Embedding Static Analysis into Tableaux and Sequent based Frameworks

Tobias Gedell

Department of Computing Science
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
gedell@cs.chalmers.se

Abstract. In this paper we present a method for embedding static analysis into tableaux and sequent based frameworks. In these frameworks, the information flows from the root node to the leaf nodes. We show that the existence of free variables in such frameworks introduces a bi-directional flow, which can be used to collect and synthesize arbitrary information.
We use free variables to embed a static program analysis in a sequent style theorem prover used for verification of Java programs. The analysis we embed is a reaching definitions analysis, which is a common and well-known analysis that shows the potential of our method.
The achieved results are promising and open up for new areas of application of tableaux and sequent based theorem provers.

1 Introduction

The aim of our work is to integrate static program analysis into theorem provers used for program verification. In order to do so, we must bridge the mismatch between the synthetic nature of static program analysis and analytic nature of tableaux and sequent calculi. One of the major differences is the flow of information.

In a program analysis, information is often synthesized by dividing a program into its subcomponents, calculating some information for each component and then merging the calculated information. This gives a flow of information that is directed bottom-up, with the subcomponents at the bottom.

Both tableaux and sequent style provers work in the opposite way. They take a theorem as input and, by applying the rules of their calculi, gradually divide it into branches, corresponding to logical case distinction, until all branches can be proved or refuted. In a ground calculus, there is no flow of information between the branches. Neither is there a need for that since the rules of the calculus only extend the proof by adding new nodes. Because of this, the information flow in a ground calculus is uni-directional—directed top-down, from the root to the leafs of the proof tree.

Tableaux calculi are often extended with *free variables* which are used for handling universal quantification (in the setting of sequent calculi, free variables correspond to *meta variables*, which are used for existential quantification) [Fit96]. Adding free variables breaks the uni-directional flow of information. When a branch chooses to instantiate a free variable, the instantiation has to be made visible at the point where the free

variable was introduced. Therefore, some kind of information flow backwards in the proof has to exist. By exploiting this bi-directional flow we can collect and synthesize arbitrary information which opens up for new areas of application of the calculi.

We embed our program analysis in a sequent calculus using meta variables. The reason for choosing a program analysis is that logics for program verification could greatly benefit from an integration with program analysis. An example of this is the handling of loops in programs. Often a human must manually handle things like loops and recursive functions. Even for program constructs, which a verification system can cope with automatically, the system sometimes performs unnecessary work. Such a system could benefit from having a program analysis that could cheaply identify loops and other program constructs that can be handled using specialized rules of the program logics that do not require user interaction. An advantage of embedding a program analysis in a theorem prover instead of implementing it in an external framework, is that it allows for a closer integration of the analysis and prover.

The main contributions of this work are that:

- We show how synthesis can be performed in a tableau or sequent style prover, which opens up for new areas of application.
- We show how the rules of a program analysis can be embedded into a program logic and coexist with the original rules by using a tactic language.
- We give a proof-of-concept of our method. We do this by giving the full embedding of a program analysis in an interactive theorem prover.

The outline of this paper is as follows: In Section 2 we elaborate more on how we use the bi-directional flow of information; In Section 3 we briefly describe the theorem prover used for implementing our program analysis; In Section 4 we describe the program analysis; in Section 5 we present the embedding of the analysis in the theorem prover; in Section 6 we draw some conclusions; and in Section 7 we discuss future work.

2 Flow of Information

By using the mechanism for free variables we can send information from arbitrary nodes in the proof to nodes closer to the root. This is very useful to us since our program analysis needs to send information from the subcomponents of the program to the root node. In a proof, the subcomponents of the program correspond to leaf nodes. To show how it works, consider a tableau created with a destructive calculus where, at the root node, a free variable I is introduced. When I is instantiated by a branch closure, the closing substitution is applied to all branches where I occurs. This allows us to embed various analyses. One could, for example, imagine a very simple analysis that finds out whether a property P is true for any branch in a proof. In order to do so, we modify the closure rule. Normally, the closure rule tries to find two formulas φ and $\neg\psi$ in the same branch and a substitution that unifies φ and ψ. The new closure rule is modified to search for a closing substitution for a branch and if it finds one, check whether P is true for the particular branch. If it is, then the closing substitution is extended with an instantiation of the free variable I to a constant symbol c. We can now use this calculus

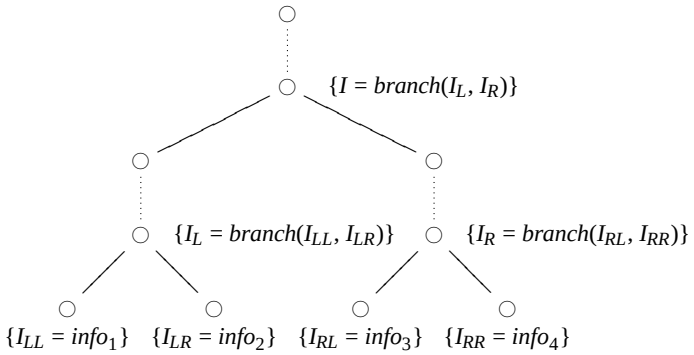

Fig. 1. Example tableau

to construct a proof as usual and when it is done, check whether I has been instantiated or not. If it has, then we know that P was true for at least one of the branches. Note that we are not interested in *what* I was instantiated to, just the fact that it *was* instantiated.

There is still a limit to how much information that can be passed to the root node. It is not possible to gather different information from each branch closure since they all use the same variable, I, to send their information. In particular, the reaching definitions analysis that we want to embed needs to be able to compute different information for each branch in the proof.

This can be changed by modifying the extension rule. When two branches are created in a proof, two new free variables, I_L and I_R, are introduced and I instantiated to $branch(I_L, I_R)$. I_L is used as the new I-variable for the left branch and I_R for the right branch. By doing this we ensure that each branch has its own variable for sending information. This removes the possibility of conflicting instantiations, since each I-variable will be instantiated at most once, either by extending or closing the branch to which it belongs.

When the tableau shown in Figure 1 has been closed, we get the instantiation of I, which will be the term $branch(branch(info_1, info_2), branch(info_3, info_4))$ that contains the information collected from all four branches. We have now used the tableau calculus to synthesize information from the leaf nodes.

We are now close to being able to implement our program analysis. The remaining problem is that we want to be able to distinguish between different types of branches. An example of this is found in Section 4.2 where different types of branches compute different collections of equations. We overcome this problem by, instead of always using the function symbol *branch*, allowing arbitrary function symbols when branching.

2.1 Non Destructive Calculi

In a non destructive constraint tableau, as described in [Gie01], it is possible to embed analyses using the same method.

In a constraint tableau, each node *n* has a *sink* object that contains all closing substitutions for the sub tableau having *n* as its top node. When adding a node to a branch, all closing substitutions of the branch are added to the node's sink object. The substitutions in the sink object are then sent to the sink object of the parent node. If the parent node is a node with more than one child, it has a *merger* object that receives the substitution and checks whether it is a closing substitution for all children. If it is, then it is propagated upwards to the sink object of the parent node, otherwise it is discarded. If the parent node only has one child, the substitution is directly sent to the node's parent node.

A tableau working like this is called non destructive since the free variables are never instantiated. Instead, a set of all possible closing instantiations is calculated for each branch and propagated upwards. When a closing substitution reaches the root node, the search is over since we know that it closes the entire tableau.

Using our method in a non destructive constraint tableau is easy. We modify the sink object of the root node to not only, when a closing substitution is found, tell us that the tableau is closable but also give us the closing substitution. The infrastructure with the sink objects could also make it easy to implement some of the extensions described in Section 7.

3 The KeY Prover

For the implementation, we choose an interactive theorem prover with a tactic programming language, the KeY system [ABB+04]. The KeY system is a theorem prover for the Java Card language that uses a dynamic logic [Bec01]. The dynamic logic is a modal logic in which Java programs can occur as parts of formulas. An example of this is the formula,

```
<{ i = 1; }> i > 0 ,
```

that denotes that after executing the assignment `i = 1;` the value of the variable *i* is greater than 0.

The KeY system is based on a non destructive sequent calculus with a standard semantics. It is well known that sequent calculi can be seen as the duality of tableaux calculi and we use this to carry over the method described in Section 2 to the sequent calculus used by KeY.

3.1 Tactic Programming Language

Theorem provers for program verification typically need to have a large set of rules at hand to handle all constructs in a language. Instead of hard-wiring these into the core of the theorem prover, one can opt for a more general solution and create a domain specific tactic language, which is used to implement the rules.

The rules written in the tactic language of KeY are called taclets [BGH+04]. A taclet can be be seen as an implementation of a sequent calculus rule. In most theorem provers for sequent calculi, the rules perform some kind of pattern matching on sequents. Typically, the rules consist of a guard pattern and an action. If a sequent matches the guard pattern then the rule is applied and the action performed on the sequent. What it means

for the pattern of a taclet to match a sequent is that there is a unifying substitution for the pattern and the sequent under consideration. The actions that can be performed include closing a proof branch, creating modified copies of sequents, and creating new proof branches.

We now have a look at the syntax of the tactic language and start with one of the simplest rules, the `close_by_true` rule.

```
close_by_true {
 find (==> true)
 close goal
};
```

The pattern matches sequents where *true* can be found on the right hand side. If *true* can be found on the right hand side, we know that we can close the proof branch under consideration, which is done by the `close goal` action.

If we, instead of closing the branch, want to create a modified copy of the sequent we use the `replacewith` action.

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

If we find a negated formula *b* on the left hand side we replace it with *b* on the right hand side.[1] The proof branch will remain open, but contain the modified sequent. We can also create new proof branches by using multiple `replacewith` actions.

So far, we have only considered sequents that do not contain embedded Java programs. When attaching programs to formulas, one has to choose a modality operator. There are a number of different modality operators having different semantics. The diamond operator **<{p}>**ϕ says that there is a terminating execution of the program `p` after which the formula ϕ holds. The box operator **[{p}]**ϕ says that after all terminating executions the formula ϕ holds. For our purpose, the modalities do not have any meaning since we are not trying to construct a proof in the traditional way. Regardless of this, the syntax of the taclet language forces us to have a modality operator attached to all programs. We, therefore, arbitrarily choose to use the diamond operator. In the future, it would be better to have a general-purpose operator with a free semantics that could be used for cases like this.

As an example of a taclet matching an embedded Java program, consider the following taclet, that matches an assignment of a literal to a variable attached to the formula *true* and closes the proof branch:

```
term_assign_literal {
 find (==> <{#var = #literal;}>(true))
 close goal
};
```

[1] Note that Γ and Δ are only implicitly present in the taclet.

4 Reaching Definitions Analysis

The analysis we choose to implement using our technique is *reaching definitions analysis* [NNH99]. This analysis is commonly used by compilers to perform several kinds of optimization such as, for example, loop optimization and constant computation [ASU86]. The analysis calculates which assignments may reach each individual statement in a program. Consider the following program, consisting of three assignments, where each statement is annotated with a label so that we can uniquely identify them.

$$\texttt{a} \overset{0}{=} \texttt{1;}\ \texttt{b} \overset{1}{=} \texttt{1;}\ \texttt{a} \overset{2}{=} \texttt{1;}$$

Let us look at the statement annotated with 1. The statement executed before it (which we will call its previous statement) is the assignment $\texttt{a} \overset{0}{=} \texttt{1;}$ and since `a` has not yet been reassigned it still contains the value 1. We say that the assignment annotated with 0 *reaches* the statement annotated with 1. For each statement, we calculate the set of labels of the assignments that reach the statement before and after it has been executed. We call these sets the entry and exit sets. For this example, the label 0 will be in the entry set of the last assignment but not in its exit set, since the variable `a` is re-assigned. We do not just store the labels of the assignments in the sets, but also the name of the variable that is assigned. The complete entry and exit sets for our example program look as follows:

label	**Entry**	**Exit**
0	{}	{(a, 0)}
1	{(a, 0)}	{(a, 0), (b, 1)}
2	{(a, 0), (b, 1)}	{(b, 1), (a, 2)}

It is important to understand that the results of the analysis will be an *approximation*. It is undecidable to calculate the exact reaching information, which can easily be proven by using the halting problem. We will, however, ensure that the approximation is *safe*, which in this context means that if an assignment reaches a statement then the label of the assignment must be present in the entry set of that statement. The reverse may not hold, a label of an assignment being present in an entry set of a statement, does not necessarily mean that the assignment may reach that statement.

It is easy to see that for any program, a sound result of the analysis would be to let all entry and exit sets be equal to the set of all labels occurring in the program. This result would, of course, not be useful; what we want are as precise results as possible.

The analysis consists of two parts: a constraint-generation part and a constraint-solving part. The constraint-generation part traverses the program and generates a collection of equations defining the entry and exit sets. The equations are then solved by the constraint-solving part that calculates the actual sets.

4.1 Input Language

As input language, we choose a very simple language, the WHILE-language, which consists of assignments, block statements and if- and while-statements. We choose a

simple language because we do not want to wrestle with a large language but instead show the concept of how the static program analysis can be implemented.

In the language, a program consists of a number of statements.

$$
\begin{array}{lll}
\text{Programs} & \mathit{program} ::= & \mathit{stmt}^{+} \\
\text{Statements} & \mathit{stmt} ::= & \mathit{var} \overset{lbl}{=} \mathit{expr}\textbf{;} \\
 & \mid & \textbf{if}_{lbl}\textbf{(}\mathit{term}\textbf{)}\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt} \\
 & \mid & \textbf{while}_{lbl}\textbf{(}\mathit{term}\textbf{)}\ \mathit{stmt} \\
 & \mid & \textbf{\{}\mathit{stmt}^{*}\textbf{\}}
\end{array}
$$

lbl ranges over the natural numbers and will be unique for each statement. We do not annotate block statements since they are just used to group multiple statements.

To simplify our analysis, we impose the restriction that all expressions *expr* must be free from side-effects. Since removing side-effects from expressions is a simple and common program transformation, this restriction is reasonable to make.

4.2 Rules of the Analysis

We now look at the constraint-generation part of the analysis and start by defining the collections of equations that will be generated. These equations will characterize the reaching information in the analyzed program.

$$
\begin{array}{lll}
\text{Equations} & \Pi ::= & \emptyset \\
 & \mid & \textbf{Entry}(\mathit{lbl}) = \Sigma \\
 & \mid & \textbf{Exit}(\mathit{lbl}) = \Sigma \\
 & \mid & \Pi \wedge \Pi
\end{array}
\tag{1}
$$

$\emptyset$ is the empty collection of equations. **Entry**(*lbl*) = Σ and **Exit**(*lbl*) = Σ are equations defining the entry and exit sets of the statement annotated with *lbl* to be equal to the set expression Σ. We let $\wedge$ be the conjunction operator that merges two collections of equations.

The set expressions,

$$
\begin{array}{lll}
\text{Set expressions} & \Sigma ::= & \emptyset \\
 & \mid & (\mathit{var}, \mathit{lbl}) \\
 & \mid & \textbf{Entry(}\mathit{lbl}\textbf{)} \\
 & \mid & \textbf{Exit(}\mathit{lbl}\textbf{)} \\
 & \mid & \Sigma \cup \Sigma \\
 & \mid & \Sigma\text{-}\Sigma
\end{array}
\quad , \tag{2}
$$

are used to build up the entry and exit sets. $\emptyset$ is the empty set (the overloading of this symbol will not cause any confusion). $(\mathit{var}, \mathit{lbl})$ is the set consisting of only a single reaching assignment. **Entry**(lbl) and **Exit(***lbl***)** refer to the values of the entry and exit sets of the statement annotated with *lbl*. $\cup$ and **-** are the union and difference operators.

The rules of the analysis are of the form $\ell_0 \vdash s \Downarrow \ell_1 : \Pi$, where *s* is the statement under consideration, ℓ_0 is the label of the statement executed before *s* (we will sometimes call this statement the *previous* statement), ℓ_1 the label of the last executed statement in *s*, and Π the equations characterizing the reaching information of the statement *s*.

The intuition behind this form is that we need to know the label of the statement executed before s because we will use its exit set when analyzing s. After we have analyzed s, we need to know the label of the last executed statement in s (which will often be s itself) because the statement executed after s needs to use the right exit set. Then, the most important thing to know is, of course, what equations were collected when analyzing s.

In the assignment rule,

$$\textsc{Assign}\quad \frac{}{\begin{array}{c}\ell_0 \vdash x \overset{\ell_1}{:=} e; \Downarrow \ell_1 : \mathbf{Entry}(\ell_1) = \mathbf{Exit}(\ell_0) \wedge \\ \mathbf{Exit}(\ell_1) = (x,\ell_1) \cup (\mathbf{Entry}(\ell_1) - \bigcup_{\ell \in lbl} (x,\ell))\end{array}},$$

we know that the reaching assignments in the entry set will be exactly those that were reaching after the previous statement was executed. This is expressed by the equation $\mathbf{Entry}(\ell_1) = \mathbf{Exit}(\ell_0)$. For the exit set, we know that all previous assignments of x will no longer be reaching. The assignments of all other variables will remain untouched. We therefore let the exit set be equal to the entry set from which we have first removed all previous assignments of x and then added the assignment (x,ℓ_1). This is expressed by the equation $\mathbf{Exit}(\ell_1) = (x,\ell_1) \cup (\mathbf{Entry}(\ell_1) - \bigcup_{\ell \in lbl}(x,\ell))$.

So far, we have not seen the need for including the label of the previous statement in the rules. This is illustrated by the rule for if-statements:

$$\textsc{If}\quad \frac{\ell_0 \vdash s_0 \Downarrow \ell_2 : \Pi_0 \qquad \ell_0 \vdash s_1 \Downarrow \ell_3 : \Pi_1}{\begin{array}{c}\ell_0 \vdash \mathbf{if}_{\ell_1}(e)\ s_0\ \mathbf{else}\ s_1 \Downarrow \ell_1 : \Pi_0 \wedge \Pi_1 \wedge \mathbf{Entry}(\ell_1) = \mathbf{Exit}(\ell_0) \wedge \\ \mathbf{Exit}(\ell_1) = \mathbf{Exit}(\ell_2) \cup \mathbf{Exit}(\ell_3)\end{array}}$$

For an if-statement, the entry set will be equal to the exit set of the previous statement, which is expressed by the equation $\mathbf{Entry}(\ell_1) = \mathbf{Exit}(\ell_0)$. When analyzing the two branches s_0 and s_1, we use l_0 as the label of the previous statement since it is important that they, when referring to the exit set of the previous statement, use $\mathbf{Exit}(l_0)$ and not the exit set of the if-statement. From the two branches, we get the collections of the generated equations Π_0 and Π_1, along with the labels l_2 and l_3, which are the labels of the last executed statements in s_0 and s_1. Since we do not know which branch is going to be taken, we must approximate and assume that both branches can be taken. The exit set of the if-statement will therefore be equal to the union of the exit set of the last executed statements in s_0 and s_1, expressed by the equation $\mathbf{Exit}(\ell_1) = \mathbf{Exit}(\ell_2) \cup \mathbf{Exit}(\ell_3)$.

The rule for while-statements,

$$\textsc{While}\quad \frac{\ell_1 \vdash s \Downarrow \ell_2 : \Pi_0}{\begin{array}{c}\ell_0 \vdash \mathbf{while}_{\ell_1}(e)\ s \Downarrow \ell_1 : \Pi_0 \wedge \mathbf{Entry}(\ell_1) = \mathbf{Exit}(\ell_0) \cup \mathbf{Exit}(\ell_2) \wedge \\ \mathbf{Exit}(\ell_1) = \mathbf{Entry}(\ell_1)\end{array}},$$

differs significantly from the rule for if-statements. For the entry set, we include the exit set of the last executed statement before the loop, but also the exit set of the last

executed statement in the loop body. We must do this because there are two execution paths leading to the while loop. The first is from the statement executed before the loop, and the second from executing the loop body. For the exit set, we do not know if the body was executed or not. We could, therefore, let the exit set be equal to the union of the entry set of the while-statement and the exit set of the last executed statement in s. Since this is exactly what the entry set is defined to be, we just let the exit set be equal to the entry set. When analyzing the body of the loop we must once again approximate. The first time s is executed, it should use the exit set of l_0, since that was the last statement executed. The second time and all times after that, it should instead use the exit set of l_1, since the body of the while loop was the last statement executed. We approximate this by not separating the two cases and always use l_1 as the label of the previous statement.

We do not have a special rule for programs. Instead, we treat a program as a block statement and use the rules for sequential statements, which should not require much description:

$$\textsc{Seq-Empty}\quad \frac{}{\ell_0 \vdash \{\} \Downarrow \ell_0 : \emptyset}$$

$$\textsc{Seq}\quad \frac{\ell_0 \vdash s_1 \Downarrow \ell_1 : \Pi_1 \quad \cdots \quad \ell_{n-1} \vdash s_n \Downarrow \ell_n : \Pi_n}{\ell_0 \vdash \{s_1 \ \ldots \ s_n\} \Downarrow \ell_n : \Pi_1 \wedge \cdots \wedge \Pi_n}$$

5 Embedding the Analysis into the Prover

5.1 Encoding the Datatypes

In order to encode Σ, Π, and labels, we must declare the types we want to use. We declare **VarSet** which is the type of Σ, **Equations** which is the type of Π and **Label** which is the type of labels. The type of variable names, **Quoted**, is already defined by the system.

In the constructors for Σ, defined by (2), we have, for convenience, replaced the difference operator with the constructor **CutVar**. **CutVar**(s, x) denotes the set expression $s - \bigcup_{\ell \in lbl}(x, \ell)$. Our constructors are defined as function symbols by the following code:

```
VarSet Empty;
VarSet Singleton(Quoted, Label);
VarSet Entry(Label);
VarSet Exit(Label);
VarSet Union(VarSet, VarSet);
VarSet CutVar(VarSet, Quoted);
```

The constructors for Π, defined by (1), are defined analogously to the ones for Σ:

```
Equations None;
Equations EntryEq(Label, VarSet);
Equations ExitEq(Label, VarSet);
Equations Join(Equations, Equations);
```

The KeY system does not feature a unique labeling of statements so we need to annotate each statement ourselves. In order to generate the labels we define the **Zero** and **Succ** constructors with which we can easily enumerate all needed labels. The first label will be **Zero**, the second **Succ(Zero)**, the third **Succ(Succ(Zero))**, and so on.

```
Label Zero;
Label Succ(Label);
```

Since the rules of the analysis refer back to the exit set of the previous statement, there is a problem with handling the very first statement of a program (which does not have any previous statement). To solve this problem we define the label **Start** which we exclusively use as the label of the (non-existing) statement before the first statement. When solving the equations we let the exit set of this label, **Exit(Start)**, be the empty set.

```
Label Start;
```

Since one can only attach *formulas* to embedded Java programs, we need to wrap our parameters in a predicate. The parameters we need are exactly those used in our judgments,

$$\ell_0 \vdash s \Downarrow \ell_1 : \Pi \quad .$$

We wrap the label of the previous statement, ℓ_0, the label of the last executed statement, ℓ_1, and the collection of equations, Π, in a predicate called *wrapper* (we do not need to include the statement *s* since the wrapper will be attached to it). In the predicate, we also include two labels needed for the generation of the labels used for annotating the program: the first unused label before annotating the statement and the first unused label after annotated the statement. The wrapper formula looks as follows:

```
wrapper(Label, Label, Equations, Label, Label);
```

5.2 Encoding the Rules

Before implementing the rules of our analysis as taclets, we declare the variables that we want to use in our taclets. These declarations should be fairly self explanatory.

```
program variable #x;
program simple expression #e;
program statement #s, #s0, #s1;
Equations pi0, pi1, pi2;
Label lbl0, lbl1, lbl2, lbl3, lbl4, lbl5;
Quoted name;
```

We now look at how the rules of the analysis are implemented and start with the rule for empty block statements. When implemented as a taclet we let it match an empty block statement, written as **<{ {} }>**, and a wrapper formula where the first argument is equal to the second argument, the collection of equations is empty, and the fourth

argument is equal to the fifth. The formula pattern is written as **wrapper(lbl0, lbl0, None, lbl1, lbl1)**. The action that should be performed when this rule is applied is that the current proof branch should be closed. This is the case because the Seq-Empty rule has no premises. The complete taclet is written as follows:

```
rdef_seq_empty {
 find (==> <{{}}>(wrapper(lbl0, lbl0, None, lbl1, lbl1)))
 close goal
};
```

The rule for non-empty block statements is a bit more tricky. The rule handles an arbitrary number of statements in a block statement. This is, however, hard to express in the taclet language. Instead, we modify the rule to separate the statements into the head and the trailing list. This is equivalent to the original rule except that a block statement needs one application of the rule for each statement it contains. After being modified, the rule looks like this, where we let $\overline{s_2}$ range over lists of statements:

$$\text{SEQ-MODIFIED}\quad \frac{\ell_0 \vdash s_1 \Downarrow \ell_1 : \Pi_1 \qquad \ell_1 \vdash \{\overline{s_2}\} \Downarrow \ell_2 : \Pi_2}{\ell_0 \vdash \mathbf{\{}s_1\ \overline{s_2}\mathbf{\}} \Downarrow \ell_2 : \Pi_1 \wedge \Pi_2}$$

When implemented as a taclet, we let it match the head and the tail of the list, written as **<{.. #s1 ...}>**. In this pattern, **#s1** matches the head and the dots, **.. ...**[2], match the tail. We also let it match a wrapper formula containing the necessary labels together with the conjunction of the two collections of equations Π_1 and Π_2. For each premise, we create a proof branch by using the **replacewith** action. Note how the two last labels are threaded through the taclet:

```
rdef_seq {
 find (==> <{.. #s1 ...}>(wrapper(lbl0, lbl2, Join(pi1, pi2), lbl3, lbl5)))
 replacewith (==> <{#s1}>(wrapper(lbl0, lbl1, pi1, lbl3, lbl4)));
 replacewith (==> <{.. ...}>(wrapper(lbl1, lbl2, pi2, lbl4, lbl5)))
};
```

In the rule for assignments, we must take care of the annotation of the assignment. Since we know that the fourth argument in the wrapper predicate is the first free label, we bind `lbl1` to it. We then use `lbl1` to annotate the assignment. Since we have now used that label, we must increment the counter of the first free label. We do that by letting the fifth argument be the successor of `lbl1`. (Remember that the fifth argument in the wrapper predicate is the first free label after annotated the statement.) In the taclet we use a **varcond** construction to bind the name of the variable matching `#x` to `name`.

```
rdef_assign  {
 find (==> <{#x = #e;}>
  (wrapper(lbl0, lbl1,
           Join(EntryEq(lbl1, Exit(lbl0)),
```

[2] The leading two dots match the surrounding context which for our analysis is known to always be empty. They are however still required by the KeY system.

```
               ExitEq (lbl1, Union(Singleton(name, lbl1),
                                   CutVar(Entry(lbl1), name)))),
          lbl1, Succ(lbl1))))
 varcond (name quotes #x)
 close goal
};
```

The taclet for if-statements is larger than the previously shown taclets, but since it introduces no new concepts, it should be easily understood:

```
rdef_if {
 find (==> <{if(#e) #s0 else #s1}>
  (wrapper(lbl0, lbl1,
           Join(Join(pi0, pi1),
                Join(EntryEq(lbl1, Exit(lbl0)),
                     ExitEq (lbl1, Union(Exit(lbl2), Exit(lbl3))))),
           lbl1, lbl5)))
  replacewith (==> <{#s0}>(wrapper(lbl0, lbl2, pi0, Succ(lbl1), lbl4)));
  replacewith (==> <{#s1}>(wrapper(lbl0, lbl3, pi1, lbl4, lbl5)))
};
```

This is also the case with the taclet for while-statements and it is, therefore, left without further description:

```
rdef_while {
 find (==> <{while(#e) #s}>
 (wrapper(lbl0, lbl1,
          Join(pi0, Join(EntryEq(lbl1, Union(Exit(lbl0),Exit(lbl2))),
                         ExitEq (lbl1, Entry(lbl1)))),
          lbl1, lbl3)))
  replacewith (==> <{#s}>(wrapper(lbl1, lbl2, pi0, Succ(lbl1), lbl3)))
};
```

5.3 Experiments

We have tested the implementation of our analysis on a number of different programs. For all tested programs the analysis gave the expected entry and exit sets, which is not that surprising since there is a one-to-one correspondence between the rules of the analysis and the taclets implementing them.

As an example, consider the minimal program `a = 1;`, consisting of only an assignment. We embed this program in a formula, over which we existentially quantify the equations, `s`, the label of the last executed statement, `lbl0`, and the first free label after annotated the program, `lbl1`:

```
ex lbl0:Label. ex s:Equations. ex lbl1:Label.
 <{ a = 1; }>wrapper(Start, lbl0, s, Zero, lbl1)
```

When applying the rules of the analysis, the first thing that happens in that `lbl0`, `s`, and `lbl1` are instantiated with meta variables. This is done by a built-in rule for

existential quantification. The resulting formula is the following where `L0`, `S` and `L1` are meta variables:

```
<{ a = 1; }>wrapper(Start, L0, S, Zero, L1)
```

We know that the KeY system will succeed in automatically applying the rules since the analysis is complete and, therefore, works for all programs. Being complete is an essential property for all program analyses and for our analysis it is easy to see that for any program there exists a set of equations which characterize the reaching information of the program.

When the proof has been created, we fetch the instantiation of all meta variables, which for our example are the following.

```
{
 S : Equations =
    Join(
     EntryEq(L0, Exit(Start)),
     ExitEq (L0, Union(Singleton(a, L0), CutVar(Entry(L0), a)))),
 L0 : Label = Zero,
 L1 : Label = Succ(L0)
}
```

We take these constraints and let a stand-alone constraint solver solve them. Recall that the analysis is divided into two parts. The first part, which is done by the KeY system, is to collect the constraints. The second part, which is done by the constraint solver, solves the constraints.

The constraint solver extracts the equations from the constraints and solves them yielding the following sets, which is the expected result:

```
Entry_0 = {}
Exit_0  = {(a, 0)}
```

6 Conclusions

It is interesting to see how well-suited an interactive theorem prover such as the KeY system is to embed the reaching definitions analysis in. One reason for this is that the rules of the dynamic logic are, in a way, not that different from the rules of the analysis. They are both syntax-driven, i.e., which rule to apply is decided by looking at the syntactic shape of the current formula or statement. This shows that theorem provers with free variables or meta variables can be seen as not just theorem provers for a specific logic but, rather, as generic frameworks for syntactic manipulation of formulas. Having this view, it is not that strange that we can be rather radical and disregard the usual semantic meaning of the tactic language, and use it for whatever purpose we want.

The key feature that allows us to implement our analysis is the machinery for meta variables, that we use to create a bi-directional flow of information. Using meta variables, we can let our analysis collect almost any type of information. We are, however,

limited in what calculation we can do on the information. So far, we cannot do any calculation on the information while constructing the proof. We cannot, for example, do any simplification of the set expressions. One possible way of overcoming this would be to extend the constraint language to not just include syntactic constraints but also semantic constraints.

When it comes to the efficiency of the implementation of the constraint-generation part, it is a somewhat open issue. One can informally argue that the overhead of using the KeY system, instead of writing a specialized tool for the analysis, should be a constant factor. It might be the case that one needs to optimize the constraint solver to handle unification constraints in a way that is more efficient for the analysis. An optimized constraint solver should be able to handle all constraints, generated by the analysis, in a linear way.

7 Future Work

This work presented in this paper is a starting point and opens up for a lot of future work:

- Try different theorem provers to see how well the method presented in this paper works for other theorem provers.
- Further analyse the overhead of using a theorem prover to implement program analyses.
- Modify the calculus of the KeY prover to make use of the information calculated by the program analysis. We need to identify where the result of the analysis can help and how the rules of the calculus should be modified to use it. It is when this is done that the true potential of the integration is unleashed.
- Explore other analyses. We chose to implement the *reaching definitions analysis* because it is a well known and simple analysis that is well suited for illustrating our ideas. Now that we have shown that it is possible to implement a static program analysis in the KeY system, it is time to look for the analyses that would benefit the KeY system the most. Among the possible candidates for this are:
  - An analysis that calculates the possible side-effects of a method. For example what objects and variables that may change.
  - A path-based flow analysis helping the KeY system to resolve aliasing problems.
  - A flow analysis calculating the set of possible implementation classes of objects. This would help reducing the branching for abstract types like interfaces and abstract classes
  - A null pointer analysis that identifies object references which are not equal to null. This would help the system which currently has to always check whether a reference is equal to null before using it.

One limitation of the sequent calculus in the KeY prover is that the unification constraints, used for instantiating the meta variables, can only express syntactic equality. This is a limitation since it prevents the system from doing any semantic simplification

of the synthesized information. If it was able to perform simplification of the information while it is synthesized, not only could it make the whole process more efficient, but also let it guide the construction of the proof to a larger extent. Useful extensions of the constraint language are for example the common set operations: test for membership, union, intersection, difference and so on. In a constraint tableaux setting, the simplification of these operations would then take place in the sink objects associated with each node in the proof.

A more general issue that is not just specific to the work presented in this paper is on which level static program analysis and theorem proving should be integrated. The level of integration can vary from having a program analysis run on a program and then give the result of the analysis together with the program to a theorem prover, to having a general framework in which program analysis and theorem proving are woven together. The former kind of integration is no doubt the easiest to implement but also the most limited. The latter is much more dynamic and allows for an incremental exchange of information between the calculus of the prover and program analysis.

Acknowledgments

We would like to thank Wolfgang Ahrendt, Martin Giese and Reiner Hähnle for the fruitful discussions and help with the KeY system. We thank the following people for reading the draft and providing valuable feedback: Richard Bubel, Jörgen Gustavsson, Kyle Ross, Philipp Rümmer and the anonymous reviewers.

References

[ABB+04] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle and Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 2004. Online First issue, to appear in print.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Princiles, Techniques, and Tools.* Addison-Wesley, 1986.

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

[BGH+04] Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Andreas Roth, Philipp Rümmer, and Steffen Schlager. Taclets: a new paradigm for constructing interactive theorem provers. *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales, Serie A: Matemáticas*, to appear, 2004. Special Issue on Computational Logic.

[Fit96] Melvin C. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, New York, second edition, 1996.

[Gie01] Martin Giese. Incremental Closure of Free Variable Tableaux. In *Proc. Intl. Joint Conf. on Automated Reasoning, Siena, Italy*, number 2083 in LNCS, pages 545–560. Springer-Verlag, 2001.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.