified problem. The found optimal solution is then returned to the controller. With the help of the boolean encoding the UTI variable now knows the infered type annotations and can deliver it to the view. The displayed solution is not satisfying. The *item* field and the parameter are annotated with *rep*. Since this class should be a widely used storage container we prefer a read-only annotation. This can be achieved with fixing the parameter to *readonly*. The corresponding UTI method call is shown in the figure. The method *retrieve* is a pure method. This purity information is also added to the constraints.

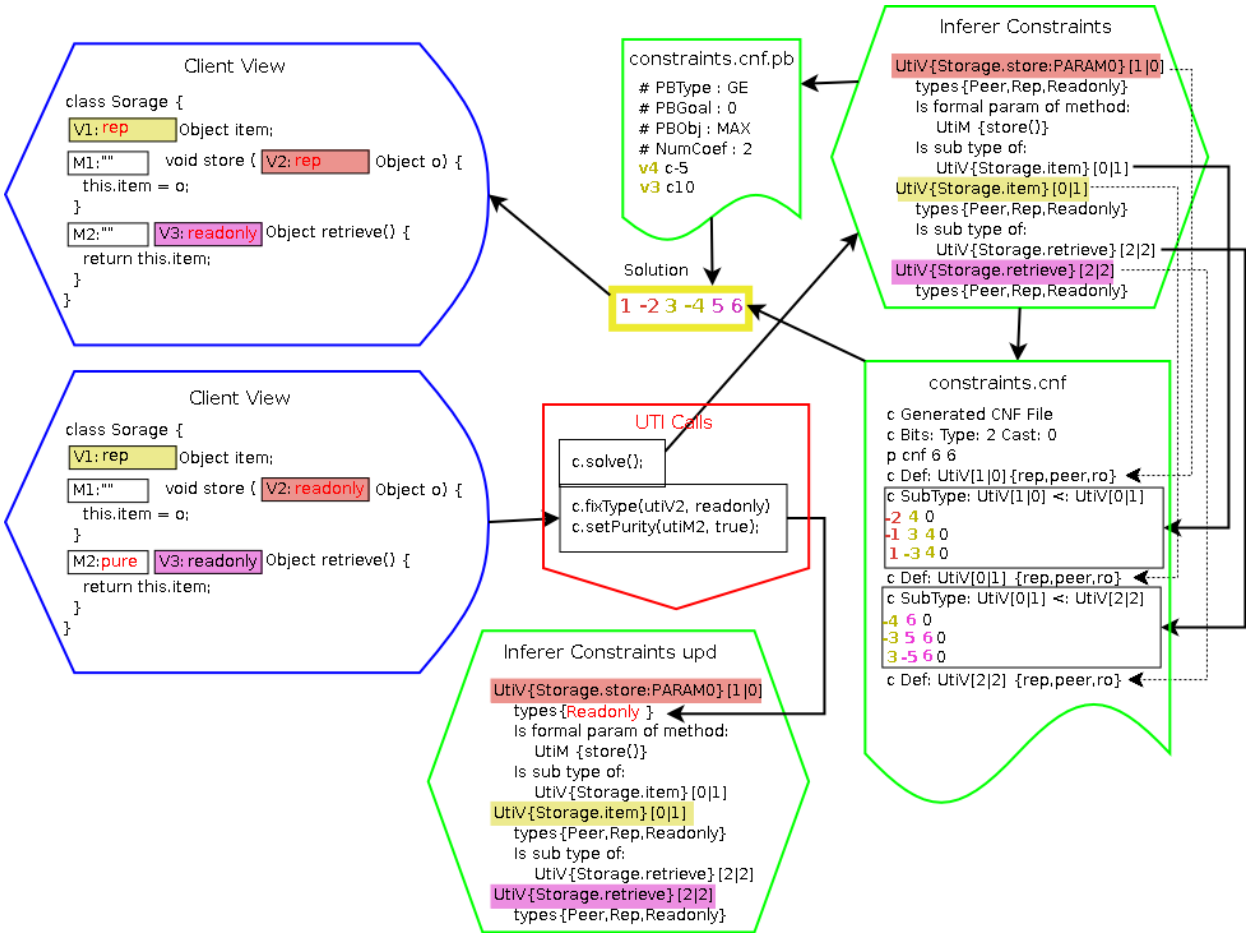

Figure 3.12: A UTI method call.

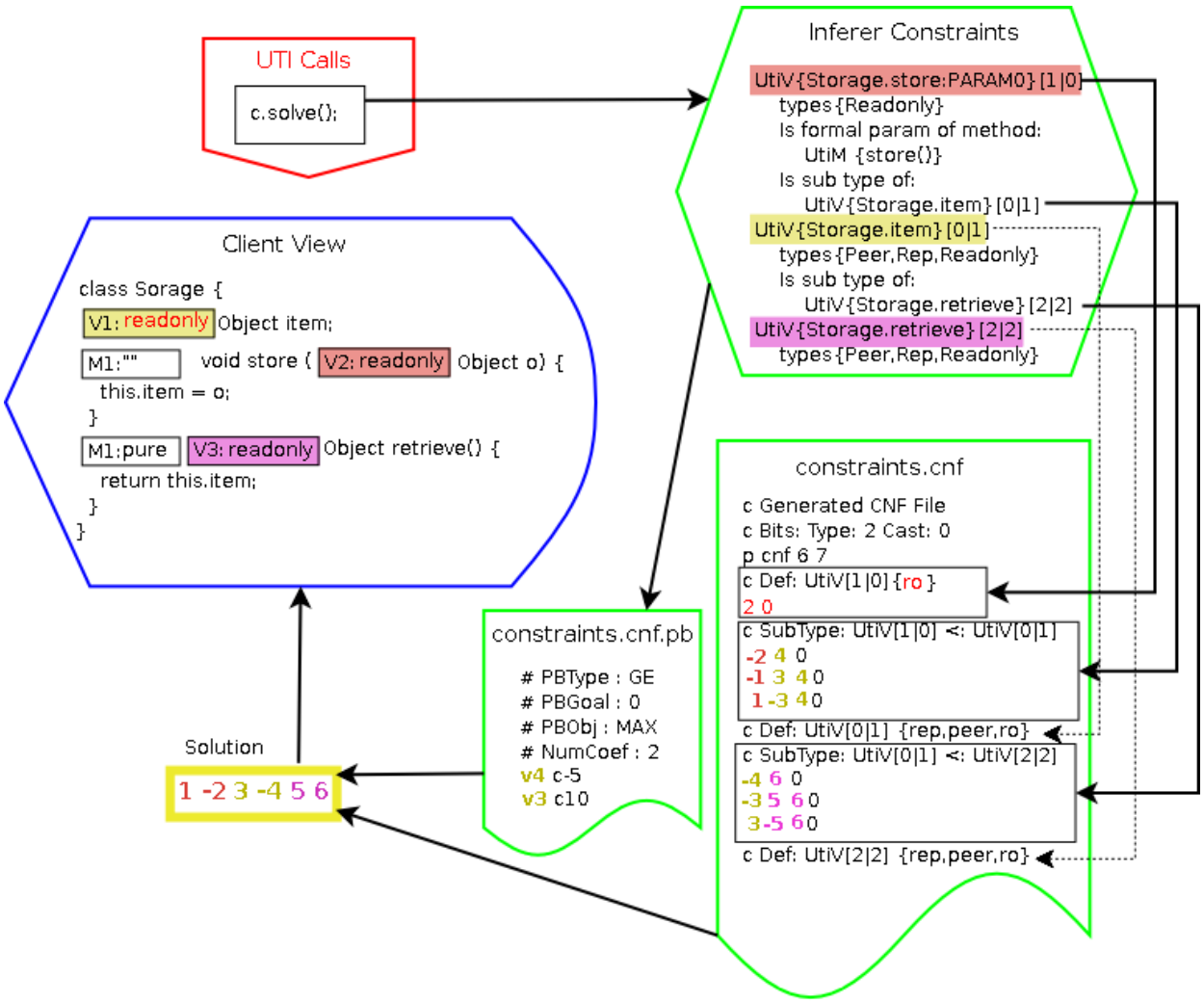

Figure 3.13: The new solution.

Providing these informations changes the internal constraints. The view now shows an inconsistent type annotation. To fix the inconsistent annotation the modified constraint system must be solved again.

The figure 3.13 shows the new solution. The internal constraints are again translated to the required files. The *constraints.cnf* file now consists of an additional definition clause. This red clause represents the fixing of the parameters annotation to *readonly*. The *constraints.cnf.pb* file did not change. When the files are created the pseudo boolean solver is used to find an optimal solution. The only allowed solution is returned and all annotations are set to *readonly*.

Chapter 4

Implementation

The components and the architecture of the SUTI2 project were introduced before. Now we want show some implementation details. In section 4.1 we give an overview over all packages and their classes. Then in 4.2 we show how the tool can be configured. How to use the tool is described in 4.3. And at the end of this chapter in 4.4 we show the tools used for the implementation.

4.1 The Four Packages of SUTI-Tool 2

For our project we have four packages. The main package containing two other packages and the JML visitor package. First we discuss the main package 4.1.1 then the JML visitor package 4.1.2. The two packages contained by the main packages are the UTI package 4.1.3 and the PBS implementation package 4.1.4.

4.1.1 Main Package

The main package `ch.ethz.inf.sct.static_typinfer2` contains a JML client, which is the type inferring tool developed in this master thesis. Additionally it contains two other packages with the UTI interface and the PBS implementation of the UTI abstraction.

We tried to reuse as much code as possible from the SUTI1 project. Therefore the classes in the main package have some correlation with the ones from the previous project.

- **Main**
  The Main class extends the Multi Java Compiler MJC. It is the starter and coordinator of the tool.
- **Configuration**
  This class reads in the configuration file necessary to configure the tool.
- **JmlToUtiMapper**
  This class maps the JML AST entries to the UTI representatives. It is used by the UniverseToJmlVisitor class.
- **VariableContainer**
  This is a simple container used by the mapper class.

- **UniverseJmlVisitor**
  This visitor generates all necessary UTI method calls by visiting the AST. This class can be compared with the constraints generator in the SUTI1 project.
- **AnnotationWriter**
  This class represents the XML structure for the annotation output but holds the UTI representatives.
- **UserInteraction**
  This class a simple interface for user interaction classes.
- **SimpleUi**
  The SimpleUi class is an implementation of the above interface and provides a UI to interact with the inferring process.
- **Helper**
  The Helper class is used by the Main class and is still very similar to the one in SUTI1.

Main

The Main class basically the same things as in SUTI1. Since the MJC had changed in the mean time we had to addapt the class to the new situation. Some major changes also came from the usage of the UTI interface.

The Main class uses the information from the Configuration class to infer the requested programs. It uses the MJC to compile the program. Then it instantiates the requested UTI Controller and the visitor. The visitor then visits the AST and provides the information to the UTI constraint builder. If a user interaction class is provided in the configuration, then the user interaction is triggered, else the build up constraints are solved and the solution is written to the specified XML file.

Configuration

The configuration class is taken form the SUTI1 project. It is adapted to the new configuration.xsd file, which describes the used configurations. Now we also allow to specify the config file, which needs to be read in. In section 4.2 the possible configurations are described.

JmlToUtiMapper

This mapper is implemented with a bunch of `java.util.HashMaps`. It maps a provided JML element to a UTI representative. This can be a `UtiMethod` or `UtiCast`. If a requested element is not yet in the map, then the mapper creates a new representative and inserts it into the map.

With some JML elements we run into a problem. Their `equals` and `hashCode` methods were overridden, so that we could not distinguish different objects. This problem was solved with an ugly hack. We implemented our own *StupidMap*. This problem should be solved better in a future release.

Variable Container

The variable container relates a UtiVariable with a JML AST element. So this class can be used where the relation between JML and UTI is important.

Universe Jml Visitor

The UniverseJmlVisitor class is comparable with the ConstraintGenerator and the AnnotationWriter class in the SUTI1 project. It extends the `SimpleJmlVisitor`. It visits the AST and produces the necessary method calls on the UTI constraint builder and extracts the information used to write the annotation output.

Because the constraint builder always returns the type of the provided expression, the processing of complex statements is straight forward. They are composed of the returned types from the UTI constraint builder.

In the previous work [3] there was the so called deepest node problem. We did not really encounter the same problem. We only use one flag for the constraint generation part. This flag specifies if we are visiting the target of a static method. This is necessary to process the static target correctly.

We traverse the AST only once. With the support of the UTI constraint builder it is no longer necessary to traverse some parts of the AST twice, as this was the case in the SUTI1project. We make a post order traversion, so all sub expressions are handled before we process an expression.

Overridden Methods The parameters and the result type of overridden methods must be consistent. To achieve this consistency we use the same bits to represent a specific parameter of all methods.

Annotation Writer

This class implements the same structure as the annotation XML files. We use this class to store the information used to generate the annotation output file. This structure allows us to generate the constraints and extract the information needed for the annotations. It is also used by the simple UI as an information source to access to the UTI representats of the types to infer.

After a successful inferring the annotations must be written to the output file. Then the annotation writer uses the generated Annotations.jar to write the XML file.

We model all possible casts with the UTI interface and also in the boolean representation. So we try to keep the number of casts small. Therefore we insert for an access chain only one cast for tho whole chain and not one cast for each reference in the chain. This lead to a change of the XML format to save the annotations. The `annotations.xsd` file contains the used XML scheme. Instead of inserting casts to an arbitrary expression we distinguish between four types of possible cast insertions.

- Assignment
  In the assignment two different casts can be inserted. One for the expression which is assigned and one for to updated target.
- Method Call
  In the method call also the target may be casted. Next to the target all actual parameters may be casted as well.
- Array Initializer
  In the array initializer we only cast the array element.
- Return Statement
  Here the whole expression of the return statement may be casted.

So the addcast entry in the XML annotation output file has now one of these four types, and an index to indicate which occurrence is meant. Further more there is another number used to specify the internal position. For example a ”0” in a method call indicates that the first parameter is casted. A ”-1” must be there if the target of the method call is casted.

User Interaction

A very simple interface to give the main class a common way to handle different user interaction classes.

Simple Ui

This class provides a mechanism to interact with the inferring process. Together with a text editor this gives a powerful way to check intermediate solutions and give some feedback.

With the simple UI one can reconfigure the Universe type inferer without a complete restart of the tool. This is important, because compiling the sources and starting the tool takes up much time.

The simple UI consists of a console and a command part. It accepts simple commands and uses the UTI interface to make the requested changes or display the queried information.

Helper

This class is also taken from the SUTI1 project. Its functionality is reduced to the remaining necessary operations.

4.1.2 JML Visitor

The JML Visitor package `ch.ethz.inf.sct.jmlvisitor` contains a simple implementation of the `JmlVisitor` interface. The SimpleJmlVisitor class walks through all elements of the AST.

It also has the *optgen* files to define some new options. Here we define the option to specify which config file to use.

The following classes and files are in this package:

- **SimpleJmlVisitor**
  The simple implementation of the `JmlVisitor` interface.
- **VisitorCommonOptions**
  This class is generated by VisitorCommonOptions.opt.
- **VisitorMessages**
  This class is generated by VisitorMessages.msg.
- **VisitorOptions**
  This class is generated by VisitorOptions.opt, where we added the additional argument.
- **VisitorVersionOptions**
  This class is generated by VisitorVersionOptions.opt.

4.1.3 UTI Package

The UTI interface was discussed already before in 3.3. So now we only show what is contained in the `ch.ethz.inf.sct.static_typeinfer2.uti` package.

- **UtiController**
  The Controller of a Universe type inferer.
- **UtiVariable**
  The representative of a type to infer.
- **UtiMethod**
  The representative of a method. Used to express families of overloaded methods and set the purities.
- **UtiCast**
  The representative of a cast. Casts are inserted by the constraint builder.
- **UtiConstraintBuilder**
  The constraint builder consumes the program properties and builds the constraints which must be satisfied by a valid Universe annotation.
- **UtiCBReturnValue**
  This class is used as the return type by multiple methods of the constraint builder. It contains the inserted casts and the expression type.
- **UtiSolutionDescription**
  Is an abstract description of the Universe type properties.
- **UtiSingleVar**
  This class provides a simple implementation of the *readonly*, *this*, *null* and *primitive* UTI variable.
- **NameInfo**
  Interface which must be implemented by some objects provided to the constraint builder.
- **MemberInfo**
  Interface which must be implemented by some objects provided to the constraint builder.
- **MethodInfo**
  Interface which must be implemented by some objects provided to the constraint builder.

4.1.4 PBS UTI Implementation

The `ch.ethz.inf.sct.static_typinfer2.pbs_uti` contains an implementation of the UTI interface using a pseudo boolean solver as constraint solver and optimizer. The code in this package is responsible for finding a good Universe annotation.

The classes in this package can be categorized in three categories. The interface classes implementing a UTI interface, the boolean representation classes implementing a boolean representation and some other internal used classes.

- UTI interface implementations
  The following classes implement the UTI interface.

  - **PbsController**
    This class implements the UtiController interface. Further more it implements the UtiVariable, UtiMethod and UtiCast interfaces as inner classes. The controller is the center of the PBS UTI implementation. All other objects without the constraint builder in the PBS implementation are in a sub universe of the controller.
  - **PbsConstraintBuilder**
    This is a simple interface extending the UTI constraint builder interface. All constraint builders must implement this interface.
  - **PbsCBNoCasts**
    This is a constraint builder implementation which does not insert any casts. This builder is intended to be used in combination with a static Universe type boolean representation.
  - **PbsCBWithCasts**
    This is a constraint builder implementation which inserts casts.

- Other internal classes
  The following classes implement all non UTI interface behavior.

  - **PbsConstraints**
    This class implements the internal constraints of the PBS backend.
  - **PbsConfiguration**
    This class is implemented with the singelton pattern. It stores the configuration state of the inferer.
  - **PbsBooleanWriter**
    An interface for the boolean writers. The internal type system uses this interface to write the boolean formulas to the required format.
  - **PbsCnfPbWriter**
    The CnfPb writer implements the boolean writer interface and writes the boolean formulas to the cnf.pb format. The clauses are written to the *.cnf file and the objective function is written to the *cnf.pb file.
  - **PbsOpbWriter**
    The Opb writer is another boolean writer. It writes the provided boolean formulas in the opb format. The constraints and the objective functions are written to the *opb file.
  - **PbsBooleanRepresentation**
    This is an interface which must be implemented by each boolean representation. The interface consumes the basic constraints.
  - **PbsBrAbstractImpl**
    An abstract implementation of a boolean representation. This class implements the common behavior of all boolean implementations. It serves as base class for all provided boolean representations.

- Boolean representations
  The boolean representation classes implement four different semantics. For all semantics we have multiple boolean encodings. Before inferring a Universe annotation, one of the the semantics must be chosen. The different boolean representations are explained in section 3.5.5.

  - Static Type System
    All these classes implement the static boolean type system.
    * **PbsBrAbstractStatic**
      This is the abstract implementation of the behavior of the static type system.

    - **PbsBrStatic3**
      This class implements an encoding with three bits for the class above.
    - **PbsBrStaticUniverse**
      Another implementation of the static Universe type system. Uses only two bits.

- Safe Casts
  These classes implement the safe cast type system. Only save casts are introduced.
    - **PbsBrAbstractSafeCast**
      The abstract implementation of the safe cast semantics.
    - **PbsBrSafeCast5**
      An encoding with five bits for the safe cast semantics.
    - **PbsBrSafeCast3**
      An encoding with three bits for the safe cast semantics.
    - **PbsBrSafeCast3H2**
      This is a hybrid encoding for the safe cast semantics using 3 + 2 bits.

- Conflict System
  The classes here have the same semantics as the conflict internal type system from the SUTI1 project.
    - **PbsBrAbstractConflict**
      The abstract implementation of the behavior of this internal type system.
    - **PbsBrConflict5**
      An encoding with five bits for the conflict type system.
    - **PbsBrConflict3**
      The compact three bit encoding for this internal type system.
    - **PbsBrConflict3H2**
      An implementation of the conflict internal type system using the hybrid encoding.
    - **PbsBrPrologSemantic**
      This class is another implementation of the conflict semantics which uses three bits.

- Extended Conflict System
  The extended conflict type system consists of six internal states.
    - **PbsBrAbstractExtConflict**
      The abstract implementation of the type system behavior.
    - **PbsBrExtConflict6**
      A six bit encoding for this type system.
    - **PbsBrExtConflict3**
      A three bit encoding.
    - **PbsBrExtConflict3H2**
      The hybrid 3 + 2 bit encoding for the internal extended type system.

Controller

The controller is the main part of the inferer. The cast, method and type annotation representatives are implemented as inner classes of the controller. It coordinates the inferers components. It also provides some interface to get writable access. All writable access to the inferer must go through the controller. Only in the constraint building phase the constraint generator can also be accessed for writing operations.

Solving the Constraints When the solve method is triggered the controller checks the configuration for components it must use for the inferring process. It uses the boolean representation from the configuration together with the necessary boolean writer to translate the internal constraints into the needed files. Then the controller uses the specified PB Solver to solve the constraints. After solving the solution is parsed and an array with the booelan values is provided to the boolean representation.

If the type of an annotation is queried then the boolean representation is asked for that type. The boolean representation is the only component in the inferer which knows how to translate the bits to Universe types.

Constraint Building During the constraint building phase the controller serves as a proxy for the constraint builder. This is necessary because the internal constraints are owned by the controller and the constraint builder must be in the same universe as the controller since the clients must have writable access to both components.

Constraint Builder

Both implementations are builders. They consume the program structure and build the necessary constraints. The constraint builder with casts tries to insert casts only where they may be used.

Handling of Non Variable UtiVariables In the basic constraints we only want to deal with variable annotations. Therefore the constraint builder filters all not variable UTI variables. The effect of such non variable items is then translated into fixed or prevented types.

For example if a `String` is combined with an array, then the array must always be typed read-only, since the array was created in the string's context. Therefore the combination leads to fixed read-only types. The `split` method of the type `String` is such a case.

The Context To keep track of the context, the constraint builders use a Context class. The context knows if we are in a static scope. It is also used to set access for the current method information.

Constraints

The Constraints class builds up the internal constraints with the help of the constraint builder. The constraint builder provides the relations between different type annotations and the constraints class stores them internally.

To store the relations between different type annotations the Constraint class inserts a VarConstraint for each variable. The VarConstraint object represents the constrains for exactly one variable.

Generating the Boolean Formula To generate the boolean formula from the constraints the Constraint class provides a method `generateBoolean`. This method takes as argument a boolean representation. Then all variable constraints provide their constraints to the boolean representation, which transforms the constraints into the desired form.

Minimum Number of Variables for the Boolean Representation Before the constraints can be provided to a boolean representation, an index must be assigned to each variable. This index is then used by the boolean representation to bind the variable to the bits. So if the same index is assigned to two variables, then the boolean representation uses the same bits to model them. The algorithm used to assign the bits tries to assign the same index to all variables which must be equal. With this mechanism we try to keep the amount of variables for the boolean representation as small as possible.

Boolean Representation

All the boolean representations use the same abstract base class to share the common behavior. The figure 4.1 shows an inheritance diagram of the different boolean encodings. There are basically three different ways to implement the boolean representation.

- Boolean encoding strongly interwoven with the constraints to express.
  The Prolog semantics and the static Universe implementations are implemented according to this paradigm. This results in a very short encoding. But the overhead to implement is bigger.
- Express constraints abstractly and design an encoding for the used states.
  All other boolean representations use this paradigm. So the encoding for the used states can be chosen to fit the requirements. But it implies some programming overhead in contrast to the third paradigm.
- Express constraints and use a general boolean encoding for n states.
  With such an encoding only one encoding for n states would be necessary. Each internal representation with n states could then use the same encoding.

Weighting The abstract boolean representation base class has a method determining the weight for each type specific to a variable. So the different encodings must only assign the weight to the right bits. In the hybrid type systems the weighting bit for a type is only inserted when the specific type is weighted for this annotation.

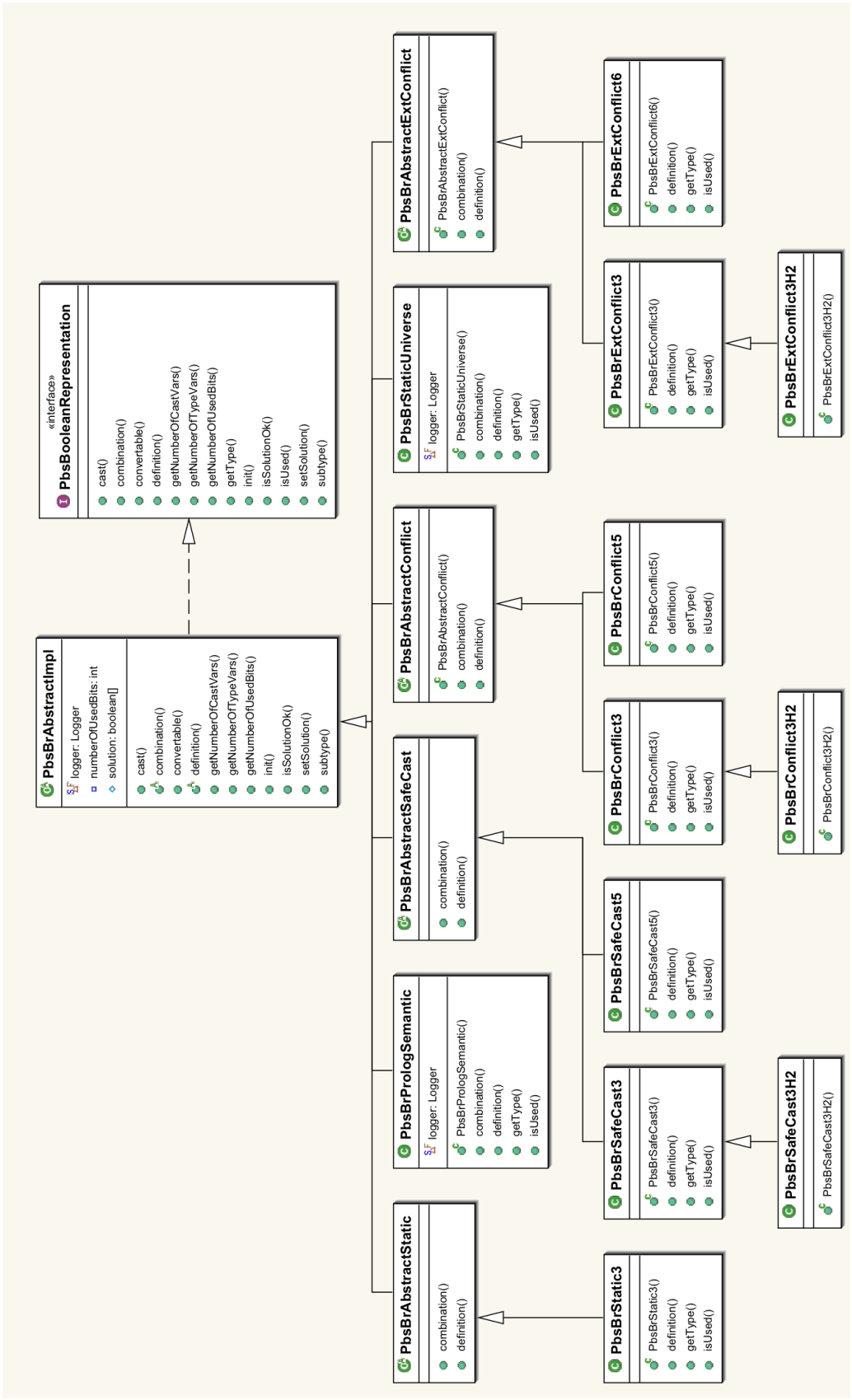

Figure 4.1: An inheritance diagram of the boolean representation.

4.2 Configuration

An UTI inferer can be configured in two different ways. First the inferer needs an initial configuration when it is created. And then after the initialization the inferer can be configured again. The SUTI tool basically only accepts a static configuration via the config file. The SUTI tool needs the following configuration to work.

- Java input
  The files which should be annotated. With fully qualified names.
- annotation input
  The annotation input is not used at the moment.
- annotation output
  The XML file where the inferred annotations should be written to.
- user interaction
  A class which allows some user interaction. This class is used after the constraint building phase.
- annotation tool
  The annotation tool is used to annotate source files with Universe annotations.
- UTI inferer
  This must be the name of a class implementing the UTI Controller interface. This class is used as the inferer. Here also some options for the inferer may be provided. This is depending of the used inferer.

By default the SUTI tool searches in the working directory for a `config.xml` file to use. The user may use another configuration file but this must be specified by a command line option at the start. If no file is found the SUTI tool terminates with an error.

The UTI interface also allows an interaction and configuration after the initialization. This feature cannot be used with the basic configuration of the SUTI tool. A specified user interaction class may be used to reconfigure the UTI configuration. The simple UI is made for that purpose.

Since the SUTI tool can basically work with different UTI implementation, we cannot give any implementation specific support in the config file. Therefore only key value pairs are allowed in the inferer section of the config file. These key value pairs must be set accordingly the used UTI class.

4.2.1 PBS UTI Configuration

The PBS implementation of the UTI interface has some required and some optional accepted configuration keys. At the initialization a value for the following keys must be provided.

- `pbs_constraint_builder`
  The constraint builder is instantiated at the beginning. A class implementing the PbsConstraintBuilder interface.
- `pbs_boolean_rep`
  The tool always requires a boolean representation. A class implementing the PbsBooleanRepresentation interface.

- `pbs_tool`
  Here a PB-Solver must be provided. The provided string must define an executable program. The provided solver must understand the specified format.
- `pbs_args`
  The arguments for the solver. This option is not really necessary. If not specified no arguments are used. At the end of these arguments the file containing the constraints is appended.
- `pbs_format`
  The value for this key must be `cnf.pb` or `opb`. These are the two supported file formats. As default the first format is used.

Next to these necessary options there are some other options which can be provided to influence the inferers behavior.

- `pbs_constraint_file`
  This string must be a valid file address. The solver writes the boolean encoded constraints to this file. The file is then provided to the solver as input. This file must not have any suffix. The suffix is automatically chosen according to the configuration format. As default `constraints` is used.
- `pbs_internal_constraint_file`
  Here a file name can be provided to get the internal constraints. This is only used for debug purposes. If no file is provided, then nothing is done.
- `pbs_insert_no_casts`
  This is a boolean flag. It understands the values `true` and `false`. This flag is intended to be used by the constraint generators. But is is not used so far.
- `pbs_simple_cast_scope`
  This is also a boolean flag. If this flag is set to true, then it is forbidden to cast a variable in one scope to *peer* and *rep*. How good this flag works depends on the constraint generator and the boolean implementation.

Besides the configuration the UTI interface allows a lot of interaction with the inferring process. The simple UI provides only a limited set of the possible capabilities.

4.3 Usage

The tool must first be compiled and configured. Then the type inferring tool can be run. With the argument `--suticonfig, -Z <config xml file>` the config file which is used to configure the solver can be set. All other arguments are ignored.

First we show how the tool can be compiled and run. Then we point out the capabilities of the simple UI.

4.3.1 Compile and Run

Before you can compile the tool you have to install some dependant projects. When all the needed projects are installed then the `compileAndRun.sh` script must be configured. And also an XML configuration file must be created. Then you can call the script to compile the tool and run it.

The additional components you need to install:

- Retroweaver
  This tool is used by the compilation and some libraries are used to run the type inferer. The location of this tool must be set in the compile script. Also the name of the required JARs.
- JML tools
  The JML compiler is used to compile and run the tool. The root directory of the JML and MJ installation must be set in the script.
- PB Solver
  A supported PB Solver must be installed. The PB solver is only used during the tools execution. The command to start the tool must be inserted in the config file.

All other necessary libraries are provided with the source code.

To only run the tool you can make a symlink to `run.sh` to the `compileAndRun.sh` script. Then calling `./run.sh` does only start the tool without compile.

4.3.2 Simple UI

If the tool is configured to use the simple UI as user interface, then after the code is compiled and the constraints are built you get a command line interface with the following prompt.
`:>`

In the simple UI the following commands can be used:

```
Simple UI Help
---------------
Short introduction to available commands.
ls PATH
           Prints the items in the specified item.
           Eg: 'ls .' or 'ls /Tree.java'
cd PATH
           Changes to the specified item.
           Eg: 'cd ..' or 'cd /Tree.java'
pwd
           Prints the current item.
show PATH
           Prints the properties of the specified item.
           Eg: 'show .' or 'show list:2'
fix PATH MODIFIER [MODIFIER]
           Fixes the Universe type of the specified item,
             second optional mod for arrays.
           Eg: 'fix ../data rep' or 'fix . readonly'
unfixType PATH MODIFIER [MODIFIER]
           Unfixes the Universe type.
           Eg: 'unfix ../data' or 'unfix .'
prevent PATH MODIFIER [MODIFIER]
           Prevents the specified item to become modifier.
           Eg: 'prevent .'
setAllow PATH BOOL
           Allows or disallows the specified cast.
           Eg: 'setAllow ../data false' or 'setAllow . true'
```

```
infer
           Infers a new solution.
w2xml
           Writes the current solution to xml.
iw
           Infers a new solution, writes it to xml and annotates the sources.
setPurity PATH BOOL
           Set the purity of the specified method.
           Eg: 'setPurity . true' or 'setPurity search:10 false'
getSD
           get the solution description.
reconfig [option = value]+
           Reconfigures the UTI inferer.
           Eg: 'reconfigure pbs_boolean_rep = PbsBrStaticUniverse'
hotconfig [option = value]+
           Set hot options for the UTI backend.
           Eg: 'hotconfig pbs_flags = "-D 1 -I 1 -a -f -t 10000 "'
quit
           Terminates the program.
help
           Prints this help.
```

With these commands you can change intermediate results and ask to infer a new result after providing some new constraints with fixing some annotations to a specified type.

Together with a text editor the simple UI becomes a quite powerful GUI, where the annotated solution can be comfortable visualized (see Fig.4.2).

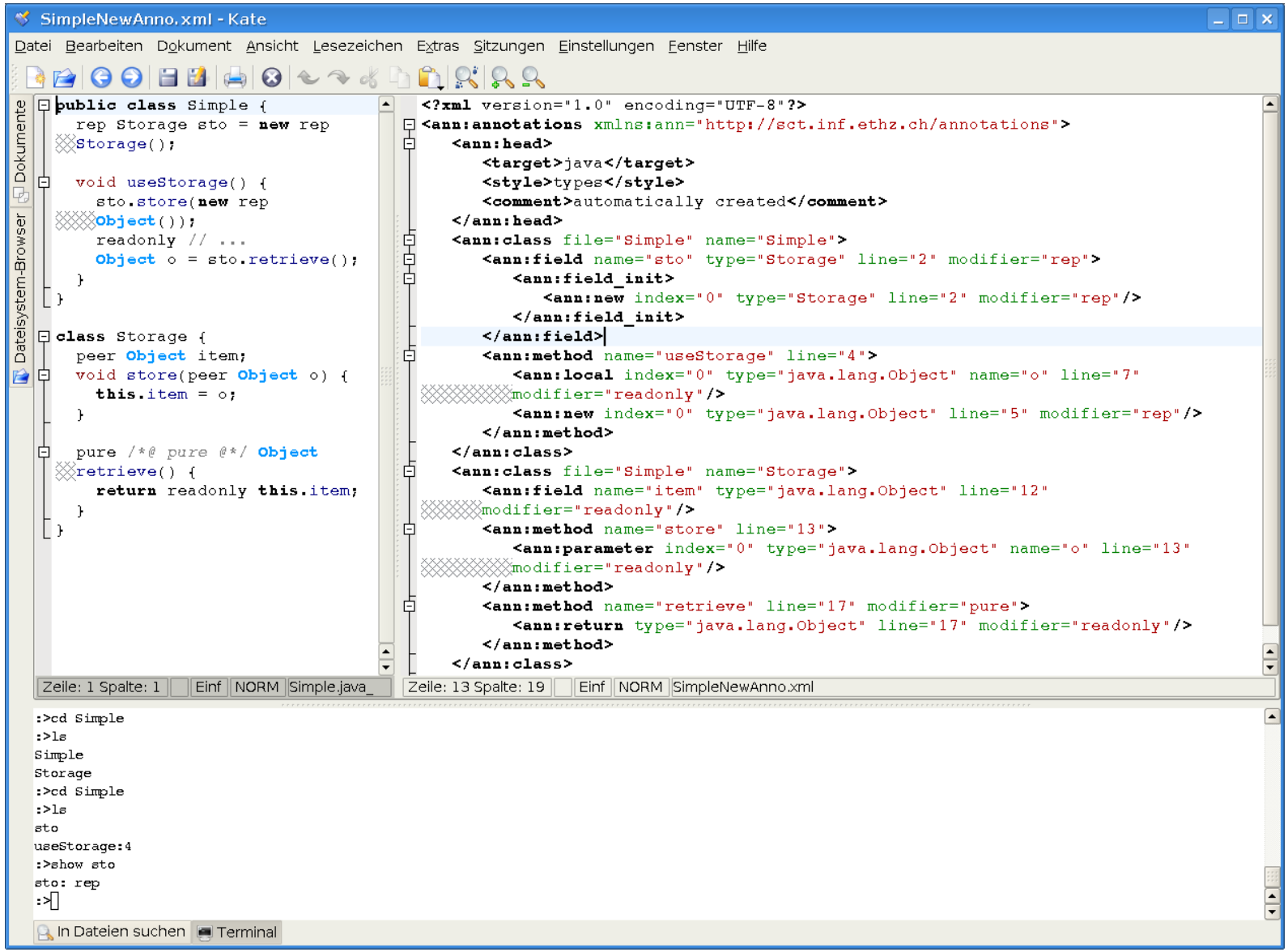

Figure 4.2: A GUI composed of a text editor and the simple UI.

4.4 Used Tools

In this section we consider some non standard tools used to generate and transform our code. Since this project is based on the SUTI1 project, we used the same tools as were used by the previous project. So we don't introduce these tools again. We only mention some experiences we made with the tools. Additionally we use another tool from the MJC package.

4.4.1 Reading and Writing XML

We use the `org.apache.xmlbeans` package to handle the XML files. This package works quite well. But be careful when using the libraries from the CVS. The *scomp* tool used to compile the XML schema into Java classes must be compatible with these libraries. If you use a newer compiler the code may not work.

Therefore we also check in a generated `Annotations.jar` which is compatible with the provided library.

4.4.2 Retroweaver

The retroweaver tool is used to convert the Java 1.5 byte code into Java 1.4 compatible byte code. This is necessary since we implemented our Java classes with Java 1.5 and the MJC depends on Java 1.4.

The transformation of the code using the retroweaver tool works fine. But it seams that not all new Java 1.5 API methods are supported.

4.4.3 Optgen

The optgen tool is a tool from the MJC project. This tool is used to generate the classes responiable for the command line argument handling. The optgen tool takes *.opt files as input. In the *.opt files the command line options can be described.
Usage:
`java -cp /MJ_Path/:/lib_Path/antlr-2.7.5.jar org.multijava.util.optgen.Main myOptions.opt`
Where MJ_Path is the path to the MJ root directory and lib_Path the path to your libraries.

Chapter 5

Results and Conclusions

In this chapter we show two annotated examples and consider them a little bit closer. Then we state our conclusions. And at the end we will outline the future work.

5.1 Results

The goal of this master thesis was to develop a static Universe type inferer using a SAT backend. The SAT backend was realized with a pseudo boolean solver. This is an optimization tool. With this tool we hoped to be able to use the optimization capability to get good annotations.

We implemented some different internal type representations. Two internal representations were already used by the SUTI1 tool. With the new more relaxed internal representation we want to provide better support for already annotated programs.

The first annotated program implements a linked list and an iterator. The second program implements a simple container and two clients using it.

5.1.1 Linked List with Iterator

This example was already used in SUTI1. The program consist of a class `Node`, `Iter`, and `LinkedList`. We annotated the program with the hybrid conflict boolean representation and used the current standard solution description.

In the standard solution description the *rep* fields and *rep* object creation were prefered the most. The parameters are all likely to be *readonly* when they are not constrained to a writable type. The cast acceptance was set to 90. This seams to be a quite high cast acceptance.

A textual representation of the solution description:
Cast acceptance: 90
FIELD PUBLIC INSTANCE WEIGHT: 10
FIELD PUBLIC INSTANCE REP: 10
FIELD PRIVATE INSTANCE WEIGHT: 10
FIELD PRIVATE INSTANCE REP: 10
FIELD PROTECTED INSTANCE WEIGHT: 10
FIELD PROTECTED INSTANCE REP: 10
FIELD STANDARD INSTANCE WEIGHT: 10

FIELD STANDARD INSTANCE REP: 10
PARAMETER PUBLIC INSTANCE WEIGHT: 10
PARAMETER PUBLIC INSTANCE READONLY: 1
PARAMETER PRIVATE INSTANCE WEIGHT: 10
PARAMETER PRIVATE INSTANCE READONLY: 1
PARAMETER PROTECTED INSTANCE WEIGHT: 10
PARAMETER PROTECTED INSTANCE READONLY: 1
PARAMETER STANDARD INSTANCE WEIGHT: 10
PARAMETER STANDARD INSTANCE READONLY: 1
CREATION PUBLIC INSTANCE WEIGHT: 10
CREATION PUBLIC INSTANCE REP: 10
CREATION PRIVATE INSTANCE WEIGHT: 10
CREATION PRIVATE INSTANCE REP: 10
CREATION PROTECTED INSTANCE WEIGHT: 10
CREATION PROTECTED INSTANCE REP: 10
CREATION STANDARD INSTANCE WEIGHT: 10
CREATION STANDARD INSTANCE REP: 10

Depending on the cast acceptance the linked nodes became *rep* or *peer* to the linked list. This is the behavior we expected. The annotated program below was inferred with a high cast acceptance to get the nodes into the representation of the list. The required cast in the `set` method was inserted by the target of the assignment.

As in the found solution by the Prolog inferer we also annotate the `prev` field in the `Node` class as *readonly*. This is done like that, because we didn't specify a preference between *readonly* and *peer* for fields and the `prev` field is nowhere constrained. So the solver has chosen the *readonly* Universe type modifier.

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


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

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

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

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

```
public class LinkedList {
        rep Node first;

        void set(readonly Node np, readonly Object e) {
                readonly Node n1 = np;
                ((rep)n1).elem = e;
        }

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

  public pure /*@ pure @*/ boolean equals( readonly /*@ nullable @*/Object l) {
                if  (!( l instanceof LinkedList))
                return false;
                rep 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;
        }
}
```

Also remarkable is the annotation of the `f1` local variable in the `equals` method with the *rep* modifier. This is different than what the Prolog inferer found in SUTI1 [3]. The annotation with *rep* is also ok, but it is not necessary.

The found annotation differs in some details from the one found in the previous work. The general Universe type structure is the same.

5.1.2 Storage

This program is already heavy annotated. It consist of three classes. The `Storage` class implements a simple container. The other two classes implement the clients using the `Sotrage` class. The `RepClient` uses the container to store his representation. The `PeerClient` uses the container to store an object which is in the *peer* context. The preannotations are written as JML comments in the code. The infered annotations are inserted without the comment sourrounding.

When we try to annotate this code with the conflict internal type representation then the inferer will not be able to find an annotation. The system is overconstrained. The same is true for the Prolog solver. To infer such a heavy annotated program we must use the most liberal internal type representation: The extended conflict internal type system.

The found annotation with the same solution description as above is shown below.

```
class  Storage {
  readonly Object item;
```

```
    void store(readonly Object o) {
    this.item = o;
  }
  Object retrieve() {
   return readonly this.item;
 }
}

class RepClient {
  /*@rep@*/ Object myObj;
  /*@peer@*/ Storage mySto;
  void doSomething() {
    myObj = new rep Object();
    mySto.store(myObj);
    // ...
   myObj = (rep)mySto.retrieve();
  }
}

class PeerClient {
  /*@peer@*/ Object myObj;
  /*@peer@*/ Storage mySto;
  void doSomething() {
    myObj = new peer Object();
    mySto.store(myObj);
    // ...
    myObj = (peer)mySto.retrieve();
  }
}
```

In the `Storage` class the fields and the parameters are annotated as *readonly*. The object creation expressions are annotated as *rep* and *peer*. And both casts are inserted where they are used. With the extended conflict type system is is possible to annotate heavy restricted programs. But we must review the found annotations carefully since more casts are allowed to be inserted. There is no guaranty that inserted casts will work at the execution time.

When using a static Universe type inference tool it is preferable to have not only the data structure but also as many use cases as possible. The more use cases are provided the more hints the tool gets to infer an appropriate Universe type annotation.

5.2 Future Work

We tested the developed tool with some Java programs. But due to time constraints we were not able to implement all planed features. Further testing needs to be done with the software. The following features are prepared to be implemented but are still on the to do list.

- Correct handling of inner classes.
- Signature handling of overloaded methods, not jet tested.
- The handling of the cast policy for the extended internal representation.
- Extending the Simple UI to be able to also change the solution description.

- Using annotations given in additional XML files.

Next to the testing and completion of the tool it would be very intersting to use the tool to annotate large programs. When annotating other Java programs also the solution description may be enhanced.

Early in our testing runs we realized that the tool's capabilities can be enhanced with some user interaction. So implementing an intuitive GUI which allows fast user interaction with the inferer seams to be a fundamental step to really support the developers in annotating existend code.

5.3 Conclusion

The goal of this master thesis was to develop a static Universe type inferer using a SAT backend. With the realization of the backend with a pseudo-boolean solver we were able to model the whole Universe type system in the backend. So it is no longer necessary to split up the type infering problem into two parts: The constraint solving and the optimization part. This is now all done in the pseudo boolean backend.

Modeling tho whole system in the backend brings also the advantage that we do not have to visit the AST representing the Java program for inserting the needed casts. This is done in the backend by explicitly modeling the casts.

The additionally introduced internal type system representations allow the handling of a wider set of programs. So the extended conflict internal representation allows us to annotate heavy preannotated programs. But such a liberal internal representation may introduce more casts which may not work at runtime. Therefore we suggest the usage of a GUI to support reviewing by a developer.

The type constraints by themselves seam not to be enough information to completely annotate programs when we allow to insert casts. There are situations where we do not have any information about how to cast a variable. In such situations the information retrieved with a runtime type inferer should be helpful. So any correct preannotations from a runtime inferer are welcome.

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] Natalie Kellenberger: Static Universe Type Inference, http://www.sct.inf.ethz.ch/projects/student_docs/Nathalie_Kellenberger/, 2005. Master Thesis.

[4] James Gosling, Bill Joy, Guy Steele, Gilad Bracha: Universes: Java(TM) Language Specification, The (3rd Edition). *Java Series, Sun*, 2005.

[5] Werner Dietl, Peter Müller, and Daniel Schregenberger: Universe Type System - Quick-Reference. *Technical Report, ETH Zürich*, 2005.

[6] Dietl, W. and Müller, P.: Exceptions in Ownership Type Systems, *Formal Techniques for Java-like Programs*, 2004.

[7] Jonathan Aldrich: Incremental Type Inference for Software Engineering. *University of Washington*, December 1997.

[8] Peter Müller: Modular Specification and Verification of Object-Oriented Programs, volume 2262 of *Lecture Notes in Computer Science*, Springer Verlag, 2002.

[9] Jakarta Project. XMLBeans. Available from http://xmlbeans.apache.org/.

[10] Retroweaver. Available from http://retroweaver.sourceforge.net/.

[11] Frank Lyner: Runtime Universe Type Inference, http://www.sct.inf.ethz.ch/projects/student_docs/Frank_Lyner/, 2005. Master Thesis.

[12] Marco Meyer: Interaction with ownership graphs, http://www.sct.inf.ethz.ch/projects/student_docs/Marco_Meyer/, 2005, Semester Project.

[13] Wikipedia: Boolean satisfiability problem, http://en.wikipedia.org/wiki/Boolean_satisfiability_problem, 2006

[14] Fadi Aloul: PBS v2.1: Incremental Pseudo-Boolean Backtrack Search SAT Solver and Optimizer, http://www.eecs.umich.edu/ faloul/Tools/pbs/, 2003

[15] Dave A.D Tompkins, Holger H. Hoos: UBCSAT: An Implementation and Experimentation Environment for SLS Algorithms for SAT and MAX-SAT, *SAT 2004*, 2004.

[16] SATLIB - The Satisfiability Library ,
http://www.satlib.org/

[17] DIMACS - Satisfiability Suggested Format ,
ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/, 1993

[18] Choco - Choco constraint programming system,
http://choco-solver.net.

[19] D.G. Mitchell. A SAT Solver Primer. Bulletin of the European Association for Theoretical Computer Science, 85:112-133, February 2005. Columns: Logic in Computer Science.

[20] P. Van Hentenryck and L. Michel. Control abstraction for local search. In LNCS 2833: *Proc of the Ninth Int'l Conf. of Principles and Practice of Constriant Programming (CP-03)*, 65-80, 2003.

[21] F. Aloul, A. Ramani, I. Markov, and K. Sakallah: Generic ILP versus Specialized 0-1 ILP: an Update *International Conference on Computer Aided Design (ICCAD)*, 450-457, 2002.

[22] Pseudo Boolean Evaluation 2005,
http://www.cril.univ-artois.fr/PB05/