A Practical Course on KIV

Gidon Ernst          Dominik Haneberg            Jörg Pfähler
Wolfgang Reif        Gerhard Schellhorn         Kurt Stenzel
                       Bogdan Tofan

Institute for Software and Systems Engineering
University of Augsburg
Contents

1 Getting Started 4
  1.1 Overview .............................................. 4
  1.2 Starting KIV .......................................... 4
  1.3 Logical Foundations ................................ 4

2 Predicate Logic Proofs in KIV 9
  2.1 Editing and Loading Theorems ....................... 9
  2.2 Syntax and Special Character Input .................. 11
  2.3 Proofs in KIV ......................................... 13
  2.4 Rule set for predicate logic .......................... 16
  2.5 Basic rules ........................................... 18
  2.6 Heuristics for basic rules ............................ 19
  2.7 Exercise 1 ........................................... 21

3 Simplification 26
  3.1 General Comments ..................................... 26
  3.2 Typical simplification steps ......................... 26
  3.3 Syntactical form of simplifier rules ................ 27
  3.4 Input and use of simplifier rules .................... 29

4 Theory Proofs 30
  4.1 Editing Theorems ...................................... 30
  4.2 Short Description of the Menu Commands ............ 31
  4.3 Rules for Predicate Logic ............................ 32
  4.4 Predefined Sets of Heuristics ....................... 36
  4.5 Description of the Heuristics ....................... 37
  4.6 Exercise 2 ........................................... 39

5 Algebraic Specifications 42
  5.1 Structured Algebraic Specifications ................. 42
  5.2 The Specification Methodology in KIV ............... 48
  5.3 Creating Specifications in KIV ..................... 50
  5.4 Exercise 3 ........................................... 52

6 Hoare's Proof Strategy 54
  6.1 Introduction .......................................... 54
  6.2 The Proof Strategy .................................... 54
  6.3 Syntax and Semantics of Programs .................... 55
  6.4 The Proof Rules ...................................... 56
  6.5 An Example Proof ..................................... 59
  6.6 Normalization ........................................ 61
  6.7 Heuristics ............................................ 61
Chapter 1

Getting Started

1.1 Overview

This course deals with the development of provably correct programs and software systems. The KIV system is used as a tool and development environment throughout the course. The first exercise deals with propositional and predicate logic, proofs and their representation in KIV. The topic of the second exercise are proofs over data structures: theorems over specifications of lists and heaps have to be proved. In the third exercise you will create a structured specification yourself. The fourth and fifth exercises deal with program verification “in the small” and “in the large”.

The exercises are solved with the KIV system. As a preparation, documentation with introductory material, theoretical background, and the exercises is issued one week in advance. It is indispensible to read the documentation in advance and to prepare a solution. Otherwise it will not be possible to solve the exercises with the system during the scheduled time. However, the computers are also available at other times, if the normal work is not hindered.

1.2 Starting KIV

Installation

Directories for every group have been set up. Instructions how to do the initial installation will be given in the first course.

Starting KIV

Just type

 ./startkiv.sh

in a shell in your home directory.

1.3 Logical Foundations

The logic underlying the KIV system combines Higher-Order Logic and Dynamic Logic. Higher-Order Logic extends first-order logic with functions that have functions as arguments and results. Also function variables and lambda expressions \( \lambda x.e \) that denote anonymous functions are allowed. Dynamic-Logic is an extension of first-order logic by modal program formulae. Dynamic Logic (DL) allows the expression of properties of programs like partial and total correctness, program equivalence etc. By a suitable axiomatisation, proofs that show these properties become possible. The following chapter defines the syntax and the semantics of the first-order part of DL formally.
Finally a sequent calculus for DL is defined. For a survey on DL see [Har84, SA91] resp. [And86] are good introduction to first-order logic and to sequent calculus resp. to higher-order logic. In this chapter we introduce the first-order fragment of the logic.

Syntax
Signatures
A signature $\Sigma = (S, OP, X)$ consists of a finite set $S$ of sorts, a finite family $OP = \bigcup_{s,s' \in S} OP_{s,s'}$ of operations (with a list of argument sorts $s$ and target sort $s'$) and a family $X = \bigcup_{s \in S} X_s$ of countably infinite sets of variables.

We always assume that the sorts $S$ contain at least bool and nat, and the operations contain the usual operations on bool ($true, false, \land, \lor, \rightarrow, \leftrightarrow, \neg$) and nat ($0, succ, pred, +$).

First-order expressions
For a given signature $\Sigma$, the set of expressions $EXPR := \bigcup_{s \in S} EXPR_s$, where $EXPR_s$ are defined to be the smallest sets with

- $X_s \subseteq EXPR_s$ for every $s \in S$
- If $f \in OP_{s,s}$ and $t \in T_s$ then $f(t) \in EXPR_s$
- If $\varphi \in FMA$ and $\overline{x} \in \hat{X}_s$ then $\forall \overline{x}. \varphi \in FMA$.
  $\hat{X}_s$ is the set of duplicate free lists of variables of sorts $s$.
- If $\varphi \in FMA$ and $\overline{x} \in \hat{X}_s$ then $\exists \overline{x}. \varphi \in FMA$
- If $t, t' \in T_s$, then $t = t' \in FMA$
- If $\varphi \in FMA$ and $t, t' \in EXPR_s$, then $(\varphi \supset t; t') \in EXPR_s$

In the definition, $FMA$ (formulas) abbreviates $EXPR_{\text{bool}}$, the set $T_s$ (terms of sort $s$) is the subset of $EXPR_s$, that contains neither quantifiers nor programs. $BXP$ (boolean expressions) is $T_{\text{bool}}$. We also write $EXPR(\Sigma)$ and $FMA(\Sigma)$ for $EXPR$ and $FMA$, when we want to emphasize that the the sets depend on $\Sigma$.

Semantics
Algebra
Given a signature $\Sigma$, an algebra $A$ consists of a nonempty set $A_s$ for every sort $s$ and an operation $f_A : A_s \to A_{s'}$ for every $f \in OP_{s,s'}$. We assume $A_{\text{bool}} = \{tt, ff\}$, $A_{\text{nat}} = \mathbb{N}$ and that operations on booleans and naturals have their usual semantics.

States
Given a signature and an algebra for that signature, a state $z \in ST_A$ is a mapping $z$, which maps variables of sort $s$ to values of $A_s$. The state $z[\overline{x} \leftarrow \overline{a}]$ is the same as $z$, except that the variables in $\overline{x}$ are mapped to the values $\overline{a}$. States are sometimes also called valuations.

Semantics of Expressions
Given an Algebra $A$ and a state $z$ the semantics $[e]_z \in A_s$ of a DL expression $e \in EXPR_s$ is defined as:

- $[x]_z = z(x)$
- $[f(t)]_z = f_A([t]_z)$ for $f \in OP_s$ and $t \in T_s$
CHAPTER 1. GETTING STARTED

• $[\forall x. \varphi]_z = \text{tt}$ with $x \in \hat{X}_z$ iff $[\varphi]_{z[a/z]} = \text{tt}$ for all values $a \in A_z$

• $[\exists x. \varphi]_z = \text{tt}$ with $x \in \hat{X}_z$ iff $[\varphi]_{z[a/z]} = \text{tt}$ for some value $a \in A_z$

• $[(\varphi \supset e; e')]_z$ is $[e]_z$, if $[\varphi]_z = \text{tt}$, and $[e']_z$ otherwise.

Models and Validity

A formula $\varphi$ holds over a $\Sigma$-Algebra and a valuation $z$, short $A, z \models \varphi$, iff (if and only if) $[\varphi]_z = \text{tt}$.

A $\Sigma$-Algebra is a model of a formula $\varphi$, short $A \models \varphi$, iff for all states $z$: $A, z \models \varphi$. A formula $\varphi$ is valid, if all $\Sigma$-algebras are a model of $\varphi$. A formula $\varphi$ follows from a set of formulas $\Phi$, short: $\Phi \models \varphi$, if any model of all formulas from $\Phi$ is also a model of $\varphi$.

Sequent calculus

To define the basic axioms for KIV supports a sequent calculus, since formal reasoning with its rules resembles informal proofs by hand quite closely. For an introduction to 'natural deduction' with sequent calculi see [Ric78] und [SA91].

In the following we will define informally the important concepts of sequent calculus. This should suffice to work with the rules.

Sequents

Let $\varphi_1, \ldots, \varphi_n, \psi_1, \ldots, \psi_m \in \text{FMA}$ be two lists of formulas with $n, m \geq 0$. Then the scheme

$$\varphi_1, \ldots, \varphi_n \vdash \psi_1, \ldots, \psi_m$$

is called a sequent. $\varphi_1, \ldots, \varphi_n$ is called the antecedent, $\psi_1, \ldots, \psi_m$ the succedent of the sequent. A sequent is a simply a way to present the formula

$$\varphi_1 \land \ldots \land \varphi_n \rightarrow \psi_1 \lor \ldots \lor \psi_m$$

The meaning of sequent therefore is: The conjunction of the antecedent formulas implies the disjunction of the succedent formulas. Note that to determine, if a sequent has a model, it’s free variables have to be treated, as if the whole sequent were universally quantified.

The empty conjunction is defined to be true, the empty disjunction is defined to be false. A sequent with empty antecedent therefore is true, if and only if the antecedent contains a contradiction. In the following rules, instead of using concrete formulas we will use formula schemes (e. g. $\varphi, \psi$). Such placeholders are called meta variables, in contrast to elements from $X$, which are called object variables. We will use meta variables $\varphi, \psi$ for formulas. We also use the capital greek letters $\Gamma, \Delta$ as meta variables for lists of formulas. As an example $\Gamma, \varphi \vdash \Delta$ is a scheme for a sequent. Any sequent with nonempty antecedent would be an instance of the scheme.

Rules of sequent calculus

If $S_1, \ldots, S_n, S (n \geq 0)$ are sequents (possibly containing meta variables), then the scheme

$$\frac{S_1 \quad S_2 \quad \ldots \quad S_n}{S}$$

is called a sequent rule. $S$ is called the conclusion, $S_1, \ldots, S_n$ die premises of the rule. sequent rules, which have an empty set of premises ($n = 0$) are called axioms. $C$ is a side condition to form instances of the meta variables occurring of the rule (usually it restricts the formulas that instantiate the meta variables not to contain certain object variables). Often this condition is true, and we drop it in this case.

The semantics of a rule is: the conclusion follows from the premises. It views all the sequents as they were formulas. Sequent calculus consists of a finite set of such rules.
Derivations

Rules of sequent calculus can be combined to derivations. A derivation is a tree, whose nodes are sequents. All elementary subtrees of such a derivation must be instances of sequent rules. Exactly as for rules the root of the tree is called ‘conclusion’, the leaves are called ‘premises’. A more formal definition of derivation is:

- A sequent is a derivation which has itself as root and as only premise.
- A tree with conclusion K and subtrees \( T_1 \ldots T_n \) (\( n \geq 0 \)) is a derivation, if all subtrees \( T_1 \ldots T_n \) are derivations, and there is a sequent rule R with conclusion K', condition C and premises \( P_1 \ldots P_n \), such that there is a substitution s (for the meta variables of R) such that \( s(K') = K \), the condition \( s(B) \) is true and \( s(P_1) \ldots s(P_n) \) are the conclusions of \( T_1 \ldots T_n \).

The semantics of a derivation in sequent calculus is again: The conclusion follows from the premises. A derivation with an empty set of premises is a proof, that its conclusion is valid. The basic rules of sequent calculus for the first-order part of DL are summarized in Sect. 2.5.

In practical work with sequent calculus one usually starts with a goal \( S \) (written as a sequent) which should be proved. Then rules are applied backwards, reducing the goal to simpler subgoals. Note that for most of the rules defined in Sect. 2.5 the premises are simpler than the conclusion in the sense that the maximal number of symbols in the formulas of the sequent decreases. This process stops when the axiom axiom of sequent calculus is reached.

Theories

Finding a proof for a sequent K over some signature \( \Sigma \) using the basic rules establishes, that the sequent is valid in all models of \( \Sigma \). This is called logical reasoning.

But usually we want to have some meaning to be associated with the symbols. To do this we will use a (finite) sets of axioms \( \text{Ax} \subseteq \mathcal{L}(\Sigma,X) \) as preconditions. Theory reasoning allows proofs with open premises that are axioms. A pair \( (\Sigma, \text{Ax}) \) is called theory or a (basic) specification. The existence of a theory is the usual case in program verification. Usually the theory describes data types used in a program like naturals, lists or records. In a later chapter, theories will also be used to describe program systems.

Term generatedness and induction

To describe datastructures adequately, apart from first-order axioms we also need axioms for (structural) induction. Unfortunately an induction scheme is an infinite set of axioms, that cannot be replaced by a finite set of formulas. As a replacement we use generation clauses as axioms. Such a clause has the form

\[ s \text{ generated by } f_1, f_2, \ldots, f_n \]

(‘generated by’-clauses are also possible with several sorts, but since they are rare we will not discuss them.) In the clause \( s \) is a sort, and \( f_1, f_2, \ldots, f_n \) are operation symbols with target sort \( s \). This axiom holds in an algebra \( A \) of the signature, if all elements \( a \in A_s \) can be represented by a term \( t \), which consists of operation symbols from \( f_1, f_2, \ldots, f_n \) only and contains no variables of sort \( s \). A term ‘represents’ an element \( a \), if for a suitable state \( z : z(t) = a \). In other words we can say that the carrier set \( \in A_s \) can be generated by looking at the values of terms as described above.

If the argument sort of the function symbols \( f_1, f_2, \ldots, f_n \) contain no other sort than \( s \), then \( s \) is often called a sort generated by (the constructors) \( f_1, f_2, \ldots, f_n \). In this case every element of \( A_s \) can be represented as the value of a variable free term \( t \).

The simplest example for a generated sort are natural numbers, which are generated by \( 0 : \rightarrow \text{nat} \) and the successor function \( +1 : \text{nat} \rightarrow \text{nat} \). If we write \( +1 \) postfix (i.e. \( x +1 \) instead of \( +1(x) \)) this means that every natural number is of the form

\[ 0 +1 +1 +1 \ldots +1 \]
That is, we have the axiom
\[
\text{nat generated by } 0, +1
\]
for natural numbers. In this special case two different terms always denote two different numbers (such a datastructure is \textit{freely generated}). This is not always the case: integers can be generated by \(0 : \text{int}\), the successor function and \(+1 : \text{int} \rightarrow \text{int}\) and the predecessor function \(-1 : \text{int} \rightarrow \text{int}\). We have
\[
\text{int generated by } 0, +1, -1
\]
But here the two different terms ‘0’ and ‘0 +1 –1’ represent the same number. Typical examples for data structures in which the function symbols \(f_1, f_2, \ldots f_n\) have other argument sorts than \(s\), are parameterized data structures like lists, arrays or (finite) sets. For the latter
\[
\text{set generated by } \emptyset, \text{insert}
\]
holds, if \(\emptyset : \rightarrow \text{set}\) is the empty set and function \(\text{insert} : \text{elem} \times \text{set} \rightarrow \text{set}\) adds an element to a set. The elements of sort \text{elem} are the parameter which can be chosen freely.

Generation clauses correspond to induction schemes, which allow induction over the structure of a term. The argument is as follows: To show a goal \(\varphi(x)\) for all elements \(x\) of some sort \(s\), it suffices to show that for every \(f_i\) \(\varphi(f_i(x_1, \ldots x_n))\) holds, assuming that \(\varphi(x_i)\) holds for all \(x_i\) which have sort \(s\). In those cases, where \(f_i\) has no arguments of sort \(s\) (e.g. if \(f_i\) is a constant), \(\varphi(f_i(x_1, \ldots x_n))\) must be proven without any preconditions. We will not give a formal definition of the induction rule for arbitrary generated by clauses. Instead we only show the two examples for natural numbers and sets, which should make the principle clear:

\[
\frac{\varphi(0)}{} \quad \frac{\varphi(n)}{} \quad \frac{\varphi(n + 1)}{} \quad \frac{\forall n. \varphi(n)}{}
\]
\[
\frac{\varphi(\emptyset)}{} \quad \frac{\varphi(s)}{} \quad \frac{\varphi(\text{insert}(e, s))}{\forall s. \varphi(s)}
\]
Chapter 2

Predicate Logic Proofs in KIV

2.1 Editing and Loading Theorems

As far as it is necessary for the first exercises we will shortly describe the structure of the software development environment of the KIV system.

2.1.1 Directory structure

The KIV system is a system for the specification and verification of software systems. Every single software system is handled in a project. After the start of the system (the software development environment) you have the possibility to select an existing project (or install a new one) on the project selection level. In the practical course you have to select for every exercise $k$ the corresponding project Exercise($k$). All data of a project is located in a unix directory with the same name. Therefore the project for the first exercise is located in

(Your project directory)/Exercise1

By selecting a project you reach the project level. If you want back to the project selection level you have to click on Edit – Go Back (Go Back is a sub menu of the menu Edit of the graph visualization tool uDraw). Every project consists of specifications and modules. Their dependency forms a directed acyclic graph. This graph is visualized with the graph visualization system uDraw. Rectangles correspond to specifications, rhombs to modules. Specifications are described in detail in later chapters. For the moment it is enough to know that every specification and every module contains a logical theory (i.e. a signature and axioms, e.g. the theory of natural numbers) where you can prove theorems. Every specification itself is located in its own subdirectory of the project. The name of the subdirectory is specs/<name>. So you find the specification “proplogic” in the directory

(Your project directory)/Exercise1/specs/proplogic

The specification text is in the file specification in this directory. You can click on a rectangle (or rhomb) with the left mouse button to view or edit the specification. To work with a specification you select Work on ....

After a short time you will get another window that contains a detailed of the specification’s theorem base. The theorem base contains the axioms of the specification, and user defined theorems (properties, lemmas), and their proofs. The specification window starts with the Summary register tab that shows the names of the axioms and theorems sorted by their state (axiom, proved, unproved, partial, invalid, sig invalid). You can select an entry by clicking on it with the left mouse button (this will highlight the entry). Then you can click the right mouse button to get a popup menu where you can View the theorem, begin a New Proof, etc. When you select the register tab Theorem Base you get another view of the axioms and theorems that also displays
their sequents (or at least part of them). Here you don’t have to select a theorem first with the left mouse button; instead you can just click on a theorem with the right mouse button to get the popup menu. Both tabs have the same functionality – they just present different views of the theorem base.

The most important menu entry is File – Save that saves the theorem base back to the hard disk (KIV has no autosave feature). We recommend to save the theorem base regularly. If you leave the specification with File – Close the theorem base will be saved automatically if necessary. Note that the uDraw window the development graph is still active while the specification window is open, i.e. you can click in the graph to view other specifications etc.

2.1.2 Editing Theorems

You can enter new theorems (sequents, not only formulas) either directly with the menu entry Theorems – Enter New, or through a file. Because theorems are often quite large and it is difficult to enter them without a fault you can use your favorite editor to edit a file. For new theorems the file sequents in a specification directory is used. In our example you have to edit

⟨Your project directory⟩/Exercise1/specs/prologic/sequents

The command Edit – Theorems starts the xemacs editor with that file loaded, and is a comfortable shortcut. After editing and saving the file, KIV has to be informed that it should load new or changed theorem. This is done with the commands Theorems – Load New and Theorems – Load Changed. Both of these commands can also be accessed using the corresponding buttons in the Theorem Base tab.

The syntax of the file is as follows:

1. Two or more semicolons (;;; ) begin a one line comment (as // in Java)
2. (: begins and :) ends a multi line comment (as /* and */ in Java). In contrast to Java these comments can be nested.
3. Parsing stops after a line beginning with ;; END
4. The file may not contain double quotes anywhere.
5. The file begins with an optional keyword variables that allows to declare auxiliary variables for theorems.
6. The keyword lemmas begins the list of theorems.
7. A theorem has the form

⟨theorem name⟩ : ⟨sequent⟩ ;

The trailing semicolon is mandatory. Instead of a sequent you can also just write a formula ϕ, which is read as ⊢ ϕ.

Every theorem can be written in this file. When an already defined theorem is loaded a second time, this definition is ignored. A theorem can be modified with the menu entry Theorems – Load Changed or the Load Changed button in the Theorem Base tab likewise in this file. Other theorems which are not modified are ignored.

To prove a theorem you select either

• Begin to prove an unproved theorem,
• Continue to continue a partial proof,
• Load to load a complete proof, or
• Reprove to start a new proof for a completely or partially proved theorem
in the Proof menu. You can also use the popup menu by right-clicking on a theorem and select New Proof (for Begin or Reprove), Continue Proof, or Load Proof.

The selection of any of these commands opens another window where proofs are done. As before, all other open windows are still active. However, some commands are not possible when you have a current proof. E.g. Theorems – Delete will issue the message ‘You can’t use this command if you have a current proof.’ You finish a proof by closing the proof window with File – Close. If you want to keep the proof you should save the theorem base first with File – Save. (This also saves the current proof.) If you want to discard the proof (because you don’t want to overwrite the old proof for some reason — KIV normally doesn’t keep different versions of proofs) you simply close the window and answer the following question ‘The proof is modified. Update the theorem first?’ with ‘No’. If you answer ‘Yes’ the current proof will be stored in the theorem base and saved the next time the theorem base is saved. (If you change your mind you can also cancel the action.)

### 2.2 Syntax and Special Character Input

The syntax of predicate logic in KIV is essentially the same used e.g. in chapter [3.3](#). The following operator precedences hold:

\[
\{\neg\} \succ \{\land\} \succ \{\lor\} \succ \{\rightarrow\} \succ \{\leftrightarrow\} \succ \{\forall, \exists\}
\]

Quantifiers can contain one or more variables. If more than one variable exists they are separated by commas and after the last variable (and before the quantified formula) a dot follows. Theorems are sequents and not only formulas. The sequent sign is \(\vdash\).

Many of the logical symbols use a special character, and KIV offers even more like \(\cup\) or \(\leq\), or greek letters. A full list is at the end of this section. To insert these characters in the xemacs one has to press F12 and enter the name of the special character (followed by return). The most important names are

| ¬ | ∧ | ∨ | → | ↔ | ∀ | ∃ | \(\vdash\) |
| not | and | or | implies | equiv | all | ex | follows |

It is sufficient to type in a unique prefix, e.g. instead of typing `implies` it is enough to enter `im` and hit return. The available completions for a prefix are shown by hitting the tab key.

Similar support for special characters as in xemacs is also available in all editable text fields of dialog windows. Pressing F12 in such a field will open a dialog, where all the symbols supported by KIV are shown. You can insert a symbol by either double clicking it or entering a unique prefix and hitting ENTER. After that the dialog is closed automatically. You can cancel the dialog by pressing ESC or clicking the close button.

Note also, that you can always paste an arbitrary text selection with your mouse, or paste from your clipboard. Finally, since KIV originally had an ascii-syntax, the most important special characters still have the following ascii-syntax (but it is not recommended that you use it).

| ¬ | ∧ | ∨ | → | ↔ | ∀ | ∃ | \(\vdash\) |
| not | and | or | \(\rightarrow\) | \(\leftrightarrow\) | all | ex | \(\mid\vdash\) |

When entering terms and predicates and formulas some common pitfalls should be avoided:

1. In KIV nearly arbitrary identifiers are allowed for symbols, e.g. 0, +, ++, <, list<, |, \(\rightarrow\)3, \(\Rightarrow\), \(\exists\), \(\mid\vdash\) etc. The disallowed characters are brackets, space, comma, semicolon, backquote, and double quotation mark. Colon, star, and the dollar sign are treated specially. Therefore:

   There must be spaces around identifiers unless the previous or next character is a bracket, comma, or semicolon!
"x = y" is not the equation "x = y", but a three character identifier!

2. Variables, function and predicate symbols can only be entered when they are already defined in the signature of the specification.
   In the specification barbie the variables x and y of sort person are defined. Only these two variables can be used to formulate the propositions.

3. Functions and predicates can be written in the usual form $f(t_1, \ldots, t_n)$, but there are also infix, prefix, and postfix operators. E.g. + is an infix operator so you must write “m + n” instead of “+(m,n)” +1 is a postfix operator and is placed behind its argument: “(m + n) +1”. A prefix operator precedes its single argument without brackets. An infix, prefix, or postfix operator can’t be written in another manner, i.e. it is not possible to write “+(m, n)” or “+1(n)”.

Postfix operators have a higher precedence than prefix operators. Infix operators have lowest precedence, so the expression “m + n +1” is equivalent to “m + (n +1)” and not to “(m + n) +1! Furthermore, for infix operators a precedence can be specified, e.g. $\ast$ is defined to have higher precedence than +. If you are not sure about precedences you should place brackets around the expression.

Infix operators are specified in the signature definition of the specification with two dots around the symbol. E.g. in the specification nat

   

```
. + . : nat × nat → nat;
```

Postfix operations have got a dot before the symbol in the definition (the dots mark the positions of the parameters):

```
. +1 : nat → nat;
```

4. The dot is allowed at the beginning of a symbol, but nowhere else. It is often used to define postfix operations. The leading space before a symbol starting with a dot can be dropped. As an example in “x.first” the postfix operation “.first” is applied on x, and the expression needs no space in the middle. A single dot is not a symbol, but used only in quantifiers. There, no space is needed before the dot, but one space is needed after it to separate the next symbol.

5. Special characters in Xemacs normally have the same name as the corresponding \LaTeX symbol, so ⊙ is odot, ⊕ is called oplus and so on. In ascii syntax these symbols can be entered as the xemacs name with a leading backslash, e.g. \odot for ⊙, \oplus for ⊕, etc.

A parser error occurs if the input is not correct. In the specification nat e.g. the variable x is not defined. So the input “n + x” results in the parser error

```
Parser: parse error in "n + ?? x".
```

The parser displays two question marks before the next token that can’t be parsed. The text up to this point was parsed successfully (though perhaps not in the manner you expected). The input “n + = n” results in

```
Parser: parse error in "n + ?? = n".
```

If the beginning of the input is already faulty (e.g. the input “x” in the specification nat) you get

```
Parser: parse error at the beginning of "x".
```

If something is missing at the end (e.g. the semicolon after a sequent in the sequents file, see 2.1.2) you get

```
Parser: parse error at the end of "lemmas lem-01 : ⊢ 0 + m = m".
```
2.3 Proofs in KIV

The proof window (the window has the title “KIV - Specification Strategy”) is used for proofs in the sequent calculus. In the large frame on the right you always find the currently selected premise of the proof tree, the actual “goal” you are just now working on. On the left side all applicable rules are shown. Another window with the title “current proof” shows the current proof tree.

Selecting basic rules

In the first exercise the basic rule set of the sequent calculus is used instead of the optimized rule set that is normally (and in later exercises) used. For your convenience this switching of rule sets has already been done (using a configuration file). Switching rule sets can also be done manually (and we will need this feature later on) by selecting the menu entry Control – Options. After that a window will appear – with a rather longish list of options. For the first exercise only the option “Use basic rules” is of interest, which is already selected. Clicking on this option will deselect it or select it again. Clicking the “Ok” button will activate your selection of options. For the first exercise the “Use basic rules” option should be activated.

Short description of the menu commands

In the following section all menu commands of the proof level which are relevant for first order proofs are described. Some important commands are also present in the icon bar.

<table>
<thead>
<tr>
<th>name</th>
<th>symbol</th>
<th>name</th>
<th>symbol</th>
<th>name</th>
<th>symbol</th>
<th>name</th>
<th>symbol</th>
<th>name</th>
<th>symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>follows</td>
<td>⊢</td>
<td>and</td>
<td>∧</td>
<td>integral</td>
<td>∫</td>
<td>phi</td>
<td>ϕ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unequal</td>
<td>≠</td>
<td>beta</td>
<td>β</td>
<td>intersection</td>
<td>∩</td>
<td>pi</td>
<td>π</td>
<td></td>
<td></td>
</tr>
<tr>
<td>not</td>
<td>¬</td>
<td>bottom</td>
<td>⊥</td>
<td>iota</td>
<td>ι</td>
<td>psi</td>
<td>ψ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>∧</td>
<td>box</td>
<td>□</td>
<td>kappa</td>
<td>κ</td>
<td>rcel</td>
<td>⊥</td>
<td></td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>∨</td>
<td>chi</td>
<td>χ</td>
<td>lambda</td>
<td>λ</td>
<td>revsemimp</td>
<td>⇐</td>
<td></td>
<td></td>
</tr>
<tr>
<td>implies</td>
<td>→</td>
<td>circle</td>
<td>⊙</td>
<td>lceil</td>
<td>[</td>
<td>rfloor</td>
<td>]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>equivalent</td>
<td>↔</td>
<td>congruent</td>
<td>≡</td>
<td>le</td>
<td>[</td>
<td>rho</td>
<td>ρ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>exists</td>
<td>∃</td>
<td>delta</td>
<td>δ</td>
<td>lessorequal</td>
<td>≤</td>
<td>rquine</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>all</td>
<td>∀</td>
<td>diamond</td>
<td>⊘</td>
<td>llfloor</td>
<td></td>
<td></td>
<td>semequiv</td>
<td>⇐</td>
<td></td>
</tr>
<tr>
<td>emptyset</td>
<td>⊕</td>
<td>}</td>
<td></td>
<td></td>
<td>lower</td>
<td>]</td>
<td>semimp</td>
<td>⇒</td>
<td></td>
</tr>
<tr>
<td>in</td>
<td>∈</td>
<td>{</td>
<td></td>
<td></td>
<td>lquine</td>
<td></td>
<td></td>
<td>sigma</td>
<td>σ</td>
</tr>
<tr>
<td>union</td>
<td>∪</td>
<td>downarrow</td>
<td>↓</td>
<td>lsem</td>
<td></td>
<td></td>
<td>sqgreater</td>
<td></td>
<td></td>
</tr>
<tr>
<td>intersection</td>
<td>∩</td>
<td>emptyset</td>
<td>◯</td>
<td>llfloor</td>
<td></td>
<td></td>
<td>sqless</td>
<td></td>
<td></td>
</tr>
<tr>
<td>times</td>
<td>×</td>
<td>epsilon</td>
<td>ε</td>
<td>lfloor</td>
<td></td>
<td></td>
<td>subset</td>
<td>⊆</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>⟨</td>
<td>equivalent</td>
<td>⇐</td>
<td>lquine</td>
<td></td>
<td></td>
<td>subseteq</td>
<td>⊆</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>⟩</td>
<td>eta</td>
<td>η</td>
<td>lub</td>
<td></td>
<td></td>
<td>supset</td>
<td>⊇</td>
<td></td>
</tr>
<tr>
<td>Delta</td>
<td>Δ</td>
<td>exists</td>
<td>⊆</td>
<td>models</td>
<td></td>
<td></td>
<td>supseteq</td>
<td>⊇</td>
<td></td>
</tr>
<tr>
<td>Gamma</td>
<td>Γ</td>
<td>forall</td>
<td>∀</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>tau</td>
<td>τ</td>
</tr>
<tr>
<td>Lambda</td>
<td>Λ</td>
<td>follows</td>
<td>⊣</td>
<td>ni</td>
<td></td>
<td></td>
<td>theta</td>
<td>θ</td>
<td></td>
</tr>
<tr>
<td>Omega</td>
<td>Ω</td>
<td>gamma</td>
<td>γ</td>
<td>not</td>
<td></td>
<td></td>
<td>times</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>Phi</td>
<td>Φ</td>
<td>ge</td>
<td>≥</td>
<td>mu</td>
<td></td>
<td></td>
<td>top</td>
<td>⊤</td>
<td></td>
</tr>
<tr>
<td>Pi</td>
<td>Π</td>
<td>ghd</td>
<td>⊏</td>
<td>odiv</td>
<td></td>
<td></td>
<td>unequal</td>
<td>≠</td>
<td></td>
</tr>
<tr>
<td>Psi</td>
<td>Ψ</td>
<td>greaterorequal</td>
<td>≥</td>
<td>odot</td>
<td></td>
<td></td>
<td>union</td>
<td>⊆</td>
<td></td>
</tr>
<tr>
<td>Sigma</td>
<td>Σ</td>
<td>grgr</td>
<td>⊑</td>
<td>omega</td>
<td></td>
<td></td>
<td>uparrow</td>
<td>↑</td>
<td></td>
</tr>
<tr>
<td>Theta</td>
<td>Θ</td>
<td>higher</td>
<td>⊈</td>
<td>ominus</td>
<td></td>
<td></td>
<td>uppsilon</td>
<td>ν</td>
<td></td>
</tr>
<tr>
<td>Xi</td>
<td>Ξ</td>
<td>implies</td>
<td>→</td>
<td>oplus</td>
<td></td>
<td></td>
<td>xi</td>
<td>ξ</td>
<td></td>
</tr>
<tr>
<td>all</td>
<td>∀</td>
<td>in</td>
<td>∈</td>
<td>or</td>
<td></td>
<td></td>
<td>zeta</td>
<td>ζ</td>
<td></td>
</tr>
<tr>
<td>alpha</td>
<td>α</td>
<td>infinity</td>
<td>∞</td>
<td>otimes</td>
<td>⊗</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>symbol</th>
<th>name</th>
<th>symbol</th>
<th>name</th>
<th>symbol</th>
<th>name</th>
<th>symbol</th>
<th>name</th>
<th>symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>symbol</th>
<th>name</th>
<th>symbol</th>
<th>name</th>
<th>symbol</th>
<th>name</th>
<th>symbol</th>
<th>name</th>
<th>symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>follows</td>
<td>⊢</td>
<td>and</td>
<td>∧</td>
<td>integral</td>
<td>∫</td>
<td>phi</td>
<td>ϕ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unequal</td>
<td>≠</td>
<td>beta</td>
<td>β</td>
<td>intersection</td>
<td>∩</td>
<td>pi</td>
<td>π</td>
<td></td>
<td></td>
</tr>
<tr>
<td>not</td>
<td>¬</td>
<td>bottom</td>
<td>⊥</td>
<td>iota</td>
<td>ι</td>
<td>psi</td>
<td>ψ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>∧</td>
<td>box</td>
<td>□</td>
<td>kappa</td>
<td>κ</td>
<td>rcel</td>
<td>⊥</td>
<td></td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>∨</td>
<td>chi</td>
<td>χ</td>
<td>lambda</td>
<td>λ</td>
<td>revsemimp</td>
<td>⇐</td>
<td></td>
<td></td>
</tr>
<tr>
<td>implies</td>
<td>→</td>
<td>circle</td>
<td>⊙</td>
<td>lceil</td>
<td>[</td>
<td>rfloor</td>
<td>]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>equivalent</td>
<td>↔</td>
<td>congruent</td>
<td>≡</td>
<td>le</td>
<td>[</td>
<td>rho</td>
<td>ρ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>exists</td>
<td>∃</td>
<td>delta</td>
<td>δ</td>
<td>lessorequal</td>
<td>≤</td>
<td>rquine</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>all</td>
<td>∀</td>
<td>diamond</td>
<td>⊘</td>
<td>llfloor</td>
<td></td>
<td></td>
<td>semequiv</td>
<td>⇐</td>
<td></td>
</tr>
<tr>
<td>emptyset</td>
<td>⊕</td>
<td>}</td>
<td></td>
<td></td>
<td>lower</td>
<td>]</td>
<td>semimp</td>
<td>⇒</td>
<td></td>
</tr>
<tr>
<td>in</td>
<td>∈</td>
<td>{</td>
<td></td>
<td></td>
<td>lquine</td>
<td></td>
<td></td>
<td>sigma</td>
<td>σ</td>
</tr>
<tr>
<td>union</td>
<td>∪</td>
<td>downarrow</td>
<td>↓</td>
<td>lsem</td>
<td></td>
<td></td>
<td>sqgreater</td>
<td></td>
<td></td>
</tr>
<tr>
<td>intersection</td>
<td>∩</td>
<td>emptyset</td>
<td>◯</td>
<td>llfloor</td>
<td></td>
<td></td>
<td>sqless</td>
<td></td>
<td></td>
</tr>
<tr>
<td>times</td>
<td>×</td>
<td>epsilon</td>
<td>ε</td>
<td>lub</td>
<td></td>
<td></td>
<td>subset</td>
<td>⊆</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>⟨</td>
<td>equivalent</td>
<td>⇐</td>
<td>lquine</td>
<td></td>
<td></td>
<td>subseteq</td>
<td>⊆</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>⟩</td>
<td>eta</td>
<td>η</td>
<td>models</td>
<td></td>
<td></td>
<td>supset</td>
<td>⊇</td>
<td></td>
</tr>
<tr>
<td>Delta</td>
<td>Δ</td>
<td>exists</td>
<td>⊆</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>tau</td>
<td>τ</td>
</tr>
<tr>
<td>Gamma</td>
<td>Γ</td>
<td>forall</td>
<td>∀</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>theta</td>
<td>θ</td>
</tr>
<tr>
<td>Lambda</td>
<td>Λ</td>
<td>follows</td>
<td>⊣</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>times</td>
<td>×</td>
</tr>
<tr>
<td>Omega</td>
<td>Ω</td>
<td>gamma</td>
<td>γ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>top</td>
<td>⊤</td>
</tr>
<tr>
<td>Phi</td>
<td>Φ</td>
<td>ge</td>
<td>≥</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>unequal</td>
<td>≠</td>
</tr>
<tr>
<td>Pi</td>
<td>Π</td>
<td>ghd</td>
<td>⊏</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>union</td>
<td>⊆</td>
</tr>
<tr>
<td>Psi</td>
<td>Ψ</td>
<td>greaterorequal</td>
<td>≥</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>uparrow</td>
<td>↑</td>
</tr>
<tr>
<td>Sigma</td>
<td>Σ</td>
<td>grgr</td>
<td>⊑</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>uppsilon</td>
<td>ν</td>
</tr>
<tr>
<td>Theta</td>
<td>Θ</td>
<td>higher</td>
<td>⊈</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xi</td>
<td>ξ</td>
</tr>
<tr>
<td>Xi</td>
<td>Ξ</td>
<td>implies</td>
<td>→</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>zeta</td>
<td>ζ</td>
</tr>
<tr>
<td>all</td>
<td>∀</td>
<td>in</td>
<td>∈</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>alpha</td>
<td>α</td>
<td>infinity</td>
<td>∞</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
• Menu File
  – Save: save the theorem base and the current proof (Icon: diskette)
  – Close: end the current proof (Icon: closing door)

• Menu View
  – Specification: view the used specification (the specification can also be viewed under the specification tab)

• Menu Proof
  – Begin/Continue: choose a new theorem to prove

• Menu Goal
  – Open Goals: display informations about open premises
  – Switch Goal: continue working at another goal (Icon: Green Arrow left/right to switch to the previous/next goal).
  – Show Tree: display the current proof tree. This command always is very useful to get informations about the current proof. (Icon: Green Three-Node Tree).
  – Prune Tree: prune a branch in the current proof tree. This makes sense if there is an error in the proof. The position where the tree should be pruned is chosen by clicking on the node and pressing p (or choosing Operations – Prune Tree in the tree window). If the current proof tree is displayed it’s also possible to click on the node and use p or Operations – Prune Tree (without using a menu command). Note that this command doesn’t change the chosen heuristics. It is therefore possible that the new goal is treated by the heuristics at once. In this case all heuristics should be deactivated before pruning the tree by clicking “Use Heuristics” in the bottom left corner of the proof window.

• Menu Control
  – Heuristics: choose heuristics (see below; Icon: Terminal with On/Off).
  – Backtrack: got back to the last backtrack point. The system generates always a backtrack point if rules are applied or the systems behavior changes because of an interaction. Nevertheless only a certain number of backtrack points are stored. (Icon: yellow, curved arrow)
  – Options: To select the option Use basic rules. A window appears with a rather longish list of options. For the first exercise only the option “Use basic rules” is of interest. Clicking on this option will mark it as selected. By clicking the “Ok” button the selection of options is ended.
    Options are valid as long as you are working on a specification or doing proofs in this specification, i.e. the option remains in charge until you go back to the project level.

Viewing proof trees
Proof trees can be displayed graphically in the tree windows. A brief description how such a tree can be interpreted follows.

Colors
A proof tree appears in different colors which serves for an easier orientation.

  • Color of branches: closed branches are displayed in green (i.e. there exist no more open premises resp. all open premises are axioms or theorems). Red branches lead to open premises which have to be proven.
CHAPTER 2. PREDICATE LOGIC PROOFS IN KIV

• Color of nodes:
  – Colors of inner nodes: The color of the inner nodes marks the kind of rule that was used.
    * black: propositional rules not introducing case distinctions, i.e. conjunction left, negation left, disjunction right, implication right, negation right, and other “unimportant” rules like simplifier, elimination etc.
    * red: induction rules
    * blue: cut formula, disjunction left, implication left, equivalence left, conjunction right, equivalence right, case distinction, program calls
    * brown: weakening
    * green: quantifier instantiation
    * violet: rules for loops and whiles
  – Color for leave nodes: The color of the leave nodes sign how the corresponding branch was closed (if at all).
    * green: axiom or theorem of the specification.
    * blue: axiom or theorem of a subspecification.
    * red: open premise.

• filled vs. non-filled nodes: Premises are always drawn as non-filled nodes. Inner nodes of a tree are non-filled if the applied rule was chosen by the user. If a rule was applied automatically by an heuristic (see below) the node is filled.

• enumeration of premises: The open premises of a tree are enumerated. If the proof uses some theorems as lemmas their names are displayed (lemmas are not treated as open premises).

Clicking a proof tree

Clicking on a node with the left mouse button marks the node and displays the corresponding sequent in a sequent window. Clicking with the middle button displays the sequent in a new sequent window without marking that node. Clicking a further node with the middle mouse button opens a second, new sequent window and so on (useful if you want to see more than one sequent). Clicking a node with the right button marks it without displaying its sequent. By left-(or right-)clicking a part of the tree window where no node is displayed the marking is reset.

Operations

The menu operations contains a number of commands. But only a few of them are interesting:

Prune Tree After clicking on a node with the left or right mouse button this menu item causes that the command Prune Tree is executed (see above). The tree is pruned at the marked point.

The key p is an abbreviation of this menu item.

Switch Goal After the choice of an open premise with the left mouse key the proof is continued at this premise.

The key g is a shortcut for this item.

Quit The tree window (and all its sequent windows) are closed. (Shortcut: key q)

Note that these menu items are disabled when they’re not applicable. E.g. Switch Goal needs that both a node is selected in the tree and that this node is an open premise in order to be applicable and thus enabled.

\[1\] The marked node will be shown in a red rectangle.

\[2\] The window will be titled Goal <nr>. For reference, in the proof tree window <nr> will be written on the right of the node.
Sequent Windows

The size of sequent windows is variable. The system tries to increase the size of the window until the whole output (generated by the pretty printer) fits. If that is not possible scrollbars are added to the window.

Above the sequent in some cases a text is displayed. This text is extracted from the tree-comment of the (sub-)tree whose conclusion is the printed sequent. It normally contains the name (and in some case also the arguments) of the used rule. In case of open premises nothing is displayed. To close the sequent window press key q in the window (or click the Close button).

2.4 Rule set for predicate logic

The rule set for predicate logic can be found in the next section (2.5). Here we describe the usage of three selected rules.

2.4.1 all left

The quantifier rules all left and exists right require (or allow) an instantiation that must be entered interactively. We describe the rest for all left, but everything also works for exists right.

\[ \frac{\phi^\tau, \forall x. \varphi, \Gamma \vdash \Delta}{\forall x. \varphi, \Gamma \vdash \Delta} \hspace{1cm} \text{(all left)} \]

\( \phi^\tau \) is the substitution of \( x \) by \( \tau \) in \( \varphi \), \( \tau \) may be an arbitrary term. The rule is also applicable for a list of quantified variables \( \forall x_1, \ldots, x_n. \varphi \). In this case a list of terms \( \tau_1, \ldots, \tau_n \) is used and a parallel substitution takes place.

You can select the rule from the rule list. If there is more than one formula with a universal quantifier in the antecedent, the first step is the selection of formula you want to apply the rule on. Just select the formula and click ‘Okay’. Instead of selecting the rule from the rule list you can also select the rule and the formula in one step with the mouse. Just click with the right mouse button anywhere in the formula.

The next step is the input of the instantiation. A window appears that displays the formula, the list of variables to instantiate, a list of suggestions (or no suggestion), a text field for your input, and three buttons. Now you have three possibilities. You can select a substitution from the list of suggestions. The substitution appears in the text field. Or you can enter a substitution yourself. For every variable to instantiate you enter one term. Several terms are separated by comma. E.g., if the variables to substitute are \([x, y]\), you must enter something like \(3, x + y\). (You can also type \[3, x + y\].) This means that \( x \) will be substituted by \( 3 \), and \( y \) by \( x + y \) in parallel! (I.e. the second substitution is not equivalent to \( 3 + y \).) The third possibility is to choose a previously typed substitution by clicking on the down arrow at the right of the text field. No matter how you selected the substitution, you can still edit it in the text field. This is very convenient if a suggestion is slightly wrong. Just select and edit it.

After you are satisfied with your substitution, one more decision remains: You can discard or keep the quantified formula. Discarding it is useful if you are sure that you don’t need it anymore, because it makes the sequent smaller, easier to read, and you are not tempted to use the rule again with useless instances. However, the goal may become unprovable if your current instance is not correct or you must instantiate the quantifier more than once. It’s your choice! If you want to keep the quantified formula select the button ‘Okay (Keep)’, otherwise select ‘Okay (Discard)’.

If your substitution contains a variable that is not free in the sequent, a confirmation window appears whether you really want to use that substitution. Normally, a new variable (not occurring free in the sequent) indicates a typing error, because such a substitution is normally useless. However, there are cases (that occur in the exercises!) where the substitution is really irrelevant. Confirm the window this ‘Yes’ or ‘No’, and the rule is applied.
2.4.2 insert equation

Equations can be inserted with the rule insert equation

\[
\frac{\sigma = \tau, \Gamma \vdash \Delta}{\sigma = \tau, \Gamma \vdash \Delta} \quad \text{(insert equation)}
\]

In this rule \( \sigma = \tau \) is an arbitrary equation of the antecedent. The system displays all possible equations together with their symmetric counterpart \( \tau = \sigma \). Since predicates in the antecedent can be read as predicate = true (and predicates in the succedent as predicate = false) they are also listed. Beware: every equation and predicate is offered for insert equation even if there is no useful usage for the equation!

Now you can select one equation to insert. There are two buttons: the left is labeled Okay (keep), the right Okay (discard). ‘keep’ means that the equation remains in the new goal (as shown above), ‘discard’ means that the equation will be discarded after insertion (with weakening). Beware: it is your decision! If you discard the equation the goal may become unprovable. (This happens if the equation is false, e.g. \( 0 = 1 \), or if the right hand side contains variables that still occur in the sequent after insert equation.)

After selecting the equation to insert (and whether to keep or to discard it) the positions where the equation can be inserted are displayed (marked with ##. . . ). It is possible to substitute all occurrences of \( \sigma \) by \( \tau \) (the normal case) or just one (or several occurrences). You will not get this question if no or only one possibility exists.

The rule can also be applied in a context sensitive manner. If you click with the right mouse button anywhere on the left hand side of an equation, a window with the text insert equation (left to right) appears. This will insert the equation left to right everywhere, i.e. all occurrences of the right side are substituted by the left side, and the equation is discarded. If you click on the right hand side the left side is replaced by the right side, again everywhere, and again the equation is discarded. If you want to keep the equation you have to use the rule from the rule list.

2.4.3 insert lemma

To use axioms or theorems the following rule exists:

\[
\frac{\Gamma' \vdash \Delta'}{\Gamma \vdash \Theta(\Gamma'), \Delta \vdash \Theta(\Delta'), \Gamma \vdash \Delta} \quad \text{(insert lemma)}
\]

- \( \Gamma' \vdash \Delta' \) is the lemma (theorem)
- \( \Theta \) is a parallel substitution for the free variables of the lemma

In order to apply a theorem \( \Gamma' \vdash \Delta' \) it is necessary to enter a substitution \( \Theta \) for the free variables of the theorem to obtain the instances for the actual goal. The system shows the free variables of the sequent and you have to enter a list of terms \( \text{term}_1, \text{term}_2, \ldots, \text{term}_n \) for every variable separated by comma. The list of terms may be optionally enclosed in square brackets. The selection of the substitution is identical to the quantifier instantiation all left. However, there is only one ‘Okay’ button because you can’t ‘keep’ the lemma. If you need it more than once you have to use the rule several times.

The rule adds three new premises to the proof tree. The first one is the theorem itself. This premise stays open and is managed by the correctness management which takes care that no cyclical proof dependencies occur. (For example you prove the theorem A with the lemma B and the lemma B with the theorem A). For the second premise you have to show that the (instances of the) precondition for the lemma holds. This is the conjunction of the formulas of the antecedent of the lemma, written as \( \Theta \Gamma' \). If the lemma has no conditions then \( \Theta \Gamma' \) is true and the statement is an axiom. In the last premise the results (i.e. the disjunction of the formulas of the succedent, often only one formula) are added to the previous goal.
A similar rule is *insert axiom*. An axiom or theorem (with all quantifiers added) is added to the antecedent of the goal. If you instantiate the quantifiers with *all left*, discard the formula, and make a case distinction (provided the axiom is $\varphi \rightarrow \psi$), you obtain the same premises as with *insert lemma*.

2.5 Basic rules

The following notation is used for the rules: $\varphi, \psi$ denote arbitrary formulas, $\Gamma, \Delta$ stand for arbitrary lists of formulas. $\sigma, \tau$ denote terms. The rules are written in a little bit simplified notation. Even though we write $\varphi \land \psi$, $\Gamma \vdash \Delta$ the rule *conjunction left* is applicable on any conjunction in the antecedent, not only on the first formula of the antecedent. The correct notation would be $\Gamma_1, \varphi \land \psi, \Gamma_2 \vdash \Delta$.

2.5.1 Axioms

$$\Gamma \vdash \varphi, \Delta \quad \text{(axiom)}$$

$$\varphi, \Gamma \vdash \varphi, \Delta \quad \text{(false left)}$$

$$\Gamma \vdash \psi, \Delta \quad \text{(true right)}$$

2.5.2 Propositional and equational rules

$$\Gamma \vdash \varphi, \psi, \Delta \quad \text{(disjunction left)}$$

$$\Gamma \vdash \varphi \rightarrow \psi, \Delta \quad \text{(implication left)}$$

$$\Gamma \vdash \varphi, \psi, \Delta \quad \text{(equivalence left)}$$

$$\Gamma \vdash \varphi, \Delta \quad \text{(weakening)}$$

$$\Gamma \vdash \psi, \Delta \quad \text{(cut formula)}$$

2.5.3 Quantifiers

Note: $\varphi^x_\tau$ is the substitution of $x$ by $\tau$ in $\varphi$.

- $\forall x. \varphi, \Gamma \vdash \Delta$ (all left)
- $\Gamma \vdash \exists x. \varphi, \Delta$ (exists right)

$\tau$ may be an arbitrary term.

The rule is also applicable for a list of quantified variables $\forall x_1, \ldots, x_n. \varphi$. In this case a list of terms $\tau_1, \ldots, \tau_n$ is used and a parallel substitution takes place.

The quantified formula can optionally be discarded.
CHAPTER 2. PREDICATE LOGIC PROOFS IN KIV

2.5.4 Theory rules

• \( \vdash \varphi(c) \quad \varphi(x) \vdash \varphi(f(x)) \) (structural induction)

\( \varphi = \forall x' \Gamma \rightarrow \Delta \) with \( x' = \text{Free}(\Gamma \rightarrow \Delta) \setminus x \)

The specification contains a generation principle sort(\( x \)) generated by \( c, f \)

• \( \vdash \forall x. \varphi \quad \forall x', \varphi, \Gamma \vdash \Delta \) (insert axiom/insert spec-axiom)

\( \varphi \) is an axiom, \( x \) the free variables of \( \varphi \).

insert axiom applies a lemma from the current specification, insert spec-axiom from a sub-
specification.

• \( \Gamma' \vdash \Delta' \) \( \Gamma \vdash \Theta(\Gamma'), \Delta \quad \Theta(\Delta'), \Gamma \vdash \Delta \) (insert lemma/insert spec-lemma)

\( \Gamma' \vdash \Delta' \) is the lemma (theorem)

\( \Theta \) is a parallel substitution for the free variables of the lemma.

insert lemma applies a lemma from the current specification, insert spec-lemma from a sub-
specification.

• \( \vdash \varphi \rightarrow \sigma = \tau \) \( \Gamma \vdash \Theta(\varphi), \Delta \quad \Theta(\Delta), \Gamma \vdash \Delta \) (insert rewrite lemma)

\( \vdash \varphi \rightarrow \sigma = \tau \) is the rewrite theorem.

\( \Theta \) is a parallel substitution for the free variables of the theorem. (Computed automatically.)

2.6 Heuristics for basic rules

If you have worked for a while with the basic rules of the sequent calculus you will see that a
lot of steps always repeat. For example, you will always apply rules for propositional connectives
that do not introduce case distinctions. Such regular patterns of the proof can be detected by the
system and it can apply the corresponding rules automatically. Heuristics take the decision from
the user what has to be done next. But the user has to decide which heuristics to use and which
to omit. They can be switched on/off at any position in the proof. If a heuristic is not applicable
on the actual sequent the system tries to apply the next heuristic. If there are no more applicable
heuristics the system stops the automatic proof attempt and asks the user for an interaction. The
system displays all rules that are applicable on the current goal.

The heuristics are chosen by the menu entry Control – Heuristics. If you are working with
the basic rules of the sequent calculus a two column window appears where the left column contains
all heuristics for basic rules, and the right column the currently selected, active heuristics. You
can add heuristics by clicking on them in the left window. The order of the application of the
heuristics corresponds to the order of their selection. By clicking on the OK button the heuristics
are applied on the current goal. You can turn the heuristics on or off by clicking ‘Use Heuristics’
in the lower left bottom of the proof window. If no heuristics are selected, and ‘Use Heuristics’ is
deselected, clicking on the field is a shortcut to the menu entry Control – Heuristics.
Heuristics are only heuristics: They may do the wrong thing, or something useless. And the order of the heuristics can have considerable impact on the size of the proof. (E.g. if you do case distinctions too early.) The following heuristics are for the basic rules:

- **axiom**: This heuristic searches for a possibility to apply the axiom rules \textit{axiom}, \textit{false left}, \textit{true right}, and \textit{reflexivity right}.

  This heuristic closes the goal, i.e. it can’t do anything wrong or useless, and should be always used as the first heuristic.

- **prop simplification**: applies the propositional rules with one premise \textit{negation left}, \textit{negation right}, \textit{conjunction left}, \textit{disjunction right}, and \textit{implication right}.

  Can’t do something wrong or useless.

- **prop split**: applies the propositional rules with two premises \textit{disjunction left}, \textit{conjunction right}, \textit{implication left}, \textit{equivalence left}, and \textit{equivalence right}.

  Can’t do something wrong, but may do case distinctions too early.

- **smart basic case distinction**: tries to apply propositional rules with two premises where one premise can be immediately closed, so that no ‘real’ case distinction is introduced.

  Can’t do something wrong or useless.

- **insert equation**: inserts equation with the rule \textit{insert equation} without discarding the equation.

  May be useless, but not wrong.

- **Quantifier closing**: applies the rules \textit{all left} and \textit{exists right} if it can find an instantiation that will close the goal.

  Can’t do something wrong or useless.

- **discard quantifier**: applies the rules \textit{exists left} and \textit{all right}.

  Can’t do something wrong or useless.

- **Quantifier**: applies the rules \textit{all left} and \textit{exists right} if it finds (or rather guesses) an instantiation that may be useful for the proof. This guessing is not perfect at all (and can not be due to the undecidability of predicate logic), and therefore this heuristic is “dangerous” in multiple regard: It can happen that this heuristic tries unnecessary instances for a quantifier, thereby producing a large proof tree. It can also happen that this heuristic does not find the correct instance and you have to insert it yourself (maybe several times if unnecessary instances arose). In the worst case this heuristic tries again and again an incorrect instance an the system does not stop at all. In this case press the \textit{Stop} button. This will enforce a stop. But for all that this heuristic often finds the correct instance and is therefore very useful.

- **batch mode**: This ‘heuristic’ just switches to the next goal. It should be only used as the last heuristic. Its effect is that all branches of the proof are treated by the heuristics as much as possible.
2.7 Exercise 1

For every exercise (except the last one) the basic rules have to be used, i.e. the option *Use basic rules* (by selecting the menu command `Control - Options`) must be switched on. This should be the case by default.

**Exercise 1.1 Propositional logic**

Then prove without heuristics the three propositional axioms of the Hilbert calculus:

Hilbert-1: \( \vdash \varphi \rightarrow (\psi \rightarrow \varphi) \)

Hilbert-2: \( \vdash (\varphi \rightarrow \psi) \rightarrow (\neg \psi \rightarrow \neg \varphi) \)

Hilbert-3: \( \vdash ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi)) \rightarrow (\varphi \rightarrow (\psi \rightarrow \chi)) \)

**Exercise 1.2 (Example) Propositional logic**

Consider the following logical puzzle:

The oracle says:

The left and the right path lead to Delphi if the middle path leads somewhere else.

If at most one of the left and middle path lead to the oracle, then the right path is wrong.

However, if the right or left path leads to Delphi, the middle path definitely does not lead there.

Which of the three paths leads to Delphi?

To solve this puzzle we can use three propositional constants `left`, `right`, and `middle`. Their intuitive meaning is: If the constant is true the path leads to Delphi, otherwise not. For example, if `left` is true the left path leads to Delphi. We can now formalize the three sentences (or known facts):

\[
\neg \text{middle} \rightarrow \text{left} \land \text{right} \\
\neg (\text{left} \land \text{middle}) \rightarrow \neg \text{right} \\
\text{left} \lor \text{right} \rightarrow \neg \text{middle}
\]

Together we have

\[
(\text{facts}) = (\neg \text{middle} \rightarrow \text{left} \land \text{right}) \\
\quad \land (\neg (\text{left} \land \text{middle}) \rightarrow \neg \text{right}) \\
\quad \land (\text{left} \lor \text{right} \rightarrow \neg \text{middle})
\]

(Note the brackets!) Now we have to find a solution. Here it is important to read exactly what is requested. Let’s assume we found out (by hard thinking) that the middle path leads to Delphi. This means a possible solution is `middle`. On the other hand, we may have found out that the middle path leads to Delphi and the other two paths do not. This means another (more complete) solution is `middle \land \neg \text{left} \land \neg \text{right}`. Now we can prove two things:

1. The solution fulfills the facts, i.e. the solution is indeed a solution. (This also proves that the facts are not contradictory!)

Prove \( \langle \text{solution} \rangle \vdash \langle \text{facts} \rangle \):

\[
\begin{align*}
\text{middle} \land \neg \text{left} \land \neg \text{right} & \\
\vdash (\neg \text{middle} \rightarrow \text{left} \land \text{right}) \\
& \land (\neg (\text{left} \land \text{middle}) \rightarrow \neg \text{right}) \\
& \land (\text{left} \lor \text{right} \rightarrow \neg \text{middle})
\end{align*}
\]

(Note the brackets because of the priorities of \( \neg, \land, \rightarrow \)) This only works for complete solutions, i.e. if a value is given to all constants.
If we want to check a partial solution, we can prove
\neg \langle \text{partial solution} \rangle \vdash \neg \langle \text{facts} \rangle :
\neg \text{middle}
\vdash \langle \neg \text{middle} \rightarrow \text{left} \land \text{right} \rangle 
\land \langle \neg (\text{left} \land \neg \text{middle}) \rightarrow \neg \text{right} \rangle 
\land \langle \text{left} \lor \neg \text{right} \rightarrow \neg \text{middle} \rangle

2. The solution follows from the facts. (This means the solution is unique.)
Prove \langle \text{facts} \rangle \vdash \langle \text{solution} \rangle :
\neg \text{middle} \rightarrow \text{left} \land \text{right},
\neg (\text{left} \land \text{middle}) \rightarrow \neg \text{right},
\text{left} \lor \text{right} \rightarrow \neg \text{middle}
\vdash \text{middle} \land \neg \text{left} \land \neg \text{right}

It is somewhat unsatisfactory that we have to know the solution before we can prove anything. However, we can also use the system to find a solution. We begin ‘proving’ with only the facts, i.e. \langle \text{facts} \rangle \vdash \langle \text{empty succedent} \rangle :
\neg \text{middle} \rightarrow \text{left} \land \text{right}, \neg (\text{left} \land \text{middle}) \rightarrow \neg \text{right}, \text{left} \lor \text{right} \rightarrow \neg \text{middle}
\vdash \text{middle} \land \neg \text{left} \land \neg \text{right}

Using the heuristics we get a proof tree that contains one open premise:
\langle \text{middle} \land \neg \text{left} \land \neg \text{right} \rangle
And this is the solution! \text{middle} must be true and \text{left} and \text{right} must be false. The open premise also shows that the facts are not contradictory. However, the result may be more complicated to interpret:

- We must use only equivalence rules, i.e. rules that are true in both directions (i.e. the conclusion is equivalent to the conjunction of the premises). This means that e.g. the rule weakening formulas may not be used.

- One constant may be missing. If the result is just \vdash \text{right}, \text{left}, then we don’t know anything about the middle path (it may or may not lead to Delphi). It depends on the problem whether this is a correct solution.

- Often the resulting proof tree has more than one open premise. Then every premise yields a possible solution that fulfills the facts. However, this does not mean that the solution follows from the facts (i.e. is unique). The solution is unique if all premises yield the same solution (and this solution contains all constants).

If we have the two premises
\langle \text{middle} \land \neg \text{left} \land \neg \text{right} \rangle
we have two solutions, \text{middle} \land \neg \text{left} (and we don’t know anything about \text{right}), and a second solution \neg \text{right} (and we don’t know anything about \text{middle} and \text{left}). This is certainly not a correct solution!

If the premises are \text{middle} \land \text{left}, and \text{middle} \land \text{right}, the solution is not unique. \text{middle} must be true, but we don’t know anything about \text{left} or \text{right}. (This may be a correct solution.)

- The resulting proof tree is closed. This means that the facts are contradictory. (Which may be a correct solution.)

Your task: Work on the specification proplogic, use basic rules, and any heuristics you like (a good selection is all heuristics – how does their order influence the proofs?). Prove the three theorems delphi-try, delphi-solution, delphi-fulfills, and compare the proof trees. (Of course, delphi-try is not provable.)
Exercise 1.3 Propositional logic
Work on the specification proplogic, use basic rules, and any heuristics you like. Formalize and prove the following puzzle using propositional logic. Use the propositional constants a, b, c. What is their intuitive meaning?

Mr. McGregor, a London shopkeeper, phoned Scotland Yard that his shop had been robbed. Three suspects A, B, C were rounded up for questioning. The following facts were established:

1. Each of the men A, B, C had been in the shop on the day of the robbery, and no one else had been in the shop that day.
2. If A was guilty, then he had exactly one accomplice.
3. If B is innocent, so is C.
4. If exactly two are guilty, then A is one of them.
5. If C is innocent, so is B.

Whom did Inspector Craig indict?

Add the formalization as a theorem in the file sequents (see section 2.1.2). Note that you should enter a sequent, not a formula, and that you can only use the symbols defined in the specification.

Exercise 1.4 Propositional logic
Work on the specification proplogic, use basic rules, and any heuristics you like. Formalize and prove the following puzzle using propositional logic. Use the propositional constants af, bf, ad, bd. (What is their intuitive meaning? a = ‘amplifier’, b = ‘bycicle’, f = ‘flux generator’, d = ‘dragon trap’)

Engineer Trurl wants to construct two machines (a probabilistic flux generator and a universal dragon trap) from a heap of scrap metal. The most valuable components at his hands are a chance amplifier and a bicycle. His colleague Klapauzious asks, ‘Is it true that if you need the bicycle for the probabilistic flux generator, and the chance amplifier for the dragon trap if and only if you also need the bicycle for it, you then don’t have to install the amplifier in the flux generator?’ Trurl ponders ‘If this statement is true, then I need the bicycle for exactly one machine, and the same holds for the chance amplifier. On the other hand, if I need the bicycle at all then the statement must be false. But in each case I do not need both components for both machines.’

Now, Which component is needed for which machine?

Show that a unique solution exists. What is it?

Exercise 1.5 Predicate logic
Work on the specification predlogic, use basic rules or not (you can change it with the menu Control – Options and Use Basic Rules), and any heuristics you like. Prove the following theorems:

1. Allneg : ⊢ (∀ x. ¬ p(x)) ↔ ¬ ∃ x. p(x);
2. Exneg : ⊢ (∃ x. ¬ p(x)) ↔ ¬ ∀ x. p(x);
3. Allimpleft : ⊢ ((∀ x. p(x)) → q(y)) ↔ ∃ x. p(x) → q(y);
4. Allimpright : ⊢ (p(x) → ∀ y. q(y)) ↔ ∀ y. p(x) → q(y);
5. Eximpleft : ⊢ ((∃ x. p(x)) → q(y)) ↔ ∀ x. p(x) → q(y);
6. Eximpright : ⊢ (p(x) → ∃ y. q(y)) ↔ ∃ y. p(x) → q(y);
Note that these rules allow to shift quantifiers in a formula (possibly with renaming of bounded variables). Specifically, all quantifiers can be shifted to the beginning of a formula so that the body is quantifier free (the only problem being equivalences, which must be first split into a conjunction of two implications). The resulting normal form is not unique. It is called a \textit{prenex} form.

\textbf{Exercise 1.6 Predicate logic}

In this exercise you have to invent formulas yourself. Note that you cannot use arbitrary variables, but only those that are declared in the specification! (Try the menu command \textit{View} – \textit{Specification} or click in daVinci on the node and select ‘View’.)

Work on the specification predlogic, use basic rules or not, and any heuristics you like. Solve the following tasks

\begin{enumerate}
\item prove \textit{exall-allex} : \(\vdash (\exists x. \forall y. \text{pr}(x,y)) \rightarrow \forall y. \exists x. \text{pr}(x,y)\);
\item prove \textit{allex-exex} : \(\vdash (\forall x. \exists y. \text{pr}(x,y)) \rightarrow \exists x. \exists y. \text{pr}(x,y)\);
\item prove \textit{not-exall} : \(\text{pr}(a, a), \text{pr}(b, b), \neg \text{pr}(a, b), \neg \text{pr}(b, a), \forall x. x = a \lor x = b\) \(\vdash \neg ((\forall y. \exists x. \text{pr}(x,y)) \rightarrow \exists x. \forall y. \text{pr}(x,y))\);
\end{enumerate}

\textbf{Exercise 1.7 Predicate logic}

Work on the specification predlogic, use basic rules or not, and any heuristics you like. Formalize and prove the following statements:

\begin{enumerate}
\item A barber is a man who shaves precisely those men who do not shave themselves.
It follows that no barber exists.
(use \textit{barber}(x) and \textit{shaves}(x,y))
\item Everybody loves my baby, but my baby loves nobody but me.
This means that ‘me’ and ‘baby’ are identical.
(use \textit{loves}(x,y), \textit{baby}, and \textit{me})
\item If everyone loves somebody and no one loves everybody, then someone loves some and doesn’t love others.
(use \textit{loves})
\end{enumerate}

Remember that you can only use the symbols of the specification. This is also true for variables. The specification contains the variables \(x, y, z\), etc., that you have to use.

\textbf{Exercise 1.8 Natural numbers}

In the specification \textit{nat}, the natural numbers are specified. This specification contains the constant 0, the successor function succ, and an (infix) addition function +. The axioms, which describe the natural numbers are

\begin{itemize}
\item distinctiveness : 0 \(\neq\) succ(n);
\item injectivity : succ(m) = succ(n) \(\iff\) m = n;
\item add-zero : n + 0 = n;
\item add-succ : m + succ(n) = succ(m + n)
\end{itemize}

plus the induction principle expressed through the generated by clause as described in chapter 13.

(Note: you can view the specification with the command \textit{View} – \textit{Specification}.)

Prove with the basic rules and without heuristics the correctness of the following theorems:

\begin{itemize}
\item \textit{lem-01} : \(\vdash 0 + n = n\)
\item \textit{lem-02} : \(\vdash \text{succ}(m) + n = \text{succ}(m + n)\)
\item \textit{com} : \(\vdash m + n = n + m\)
\end{itemize}
Hint: The first two theorems are propositions which should help to prove \textit{com}. Therefore prove the theorems in the above order without the use of \textit{com}. All proofs need induction.

**Exercise 1.9** Rewriting

This exercise shows that it must not be tedious to prove the theorem from the exercise above. The convenient proof technique is called “term rewriting”. This proof method uses equations \( \sigma = \tau \) in the following way as “rewrite rules”: If there is an instance of the term \( \sigma \) in the goal it will always be substituted through the corresponding instance of \( \tau \). (This requires that all variables of \( \tau \) also appear in \( \sigma \).) Therefore \( \tau \) should be ‘easier’ than \( \sigma \). The substitution of terms through easier ones can happen recursively as long as possible. Term rewriting is (beneath other things) done by the simplifier rule from the normal calculus.

Switch off the basic rules by using the menu command \textit{Control – Options}, and unmark the option \textit{Use Basic Rules}. Now try the proofs for lem-01, lem-02, and com again (with \textit{Proof Reprove}). If a theorem was successfully proved, it will used as a simplifier rule in the following proofs. You can also use heuristics for the proof if you want. Therefore select \textit{PL Heuristics + Struct. Ind.}.
Chapter 3

Simplification

3.1 General Comments

Deduction on algebraic specifications in KIV is mainly based on the idea that most proof steps of predicate logic proofs (especially the steps that can be done automatically) are simplification steps. We use “simplification” in an intuitive way, so it may depend on the ideas of the user. A complete proof for a formula \( \varphi \) is in this point of view a special case where \( \varphi \) is simplified to the (“simplest”) formula \( \text{true} \).

KIV is told about the exact nature of simplification steps by giving simplifier rules. Simplifier rules are sequents whose syntactical form describes what simplification step should be done. Simplification steps must obviously be correct, but they are also required to be invertible: the application of an inverted (“complicating”) step must also correct. This implies that it is impossible that a provable goal becomes unprovable by simplification. We treat here only two forms, the rest is described in appendix A.

Important: One of the main obstacles to using KIV effectively, is to understand what exactly the effects of a simplifier rule are. Adding simplifier rules blindly (just because this is offered) is a very bad idea. In particular it may lead to infinite loops in the simplifier.

3.2 Typical simplification steps

Typically one can distinguish between two sorts of simplification steps:

- formula substitution steps: In this case a formula is substituted by a simpler one. Typical examples are the substitution of formulas of the form \( \varphi \land \varphi \) by \( \varphi \), and the substitution of \( \neg \sigma < \text{succ}(\tau) \) by \( \tau < \sigma \). The first rule is independent of concrete data types, the second one depends on the data type (it is only valid for natural numbers).

- term rewriting steps: In this step a term is rewritten to another, simpler one. A typical example is the simplification of a term \((\sigma + \tau) - \tau \) to \( \sigma \). Term substitution steps always depend on the data type.

Application of data type independent simplification steps is built into the KIV rule ‘simplifier’. All propositional simplifications are done. In particular all propositional rules of sequent calculus with one or no premise are applied. Also, e.g. a formula of the form \( \varphi \to \varphi \land \psi \) is simplified to \( \varphi \to \psi \). The quantifier rules ‘all left’ and ‘exists right’ that drop quantifiers in favor of new variables are applied too. Quantifiers of the form \( \exists x.\varphi \) are simplified to \( \varphi^\tau_x \). Equation of the form \( x = \tau \) are inserted and the equation is dropped afterwards.

Data type dependent simplification steps may often be used only if certain preconditions are fulfilled. An example is \((n - m) + n = n \) which may only be applied if \( m \leq n \) (otherwise \( n - m \)
yield an unspecified number, and the equation may be false). The validity of preconditions can be shown in KIV in two ways:

- **recursive call of the simplifier**: This strategy attempts to simplify the precondition to true. This is the common strategy in most systems (there are also many automated provers which combine this strategy with the search for simplification rules according to certain strategies). But there is one disadvantage in this strategy: It is very inefficient because it invests a large amount of time in the attempts to prove these rules (especially if the attempt fails). Furthermore, using this strategy makes it very obscure what happens during the simplification. Therefore KIV usually does not call the simplifier recursively to prove the preconditions of a rule but uses the following strategy:

- **search for preconditions**: With this strategy the system tries to find the precondition in the sequent. This corresponds to the first strategy, but instead of proving the precondition with the full simplifier it is only tested on “is axiom”. This strategy is obviously weaker than the first one and leads often to the fact that variants of preconditions are needed. In the above example a variant with the stronger precondition \( m < n \) (instead of \( m \leq n \) is often needed.

The user can decide which variant to use by placing a precondition in the antecedent (first strategy) or before an implication (second strategy). We have found that the first strategy is usually too inefficient, even if it does some more small proofs can be found automatically. Therefore we typically use the second strategy.

An exception are preconditions that should always be fulfilled in ‘reasonable’ sequents. In our example \((m - n) + n\) the precondition should always be true in any reasonable goal. If it is not possible to prove \( n \leq m \) this is usually a hint that the goal is not provable and should lead to the abortion of the whole proof. Therefore one should always try to prove \( n \leq m \) with the simplifier to verify the precondition for the simplification of \((m - n) + n\) to \( m \). The correct form for this rewrite rule therefore is

\[
m \leq n \vdash (n - m) + m = n
\]

An example, where the precondition is typically not true is \( x \neq [] \rightarrow (x + y).\text{first} = x.\text{first} \) (from the list exercise at the end of the next chapter). This rule says that the first element of the list \( x + y \) (which appends lists \( x \) and \( y \)) is the same as the first element of \( x \), provided \( x \) is not the empty list. Replacing \((x + y).\text{first} \) with \( x.\text{first} \) obviously makes the formula simpler, but the current goal may not specify whether \( x \) is empty or not (maybe the proof requires a case split). The correct form for this simplifier rule therefore is

\[
\vdash x \neq [] \rightarrow (x + y).\text{first} = x.\text{first}
\]

It remains to explain the general form of the simplification rules for the simplification steps shown above.

### 3.3 Syntactical form of simplifier rules

- **term rewriting steps**: term rewriting rules have the following form

\[
\Gamma \vdash \varphi \rightarrow \tau = \sigma
\]  

(1)

where \( \tau \) is the term to be simplified, \( \sigma \) the result of the simplification. \( \Gamma \) contains the preconditions which should be validated by a recursive call of the simplifier, \( \varphi \) are the preconditions to be searched for. The rule has to be read as a scheme, i.e. it can be used for every substitution of the variables by terms. There are the following restrictions for rules:
CHAPTER 3. SIMPLIFICATION

1. All variables contained in the sequent must already be contained in \( \tau \) (i.e. if in a sequent an instance of \( \tau \) is found the instantiation of the whole rule is fixed.)

2. \( \tau \) must not be a variable.

3. \( \phi \) must be a conjunction of literals (i.e. predicates and equations or their negations).
   \( \phi \) may be omitted so that the succedent of the sequent is simply the equation \( \tau = \sigma \).

4. \( \Gamma \) contains no quantifiers.

- **Formula substitution steps:** Formula substitution rules have the form

\[
\Gamma \vdash \phi \rightarrow (\psi \leftrightarrow \chi)
\]

where \( \psi \) is the formula to be simplified, \( \chi \) the result of the simplification. \( \Gamma \) contains (as in the previous case) the preconditions to be proven by a recursive call of the simplifier, \( \phi \) are the preconditions to be searched for. The rule has to be read as a scheme as well and there are again some restrictions:

1. All variables contained in the sequent must already be contained in \( \phi \) and/or in \( \psi \). This restriction is weaker than above: if in a sequent an instance of \( \phi \) and \( \psi \) is found the instantiation of the whole rule is fixed.

2. \( \psi \) must be a literal or a negated literal.

3. \( \phi \) must be a conjunction of literals. \( \phi \) may be omitted so that the succedent of the sequent is simply the equivalence \( (\psi \leftrightarrow \chi) \)

4. \( \Gamma \) contains no quantifiers.

The effect of the rule depends on the number of negations in front of \( \psi \). If \( \psi = \neg \psi_1 \), instances of \( \psi_1 \) in the succedent are simplified to \( \chi \). If \( \psi = \neg \neg \psi_1 \) then instances of \( \psi_1 \) in the antecedent are simplified to \( \chi \). If \( \psi \) is an atomic formula (i.e. an equation or a predicate and not a negated formula) \( \psi \) is simplified to \( \chi \) in the antecedent and in the succedent. The last case is the common one because in most cases both simplifications should be applied.

In the last case (where \( \psi \) is an atomic formula) the rule has a modified effect if \( \chi \) is a disjunction or a conjunction of literals (this effect can obviously be avoided if \( \chi \) is substituted by \( \neg \neg \chi \)). An example for this effect is the following rule:

\[
\Gamma \vdash ar = ar[n, d] \leftrightarrow \neg n < \#ar \lor d = ar[n]
\]

Normally \( ar = ar[n, d] \) should be substituted in the succedent by \( \neg n < \#ar \lor d = ar[n] \) since this simply leads to two formulas. Substituting it in the antecedent simply leads to a disjunction which is more difficult to read than the original formula and which can only be removed by an explicit case distinction. Therefore the formula is substituted in the antecedent only if one of the elements of the disjunction is already present in the succedent.

This gives the following three rules:

\[
\begin{align*}
\Gamma, n < \#ar \vdash d = ar[n], \Delta \\
\Gamma \vdash ar = ar[n, d], \Delta
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash n < \#ar, d = ar[n], \Delta \\
\Gamma, ar = ar[n, d] \vdash \Delta, d = ar[n]
\end{align*}
\]

\[
\begin{align*}
\Gamma, ar = ar[n, d], n < \#ar \vdash \Delta
\end{align*}
\]
3.4 Input and use of simplifier rules

Simplifier rules are added in KIV with the command Theorems - Load New as theorems. The sequent is defined in the sequents file. In this file theorems may be marked as simplifier rules by using the keywords used for: together with the key ‘s’ for a global simplifier rule or the key ‘ls’ for a local simplifier rule (see below). An example:

\[
\text{lemmas}
\]

\[
\text{simp-rule : } \vdash n < \# ar \rightarrow ar[n, d][n] = d ; \text{used for : s, ls;}
\]

Another possibility is to add or delete a theorem or an axiom as a simplifier rule using a menu command. Theorems may be used as local simplifier rules (used only in the specification they are defined in) or as global simplifier rules in all higher specifications (or as both). Therefore there are two commands to add (Simplifier - Add Local Simplifier Rules and Simplifier - Add Simplifier Rules) or to delete simplifier rules (Simplifier - Delete Local Simplifier Rules and Simplifier - Delete Simplifier Rules; the theorem itself is not deleted), in the view menu there are two commands as well. Global simplifier rules are here structured, that is they are sorted by the subspecification they are defined in.

Application of local simplifier rules is guarded by the correctness management, i.e. the system prevents applications which would lead to a cycle in the proof dependencies of the theorems. Local simplifier rules are shown in the proof tree like normal lemmas as premises. Changes in local simplifier rules have direct effect on the proofs in which they are used. If a theorem used in a proof as local simplifier rule is deleted the proof gets invalid at once.

In contrast, global simplifier rules need not be checked by the correctness management because they cannot lead to cyclical dependencies in proofs. The deletion of a simplifier rule or even the deletion of the whole theorem does not make proofs in higher specifications invalid. The check if all used simplifier rules are still marked as simplifier rules or do still exist can explicitly be done by Specification - Check Simplifier Rules and Specification - Check Spec Theorems. It may also be delayed until the specification enters the proved state (Proof - Enter Proved State). The advantage is that deleting a simplifier rule by accident or doing an incorrect change does not make all proofs invalid.

It often happens that during working on a proof a new simplifier rule is needed. This new simplifier rule in a subspecification of the current specification should be added in the following way:

1. Load the subspecification with File - Load or - in case it is already loaded - with File - Specification ⟨name⟩.
2. Add the theorem in the subspecification. Now you can choose between two possibilities:
   Either you prove the theorem at once and do not risk to use an incorrect theorem or you can postpone the proof if you do not want to loose concentration for the main proof.
3. Add the theorem as a simplifier rule.
4. Save the subspecification with File - Save.
5. Switch back to the main specification with File - Specification ⟨name⟩.
6. Continue the proof. (If the last step has been the use of the simplifier it should be pruned. Then the heuristics apply the simplifier again using the new rule.)
Chapter 4

Theory Proofs

In this exercise you will prove theorems over two data structures: lists and heaps. As you will see these proofs differ from the proofs of the previous chapter. In theory proofs over lists it is very important to find a suitable induction scheme and to apply the correct lemmas. For proofs about heaps the extensionality axiom is important.

4.1 Editing Theorems

The Specification level is used to work on a single specification. Besides viewing and printing a specification (or parts of it) it is possible to enter and change theorems and to start and continue the proof a theorem.

All theorems entered at this level are stored in a file called theorem base. When a specification is selected the theorem base is loaded. So two copies of the theorem base exist while working on a specification, the “internal” version in KIV and the “external” version on disk. It is possible that several people work on the same unit in parallel. Therefore a locking mechanism exists which prevents that inconsistent changes are made to the theorem base.

If you want to add a new theorem or change an existing one there are always two possibilities: The first one is to enter the theorem directly with the commands Theorems - New Theorem resp. Theorems - Edit Theorem. The system prompts for all information needed. How special symbols can be entered in a dialog window or in a file is described in section 2.2.

The second possibility is to use the file sequents to add or change theorems. This file contains a template starting with a short description of the syntax. If the comment around (: :variables : :) is removed new variables can be defined using the same syntax as in specifications. The theorems can be entered between the keyword lemmas and the quotation mark according to the following syntax:

\[
\begin{align*}
\text{name} & : \text{sequent} ; \\
\text{used for} & : \text{some_flags} ; \\
\text{comment} & : \text{some_comment} ;
\end{align*}
\]

where 'used for' and 'comment' are optional. <some_comment> may be any text not containing a ';', <some_flags> is a comma separated list with information that the lemma should be used as a simplifier rule. Since you can add or delete simplifier rules etc. by menu commands, you can always omit 'used for' if you want. Possible flags are:
simplifier rule: simp, s, S, simplifier_rule, simplifier, simplification
local simplifier rule: localsimp, ls, LS, local_simplifier_rule, locsimp, local_simplifier, local_simplification
elimination: elim, e, E, elimination
forward: forward, f, F
local forward: localforward, lf, LF, local_forward
cut: cut, c, C
local cut: localcut, lc, LC, local_cut

A description of these simplifier commands can be found in chapter 3 and in appendix A.

All text that is written behind the exit command is ignored by the system.

4.2 Short Description of the Menu Commands

Here all menus and the important commands are described:

- Menu File: The following commands are available:
  - Save: Save all changes in the theorem base (as proofs, new theorems, ...). A locking mechanism prevents two people saving the theorem base at the same time.
  - Load: Load additional units to work on (units are specifications or modules). You can switch between these units with the command Specification <name>. The command has a similar effect as Work on ... on the project selection level.
  - Specification <name>: Switch between different units already loaded.
  - Unlock Theorem: Unlock theorems. A theorem is locked while somebody is working on it to prevent two people storing the same proof in parallel. If the system crashes or changes are not saved a theorem may be locked when you start working on a unit. In this case it should be unlocked with this command.
  - Unlock Proofsdir: Unlock the proof directory. This is only needed if KIV was aborted.
  - Go Back: Go back to the project level. All changes should be saved using File – Save first.
  - Exit All: Go back to the project selection level. All changes should be saved using File – Save first.

- Menu Edit: Edit some files using Emacs.

- Menu Print: Print some information.

- Menu Latex: Generate \LaTeX files containing information about theorems, proofs, statistics, used properties and specifications. These files are stored in the directory doc.

- Menu View: View some information.

- Menu Specification: Change parts of the specification.

- Menu Simplifier: Perform changes in the simplifier. For details see chapter 3 and appendix A.

- Menu Theorems:
  - Load New: Load new theorems from the sequents file.
  - Enter New: Enter new theorems directly.
  - Load Changed: Load changed theorems from the sequents file. Changing theorems is possible only if you are the only one working on the unit.
  - Enter Changed: Enter changed theorems directly.
– Change Comment: Change the comment of a theorem.
– Rename: Rename theorems.
– Delete: Delete theorems. Deleting theorems is possible only if you are the only one working on the unit.
– View: View theorems.

• Menu Control: Change some options.
• Menu Proof:
  – Begin: Begin a new proof. A locking mechanism prevents two people proving the same theorem in parallel.
  – Continue: Continue to prove a partially proven theorem.
  – Load: Load the proof of a theorem and modify it.
  – Reprove: Prove a theorem again starting with an empty proof.
  – Begin Some: Begin new proofs for several theorems.
  – Reprove Some: Prove some theorems again.
  – Replay Proofs: Replay an existing proof. This command is used if some proofs are invalid because theorems have changed.
  – Discard: Should not be used.
  – Delete Some: Delete some proofs. It is also possible to delete only invalid or partial proofs.
  – Add Extern: Should not be used.
  – Enter/Leave Proved State: Enter/Leave proved state (see project level).

4.3 Rules for Predicate Logic

The rules for predicate logic as used in KIV is similar to the set of basic rules, but in some respects optimized. For example, the propositional rules with two premises are combined in one rule *case distinction* that can be applied context sensitively by clicking anywhere in the formula. The propositional rules with one premise are combined in *prop simplification*, which itself is subsumed by *simplifier*. *insert spec-lemma* applies a lemma as *insert lemma*, but from a subspecification etc.

4.3.1 General Rules

**weakening**

\[ \frac{\Gamma' \vdash \Delta'}{\Gamma \vdash \Delta} \]

- \( \Gamma' \subseteq \Gamma, \Delta' \subseteq \Delta \)
- The formulas to discard can be selected.

**cut formula**

\[ \frac{\varphi, \Gamma \vdash \Delta \quad \Gamma \vdash \varphi, \Delta}{\Gamma \vdash \Delta} \]

- \( \varphi \) is the cut formula. It can be given by typing it in or by loading it from the file *formulas* (where it must be inclosed in "%...").
case distinction

\[
\frac{\psi_1, \Gamma \vdash \Delta \quad \psi_2, \Gamma \vdash \Delta \quad \psi_n, \Gamma \vdash \Delta}{\psi_1 \lor \psi_2 \lor \ldots \lor \psi_n, \Gamma \vdash \Delta}
\]

\[
\frac{\varphi, \psi, \Gamma \vdash \Delta}{\neg \varphi, \chi, \Gamma \vdash \Delta}
\]

\[
\frac{\varphi, \Gamma \vdash \psi, \Delta}{\neg \varphi, \Gamma \chi, \vdash \Delta}
\]

\[
\frac{(\varphi \supset \psi; \chi), \Gamma \vdash \Delta}{\Gamma \vdash (\varphi \supset \psi; \chi), \Delta}
\]

- the rule is also applicable for →, ↔ and for formulas in the succedent with ∧, ↔ (yielding an appropriate premise, of course).

insert equation

\[
\frac{\sigma = \tau, \Gamma \vdash \Delta}{\sigma = \tau, \Gamma \vdash \Delta}
\]

- The rule presents all equations \( \sigma = \tau \) of the sequent (with both orientations). Selecting an equation replaces the left side by the right.
- The positions where \( \sigma \) can be replaced by \( \tau \) are indicated by markers (##. . . ). You can select some of them or “All positions”.
- The equation may be discarded in the premise.

4.3.2 Simplification Rules

simplifier

\[
\frac{\Gamma' \vdash \Delta'}{\Gamma \vdash \Delta}
\]

- \( \Gamma \) and \( \Delta \) are simplified to \( \Gamma' \) and \( \Delta' \) (see chapter \( A \)).

prop simplification

\[
\frac{\varphi, \psi, \Gamma \vdash \Delta}{\varphi \land \psi, \Gamma \vdash \Delta}
\]

- The rule combines the propositional logic rules with one premise, i.e. negation left/right, conjunction left, disjunction right, implication right.
- The rule is rarely needed because the simplifier rule also applies these rules. However, the simplifier also applies rewrite rules et.al., which is sometimes undesirable.

4.3.3 Induction Rules

induction

\[
\frac{\Gamma \vdash \Theta(\varphi), \Delta \quad \Gamma, \Theta(\psi) \vdash \Delta \quad \varphi, \text{Ind-Hyp} \vdash \psi}{\Gamma \vdash \Delta}
\]

- \( \text{Ind-Hyp} = \forall \bar{x}, u'.(u' \ll u \rightarrow (\varphi \rightarrow \psi)^{u'}) \)
- \( \bar{x} = \text{free}(\varphi \rightarrow \psi) \setminus \{u\} \)
- \( u \) is the induction variable
- The variant term induction allows to induct over a term \( \tau \). This is equivalent to adding an equation \( x = \tau \) with new variable \( x \) to the sequent and induction on \( x \).
- \( \Theta \) is a parallel substitution for \( u, \bar{x} \)
- \( \ll \) is a well founded ordering
apply induction

\[ \Gamma \vdash \Theta(u') \ll u, \Delta \quad \Gamma \vdash \Theta(\varphi), \Delta \quad \Theta(\psi), \Gamma, \text{Ind-Hyp} \vdash \Delta \]

\[ \Gamma, \text{Ind-Hyp} \vdash \Delta \]

- Ind-Hyp = \forall \, x, u'. (u' \ll u \rightarrow (\varphi \rightarrow \psi))
- \Theta is a parallel substitution for \(x\)

structural induction (example)

\[ \vdash \varphi(c) \quad \forall \, x'. \varphi(x) \vdash \varphi(f(x)) \]

\[ \Gamma \vdash \Delta \]

- \(\varphi = \land \Gamma \rightarrow \lor \Delta\)
- \(x' = \text{free}(\varphi) \setminus x\)
- sort(x) generated by \(c, f\)

4.3.4 Quantifier Rules

all left/exists right

\[ \varphi^x, \forall \, x. \varphi, \Gamma \vdash \Delta \]

\[ \forall \, x. \varphi, \Gamma \vdash \Delta \]

\[ \Gamma \vdash \varphi^x, \exists \, x. \varphi, \Delta \]

\[ \exists \, x. \varphi, \Gamma \vdash \Delta \]

- An instance \(\tau\) of \(x\) must be given as \(\tau_1, \ldots, \tau_n\) manually or chosen from some precomputed substitutions.
- The quantified formula may be discarded in the premise

exists left/all right

\[ \varphi^x, \exists \, x. \varphi, \Gamma \vdash \Delta \]

\[ \exists \, x. \varphi, \Gamma \vdash \Delta \]

\[ \Gamma \vdash \forall \, x. \varphi, \Delta \]

\[ \forall \, x. \varphi, \Gamma \vdash \Delta \]

- \(\varphi'\) are new variables (automatically computed)

4.3.5 Lemma Insertion Rules

insert lemma/insert spec-lemma

\[ \Gamma' \vdash \Delta' \quad \Gamma \vdash \Gamma', \Delta \quad \Delta', \Gamma \vdash \Delta \]

\[ \Gamma \vdash \Delta \]

- \(\Gamma' \vdash \Delta'\) is the theorem with free variables \(x\).
- An instance \(\tau\) of \(x\) must be given as \(\tau_1, \ldots, \tau_n\) manually or chosen from some precomputed substitutions.
**insert elim lemma**

\[
\Gamma' \vdash \Delta', \sigma(\varphi) \land \sigma(\Gamma) \quad \sigma(\psi), \Theta(\Gamma') \vdash \Theta(\Delta')
\]

- \( \Gamma \vdash \varphi \to (x_1 = t_1 \land \ldots \land x_n = t_n \leftrightarrow \exists y v = t \land \psi) \) is the elimination rule
- \( \sigma \) is a substitution for the free variables of the elimination rule
- \( \Theta \) is the substitution \( \sigma(t_1), \ldots, \sigma(t_n) \leftarrow \sigma(x_1), \ldots, \sigma(x_n)[\sigma(v) \leftarrow \sigma(t)] \)
- For details see appendix A

**insert rewrite lemma**

\[
\Gamma \vdash \varphi \to \sigma = \tau
\]

\[
\Gamma \vdash \Theta(\varphi), \Delta
\]

\[
\Gamma' \vdash \Delta'
\]

\[
\varphi^{\Theta(\tau)}, \Delta^{\Theta(\tau)}, \Gamma^{\Theta(\tau)} \vdash \Delta^{\Theta(\tau)}
\]

- \( \vdash \varphi \to \sigma = \tau \) is the rewrite theorem
- An instance \( \Theta(\sigma) \) resp. \( \Theta(\psi) \) must be contained in the sequent.
- The system offers all applicable instances. Select one of these.

### 4.3.6 Insert rewrite lemma

Some extra remarks about a very useful new rule: insert rewrite lemma.

\[
\vdash \varphi \to \sigma = \tau \quad \Gamma \vdash \Theta(\varphi), \Delta
\]

\[
\Gamma \vdash \Delta
\]

\[
\varphi^{\Theta(\tau)}, \Delta^{\Theta(\tau)}, \Gamma^{\Theta(\tau)} \vdash \Delta^{\Theta(\tau)}
\]

Suppose you have a theorem \( n \neq 0 \to \text{succ}(\text{pred}(n)) = n \), your goal contains somewhere the subterm \( \text{succ}(\text{pred}(m + m0)) \), and you want to rewrite \( \text{succ}(\text{pred}(m + m0)) \) to \( m + m0 \). You can cut formula with \( m + m0 = 0 \), insert lemma the theorem, add a substitution, apply case distinction, and insert equation. It works, but is quite tedious. Instead, you can use the rule insert rewrite lemma, select the theorem, and get the same result without any more interaction. You can also apply the rule context sensitive by clicking on the \text{succ} symbol in \( \text{succ}(\text{pred}(m + m0)) \) (with the right mouse button). You will get a popup menu that contains only those rewrite rules that are applicable on the selected term. This means you don’t have to know that this rewrite rules exists (and where it is stored), but can try it out.

If you select the rule from the rule list, the system displays a list with the specification, all subspecifications, and how many potentially applicable rewrite rules they contain. Then you have to select a specification. The system know computes those rewrite rules that are indeed applicable to the current goal. It may be that no rule remains. (The reason for this approach is that there may be a lot of subspecifications with lots of rewrite rules – checking their applicability may be to inefficient.) Otherwise the applicable rules are displayed. If you want to apply the rule context sensitive you have to click on a function (or predicate) symbol. The system will display all rewrite rules that have this function symbol as their top- (or outer-)most function symbol, and are applicable at the current position in the goal. Be warned: this can be confusing if the outermost symbol is an infix or postfix operation! If you click somewhere else or no rewrite rule is applicable, nothing happens. (But context sensitive insert equation or quantifier instantiation...
works as described.) Otherwise the system will display the applicable rules with their name and the sequent. For some rewrite rules it is not possible to automatically compute a complete substitution. In these cases you have to select or enter a substitution as in insert lemma. Finally, the rule is applied.

There are quite a lot of theorems that can be used with this rule, but not all:

- $\Gamma \vdash \varphi \rightarrow \sigma = \tau$ rewrites either $\sigma$ by $\tau$ or $\tau$ by $\sigma$ (comparable to insert equation where you can also replace the left side by the right or vice versa). $\Gamma$ may be empty, and $\varphi \rightarrow$ may be missing. $\Gamma$ and $\varphi$ may be arbitrary formulas (compare this to the restricted form of rewrite rules the simplifier uses).

- $\Gamma \vdash \varphi \rightarrow (\psi_1 \leftrightarrow \psi_2)$ rewrites $\psi_1$ by $\psi_2$ or vice versa. Again $\Gamma$ and $\varphi \rightarrow$ may be missing. However, there are restrictions on the formula to rewrite: To rewrite $\psi_1$ by $\psi_2$, $\psi_1$ must be a literal.

- $\Gamma \vdash \varphi \rightarrow \psi$ where $\psi$ is a predicate or a negated predicate. Read this as $\Gamma \vdash \varphi \rightarrow (\psi \leftrightarrow \text{true/false})$.

And finally one word of warning about the simplifier: You can’t do serious proofs without the simplifier (as you will probably notice very fast). But it may do things you didn’t intended it to do. For example, if you insert a lemma that is also a simplifier rule the simplifier will probably ‘remove’ the lemma from your goal by rewriting it (with the lemma) to true. This particularly happens if you use the simplifier heuristic, which you should do. In such a case you must

1. turn the heuristics off (by clicking on ‘Use Heuristics’ in the lower left corner of the proof window),
2. prune the last proof step in the proof window,
3. use the lemma as you intended it (e.g. with insert equation),
4. and turn the heuristics on again.

insert rewrite lemma avoids this phenomenon. So use this rule whenever possible!

### 4.4 Predefined Sets of Heuristics

Heuristics are used to decrease the number of user interactions. In the best case they find proofs automatically. Even if a proof is not found completely automatically heuristics can help the user by applying often used, mechanical rules automatically. But there is also a risk when using heuristics: They may apply steps which are unnecessary or even unwanted.

During a proof heuristics can be enabled and disabled at any time (using the command Control – Heuristics). Therefore heuristics can be used for a single branch of the proof or for the whole proof. An important property of heuristics is that there is an arbitrary hierarchical ordering: If a heuristic can’t be applied the next heuristic according to the hierarchical ordering is tested. Furthermore a heuristic can determine which other heuristics should be applied next. Therefore heuristics are a very flexible concept.

To choose a good set of heuristics some experience is needed. To simplify matters KIV offers three predefined sets of heuristics which are usually sufficient. Therefore after selecting the command Control – Heuristics a window is displayed where a set of heuristics can be chosen:

Let the system choose the heuristics
PL Heuristics + Induction
PL Heuristics + Case Splitting
PL Heuristics
No heuristics at all
Select your own heuristics
Read from file ‘default heuristics’
Let the system choose the heuristics: If this option is selected KIV chooses one of the sets of heuristics described below. The criterion is a very basic estimation of the complexity of the sequent to be proven (i.e. number and kind of formulas and their combinations). This is an experimental feature and is not recommended.

**PL Heuristics + Induction**: This set of heuristics is best fitting for simple goals. The proof idea should be induction, and it shouldn’t be necessary to avoid case distinctions. This set of heuristics finds proofs often fully automatically. The following heuristics are contained:

- simplifier, elimination, module specific, Quantifier closing, cut, if-then-else split, pl case distinction, structural induction

**PL Heuristics + Case Splitting**: This set of heuristics is used for more complex goals. It is used e.g. for uncommon inductions or if some lemmas have to be applied first. In contrast to the heuristics for simple goals the induction heuristic and some heuristics that introduce case distinctions are missing. This set contains the following heuristics:

- simplifier, elimination, module specific, Quantifier closing, weak cut, if-then-else split

**PL Heuristics**: This set of heuristics is used for very complex goals where case distinctions must be avoided. It contains the following heuristics:

- simplifier, elimination, module specific, Quantifier closing

**No heuristics at all**: All heuristics are disabled.

**Select your own heuristics**: You can choose yourself which heuristics you want to use, or you can modify the current selection. A description of the heuristics is given in the next section.

After selecting this command a window opens containing two columns: on the left side all available heuristics are displayed, on the right side the currently selected heuristics are shown in order of selection. To select a heuristic click on this heuristic on the left side. It appears on the right side at the position of the black bar. To delete a heuristic currently selected click again on the heuristic in the left column. After leaving the window by clicking on the OK-Button the selected heuristics are applied according to their order.

**Read from file 'default heuristics'**: The file 'default-heuristics' should contain a PPL list of heuristics: (list "..." "..." ...). This option is for experts only.

When you begin (or continue) a proof all heuristics are disabled (but still selected).

### 4.5 Description of the Heuristics

**Induction**

This heuristic applies the rule induction if a suitable induction variable and an ordering is found. Induction orderings are found in the heuristic info where size functions (as the length of a list or the number of elements of a set) are listed. If a data-specification is installed size functions and order predicates are automatically inserted into the heuristic info. Additional size functions can be added using the command Add Heuristic Info.

The induction rule is applied only once in each branch of a proof.
**structural induction**
This heuristic applies the rule *structural induction* on first order goals trying to find an induction variable. This heuristic applies the structural induction only once in each branch.

**apply ind once**
This heuristic applies the induction hypothesis (i.e. the rule *apply induction*) if a suitable instance is found. The same mechanism is used to find instances as is used to search for substitution proposals if the rule is used in an interactive way.

This heuristic does not apply the induction hypothesis if it was applied already in this branch by any heuristic or by the user. Therefore it cannot lead to infinite loops.

**apply ind**
This heuristic applies the induction hypothesis always if an instance is found. Therefore it can lead to infinite loops (not recommended).

**apply ind closing**
This heuristic applies the induction hypothesis if this closes the goal at once. It can’t lead to wasted applications of rules or infinite loops but is used rather seldom.

**Simplification**

**simplifier**
This heuristic decides when the simplifier (i.e. the rule *pl simplifier*) should be applied on a goal. The simplifier is called only if the goal has changed or the number of predicate logic formulas is increased. This heuristic should be used always.

**pl case distinction**
This heuristic resolves propositional logical combinations into case distinctions.

**Other Heuristics**

**module specific**
This heuristic is controlled by the information provided in the file module-specific. It is used for specific treatment of different modules or specifications (see appendix E).

**batch mode**
This heuristic changes the goal to be proven. If this heuristic is added after all other heuristics it is only applied if no other heuristic can be applied. The program works as long as possible without human interactions. It stops if a goal is reached again that was treated before.

**elimination**
This heuristic applies the rule *insert elim lemma* (see appendix 4.3.1 and A.3).

**var elimination**
This heuristic applies the rule *insert elim lemma* (see appendix 4.3.1 and A.3) only if the term to be eliminated is a variable.
Quantifier closing

This heuristic tries to find instances for quantified variables which close the goal at once. It applies in these cases the rules all left and exists right resp.

Quantifier

This heuristic instantiates quantifiers if a sensible instance is found even if the goal is not closed at once (here unification and a heuristical measure for “suitable” of instances is used). This heuristic can lead to wasted instantiations.

weak cut

This heuristic applies cut rules on a sequent (see appendix A.6).

cut

This heuristic applies cut rules on a sequent as weak cut but applies a cut with \( \varphi \) if \( \varphi \) is an atom and \( \varphi \lor \psi \) is contained in the antecedent of a goal (or \( \varphi \land \psi \) in the succedent).

if-then-else split

This heuristic applies case distinction on if-then-else’s.

4.6 Exercise 2

The specification list-basic defines lists of natural numbers. It specifies various list operations, and has several unproved theorems. Your task is to prove these theorems. You can use the basic rules (to get a feeling for the proofs) or the normal set of rules (described in appendix 4.3.1). Note especially the rule insert elim-lemma which is very helpful. You can also use any heuristics you like (see appendix 4.5). You can also invent new theorems (provided you prove them), but it is not necessary. Note that the option ‘Use only proved locals’ is active. Its effect is that the simplifier will only use theorems for simplification that are already proved (including all used theorems in the proof). The advantage is that a simple theorem is not proved unintentionally with a complex one. (This can lead to problems if the proof of the complex theorem requires the simple theorem.) On the other hand, the order in which the theorems are proved has a significant impact on the simplicity of the proofs. To make things easier, we provide a suitable order for you.

Exercise 2.1 (list-basic)

The specification list-basic defines lists and elements. The elements have a total irreflexive order \( < \) (note that \( < \) is also the symbol for less on the natural numbers). This is later needed to sort lists. \([\ ]\) is the empty list, \(+\) is used to add an element to the front of the list. \(+\) is also used to append two lists (and it also denotes addition of natural numbers). The postfix operators .first and .rest select the first element and the rest of a nonempty list. \# is the length of the list, \( \in \) tests if an element occurs in a list.

list-basic =

enrich nat with
  sorts elem, list;
  constants [ ] : list;
  functions
    . + . : elem \times list \rightarrow list prio 9;
    . + . : list \times list \rightarrow list prio 9;
    . first : list \rightarrow elem ;
    . rest : list \rightarrow list ;
    # : list \rightarrow nat ;
predicates
   . < . : elem \times elem;
   . \in . : elem \times list;

variables c, b, a : elem; z_2, y_2, x_2, z_1, y_1, x_1, z_0, y_0, z, y, x : list;

induction list generated by [ ], +;

axioms

irreflexivity : \vdash \neg a < a; used for : s, ls;
transitivity : \vdash a < b \land b < c \rightarrow a < c; used for : f, If;
totality : \vdash a < b \lor a = b \lor b < a; used for : s, ls;
constructors : \vdash [ ] \neq a + x; used for : s, ls;
first : \vdash (a + x).first = a; used for : s, ls;
rest : \vdash (a + x).rest = x; used for : s, ls;
append-base : \vdash [ ] + x = x; used for : s, ls;
append-rec : \vdash (a + x) + y = a + x + y; used for : s, ls;
size-base : \vdash \#([ ]) = 0; used for : s, ls;
size-rec : \vdash \#((a + x)) = \#(x) + 1; used for : s, ls;
In : \vdash a \in x \leftrightarrow (\exists y, z. x = y + a + z);

end enrich

The following theorems are already proven. All except the last theorem deal with the simplification of the total order < on elements. The last theorem is a very elementary property of lists, but it requires a little trick to prove.

<table>
<thead>
<tr>
<th>Theorem</th>
<th>Statement</th>
<th>Used for</th>
</tr>
</thead>
<tbody>
<tr>
<td>lel</td>
<td>\vdash a &lt; b \land b = c \rightarrow a &lt; c;</td>
<td>f, If</td>
</tr>
<tr>
<td>lel-01</td>
<td>\vdash a &lt; b \land a = c \rightarrow c &lt; b;</td>
<td>f, If</td>
</tr>
<tr>
<td>loc</td>
<td>\vdash \neg a &lt; b \leftrightarrow \neg (a = b \lor b &lt; a);</td>
<td>ls</td>
</tr>
<tr>
<td>nen</td>
<td>\vdash \neg b &lt; a \land b = c \rightarrow \neg c &lt; a;</td>
<td>f, If</td>
</tr>
<tr>
<td>nen-01</td>
<td>\vdash \neg b &lt; a \land a = c \rightarrow \neg b &lt; c;</td>
<td>f, If</td>
</tr>
<tr>
<td>null</td>
<td>\vdash \neg b &lt; a \land b &lt; c \rightarrow a &lt; c;</td>
<td>f, If</td>
</tr>
<tr>
<td>nnm</td>
<td>\vdash \neg a &lt; b \land \neg a &lt; c \rightarrow \neg a &lt; b;</td>
<td>f, If</td>
</tr>
<tr>
<td>seq</td>
<td>\vdash \neg b &lt; a \rightarrow (\neg a &lt; b \leftrightarrow a = b);</td>
<td>s, ls</td>
</tr>
<tr>
<td>sis</td>
<td>\vdash a \neq b \rightarrow (\neg a &lt; b \leftrightarrow b &lt; a);</td>
<td>s, ls</td>
</tr>
<tr>
<td>injectivity</td>
<td>\vdash a + x = b + y \leftrightarrow \neg \neg (a = b \land x = y);</td>
<td>s, ls</td>
</tr>
</tbody>
</table>

These operations are the most basic list operations. Many more can be defined with them. However, as you will see, they allow to formulate quite complicated theorems.

**Task:** Prove the following theorems in specification list-basic:

<table>
<thead>
<tr>
<th>Theorem</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>elim-list-c</td>
<td>\vdash x \neq [ ] \vdash a = x.first \land y = x.rest \leftrightarrow x = a + y;</td>
</tr>
<tr>
<td>case</td>
<td>\vdash x = [ ] \lor x = x.first + x.rest;</td>
</tr>
<tr>
<td>exrew</td>
<td>\vdash x \neq [ ] \leftrightarrow \exists a, y. x = a + y;</td>
</tr>
<tr>
<td>ex</td>
<td>\vdash x \neq [ ] \vdash \exists a, y, x = a + y;</td>
</tr>
<tr>
<td>append</td>
<td>\vdash x + [ ] = x;</td>
</tr>
<tr>
<td>append-01</td>
<td>\vdash x + y = x + z \leftrightarrow y = z;</td>
</tr>
<tr>
<td>append-02</td>
<td>\vdash x + y = [ ] \leftrightarrow x = [ ] \land y = [ ];</td>
</tr>
<tr>
<td>append-04</td>
<td>\vdash x = x + y \leftrightarrow y = [ ];</td>
</tr>
<tr>
<td>associativity</td>
<td>\vdash (x + y) + z = x + y + z;</td>
</tr>
<tr>
<td>first-append</td>
<td>\vdash x \neq [ ] \rightarrow (x + y).first = x.first;</td>
</tr>
<tr>
<td>rest-append</td>
<td>\vdash x \neq [ ] \rightarrow (x + y).rest = x.first + y;</td>
</tr>
<tr>
<td>first-split</td>
<td>\vdash y \neq [ ] \vdash a + x = y + z \leftrightarrow a = y.first \land x = y.rest + z;</td>
</tr>
<tr>
<td>first-split-01</td>
<td>\vdash y \neq [ ] \vdash y + z = a + x \leftrightarrow a = y.first \land x = y.rest + z;</td>
</tr>
</tbody>
</table>
length: \[ \vdash \#(x + y) = \#(x) + \#(y); \]

zero-length: \[ \vdash \#(x) = 0 \leftrightarrow x = []; \]

append-05: \[ \vdash x = y + x \leftrightarrow y = []; \text{ Hint: no struct. ind., use theorem length} \]

append-03: \[ \vdash x + y = z + y \leftrightarrow x = z; \]

append-06: \[ \vdash x + a + y = z + y \leftrightarrow x + a + [] = z; \text{ Hint: no struct. ind., use append-03} \]

append-07: \[ \vdash x + a + y = z + b + y \leftrightarrow x = z \land a = b; \]

append-05: \[ \vdash x + y = z + y \leftrightarrow x = z; \text{ Hint: no struct. ind., use theorem length} \]

in-empty: \[ \vdash \neg a \in []; \text{ Hint: no struct. ind., use axiom In} \]

in-cons: \[ \vdash a \in b + x \leftrightarrow \neg \neg (a = b \lor a \in x); \text{ Hint: no struct. ind., use axiom In} \]

in-append: \[ \vdash a \in x + y \leftrightarrow \neg \neg (a \in x \lor a \in y); \]

lastdiv: \[ \vdash x \neq [] \leftrightarrow (\exists y, z. x = y + a + z) \land a \in z); \]

lastoc: \[ \vdash a \in x \leftrightarrow (\exists y, z. x = y + a + z) \land a \in z); \]

split-first: \[ n \leq \#(x) \vdash \exists y, z. x = y + z \land \#(y) = n; \]

firstoc: \[ \vdash a \in x \leftrightarrow (\exists y, z. x = y + a + z) \land a \in y); \]

len-01: \[ \vdash \#(x) = \#(z) \rightarrow (x + y = z + y_0 \leftrightarrow x = z \land y = y_0); \]

split-rest: \[ n \leq \#(x) \vdash \exists y, z. x = y + z \land \#(z) = n; \text{ Hint: no struct. ind., use split-first} \]

Some comments: All theorems without a hint can be proved by simple structural induction. However, you have to find the correct variable for structural induction. (And correct instances for quantifiers, and the right lemmas to use, etc.) append – zero-length are some simple properties that are needed in lots of other proofs. They are all simplifier rules. append-05 – append-07 are also very useful simplifier rules, but more tricky to prove. in-empty and in-cons could be used as axioms instead of the declarative axiom In. With these two simplifier rules it is possible to prove other properties about ∈ with structural induction. lastdiv is also a useful theorem. lastoc and split-first are needed for the remaining theorems. They are more or less tricky to prove, but independent from each other.

Exercise 2.2 (list-last)

Specification list-last defines two new functions .last and .butlast. Both are written postfix (like .first and .rest). x.last selects and x.butlast removes the last element of a non-empty list. They are specified with the axiom

\[ \text{Last: } \vdash x \neq [] \rightarrow x\text{.butlast} + x\text{.last} = x \]

Task: Prove the following theorems in specification list-last:

<table>
<thead>
<tr>
<th>Theorem</th>
<th>Proof</th>
</tr>
</thead>
<tbody>
<tr>
<td>one-attach:</td>
<td>[ \vdash a + [] = x + a0 + [] \leftrightarrow a = a0 \land x = [] ]</td>
</tr>
<tr>
<td>last-elim:</td>
<td>[ x \neq [] \vdash a = x\text{.last} \land y = x\text{.butlast} \leftrightarrow x = y + a + [] ]</td>
</tr>
<tr>
<td>b:</td>
<td>[ (a + []).\text{butlast} = [] ]</td>
</tr>
<tr>
<td>b-02:</td>
<td>[ (a + x).\text{butlast} = [] \leftrightarrow x = [] ]</td>
</tr>
<tr>
<td>butlast-def:</td>
<td>[ (x + a + []).\text{butlast} = x ]</td>
</tr>
<tr>
<td>l:</td>
<td>[ (a + []).\text{last} = a ]</td>
</tr>
<tr>
<td>l-05:</td>
<td>[ (a + b + x).\text{last} = (b + x)\text{.last} ]</td>
</tr>
<tr>
<td>last-def:</td>
<td>[ (x + a + []).\text{last} = a ]</td>
</tr>
</tbody>
</table>

Hint: None of the proofs needs induction. last-elim is an elimination rule (see appendix A.5 for a detailed description) that can be used to get rid of the .last and .butlast function.
Chapter 5

Algebraic Specifications

The KIV system is based on structured algebraic specifications. In this chapter we describe the syntax of structured algebraic specifications and a methodology for their development. A formal description of the syntax of structured algebraic specifications is given in appendix D. A grammar of dynamic logic is given in appendix C. A more detailed description of the simplifier (with some additional features) can be found in appendix A.

5.1 Structured Algebraic Specifications

Basic Specifications

A basic specification consists of three parts: a description of the signature, the axioms and the principles of induction. A basic specification begins with the keyword specification and the declarations of sorts, constants, functions, predicates and variables. After the keyword induction follow the principles of induction consisting of 'generated by'-clauses (separated by ';'). Next are the axioms (after the keyword axioms, separated by ';'). The specification ends with the keyword end specification. Here is an example a specification of the integers:

```
specification
sorts int;
constants 0,1 : int;
functions . +1 : int \rightarrow int;
   . -1 : int \rightarrow int;
   . + . : int \times int \rightarrow int;
   . * . : int \times int \rightarrow int prio 10;
   # . : int \rightarrow int;
predicates . < . : int \times int;
variables x,y,z : int;
induction
   int generated by 0, +1, -1;
axioms
   x +1 -1 = x; x -1 +1 = x; 1 = 0 +1;
   \neg x < x; x < y \land y < z \rightarrow x < z;
   x < x +1; x +1 < y +1 \leftrightarrow x < y;
   x + 0 = x; x + y +1 = (x + y) +1;
   x * 0 = 0; x * y +1 = x * y + x;
   \neg x < 0 \rightarrow \# x = x; x < 0 \rightarrow \# x + x = 0;
end specification
```

In the example, infix, prefix, and postfix declarations are used. Normal function application binds more tightly than postfix application, postfix binds more tightly than prefix, infix operations
have the lowest priority. Infix operations have a priority given with \( 'prio n' \) with \( 6 \leq n \leq 15 \). If it is omitted it defaults to 9. The additional keyword \texttt{left} indicates that the function is left associative (otherwise it binds to the right). Functions and predicates with higher priorities bind more tightly than those with a lower one. Some examples:

\[
\begin{align*}
  x + y + z & \equiv x + (y + z) \\
  x \cdot y + x & \equiv (x \cdot y) + x \\
  x + y + 1 & \equiv x + (y + 1) \\
  \# x + 1 & \equiv \#(x + 1) \\
  \# x + x & \equiv (\# x) + x
\end{align*}
\]

A detailed explanation of all mixfix notations and all possibilities for overloading can be found in appendix B.

The following propositional logical operations are predefined: \( \neg, \land, \lor, \rightarrow, \leftrightarrow, \supset \) (in order of binding: \( \neg \) binds more tightly than \( \land \), \( \land \) binds more tightly than \( \lor \), \ldots). \( (\varphi \supset \psi; \chi) \) is an if-then-else operation meaning if \( \varphi \) then \( \psi \) else \( \chi \). For first order logic the Quantors \( \forall \) and \( \exists \) are predefined. They have always the largest possible scope. Some examples:

\[
\begin{align*}
  \neg x \land y & \equiv (\neg x) \land y \\
  (x \rightarrow y \supset \neg z; y \leftrightarrow z) & \equiv ((x \rightarrow y) \supset (\neg z); (y \leftrightarrow z)) \\
  \forall x. f(x) \land g(x) & \equiv \forall x. (f(x) \land g(x)) \land (\forall x. f(x)) \land g(x)
\end{align*}
\]

It is also possible to enter the name of the operation as \texttt{not, and, or, \ldots}.

### Generic Specifications

In contrast to basic specifications, generic specifications have a \textit{parameter}. Typical examples for generic specifications are sets or lists where the elements form the parameter or arrays and stores with the data stored in them as parameter. In most cases the parameter is a basic specification which only defines the name of the parameter sort and a few properties (e.g. that there is an ordering on the elements or that a ‘default’ constant exists). A parameter specification may not be generic itself. Usually it contains no ‘generated by’ clauses (because this restricts the possible elements too much). Their sorts and operations are called parameter sorts and parameter operations respectively. All other sorts and operations of a generic specification are called target sorts and target operations. The syntax of generic specifications begins with the keywords \texttt{generic specification} and \texttt{parameter} followed by the name of the parameter specification. The optional keyword \texttt{using} starts a list of additionally used specifications (separated by comma). Frequently the specification of natural numbers is used as an index sort, e.g. in the specification of arrays. Used specifications may be generic. Their parameters are added to the parameter of the whole specification. After the keyword \texttt{target} the description of sorts, constants, functions, predicates, generated by clauses and axioms is given in the same syntax as in a basic specification. The specification is terminated by the keyword \texttt{end generic specification}. The parameter of a generic specification may be instantiated by an \textit{actualization} (see below). An example for a generic specification is the following specification of arrays.

\[
\begin{align*}
\text{generic specification} \\
\text{parameter} & \texttt{elem-spec} \\
\text{using} & \texttt{nat-spec} \\
\text{target} & \texttt{sorts} \texttt{array;} \\
\text{functions} & \texttt{mkarray} : \texttt{nat} \rightarrow \texttt{array;} \\
& \quad . \ [\cdot ] : \texttt{array} \times \texttt{nat} \rightarrow \texttt{elem;} \\
& \quad . \ [\cdot ] : \texttt{array} \times \texttt{nat} \times \texttt{elem} \rightarrow \texttt{array;} \\
& \quad \# . : \texttt{array} \rightarrow \texttt{nat};
\end{align*}
\]
variables ar, ar0, ar1, ar2 : array;

induction
  array generated by mkarray, [ ] :: (array × nat × elem → array);

axioms
  Extension : ar1 = ar2 ↔ # ar1 = # ar2 ∧ ∀ n. n < # ar1 → ar1[n] = ar2[n]; used for : ls;
  Size-mkarray : # mkarray(n) = n; used for : s, ls;
  Size-put : # ar[n, a] = # ar; used for : s, ls;
  Put-same : m < # ar → ar[m, a][m] = a; used for : s, ls;
  Put-other : m ≠ n → ar[n, a][m] = ar[m]; used for : s, ls;

end generic specification

This specification is also used in chapter 5.2 as an example for the specification methodology in KIV. The parameter specification elem-spec declares only the sort elem and the variables a, b, c (KIV requires to declare at least one variable for every sort). The specification introduces further possibilities. [ ] is used to declare a mixfix notation so that array access can be written as usual. Updating an array is written as ar[n, a] (the array ar is updated at index m with value a). Because [] is overloaded we have add the types of the operation in the induction principle (:: (array × nat × elem → array)). The axioms have names and hints for the simplifier (used for : s, ls).

Union Specifications

One possibility to combine several specifications is to unite them into one specification. The syntax of a union specification begins with the keyword union specification followed by the specifications (separated by ‘+’) and the keyword end union specification. Signature, axioms, and parameters of a union specification are as expected the unions of the signatures, axioms and parameters of the subspecifications.

Enrichments

Another possibility to create larger specifications from smaller ones is the enrichment by new sorts and operations. These are added to the specification. The parameter of the enriched specification is the parameter of the subspecification. (A specification cannot be enriched by parameter operations. New parameter operations are needed very rarely, and the corresponding enrichment can be simulated by an actualization.) The syntax of an enriched specification starts with the keyword enrich and the name of the specification to be enriched. It is possible to give several specifications separated by comma. Then the union of these specifications is enriched. After the keyword with the new signature, induction principles, and axioms are added. The enriched specification ends with the keyword end enrich.

Actualizations

Instances of generic specifications like sets of numbers, lists of booleans, or arrays of lists of elements (the instance may again be generic!) are created by actualization. An actualization consists of three parts: First, a parameter specification to be instantiated. This is in the most simple case a generic specification, but may also be every other specification containing a parameter. Second, the parameter is instantiated by one or more actual specifications. The actual specification may be parameterized.

The third part is a mapping called morphism between the parameterized and the actual specification. It must fulfill several requirements that are demonstrated with the following example, the actualization of the elements of the generic arrays with booleans:

Parameter In most cases, each parameter sort and operation of the parameterized specification is mapped to sorts and operations of the actual specification. In our example the parameter sort
'elem' is mapped to the sort 'bool'. The variables a, b, c must also be mapped to boolean variables. Their names can be new (i.e. not existing in the bool specification).

Parameter sorts and operations of the parameterized specification which are not mapped to something are not actualized. They form -- together with the parameter of the actual specification -- the parameter of the instance. In the example below there are no such sorts. An example is the actualization of only one of two parameters (e.g. of pairs).

It’s obvious that in an actualization of a parameter sort by a sort of the actual specification all operation symbols of the parameter containing this sort as argument or target sort must also be mapped to an operation symbol and vice versa. Furthermore it’s possible to actualize several parameter sorts by the same sort.

**Target** The target sorts and operations of the parameterized specification can – but need not be – renamed during the actualization. A renaming must be injective, the results of a renaming may not be contained in the actual specification.

The syntax of an actualization starts with the keyword **actualize** followed by the parameterized specification. After the keyword **with**, a list of actual specifications (separated by comma), and the keyword **by morphism**, a list (separated by ‘;’) of renamings of sorts and operations in arbitrary order follows. A renaming consists of the symbol to be renamed, the implication symbol ‘→’ (or ‘->’), and the renamed symbol. Usually infix, prefix, and postfix declarations, and priorities are inherited by the renamed symbol. The default can be changed by adding dots in front of or behind the symbol and by a priority declaration. A ‘prio 0’ declaration creates an ‘ordinary’ (non-infix, non-prefix, non-postfix) symbol.

In this example arrays are actualized by booleans:

```plaintext
actualize array-spec
with bool-spec
by morphism
  ( : parameter :)
  elem → bool; a → boolvar; b → boolvar0; c → boolvar1;
  ( : No target renamings in this actualization :)
end actualize
```

The sort mapping automatically changes the types of the array operations, e.g. `. [ . ] : array × nat → elem` is changed to `.[. ] : array × nat → bool`.

**Renamings**

The sorts and operations of a specification may be renamed by an injective morphism. A renaming specification starts with the keyword **rename** and the specification whose sorts or operations should be renamed. After by **morphism** a list of renamings follows as described in the case of actualization. The specification ends with **end rename**.

**Data Type Specifications**

Often the constructors of a sort have the property that two different constructors always yield two different objects, and that each application of a constructor yields a new object. Data type specifications provide a convenient way to specify such sorts and their constructors. More specifically, they are useful if:

- All syntactically different terms built up from the constructors denote different values (The constructors are the constants and functions contained in the generated by clause. If a term contains parameter variables it is sufficient that there exists an arbitrary variable assignment where the terms are different). Such data types are called ‘freely generated’. An example of freely generated datatypes are lists with the two constructors nil (empty list) and cons (add an element at the beginning of a list) for lists.
• There exists for each constructor and each of its arguments a function selecting the corresponding argument (in case of lists these are the functions car and cdr selecting the first element or the rest of a list respectively).

Data types fulfilling these requirements can be described in KIV by data type specifications. Data type specifications may also be generic. A data type specification starts with the keyword \texttt{data specification} or \texttt{generic data specification}. In the second case a parameter specification follows after the keyword \texttt{parameter}. An optional list of used specifications may be given (starting with the \texttt{using} keyword, separated by ','). The parameter of the whole specification is the union of the parameters of all subspecifications which is in the second case enriched by the explicitly defined parameter (this means that a specification may be generic even without an explicit parameter). Now some \texttt{data type declarations} follow. They define some mutually recursive defined sorts (in most cases only one sort is declared so that there is only one data type declaration). Each data type declaration consists of the name of one sort. After the equals sign some \texttt{constructor declarations} (separated by '|') follow, which end with a ';'. Each constructor declaration starts with the constructor. If the constructor is not a constant a list of \texttt{selector declarations} separated by ';' follows in parentheses. Each selector declaration is of the form 'selector name : argument sort'. A constructor declaration declares a constructor whose target sort is the sort of the data type declaration and whose argument sorts correspond to the sorts of the selector declaration. The selectors select the corresponding arguments. Each constructor declaration may be followed by the keyword \texttt{with} and a predicate to test if a data was generated by this constructor.

The data type declaration is followed by a declaration of variables as in other types of specifications. Furthermore there are two (optional) abbreviated definitions. The first begins with the keyword \texttt{size functions}. Size functions may only be declared if the systems own specification of naturals is (directly or indirectly) contained in the \texttt{using} list. A size function must be unary, the argument sort must be declared in a data type declaration, and the target sort must be 'nat'. For such a function axioms are created that count the number of constructors of the data. Constructors that are constants are not counted (so the size function returns the usual length in case of lists).

The second abbreviated declaration \texttt{order predicates} contains binary predicates describing the ‘is subterm of’ relation. The corresponding axioms are generated automatically. These predicates are always noetherian and may therefore be used for induction. In case of lists the order ‘is an endpiece of’ is generated. Together with an ‘emptyp’ predicate testing if a list is empty results the following specification of lists which is based on the specifications ‘elem-spec’ for the elements and ‘nat-spec’ for the naturals:

\begin{verbatim}
generic data specification
parameter elem-spec using nat
list = [] with emptyp
 | . + . (. .first : elem; . .rest : list);

variables x, y, z : list;
size functions # . : list → nat;
order predicates . < . : list × list;
end generic data specification
\end{verbatim}

Note the infix declaration of + (overloaded with addition on natural numbers) and the postfix declarations of the selectors .first and .rest.

The axioms generated can be divided in the following groups (as shown for the example of lists):

• ‘Generated by’ clause: This clause declares that the sorts contained in the data type specifications are generated by all of its constructors: list \texttt{generated by} [], +
CHAPTER 5. ALGEBRAIC SPECIFICATIONS

- Injectivity of constructors: \(a + x = b + y \iff a = b \land x = y\)
- Uniqueness (or distinctiveness) of constructors: \(a + x \neq []\)
- Selector axioms: \((a + x).\text{first} = a, (a + x).\text{rest} = x\)
- Test predicates: emptyp([]), \(\neg\) emptyp(a + x)
- Size functions: \(# [] = 0, # (a + x) = # x + 1\)
- Order predicates: \(\neg x < x, x < y \land y < z \rightarrow x < z, x < a + y \leftrightarrow x = y \lor x < y, \neg x < \text{nil}\)

Actually, some more useful theorems are generated.

Basic and generic data specifications are a very common type of specification. Here are a few examples for other data types which can be described by data specifications:

- **enumeration types** like booleans, or days of the weeks:

  \[
  \text{data specification} \\
  \text{day} = \text{Mon} | \text{Tue} | \text{Wed} | \text{Thu} | \text{Fri} | \text{Sat} | \text{Sun}; \\
  \text{variables} \ da : \text{day}; \\
  \text{end data specification}
  \]

- **tupels** (or synonym records) like pairs:

  \[
  \text{generic data specification} \\
  \text{parameter elem12-spec} \\
  \text{pair} = . \times . ( . .1 : \text{elem1}; . .2 : \text{elem2} ); \\
  \text{variables} \ p : \text{pair}; \\
  \text{end data specification}
  \]

- **variant records**:

  \[
  \text{data specification} \\
  \text{using name-spec} \\
  \text{literature} = \text{mkbook} ( \text{author} : \text{name}; \text{title} : \text{name} )
  \| \text{mkarticle} (\text{author} : \text{name}; \text{title} : \text{name}; \text{booktitle} : \text{name}); \\
  \text{variables} \ l : \text{literature}; \\
  \text{end data specification}
  \]

- **natural numbers**. This specification is the most basic recursive data type specification and is predefined in KIV:

  \[
  \text{data specification} \\
  \text{nat} = 0 | . +1 ( . -1 : \text{nat}); \\
  \text{variables} \ m, n : \text{nat}; \\
  \text{order predicates} \ . < . : \text{nat,nat}; \\
  \text{end data specification}
  \]
5.2 The Specification Methodology in KIV

This section provides some hints about the methodology that is used to specify single data types or small groups of data types in KIV. It does not describe how to divide a large specification into its components and should be considered only as a quick starter.

We assume that an informal description of a data type is given. In most cases it is useful to proceed as follows:

1. Determine all sorts, functions, and predicates.

2. Determine the constructors which generate the elements of the data types. The result is written in the ‘generated by’ clauses and determines in most cases how the specification hierarchy should be built. Sorts having no constructors should be entered in a parameter specification (no computer is able to generate data elements without constructors). In real systems typical parameters are e.g. names (if you don’t want to decide at an early time that they are strings, numbers, . . .).

3. The next step is to decide in which case two terms generated by the constructors are equal. If they are only equal if the terms are identical it’s the simple case of a data specification (which may be enriched by some functions). Otherwise it is in most cases possible to write down an axiom of extensionality of the form \( d_1 = d_2 \leftrightarrow \varphi \) describing when two datas \( d_1 \) and \( d_2 \) are equal. The operations contained in \( \varphi \) are normally selectors which form together with the constructors the core of a specification. All other things should be added by enrichments.

4. The selectors should be defined recursively by the constructors. Here you often need case distinctions which should be complete (every case is treated) and sound (if cases are contained twice the definitions should be equal).

5. At the end all other operations needed should be defined. It is recommended to use separate enrichments for operations not directly depending on each other (especially if the data type is complex). There are several possibilities recommended to define new operations:

   - **nonrecursive definition:** In this case a new predicate \( p \) or function symbol \( f \) is defined by a form \( p(x) \leftrightarrow \varphi \) or \( f(x) = t \) respectively where \( p \) or \( f \) are not contained in \( \varphi \) or \( t \). Case distinctions are possible. If case distinctions are sound (see above) they do not lead to inconsistencies.

   - **recursive definition by constructors:** The principle of definition remains the same as above and there is a complete case distinction containing all constructors of \( x \). That is, each occurrence of \( x \) is replaced by \( c(x_1, \ldots, x_n) \) for each constructor \( c \). The restriction ‘\( p \) or \( f \) are not contained in’ is replaced by ‘\( p \) or \( f \) is only applied to \( x_1, \ldots, x_n \).’

   - **definition of modification functions by selector operations:** In this method the effect of modification operations is defined by their effect on the selectors used in the axiom of extensionality.

As an example of how to use this methodology we describe the development of the specification of arrays:

1. Three sorts are involved in the specification of arrays: the arrays itself, the index sort, and the sort of the elements. It is obvious that the element sort is a parameter sort and the array sort is a target sort. In case of the index sort there are two possibilities: If natural numbers are to be used as an index sort (easier) it must be a target sort. If other enumerations types should also be allowed to be the index, it has to be chosen as a parameter sort. Nevertheless, a first element and a successor function is needed (to iterate over the array) so that the specification will roughly describe natural numbers. We choose the first solution.
2. As constructors a function ‘mkarray : nat → array’ is needed to create an array of size \( n \). To create an array with an arbitrary content we need a modification operation ‘. [ . ] : array × nat × elem → array’. \( ar[n,a] \) changes the content of the array \( ar \) at the position \( n \) into \( a \) if \( n \) is smaller than the size of the array. Therefore the generation clause is ‘array \textit{generated by} mkarray, [ ]’.

3. Do two different terms built only by mkarray and put always differ? The answer is no because the arrays \( ar[n,a] \) and \( ar[n,a][n,a] \) (read as \( (ar[n,a])[n,a] \)) are equal. According to the methodology you should try to write down an axiom of extensionality. To do this a selector ‘. [ . ] : array × nat → elem’ (lookup) is needed. Furthermore arrays of different size have to be distinguished and so a length function ‘# : array → nat’ has to be defined. Now it is possible to formulate the axiom of extensionality as
\[
ar_1 = ar_2 \iff # ar_1 = # ar_2 \land \forall n. n < # ar_1 \rightarrow ar_1[n] = ar_2[n].
\]

4. The selectors ‘lookup’ and ‘length’ should be defined recursively for the constructors of an array. For the function ‘#’ it is easy: ‘# mkarray(n) = n, # ar[n,a] = # ar’. (But note that we \textit{need} the axiom that updating an array does not change the length of the array by some queer side effect.) In the case of ‘lookup’ there are two possibilities to handle the problem of ‘violation of array bounds’. These possibilities exist in each case where it is not clear which result a function returns on a certain input value (in programming languages these problem lead to overflows, . . .):

- **underspecification**: In this case you simply do \textit{not} specify the result value. That means that the function may return on a critical input value \textit{any} result. Because \textit{nothing} can be induced from this fact, it’s impossible to prove the correctness of a program containing such a function call. (Well, at least for normal programs.) This method is chosen in data type specifications if selectors are applied on ‘wrong’ constructors (e.g. selecting the first element of an empty list).

- **error and default elements**: This possibility is chosen if it makes sense that programs (or functions) using this data type are calculating with “error” results. Typical examples are specifications of algorithms for searching, pattern matching, or unification where failures are possible. In this case the concerned sort has to be enriched by an error constant which also generates an element of the data type (and must therefore be contained in the generation principle). Adding a new error constant often causes problems because all other operations must be able to deal with it.

For example in the case of lists the constant ‘errlist’ may added to the list sort to specify ‘[[].rest = errlist’. In this case it’s necessary to specify \( a + \text{errlist} = \text{errlist} \) (as errlist), too. Furthermore the axiom \( (a + x).\text{rest} = x \) is only valid for \( x \neq \text{errlist} \). . . . . Therefore it is often easier to use existing constants as ‘default’ and specify ‘[].\text{rest} = \text{rest}’.

In case of parameter sorts it may be useful to define a default constant ‘noelem: elem’. The advantage of this solution is that arrays need not be initialized explicitly. Instead, for every actualization of the elements by concrete datas (e.g. nat) a default constant has to be given which is used to initialize arrays. An even more flexible solution is to add a further argument of the sort elem to the operation mkarray which is used to initialize the array.

However, the two possibilities to define ‘lookup’ on an index value beyond the length of the array remain.

5. Additional functions for arrays are e.g. copying parts of one array into another, selecting a subarray, filling part of an array with a given element, comparing parts of two arrays etc. These functions are typically specified in one (or more) enrichments.
5.3 Creating Specifications in KIV

When a project is selected the system switches to the project level which is displayed by a graph visualisation tool.

Each project consists of specifications (and modules which contain the implementations of the software system) whose dependencies form an acyclic graph. Rectangles correspond to specifications, rhombs correspond to modules.

Each specification is located in its own subdirectory specs/⟨name⟩ in the project directory. The specification text itself is contained in the file specification. A specification is created with the command Specification – Create (or right-click – Create Specification). Then you have to choose the name and type of the specification. In case of generic, enriched, union, actualized, or renamed specifications the used subspecifications have to be selected. After creation of the specification, the system actualizes the specification graph.

A created specification containing no text is displayed in white in the development graph. The specification text has to be typed in using the Eclipse Plugin. A specification can be edited by clicking on the corresponding node in the development graph with the mouse button and choosing Edit in the opening menu. A newly created specification already contains a template of the specification text with some commands to load the used subspecifications.

When the text of the specification is completed it can be installed. That means that the specification is loaded, its syntactical correctness is checked and an internal data structure is generated. To install a specification use the Install command which you get by clicking on the corresponding node in the development graph with the mouse button and choosing Install in the opening menu. If there are some errors in the specification a window is displayed showing the errors. Note that in case of parser errors the two question marks are displayed immediately in front of the token that cannot be parsed. Frequent errors are missing spaces between tokens (‘a=b’ is one token, not an equation!), signature conflicts within the project, and type checking errors (multiple defined symbols, invalid or incomplete morphisms, . . . ). When an error occurs you can leave the error window open, continue editing the file, and try to install it once more by selecting Yes in the error window. If a specification is successfully installed the node in the development graph changes its color to blue.

Note that you can change the type of the specification or the used specifications simply by editing the file (e.g. by replacing basic specification with enrich). You do not have to delete and re-create the specification.

Finally you can work on the specification using the work on . . . command. If a specification has to be changed the Reload command in the menu of the specification is used. The system reloads the specification and tries to keep theorems already proven valid. If the specification is subspecification of other specifications the system tries to use the changed subspecification instead of the old one. If this is not possible (because a symbol is deleted in the subspecification, an added symbol is missing in a morphism, . . . ) the system tries to reload these specifications, too. If that is not possible the user is asked whether he wants to correct these specifications, to abort the whole action, or to leave the specifications that are currently not loadable invalid. Invalid specifications have a red color.

When all theorems of a specification are proven the specification can be set into Proved State. Specifications which are in proved state are shown as green nodes. When a specification enters the proved state all dependencies between the proofs in this specification and used lemmas from subspecifications are checked. In particular it is checked that all simplifier rules used in the proofs do still exist in the subspecifications (see below).

Short description of the menu commands

The following operations can be performed on a specification by either clicking on the specification and choosing the corresponding menu item or by choosing Specification – . . .:

- **Install**: Install a specification. The text of the specification is loaded. The color of the corresponding rectangle changes to blue.
• **Uninstall**: Uninstall a specification. The color of the corresponding rectangle changes to white.

  This is a dangerous operation because it deletes all theorems and proofs!

• **Reload**: Reload a specification if the specification text has changed.

• **Edit**: Edit the text of the specification using the Eclipse Plugin. (It is possible to use another editor, too.) The syntax of specifications is described in chapter 5.

• **View**: View the text of a specification.

• **Work on . . .**: Work on a specification. This command is used to change to the specification level.

• **Rename**: Rename a specification. The name is changed in all other specifications referring to the specification to be renamed, too.

• **Delete**: Delete the specification.

• **Delete edges**: Delete edges in the development graph. The corresponding files are also changed.

• **Print**: Print the text of a specification as a LATEX document.

• **Enter Proved State**: Enter into the proved state. The system checks if all dependencies are fulfilled and if there are no cyclic proofs. If a specification is in the proved state it is not possible to make any changes to it.

• **Leave Proved State**: Leave the proved state.

If a command is not contained in the popup menu it is not applicable on the specification. E.g.

you can only delete specifications that are uninstalled, and you can only uninstall a specification if all its super specifications are already uninstalled.

To leave the project level use the command **Edit – Go Back**.
CHAPTER 5. ALGEBRAIC SPECIFICATIONS

5.4 Exercise 3

All specifications for this exercise should be developed in advance with pencil and paper. The project for this exercise contains only some specifications of natural number from the KIV library of data types (with orange colour).

**Exercise 3.1** Generic specification of sets

Specify finite sets with arbitrary parameter elements. (Note that the element specification must be created first.) Define a constant for the empty set, an operation to insert an element into a set, and a predicate to test if an element is contained in a set. Do not specify other operations.

Prove that an element is a member of a set with only one element if and only if the two elements are equal. The theorem should be formulated in such a manner that it can be used as a simplifier rule.

**Exercise 3.2** Enrichment by a union operation

Enrich the specification of sets by a function that unites two sets.

**Exercise 3.3** Data type specification of binary trees

Define a data type binary trees in a generic data type specification. The data type should have the same parameter elements in its nodes as the sets. Leaves have no elements.

Define an order predicate < for binary trees.

**Exercise 3.4** Complete binary trees

Enrich your specification of binary trees with a predicate that states that the tree is complete (every leaf in the tree has the same depth).

Use your specification of binary trees and the given specification nat-pot (for natural numbers) as foundation for your enrichment.

Hints:

• You will need to specify some other functions on natural numbers and/or on trees.

• Use those functions to specify the predicate for complete binary trees.

• The depth of a leaf is 0.

**Exercise 3.5** Properties of complete binary trees

1. Prove that the number of leaves in a complete binary tree \( bt \) is \( 2^{\text{depth}(bt)} \).

2. Prove that the number of nodes (including leaves) in a complete binary tree \( bt \) is \( 2^{\text{depth}(bt)+1} - 1 \).

Hint: Of course you have to specify functions for counting nodes and leaves first.

**Exercise 3.6** Actualization of sets by trees

Specify sets of binary trees (including the union operation on sets) by actualizing the union specification with the specification of binary trees.

**Exercise 3.7** Enrichment of sets of binary trees by a subtree operation

Specify a function \( \text{subtrees} \) that return the set of all subtrees of a given tree (including the tree itself).

**Exercise 3.8** Prove the injectivity of \( \text{subtrees} \)

Prove that \( \text{subtrees} \) is injective, i.e. two trees are equal if and only if they have the same subtrees. A little hint: You should formulate a lemma that connects the order predicate for binary trees with \( \text{subtrees} \).
Sometimes it is very easy to write inconsistent specifications that allow to 'prove' everything. The following two examples demonstrate this.

**Exercise 3.9** Inconsistent specifications (1)

Enrich the *subtrees* specification from the last exercise with the following specification:

- **functions** \( \text{min} : \text{set} \rightarrow \text{bintree}; \)
- **axioms**
  
  \[
  \text{min-def} : \text{min}(s) = t \iff t \in s \land \forall t0. \ t0 \in s \rightarrow t < t0 \lor t = t0;
  \]

The idea is to specify a function \( \text{min} \) that selects the smallest element of a set of binary trees. (Here we assume that your sorts are named \( \text{set} \) and \( \text{bintree} \), and that \( s \) is a variable for sets and \( t \) and \( t0 \) variables for trees, and that \( < \) is the order predicate for the binary trees. You have to modify the names to fit your specifications.)

Show that this specification is inconsistent by proving \( \vdash \text{false} \). (You can ignore the warning about a sub signature when adding this theorem.)

**Exercise 3.10** Inconsistent specifications (2)

Another try: Enrich the *subtrees* specification a second time with the specification

- **functions** \( \text{min} : \text{set} \rightarrow \text{bintree}; \)
- **axioms**
  
  \[
  \text{min-in} : s \neq \emptyset \rightarrow \text{min}(s) \in s;
  \]

Show that this specification is also inconsistent by proving \( \vdash \text{false} \).
Chapter 6

Hoare’s Proof Strategy

6.1 Introduction

The Hoare logic ([Hoa69], [LS80]) is a well known approach to verifying “small” programs. It was developed during the late sixties and allows to formulate properties for the partial correctness of while programs. These properties can then be verified using the Hoare calculus. Originally it was not possible to express and verify termination properties. This drawback has been mended in different enhancements later on (e.g. [LS80]). In this chapter we will use a small imperative programming language with while loops. The next chapter will extend the programming language with local variables and recursive procedures.

The next section gives an overview on the proof method of Hoare. The syntax and semantics of programs is defined formally in Section 6.3. This section also defines Dynamic Logic, which extends predicate logic formulas with two new formulas that contain programs. The new formulas can be used to express Hoare triples in the sequent calculus of KIV as described in Section 6.4. This section also gives the original Hoare rules, and the rules used in KIV. Section 6.5 gives an example proof. Section 6.6 describes the strategy used in KIV to normalize goals (all predicate logic is moved to the end of the antecedent). Finally, Section 6.7 describes the heuristics for symbolic execution, that can be used to automate proofs.

6.2 The Proof Strategy

A Hoare formula has three parts:

precondition: \( \varphi \) (formula in predicate logic)
program: \( \alpha \)
postcondition: \( \psi \) (formula in predicate logic)

notation:

\[ \{ \varphi \} \alpha \{ \psi \} \]

Its interpretation is as follows: if precondition \( \varphi \) holds and program \( \alpha \) terminates, then postcondition \( \psi \) holds after \( \alpha \) has been executed.

In order to prove, that a specification \( SPEC \) implies a Hoare formula

\[ \{ \varphi \} \alpha \{ \psi \}, \]

i.e.

\[ SPEC \models \{ \varphi \} \alpha \{ \psi \} \]
holds, proof rules are used. For each program construct exactly one proof rule exists. These rules make up the core of the Hoare calculus. They combine logical and algebraic properties of used data with properties of the program structure. The rules do not depend on an interpretation of the data type, but are valid in general.

The proof strategy is as follows: Proof rules are used to reduce the program of a given Hoare formula $\{ \varphi \} \alpha \{ \psi \}$ to simpler programs, until formulas in pure predicate logic result. Proving these predicate logical sub goals then is sufficient for the correctness of the overall Hoare formula relative to the chosen interpretation. Because of this strategy, the Hoare calculus is also called a VCG (Verification Condition Generator) approach in program verification.

In the following, programs are defined as relations between initial and final state. The notation for Hoare formulas is slightly different in the Dynamic Logic of KIV. A Hoare formula $\{ \varphi \} \alpha \{ \psi \}$ will be written as $\varphi \vdash \alpha \psi$.

### 6.3 Syntax and Semantics of Programs

#### Syntax

The set of programs PROG is defined as being the smallest possible set for which the following holds:

- $\mathit{skip} \in \text{PROG}$ (empty program)
- $\mathit{abort} \in \text{PROG}$ (never terminating program)
- $x := t \in \text{PROG}$ (parallel/random assignment), where $x \in X_s$ and $t \in T_s \cup \{?\}$
- if $\varepsilon$ then $\alpha$ else $\beta \in \text{PROG}$ (conditional), where $\alpha, \beta \in \text{PROG}$, $\varepsilon \in \text{BXP}$
- $\{ \alpha; \beta \} \in \text{PROG}$ (composition), where $\alpha, \beta \in \text{PROG}$
- while $\varepsilon$ do $\alpha \in \text{PROG}$ (loop), where $\alpha \in \text{PROG}$ and $\varepsilon \in \text{BXP}$

Remarks:

1. Superfluous $\{ \ldots \}$ blocks are omitted.
2. if $\varepsilon$ then $\alpha$ else skip is abbreviated as if $\varepsilon$ then $\alpha$
3. skip is equivalent to $x := x$
4. abort is equivalent to while true do skip

#### Semantics

Given an algebra $\mathcal{A}$ and states $z$ and $z'$ the semantics of a program $z[\alpha]z'$ is defined as follows:

- $z[\mathit{skip}]z'$, iff (if and only if) $z = z'$
- $z[\mathit{abort}]$ is false for every pair of states (empty relation)
- $z[x := t]z'$, iff $z' = z[x := [t]_z]$, where $[?]_z$ represents an arbitrary value
- $z[\{ \alpha; \beta \}]z'$, iff there is a $z''$ such that $z[\alpha]z''$ and $z''[\beta]z'$
- $z[\mathit{if} \varepsilon \mathit{then} \alpha \mathit{else} \beta]z'$, iff $[\varepsilon]_z = \mathit{tt}$ and $z[\alpha]z'$, or $[\varepsilon]_z \neq \mathit{tt}$ and $z[\beta]z'$
- $z[\mathit{while} \varepsilon \mathit{do} \alpha]z'$, iff there is a finite sequence of states $z_0, \ldots, z_n (n \geq 0)$ such that

$z[\alpha]z'$ is a relation on states in infix notation. It depends on the algebra $\mathcal{A}$, so we write $z[\alpha]_\mathcal{A}z'$, if we want to emphasize the dependency.
– $z = z_0$, $z' = z_n$.
– $[e]_{z_0} = tt = \ldots = [e]_{z_{n-1}} = tt$, $[e]_{z_n} = ff$, and
– $z_i[α] z_{i+1}$ for each $i$ with $0 \leq i < n$

($z_i$ is the state after the $i$th repetition of the loop)

Semantics of Formulas in Dynamic Logic

The semantics of formulas in Dynamic Logic is similar to the semantics of formulas in predicate logic with the following additional two constructs.

• $[[α] ϕ]_z = tt$, iff for all states $z'$, $z[α] z'$ implies $[[ϕ]'_z = tt$ gilt. (The different $z'$ represent the reachable final states of program $α$.)

• $[[⟨α⟩ ϕ]_z = tt$, iff there is a state $z'$ for which $z[α] z'$ and $[[ϕ]'_z = tt$ hold. (This implies, that program $α$ terminates, i.e. there is a reachable final state $z'$.)

Program Equivalence

Two programs $α, β$ are equivalent, short $α \equiv β$, iff their semantics are the same in all models, i.e. for all $A \in Alg(Σ)$ $[α]_A = [β]_A$ holds.

Remarks on Dynamic Logic

• The semantics of Dynamic Logic does not evaluate program formulas “statically”, using a fixed variable setting, but “dynamically”, depending on the location within other dynamic logic formulas. The initial states may result from the execution of other programs. Therefore the term “Dynamic Logic” is used.

• $[[α] ϕ]_z = tt$ is always true, if $α$ does not terminate. Thus $[α] ϕ$ models partial correctness. Contrary to this $⟨α⟩ ϕ$ is always false, if $α$ does not terminate.

• Formula $⟨α⟩ ϕ \leftrightarrow ¬[α] ¬ϕ$ is valid. Therefore diamonds can be defined using boxes and vice versa.

• If $α \equiv β$, then for all $ϕ$, $⟨α⟩ ϕ \leftrightarrow ⟨β⟩ ϕ$ is valid.

• If $x_1, \ldots, x_n$ are the variables of $α$ and/or $β$, and $y_1, \ldots, y_n$ are different variables, then the validity of $⟨α⟩ (x_1 = y_1 \land \ldots \land x_n = y_n)$ $\leftrightarrow ⟨β⟩ (x_1 = y_1 \land \ldots \land x_n = y_n)$ implies $α \equiv β$

• The last two remarks show, that program equivalence can be expressed in Dynamic Logic.

6.4 The Proof Rules

In the following, the rules of Hoare calculus are explained. First a general translation of Hoare notation into Dynamic Logic is given. Afterwards all Hoare rules—written as sequent calculus rules—which are currently available in KIV are listed. In general more than one calculus rule for a program construct exists. The additional rules allow to partly overcome the disadvantages of a pure VCG: trivial subgoals are omitted.
Hoare Formulas in KIV

Hoare formulas can be expressed in Dynamic Logic as follows:

**Hoare formula**

\( \{ \varphi \} \alpha \{ \psi \} \)

**sequent**

\( \varphi \vdash [\alpha] \psi \)

In Dynamic Logic \( \psi \) may also contain programs.

Hoare Rules in KIV

**Assignment**

assign (Hoare)

\( \{ \varphi \} x := \tau \{ \varphi \} \)

If a formula \( \varphi \) holds for (free instances of) variables \( x \) being substituted with term \( \tau \), then \( \varphi \) holds after the assignment has been executed.

assign right (DL)

\[
\begin{align*}
\Gamma \vdash \varphi \Delta' & \quad \Gamma \vdash [x := \tau] \varphi \Delta \\
\Gamma \vdash [x := \tau] \varphi & \quad \Gamma \vdash \Delta' \backslash \varphi, \Delta
\end{align*}
\]

These rules are not axioms as with Hoare, because the *consequence rule* of Hoare has been integrated. The first variant of the rule is used if it is possible to substitute \( x \) with \( \tau \) in \( \varphi \) (e.g., if \( \varphi \) is a predicate logic formula). Otherwise the second variant is used, where \( x' \) is a new variable.

The name of the rule is extended with *right* because this rule is applicable, if an assignment occurs in the succedent of a sequent.

**Conditionals**

if-then-else (Hoare)

\( \{ \varphi \land \varepsilon \} \alpha \{ \psi \} \quad \{ \varphi \land \neg \varepsilon \} \beta \{ \psi \} \quad \{ \varphi \} \text{if} \varepsilon \text{then} \alpha \text{else} \beta \{ \psi \} \)

- If \( \varepsilon \) holds \( \alpha \) is executed.
- If \( \neg \varepsilon \) holds \( \beta \) is executed.

if right (DL)

\[
\begin{align*}
\Gamma, \varepsilon \vdash [\alpha] \varphi \Delta & \quad \Gamma, \neg \varepsilon \vdash [\beta] \varphi \Delta \\
\Gamma \vdash \text{if} \varepsilon \text{then} \alpha \text{else} \beta \varphi \Delta
\end{align*}
\]

if positive right (DL)

\[
\begin{align*}
\Gamma \vdash \varepsilon & \quad \Gamma \vdash [\alpha] \varphi \Delta \\
\Gamma \vdash \text{if} \varepsilon \text{then} \alpha \text{else} \beta \varphi \Delta
\end{align*}
\]

Applying this rule only makes sense if \( \Gamma \rightarrow \varepsilon \) is provable.

if negative right (DL)

\[
\begin{align*}
\Gamma \vdash \neg \varepsilon & \quad \Gamma \vdash [\beta] \varphi \Delta \\
\Gamma \vdash \text{if} \varepsilon \text{then} \alpha \text{else} \beta \varphi \Delta
\end{align*}
\]

Applying this rule only makes sense if \( \Gamma \rightarrow \neg \varepsilon \) is provable.
Loops

while (Hoare)

\[ \{ \text{Inv} \land \varepsilon \} \text{while } \varepsilon \text{ do } \{ \text{Inv} \land \neg \varepsilon \} \]

- Inv is called loop invariant.
- If \( \varepsilon \) holds, the body of the loop is executed and Inv must hold afterwards.
- If \( \neg \varepsilon \) holds, the body of the loop is not executed and Inv \( \land \neg \varepsilon \) is trivially true, because Inv is a precondition.

**Invariant Right (DL)**

\[
\begin{align*}
\Gamma \vdash \text{Inv} & , \Delta \\
\Gamma, \varepsilon \vdash [\alpha] \text{Inv} & , \varepsilon \not\vdash \varphi \\
\Gamma \vdash \text{while } \varepsilon \text{ do } \alpha \varphi & , \Delta
\end{align*}
\]

- Inv is called loop invariant.
- The consequence rule has been integrated into this rule.

**While Right (DL)**

\[
\begin{align*}
\Gamma, \varepsilon \vdash [\alpha] & \text{while } \varepsilon \text{ do } \alpha \varphi, \Delta \\
\Gamma, \neg \varepsilon \not\vdash \varphi, \Delta
\end{align*}
\]

This rule can be used if an invariant is not needed, e.g. if the loop is executed exactly three times. The next two rules are special cases of this rule (comparable to if vs. if positive/negative).

**While Unwind Right (DL)**

\[
\begin{align*}
\Gamma \vdash \varepsilon & \\
\Gamma \vdash [\alpha] \text{while } \varepsilon \text{ do } \alpha \varphi, \Delta
\end{align*}
\]

This rule is applicable if the rule test \( \varepsilon \) is provable before executing the loop. In this special case the body of the loop can be executed once instead of giving an invariant.

**While Exit Right (DL)**

\[
\begin{align*}
\Gamma \vdash \neg \varepsilon & \\
\Gamma \vdash \varphi, \Delta
\end{align*}
\]

If the rule test \( \varepsilon \) does not hold the loop is exited.

Remark: The rules while right, while unwind right and while exit right are special cases of the invariant rule invariant right. It is possible to only use invariant right, but with the additional rules proofs may become shorter.

**Composition**

**Composition (Hoare)**

\[
\begin{align*}
\{ \varphi \} & \alpha \{ \chi \} \\
\{ \chi \} & \beta \{ \psi \}
\end{align*}
\]

For this rule an intermediate assertion \( \chi \) has to be guessed. In order to avoid this, program formulas are normalized in the sequent calculus of KIV.

**Normalize (DL)**

\[
\begin{align*}
\Gamma \vdash [\alpha] & [\beta] \varphi, \Delta \\
\Gamma \vdash [\alpha; \beta] \varphi, \Delta
\end{align*}
\]

This rule is automatically applied after each rule application. Therefore it is not a calculus rule on its own.
Rule normalize offers two possibilities to approach program verification:

**forward:** A sequent of program constructs is executed from left to right, i.e. a rule application simplifies the first construct of a program.

**backward:** A sequent of program constructs is executed from right to left, i.e. a rule application simplifies the last construct of a program.

Here, the forward direction is preferred, because it follows the intuition of symbolically executing programs. As a disadvantage of this strategy a slightly more complicated assignment rule is needed which is capable of substituting variables in programs (see above).

**Consequence**

\[ \varphi \rightarrow \psi \quad \{ \psi \} \alpha \{ \chi \} \quad \chi \rightarrow \delta \]

\[ \{ \varphi \} \alpha \{ \delta \} \]

In KIV a consequence rule is not explicitly provided, since the predicate logic rules are available already (with simplifier, case distinction etc.).

**Additional Rules**

In DL we additionally use the skip program that does nothing, and occasionally the abort program that never terminates.

**skip right (DL)**

\[ \Gamma \vdash \varphi, \Delta \]

\[ \Gamma \vdash [\text{skip}] \varphi, \Delta \]

**abort right (DL)**

\[ (\text{abort}) \varphi, \Gamma \vdash \Delta \]

### 6.5 An Example Proof

We will try to prove the partial correctness of a program that implements the function square on natural numbers by repeated addition.

\{0 \leq n \land m = 0\}

\{
  k := 0;
  while k < n do
  {
    k := k + 1;
    m := m + n;
  }
\}

\{m = n \times n\}

This Hoare formula is expressed in Dynamic Logic as follows:

\[
\{0 \leq n \land m = 0\}
\[
\{k := 0;
\}

while k < n do
{
...
CHAPTER 6. HOARE’S PROOF STRATEGY

\[ k := k + 1; \]
\[ m := m + n \]
\] \[ m = n \times n \]

This formula can be found in project Exercise4 as the theorem \texttt{1-example} in the specification \texttt{hoare}. (This specification is just an empty enrichment, and used only to store our theorems/exercises.) In order to trace all subgoals of rule applications, choose Control/Options and select option “No automatic predtest”. This option prevents simple predicate logical subgoals from disappearing.

We begin with the normalised sequent

\[ 0 \leq n, m = 0 \]
\[ \vdash \]
\[ [k := 0;] \]
\[ \text{while } k < n \text{ do} \]
\[ \{ \]
\[ k := k + 1; \]
\[ m := m + n \]
\[ \} \]
\[ m = n \times n \]

The first program statement is an assignment. Applying rule \texttt{assign right} results in the following sequent:

\[ 0 \leq n, m = 0, k = 0 \]
\[ \vdash \]
\[ [k := 0;] \]
\[ \text{while } k < n \text{ do} \]
\[ \{ \]
\[ k := k + 1; \]
\[ m := m + n \]
\[ \} \]
\[ m = n \times n \]

Next, rule \texttt{invariant right} is applied. This rule requires an invariant. The appropriate invariant is \( m = k \times n \land k \leq n \). It can be entered in concrete syntax as \( m = k \times n \) and \( k \leq n \). You may wonder why we need the condition \( k \leq n \) – we will see later.

Applying \texttt{invariant right} yields three subgoals:

1. \( 0 \leq n, m = 0, k = 0, m \neq k \times n \lor \neg k \leq n \vdash \)
2. \( k < n, m = k \times n, k \leq n \vdash [k := k + 1] \]
\[ [m := m + n] (m = k \times n \land k \leq n) \]
3. \( \neg k < n, m = k \times n, k \leq n, m \neq n \times n \vdash \)

Remark: Goal 1 and goal 3 differ from the first and the third premise of the invariant rule in that all formulas appear in the antecedent of the sequent. This is due to the ‘normalization’ that keeps all predicate logic formulas in the antecedent for program proofs.

The intention of the three goals of rule \texttt{invariant right} is as follows: Partial correctness of a while loop is guaranteed if the loop invariant

- holds at the beginning of a loop (goal 1),
- holds after each execution of the loop’s body (goal 2), and
- implies the postcondition after the while loop has terminated (goal 3).

We are now trying to prove goal 1. The sequent of this goal no longer contains program constructs because the application of rules of the Hoare calculus resulted in a predicate logic subgoal. In order to prove this goal, rule \texttt{simplifier} is applied. This rule automatically closes goal 1.

Goal 2 still contains two assignments which can be eliminated by applying rule \texttt{assign right} twice. The first application leads to
\[ k < n, m = k \cdot n, k \leq n \vdash [m := m + n] (m = (k + 1) \cdot n \wedge k + 1 \leq n) \]

the second application results in
\[ k < n, m = k \cdot n, k \leq n, m + n \neq (k + 1) \cdot n \lor \neg (k + 1) \leq n \vdash \]

Now also goal 2 can be closed by applying rule simplifier.

Similar to goal 1, goal 3 only contains formulas in predicate logic. Applying rule simplifier also closes this last goal. However, here the condition \( k \leq n \) is essential, because we need to know that \( k = n \). The negated loop condition only gives the information \( n < k \). The information that \( k \) is incremented only by 1 so that afterwards \( k = n \) must be explicitly stated in the invariant by \( k \leq n \).

Remark: If option ‘No automatic predtest’ is not selected, subgoals in pure predicate logic which can be automatically closed with the help of the simplifier are not generated at all. The simplifier is internally applied. In this example, rule invariant right produces only goal 2.

### 6.6 Normalization

When proving properties of programs the predicate logic formulas form the context of the current goal. Experience shows that a user ignores them most of the time and concentrates on the program formulas. In order to help this concentration, sequents containing program formulas are ordered in a special manner.

1. The first part of the antecedent contains only program formulas.
2. An optional induction hypothesis follows.
3. The rest of the antecedent contains all(!) predicate logic formulas of the sequent.
4. The succedent contains only program formulas.

Predicate logic formulas are kept in the antecedent even if this means that negations are not resolved. The rule case distinction is applicable also on negated conjunctions etc. For program formulas, negations are resolved in the usual manner. This ordering is introduced automatically when beginning a new proof (this may lead to a first proof step called normalize that can never appear anywhere else in a proof), and kept intact automatically during a proof (without visible proof steps).

The normalization also deals with compounds. (It may have been noted that no proof rule for compounds exists.) A formula \( (\alpha; \beta) \varphi \) is equivalent to \( (\alpha) (\beta) \varphi \). Using this equivalence, compounds are eliminated.

### 6.7 Heuristics

When proving properties using the Hoare strategy, the following choice of heuristics are of interest:

- **symbolic execution:**
  The following noncritical rules are automatically applied on the first program construct in the succedent:
  - assign right
  - if positive right
  - if negative right
  - skip right

- **loop exit:**
  If it can be automatically shown, that the loop test is false, rule while exit right is applied.
• **unwind:**
  If a loop test evaluates to true, the body of the loop is executed once (rule *while unwind right*).

• **simplifier:**
  This heuristic decides, if rule *simplifier* should be applied, e.g. if predicate logical formulas have changed.

• **conditional right split:**
  This heuristic tries to apply rule *if right* on the first formula in the succedent.

Heuristics are selected by choosing *Control/Heuristics*. An additional window is displayed which allows to

• select a predefined set of heuristics (all heuristics above are included in the set of heuristics called *DL Heuristics + Case Splitting*),

• deactivate all heuristics, or

• arrange an individual set of heuristics.

In the last case, a window with two columns is displayed, listing all available heuristics on the left and all selected heuristics on the right. A heuristic is selected by clicking on its name. The order of selection determines the application order of heuristics. By pressing ‘OK’, the heuristics are activated and are automatically applied on the current goal.
6.8 Exercise 4, Part 1

Task: Prove the Hoare formulas listed below with KIV. They are formulated as theorems in specification hoare in the project “Exercise4”. For each exercise a corresponding theorem exists.

Exercise 4.1 example
Redo the example described above.

Exercise 4.2 sum
The following program calculates \( z = \sum_{x=0}^{n} x \). Prove the following theorem:

\[
\begin{align*}
    z &= 0 \land m = 0 \\
    \vdash & \quad [ \text{while } m \leq n \text{ do} \\
            & \quad \{ \\
            & \quad \quad z := z + m; \\
            & \quad \quad m := m + 1 \\
            & \quad \} ] \\
    & \quad z = \frac{n * (n + 1)}{2}
\end{align*}
\]

Exercise 4.3 modulo
The following program calculates the remainder of a division of natural numbers \( n \mod m \).
The original input to the program is stored in variables \( n_0 \) and \( m_0 \). The result of the calculation is contained in \( n \).

\[
\begin{align*}
    n &= n_0 \land m = m_0 \\
    \vdash [ \text{while } m \leq n \text{ do} & \\
            & \quad n := n - m & \quad (\exists k. k \cdot m_0 + n = n_0 \land n < m_0)
\end{align*}
\]

Additionally try to prove \textit{3-modul-termination}. The theorem is the same as for modulo, but now with diamonds instead of boxes (meaning you have to prove termination now). When using the ‘invariant right’ rule, you will now additionally have to give a decreasing term.

If your proof attempt fails think about the reason and change the theorem accordingly. Then prove the corrected theorem.

Exercise 4.4 binary search
Given is an array \( ar \) of some totally ordered elements with values stored between the two indices \( mi \) (minimum) and \( ma \) (maximum). The values in the array are sorted in ascending order. Additionally an element \( searched \) is given, which is contained in the array between the two given indices. The program computes the index of the element \( searched \) by applying a binary search strategy. The correctness of the program is to be verified with the help of the standard set of heuristics.

\[
\begin{align*}
    mi \leq ma & \land \text{sor} < (ar, mi, ma) \land \exists n. ar[n] = searched \land mi \leq n \land n \leq ma \\
    \vdash & \quad [ \text{lower} := mi; \\
    & \quad \quad upper := ma; \\
    \text{while } lower \neq upper \text{ do} \\
    & \quad \quad \{ \\
    & \quad \quad \quad \text{middle} := (lower + upper) / 2; \\
    & \quad \quad \quad \text{if } ar[\text{middle}] < searched \\
    & \quad \quad \quad \text{then } lower := \text{middle} + 1 \\
    & \quad \quad \quad \text{else } upper := \text{middle};
\end{align*}
\]
(ar[lower] = searched ∧ mi ≤ lower ∧ lower ≤ ma)

$\text{ar}[n]$ selects the element stored in array $\text{ar}$ with index $n$. $\text{sor}< (\text{ar}, \text{mi}, \text{ma})$ is a predicate which is true if and only if the elements between the indices $\text{mi}$ and $\text{ma}$ of array $\text{ar}$ are sorted in ascending order (relative to an order predicate $<$). Predicate $\text{sor}< \text{is defined in specification } oarray$. Note: Properties, i.e. theorems, of this specification must be used (by insert rewrite lemma or insert spec-lemma).
Chapter 7

Procedures and Local Variables

In this chapter we extend Dynamic Logic with procedures and local variables. Their syntax and semantics are given is defined in the first four sections. Section 7.5 gives an example proof. The last two subsections give new symbolic executions rules and describe new heuristics for unfolding calls.

7.1 Syntax of Procedure Declarations and Calls

Besides the program statements that were used in the first Hoare exercise (like while and if) it is also possible to use procedures and local variables. Procedures have to be declared globally in specifications:

predicates . . .
procedures $p_1$ $s_{1,1}, s_{1,2}, \ldots, s_{1,n_1}; s'_{1,1}, s'_{1,2}, \ldots, s'_{1,m_1}$;
$p_2$ $s_{2,1}, s_{2,2}, \ldots, s_{2,n_2}; s'_{2,1}, s'_{2,2}, \ldots, s'_{2,m_2}$;
$\vdots$
p$k$ $s_{k,1}, s_{k,2}, \ldots, s_{k,n_k}; s'_{k,1}, s'_{k,2}, \ldots, s'_{k,m_k}$;
variables . . .
$\vdots$
axioms . . .
declaration $p_1$ $(x_{1,1}, x_{1,2}, \ldots, x_{1,n_1}; y_{1,1}, y_{1,2}, \ldots, y_{1,m_1}) \{ \alpha_1 \}$
p$2$ $(x_{2,1}, x_{2,2}, \ldots, x_{2,n_2}; y_{2,1}, y_{2,2}, \ldots, y_{2,m_2}) \{ \alpha_2 \}$
$\vdots$
p$k$ $(x_{k,1}, x_{k,2}, \ldots, x_{k,n_k}; y_{k,1}, y_{k,2}, \ldots, y_{k,m_k}) \{ \alpha_k \}$

The procedures part defines procedure symbols as an additional part of the signature. The declaration part gives procedure declarations for each of the procedures $p_1, \ldots, p_k$ (which may be mutual recursive). $p_1$ has the value parameters $x_{1,1}, x_{1,2}, \ldots, x_{1,n_1}$ of sorts $s_{1,1}, s_{1,2}, \ldots, s_{1,n_1}$ and the reference parameters $y_{1,1}, y_{1,2}, \ldots, y_{1,m_1}$ of sorts $s'_{1,1}, s'_{1,2}, \ldots, s'_{1,m_1}$. All these parameters must be different variables. The semantics of value and reference parameters is input and output parameters. The reference parameters can be used as input/output parameters by adding the keyword nonfunctional at the end of the signature definition. Otherwise any procedure body, which does not write the reference parameters, or reads some parameter before writing it, will be rejected. Similarly, the signature definition must contain the keyword indeterministic at the end of the signature definition. Otherwise any procedure body helps KIV to have more efficient symbolic execution rules for procedures. There
are no nested procedure declarations. A procedure call of \( p_1 \) has the form

\[
p_1(\sigma_{1,1}, \sigma_{1,2}, \ldots, \sigma_{1,n_1}; z_{1,1}, z_{1,2}, \ldots, z_{1,m_1})
\]

\( \sigma_{1,1}, \ldots, \sigma_{1,n_1} \) are arbitrary terms (or variables), \( z_{1,1}, \ldots, z_{1,m_1} \) must be pairwise different variables. Typically for every procedure defined in the procedures section a declaration is given. But this is not necessary. It is also possible to leave away the declaration and to just write axioms about procedure calls (indeed, a procedure declaration can be translated to axioms that specify the procedure semantics unambiguously).

### 7.2 Syntax of Local Variable Declarations

Local variable declarations have the form

\[
\text{let } x_1 = t_1, \ldots, x_n = t_n \text{ in } \{ \alpha \}
\]

The local variables \( x_1, \ldots, x_n \) are defined with \( \alpha \) as its scope. It is not necessary to give a type (as in Java) because they must have a signature entry already which fixes their type (it is not possible to use a local variable with different types in different places). Initial values \( t_1, \ldots, t_n \) are mandatory to avoid undefined values. The initial values are evaluated in parallel, i.e. \( x_j \) used inside of \( \sigma_i \) references the global value of \( x_j \), not \( t_j \).

### 7.3 Semantics of Procedures

To define the semantics of procedures we add a new component to signatures: \( \Sigma = (S, OP, P) \) where \( P = \bigcup_{\exists s \in S, \exists s' \in S} P_{s \leftrightarrow s'} \) is the of procedure names with value parameters of sorts \( s \) and reference parameters of sorts \( s' \) (procedure names are used in programs). A \( \Sigma \)-Algebra now contains for every \( p \in P_{s \leftrightarrow s'} \) a relation \( p_A \) on \( A_s \leftrightarrow A_{s'} \). The idea is that \( (a, b, c) \in p_A \) iff the procedure started in a state \( z \) where value parameters have values \( a \) and reference parameters have values \( b \) will give a final state where reference parameters have values \( c \). If the procedure does not terminate in state \( z \) with, then there will be no \( c \) in the relation. For a functional procedure (the usual case), the result does not depend on \( b \). For a deterministic procedure (the usual case again), there is at most one value \( c \) for every \( (a, b) \). The semantics of a procedure call therefore is:

\[
z[\left[ p(l; u) \right]] z' \iff z(l), z(u), z'(u) \in [p] \text{ and } z(y) = z'(y) \text{ for all } y \notin u.
\]

A procedure declaration \( p(x; y) \{ \alpha \} \) roughly states, that the semantics of a procedure call is equivalent to the semantics of the body

\[
z[\left[ p(x; y) \right]] z' \iff z[\alpha] z'
\]

Dynamic Logic can express this semantic equivalence as program equivalence:

\[
\langle p(x; y) \rangle \ y = z \leftrightarrow \langle \alpha \rangle \ y = z
\]

This equivalence is used in the call rule for procedures.

\( ^1 \)for recursive procedure declarations this definition is too weak, and the semantics must be given as the least fixpoint of procedure calls with bounded depth. We ignore this problem here
7.4 Semantics of Local Variables

The semantics of a local variable declaration \( \text{let } x = t \text{ in } \alpha \) is simpler. Given an initial state \( z \) the definition first evaluates terms \( t \) in the current state to give the initial values \( a \) for the local variables. Then it overwrites the values of \( x \) in \( z \) with \( a \) and runs \( \alpha \) in the resulting state. If \( \alpha \) terminates and gives a final state \( z' \), the local variables must finally be deallocated. To do this, they are replaced with the initial values \( z(x) \) of the global variables to give the final state \( z' \) of the whole program. Formally

\[
\begin{align*}
    z[\text{let } x = t \text{ in } \alpha]z' & \text{ iff } z[x ← a][\alpha]z'' \quad \text{and} \quad z'' = z''[x ← z'(x)] \text{ where } a_i = [t_i]z,
\end{align*}
\]

7.5 An Example Proof

This section gives an example for a proof involving a recursive procedure. The proof can be found in specification `hoare-proc` of the specifications for this exercise. The procedure checks the membership of a natural number in a list of natural numbers. The procedure is defined in KIV as follows:

\[
\begin{align*}
    \text{member#}(n, nats; \text{var boolvar}) & \\
    \text{if } nats = \[] & \text{ then boolvar := false} \\
    \text{else if } n = nats . \text{first} & \text{ then boolvar := true} \\
    \text{else member#}(n, nats . \text{rest}; \text{boolvar});
\end{align*}
\]

The signature of the procedure consists of three parameters, two input parameters and one output parameter. The input parameters are \( n \) and \( nats \), where \( n \) is a natural number and \( nats \) is a list of natural numbers. The output parameter is named \( boolvar \) and is of type bool. When the procedure is called, it is first determined, whether the list of natural numbers is empty. If this is so, the result is set to false, as no number is element in the empty list. If the list of natural numbers is non-empty, the number \( n \) is checked against the first member of the list. If \( n \) and the first member of the list are equal, the MEMBER# function terminates with the result value true, otherwise, MEMBER# is called recursively.

We expect \( boolvar \) to be true after termination, iff the list \( nats \) contains the number \( n \). We want to prove this property in KIV. To do so, we formulate a lemma in KIV with the following form:

\[
\vdash \langle \text{MEMBER#}(n, nats; \text{boolvar}) \rangle \text{ boolvar} \leftrightarrow n \in nats;
\]

To verify this lemma, induction is required, since the procedure is recursive. We choose structural induction for the list \( nats \), since the recursive call has an argument, which is one constructor smaller than the original argument (if in general, the recursive argument is smaller in some well-founded order, then induction instead of structural induction must be used). The basis of this induction is rather simple. As \( nats \) is the empty list, it can be evaluated, that \( n \in nats \) is false. Therefore, the proof obligation is simply to verify, that MEMBER# evaluates to false for the empty list:

\[
\vdash \langle \text{MEMBER#}(n, \[]; \text{boolvar}) \rangle \neg \text{boolvar}
\]

With the application of the rule call left, the MEMBER# function call is replaced by the procedure body:
\[ \begin{align*}
\text{if } \text{[]} = \text{[]} & \text{ then boolvar0 := false} \\
\text{else if } n = \text{[]}.\text{first} & \text{ then boolvar0 := true} \\
\text{else member#(n, [].\text{rest}; boolvar0)} & \text{ boolvar0}
\end{align*} \]

The conditional can be decided using the \textit{if positive} rule. This leaves the assignment on the antecedent of the sequent:

\[ \vdash \langle \text{ boolvar0 := false} \rangle \neg \text{ boolvar0} \]

Using the \textit{assign} rule, this goal can be closed. For the inductive case, the following goal results after simplification:

\[ \forall n. \langle \text{MEMBER#}(n, \text{nats0}; \text{boolvar}) \rangle (\text{boolvar} \leftrightarrow n \in \text{nats0}) \]

Note that the KIV library of lists has simplifier rules, that normalize \( m + \text{nats0} \) (where \(+\) nat \times natlist \rightarrow natlist) to \( m' + \text{nats0} \), where \(+\) appends lists and the quote ('') operator creates a one element list from an element (this allows to use associativity of append more often).

The idea for a proof is to apply rules, such that the procedure call in the induction hypothesis (with argument \text{nats}) can be matched to the procedure call in the sequent. Since the current call with \( m' + \text{nats0} \) has a recursive call in its body with arguments \text{nats0}, this can be done by symbolically executing the procedure MEMBER#.

The first step to do this is to apply the \textit{call right} rule, which again replaces the call with the body. The first test \( m' + \text{nats0} = \[] \) is false, so \textit{if negative right} can be applied and gives

\[ \forall n. \langle \text{MEMBER#}(n, \text{nats0}; \text{boolvar}) \rangle (\text{boolvar} \leftrightarrow n \in \text{nats0}) \]

Applying \textit{if right} gives two goals. The first, where \( m = n \) can be closed with \textit{assign right}. The second goal is

\[ \forall n. \langle \text{MEMBER#}(n, \text{nats0}; \text{boolvar}) \rangle (\text{boolvar} \leftrightarrow n \in \text{nats0}), n \neq m \]

The formula on the right now is almost the induction hypothesis. Instantiating the quantifier with \( n \) and discarding it (since we only need this one instance) gives:

\[ \langle \text{MEMBER#}(n, \text{nats0}; \text{boolvar}) \rangle (\text{boolvar} \leftrightarrow n \in \text{nats0}) \]

Now symbolic execution of the two formulas does not help (it will only lead deeper into the recursion). We need a method to reduce a goal of the form

\[ \Gamma, \langle \alpha \rangle \varphi \vdash (\alpha) \psi, \Delta \]
to a goal that $\varphi$ implies $\psi$. The execute call rule does this reduction. If both sides are procedure calls, then this rule allows different output parameters in the calls in the antecedent and succedent (they can be renamed to be equal). The rule gives a slightly stronger premise than just the implication:

$$\Gamma, \langle \alpha \rangle (x = y), \varphi_{y/x} \vdash \psi_{y/x}, \Delta$$

It says that the implication must be proved only for values $y$, which are possible results of executing $\alpha$. The first formula of the antecedent says that $y$ are such values: $x$ are the variables that are modified by $\alpha$ (if $\alpha$ is a call, then these are just the output parameters of the call). $y$ are new variables, which store the resulting values. Note that KIV automatically applies execute call as soon as heuristics for DL goals are used.

Applying execute call closes the proof, since the resulting predicate logic goal can be closed by the simplifier.

### 7.6 Functional Specification of Procedures

Instead of giving an implementation for a procedure, KIV also allows a functional specification of what the call does, leaving the implementation open. An example for such a specification is NEXTTOKEN#:

```plaintext
nextok-spec:
nats = nats0
\vdash (NEXTTOKEN#(;nats,tok))
   ( (tok = error \to nats = nats0)
     \\land (tok \neq error \to toString(tok) \neq [] \land toString(tok) + nats = nats0))
```

The idea of NEXTTOKEN# is to do lexical analysis in a parser: given a string (for simplicity we represent the string as a list of ASCII codes, i.e. natural numbers). NEXTTOKEN# searches for an occurrence of some token at the start of the string (if the string is a Java program read from a file, example tokens are key words like "'try'" or "'if'"). The specification of tokens abstracts from the exact nature of tokens (in a real parser token would be numbered), it just defines a sort of tokens and a function `toString` which a string representation, and that there is a special token `error` (a constant). The `error` token is used as the result of NEXTTOKEN#, when the input string does not start with a token. In this case the first clause of the postcondition says, that the string `nats` should be returned unchanged (`nats` is used as input and output, `tok` is used as output only; `nats0` is used to save the initial value of `nats`). Otherwise the second clause says, that the token should have a nonempty string representation, and the string should have been shortened by the string representation of the token$^2$.

Since the implementation of NEXTTOKEN# is left open, symbolic execution of this function is not possible. Instead it is necessary to insert the lemma `nextok-spec`, whenever NEXTTOKEN# is used in a sequent, and use rules execute call (see the example proof above) or contract call. The latter rule is similar to execute call, but it assumes that both calls are in the antecedent.

$^2$this is a simplification: in reality, the string would also be shortened by additional spaces, so the equality would be modulo spacing
CHAPTER 7. PROCEDURES AND LOCAL VARIABLES

7.7 Rules for Procedures and Local Variables

This section documents the rules for procedures and local variables. The name given to each rule is the name used in KIV. The actual implementation of a rule might differ in some details from its representation. Especially we try to introduce as less new variables and equations as possible. Those simplifications, however, should allow the user to choose a rule — it is the purpose of the system to handle these technical details.

Almost all rules are also applicable for box formulas instead of diamond formulas.

Almost all rules are applicable not only on the first formula in the antecedent or succedent but on a formula at an arbitrary position.

call right

\[
\frac{y = σ, \Gamma \vdash \langle α \overrightarrow{x} \rangle φ, Δ}{\Gamma \vdash \langle f(σ, x) \rangle φ, Δ}
\]

for bounded procedure blocks:

\[
\frac{\Gamma \vdash \langle α \overrightarrow{x} \rangle φ, Δ}{\Gamma \vdash \langle f(σ, x) \rangle φ, Δ + 1, Δ} \quad \frac{\Gamma \vdash \langle \text{abort} \rangle φ, Δ}{y = σ, \Gamma \vdash \langle f(σ, x) \rangle φ, Δ + 1, Δ}
\]

- The specification contains a declaration \( f(y, z).α \)
- Variables \( y, z \) of the declaration are not free in the sequent (otherwise they are renamed)
- The counter \( κ \) specifies the number of calls allowed at most.
- Rule applicable on boxes too

call left

\[
\frac{y = σ, \langle α \overrightarrow{x} \rangle φ, Γ \vdash Δ}{\langle f(σ, x) \rangle φ, \Gamma \vdash Δ}
\]

for bounded procedure blocks:

\[
\frac{y = σ, \langle α \overrightarrow{x} \rangle φ, Γ \vdash Δ}{\langle f(σ, x) \rangle φ, Γ \vdash Δ + 1, Δ} \quad \frac{\langle \text{abort} \rangle φ, Γ \vdash Δ}{\langle f(σ, x) \rangle φ, Γ \vdash Δ + 1, Δ}
\]

- The specification contains a declaration \( f(y, z).α \)
- Variables \( y, z \) of the declaration are not free in the sequent (otherwise they are renamed)
- The counter \( κ \) specifies the number of calls allowed at most.
- Rule applicable on boxes too

evaluate call

\[
\frac{\Gamma \vdash x = z \quad \langle f(σ; x) \rangle \langle x = y \rangle, φ \vdash ψ}{\langle f(σ; x) \rangle φ, Γ \vdash \langle f(σ; z) \rangle ψ}
\]

- \( y \) are new variables
- the procedure \( f \) is functional
- the rule is also applicable for non-functional procedures with the additional condition \( Γ \vdash x = z \)
CHAPTER 7. PROCEDURES AND LOCAL VARIABLES

**contract call left**

\[
\frac{\Gamma \vdash \sigma = \tau \quad \langle f(\sigma, z) \rangle (z = x'), \psi_{x'}, \psi_{x}, \Gamma \vdash \Delta}{\langle f(\sigma, z) \rangle \varphi, \langle f(\tau, y) \rangle \psi, \Gamma \vdash \Delta}
\]

- \(z, x\) are new variables
- the procedure \(f\) is functional
- the rule is also applicable for non-functional procedures with the additional condition \(\Gamma \vdash x = y\)

**contract call right**

\[
\frac{\Gamma \vdash \sigma = \tau \quad \langle f(\sigma, z) \rangle (z = x'), \Gamma \vdash \psi_{x'}, \psi_{x}, \Delta}{\Gamma \vdash \langle f(\sigma, z) \rangle \varphi, \langle f(\tau, y) \rangle \psi, \Delta}
\]

- \(z, x\) are new variables
- the procedure \(f\) is functional
- the rule is also applicable for non-functional procedures with the additional condition \(\Gamma \vdash x = y\)

**vardecls right**

\[
\frac{x' = \tau, \Gamma \vdash (\alpha_{x'}) \varphi}{\Gamma \vdash (\text{let } x = \tau \text{ in } \alpha) \varphi}
\]

\[
\frac{\Gamma \vdash \exists x'. (\alpha_{x'}) \varphi}{\Gamma \vdash (\text{let } x = ? \text{ in } \alpha) \varphi}
\]

- \(x = ?\) is a random variable declaration
- both rules are integrated into one
- for boxes replace the \(\exists\) by a \(\forall\)

**vardecls left**

\[
\frac{(\alpha_{x'}) \varphi, x' = \tau, \Gamma \vdash \Delta}{\langle \text{let } x = \tau \text{ in } \alpha \rangle \varphi, \Gamma \vdash \Delta}
\]

\[
\frac{\exists x'. (\alpha_{x'}) \varphi, \Gamma \vdash \Delta}{\langle \text{let } x = ? \text{ in } \alpha \rangle \varphi, \Gamma \vdash \Delta}
\]

- \(x = ?\) is a random variable declaration
- both rules are integrated into one
- Rule applicable on boxes too

**split left/split right**

\[
\frac{(\alpha) z = x', \psi_{x'}, \Gamma \vdash \Delta}{(\alpha) \varphi, \Gamma \vdash \Delta}
\]

\[
\frac{\Gamma \vdash (\alpha) \text{true}}{(\alpha) z = x', \Gamma \vdash \psi_{x'}, \Delta}
\]

- \(z = \text{asgvars}(\alpha)\)
- \(x\) are new variables
7.8 Heuristics for Procedures

Predefined Sets of Heuristics

Heuristics are used to decrease the number of user interactions. In the best case they find proofs automatically. Even if a proof is not found completely automatically heuristics can help the user by applying often used, mechanical rules automatically. Ideally, they find a proof completely automatic. But there is also a risk when using heuristics: They may apply steps which are unnecessary or unwanted or may even lead to infinite loops. When selecting heuristics, the following (partly antagonistic) issues have to be considered.

1. Efficiency
   Since the prover works interactively, the heuristics cannot take too much time, especially in cases where they can’t do anything at all. When large goals appear, usefulness vs. efficiency of a heuristic is a real issue.

2. Probability of wrong decisions
   A wrong decision (e.g. making a case distinction when none is necessary, or instantiating and dropping a quantifier) can lead to unprovable goals or unnecessary work for the user. Or it can make the proofs harder. A wrong decision normally means that the heuristic is deselected and the proof tree pruned.

   The more complicated the proof is, the higher the probability of wrong decisions becomes. This means, that fewer heuristics should be selected.

Unfortunately (or luckily?) the universal problem solving heuristic hasn’t been published yet. Therefore KIV offers quite a large number of heuristics (ranging from general but “dumb” to very special but “intelligent”, and from established and useful to experimental and doubtful). An expert user can combine the heuristics in a manner that is most useful for the given problem.

During a proof heuristics can be enabled and disabled at any time (using the command Control – Heuristics). Therefore heuristics can be used for a single branch of the proof or for the whole proof. An important property of heuristics is that there is an arbitrary hierarchical ordering: If a heuristic can’t be applied the next heuristic according to the hierarchical ordering is tested. Furthermore a heuristic can determine which other heuristics should be applied next. Therefore heuristics are a very flexible concept.

For example, there is a heuristic that just tests if there is an assignment, a conditional or a variable declaration. If this is the case, the corresponding rule is applied. Another heuristic only executes procedure calls and leaves the other work (the symbolic execution of procedure bodies) for other heuristics. There is also a global induction heuristic that only calls other heuristics — depending on the progress of the proof.

To choose a good set of heuristics some experience is needed. To simplify matters KIV offers three predefined sets of heuristics which are usually sufficient. Therefore after selecting the command Control – Heuristics a window is displayed where a set of heuristics can be chosen:

Let the system choose the heuristics
DL Heuristics + Induction
DL Heuristics + Case Splitting
DL Heuristics
No heuristics at all
Select your own heuristics

Let the system choose the heuristics : KIV chooses one of the predefined sets of heuristics described below. It uses a very simple heuristic to determine the complexity of the goal based on the number and type of program formulas and their connection. The lower the complexity, the more heuristics are chosen. The idea is that the goal complexity has some connection to the difficulty of the proof. The implementation of the programs is not considered (thereby limiting the usefulness of this option).
DL Heuristics + Induction: This set is useful for simple goals that can be proven by simple induction and symbolic execution of programs, and where case distinctions must not be avoided at any cost. This heuristic set often finds proofs automatically for simple recursive programs. The set consists of the following heuristics (in the listed order):

- symbolic execution, conditional right split, conditional left split, contract and execute, split, simplifier, elimination, module specific, weak unfold, Quanti fier closing, loop exit, execute loop, omega, unwind, weak cut, dl case distinction, unfold, induction, apply ind once, batch mode

DL Heuristics + Case Splitting: This set is useful for more complex goals, where nonstandard induction (or no induction at all) is used, or some lemmas are introduced before the induction step, or some procedures shouldn't be unfolded. Compared to the previous set, the induction heuristics (induction, apply ind once) and the strong heuristic for unfolding procedures (leading to the symbolic execution of the procedure bodies) is missing. The remaining heuristics are

- symbolic execution, conditional right split, conditional left split, contract and execute, split, simplifier, elimination, module specific, weak unfold, Quanti fier closing, dl case distinction, omega, unwind, loop exit, execute loop

DL Heuristics: This set is for very complex goals where case distinctions (by splitting conditionals in programs or propositional connectives) have to be avoided at all costs. It uses the heuristics

- symbolic execution, contract and execute, module specific, split, simplifier, elimination, conditional

No heuristics at all: Deselect all heuristics.

Select your own heuristics: This allows to select a different set of heuristics or to modify an existing set. It is possible to select a predefined set of heuristics and this option together.

After selecting this command a window opens containing two columns: on the left side all available heuristics are displayed, on the right side the currently selected heuristics are shown in order of selection. To select a heuristic click on this heuristic on the left side. It appears on the right side at the position of the black bar. To delete a heuristic currently selected click again on the heuristic in the left column. After leaving the window by clicking on the OK–Button the selected heuristics are applied according to their order.

Read from file 'default heuristics': The file 'default-heuristics' should contain a PPL list of heuristics: (list "..."..."..."...). This option is for experts only.

When you begin (or continue) a proof all heuristics are disabled (but still selected). This is indicated in the status bar (by heuristics off). They can be enabled by clicking on the button Heuristics On/Off in the lower left corner of the window.

Description of the Heuristics

This section describes only heuristics dealing with programs.

Procedure calls

weak unfold: This heuristic unfolds procedure call with the rules call left or call right. A procedure call can be safely unfolded if the procedure

1. is not recursive, or
2. it is recursive, but the induction hypothesis will be applied for the recursive call, or
3. it is recursive, but the context is such that the execution path is not leading to a recursive call (i.e. the tests of if’s can be decided).

Unconditionally unfolding a procedure call and executing the body of a recursive procedure usually does not terminate, and has to be avoided. Additionally, the order in which procedures are unfolded and executed (or whether to unfold them at all) can have a significant impact on the size of the proof. If a procedure body does not contain any conditionals, or all conditionals can be decided in a given context, then the execution of the body does not introduce any case distinctions. Furthermore, if the execution path does not lead to a recursive call, it is safe (and normally useful) to unfold the call.

The heuristic weak unfold deals with such procedures. It is called “weak” because it unfolds fewer procedure calls than the following two heuristics.

**unfold** This heuristic unfolds the same calls as weak unfold and some more. It also unfolds calls that lead to case distinctions if one execution path contains a recursive call where the induction hypothesis can be applied. Usually these procedures have to be unfolded to finish the proof. The heuristic keeps track of the procedures and will unfold each call only once. It tries to find the best call to unfold that will minimize the size of the proof. This depends on the statical call hierarchy (one procedure calls another) and the dynamical information flow (the result of one procedure call is the input for another). The heuristic also unfolds nonrecursive procedures.

**strong unfold** This heuristic may unfold recursive procedures several times under certain circumstances. It may also unfold procedures that do not appear in the induction hypothesis. This heuristic should be used with care.

**calls concrete** This heuristic unfolds a call if all actual parameters are concrete, i.e. do not contain variables, unless they are variables of a parameter sort. It uses the assumption that all tests in conditionals can be decided. If this is not the case (e.g. due to missing simplifier rules) the heuristic may lead to non-termination for recursive procedures. Normally, weak unfold is a better choice, so this heuristic should be used only in special circumstances.

**calls nonrecursive** This heuristic unfolds procedures that are not recursive.

**bounded calls** Experimental – don’t use it.

**Symbolic execution**

**symbolic execution** This heuristic is normally the first in the list of chosen heuristics. It executes those program statements that are uncritical. “Uncritical” means that application of the appropriate rule does not introduce case distinctions and will not lead to non-termination. This means it does not deal with calls, loops, and if’s with an undecidable test. It will only look at the first formula in the antecedent and the succedent. (The idea is to execute a program until it is finished or a critical statement is reached. This means that assignments etc. normally only appear in the first formula.) The following rules will be applied if possible:

- `abort right, abort left, skip right, skip left, assign right, assign left, vardecls right, vardecls left, if positive right, if negative right, if positive left, if negative left`

**split** This heuristic tries to apply the rule split left anywhere in the antecedent. This is an uncritical heuristic and can be always used.
CHAPTER 7. PROCEDURES AND LOCAL VARIABLES

conditional  This heuristic tries to apply the rules \textit{if positive right}, \textit{if negative right}, \textit{if positive left}, and \textit{if negative left} anywhere in a sequent. In most proofs, conditionals only appear at the first position in the antecedent or succedent where the heuristic \textit{symbolic execution} deals with them. However, they may appear anywhere if the heuristics \textit{conditional left/right split} are not used. This heuristic may need some time if a goal contains several conditionals.

conditional right split  This heuristic applies the rule \textit{if right} on every formula in the succedent if possible.

conditional left split  This heuristic applies the rule \textit{if left} on every formula in the succedent if possible.

contract and execute  This heuristic applies the rules \textit{execute call}, \textit{contract call left}, and \textit{contract call right} if the actual parameters of the calls are identical, and no termination goal is introduced.

Case distinctions

pl case distinction  This heuristic will do nothing for goals containing programs. This means that case distinctions for predicate logic formulas have to be done by hand.

dl case distinction  This heuristic resolves propositional connectives between program formulas by introducing case distinctions.

7.9 Exercise 4, Part 2

Exercise 4.5 difference

Program a function DIFFERENCE\# (in the declaration section of specification \texttt{hoare-proc}, which takes two lists as arguments and returns a list, which contains all elements of the first list that are not in the second list. Use a recursive procedure, not a while loop. Do not use predicate $\in$ in the program, but call MEMBER\# (see the example proof). Verify that your implementation of difference follows this specification (using predicate $\in$ on lists). In particular, it should always terminate.

Exercise 4.6 quicksort

Verify the termination of QUICKSORT\#.

\textbf{Hint:} Use well-founded induction on list length, and a suitable correctness lemma for DIVIDE\#.

Exercise 4.7 parsing

A function PARSE\# is implemented in specification \texttt{hoare-proc}, which repeatedly calls NEXT-TOKEN\# and finally returns a list of tokens, and a rest of the initial input, that could not be parsed. Verify the termination and functional correctness of the parse-function PARSE\#. Functional correctness for PARSE\# means, that the string representations of the resulting tokens can be appended with the restlist such that the original list is restored.

\textbf{Hint:} An additional recursive function tostring is needed, which turns a list of tokens into a string.
Chapter 8

Contracts, Abstract Data Types and Refinement

In this exercise we will have a look at the problem, how to get from an abstract specification to executable code. This is a general problem of Software Engineering, independent of whether formal methods are used or not. There are many proposals, what to use for informal specifications (e.g. the various diagrams of UML) and how to proceed (e.g. Unified Process). Almost all methods based on abstract models propose to construct more concrete models incrementally.

When formal methods are used, one step from a more abstract to a more concrete specification is called a refinement step. The main question for such a step is: Is the concrete model guaranteed to meet the requirements given in the abstract specification, is the refinement correct? Correctness of a refinement should imply that certain properties that were true for the abstract specification are also true for the concrete specification. What the properties that should be preserved are (functional properties, safety properties, security properties, termination etc.) often depends on the application and the specification language. Therefore there are many formal definitions of refinement.

We will look at one, that assumes that an abstract data type is specified as a number of contracts for procedures (or methods), and that we want the implementation to preserve inputs and outputs that were specified in these contracts. The next section defines contracts. Based on these abstract data types and their semantics are defined.

8.1 Contracts

A contract for a procedure consists of

- A name
- A definition of the input and output variables of a procedure (these may overlap)
- A precondition $\varphi$
- A postcondition $\psi$
- Some informal text specifying the purpose of the procedure

A procedure satisfies a contract, if the procedure terminates on any input that satisfies $\varphi$ and yields output that satisfies $\psi$ (total correctness). In our formal specification language, the specification of a contract requires specifications of data types to be used as the type of variables, and predicate logic formulas $\varphi$ and $\psi$. A simple example is the contract for a reverse function:

- Name: reverse#
- Input: x of type list
• Output: \( x \)

• Precondition: \( x = y \)

• Postcondition: \( x = \text{reverse}(y) \)

This definition assumes a data type of lists and that a function \( \text{reverse} \) on it has been given. The name \( \text{reverse} \) follows the usual KIV convention to have procedure names that end with \( \# \). The pre- and postcondition use the program variable \( x \) and the auxiliary variable \( y \) which is not used in the implementation. Such a variable is often called a logical variable.

Another example is the following contract for a dequeue operation:

• Name: \( \text{dequeue} \)

• Input: \( q \) of type queue

• Output: the modified queue \( q \) and an element \( o \)

• Precondition: \( q = q_0 \land q_0 \neq \text{empty} \)

• Postcondition: \( q = \text{deq}(q) \land o = \text{hd}(q) \)

A specification of queues is assumed, where \( \text{deq}(q) \) removes the first queue element \( \text{hd}(q) \) and where \( \text{empty} \) is the empty queue.

For formal specification of contracts it is often convenient to merge the pre- and postcondition into a single formula, that describes a relation between initial and final state. This is usually done using a convention that distinguishes between the value of a variable in the state before the procedure is called and in the state at the end. We use the convention of the specification language Z [Spi92]: \( x \) refers to the value of \( x \) in the initial state, and \( x' \) refers to the value at the end. Other conventions are \( \text{pre}(x) \) and \( x \) in VDM [Jon87], or \( \text{old}(x) \) and \( x \) in JML [LBR06], a specification language for Java programs. With our convention the contract above can be specified by the predicate

\[
\text{REV}(x,x') \leftrightarrow \exists y. x = y \land x' = \text{reverse}(y)
\]

Note that this definition was systematically constructed: the quantified variables are all logical variables, and the formula below the quantifier is the conjunction of the preconditon and a modified postcondition, that primes all program variables. The definition can be simplified to

\[
\text{REV}(x,x') \leftrightarrow x' = \text{reverse}(x)
\]

The logical variables always disappear, when the precondition has equations \( x = y \). At first glance, a contract adds little new information to the algebraic specification of the reverse function. But note, that the contract states which variables will be affected: for the contract above, variable \( x \) will be overwritten by the procedure. A contract

\[
\text{REV}(x,x') \leftrightarrow z' = \text{reverse}(x)
\]

would require that \( x \) is unchanged and \( z \) overwritten. The contract

\[
\text{REV}(x,x', z') \leftrightarrow z' = \text{reverse}(x)
\]

allows \( x \) to change arbitrarily.

For the queue example above the relational contract becomes

\[
\text{DEQ}(q,q',o') \leftrightarrow q \neq \text{empty} \land q' = \text{deq}(q) \land o' = \text{hd}(q)
\]
CHAPTER 8. CONTRACTS, ABSTRACT DATA TYPES AND REFINEMENT

For a contract $OP(s,s')$ in relational form the precondition can be recomputed as

$$\text{pre}(OP)(s) := \exists s'. \ OP(s,s')$$

For the dequeue example above, this gives

$$\text{pre}(\text{DEQ})(q) := q \neq \text{empty}$$

8.2 Abstract Data Types

A standard view of a software system is the black box view: the system is defined as the interface it offers to a user, which consists of a number of (system) operations specified by contracts. This definition of a software system encompasses systems of any size: it can be a big system consisting of many components, packages and classes. It can also be a package or a class. Even an object and the methods for a single object in Java would be acceptable as a system.

The system operations all modify a common (system) state, and have inputs and outputs. The system is used in an environment which calls the operations with suitable inputs and receives their outputs. The system state is not directly accessible (information hiding), but only using the operations.

In formal specification, the black box view of a software system is commonly called an abstract data type (which should not be confused with algebraic data types, although algebraic data types are often called abstract data types too).

Definition 1 An abstract data type $DT = (S, I, \{OP_i\}_{i=1 \ldots n})$ consists of

- A set of states $S$.
- A set of initial states $I \subseteq S$.
- A definition of a number of contracts $OP_i$ ($i = 1 \ldots n$) for the methods (= operations) of the data type. Each contract is specified as a relation over some inputs $I_i$, the initial and final state of type $S$, and some outputs $O_i$:

$$OP_1 : I_1 \times S \times S \times O_1$$

In KIV abstract data types are specified using an algebraic data type that defines the set of states. To formally specify an abstract data type in KIV one therefore has to do the following:

- Algebraically specify a data type with sort $S$. Its values are the possible states. Typically the sort $S$ is specified as a tuple type. Each element of a tuple will become the value of one program variable in an implementation.
- Specify a predicate $I$ on sort $S$, which describes the initial states.
- Specify the types $I_i$ and $O_i$ to be used as inputs and outputs
- Define the signature of the contracts as predicates:

$$\text{predicates } OP_1 : I_1 \times S \times S \times O_1$$
$$OP_2 : I_2 \times S \times S \times O_2$$

- Specify the contracts with axioms:

$$OP_1(i,s,s',o) \leftrightarrow \text{<some formula>}>$$
8.3 Example of an Abstract Data Type: Queues

In this section we specify an abstract data type of queues, that will be used in the exercise. The state of a queue consists of a value of type queue. In the exercise, the algebraic type of queues is specified by renaming lists: the empty list is the empty queue, enqueuing an element adds it to the end of the list, dequeuing removes the first element of the list. The three operations on queues are EMP (setting a queue to the empty queue), ENQ and DEQ (enqueuing/dedequeuing an element). Their contracts are:

\[\text{EMP}(q,q') \leftrightarrow q' = \text{empty}\]

\[\text{DEQ}(q,q',\text{an}) \leftrightarrow (q \neq \text{null} \supset q' = \text{deq}(q) \land \text{an} = \text{mk(hd(q))}; q' = q \land \text{an} = \text{none})\]

\[\text{ENQ}(i,q,q') \leftrightarrow q' = \text{enq}(q,i)\]

In contrast to the previous section, the dequeue operation is defined for all queues including the empty queue. Its output is of type \text{elemornone}, which is a free data type specified as:

\begin{verbatim}
data specification
  elemornone = \text{mk(. .elem : elem)} | \text{none};
variables \text{an : elemornone};
end data specification
\end{verbatim}

For an empty queue the operation returns \text{none} and does not change the queue, for a nonempty queue the output is \text{mk(hd(q))}.

8.4 Semantics of Abstract Data Types

An abstract data type is used by starting in an initial state and randomly invoking operations. Starting from an initial state \(s_0 \in I\), calling one of the operations, e.g. \(\text{OP}_{k_1}\) with input \(i_1\) (of suitable type) may change the state to \(s_1\) and gives an output \(o_1\), if \(\text{OP}_{k_1}(i_1,s_0,s_1,o_1)\) holds. A second operation may result in \(\text{OP}_{k_2}(i_2,s_1,s_2,o_2)\) and so on.

To define a semantics it is assumed, that all a user can observe when calling an operation is the inputs he chooses and the output he gets. All changes to the state are invisible for the user (information hiding). A first idea for a formal definition therefore is to define the semantics as all sequences

\[(i_1,k_1,o_1), (i_2,k_2,o_2), \ldots (i_m,k_m,o_m)\]

that result from calling operations as describe above. This has a small problem: Assume the first operation is invoked with a state and an input that violates its precondition. Then according to the specification, the effect is unspecified, and all following operations are allowed to do anything. Therefore the formal definition must leave open what happens after an operation (say \(\text{OP}_{k_p}\) for some \(p \leq m\)) is invoked outside its precondition:

\textbf{Definition 2} (Semantics of Abstract Data Types)

The semantics \(\text{SEM(DT)}\) of a data type is the set of all finite sequences

\[[(i_1,k_1,o_1), (i_2,k_2,o_2), \ldots (i_m,k_m,o_m)]\]

such that

- All \(\text{OP}_{k_j}\) (\(1 \leq j \leq n\)) are operations, i.e. \(k_j \leq n\). The inputs and outputs \(i_j\) and \(o_j\) have suitable types for \(\text{OP}_{k_j}\).
There are suitable intermediate states $s_0, s_1, \ldots, s_m$ and $s_0$ is an initial state.

There is a $1 \leq p \leq m$ such that all operations before step $p$ are invoked within their precondition, and therefore give a result (state and output) that satisfies the contract:

$$\text{OP}_{k_j}(i_j, s_{j-1}, s_j, o_j) \text{ holds for } j < p.$$  

If $p \neq m$ (otherwise all steps are ok), then step $p$ is invoked with an input outside its precondition:

$$\neg \exists s, o. \text{OP}_{k_p}(i_p, s_{p-1}, s, o)$$

The result state $s_p$ and output $o_p$ are arbitrary, as well as all states $s_j$, inputs $i_j$ and outputs $o_j$ for later steps with $j > p$.

8.5 Refinement of Abstract Data Types

The idea of refinement of data types is that a user works with an abstract specification $\text{ADT} = (\text{AS}, \text{AI}, \{\text{AOP}_i\}_{i=1}^{n})$ like the example queue above. He will invoke operations of the data type and can check, whether the outputs he gets when giving some inputs conform to the abstract contracts. The implementation will be a concrete data type $\text{CDT} = (\text{CS}, \text{CI}, \{\text{COP}_i\}_{i=1}^{n})$, which has the same operations, inputs and outputs, but which works with an implementation of the abstract state $\text{AS}$ by a concrete state $\text{CS}$. In our exercise below, the concrete state will consist of a heap of cons cells. The concrete state represents the queue as (the values in) a chain of heap cells.

A refinement of $\text{ADT}$ by $\text{CDT}$ is correct, if the user cannot note the difference. Substituting each call to the abstract operation by a call to the concrete operation gives the same result (this is called the principle of substitutivity). Formally, each sequence $[(i_1, k_1, o_1) (i_2, k_2, o_2), \ldots$ of observations a user can make with $\text{CDT}$ (i.e. that is in the semantics of $\text{CDT}$) must also be possible when using $\text{ADT}$.

**Definition 3** (Refinement of Abstract Data Types)

A refinement of one data type $\text{ADT} = (\text{AS}, \text{AI}, \{\text{AOP}_i\}_{i=1}^{n})$ to another data type $\text{CDT} = (\text{CS}, \text{CI}, \{\text{COP}_i\}_{i=1}^{n})$ is correct, if

- Both data types have the same number $n$ of operations with the same input and output types.

- $\text{SEM}(\text{CDT}) \subseteq \text{SEM}(\text{ADT})$

It is not necessary that all observations the user can make with $\text{ADT}$ are also observations with $\text{CDT}$. This happens in two cases: first, when the abstract operation was unspecified for some input $i$ and state $s$ (so calling it results in any result state $s'$ and any output $o$) the concrete operation is free to return one single well-defined result ($s'$ and $o$). Second, when the abstract specification can return one of several results, the implementation is free to choose one specific result.

As an example consider an abstract specification of sets (the state is of type set) with an operation that chooses an (arbitrary) element from the set. A contract for this operation would be:

$$\text{GET}(s,s',a) \leftrightarrow s' = s \land (s \neq \emptyset \rightarrow a \in s)$$

An implementation could e.g. represent sets by lists (or some more elaborate data structure like a hash table or a balanced tree) and choose some element from the representation, e.g. the first element of the list. The implementation is free to always return the same element (e.g. the minimum if the list is sorted). Its answer may also depend on the internal order, that resulted from inserting the elements (e.g. the implementation could always return the element that was inserted last), even though the abstract set does not have such an internal order.
8.6 Verifying Refinement Correctness

The definition of refinement correctness talks about arbitrary sequences of operations. This is not practical for verification. Instead one would like to verify proof obligations for single operations. To define such proof obligations the usual approach is to define an abstraction relation \( \text{ABS} \) between abstract states \( as \) and concrete states \( cs \). \( \text{ABS} \) is typically a partial function from concrete to abstract states. Its domain are the legal representations (for a queue e.g. the non-cyclical linear pointer structures). Its result is the abstract data structure that it represents. With such an abstraction function it can be shown that the following three conditions are sufficient to verify data refinement.

- **initialization:**
  \[ \text{CI}(cs) \rightarrow \exists \ as. \ \text{ABS}(as,cs) \land \text{AI}(as) \]

- **applicability:**
  \[ \text{ABS}(as,cs) \land \text{pre}(AOP_k)(i,as) \rightarrow \text{pre}(COP_k)(i,cs) \]

- **correctness**
  \[ \text{pre}(AOP_k)(i,as) \land \text{ABS}(as,cs) \land \text{COP_k}(i,cs,cs',o) \rightarrow \exists as'. \ \text{ABS}(as',cs') \land \text{AOP}_k(i,as,as',o) \]

The initialization condition guarantees that each concrete initial state represents some abstract state. The applicability condition guarantees that as long as the precondition of the abstract operations is not violated, the precondition of the concrete operation must hold too. Finally, the correctness condition guarantees, that invoking the concrete operation (with the abstract precondition satisfied) gives an output that the abstract operation can give too. The resulting state \( cs' \) of the concrete operation should be a state that represents the abstract state \( as' \) again. The correctness condition propagates the abstraction relation forwards through operations. If a relation satisfies the conditions it is therefore called a *forward simulation* (some authors prefer the term *downward simulation*). A forward simulation enables an inductive argument over the number of operations applied that formally proves refinement correctness.

**Theorem 1** If the conditions initialization, applicability and correctness can be proved (the latter two for all operations) then the refinement from ADT to CDT is correct.

A simpler form of this theorem was first proved by Hoare and He [HHS86], the theorem here is from [WD96]. These papers also prove that there are cases where forward simulations alone are not sufficient to prove every data refinement: in rare cases one also needs backward simulations. We do not discuss these here, since we will not need them for the exercise.

8.7 Refinements of Contracts to Programs

Data refinement considers the case where a number of operations is implemented by operations on a lower level data structure. In general there are many other types of refinement, e.g. implementing one operation by a sequence of lower level instructions (non-atomic refinement) or refinements where the lower level operations may be interleaved (e.g. when considering several threads on multi-processors). In this section we will consider the case, where the abstract contracts are implemented by programs working on a lower level data structure. This case can be viewed as an instance of data refinement, by viewing a program as specific form of defining a contract. This is possible by allowing dynamic logic formulas as pre- and postconditions. To see this, assume we have specified a contract \( \text{AOP}(i,as,as',o) \) and that we have a procedure

```plaintext
procedures COP#(i; cs,o) nonfunctional;
```

The procedure has input \( i \), modifies a concrete state \( cs \) (therefore it must get the keyword *nonfunctional* in its signature definition) and returns output \( o \). Such a procedure trivially satisfies the following contract:
CHAPTER 8. CONTRACTS, ABSTRACT DATA TYPES AND REFINEMENT

\[ \text{COP}(i,cs,cs',o') \leftrightarrow \langle \text{COP}(i;cs,o') \rangle (cs = cs' \land o = o') \] (1)

The precondition of this contract is just termination of the procedure.

\[ \text{pre}(\text{COP})(i,cs) \leftrightarrow \exists cs',o'. \langle \text{COP}(i;cs,o') \rangle (cs = cs' \land o = o') \leftrightarrow \langle \text{COP}(i;cs,o') \rangle \text{true} \]

This contract can be substituted into the applicability and correctness condition for data refinement. The resulting two conditions can be simplified into one proof obligation which is

\[ \text{ABS}(as,cs) \land \text{pre}(\text{AOP})(i,as) \rightarrow \langle \text{COP}(i;cs,o) \rangle (\exists as'. \text{AOP}(i,as,as',o) \land \text{ABS}(as,cs')) \] (2)

In words: If two states with \( \text{ABS}(cs,as) \) are given, and the contract of the abstract operation guarantees a well-defined result for some input \( i \) (i.e. \( \text{pre}(\text{AOP})(i,as) \) holds), then the concrete operation must terminate and give a state \( cs \) and an output \( o \) after its termination, such that it simulates a possible execution of the abstract operation with the same output \( o \): Some result state \( as' \) of the abstract operation must exist, that is the abstraction of the final state \( cs \).

### 8.8 Nondeterministic Programs

In the exercise below we will use nondeterministic programs to choose new references in the heap (see below). Nondeterminism is necessary, as soon as parallel programs are considered (one typically wants to abstract from scheduling strategies), but it is useful for specification too, e.g. to choose some element of a set, when we do not want to fix a concrete representation of sets on the computer yet. KIV has two constructs for nondeterminism. The first is

\[ \alpha \lor \beta \]

which randomly chooses between executing \( \alpha \) and \( \beta \). The second construct is

\[ \text{choose } x \text{ with } \varphi \text{ in } \alpha \text{ ifnone } \beta \]

It binds local variables \( x \) to arbitrary values that satisfy \( \varphi \) (\( \varphi \) may depend on \( x \) as well as on other variables) and then executes the program \( \alpha \), which may now use the local variables in assignments. If no values that satisfies \( \varphi \) exists, \( \beta \) is executed. The ifnone can be dropped, and the default behavior then is nontermination (i.e. ifnone abort). choose is more general than or: or can be simulated by choosing a boolean variable \( b \):

\[ \text{choose } b \text{ with } \text{true in if } b \text{ then } \alpha \text{ else } \beta \]

It is strictly more powerful than or, which cannot be used to implement a choice between infinitely many values (e.g. to choose an arbitrary natural number). The relational semantics of or is the union of both semantics

\[ z[\alpha \lor \beta]z' \text{ iff } z[\alpha]z' \text{ or } z[\beta]z' \]

The semantics of choose is somewhat more complex:
CHAPTER 8. CONTRACTS, ABSTRACT DATA TYPES AND REFINEMENT

The relational semantics of nondeterministic programs is sufficient to answer the question: what are the possible final states of the program when started in a state \( z \)? It is no longer sufficient to answer the question: Is the program guaranteed to terminate, when started in a state \( z \)? This can be seen when comparing the semantics of \texttt{skip} with the semantics of \texttt{skip or abort}: both programs have the same relational semantics (since the semantics of \texttt{abort} is empty)! This is different from deterministic programs where possible and guaranteed nontermination are the same.

Therefore in addition to defining a relational semantics, a formal semantics for nondeterministic programs \( \alpha \) must in addition specify a set of states where \( \alpha \) is guaranteed to terminate. We write \( \alpha \downarrow z = \text{tt} \) iff \( \alpha \) is guaranteed to terminate, when started in \( z \). A full formal definition for all cases is left to the reader, some cases of the definition are:

- \( \texttt{abort} \downarrow z = \text{ff} \)
- \( \texttt{skip} \downarrow z = \text{tt} \)
- \( \mathbf{x} := e \downarrow z = \text{tt} \)
- \( \alpha \lor \beta \downarrow z \) iff \( \alpha \downarrow z = \text{tt} \) and \( \beta \downarrow z = \text{tt} \) for all states \( z \) that \( \alpha \) can reach, when started in \( z \). These are the states with \( \mathbb{z}[\alpha]z' \).
- \( \texttt{choose} \ \mathbf{z} \ \mathbf{with} \ \varphi \ \mathbf{in} \ \alpha \ \mathbf{ifnone} \ \beta \downarrow z \) iff either there is an \( a \) with \( A,v_a \models \varphi \), and all such \( a \) satisfy \( \alpha \downarrow v_a' = \text{tt} \) or there is no such \( a \) and \( \beta \downarrow z \) holds.

With this additional semantic definition we can distinguish between \texttt{skip or abort} and \texttt{skip}, since \( \texttt{skip or abort} \downarrow z = \text{ff} \) (this program is not guaranteed to terminate in any state), while \( \texttt{skip} \downarrow z = \text{tt} \). For deterministic programs only we have

\[ \alpha \downarrow z = \text{tt} \] iff there is a state \( z' \) with \( \mathbb{z}[\alpha]z' \)

The diamond operator is no longer sufficient to show guaranteed termination for nondeterministic programs. It shows only, that the program has a terminating run. This is an interesting property, but for total correctness one usually wants that all runs of a program terminate (and satisfy some postcondition \( \varphi \)). Therefore we define a new operator \( \langle |\alpha| \rangle \varphi \), which makes this assertion. Its semantics is

\[ \mathbb{z}[\langle |\alpha| \rangle \varphi]z = \text{tt} \] iff \( \alpha \downarrow z = \text{tt} \) and all \( z' \) with \( \mathbb{z}[\alpha]z' \) satisfy \( \mathbb{z}[\varphi]z' = \text{tt} \).

The “strong diamond” operator \( \langle |\alpha| \rangle \varphi \) of KIV is usually written \( \text{wp}(\alpha, \varphi) \) (from weakest precondition) in the literature. It was first defined in Dijkstra's book [Di j76]. Note that “\( \langle |\alpha| \rangle \varphi \) true” states, that \( \alpha \) is guaranteed to terminate.

Guaranteed termination must be added to the definition of the contract for an implemented procedure given in formula 1 of Section 8.7, when the procedure is nondeterministic. It must change to

\[ \text{COP}(i,cs,cs',o') \leftrightarrow (\langle |\text{COP#(i;cs,o')}| \rangle \text{true}) \land (\text{COP#(i;cs,o')}) \ (cs = cs' \land o = o') \] (3)
The proof obligation also changes slightly. It now uses the strong diamond:

\[ \text{ABS}(as,cs) \land \text{pre}(\text{AOP})(i,as) \rightarrow \langle \text{COP}#(i;cs,o) \rangle \ (\exists as'. \ \text{AOP}(i,as,as',o) \land \text{ABS}(as',cs)) \]  

(4)

This is the proof obligation we will use for the implementations of the procedures in the exercise.

### 8.9 Rules for nondeterministic programs

Since the exercise uses a nondeterministic choose for allocation in heaps (see below), the proof obligations for the exercises below will use the strong diamond. For the deterministic programs we have seen so far, the calculus rules for the strong diamond are exactly the same as the ones for the diamond, so there is nothing new to learn for these. The strong diamond rules for the two nondeterministic program constructs in the succedent are given below. Note that for these the rules are similar to their box rules, since all results have to satisfy the postcondition. In particular, for \( \alpha \) or \( \beta \), the strong diamond requires to prove that both \( \alpha \) and \( \beta \) terminate and \( \psi \) holds at the end.

\begin{align*}
\text{or right} \\
\Gamma \vdash (\alpha) \psi, (\beta) \psi, \Delta & \quad \Gamma \vdash [\alpha] \psi, \Delta & \quad \Gamma \vdash [\beta] \psi, \Delta \\
\Gamma \vdash (\alpha \lor \beta) \psi, \Delta \\
\Gamma \vdash \langle [\alpha \lor \beta] \psi \rangle, \Delta
\end{align*}

\begin{align*}
\text{choose right} \\
\Gamma \vdash \exists y. \varphi^y \land (a^y_x) \psi, (\forall x. \neg \varphi) \land (\beta) \psi, \Delta \\
\Gamma \vdash \langle \text{choose } x \text{ with } \varphi \text{ in } \alpha \text{ ifnone } \beta \rangle \psi, \Delta \\
\varphi^y, \Gamma \vdash [a^y_x] \psi, \Delta & \quad (\forall x. \neg \varphi, \Gamma \vdash [\beta] \psi, \Delta \\
\Gamma \vdash \langle \text{choose } x \text{ with } \varphi \text{ in } \alpha \text{ ifnone } \beta \rangle \psi, \Delta
\end{align*}

The variables \( y \) are new. The rules \text{or left} and \text{choose left} for the antecedent are similar.

### 8.10 Allocation in heaps

Allocation in heaps can be done via a new function

\[ \text{new} : \text{heap} \rightarrow \text{Ref}; \]

that is specified with the axiom

\[ \neg \text{new}(H) \in H \land \text{new}(H) \neq \text{null} \]
It is somewhat more elegant and gives simpler proofs, to use nondeterminism in the programs that have to allocate a new address. It is also slightly more realistic, since choose may choose a different value every time it is invoked, while for a fixed heap $H$, the reference $new(H)$ is always the same ($new$ is a function!). The standard code for allocation is:

\[
\text{choose } r \text{ with } \neg r \in H \land r \neq \text{null in } H[r] := \ldots
\]

Note that choosing the $r$ itself does not allocate the reference. Only the assignment $H[r] := \ldots$ does.
8.11 Exercise 5

Select the project Exercise5. It contains a specification enqueue of queues, based on a specification of lists from the library. This specification is used as the abstract specification of a refinement. The specification cons-heap of heaps specifies the contents of a heap cell as a pair of an elem selected with .val and a reference to a next cell, selected with .nxt. Your task is to do one implementation of queues by sequences of cons-cells.

Exercise 5.1 (implementation of queues by sequences)

In this exercise a queue is represented as the sequence of values reachable from a start pointer r. H[r].val should be the first element of the queue. This element should be dequeued by the deq# operation. Specification queue-refinement already contains a specification of the abstraction function abs. abs(r,H,q) is true if the sequence starting at r indeed represents the queue q. The exercise consists of two parts:

First, provide a proper implementation of the operations emq# enq#, deq#, by modifying the empty implementations in specification queueimpl. Prepare initial versions of these implementations at home. Second, you should prove the (already defined) proof obligations for refinement in specification queueimpl (these are instances of the proof obligation 4; the initialization condition would be abs(null,H,empty); it is trivial).

Hints:
- The only difficult implementation enq#, since it must walk through the list, to attach the element at the end. You can either use recursion or a while loop.
- For recursion: an information you may need is: Given the heaps H and H' before and after enq#, and any allocated reference r (r ∈ H, r ≠ null) with H[r].nxt ≠ null before the operation. Then this reference will not change: r ∈ H' ∧ H[r] = H'[r] holds.
- For a while loop: Invariants may not help. An alternative is to use induction over queues and while right to unwind a loop. execute while has the same purpose for while loops as execute call has for calls.
- A simplifier rule about the case where r = null in abs is a good idea.

Exercise 5.2 (implementation of queues by sequences with tail pointer)

In this exercise, you should implement heap based queues that have an additional pointer r1 to the last cell (tail-pointer), besides a pointer to the first cell, r0. Specification queueimpl contains procedure skeletons for enq#, deq# and emq#. Your task is to provide an implementation for these procedures. You might need special code for the cases r0 = null or r0 = r1.

In specification queue-refinement, complete the definition of the abstraction relation abs(r0, r1, H, q). Then, prove correctness of the refinement.

Hints:
- The recursive definition of abs has three cases. One for the empty queue, one for a one-element queue, and one for a queue with more elements. The axioms should be usable as often as possible as a simplifier rule.
- The implementation is quite simple. It has no recursion and no while loops.
- A useful lemma (and simplifier rule) may start with: abs(null, r1, H, q) ↔ . . .
- You will also need a lemma such as abs(r, r, H, a + q) → q = ]. This lemma is hard to prove, a direct inductive proof will fail since the induction hypothesis is not applicable. The trick is to find and prove a generalization. Intuitively, this lemma says that there are no cycles.
Chapter 8

Verification of Parallel Programs

The goal of this chapter is to learn how to verify temporal properties of simple parallel programs with KIV. Temporal logic is defined and a simple programming language for parallel programs is introduced. Furthermore, the basics of the KIV proof method to verify temporal properties of parallel programs are explained.

8.1 Introduction

Example Consider the following program semaphore which contains two processes running in parallel.

semaphore ≡

while true do
  await \( S > 0; S := S - 1; \)
  (\( L_1: skip; \))
  \( S := S + 1; \)
  (\( L_2: skip; \))

end while

The two processes consist of while loops which never terminate. The processes make use of a so called semaphore \( S \) to synchronise the execution of a critical program fragment. The critical fragment is here abstracted to a no operation statement \( skip \) which is labelled \( L_1 \) for the first process and \( L_2 \) for the second. Before entering the critical section, the processes wait for the semaphore \( S \) to be greater than 0. If this is the case, the semaphore is immediately decremented to prevent the other process from also entering its critical section. After the execution of the critical statements, semaphore \( S \) is restored to its original value 1.

As a requirement, the two processes must never execute their critical program fragments at the same time.

The requirements we consider here are formalised in temporal logic which can be used to restrict the behaviour of a program during execution. In contrast to Hoare Logic, we need to consider not only the initial and final states of program execution, but all of the intermediate states as well.

Parallel processes communicate with each other using shared variables. It is also possible for the processes to communicate with an abstract environment. In this case, the parallel program defines a reactive system which reacts on input from the outside world.

A temporal proof obligation in KIV most often is of the following form

\[
[\ldots \alpha], \Box \psi, \varphi_{PL} \vdash \chi
\]

where \( \alpha \) is the parallel program to examine, \( \psi \) is a so called environment assumption and \( \varphi_{PL} \) describes the precondition. Formula \( \chi \) is the temporal formula to verify.
Example The following temporal proof obligation for program semaphore gives an impression of the task which is to solve within this chapter.

\[
\begin{align*}
&\text{(: parallel program :) } \\
&\text{[: \ldots \text{semaphore}], } \\
&\text{(: environment assumption :) } \\
&\mathcal{D} (S'' = S' \land L''_1 = L'_1 \land L''_2 = L'_2) \\
&\text{(: initial values :) } \\
&S = 1, \neg L_1, \neg L_2 \\
&\vdash \text{(: property to prove :) } \\
&\mathcal{D} \neg (L_1 \land L_2)
\end{align*}
\]

If the parallel program semaphore is executed in an environment that never modifies neither semaphore S nor labels L₁ and L₂, then the property \( \mathcal{D} \neg (L_1 \land L_2) \) (read “always during execution not both labels L₁ and L₂ are satisfied at the same time”) holds. Initially, semaphore S is 1 and both labels are false.

The semantics of temporal formulas and the simple programming language for parallel programs are explained in Section 8.2. An introduction to the KIV proof method is given in Section 8.3. It is your task to verify the property above in Section 8.5.

8.2 Semantics

8.2.1 Traces

In predicate logic, a so called state \( z \) maps variables to values. The value of a variable \( x \) in state \( z \) is \( z(x) \). In temporal logic, (linear) sequences of states \((z_0, z_1, z_2, z_3, \ldots)\) are considered.

A sequence of states represents all of the intermediate variable assignments of a program execution. The sequences can be finite or infinite, in the case of a nonterminating program. In the following, we will also refer to sequences of state as traces. We write down traces as \((z_0, \ldots, z_\pi)\) with variable \( \pi \in \mathbb{N} \cup \{\infty\} \): the length \( \pi \) of the trace is either finite (\( \in \mathbb{N} \)) or infinite (\( = \infty \)).

Example Program

\[
\text{while true do } N := N + 1
\]

never terminates and increments variable N in each step. If the program is executed, different sequences of states result depending on the initial value of \( N \). If \( N \) is initially 5, then the following sequence is generated.

\[
\begin{array}{c|c|c|c|c}
& z_0 & z_1 & z_2 & z_3 \\
N & 5 & 6 & 7 & 8 \\
\end{array}
\]

8.2.2 Static and dynamic variables

In KIV, we distinguish between static and dynamic variables. Static variables can be compared to constants and dynamic variables to program variables of a programming language. Throughout the execution of a program, static variables do not change their initial value, whereas dynamic variables usually have different values at different times of an execution. Dynamic variables are also called flexible. Variables in KIV are static by default. In order to define a dynamic variable, a keyword flexible is used in the variable slot of a specification.

Example Statement

\[
\text{variables } N, N_1, N_2 : \text{nat flexible;}
\]
defines \( N, N_1, \) and \( N_2 \) to be dynamic variables of type \( \text{nat} \).

As a convention, names of dynamic variables always start with an uppercase letter, while static variables are denoted by names starting with a lowercase letter.

### 8.2.3 Primed and double primed variables

A primed variable \( X' \) can be used to refer to the value of a variable \( X \) after a transition. This notation is useful to formalise properties of a program transition. A formula in predicate logic, which refers to unprimed and primed dynamic variables \( X \) and \( X' \) defines a program transition. The unprimed variables \( X \) are interpreted as program input, the primed variables \( X' \) as output.

**Example** Formula \( N' > N \) states that the program transition modifies \( N \) such that the output value \( N' \) is strictly larger than the input value of \( N \).

Reactive systems communicate with an environment (cf., e.g., [dRdBH+01]). In order to model the behaviour of the environment, double primed variables \( X'' \) are used. A formula in predicate logic, which refers to primed and double primed dynamic variables \( X' \) and \( X'' \) defines an environment transition. The primed variables \( X' \) are interpreted as input for the environment, the double primed variables \( X'' \) as output.

**Example** Formula \( N'' = N' \) states that the environment does not change the value of \( N \).

Program and environment transitions alternate. A state transition between states \( z_i \) and \( z_{i+1} \) starts with a program transition which is followed by an environment transition. This can be depicted as follows.

In state \( z_0 \), the program transition takes the initial, unprimed value of \( X \) as input and returns value \( X' \) which is the input to the environment transition. The environment returns the double primed value \( X'' \). **Important:** The double primed value \( X'' \) is equal to the unprimed value \( X \) in the next state \( z_1 \). In other words: the output of the environment \( X'' \) in state \( z_0 \) is the input \( X \) to the program in state \( z_1 \).

### 8.2.4 Semantics of temporal operators

Temporal logic (cf., e.g., [Mos86]) is an extension of standard first order logic. It is based on expressions \( e \) which are variable or function symbols. Interpretation of static variables and unprimed variables is done by evaluating the state function \( z_0 \) of an interval \( I = (z_0, z_1, \ldots) \). Evaluation of primed and double primed variables can be done by evaluating \( z_0' \) and \( z_1 \) respectively. Function symbols \( f \) are recursively interpreted on an Algebra \( A \) as \( f_A((e_1)_I, \ldots, (e_n)_I) \) in the standard way. Similarly a predicate formula \( p \) is interpreted as \( p_A((e_1)_I, \ldots, (e_n)_I) \). All other predicate logic formulas are interpreted as usual.
(z_0, \ldots, z_\pi) \models \Box \varphi \text{ iff } (z_i, \ldots, z_\pi) \models \varphi \text{ for all } 0 \leq i \leq \pi

(z_0, \ldots, z_\pi) \models \Diamond \varphi \text{ iff } \text{ there exists } 0 \leq i \leq \pi \text{ with } (z_i, \ldots, z_\pi) \models \varphi

(z_0, \ldots, z_\pi) \models \varphi \text{ until } \psi \text{ iff } \text{ there exists } 0 \leq i \leq \pi \text{ with } (z_i, \ldots, z_\pi) \models \psi

\quad \text{and } (z_j, \ldots, z_\pi) \models \varphi \text{ for all } 0 \leq j < i

(z_0, \ldots, z_\pi) \models \varphi \text{ unless } \psi \text{ iff } \Box \varphi \text{ or } \varphi \text{ until } \psi

(z_0, \ldots, z_\pi) \models \circ \varphi \text{ iff } \pi \neq 0 \text{ and } (z_1, \ldots, z_\pi) \models \varphi

(z_0, \ldots, z_\pi) \models \bullet \varphi \text{ iff } \pi = 0 \text{ or } (z_1, \ldots, z_\pi) \models \varphi

(z_0, \ldots, z_\pi) \models \text{ last} \text{ iff } \pi = 0

(z_0, \ldots, z_\pi) \models \text{ blocked} \text{ iff } \pi \neq 0 \text{ and } z_0(\text{Blk}) \text{ is true}

Table 8.1: Formal semantics of temporal operators

\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
\(z_0\) & \(z_1\) & \(z_2\) & \(z_\pi\) \\
\hline
\hline
\varphi & \varphi & \varphi & \varphi \\
\hline
\hline
\end{tabular}
\end{center}

Figure 8.1: Illustrations for \(\Box \varphi\) on the left and \(\varphi \text{ until } \psi\) on the right

In order to formalise properties of a sequence of states, a number of different temporal operators is provided. The operators used here, are informally defined as follows.

- \(\Box \varphi\): \(\varphi\) holds now and \textit{always} in the future
- \(\Diamond \varphi\): \(\varphi\) holds now or \textit{eventually} in the future
- \(\varphi \text{ until } \psi\): \(\varphi\) \texttt{eventually holds and } \(\varphi\) holds \textit{until} \(\psi\) holds
- \(\varphi \text{ unless } \psi\): \(\Box \varphi\) or \(\varphi \text{ until } \psi\)
- \(\circ \varphi\): there is a next step which satisfies \(\varphi\) (\textit{strong next})
- \(\bullet \varphi\): if there is a next step, it satisfies \(\varphi\) (\textit{weak next})
- \textit{last}: the current state is the \textit{last}
- \textit{blocked}: execution is \textit{blocked}

While the informal definitions sketch the basic idea of the operators, the formal definitions of Table 8.1 clarify the semantics of the operators in detail. A trace \((z_0, \ldots, z_\pi)\) satisfies \(\Box \varphi\), if and only if for every postfix \((z_i, \ldots, z_\pi)\) formula \(\varphi\) is satisfied. This is depicted on the left of Figure 8.1. The trace satisfies \(\varphi \text{ until } \psi\), if and only if for some \(i\), postfix \((z_i, \ldots, z_\pi)\) satisfies \(\psi\) (in other words: \(\psi \text{ eventually holds} \)), and for all \(j < i\), postfix \((z_j, \ldots, z_\pi)\) satisfies \(\varphi\) (in other words: \(\varphi \text{ always holds before } \psi \text{ holds}\)). This is depicted on the right of Figure 8.1. The notion of blocking is relevant for the interleaving of two programs.

Example Consider the following temporal formulas with their informal interpretations.

1. Formula \(\Diamond N = 3\) states that eventually the value of \(N\) is equal to 3.
2. Formula \(N' = N + 1 \text{ until } N > 10\) states that a program increments \(N\) until the value of \(N\) is greater than 10.
3. Formula $\square N'' = N'$ states that the environment always leaves $N$ unchanged.

### 8.2.5 Semantics of parallel programs

A parallel program can be interpreted as a transition system. In each step $z_i$, the program takes the unprimed values $X$ as input and executes a certain transition to calculate the output values $X'$.

A program only modifies certain variables and leaves other variables unchanged. A variable list $vl$ in front of a program denotes which variables remain unchanged, if they are not assigned by a current transition of the program. We use a special formula $\tau_{vl}$ to express that all variables $X$ from $vl$ remain unchanged by a program.

$$\tau_{vl} \equiv \bigwedge \{ x_1 = x_0 \mid X \in vl \}$$

where $x_0$ denotes the value of $X$ before a program transition and $x_1$ is the value of $X$ after this transition. In order to describe a transition, we need to know,

- which variables are modified and which variables are unchanged,
- whether a transition is blocked, and
- how execution is continued after the first transition.

In Table 8.2, we informally describe the transition system which is defined by the different program statements. Statements $X := t$ and skip define single transitions. Sequential composition, conditionals and while loops, combine transition systems $\alpha$ and $\beta$ to form a larger transition system. An await statement is used to synchronise parallel processes: execution of a process is blocked until a certain condition holds. Finally, the interleaving operator combines two parallel processes. Either a nonblocked transition of process $\alpha$ or of process $\beta$ is executed. If both processes are blocked, then the interleaving is also blocked.

In KIV, programs $\alpha$ are written in square brackets $[V_1,\ldots,V_n \mid \alpha]$ to receive a temporal formula.

**Example** The first example explains the semantics of the variable list in front of a program.

$[\cdot X,Y,Z \mid X := 1]$  

The program sets $X'$ to 1 and leaves variables $Y$ and $Z$ unchanged. As an example for a temporal proof obligation, consider

$[\cdot N \mid \textbf{while true do } N := N + 1 \textbf{, } \square N'' = N', N = 1 \vdash \emptyset N = 3]$  

The program is a nonterminating loop which increments $N$ in every step of the loop body. The program transitions are never blocked. The environment does not modify $N$. Initially, the value of $N$ is 1. Our task is to verify that eventually the value of $N$ is greater than 3.

### 8.3 Calculus

The proof method for the verification of parallel programs in KIV is symbolic execution with induction. Different from verifying sequential programs in Hoare Logic or Dynamic Logic, the temporal property must be considered during execution already. Therefore, program and property are “executed” simultaneously. Executing two interleaved parallel processes gives many different cases, i.e., proof branches: in each step either a transition of the first or the second process is executed. To counter the exponential growth of the proof size, a simple sequencing strategy is exploited in KIV. The three principles
CHAPTER 8. VERIFICATION OF PARALLEL PROGRAMS

\[ vl | X := t \quad \text{an assignment changes } X' \text{ to the value of } t \text{ and leaves all other variables in variable list } vl \text{ unchanged}; \text{ it is never blocked; it terminates after the first transition.} \]

\[ vl | \text{skip} \quad \text{the no operation statement leaves all variables in } vl \text{ unchanged; it is never blocked; it terminates after the first transition.} \]

\[ vl | \alpha; \beta \quad \text{sequential composition is to execute the transitions of program } \alpha \text{ first and then to execute } \beta. \]

\[ vl | \text{if } \epsilon \text{ then } \alpha \text{ else } \beta \quad \text{a conditional requires one step (skip) to evaluate its condition } \epsilon; \text{ it executes the transitions of } \alpha, \text{ if the condition is true, otherwise } \beta \text{ is executed.} \]

\[ vl | \text{while } \epsilon \text{ do } \alpha \quad \text{a while loop requires one step (skip) to evaluate its condition } \epsilon; \text{ it executes } \alpha \text{ as long as } \epsilon \text{ is true, otherwise it terminates.} \]

\[ vl | \text{await } \epsilon \quad \text{an await statement takes no step to evaluate its condition } \epsilon; \text{ it terminates as soon as } \epsilon \text{ is true; until then all variables are unchanged and the program is blocked.} \]

\[ vl | \alpha \parallel \beta \quad \text{interleaving two programs is to either execute the first transition of program } \alpha \text{ or } \beta, \text{ but only if the transition is not blocked and then continuing by interleaving the rest of the programs; the interleaving is blocked, if the first transitions of both programs are blocked.} \]

Table 8.2: Informal semantics of parallel programs

1. symbolic execution (see Sect. 8.3.1),

2. induction (see Sect. 8.3.2), and

3. sequencing (see Sect. 8.3.3)

are explained next.

8.3.1 Symbolic execution

For symbolic execution of simple temporal proof obligations, a single proof rule

\[
\frac{L(\Gamma) \vdash L(\Delta) \quad S(\Gamma) \vdash S(\Delta)}{\Gamma \vdash \Delta \quad \text{step}}
\]

is sufficient. Rule step generates two premises, the first assuming that execution terminates, the second executing all of the possible first steps. The rule considers every parallel program and temporal formula occurring in the sequent simultaneously! Function \(L(\varphi)\) returns a formula describing the conditions under which execution terminates. Function \(S(\varphi)\) calculates a formula which describes the first transition. The two functions are explained next.

Executing PL formulas

The basic idea of executing a PL formula is to substitute the values of the dynamic variables \(X\) in the first state \(z_0\) with fresh static variables \(x_i\). If the first transition is executed, then the values of the unprimed and primed dynamic variables \(X\) and \(X'\) are stored in distinct static variables \(x_0\) and \(x_1\). Moreover, variables \(X''\) are replaced with the corresponding unprimed variables \(X\).

\[
S(\varphi_{PL}) \equiv \varphi_{PL,X,X',X''}^{x_0,x_1,X}
\]
If execution terminates, then all unprimed, primed and double primed dynamic variables $X$, $X'$, and $X''$ are substituted with a single static variable $x_0$ which represents the final value of the dynamic variable $X$.

$$\mathcal{L}(\varphi_{pl}) \equiv \varphi_{pl}^{x_0,x_0,x_0}$$

Static variables in $\varphi_{pl}$ are not affected by functions $S$ and $\mathcal{L}$!

**Example** Executing the first transition of formula $N' = N + 1 \land N'' = N$ leads to

$$S(N' = N + 1 \land N'' = N') \equiv n_1 = n_0 + 1 \land N = n_1 .$$

After the first transition, the new initial value of $N$ is the old value $n_0$ incremented by one. If execution terminates, then we receive the condition

$$\mathcal{L}(N' = N + 1 \land N'' = N') \equiv n_0 = n_0 + 1 \land n_0 = n_0 .$$

(In this case, the condition is contradictory. Execution cannot terminate while variable $N$ is still incremented.)

### Executing TL formulas

The definitions of functions $\mathcal{L}(\varphi)$ and $S(\varphi)$ for a temporal formula $\varphi$ are given in Table 8.3. For operator $\Box \varphi$ if execution terminates, then formula $\mathcal{L}(\varphi)$ must hold in the final state. Otherwise, if execution takes a step, then formula $S(\varphi)$ holds now and after executing the first transition, $\Box \varphi$ again holds. For operator $\varphi$ until $\psi$ if execution terminates, then formula $\mathcal{L}(\psi)$ must hold. If execution takes a step, then either $S(\psi)$ holds in the current state, or formula $S(\varphi)$ holds and after the execution of the first transition, $\varphi$ until $\psi$ again holds.

For combinations of temporal formulas with conjunction $\varphi \land \psi$, disjunction $\varphi \lor \psi$, etc., functions $S$ and $\mathcal{L}$ simply recurse. For example, $S(\varphi \land \psi) \equiv S(\varphi) \land S(\psi)$.

**Example** Executing the first transition for formula $N' = N + 1$ until $N > 10$ leads to

$$S(N' = N + 1 \text{ until } N > 10)$$

$$\equiv S(N' = N + 1) \lor (N' = N + 1 \text{ until } N > 10)$$

$$\equiv n_0 > 10 \lor n_1 = n_0 + 1 \land (N' = N + 1 \text{ until } N > 10) .$$

Either the old value $n_0$ is greater than 10 (in this case, the final condition of operator until is satisfied), or $n_0$ is incremented to receive $n_1$ and the until condition again holds in the next state. If execution terminates, then we receive the condition

$$\mathcal{L}(N' = N + 1 \text{ until } N > 10)$$

$$\equiv \mathcal{L}(N > 10)$$

$$\equiv n_0 > 10 .$$

The final value $n_0$ of variable $N$ must be greater than 10.

<table>
<thead>
<tr>
<th>$\varphi$</th>
<th>$\mathcal{L}(\varphi)$</th>
<th>$S(\varphi)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Box \varphi$</td>
<td>$\mathcal{L}(\varphi)$</td>
<td>$S(\varphi) \land \Box \varphi$</td>
</tr>
<tr>
<td>$\Diamond \varphi$</td>
<td>$\mathcal{L}(\varphi)$</td>
<td>$S(\varphi) \lor \Diamond \varphi$</td>
</tr>
<tr>
<td>$\varphi \text{ until } \psi$</td>
<td>$\mathcal{L}(\psi)$</td>
<td>$S(\psi) \lor S(\varphi) \land \varphi \text{ until } \psi$</td>
</tr>
<tr>
<td>$\varphi \text{ unless } \psi$</td>
<td>$\mathcal{L}(\varphi) \lor \mathcal{L}(\psi)$</td>
<td>$S(\psi) \lor S(\varphi) \land \varphi \text{ unless } \psi$</td>
</tr>
<tr>
<td>$\bullet \varphi$</td>
<td>true</td>
<td>$\varphi$</td>
</tr>
<tr>
<td>$\circ \varphi$</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>last</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>blocked</td>
<td>false</td>
<td>blk0</td>
</tr>
</tbody>
</table>
### Executing parallel programs

For executing parallel programs, we define a special function

\[ \mathcal{T} : \text{VLPROG} \rightarrow \{ (\text{FMA} \times \{\text{F}, \text{T}\} \times (\text{VLPROG} \cup \{\text{E}\})) \} \]

which returns the set of first program transitions. A transition is a triple containing

- **FMA**: the PL formula defining the relation between unprimed and primed variables,
- **\{F, T\]**: a flag which is T, if the transition is blocked, and
- **(VLPROG \cup \{E\})**: the remaining program to execute in the next step. If no program remains, E is returned.

Table 8.4 contains the definition of function \( \mathcal{T}(\alpha) \) for all program statements in addition to the definition of function \( \mathcal{L}(\alpha) \) returning a formula which is the condition under which the program terminates. For assignments, a set containing a single transition is returned, which defines variable \( x_1 \) to be equal to \( S(t) \) and leaves all other variables from the variable list \( \text{varlist} \) unchanged (\( \tau_{\text{var}} \setminus x \)). The transition is not blocked (flag F) and the remaining program for the next state is empty (program E). A while loop always takes a step to evaluate its condition \( \varepsilon \). It is consequently never blocked. If \( \varepsilon \) is satisfied, execution continues with the loop body \( \alpha \) together with the repeated while loop. Otherwise, the loop terminates after one step. Evaluating the condition of an if-construct also consumes one step. The transition is not blocked. If \( \varepsilon \) holds in the current state, the remaining program is \( \alpha \), otherwise \( \beta \). The first transitions of an interleaving \( \alpha \parallel \beta \) of two processes \( \alpha \) and \( \beta \) are given by the union of all unblocked first transitions of \( \alpha \) and \( \beta \) first transitions of \( \beta \). Furthermore, if for both processes there are transitions which are blocked, then there is also the possibility for the interleaving to be blocked.

#### Example

The transitions of program \( N | \text{while true do } N := N + 1 \) are calculated as follows. First, the condition of the while loop is evaluated.

\[
\mathcal{T}(N | \text{while true do } N := N + 1) \equiv \{(n_1 = n_0, F, N \mid N := N + 1; \text{while true do } N := N + 1), (\text{false, F, E})\}
\]

The first transition of the loop evaluates the loop condition, which in this case is trivially true. This transition does not change \( N \) and is not blocked (F). In the first tuple, execution continues with the loop body \( N := N + 1 \), followed by the entire loop. The second tuple describes the case in which the while loop is exited (false) and no program is left (E).
Calculating $T$ for the remaining program yields the following set of transitions.

$$
T(N \mid N := N + 1; \textbf{while} \ true \textbf{ do} N := N + 1) \\
≡ \ (\{\text{false} \land \varphi, b, N \mid \beta'\} \mid (\varphi, b, N \mid \beta') \in T(N \mid \textbf{while} \ true \textbf{ do} N := N + 1)) \\
\cup \ (\{\varphi, b, N \mid \alpha'; \textbf{while} \ true \textbf{ do} N := N + 1 \mid (\varphi, b, N \mid \alpha') \in T(N \mid N := N + 1))
$$

where

$$
T(N \mid N := N + 1) \\
≡ \ \{(n_1 = n_0 + 1, F, E)\}
$$

$$
≡ \ \{(n_1 = n_0 + 1, F, N \mid \textbf{while} \ true \textbf{ do} N := N + 1)\}
$$

The result is a single transition where $N$ is incremented by one. The transition is not blocked and the program continues with again executing the while loop.

The transitions of a temporal formula $[\cdot \ v_l \mid \alpha]$ containing a parallel program are executed by calculating the set of transitions of $v_l \mid \alpha$ with function $T(.)$ and by transforming the resulting program transitions into a temporal formula:

$$
S([\cdot \ v_l \mid \alpha]) \equiv \bigvee \{\varphi \land \neg \text{blk}_0 \land [\cdot \ v_l \mid \alpha'] \mid (\varphi, \text{F}, v_l \mid \alpha') \in T(v_l \mid \alpha)\} \\
\lor \bigvee \{\varphi \land \text{blk}_0 \land [\cdot \ v_l \mid \alpha'] \mid (\varphi, \text{T}, v_l \mid \alpha') \in T(v_l \mid \alpha)\}
$$

For the unblocked transitions the boolean variable blk$_0$ is false, for the blocked transitions it is true.

**Executing temporal proof obligations**

**Example** Consider again the following proof obligation

$$
[\cdot \ N \mid \textbf{while} \ true \textbf{ do} N := N + 1], \Box N'' = N', N = 1 \vdash \Diamond N = 3
$$

Applying rule step leads to

$$
L([\cdot \ N \mid \textbf{while} \ true \textbf{ do} N := N + 1]), L(\Box N'' = N'), L(N = 1) \vdash L(\Diamond N = 3) \\
S([\cdot \ N \mid \textbf{while} \ true \textbf{ do} N := N + 1]), S(\Box N'' = N'), S(N = 1) \vdash S(\Diamond N = 3)
$$

After evaluating functions $L(.)$, the first premise finally reads:

$$
\neg \text{true}, n_0 = n_0, n_0 = 1 \vdash n_0 = 3 .
$$

Rule simplifier uses the contradiction $\neg \text{true}$ in the antecedent to close the goal. After evaluating functions $S(.)$ in the second premise, we receive

$$
n_1 = n_0 \land \neg \text{blk}_0 \land [\cdot \ N \mid N := N + 1; \textbf{while} \ true \textbf{ do} N := N + 1],
$$

$$
N = n_1 \land \Box N'' = N',
$$

$$
n_0 = 1
$$

$$
\vdash n_0 = 3 \lor \Diamond N = 3 .
$$

Simplifying the goal leads to

$$
[\cdot \ N \mid N := N + 1; \textbf{while} \ true \textbf{ do} N := N + 1], \Box N'' = N', N = 1 \vdash \Diamond N = 3 .
$$

Thus, symbolic execution of the proof obligation gives rise to a single next premise, where the new initial value of $N$ is still 1. The target property $N = 3$ has not yet been satisfied and we need to continue symbolically executing the premise by again applying rule step.
8.3.2 Induction

For proving properties of programs with loops, we basically do well-founded induction over the value of an arbitrary induction term $t$ of type nat with rule

$$
\Gamma, n = t, \text{IndHyp} \vdash \Delta \quad \text{where IndHyp} : \equiv \Box \ t < n \rightarrow (\forall \ \Gamma \rightarrow \bigvee \ \Delta)
$$

The initial value of the induction term $t$ is stored in a fresh static variable $n$. If – after the execution of a number of steps – the value of the term has decreased ($t < n$), the induction hypothesis can be applied.

Normally, induction requires an invariant which again holds after the loop has been executed. The generalization can be obtained using the rules “cut formula” and “weakening”. (Sometimes it is necessary to shut off the heuristics to avoid unintended simplification.)

Example For the proof obligation

$$
:\ N \mid \text{while true do } N := N + 1, \Box N'' = N', N = 1 \vdash \Box N \geq 1
$$

we generalise $N = 1$ with $N \geq 1$ to receive

$$
:\ N \mid \text{while true do } N := N + 1, \Box N'' = N', N \geq 1 \vdash \Box N \geq 1
$$

If an always formula $\Box \ \varphi$ occurs in the succedent, formula $\Box \ \varphi$ can be used to get an induction term. The formula is converted into an equivalent formula

$$
\exists N_0. \ N_0 = N''_0 + 1 \text{ until } \neg \varphi
$$

in the antecedent and variable $N_0$ is used as an induction term.

$$
:\ N \mid \text{while true do } N := N+1], \Box N'' = N', N \geq 1, N_0 = N''_0 + 1 \text{ until } (\neg 1 \leq N), \text{IndHyp} \vdash \Box N \geq 1
$$

After executing the first two transitions with rule step, we receive a single premise

$$
:\ N \mid \text{while true do } N := N + 1, \Box N'' = N', N = n + 1, n \geq 1, \text{IndHyp} \vdash \Box N \geq 1.
$$

In this premise, the body of the while loop has been completed, and the remaining program is identical to the program of the penultimate premise. Furthermore, the constraint $N \geq 1$ is fullfilled. Induction can be applied to close the goal.

Induction can only be applied, if the remaining program is identical to the program in the induction hypothesis. For parallel programs it is not sufficient to use induction once. Instead, induction must be used every time one of the parallel processes is in front of a while loop. In order to avoid the necessity of an induction rule every time a process loops, KIV implements a special induction strategy called VD induction which is initiated only once. Afterwards, if a premise contains a program which already occurred earlier in the proof, induction can be applied using apply VD induction. The rule requires the selection of a node in the proof tree which contains the identical parallel program. We apply the rule by highlighting the desired node in the proof tree and by selecting apply VD induction from the proof menu (see Figure 8.2). Important: It is even possible to refer to proof nodes which are on a different proof branch as long as the condition that the induction term has decreased can be established!

8.3.3 Sequencing

While executing two interleaved processes many different cases must be considered. In each step, either a transition of the first or the second process is executed. After executing several steps, in
some cases the same transitions have been executed, but in a different order. These cases can be contracted with rule *insert proof lemma*

\[
\frac{\Gamma_0 \vdash \Delta_0 \quad \Gamma_0 \land \Gamma_0 \rightarrow \bigvee \Delta_0 \vdash \Delta}{\Gamma \vdash \Delta}
\text{insert proof lemma}
\]

If you arrive with a premise, where the parallel program is the same as in a different premise, highlight the premise and chose *insert proof lemma* from the context menu (see Fig. 8.3).

**Example** Consider the temporal proof obligation

\[
[; \begin{array}{l} vl \mid \{ N := 1; \ldots \} \\ \{ M := 2; \ldots \} \end{array} ] || \begin{array}{l} N'' = N' \land M'' = M' \end{array} \vdash \ldots .
\]

After executing the first step, the following premises result

\[
N = 1, [; \begin{array}{l} vl \mid \{ \ldots \} \\ \{ M := 2; \ldots \} \end{array} ] || \begin{array}{l} N'' = N' \land M'' = M' \end{array} \vdash \ldots
\]

and

\[
M = 2, [; \begin{array}{l} vl \mid \{ N := 1; \ldots \} \\ \ldots \end{array} ] || \begin{array}{l} N'' = N' \land M'' = M' \end{array} \vdash \ldots .
\]
In the first premise, assignment \( N := 1 \) of the first process, in the second premise, assignment \( M := 2 \) of the second process have been executed. Executing a step for both premises gives four premises, and two of the four premises read

\[
N = 1, M = 2, [; vl | \{ \ldots \} \parallel \{ \ldots \}], \Box (N'' = N' \land M'' = M') \vdash \ldots .
\]

In one premise, a transition of the first process was followed by a transition of the second and vice versa. These two premises can be contracted with rule insert proof lemma.

### 8.3.4 Summary of proof method

If you want to execute a concurrent program manually, it is best to do the steps breadth first, i.e., execute a step for every open premise of the proof tree, before you continue with symbolic execution. This corresponds to the following recipee.

1. Use heuristics **TL Heuristics**.

2. If necessary, use rule **generalise** to introduce an invariant.

3. If required, define an induction hypothesis using rule **VD induction**.

4. For every open premise do:
   
   (a) If the remaining program is identical to a program occurring earlier in the proof, try to close the premise with rule **apply VD induction**.

   (b) Otherwise, execute a step with rule **step**.

   (c) Simplify the resulting premises.

5. Try to contract premises with rule **insert proof lemma**.


Instead of using rule **step**, you can also use the rules for individual program statements. The set of KIV heuristics **TL Heuristics + Breadth First Exec** automates steps 1 - 6 above, which are typically rather hard to perform manually.
8.4 Exercise 6, Part 1 (TL Formulas)

Exercise 6.1 This exercise helps you to become familiar with central temporal operators and the calculus (the step rule and VD induction). Formalize and prove the following tautologies in KIV specification “TL-Formulas”, using adequate temporal operators on flexible variables $\varphi_i : \text{bool}$.

1. "What will always be, will be"
2. "If $\varphi_1$ will always imply $\varphi_2$, then if $\varphi_1$ will always be the case, so will $\varphi_2$"
3. "If it will be the case that $\varphi$, it will be, that it will be that $\varphi$"
4. Only use the eventually operator $\Diamond$ to formalize this statement:
   "If it will never be that $\varphi$, then it will be that it will never be that $\varphi$"

Exercise 6.2 Prove the remaining lemmas from specification “TL-Formulas” in any order. Additional remarks:

- The “unless” operator is also known as the “weak until” operator of temporal logic.
- The proof of Lemma fin-progress is optional: the lemma states that in a finite concurrent system of up to "n" processes, infinite progress of some process $m \leq n$ is equivalent to the infinite progress of one individual process $m_0 \leq n$. The proof of the sufficient direction requires structural induction over "n" first. To apply the induction hypothesis ($A \ldots$), simply use rule “all left”.

8.5 Exercise 6, Part 2 (TL Programs)

Exercise 6.3 Prove lemma inc-3 from KIV specification “TL-Progs”

\[ [: N \mid \text{while true do } N := N + 1], \Box N'' = N', N = 1 \vdash \Diamond N = 3 \]

Either use the step rule or rules for the individual program statements. As the standard heuristics set choose TL Heuristics.

Exercise 6.4 Prove lemma while-ilv

\[ [: N \mid \text{while true do } N := N + 1 \parallel N := 1], \Box N'' = N', N = 1 \vdash \Box N \geq 1 \]

from specification TL-Progs in KIV. Generalise the initial condition $N = 1$ appropriately. Use rules VD induction, step and insert proof lemma. Either use the step rule or rules for the individual program statements. As the standard heuristics set choose TL Heuristics.

Exercise 6.5 Prove lemma counter-seq from specification TL-Progs, using heuristics TL Heuristics:

\[ [: X, N \mid \text{while true do} \{ \text{\scriptsize } N := X; \text{\scriptsize } N := N + 1; \text{\scriptsize } X := N; \text{\scriptsize } \}], \Box (X'' = X' \land N'' = N'), N = 0 \vdash \Box \Diamond X' > X \]

Variable $N$ can be seen as a local variable. The sequential counter program repeatedly takes a copy $N$ of variable $X$. After locally incrementing $N$, it updates $X$ to the new value of $N$. It is your task to prove that $X$ will always be eventually incremented.
Exercise 6.6 Lemma counter-ilv from specification TL-Progs regards two simple counter programs which are executed concurrently.

```
[ : X, N_1, N_2 | 
  N_1 := X;
  N_1 := N_1 + 1;
  X := N_1;
]
□ (N'_1 = N_1 ∧ N'_2 = N_2 ∧ X' = X'), X = 0
\vdash \Box (\text{last} ∧ X = 2)
```

Each program increments the value of the shared counter variable X by one. Starting with X = 0, execution of the interleaved program terminates and the value of X is expected to be equal two. Can you prove this property for the interleaved system? If yes, give a proof in KIV using heuristics TL Heuristics + Exec. Otherwise, use the await-construct and a semaphore S (similar to program semaphore above) to construct a symmetric concurrent system, which terminates with postcondition X = 2. Use heuristics TL Heuristics + Breadth First Exec to prove termination and correctness of your extended system.

Exercise 6.7 Prove lemma mutex

```
[ : L_1, L_2, S | semaphore ],
□ (S'' = S' ∧ L''_1 = L'_1 ∧ L''_2 = L'_2),
\neg L_1, \neg L_2, S = 1
\vdash \Box \neg (L_1 ∧ L_2)
```

from specification TL-Progs in KIV. Either manually apply the recipe from 8.3.4 or use the set of heuristics TL Heuristics + Breadth First Exec to automate the proof. It is recommended to try a manual proof first.

Exercise 6.8 (Peterson2) Prove lemma “peterson2-safe” from specification TL-Progs. It ensures correctness of a simple version of Peterson’s [Pet81] mutual exclusion lock for two processes. Again, instead of an interactive proof, you may use the set of heuristics TL Heuristics + Breadth First Exec to automate the proof. You must first manually apply VD Induction though. Similarly, prove lemma “peterson2-live-await”.

Remark: The automatic verification of an algorithm gives little intuition why a property holds. The interactive verification using symbolic execution is similar to the stepwise debugging of the interleaved program and thus, it gives better insight on how the algorithm works and why it is correct. Of course, the latter method is much more time consuming.

8.6 Exercise 6, Part 3 (Rely-Guarantee)

Direct symbolic execution of interleaved programs typically fails even for small programs. Rely-Guarantee (RG) reasoning is a wide-spread decomposition technique that avoids verification of parallel programs by directly computing their interleaved executions. Instead, proof obligations for individual components of an interleaved system are defined that can be composed to yield an overall property. This exercise introduces the basic ideas of rely-guarantee reasoning, by verifying the correctness of a simple concurrent program that calculates the greatest common divisor in parallel. The parallelization in this example does not really improve performance of the computation, but serves for illustration purposes only.

The algorithm concurrent-gcd computes the greatest common divisor \(gcd(m, n)\) of two arbitrary input naturals \(m\) and \(n\), using two processes. The shared memory location \(NP\) holds the input values initially, i.e., \(NP = m \times n\). Process \(P_1\) repeatedly checks whether the first slot \(NP.1\) stores a number that is greater than \(NP.2\). In this case, it sets \(NP.1\) to \((NP.1 - NP.2)\). However, if both slots \(NP.1\) and \(NP.2\) contain the same value, it terminates. Process \(P_2\) is symmetric.
chapter 8. Verification of Parallel Programs

Let $P_1 : \text{let } Done = \text{false, } X = 0, Y = 0 \text{ in } \{\text{while } \neg Done \text{ do } \{X := \text{NP.1}; \text{if } X > Y \text{ then } \text{NP.1} := X - Y; \textrm{if } X = Y \text{ then } Done := \text{true}\}\}\}$ and $P_2 : \text{let } Done = \text{false, } X = 0, Y = 0 \text{ in } \{\text{while } \neg Done \text{ do } \{X := \text{NP.2}; \text{if } X > Y \text{ then } \text{NP.2} := X - Y; \text{if } X = Y \text{ then } Done := \text{true}\}\}$.

Rely-guarantee (RG) decomposition theorems decompose global properties of a concurrent system to local proof obligations for the system’s components. This section introduces a simple rely-guarantee decomposition technique for two interleaved processes. It will be your task to prove the resulting RG assertions for the individual processes $P_1$ and $P_2$ of the concurrent $gcd$ algorithm.

The central task of RG proofs is to verify an RG assertion for each individual process. RG assertions in KIV have the following syntax

$$Pre_i(S) \vdash [R_i(S', S''), G_i(S, S'), Inv(S), P_i(S)] Post_i(S)$$

(1)

where variable $S$: state represents the overall state of the concurrent system. Semantically, the formula says that when running procedure $P_i$ from an initial state that satisfies precondition $Pre_i$, it maintains the guarantee $G_i$ and the invariant $Inv$ in its own steps, as long as previous transitions of its environment satisfy $R_i$ and maintain $Inv$ respectively; upon termination, the program establishes the postcondition $Post_i$.

The invariant predicate is irrelevant in our example. More important is the fact that RG-assertions are safety formulas, i.e., they can be verified by well-founded induction over the length of an arbitrary interval prefix: Use rules extract liveness and VD Induction on the resulting boolean variable Boolean that characterizes the length of the prefix.

Having verified an RG assertion (1) for each individual process $i = 1, 2$, we can derive the following RG assertion for their interleaving.

$$Pre_1 \land Pre_2 \vdash [R_1 \land R_2, G_1 \lor G_2, Inv(S), P_1(S)] \parallel P_2(S) \parallel Post_1 \land Post_2$$

(2)

Some additional predicate logic restrictions $RGrestr$ on the used predicates must also be satisfied to prove (2).

$$RGrestr(R_i, G_i, Pre_i, Post_i) \equiv (G_i(S, S') \rightarrow R_{1-i}(S, S')) \land (Pre_i(S') \land R_i(S', S'') \rightarrow Pre_i(S'')) \land (Post_i(S') \land R_i(S', S'') \rightarrow Post_i(S'')) \land \text{transitive}(R_i)$$

They ensure that (1) the guarantee of a process implies the rely of the other process (2) both preconditions and postconditions must be stable over rely conditions, i.e., they must be maintained by rely transitions and (3) rely predicates must be transitive. The stability of pre- and postconditions over rely steps is required, since before a process starts its execution, respectively after termination of a process, arbitrary many steps of the other process are possible. (Steps of the overall environment are ignored here.)

**Exercise 6.9 (RG Proof of concurrent-gcd)** Verify the correctness of the concurrent $gcd$ program using RG reasoning. The main task is to find suitable rely properties for each individual process such that you can prove the corresponding RG assertion. (The rely predicates of both processes will be of course symmetric.) The right instances for the guarantee, as well as the preconditions and the postconditions are already defined in specification “Concurrent-GCD-RG”. First of all, you should have a close look at these definitions and try to understand their meaning. Then proceed as follows.
Start by trying to prove all predicate logic lemmas of the specification “Concurrent-GCD-RG” first. These correspond to the predicate logic restrictions $RGr estr$ from above. The four lemmas about stability (“post-stable-r1” etc.) are not provable right away, since the initial rely conditions “true” are too weak. Think of reasonable environment restrictions and encode them into the corresponding rely predicates such that all predicate logic lemmas can be proved. (Do not forget to reload the specification.)

Once you have found rely properties such that all predicate logic lemmas can be proved, try to prove the RG assertions “Proc1-sat-rg” and “Proc2-sat-rg”, respectively. (This shows that your assumptions are met by the algorithms indeed.) The proofs of these two lemmas are symmetric and slightly more challenging. When you are at the beginning of the while loop, you need to generalize the current state (use rule “generalise”) and to apply “extract-liveness” and “VD induction” on the RG assertion to enable an inductive proof. Then the proof consists of mainly stepping through the code and applying induction if the program loop is reiterated. For the instruction that modifies the shared variable $NP$, you will have to prove the guarantee and in the last state of the program, the post condition must be shown. Use heuristics TL Heuristics + Exec if you like to have most rules of symbolic execution applied automatically.

**Exercise 6.10 (Peterson2 Revisited)** This exercise is optional:
Give a RG proof of mutual exclusion of Peterson’s lock implementation for two processes (similar to concurrent-gcd).
Appendix A

Simplifier Rules

A.1 Introduction

While verifying formulas of dynamic logic, the cosi strategy must often deal with predicate logic formulas. There are four main reasons, why simplification of such formulas is important:

1. Simplification leads to better readability.
2. Simplification of predicate logic formulas is important to eliminate superfluous variables in program formulas.
3. Goals which are true regardless of all program formulas (including those where all program formulas have been eliminated by the cosi strategy) can be closed.
4. The tests of conditionals and while loops can be decided, which means a reduction of search space.

Simplification is done by rules of the sequent calculus. The rules are (as usual in a sequent calculus) applied backwards, until no one is applicable. Although, the KIV-System stores the antecedent and succedent of a sequent as lists of formulas, the rules, described in the following sections match without considering the order of the formulas, so you can view a sequent as consisting of two sets of formulas. Simplification is done automatically when trying to decide conditional tests, it can be called manually by clicking on “simplifier” and there is a heuristic, also called “simplifier”, to do it.

Since simplification is dependent on the datastructure used in sequents to prove, the information, how to simplify, is kept with the specifications of datastructures.

Simplifier rules are split into four types: First, there are built-in rules, described in section A.2 that are used independent of the datastructure (concerned mostly with propositional logic).

The next class of simplifier rules is datastructure dependent, and given explicitly by the user, while working on a specification. The first class is given by marking some theorems of the specification with the command “Simplifier – Add Simplifier Rules” or “Simplifier – Add Local Simplifier Rules”.

Theorems which are local simplifier rules are used when proving (other) theorems over the same specification, in which the theorem is defined. Theorems which are declared as simplifier rules are used when proving in superspecifications of the specification, in which they are defined. This distinction is useful to use axioms like extensionality only locally in proofs of other theorems, which are then used (globally) as simplifier rules.

Theorems which can be declared to be simplifier rules must have one of several special forms, so that the system is able to generate rules of the sequent calculus from them. The generation of rules is done uniformly, so if you have verified the theorem, there is a uniform validation for the generated rule (which again uses the theorem). The admissible forms for the theorems, and which rules are actually generated is described in section A.3.
The third class of simplifier rules deal are associativity and commutativity axioms (AC). They are not used directly, but allow all other simplifier rules to be used modulo AC. AC rules are also used by several heuristics and tactics to do matching modulo AC (e.g. when computing proposals for instances of quantifiers, when using cut rules (see below), when replacing left hand sides of equations by right hand sides etc.).

The last class of simplifier rules is called “forward rules”. It is used to add information to a sequent, so it is datastructure dependent too. The rules are described in section A.4.

In addition to the simplifier there is an elimination heuristic, which tries to eliminate “bad” function symbols, by trading them for “good” ones. The heuristic and also the “insert elim lemma” is based on “elimination rules”. These are described in the Sect. A.5. Elimination rules are used locally and globally (so there are no special local elimination rules).

Finally there are the two heuristics weak cut and cut, which apply the cut rule on goals. These are based on cut rules, which are explained in the last section. Cut rules may be local or global.

The following conventions apply to the sections below:

- $\Gamma, \Delta, \Gamma_1, \ldots$ are (possibly empty) sets of formulas,
- $\varphi, \psi, \chi$ are formulas
- $\sigma, \tau, \rho$ are terms

A.2 datastructure independent simplification rules

The following datastructure independent simplifier rules are used:

- the usual axiom of the sequent calculus
- all basic rules of the sequent calculus, that eliminate junctors
- rules that eliminate the constants ‘true’ and ‘false’.
- rules, that eliminate reflexive equations.
- two rules for eliminating all-quantifiers in the sucedent and existential quantifiers in the antecedent by inserting new variables
- rules, that try to eliminate equations $x = \tau$ (or $\tau = x$) in the antecedent. For every such equation, where $x$ does not occur in the variables of $\tau$, the rule tries to eliminate the equation by substituting $\tau$ for $x$ in the rest of the sequent. Since substitution is not possible on all program formulas, the rule is not always applicable.
- rules, that minimize the scope of quantifiers, such as ($x \not\in \text{Free}(\psi)$)

$$\forall x.(\varphi(x) \rightarrow \psi) \vdash (\exists x.\varphi(x)) \rightarrow \psi$$

- rules, that eliminate redundant quantifiers, such as ($x \not\in \text{Vars}(\tau)$)

$$\exists x.(x = \tau \land \varphi) \vdash \varphi[x \leftarrow \tau]$$

A.3 Rules generated from theorems

Theorems which may be used to generate simplifier rules have the following general form

$$\Gamma \vdash \psi_1 \land \ldots \land \psi_n \rightarrow \varphi$$

$n$ must be $\geq 0$, for $n = 0$ the implication is omitted. Depending on the form of $\varphi$, the rule is in one of four classes:
Appendix A. Simpifier Rules

- If \( \varphi \) is an equation, then the rule is a rewrite rule.
- If \( \varphi \) is an atomic formula, but no equation, then the rule is an axiom rule.
- If \( \varphi \) is an equivalence then the rule is an equivalence rule.
- If \( \varphi \) is an implication, then the rule is an implication rule.

To avoid conflicts in the last case, \( n \) is not allowed to be 0 there. Instead, we allow in the case \( n = 1 \) \( \psi_1 \) to be true. All rules come with applicability conditions, which are the formulas in \( \Gamma \) and \( \psi_1, \ldots, \psi_n \). Those in \( \Gamma \) must be proved, the others must be present in the sequent. Applicability conditions should be put in the antecedent, if goals often contain stronger conditions which imply the condition (e.g. instead of a condition \( x \neq y \) we often have the stronger conditions \( x < y \) or \( y < x \) in the sequent). Otherwise they should only be put in the preconditions (this is likely to be the case for e.g. \( li \neq nil \)).

The four classes of rules are described in the following subsections. We will use the following notations:

- an atomic formula is either a predicate or an equation
- a literal is either \( \varphi \) or \( \neg \varphi \) for a atomic formula \( \varphi \)
- An atomic formula is in a sequent, if it is in the succedent of the sequent
- A negated atomic formula \( \neg \varphi \) is in a sequent, if \( \varphi \) is in the antedent of the sequent
- For an atomic formula \( \sim \varphi \) is \( \neg \varphi \), for a literal \( \sim \varphi \), \( \sim \varphi \) is \( \varphi \)

Rewrite Rules

Rewrite rules are of the form:

\[
\Gamma \vdash \psi_1 \land \ldots \land \psi_n \rightarrow \sigma = \tau
\]

where \( \sigma \) must not be a variable or a constant. \( \psi_1, \ldots, \psi_n \) must be literals. The variables of the theorem must be a subset of \( \mathbf{x} = \text{Vars}(\psi_1, \ldots, \psi_n, \sigma) \).

The variables of \( \tau \) must be a subset of those in \( \sigma \) (remark: this restriction may be dropped in the future). The rule rewrites instances of the term \( \sigma \) by instances of the term \( \tau \). Formally, the rule is applicable to a sequent \( \Gamma' \vdash \Delta' \) if there is a substitution \( [\mathbf{x} \leftarrow \rho] \) with domain \( \mathbf{x} \) as above such that

- \( \sigma[\mathbf{x} \leftarrow \rho] \) is a term of the sequent contained in some formula \( \varphi \)
- \( \psi_1[\mathbf{x} \leftarrow \rho], \ldots, \psi_n[\mathbf{x} \leftarrow \rho] \) are contained in the sequent
- \( \Gamma' \vdash \Delta, \quad \Gamma[\mathbf{x} \leftarrow \rho] \) is provable by the simplifier, where \( \Gamma'' \vdash \Delta'' \) is \( \Gamma' \vdash \Delta' \) without \( \varphi \).

If the rule is applicable, it replaces all occurrences of \( \sigma[\mathbf{x} \leftarrow \rho] \) in \( \varphi \) by \( \tau[\mathbf{x} \leftarrow \rho] \)

Example: \( \vdash \text{pop(push(a,x)))} = x \)
Example: \( \neg x = 0 \vdash x - 1 + 1 = x \)
Example: \( \vdash \neg li = \text{nil} \rightarrow \text{length(cdr(li))} = \text{pred(length(li))} \)

Pragmatics: These rules are mainly used to do term rewriting. \( \tau \) should be ‘simpler’ than \( \sigma \). Most rewrite rules are either used to eliminate applications of selectors and defined functions to constructor terms (first two examples), or to move such selectors and defined functions “inwards” (last example).
Axiom Rules

\[ \Gamma \vdash \psi_1 \land \ldots \land \psi_n \rightarrow \varphi \]

where \( \varphi \) is a literal, but not an equation. \( \psi_1, \ldots, \psi_n \) must be literals. The variables of the theorem must be a subset of \( \overline{x} = \text{Vars}(\psi_1, \ldots, \psi_n, \varphi) \). The rule closes sequents containing \( \varphi \) and eliminates \( \neg \varphi \) from sequents containing it.

Formally the rule is applicable to a sequent \( \Gamma' \vdash \Delta' \) if there is a substitution \( [\overline{x} \leftarrow \rho] \) with domain \( \overline{x} \) as above such that

- \( \varphi[\overline{x} \leftarrow \rho] \) or \( \neg \varphi[\overline{x} \leftarrow \rho] \) is contained in the sequent
- \( \psi_1[\overline{x} \leftarrow \rho], \ldots, \psi_n[\overline{x} \leftarrow \rho] \) are contained in the sequent
- \( \Gamma'' \vdash \Delta'' \wedge \Gamma[\overline{x} \leftarrow \rho] \) is provable by the simplifier, where \( \Gamma'' \vdash \Delta'' \) is \( \Gamma' \vdash \Delta' \) without \( \varphi[\overline{x} \leftarrow \rho] \)

If the rule is applicable, it removes all occurrences of \( \neg \varphi[\overline{x} \leftarrow \rho] \) from the sequent, resp. closes the goal, if \( \varphi[\overline{x} \leftarrow \rho] \) is contained in it.

Example: \( \neg \ 0 = x + 1 \)
Example: \( \neg \ \text{member}(x, \text{delete}(x, \text{set})) \)
Example: \( \neg \ 0 \leq x \)
Example: \( \neg \ x \leq x \)
Example: \( \neg \ x + 1 \leq x \)

Pragmatics: These rules are mainly used to separate different constructor terms (first two examples) or two specify when a predicate is true or false.

Equivalence Rules

Equivalence rules have one of the four forms, which will be described in the following subsections

**Succedent Equivalence Rules**

A succedent equivalence rule has the form

\[ \Gamma \vdash \psi_1 \land \ldots \land \psi_n \rightarrow (\neg \varphi \leftrightarrow \chi) \]

where \( \varphi \) is an atomic formula. \( \psi_1, \ldots, \psi_n \) must be literals. The variables of the theorem must be a subset of \( \overline{x} = \text{Vars}(\psi_1, \ldots, \psi_n, \varphi) \). The rule replaces instances of \( \varphi \) in the succedent by \( \chi \).

Example: \( \neg \ x \neq y \rightarrow \neg \ y < x \leftrightarrow x < y \)
Example: \( \neg \ x = \text{true} \leftrightarrow x = \text{false} \)
Example: \( \neg \ x = \text{false} \leftrightarrow x = \text{true} \)

Pragmatics: Used in cases, where we do not want the full equivalence rule (with \( \varphi \leftrightarrow \sim \chi \) instead of \( \neg \varphi \leftrightarrow \chi \)) Such cases occur, if the idea of a rule is to prefer some non-negated literal to an (under the sideconditions) equivalent negated literal. An obvious example is the first, where the nonnegated form allows to remove the sidecondition \( x \neq y \) and the full equivalence rule would lead to an infinite loop. The second and third example produces only positive boolean equations in the antecedent. If we would use the equivalence rule \( x = \text{false} \leftrightarrow \neg \ x = \text{true} \) instead, we would get \( \neg \ x = \text{true} \) in the antecedent instead of \( x = \text{false} \).

**Antecedent Equivalence Rules**

An antecedent equivalence rule has the form

\[ \Gamma \vdash \psi_1 \land \ldots \land \psi_n \rightarrow (\neg \neg \varphi \leftrightarrow \chi) \]
where $\varphi$ is an atomic formula. $\psi_1, \ldots, \psi_n$ must be literals. The variables of the theorem must be a subset of $x = \text{Vars}(\psi_1, \ldots, \psi_n, \varphi)$. The rule replaces instances of $\varphi$ in the antecedent by $\chi$.

Pragmatics: Same as in the previous case, but we prefer the negated literal to the positive (rarely used).

### Conjunctive Equivalence Rules

A conjunctive equivalence rule has the form

$$\Gamma \vdash \psi_1 \land \ldots \land \psi_n \rightarrow (\varphi \leftrightarrow \chi_1 \land \ldots \land \chi_m)$$

where $\varphi$ is an atomic formula. $\psi_1, \ldots, \psi_n$ must be literals. The variables of the theorem must be a subset of $x = \text{Vars}(\psi_1, \ldots, \psi_n, \varphi)$. The rule does two things: First, it removes occurrences of (instances of) $\neg \varphi$ an adds instead $\chi_1, \ldots, \chi_n$ to the antecedent. Second if for some $i$, (instances of) the formulas $\psi_1, \ldots, \psi_n, \varphi, \neg \chi_1, \ldots, \neg \chi_i-1, \neg \chi_i+1, \ldots, \neg \chi_m$ are all contained in the sequent, then $\varphi$ is replaced by $\chi_i$.

Formally the the rule is applicable to a sequent $\Gamma' \vdash \Delta'$ if there is a substitution $[x \leftarrow \rho]$ with domain $x$ as above such that either

- $\varphi[x \leftarrow \rho]$ is contained in the sequent
- $\psi_1[x \leftarrow \rho], \ldots, \psi_n[x \leftarrow \rho]$ are contained in the sequent
- $\Gamma'' \vdash \Delta''$, $\land \Gamma[x \leftarrow \rho]$ is provable by the simplifier, where $\Gamma'' \vdash \Delta''$ is $\Gamma' \vdash \Delta'$ without $\neg \varphi[x \leftarrow \rho]$ or there is an $1 \leq i \leq m$, such that
  - $\chi_1, \ldots, \chi_{i-1}, \chi_{i+1}, \ldots, \chi_m$ are literals
  - $\varphi[x \leftarrow \rho]$ is contained in the sequent
  - $\psi_1[x \leftarrow \rho], \ldots, \psi_n[x \leftarrow \rho]$ are contained in the sequent
  - $\neg \chi_1[x \leftarrow \rho], \ldots, \neg \chi_i-1[x \leftarrow \rho], \neg \chi_i+1[x \leftarrow \rho], \ldots, \neg \chi_m[x \leftarrow \rho]$ are contained in the sequent
  - $\Gamma'' \vdash \Delta''$, $\land \Gamma[x \leftarrow \rho]$ is provable by the simplifier, where $\Gamma'' \vdash \Delta''$ is $\Gamma' \vdash \Delta'$ without $\varphi[x \leftarrow \rho]$ or there is an $1 \leq i \leq m$, such that
  - $\chi_1, \ldots, \chi_{i-1}, \chi_{i+1}, \ldots, \chi_m$ are literals
  - $\varphi[x \leftarrow \rho]$ is contained in the sequent
  - $\psi_1[x \leftarrow \rho], \ldots, \psi_n[x \leftarrow \rho]$ are contained in the sequent
  - $\neg \chi_1[x \leftarrow \rho], \ldots, \neg \chi_i-1[x \leftarrow \rho], \neg \chi_i+1[x \leftarrow \rho], \ldots, \neg \chi_m[x \leftarrow \rho]$ are contained in the sequent
  - $\Gamma'' \vdash \Delta''$, $\land \Gamma[x \leftarrow \rho]$ is provable by the simplifier, where $\Gamma'' \vdash \Delta''$ is $\Gamma' \vdash \Delta'$ without $\varphi[x \leftarrow \rho]$

In the first case $\neg \varphi[x \leftarrow \rho]$ is removed and $\chi_1[x \leftarrow \rho], \ldots, \chi_n[x \leftarrow \rho]$ are added to the antecedent. In the second case $\varphi[x \leftarrow \rho]$ is replaced by $\chi_1[x \leftarrow \rho], \ldots, \chi_n[x \leftarrow \rho]$.

Example: $\vdash (x - 1 + 1 = x) \leftrightarrow \neg x \equiv 0$

Example: $\vdash \text{primefactorlist}(x) = \text{primefactorlist}(y) \leftrightarrow x = y$

Example: $\vdash x \leq y \rightarrow (y \leq x \leftrightarrow x = y)$

Example: $\vdash p(x) = y \leftrightarrow x = s(y)$ (where ‘s’ and ‘p’ are successor and predecessor on integers)

Example: $\vdash \neg x = 0 \land \neg y = 0 \rightarrow (\text{pred}(x) = \text{pred}(y) \leftrightarrow x = y)$

Example: $\vdash \text{zerop}(x) \leftrightarrow x = 0$

Example: $\vdash \text{member}(x, \text{delete}(y, \text{set})) \leftrightarrow \neg x = y \land \text{member}(x, \text{set})$

Pragmatics:

These rules are used in all cases, where a formula $\varphi$ is equivalent to a simpler formula $\chi$ (under some conditions). Sometimes $\chi$ is a conjunction as in the last example. Consider the last one: Here we want to replace $\text{member}(x, \text{delete}(y, \text{set}))$ in the antecedent by the two simpler formulas $\neg x = y$ and $\text{member}(x, \text{set})$. But normally we do not want the same thing in the succedent to happen (since we would get a conjunction in the succedent which leads to a case distinction). Instead we want to replace $\text{member}(x, \text{delete}(y, \text{set}))$ in the succedent by $x = y$ if we have $\text{member}(x, \text{set})$ in the succedent too (since under the condition $\neg \text{member}(x, \text{set})$ member(x,delete(y,set)) is equivalent.
to the \( x = y \). Similarly we want \( \text{member}(x, \text{delete}(y, \text{set})) \) in the succedent to be replaced by \( \text{member}(x, \text{set}) \) if we have \( x = y \) in the succedent. These three cases are covered by the rule. If you really want to introduce the conjunction use

\[
\vdash \text{member}(x, \text{delete}(y, \text{set})) \leftrightarrow \neg \neg (\neg x = y \land \text{member}(x, \text{set}))
\]

**Disjunctive Equivalence Rules**

A disjunctive equivalence rule has the form

\[
\Gamma \vdash \psi_1 \land \ldots \land \psi_n \rightarrow (\varphi \leftrightarrow \chi_1 \lor \ldots \lor \chi_m)
\]

where \( \varphi \) is an atomic formula. \( \psi_1, \ldots, \psi_n \) must be literals. The variables of the theorem must be a subset of \( \bar{x} = \text{Vars}(\psi_1, \ldots, \psi_n, \varphi) \).

The rule does two things: First, it removes occurrences of (instances of) \( \varphi \) and adds instead \( \chi_1, \ldots, \chi_m \) to the succedent. Second if for some \( i \), (instances of) the formulas \( \neg \psi_1, \ldots, \neg \psi_n, \neg \varphi, \chi_1, \ldots, \chi_{i-1}, \chi_{i+1}, \ldots, \chi_m \) are all contained in the sequent, then \( \neg \varphi \) is replaced by \( \neg \chi_i \).

Formally the rule is applicable to a sequent \( \Gamma' \vdash \Delta' \) if there is a substitution \([x \leftarrow \rho] \) with domain \( x \) as above such that either

- \( \varphi[x \leftarrow \rho] \) is contained in the sequent
- \( \neg \psi_1[x \leftarrow \rho], \ldots, \neg \psi_n[x \leftarrow \rho] \) are contained in the sequent
- \( \Gamma'' \vdash \Delta'' \land \Gamma[x \leftarrow \rho] \) is provable by the simplifier, where \( \Gamma'' \vdash \Delta'' \) is \( \Gamma' \vdash \Delta' \) without \( \varphi[x \leftarrow \rho] \) or there is an \( 1 \leq i \leq m \), such that
  - \( \neg \chi_1, \ldots, \neg \chi_{i-1}, \neg \chi_{i+1}, \ldots, \neg \chi_m \) are literals
  - \( \neg \psi_1[x \leftarrow \rho], \ldots, \neg \psi_n[x \leftarrow \rho] \) are contained in the sequent
  - \( \chi_1[x \leftarrow \rho], \chi_{i-1}[x \leftarrow \rho], \chi_{i+1}[x \leftarrow \rho], \ldots, \chi_m[x \leftarrow \rho] \) are contained in the sequent
  - \( \Gamma'' \vdash \Delta'' \land \Gamma[x \leftarrow \rho] \) is provable by the simplifier, where \( \Gamma'' \vdash \Delta'' \) is \( \Gamma' \vdash \Delta' \) without \( \neg \varphi[x \leftarrow \rho] \)

In the first case \( \varphi[x \leftarrow \rho] \) is removed and \( \neg \chi_1[x \leftarrow \rho], \ldots, \neg \chi_n[x \leftarrow \rho] \) are added to the antecedent. In the second case \( \neg \varphi[x \leftarrow \rho] \) is replaced by \( \neg \chi_i[x \leftarrow \rho] \).

Example: \( \text{member}(x, \text{insert}(y, \text{set})) \leftrightarrow x = y \lor \text{member}(x, \text{set}) \)

**Pragmatics:**

Same as for Conjunctive Case. Here \( \varphi \) is equivalent to a disjunction instead of a conjunction (or equivalently \( \neg \varphi \) is equivalent to a conjunction).

**Implication Rules**

Implication rules have the form

\[
\Gamma \vdash \psi_1 \land \ldots \land \psi_n \rightarrow (\varphi \rightarrow \chi)
\]

where \( \varphi \) is a literal. \( \psi_1, \ldots, \psi_n \) must be literals. The variables of the theorem must be a subset of \( \bar{x} = \text{Vars}(\psi_1, \ldots, \psi_n, \varphi) \). The rule removes occurrences of (instances of) \( \neg \varphi \) in the sequent and adds \( \chi \) instead.

Formally the rule is applicable to a sequent \( \Gamma' \vdash \Delta' \) if there is a substitution \([x \leftarrow \rho] \) with domain \( x \) as above such that either
• $\neg \varphi[x \leftarrow \rho]$ is contained in the sequent
• $\psi_1[x \leftarrow \rho], \ldots, \psi_n[x \leftarrow \rho]$ are contained in the sequent
• $\Gamma'' \vdash \Delta''$, $\wedge \Gamma[x \leftarrow \rho] \vdash \rho$ is provable by the simplifier, where $\Gamma'' \vdash \Delta''$ is $\Gamma' \vdash \Delta'$ without $\neg \varphi[x \leftarrow \rho]

Then the rule removes $\neg \varphi[x \leftarrow \rho]$ from the sequent and adds $\chi[x \leftarrow \rho]$ to the antecedent.

Example: $\neg \text{pred}(x) = x \rightarrow \neg x = 0$
Example: $\vdash \text{subtree}(t, \text{right}(t)) \rightarrow t = \text{emptytree}$
Example: $\vdash \text{regexp-less}(r, \text{alt1}(r)) \rightarrow \neg \text{altp}(r)$ (from the LEX-Example, regexp-less is the term-order on regular expressions, altp tests $r$ for being a regexp with two alternatives, alt1 selects the first)

Pragmatics: These rules are mainly used in two cases. In the first, $\varphi$ is equivalent to $\varphi$, but we only want to replace $\varphi$ in the antecedent by $\psi$. The other case is, when $\varphi$ is “nearly equivalent” to $\psi$, i.e. if $\varphi$ implies $\psi$, and the reverse implication would hold, if an additional axiom for an underspecified function or predicate would be added. Typical such “additional axioms” are axioms for selector applications on the wrong summand in data specifications (like $0 - 1 = 0$ or $\text{left}(\text{emptytree}) = \text{emptytree}$ as in the first two examples). The cases, where $\varphi$ is not equivalent to $\chi$ are dangerous to use, since simplifier rules like $\vdash x + y < z \rightarrow y < z$ (e.g. to prove that $x + y < z$, $\neg y < z \vdash$ is valid), are not invertible (i.e. the conclusion does not imply the premise).
Non invertible rules lose information, that may be necessary in other cases (e.g. in $(0 + 1) + y < z$, $y + 1 = z \vdash$, which is no longer provable). Therefore they should be avoided.

A.4 Forward rules

Forward rules are used to add information to sequents. They have the form:

$$\Gamma \vdash \varphi \rightarrow \psi$$

where $\varphi$ is a conjunction of literals and $\psi$ is an arbitrary first order formula. All free variables $x$ of the sequent must be contained in the free variables of $\varphi$. The rule adds instances of $\psi$ to the sequent, if $\varphi$ and $\Gamma$ can be proved.

Formally the rule is applicable to a sequent $\Gamma' \vdash \Delta'$ if there is a substitution $[x \leftarrow \rho]$ with domain $x$ as above such that

• $\varphi[x \leftarrow \rho]$ is contained in the sequent
• $\Gamma'' \vdash \Delta''$, $\wedge \Gamma[x \leftarrow \rho] \vdash \rho$ is provable by the simplifier, where $\Gamma'' \vdash \Delta''$ is $\Gamma' \vdash \Delta'$ without $\varphi[x \leftarrow \rho]

Then the rule adds $\psi[x \leftarrow \rho]$ to the sequent, if it has not already been added in the current branch of the proof.

Pragmatics: The rule is typically used to compute the closure of transitive relations
Example: $\vdash x < y \land y < z \rightarrow x < z$
Example: $\vdash x \leq y \land y < z \rightarrow x < z$
and to derive weaker formulas from stronger ones (e.g. to establish preconditions for rewrite rules).
Example: $\vdash \text{succ}(x) < y \rightarrow x < y$
One should be careful not to use the same rule as a forward rule and as an axiom rule. Otherwise the added formula will immediately be removed by the axiom rule.
A.5 Elimination rules

Elimination rules try to eliminate “bad” function symbols such as ‘-1’, ‘-’, ‘car’ and ‘cdr’, ‘delete’ for “good” ones (in the examples the good function symbols are ‘+1’, ‘+’, ‘cons’, ‘insert’). This is done by replacing a term like ‘x -1’ or ‘x - y’ by a new variable z and replacing ‘x’ by ‘z +1’ resp. ‘z + y’. The rule produces appropriate side conditions, while the heuristic tries to prove (or find) these conditions to be applicable. The heuristic and the rule are based on elimination rules of the following form:

\[ \Gamma \vdash \varphi \rightarrow (x_1 = t_1 \land \ldots \land x_n = t_n \leftrightarrow \exists \ y. v = t \land \psi) \]

where \( \varphi \) is a conjunction of literals and \( \psi \) is an arbitrary first order formula. For every \( i \), the variables of the sequent are contained in the two disjoint sets \( \text{Vars}(t_i) \) and \( \{x_1, \ldots, x_n\} \). \( v, x_1, \ldots, x_n \) are pairwise different variables. The variables \( y \) are not free in the sequent. \( v \) does not occur in \( \psi \) or \( t \) (but in every \( t_i \)). \( x_1, \ldots, x_n \) do not occur in \( \varphi \) or \( \Gamma \). The preconditions \( \varphi \) and the sideformula \( \psi \) are optional.

Example:
\[ \vdash x \neq 0 \rightarrow (x = y -1 \leftrightarrow y = x +1) \]
Example:
\[ \vdash \neg x < y \rightarrow z = x - y \leftrightarrow x = z + y \]
Example: x \( \neq \) nil \( \vdash \) (a = car(x) \( \land \) y = cdr(x) \( \leftrightarrow \) x = cons(a,y))
Example: x \( \neq \) nil \( \vdash \) (y = cdr(x) \( \leftrightarrow \) \( \exists \) a. x = cons(a,y) \( \land \) a = car(x)) (eliminates only cdr, not car)
Example: b \( \neq \) 0 \( \vdash \) (k = x div b \( \land \) r = x mod b \( \leftrightarrow \) x = k * b + r \( \land \) r < b)
Example: \( \vdash x \in \text{set} \rightarrow \) (set’ = delete(x,\text{set}) \( \leftrightarrow \) set = insert(x,\text{set’}) \( \land \) \( \neg \) x \( \in \) set’)
Example: prodp(x) \( \rightarrow \) (x = sel_1(x) \( \land \) \ldots \land x_n = sel_n(x) \( \leftrightarrow \) x = mkprod(x_1,\ldots,x_n))

where \( \text{mkprod} \) and \( \text{prodp} \) are constructor and predicate of a product type, \( \text{sel}_1, \ldots, \text{sel}_n \) are the selectors of its factors
Example: \( \vdash \) set \( \neq \) \emptyset \( \rightarrow \) set’ = butmin(\text{set}) \( \leftrightarrow \) \( \exists \) x. set = insert(x,\text{set’}) \( \land \forall \) y. \( y \in \text{set’} \rightarrow x < y \)
Example: \( \vdash n = m \mod 2^d \leftrightarrow \exists \ k. m = k * 2^d + r \land r < d \)

The elimination rule tries to find a substitution \( \sigma \), such that \( \sigma(t_i) \) is in the sequent, \( \sigma(v) \) is a variable, and all variables in \( t_i \) are instantiated to terms with disjoint variables (we do not consider terms like ‘x div x’, ‘(y - z) div x’ or ‘x div (x - y)’ in the sequent, but we do consider ‘x div y + x’). If such a substitution can be found, there is also a variant, which renames \( x_1, \ldots, x_n \) and \( y \) (these are exactly all other variables in the rule) to new variables relative to the sequent under consideration. Then it creates a sidegoal, in which \( \Gamma \) and \( \varphi \) must be proved (the heuristic must find the literals of \( \varphi \) in the sequent and must prove \( \Gamma \)). The main goal of the rule is then created by first replacing every \( \sigma(t_i) \) by \( \sigma(x_i) \), then substituting \( \sigma(t) \) for \( \sigma(v) \) and finally adding \( \sigma(\psi) \). So, if the substitution is empty, the proof tree created looks like:

\[
\frac{\Gamma' \vdash \Delta', \varphi \land \land \Gamma \therefore \psi, \Gamma'[t \leftarrow v][\psi \leftarrow t] \vdash \Delta'[t \leftarrow v][\psi \leftarrow t]}{\Gamma' \vdash \Delta'}
\]

Examples (sidegoals are trivial, hence omitted)

\[
\begin{align*}
\vdash & b \neq 0, r < b \vdash \varphi(k * b + r, b, r, k) \\
\vdash & b \neq 0 \vdash \varphi(a, b, \text{amodb}, \text{adivb}) \\
\vdash & \text{cons}(a, y) \neq \text{nil} \vdash \varphi(a, y, \text{cons}(a, y)) \\
\vdash & x \neq \text{nil} \vdash \varphi(\text{car}(x), \text{cdr}(x), x) \\
\vdash & \neg z + y < x \vdash \varphi(z, z + y, y) \\
\vdash & \neg x < y \vdash \varphi(x - y, x, y) \\
\vdash & x \in \text{insert}(x, \text{set’}) \rightarrow \varphi(\text{insert}(x, \text{set’}), \text{set’}) \\
\vdash & x \in \text{set} \rightarrow \varphi(\text{set}, \text{del}(x, \text{set}))
\end{align*}
\]
$\text{insert}(x, \text{set}') \neq \emptyset, \forall y. y \in \text{set}' \rightarrow x < y \vdash \varphi(\text{insert}(x, \text{set}'), \text{set}')$

$\text{set} \neq \emptyset \vdash \varphi(\text{set}, \text{butmin(set)})$

$r < 2^n \vdash \varphi(k + 2^n + r, r)$

$\vdash \varphi(m, m \text{mod} 2^n)$

Several examples show that elimination may produce redundant formulas. The redundant formulas are always some of the instances of preconditions in $\varphi$ or $\Gamma$. We could have formulated elimination rules, such that these redundant formulas would be eliminated (by moving them to the right side of the equivalence), but then we would not have had the choice if side conditions should be found by matching or by proof. In the first (third) example, a $(x)$ must be a variable, while $b (y)$ may be an arbitrary term.

A.6 Cut rules

The weak cut and the cut heuristic apply the cut rule with a literal $\varphi$, which is computed by using the information present in cut rules. Cut rules must have one of the four forms

• $\varphi \land \varphi_1 \land \ldots \varphi_n \rightarrow \sigma = \tau$
  where $\varphi$ is a literal with free variables contained in the variables of $\sigma$.

• $\varphi \land \varphi_1 \land \ldots \varphi_n \rightarrow \psi$
  where $\psi$ must be a literal. The free variables of $\varphi$ must be a subset of free($\psi$).

• $\varphi \land \varphi_1 \land \ldots \varphi_n \rightarrow \psi \leftrightarrow \chi$
  where $\psi$ must be a literal. The free variables of $\varphi$ must be a subset of free($\psi$).

• $\psi \leftrightarrow \varphi \land \varphi_1 \land \ldots \varphi_n$
  where $\psi$ is a literal, and free($\varphi$) $\subseteq$ free($\psi$)

All forms cause a cut with an instance $\Theta(\varphi)$ of $\varphi$. For the first form the cut is done, if $\Theta(\sigma)$ is a free term (i.e. none of it’s variables are bound) of the sequent. For the other forms a cut is done, if $\Theta(\psi)$ is a free literal that occurs in the sequent. In the second case the cut is done regardless of whether the literal is in the antecedent or in the succedent.

The weak cut heuristic computes the set of all possible cut formulas according to the cut rules. Reflexive equations, literals already present on the toplevel of the current goal and all literals with which a cut has already been done are excluded from this set. One of the most often computed formulas from this set is then chosen for the application of the cut tactic. The cut heuristic computes the same set of potential cut formulas, but additionally adds all literals, which are present in the sequent, but not on top-level (e.g. a literal $\varphi_1$ in an antecedent formula $\varphi_1 \lor \varphi_2$ is added). Again the most often computed formula is chosen.
Appendix B

Mixfix Notation and Overloading

In this section we will use the symbols $c$ for a constant with sort $s$, $f$ for a function with argument sorts $s_1, \ldots, s_n$ and target sort $s'$, and $p$ for a predicate with argument sorts $s_1, \ldots, s_n$ (the target sort is $\text{bool}$).

B.1 Classification of operations

KIV allows the following classes of operations:

- usual operations
- postfix operations
- prefix operations
- infix operations
- methodfix operations
- outfix operations
- outinfix operations
- outprefix operations
- oupostix operations

A symbol is of one of the last four classes, if and only if it starts with one of the “closing bracket” characters $\lfloor, \rfloor, \{, [\}$ or $\lceil$. It is then called an outfix symbol. For each closing bracket character, there is the corresponding “opening bracket”: $\lfloor, \rfloor, \{, [\} and $\rfloor$. Closing bracket characters cannot be used otherwise in symbols (opening bracket characters cannot be used in symbols at all). Note that the symbol $]$ alone is reserved for program formulas.

So “$\times a$” can only be an operation of one of the first four classes, while “$\}abc$” always belongs to one of the last four classes.

Infix operations have one of the priorities “1 left”, “1 right”, “2 left”, …, “15 left”, “15 right”.

B.2 Overloading

Operations from each of the eight classes can be overloaded only with operations of the same class. Two infix operations can be overloaded only if they have the same priority. When an operation is overloaded and used in a place, where its type cannot be inferred, it is always possible to clarify which type is intended by writing “$c :: s$”, “$f :: (s_1 \times \ldots \times s_n \to s')$”, or “$p :: (s_1 \times \ldots \times s_n \to \text{bool}$)” instead.
B.3 Generalities on Usage

Operations are used in five places:

- *declaration* in the signature
- *usage* of functions and predicates in formulas
- on the right hand side of a *renaming* in morphisms
- in a *generated by*-clause
- on both sides of a renaming of an operation

The following sections will define the syntax for each class of operations and each of the first three places. In *generated-by*-clauses and on the left hand side of equations, outfix symbols are always paired with their opening bracket (with a space in the middle). Otherwise just the symbol is written. As an example

```
set generated by { }, ∪
```

is a correct generated by clause (where the outfix symbol “}” has been paired with the opening bracket “{”).

Some other general remarks:

- In declarations it is always possible to use a comma instead of “×” as the separator between sorts.
- Usually the right hand side of a renaming gets the same type (and priority) as the left hand side. When the left hand side is an outfix symbol, but the right hand side is not, then as a default the right hand side is a usual operation.
- ∧, ∨, →, ↔ are infix operations with priority 4 right, 3 right, 2 right and 1 right.
- ¬ is a prefix predicate, equality is an infix operation of priority 5 right.
- The strength of binding is (form strong to weak): usual application, postfix function, prefix function, infix operation with priority 15 (associative to the left or right), 14, . . . , infix operation with priority 5, equality, postfix predicate, prefix predicate, infix operation with priority 4, . . . , infix operation with priority 1.
- Note that quantifiers (and lambda expressions) do not care about binding strength. They always bind to the farthest right as possible.

B.4 Usual Operations

Usual operations can be constants or functions or predicates (with any number of arguments).

**Declaration**

```
constants c : s;
functions f : s₁ × . . . × sₙ → s';
predicates p : s₁ × . . . × sₙ;
```

**Usage**  Usual applications are written \( f(t) \) and \( p(t) \).
Renaming If the left hand side of a renaming is a postfix, prefix, methodfix or infix operation, but the right hand side should be a usual operation, then at the right hand side a “prio 0” must be added. For example

\[ \cup \rightarrow \text{union pri}o \ 0; \]

is a correct renaming of the infix operation “\( \cup \)” to the usual operation “union”.

B.5 Postfix Operations

A postfix operation can only be a function or a predicate with one argument. Often postfix operations start with a dot.

Declaration Postfix declarations are written with a dot before the operation symbol

functions . f : s₁ → s’;
functions . . sel : s₁ → s’;
predicates . p : s₁;

Usage Postfix application is written \( t f \), \( t . \text{sel} \) and \( t p \).

Renaming If the left hand side of a renaming is not a postfix operation, but has one argument, and the renamed symbol should be postfix, then at the right hand side a dot must be added before the symbol. For example

\[ \text{succ} \rightarrow . +1; \]

is a correct renaming of the usual operation “succ” to the postfix operation “+1”.

B.6 Prefix Operations

A prefix operation can only be a function or a predicate with one argument. The symbol \# is often used as a prefix size function.

Declaration Prefix declarations are written with a dot after the operation symbol

functions f . : s₁ → s’; # . : s → nat; predicates p . : s₁;

Usage Prefix application is written \( f t \), \# \( t \) and \( p t \).

Renaming If the left hand side of a renaming is not a prefix symbol, but has one argument, and the renamed symbol should be prefix, then at the right hand side a dot has to be added behind the symbol. For example

\[ \text{size} \rightarrow \# . \]

is a correct renaming of the usual operation “size” to the prefix operation “\#”.


B.7 Methodfix Operations

Methodfix operations are used to imitate object oriented method calls, where the first argument is before the function, and the other arguments are written behind the function in brackets. In OO languages, a dot is used to separate the first argument and the function. The separating dot is unnecessary here, but usually the function name starts with a dot (note that a space must be used between the first argument and the function symbol, if the function does not start with a dot).

**Declaration**  Methodfix declarations are written with a dot after the operation symbol

\[
\text{functions} \ : \ f(\ ) : s_1 \times s_2 \times s_3 \rightarrow s' ; \ \text{predicates} \ : \ p(\ ) : s_1 ;
\]

**Usage**  Methodfix application is written \(e_1.f(e_2,e_3)\) and \(e_1.p()\).

**Renaming**  If the left hand side of a renaming is not a methodfix symbol, and the renamed symbol should be methodfix, then the right hand side has a dot before the symbol, and a dot in brackets behind it. For example

\[
f \rightarrow \ . . f ( . ) ;
\]

is a correct renaming of the usual operation "\(f\)" to the methodfix operation "\(.f\)".

B.8 Infix Operations

A infix operation can only be a function or a predicate with two arguments. An infix operation has a priority from "1 left", "1 right", "2 left", \ldots, "15 left", "15 right". The number determines the binding strength (the higher the stronger is the binding). "right" signals a right-associative operation like conjunction (\(x \land y \land z\) is the same as \(x \land (y \land z)\)), a negative a left-associative operation (like subtraction "\(-\)" in the library).

**Declaration**  Infix declarations are written with a dot before and after the operation symbol and a \texttt{prio n left} or \texttt{prio n right} at the end of the declaration. The default priority is "9 right". The “right” at the end can be omitted.

\[
\text{functions} \ : \ f : s_1 \times s_2 \rightarrow s' \ \texttt{prio n left} ; \ \text{predicates} \ : \ p : s_1 \times s_2 \ \texttt{prio n} ;
\]

**Usage**  Infix application is written \(t_1 f t_2\) and \(t_1 p t_2\). Note that a space is needed between the arguments and the function symbol (except when the arguments are in brackets).

**Renaming**  If the left hand side of a renaming is not an infix symbol, but has two arguments, and the renamed symbol should be infix, then at the right hand side a dot has to be added before and after the symbol and a priority may be given (like in a declaration) For example

\[
\text{sub} \rightarrow \ . . \ \texttt{prio 9 left} ; \ \text{add} \rightarrow \ . . ;
\]

are correct renamings.
APPENDIX B. MIXFIX NOTATION AND OVERLOADING

B.9 Outfix Operations

An outfix operation has one argument.

Declaration

Outfix operations are declarated by giving the opening bracket, then a dot and then the operation symbol:

\[
\text{functions} \{\cdot\} : s_1 \to s'_1; \quad \text{predicates} \{\cdot\} : p : s_1 ; (\text{: a predicate testing the argument to be a one-element set :) }) ;
\]

Usage

Application of an outfix symbol is written opening bracket, argument, operation symbol like in “\{a\}s”. Note that in a formula you do not need to add space before or after the opening bracket or before the operation symbol. Space is only needed behind the operation symbol.

Renaming

If the left hand side of a renaming has one argument, and should be renamed to an outfix operation write opening bracket, dot, operation symbol on the right hand side.

For example

\[
elemtoset \to \{\cdot\}
\]

is a correct renaming.

B.10 Outinfix Operations

An outinfix operation always has two arguments. Outinfix operations are often used as constructors for pairs.

Declaration

Outinfix operations are declarated by giving the opening bracket, then a dot, a comma, another dot and then the operation symbol:

\[
\text{functions} \lfloor \cdot, \cdot \rfloor : s_1 \times s_2 \to s'_1;
\]

Usage

Application of an outinfix symbol is written opening bracket, first argument, operation symbol like in “\{a, b\}x”. Note that in a formula you do not need to add space before or after the opening bracket or before the operation symbol. Space is only needed behind the operation symbol.

Renaming

If the left hand side of a renaming has one argument, and should be renamed to an outinfix operation write opening bracket, dot, comma, dot, operation symbol on the right hand side:

For example

\[
\text{mkpair} \to \lfloor \cdot, \cdot \rfloor p
\]

is a correct renaming.
APPENDIX B. MIXFIX NOTATION AND OVERLOADING

B.11 Outprefix Operations

An outprefix operation has at least two arguments. Outinfix operations are often used to apply a function to arguments, as array selectors or as array modifiers.

**Declaration**  Outprefix operations are declared by giving a dot, the opening bracket, then a dot, and finally the operation symbol:

\[
\text{functions} . [ . ] : s_1 \times s_2 \times \ldots s_n \rightarrow s';
\]

Note that only one dot is used between the brackets, regardless how many arguments the outprefix operation has.

**Usage**  Application of an outprefix symbol is written first arguments, opening bracket, remaining arguments separated with comma, and finally operation symbol like in “f [ a, b]x”. Note that in a formula you do not need to add space before or after the opening bracket or before the operation symbol. Space is only needed behind the operation symbol.

There is a special rule for using outprefix symbols in assignments: Instead of writing \( f := f[ t_1, t_2, \ldots t_n, t_{n+1}]x \) it is possible to write \( f[ t_1, t_2, \ldots t_{n}]x := t_{n+1} \), if the operation “[ ]x is overloaded to have type \( s_1 \times s_2 \times \ldots s_n \rightarrow s_{n+1} \rightarrow s_1 \) and type \( s_1 \times s_2 \times \ldots s_n \rightarrow s_{n+1} \). Usually \( s_1 \) will be an array sort (or a dynamic function), the overloaded function of the first type is an array modifier (or a modification operation for dynamic functions), the second an array selector (or application of a dynamic function).

**Renaming**  If the left hand side of a renaming has one argument, and should be renamed to an outprefix operation write dot, opening bracket, dot operation symbol on the right hand side (again only one dot between the brackets, regardless how many arguments the function has).

For example

\[
\text{update-array} \rightarrow . [ . ]p
\]

is a correct renaming.

B.12 Outpostfix Operations

An outpostfix operations has two or more arguments. The second and all other arguments are written (separated by commas) in between the brackets.

**Declaration**  Outpostfix operations are declarated by giving a dot, the opening bracket, another dot and then the operation symbol:

\[
\text{functions} [ [ . ] ] : s_1 \times s_2 \rightarrow s'; \text{ predicates} [ [ . ] ]p : s_1 \times s_2;
\]

Only one dot between the brackets is used, even if the operation has more than two arguments.

**Usage**  Application of an outpostfix symbol is written opening bracket, first argument, operation symbol, second argument like in “[[ a] s v". Note that in a formula you do not need to add space before or after the opening bracket or before the operation symbol. Space is only needed behind the operation symbol. The program formulas of Dynamic Logic are like outpostfix operations.
Renaming If the left hand side of a renaming has two arguments, and should be renamed to an out postfix operation write dot, opening bracket, dot, operation symbol on the right hand side. For example

\[ \text{eval} \rightarrow . [ . ] \]

is a correct renaming.

### B.13 Some examples

The following examples of expressions give the expressions without brackets, and with full bracketing. They use infix operations ‘+’ and ‘*’ of priorities ‘9 right’ and ‘10 right’, a postfix function ‘+1’, a prefix function ‘#’, and a postfix predicate ‘evenp’. Also the predefined operations = (infix with prio 5 right), \( \land \) (infix with prio 4 right), \( \lor \) (infix with prio 3 right), \( \rightarrow \) (infix with prio 2 right), \( \leftrightarrow \) (infix with prio 1 right), \( \neg \) (prefix predicate) are used.

\[
\begin{align*}
\# n +1 & \equiv \#(n +1) \\
\# m + n & \equiv (\# m) + n \\
\# n +1 & \equiv \#(n +1) \\
\# m + n & \equiv (\# m) + n \\
f(n) +1 & \equiv (f(n)) +1 \\
m + n \text{ evenp} & \equiv (m + n) \text{ evenp} \\
m \land k \text{ evenp} & \equiv m \land (k \text{ evenp}) \\
\# m \text{ evenp} & \equiv (\# m) \text{ evenp} \\
x + y + z & \equiv (x + (y + z)) \\
x \ast y + x & \equiv (x \ast y) + x \\
x + y +1 & \equiv (x + (y +1)) \\
\# x +1 & \equiv \#(x +1) \\
\# x + x & \equiv (\# x) + x \\
\neg x \land y & \equiv (\neg x) \land y \\
(x \rightarrow y \supset \neg z \land y \leftrightarrow z) & \equiv ((x \rightarrow y) \supset (\neg z, y \leftrightarrow z)) \\
\forall x. f(x) \land g(x) & \equiv \forall x. (f(x) \land g(x)) \text{ but } (\forall x. f(x)) \land g(x)
\end{align*}
\]
Appendix C

Syntax of Dynamic Logic

This grammar of the PPL parser uses the following assumptions described:

1. The start symbol is ‘start’.
2. Terminals are written in **bold**.
3. Pseudoterminals are written in *italics*. There are three kinds: PPL-Variables (which start with a backquote), meta variables (which start with a $) and other symbols. In the last two cases the lexical analysis returns a token, depending on the entry of the symbol in the current signature. If the symbol has no entry the pseudoterminal returned is `sym`. Otherwise it may be `sort`, `xov`, `fct`, `proc`, `praeifixct`, `prefixprd`, `postfixct`, `postfixprd` or `infixct[{l||r}]`.  
4. In priorities “right” may be omitted.
5. `[ ... ]` means: is optional.
6. `{ ... }` means: repeated any times (even 0 times)
7. `{ ... }^+$ means: repeated at least one time.
8. `{ ... //sep}^*$ means: repeated any times, separated by `sep`.
9. `{ ... //sep}^+` means: repeated any times, separated by `sep`.
10. `{ ... | ... }^+$ means: alternative.
11. Superscript index i is always in `{0 ... 4, 6 ... 15}`

```
term     :   term1 [ite term ; term ]
        :   
start   :   expr
        | prog_2
        | sorts {sym // , }^+ ;
        | constants defconstlist_aux
        | functions {deffct }^+
        | predicates {defprd }^+
        | defvarlist
        | defprocidt
        | procdeclmv
        | vlnv
        | vdlnv
        | flmv
```
APPENDIX C. SYNTAX OF DYNAMIC LOGIC

| sort |
| fct |
| proc |
| [] |
| {proc // , }^+ |
| {fct // , }^+ |
| {expr_nonppl // , }^+ |
| expr_nonppl | expr_nonppl |
| . vl . |

crosscomma : × |
| , |

symorplusorstern : sym |
| + |
| * |

gsym : symbol |
| specialsymbol |

eextsymorplusorstern : extsym |
| + |
| * |

eextsym : extsymbol |
| specialsymbol |

specialsymbol : → |
| < |
| > |
| × |
| in |
pplvarassym : symbol |
| in |

string1 : symbol1 |
| × |
| in |
| ∀ |
| ∃ |
| < |
| > |
| → |

string : sort |
| string1 |

symbol : sort |
| symbol1 |

eextsymbol : sort |
| extsymbol1 |
APPENDIX C. SYNTAX OF DYNAMIC LOGIC

symbol1 : regsymbol |
| outsymbol

extsymbol1 : regsymbol |
| extoutsymbol

regsymbol : infixfctr^i |
| infixfctl^i |
| const |
| xov |
| fct |
| postfixfct |
| praefixfct |
| postfixprd |
| praefixprd |
| infixfctl110 |
| infixfctl111 |
| infixfctl112 |
| infixfctl113 |
| infixfctl114 |
| infixfctl115 |
| proc |
| sym |

outsymbol : rceiloutfct |
| rceiloutinfct |
| rceiloutprefct |
| rceiloutsym |
| rflooroutfct |
| rflooroutinfct |
| rflooroutprefct |
| rfloorsym |
| rsemoutfct |
| rsemoutinfct |
| rsemoutprefct |
| rsemsym |
| rquineoutfct |
| rquineoutinfct |
| rquineoutprefct |
| rquinesym |
| rgeschwoutfct |
| rgenschwoutinfct |
| rgenschwoutprefct |
| rgeschwoutsym |
| rceiloutpostfct |
| rflooroutpostfct |
| rquineoutpostfct |
| rgenschwoutpostfct |
| rsemoutpostfct |
| reckigzwoutpostfct |
| reckigzwusym |
}
APPENDIX C. SYNTAX OF DYNAMIC LOGIC

extoutsymbol : ] rceiloutfct
| ] rceiloutinfct
| ] rceiloutprefct
| ] reilsym
| ] rflooroutfct
| ] rflooroutinfct
| ] rflooroutprefct
| ] rfloorsym
| ] rsempoutfct
| ] rsempoutinfct
| ] rsempoutprefct
| ] rsemsym
| ] rquinaoutfct
| ] rquinaoutinfct
| ] rquinaoutprefct
| ] rquinesym
| ] rgenschwoutfct
| ] rgenschwoutinfct
| ] rgenschwoutprefct
| ] rgenschwsym
| ] rceiloutpostfct
| ] rflooroutpostfct
| ] rquinaoutpostfct
| ] rgenschwoutpostfct
| ] rsempoutpostfct
| ] reckigzuoutpostfct
| ] reckigzusym

infixfct : infixfctl1
| infixfctl2
| infixfctl3
| infixfctl4
| infixfctl5orgreater
| =
| infixfctl6
| infixfctl7
| infixfctl8
| infixfctl9
| infixfctl10
| infixfctl11
| infixfctl12
| infixfctl13
| infixfctl14
| infixfctl15
| infixfctr1
| infixfctr2
| infixfctr3
| infixfctr4
| infixfctr5
| infixfctr6
| infixfctr7
APPENDIX C. SYNTAX OF DYNAMIC LOGIC

123

| infixfctr8  |
| infixfctr9  |
| infixfctr10 |
| infixfctr11 |
| infixfctr12 |
| infixfctr13 |
| infixfctr14 |
| infixfctr15 |

fctnonpraefix :
  fct
  infixfct
  postfixfct
  postfixprd

fct :
  fctnonpraefix
  praefixedfct
  praefixedprd
  *

extfct :
  fctnonpraefix
  praefixedfct
  praefixedprd
  *

  rceilocoutfct
  rceilocoutinfct
  rceilocoutprefct
  rceilsym
  rflooreoutfct
  rflooreoutinfct
  rflooreoutprefct
  rfloorsym
  rsemoutfct
  rsemoutinfct
  rsemoutprefct
  rsemsym
  rquineoutfct
  rquineoutinfct
  rquineoutprefct
  rquinesym
  { rgescwoutfct
  { rgescwoutinfct
  { rgescwoutprefct
  { rgescwusym
  { rceilocpostfct
  { rfloorepostfct
  rquineoutpostfct
  { rgescwpostfct
  { rsemoutpostfct
  { reckigzoutpostfct
  { reckigzusym
  } ]

reclsym :
  rceilocoutfct
APPENDIX C. SYNTAX OF DYNAMIC LOGIC

\[
\begin{align*}
\text{rfloorsym} & : \text{rflooroutfct} \mid \text{rflooroutinfct} \mid \text{rflooroutprefct} \mid \text{rflooroutpostfct} \mid \text{rfloorsym} \\
\text{rsemsym} & : \text{rsemoutfct} \mid \text{rsemoutinfct} \mid \text{rsemoutprefct} \mid \text{rsemoutpostfct} \mid \text{rsemsym} \\
\text{rquinesym} & : \text{rquineoutfct} \mid \text{rquineoutinfct} \mid \text{rquineoutprefct} \mid \text{rquineoutpostfct} \mid \text{rquinesym} \\
\text{rgeschwsym} & : \text{rgeschwoutfct} \mid \text{rgeschwoutinfct} \mid \text{rgeschwoutprefct} \mid \text{rgeschwoutpostfct} \mid \text{rgeschwsym} \\
\text{reckigzusym} & : \text{reckigzuoutpostfct} \mid \text{reckigzusym} \\
\text{var} & : \text{ppl\_variable} \mid \text{var\_nonppl} \\
\text{var\_nonppl} & : \text{xov} \mid \text{const} \mid \text{xmv} \\
\text{vl} & : [\text{ppl\_variable}] \mid \text{var} , [\text{vlmv}] , \{\text{var} / / , \}^+ \mid \text{var\_nonppl} \mid \text{var} , \{\text{var} / / , \}^+ [\text{vlmv}] , \{\text{var} / / , \}^+ \] \\
\text{expr} & : \text{ppl\_variable} \mid \text{expr\_nonppl} \\
\text{expr\_nonppl} & : \text{expr\_nonppl1} \mid \text{expr\_nonppl2} \mid \text{expr} \\
\text{expr5} & : \text{expr5a} \mid \text{praefexprd} \text{expr5}
\end{align*}
\]
APPENDIX C. SYNTAX OF DYNAMIC LOGIC

| expr_nonppl5 : expr5a_nonppl |
| praeexprd expr5          |
| boxdiaexpr                |

| expr5a : expr5b          |
| expn5a postfixrd        |

| expr5a_nonppl : expr5b_nonppl |
| expn5a postfixrd         |

| expr5b : expr6            |
| leexpr5                   |
| reexpr5                   |

| expr5b_nonppl : expr_nonppl6 |
| leexpr5                   |
| reexpr5                   |

| leexpr5 : lepxn5 infixfctl5orgreater expr6 |
| expn6 infixfctl5orgreater expr6 |
| lepxn5 = expr6            |
| expn6 = expr6            |
| lepxn5 ≠ expr6           |
| expn6 ≠ expr6           |

| infixfctl5orgreater : infixfctl5 |
| >                      |

| infixfctl5 : infixfctl5 |
| <                      |

| infixfctr2 : infixfctr2 |
| →                     |

| reexpr5 : expn6 infixfctr5 reexpr5 |
| expn6 infixfctr5 expr6        |

| infixfctr9 : infixfctr9 |
| +                      |
| ×                      |

| infixfctr10orsterm : infixfctr10 |
| *                      |

| term5 : term5a         |
| praeexprd term5       |
| boxdiaterm            |

| term5a : term5b        |
| term5a postfixrd       |

| term5b : expn6         |
APPENDIX C. SYNTAX OF DYNAMIC LOGIC

\[ lterm5 \rightarrow \begin{cases} lterm5 \text{ infixfctl5 expn6} \\ expn6 \text{ infixfctl5 expn6} \\ lterm5 = \text{ expn6} \\ \text{ expn6} = \text{ expn6} \end{cases} \]

\[ expn5 \rightarrow \begin{cases} \text{ expn5a} \\ boxdiaexpn \end{cases} \]

\[ expn5a \rightarrow \begin{cases} \text{ expn5b} \\ expn5a \text{ postfixprd} \end{cases} \]

\[ expn5b \rightarrow \begin{cases} \text{ expn6} \\ \text{ lexpn5} \\ \text{ rexpn5} \end{cases} \]

\[ lexpn5 \rightarrow \begin{cases} lexpn5 \text{ infixfctl5orgreater expn6} \\ \text{ expn6} \text{ infixfctl5orgreater expn6} \\ lexpn5 = \text{ expn6} \\ \text{ expn6} = \text{ expn6} \\ lexpn5 \neq \text{ expn6} \\ \text{ expn6} \neq \text{ expn6} \end{cases} \]

\[ rexpn5 \rightarrow \begin{cases} \text{ expn6} \text{ infixfctr5 rexpn5} \\ \text{ expn6} \text{ infixfctr5 expn6} \end{cases} \]

\[ \text{ expr}^i \rightarrow \begin{cases} \{ \text{ expr}^{i+1} \text{ // infixfctl5 } \}^+ \\ \{ \text{ expr}^{i+1} \text{ // infixfctr5 } \}^+ \end{cases} \]

\[ \text{ expn}^i \rightarrow \begin{cases} \{ \text{ expn}^{i+1} \text{ // infixfctl5 } \}^+ \\ \{ \text{ expn}^{i+1} \text{ // infixfctr5 } \}^+ \end{cases} \]

\[ \text{ term}^i \rightarrow \begin{cases} \{ \text{ term}^{i+1} \text{ // infixfctl5 } \}^+ \\ \{ \text{ term}^{i+1} \text{ // infixfctr5 } \}^+ \end{cases} \]

\[ \text{ expr\_nonppl}^i \rightarrow \begin{cases} \{ \text{ expr\_nonppl}^{i+1} \text{ // infixfctl5 } \}^+ \\ \{ \text{ expr\_nonppl}^{i+1} \text{ // infixfctr5 } \}^+ \end{cases} \]

\[ \text{ expr16} \rightarrow \begin{cases} \text{ praefixexpr} \\ \text{ outpraefixexpr} \\ \text{ expr17} \end{cases} \]

\[ \text{ expr\_nonppl16} \rightarrow \begin{cases} \text{ praefixexpr} \\ \text{ outpraefixexpr} \\ \text{ expr\_nonppl17} \end{cases} \]

\[ \text{ expn16} \rightarrow \begin{cases} \text{ praefixexpn} \\ \text{ outpraefixexpn} \\ \text{ expn17} \end{cases} \]
praefixexpr : praefixfct expr16
          | * expr16
          | \texttt{ampersand} expr16
praefixexpn : praefixfct expn16
            | * expn16
            | \texttt{ampersand} expn16
outpraefixexpr : ⌈ expr receiloutprefct expr16
                | ⌈ expr rssemoutprefct expr16
                | ⌈ expr rflooroutprefct expr16
                | ⌈ expr rqineoutprefct expr16
                | ⌈ expr rgeschwoutprefct expr16
outpraefixexpn : ⌈ expr receiloutprefct expn16
                | ⌈ expr rssemoutprefct expn16
                | ⌈ expr rflooroutprefct expn16
                | ⌈ expr rqineoutprefct expn16
                | ⌈ expr rgeschwoutprefct expn16
expr17 : expn17 postfixct
        | expr18
expr_nonppl17 : expn17 postfixct
              | expr_nonppl18
expn17 : expn17 postfixct
        | expn18
expr18 : expn18
        | allex
expr_nonppl18 : expn_nonppl18
              | allex
boxdiaexpr : [ progseq_nonparasg ] expr5
             | < progseq_nonparasg > expr5
boxdiaexpn : [ progseq_nonparasg ] expn5
            | < progseq_nonparasg > expn5
boxdiaterm : [ progseq_nonparasg ] term5
            | < progseq_nonparasg > term5
allex : \exists \textit{v} . expr
       | \forall \textit{v} . expr
expn18 : ppl_variable
       | expn_nonppl18
expn_nonppl18 : exprmv
termmv
xmv
\begin{verbatim}
const xov |
fctnonpraefix ( {expr // , }\* ) |
expn_nonppl18 [ expr rceiloutputfct |
expn_nonppl18 [ expr reckigzwoutputfct |
expn_nonppl18 [ expr ] |
expn_nonppl18 [ expr flooroutputfct |
expn_nonppl18 [ expr rsemoutputfct |
expn_nonppl18 [ expr ruineoutputfct |
expn_nonppl18 { expr rgeschwoutputfct |
( expr ) |
outfixexpr |
|
outfixexpr : | expr rceiloutputfct |
| expr , expr rceiloutinfct |
| expr flooroutputfct |
| expr , expr flooroutinfct |
| expr rsemoutputfct |
| expr , expr rsemoutinfct |
| expr ruineoutputfct |
| expr , expr ruineoutinfct |
| expr rgeschwoutputfct |
| expr , expr rgeschwoutinfct |
|
fl : [{expr // , }\* , ] flmv , {expr // , }\* ] |
| {expr // , }\* |
|
prog : | ppl_variable |
| prog_2 |

prog_2 : | progmv |
| asg |
| proc ( apl ) |
| while expr do prog |
| if expr then prog [ else prog ] |
| skip |
| abort |
| { {prog // ; }\* } |
| { {prog // ; }\* } |
| bwhile expr do {prog // ; }\* times term |
| loop {prog // ; }\* times term |
| let vdl in prog |
| choose vl with expr in prog |
| choose vl with expr in prog ifnone prog |
| \|

parasg : [{asg // | }\* | ] parasyme [ | {asg // | }\* ] |
| {asg // | }\* | asg |
|
| asg : | var {{(expr // , )\* )* := term |
| var := [? |
|
progseq_nonparasg : | prog_2 ; {prog // ; }\* ] |
\end{verbatim}
APPENDIX C. SYNTAX OF DYNAMIC LOGIC

| ppl_variable
|:
| vdl : [\{vardecl // , \}^+ , vdlmv [, \{vardecl // , \}^+ ]
| {vardecl // , }^*
|:
| vardecl : var = expr
| var = [?]
| ppl_variable
|:
| pdl : [\{procdecl // ; \}^+ , pdlmv [, \{procdecl // ; \}^+ ]
| ppl_variable
| {procdecl // ; }^*
|:
| procdecl : proc ( param ) \{ \{ prog // ; \}^+ \}
| proc ( param ) \{ \{ prog // ; \}^+ \}
| procdeclmv
|:
| param : \{ var // , \}^+ [; \{ var // , \}^+ ]
| var \{ var // , \}^+
| ε
|:
| apl : \{ expr // , \}^* ; \{ var // , \}^+
|:
| spec0 : ppl_variable
| pplvarassym
|:
| symren : sort → sym
| const → sym
| proc → sym
| xov → sym
|:
| renoutsymbol : [ . rceiloutfct
| [ . rceilsym [ , ]
| . rflooroutfct
| . rfloorsym [ , ]
| [ . rsemoutfct
| [ . rsemoutfct
| [ . rsemoutfct
| [ . rsemoutfct
| [ . rsemoutfct
| r. rguineoutfct
| r. rguinesym [ , ]
| r. rguineoutfct
| r. rguinesym [ , ]
| [ . rgeschwartfct
| [ . rgeschwarzsym [ , ]
| [ . rceiloutprefct .
| [ . rceiloutprefct .
| r. rguineoutprefct .
| r. rguineoutprefct .
| [ . rgeschwoutprefct .
| [ . rsemoutprefct .
| [ . rsemoutprefct .
| [ . rsemoutprefct .
| [ . rsemoutprefct .
| [ . rsemoutprefct .
| [ . rceiloutpostfct
| [ . rceilsym
| [ . rflooroutpostfct
| [ . rloorsym
| [ . rsemoutpostfct
| [ . rsemoutpostfct
| r. rguineoutpostfct
| r. rguinesym
| ]
APPENDIX C. SYNTAX OF DYNAMIC LOGIC

| . { . rgeschwoutinfect | . rgeschwsym | . reckigzauoutpostfct | . reckigzusym | . } |
| . , . receiloutpostfct |
| . , . receilsym |
| . , . rflooroutpostfct |
| . , . rfloorsym |
| . , . rseminoutpostfct |
| . , . rsemssym |
| . , . rquineoutpostfct |
| . , . rquinesym |
| . , . rgeschwoutinfect |
| . , . rgeschwsym |
| . , . reckigzauoutpostfct |
| . , . reckigzusym |

| . , . | ⌈ . , . rceilsym |
| . , . | ⌊ . , . rfloorsym |
| . , . | [ ⌜ . , . rquinesym |
| . , . | ⌯ . , . rgeswoutinfect |
| . , . | ⌦ . , . rgeschwsym |
| . , . | { | . , . reckigzauoutpostfct |
| . , . | ⌯ . , . reckigzusym |

| . | { |
| . | ⌦ |
| . | ⌦ |
| . | ⌦ |
| . | ⌦ |

| . | ⌦ |
| . | ⌦ |
| . | ⌦ |
| . | ⌦ |
| . | ⌦ |

| prdandprio | : sym [ , ] |
| . sym |

| defconstlist_aux | : [defconstlist_aux | sym : sort ; |
| | [defconstlist_aux | sym , {sym / / , }+ : sym ; |

| deffct | : sym : {sort / / crosscomma }+ → sort ; |
| . symorplusorsterne : sort crosscomma sort → sort ; |
| . sym : sort → sort ; |
| . sym : sort → sort ; |
| . receilsym : sort → sort ; |
| . rfloorsym : sort → sort ; |
| . rsemsym : sort → sort ; |
| . rquinesym : sort → sort ; |
| . rgeschwsym : sort → sort ; |
| . . , . receiloutpostfct |
| . . , . receilsym |
| . , . rflooroutpostfct |
| . , . rfloorsym |
| . , . rseminoutpostfct |
| . , . rsemssym |
| . , . rquineoutpostfct |
| . , . rquinesym |
| . , . rgeschwoutinfect |
| . , . rgeschwsym |
| . , . reckigzauoutpostfct |
| . , . reckigzusym |

| | { |
| | ⌦ |
| | ⌦ |
| | ⌦ |
| | ⌦ |

| defprd | : sym ; |
| sym : {sort / / crosscomma }+ ; |
APPENDIX C. SYNTAX OF DYNAMIC LOGIC

defproclist : procedures \{defproc \}+
defproc : sym mode 

mode : ( {sort // ,} *) [: sortlist_nonempty_or_sort [: ( {mode // ,} )+] ]

sortlist_nonempty_or_sort : ( {sort // ,} )+
sort

defvarlist : variables defvarlist_aux 
defvarlist_aux : [defvarlist_aux | sym : sort ]
| [defvarlist_aux | sym , {sym // ,} ]+: sort ;
Appendix D

Syntax of Algebraic Specifications and Modules

This grammar uses the same assumptions as described in the syntax of dynamic logic in appendix

1. The start symbol is ‘start’.
2. Terminals are written in **bold**.
3. Pseudoterms are written in *italics*.
4. In priorities “right” may be omitted.
5. [...] means: is optional.
6. {...}∗ means: repeated any times (even 0 times)
7. {...}+ means: repeated at least one time.
8. {.../sep}∗ means: repeated any times, separated by sep.
9. {.../sep}+ means: repeated any times, separated by sep.
10. {.../...}+ means: alternative.
11. Attached number j is always from {1 ... 9}.

```
start : expr  
    | sequent  
    | module_specific {pattern forbidden rule applyrule }∗  
    | {gen }+  
    | implementation  
    | lemmas {string ; sequent ; optionals }∗  
    | sort  
    | fct  
    | proc  
    | module  
    | spec_nonppl  
    | []  
    | [{proc // , }]∗  
    | [{fct // , }]∗  
```
morphism
signature signature end signature
;

optionals : [usedfor [{string // , }]+ ;] [remark ]
[usedfor {string // , }* ;] comment [remark ]
;

pattern : pattern {expr // , }* ⊢ {expr // , }* ;
;

forbidden : [forbidden [{expr // , }]+ ⊢ [{expr // , }]+ ;]
;

rule : rule rule1 ;
;

rule1 : insert lemma string substitution
execute call
contract call left
callleft
callright
assignleft
assignright
loopunwindright
loopunwindleft
loopexitleft
loopexitright
splitleft
splitsright
weakening {exprposarg // , }*+
cut_formula expr
allleft
existsright
insert_speclemma namesubsarg
insert_neg_rewritelemma negrewritearg
insert_pos_rewritelemma posrewritearg
ifleft
ifright
case_distinction exprposarg
;

substitution : with [{expr // , }*] → [{expr // , }*]
with ppl_variable
ε
;

exprposarg : right j
left j
;
calleleft : call left j
call_left
;
callright : call right j
call_right
;
assignleft : assign left j
assign_left
;
assignright : assign right j
APPENDIX D. SYNTAX OF ALGEBRAIC SPECIFICATIONS AND MODULES

assign_right

loopunwindleft : loopunwind_left j
   | loopunwind_left

loopunwindright : loopunwind_right j
   | loopunwind_right

loopexitleft : loopexit_left j
   | loopexit_left

loopexitright : loopexit_right j
   | loopexit_right

splitleft : splite left j
   | splite_left

splitright : split right j
   | split_right

allleft : all left j with [ {expr // , }^+ ] discard
   | all_left with [ {expr // , }^+ ] discard

existsright : exists right j with [ {expr // , }^+ ] discard
   | exists_right with [ {expr // , }^+ ] discard

ifleft : if left j
   | if_left

ifright : if right j
   | if_right

discard : and discard
   | ε

namessubsarg : string from specification string [ string ] substitution
   | string from specification string substitution

negrewritearg : [string from specification ] string substitution
   | string from specification string [ string ] substitution

posrewritearg : [string from specification ] string substitution
   | string from specification string [ string ] substitution

applyrule : [applyrule_1 ; ]
   | applyrule_11 ;
   | applyrule_once ;
   | applyrule_unique ;
   | applyrule_always ;

crosscomma : ×
   |
APPENDIX D. SYNTAX OF ALGEBRAIC SPECIFICATIONS AND MODULES

\begin{verbatim}
symorphosor stern : sym
| +
| *
:

sym : see grammar for DL
:

extsymorphosor stern : extsym
| +
| *
:

extsym : extsymbol
| specialsymbol
:

specialsymbol : \rightarrow
| <
| >
| ×
| in
:

pplvarassym : symbol
| in
:

string1 : symbol1
| ×
| in
| ∀
| ∃
| <
| >
| \rightarrow
:

string : sort
| string1
:

symbol : sort
| symbol1
:

extsymbol : sort
| extsymbol1
:

symbol1 : regsymbol
| outsymbol
:

extsymbol1 : regsymbol
| extoutsymbol
:

regsymbol : infixfctri
| infixftli
\begin{itemize}
\item const
\item xov
\item fct
\item postfxtc
\item praeftc
\end{itemize}
\end{verbatim}
APPENDIX D. SYNTAX OF ALGEBRAIC SPECIFICATIONS AND MODULES

postfixprd
praefixprd
proc
sym
;

outsymbol : rceiloutfct
rceiloutinfc
rceiloutprefct
rceilsym
rflooroutfct
rflooroutinfc
rflooroutprefct
rfloorsym
rsemoutfct
rsemoutinfc
rsemoutprefct
rsemsym
rquineoutfct
rquineoutinfc
rquineoutprefct
rquinesym
rgeschwoutfct
rgeschwoutinfc
rgeschwoutprefct
rgeschwsym
rceiloutpostfct
rflooroutpostfct
rquineoutpostfct
rgeschwoutpostfct
rsemoutpostfct
reckigzoutpostfct
reckigzusym
;

extoutsymbol : [ rceiloutfct
rceiloutinfc
rceiloutprefct
rceilsym
rflooroutfct
rflooroutinfc
rflooroutprefct
rfloorsym
rsemoutfct
rsemoutinfc
rsemoutprefct
rsemsym
rquineoutfct
rquineoutinfc
rquineoutprefct
rquinesym
rgeschwoutfct
rgeschwoutinfc
rgeschwoutprefct
rgeschwsym
rceiloutpostfct
rflooroutpostfct
rquineoutpostfct
rgeschwoutpostfct
rsemoutpostfct
reckigzoutpostfct
reckigzusym

]
APPENDIX D. SYNTAX OF ALGEBRAIC SPECIFICATIONS AND MODULES

fct : see grammar for DL

extfct : praeфикс
       praeфиксprd
       *
       ceiloutfct
       ceiloutinfct
       ceiloutprefct
       ceil sym
       flooroutfct
       flooroutinfct
       flooroutprefct
       floor sym
       semoutfct
       semoutinfct
       semoutprefct
       sem sym
       quineoutfct
       quineoutinfct
       quineoutprefct
       quinesym
       geschwoutfct
       geschwoutinfct
       geschwoutprefct
       geschwsym
       ceiloutpostfct
       flooroutpostfct
       quineoutpostfct
       geschwoutpostfct
       semoutpostfct
       reckigzuoutpostfct
       reckigzusym
       
       rceilsym : rceiloutfct
                  rceiloutinfct
                  rceiloutprefct
                  rceilsym
                  flooroutfct
                  flooroutinfct
                  flooroutprefct
                  floor sym
                  rceilsym
                  
       rfloorsym : flooroutfct
                  flooroutinfct
                  flooroutprefct
APPENDIX D. SYNTAX OF ALGEBRAIC SPECIFICATIONS AND MODULES

<table>
<thead>
<tr>
<th>rflooroutpostfct</th>
<th>rfloorsym</th>
</tr>
</thead>
<tbody>
<tr>
<td>rsemoutfct</td>
<td>rsemoutinfct</td>
</tr>
<tr>
<td>rsemoutprefct</td>
<td>rsemoutpostfct</td>
</tr>
<tr>
<td>rsemsym</td>
<td></td>
</tr>
<tr>
<td>rquineoutfct</td>
<td>rquineoutinfct</td>
</tr>
<tr>
<td>rquineoutprefct</td>
<td>rquineoutpostfct</td>
</tr>
<tr>
<td>rquinesym</td>
<td></td>
</tr>
<tr>
<td>rgeschwoutfct</td>
<td>rgeschwoutinfct</td>
</tr>
<tr>
<td>rgeschwoutprefct</td>
<td>rgeschwoutpostfct</td>
</tr>
<tr>
<td>rgeschwsym</td>
<td></td>
</tr>
<tr>
<td>reckigzuoutpostfct</td>
<td></td>
</tr>
<tr>
<td>reckigzusym</td>
<td></td>
</tr>
<tr>
<td>expr</td>
<td>see grammar for DL</td>
</tr>
<tr>
<td>sequent</td>
<td>see grammar for DL</td>
</tr>
<tr>
<td>pdl</td>
<td>see grammar for DL</td>
</tr>
<tr>
<td>spec</td>
<td>unionspec0</td>
</tr>
<tr>
<td>spec0</td>
<td>spec</td>
</tr>
<tr>
<td>spec_nonppl</td>
<td>spec1 [+ {spec0 // + }+]</td>
</tr>
<tr>
<td>spec1</td>
<td>basicspec</td>
</tr>
<tr>
<td></td>
<td>unionspec</td>
</tr>
<tr>
<td></td>
<td>genericspec</td>
</tr>
<tr>
<td></td>
<td>enrichedspec</td>
</tr>
<tr>
<td></td>
<td>actualizedspec</td>
</tr>
<tr>
<td></td>
<td>renamedspec</td>
</tr>
<tr>
<td></td>
<td>asmspec</td>
</tr>
<tr>
<td></td>
<td>basedataspec</td>
</tr>
<tr>
<td></td>
<td>gendataspec</td>
</tr>
<tr>
<td>unionspec</td>
<td>union_specification unionspec0 end union_specification</td>
</tr>
<tr>
<td>unionspec0</td>
<td>spec0 + {spec0 // + }+</td>
</tr>
</tbody>
</table>
APPENDIX D. SYNTAX OF ALGEBRAIC SPECIFICATIONS AND MODULES

basicspec : specification signature [axioms genlistandaxioms] end specification

| {gen}+ {declaration [string : | pdl]}

gen : {sort // /, }+ generated_by {extsymplusorstern // /, }+ ;
| {sort // /, }+ freely_generated_by {extsymplusorstern // /, }+ ;

signature : [sorts \{sym // /, \}+] ;
| [constants constdeflist_aux ]
| [functions \{fctdef\}]+
| [predicates \{prddef\}]+ 
| [procedures \{procdef\}] * 
| [variables vardeflist_aux ]

enrichedspec : enrich \{spec // /, \}+ enrichedpart

enrichedpart : with signature end enrich
| with signature axioms genlistandaxioms end enrich

genericspec : generic specification
| parameter spec [using \{spec // /, \}+] 
| genericpart

genericpart : target signature end generic specification
| target signature axioms genlistandaxioms end generic specification

actualizedspec : actualize spec
| with \{spec // /, \}+
| by morphism [[\{symren // /, \}+] [\, \]]
| end actualize

renamedspec : rename spec
| by morphism [[\{symren // /, \}+] [\, \]]
| end rename

morphism : morphism [[\{symren // /, \}+] [\, \]] end_morphism

symren : sort \rightarrow sym
| const \rightarrow sym
| fctren
| procren
| xov \rightarrow sym

procren : proc \rightarrow sym

fctren : extfct \rightarrow symplusorstern
| extfct \rightarrow . sym
| extfct \rightarrow sym .
| extfct \rightarrow . symplusorstern . [prionleftright]
| extfct \rightarrow sym prio_0
| extfct \rightarrow renoutsymbol
renoutsymbol
: [. rceiloutfct
| [. rceilsym [, ]
| [. rflooroutfct
| [. rfloorsym [, ]
| [. rsemoutfct
| [. rsemsym [, ]
| $. rquineoutfct
| $. rquinesym [, ]
} $. rgenschwoutfct
} $. rgenschwsym [, ]
| [. rceiloutprefct .
| $. rflooroutprefct .
| $. rquineoutprefct .
| $. rgenschwoutprefct .
| $. rgenschwsym .
| [. rceiloutpostfct
| [. rceilsym
| [. rflooroutpostfct
| [. rfloorsym
| [. rsemoutpostfct
| [. rsemsym
| $. rquineoutpostfct
| $. rquinesym
} $. rgenschwoutinfct
} $. rgenschwsym
| $. reckigzuoutpostfct
| $. reckigzusym
| [. ]
| [. . rceiloutpostfct
| [. . rceilsym
| [. . rflooroutpostfct
| [. . rfloorsym
| [. . rsemoutpostfct
| [. . rsemsym
| $. . rquineoutpostfct
| $. . rquinesym
} $. rgenschwoutinfct
} $. rgenschwsym
| $. reckigzuoutpostfct
| $. reckigzusym
| [. . ]
;

asmspec : asm specification procsym
[using {spec // , }+ ]
signature
[input variables {var // , }+ ]
state variables {var }+
initial state expr
asm rule procsym
declaration prepdl
edasm specification
;
APPENDIX D. SYNTAX OF ALGEBRAIC SPECIFICATIONS AND MODULES

prepdl : see pdl in grammar for DL, but parameterlists in the declaration (param) or in calls (apl) are optional

If left off, they are computed from the variables occurring in the bodies

basicdataspec : data specification

\[
\begin{align*}
\text{using} & \{ \text{spec} //, \} + \\
\{ \text{sym} = \{ \text{constructordef} // \}^+ ; \}^+ \\
\text{variables} & \text{vardeflist}_{\text{aux}} \\
\text{size functions} & \{ \text{fctdef} \}^+ \\
\text{order predicates} & \{ \text{prddef} \}^+ \\
\end{align*}
\]

end data specification

gendataspec : generic data specification

\[
\begin{align*}
\text{parameter} & \text{spec} \\
\text{using} & \{ \text{spec} //, \} + \\
\{ \text{sym} = \{ \text{constructordef} // \}^+ ; \}^+ \\
\text{variables} & \text{vardeflist}_{\text{aux}} \\
\text{size functions} & \{ \text{fctdef} \}^+ \\
\text{order predicates} & \{ \text{prddef} \}^+ \\
\end{align*}
\]

end generic data specification

constructordef : sym \[(\{ \text{selector} //, \}^+) \text{ with } \text{prdandprio} \]

\[
\begin{align*}
\text{sym with prdandprio} \\
\cdot \text{symorphusorstern} \cdot (\text{selector, selector}) [\text{prioleftright} ] \text{ with prdandprio} \\
\cdot \text{sym (selector) [with prdandprio] } \\
\cdot \text{sym (selector) [with prdandprio] } \\
\end{align*}
\]

selector : [ ] sym : sym

prdandprio : sym [ ]

\]
APPENDIX D. SYNTAX OF ALGEBRAIC SPECIFICATIONS AND MODULES

prioleftright : {1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9} \{left \| right \| \epsilon\}

constdeflist_aux : [constdeflist_aux | sym : sym]
| [constdeflist_aux | sym, \{sym //,\} \{\}\}+ : sym

fctdef : sym : \{sym // crosscomma \}\}+ \rightarrow sym;
| sym : sym \rightarrow sym;
| sym : sym \rightarrow sym;
| rceilsym : sym \rightarrow sym;
| rfloorsym : sym \rightarrow sym;
| rsemsym : sym \rightarrow sym;
| rquinesym : sym \rightarrow sym;
| rgenschwsym : sym \rightarrow sym;
| rceilsym : sym crosscomma sym \rightarrow sym;
| rfloorsym : sym crosscomma sym \rightarrow sym;
| rsemsym : sym crosscomma sym \rightarrow sym;
| rquinesym : sym crosscomma sym \rightarrow sym;
| rgenschwsym : sym crosscomma sym \rightarrow sym;
| rceilsym : sym crosscomma sym \rightarrow sym;
| rfloorsym : sym crosscomma sym \rightarrow sym;
| rsemsym : sym crosscomma sym \rightarrow sym;
| rquinesym : sym crosscomma sym \rightarrow sym;
| rgenschwsym : sym crosscomma sym \rightarrow sym;

prddef : [sym : ] sym;
| sym : \{sym // crosscomma \}\}+;
| sym : sym \rightarrow sym;
| rceilsym : sym;
| rfloorsym : sym;
| rsemsym : sym;
| rquinesym : sym;
| rgenschwsym : sym;
| rceilsym : sym crosscomma sym;
| rfloorsym : sym crosscomma sym;
| rsemsym : sym crosscomma sym;
| rquinesym : sym crosscomma sym;
| rgenschwsym : sym crosscomma sym;
| rceilsym : sym crosscomma sym;
| rfloorsym : sym crosscomma sym;
| rsemsym : sym crosscomma sym;
| rquinesym : sym crosscomma sym;
| rgenschwsym : sym crosscomma sym;
| rceilsym : sym crosscomma sym;
| rfloorsym : sym crosscomma sym;
| rsemsym : sym crosscomma sym;
| rquinesym : sym crosscomma sym;
| rgenschwsym : sym crosscomma sym;
| rceilsym : sym crosscomma sym;
| rfloorsym : sym crosscomma sym;
| rsemsym : sym crosscomma sym;
| rquinesym : sym crosscomma sym;
| rgenschwsym : sym crosscomma sym;
APPENDIX D. SYNTAX OF ALGEBRAIC SPECIFICATIONS AND MODULES

| . . rsensym : sym crosscomma sym ; |
| . rquinesym : sym crosscomma sym ; |
| . rgeschwsym : sym crosscomma sym ; |

procdef : sym premode ;

premode : ( {sym // , }* ) [: symlist_nonempty_or_sym [: ( {premode // , }* ) ] ]

symlist_nonempty_or_sym : ( {sym // , }* )

sortlist_nonempty_or_sort : ( {sort // , }* )

vardeflist_aux : [vardeflist_aux ] sym : sym ;

module : module

| export spec |
| refinement |
| modimpl end module |

modimpl : implementation

import spec

[procedures {procdef }* ]
[variables vardeflist_aux ]

declaration pdl

| implementation ppl_variable |

implementation : implementation

import spec

[procedures {procdef }* ]
[variables vardeflist_aux ]

declaration pdl

| implementation ppl_variable |

refinement : refinement

[representation of sorts {sym implements sort ; }* ]
[representation of operations {progclause }* ]

[equality {sym : sort ; }* ]

progclause : sym implements const ;

| sym implements fct ; |
Appendix E

The Patterns Heuristic

**Remark:** The name of this heuristic has been changed in the Scala-based system from ‘module specific’ to the more adequate ‘patterns’ (the heuristic is available in both specifications and modules). The file in which pattern entries are stored is also called patterns now and starts with keyword ‘patterns’ instead of ‘module specific’. The text below still uses the old name. Also the text below uses %
 .. 
 to enclose text that must be typed into the patterns file.

E.1 Introduction

While proving the verification conditions of a module, the user is often confronted with the following phenomenon: In many proofs, the next rule to apply on the current goal depends only a few relevant formulas, which are common to all these goals. Typical examples are:

- procedure calls (on recursive procedures), for which the current input will lead to no further recursive calls when executing the procedure like in (only relevant formulas are shown!)

  \[ \vdash (\text{add#}(x,0;y)) \ y = y_0 \]

  (in a module implementing natural numbers by binary words) or

  \[ \vdash (\text{union#}(l_1,\text{nil};l_2)) \ l_2 = l_3 \]

  (in a module implementing sets by lists).

- two procedure calls, for which executing the first will lead to a call of the second (with the correct arguments for the second) like in (only relevant formulas are shown!)

  \[ \langle \text{member#}(x,l;b) \rangle \ b \vdash \langle \text{member#}(x,\text{cdr}(l);b) \rangle \ b \]

  (in a module implementing sets by lists).

- situations, in which `insert lemma` with a simple often used theorem closes the goal. An example is the theorem

  \[ \langle r#(x) \rangle \ \text{true} \vdash \langle r#(\text{cdr}(x)) \rangle \ \text{true} \]

  where `r#` terminates, iff its argument is an ordered list (in a module implementing sets by ordered lists).

These regularities can be automated in the cosi strategy using the ‘module-specific’ heuristic.
E.2 Using the heuristic

The module-specific heuristic allows the user to specify sequent patterns and rule patterns. Together they form a module specific entry, which is one entry in the list of all entries.

Sequent patterns are schemata, describing a class of goals (i.e. sequents). The heuristic tries to match the current goal against the sequent patterns of all module specific entries (in the order in which they are in the list). Matching a goal against a sequent pattern may be successful or unsuccessful. If it is unsuccessful for all patterns, the heuristic does nothing. If the heuristic finds a first pattern that matches the goal (i.e. for which the match is successful), the success yields a substitution. The substitution is applied on the corresponding rule pattern, giving a rule and its arguments. Then this rule is applied on the goal. If the rule is not applicable, there will be an error message, and the heuristic will do nothing.

It is possible to specify that a module specific entry should only be applied once in a proof branch, or that it should be applied only once with the same substitution.

The module specific entries are defined in the file

\[ \text{\langle projectdir\rangle/modules/\langle module-name\rangle/module-specific} \]

for each module. This file is loaded when beginning to work on a module. It can be reloaded at any time using the command Reload Module Specific. The heuristic itself does not read the file, so you have to reload the file yourself if you modify it.

The file starts with the keyword module specific.

The following sections give detailed informations on the syntax of an module specific entry, the action of matching sequents to sequent patterns and its substitution result, and the syntax of rule patterns.

E.3 Syntax of module specific entries

A module specific entry is given by

- A first sequent pattern (that must be present in the goal)
- Optional, a second sequent pattern (that must not be present)
- A rule pattern
- Optional, how often the rule should be applied

The concrete syntax of a module specific entry is

\[
\text{pattern: sequent;}
\]  
\[
\text{[forbidden: sequent;]}
\]  
\[
\text{rule: name [rulearg];}
\]  
\[
\text{[apply rule: (1 | 11 | once | unique | always);]}
\]

If there is no forbidden sequent you can omit the whole line. If you omit apply rule it defaults to apply rule: always. name in the rule pattern is the rule that should be applied. For more informations about the rule pattern and apply rule see the corresponding sections below.

Formulas used in the two are really formula patterns mich may use place holders called meta variables for various syntactic classes: There are predefined metavariables for formulas, boolean expressions (no quantifier or dynamic logic), programs, variable declaration lists (vdl’s), variable lists (vl’s), terms and variables. Matching instantiates them with concrete formulas, boolean expressions etc.. All meta variables begin with a $-sign.

The following table lists some predefined meta variables.
APPENDIX E. THE PATTERNS HEURISTIC

<table>
<thead>
<tr>
<th>meta variables for</th>
<th>variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formulas</td>
<td>$\phi$, $\phi_0$, $\phi_1$, $\phi_2$</td>
</tr>
<tr>
<td></td>
<td>$\psi$, $\psi_0$, $\psi_1$, $\psi_2$</td>
</tr>
<tr>
<td></td>
<td>$\chi$, $\chi_0$, $\chi_1$, $\chi_2$</td>
</tr>
<tr>
<td>Programs</td>
<td>$\alpha$, $\beta$</td>
</tr>
<tr>
<td>Boolean Expressions</td>
<td>$\epsilon$, $\epsilon_0$</td>
</tr>
<tr>
<td>Variable List</td>
<td>$v_l$, $v_l_1$, $v_l_2$</td>
</tr>
<tr>
<td>Variable Declaration List</td>
<td>$v_{dl}$, $v_{dl}<em>1$, $v</em>{dl}_2$</td>
</tr>
</tbody>
</table>

Now you see some examples which explain when and how you could use the module specific heuristics, and how meta variables work.

1. Consider a procedure call to a procedure add#, adding two integers, where the second argument is zero. If every such procedure call in the succedent should be executed, the following entry (note that nil is the empty list) should be added:

   pattern: $\vdash < add\#(x, zero; xvar) > \phi$
   rule: call right;

   in the file module-specific.

2. Consider two related procedure calls, like in

   $\langle member\#(x_1, l_1; bvar) \rangle \vdash < member\#(x_2, l_2; bvar) > \phi$

   (in a module, implementing sets of elems by lists of elems). If the left procedure should be executed, whenever $x_1 = x_2$ and $l_2 = cdr(l_1)$, and the result of both procedures is true, add the following entry in the file module-specific:

   pattern: $\vdash < member\#(x, l; bvar) > bvar = true |- < member\#(x, cdr(l); bvar_0) > bvar_0 = true$
   rule: call left;

3. In nearly every module, a lot of goals show up that relate the restriction procedure (or its subprocedures) for different arguments. These relations are often proved by separate theorems. A typical example is the theorem $r$-rcdr

   $\langle r\#(l) \rangle true, l \neq nil \vdash < r\#(cdr(l)) > true$

   in the module implementing sets by ordered lists. To insert this theorem when the formulas of the antecedent and succedent of the theorem appear in the antecedent resp. succedent of the goal, the following module specific entry is used:

   pattern: $\vdash < r\#(l) > true |- < r\#(cdr(l)) > true$
   rule: insert lemma r-rcdr with [l] -> [l];
   apply rule: unique;

Two things have to be noticed:

(a) The theorem will be applied only once with the same substitution. Since the rule is again applicable after the insertion of the theorem, applying the rule always might lead to a loop.

(b) The condition $l \neq nil$ of the theorem does not appear in the sequent pattern. Application of the theorem makes sense only if the condition holds. However, this may very well be the case but the condition cannot be found literally in the goal. If we assume that a term $cdr(l)$ only shows up when the list $l$ is not empty we can apply the theorem and hope that the condition can be proved.
4. In many modules, the verification conditions $\text{VC}_T$ (i.e. $i-1, i-2, \ldots$) are used as theorems in other proofs. Sometimes, it is useful to insert a theorem $i-2$:

$$\langle r\#(x) \rangle \text{true} \vdash \langle f\#(x;y) \rangle \langle r\#(y) \rangle \text{true}$$

whenever the formulas $\langle r\#(x) \rangle \text{true}$ and $\langle f\#(x;y) \rangle y = y_0$ appear in the sequent to get the additional information $\langle r\#(y_0) \rangle \text{true}$. This can be done by the following module specific entry (and some declarations):

pattern: $\langle r\#(x) \rangle \text{true}, \langle f\#(x;y\text{var}) \rangle y\text{var} = y \vdash$ ;
forbidden: $\langle r\#(y) \rangle \text{true} \vdash$ ;
rule: insert lemma i-2 with $[x] \to [x]$;
apply rule: unique;

Note, that the formula $\langle r\#(y) \rangle \text{true}$ that will be added in the antecedent by inserting the theorem (and following contract call left) is explicitly prohibited to be already in the antecedent of the sequent. Otherwise the module specific heuristic would insert the theorem even though it does not add really new information. Using the declaration to apply a rule only once with the same substitution is different from the declaration to apply the rule only once, and is different from using forbidden formula patterns, because formulas may be eliminated (by weakening left, call left, etc.) and variables may be renamed, which leads to the danger of infinite loops.

### E.4 Rule patterns

A rule pattern consists of two parts,

1. a name of a rule
   Examples are
   - call left
   - call right
   - insert lemma

2. a pattern for a rule argument. This pattern is instantiated by applying $\Theta$ on it. The rule argument should be admissible for the rule.

Actually, most of the existing rules of the proof strategy are never used for the module specific heuristic, either because they are always applied by other heuristics (like the rules for symbolic execution) or because it is much too complicated to define a pattern when to apply the rule. So the following list of rules and their arguments is not complete but should cover all normal uses.

#### insert lemma

The insert lemma rule needs two informations: The name of the theorem to be inserted, given by and a substitution for the free variables of the theorem. So the rule pattern is given by

rule: insert lemma \(\text{theorem name}\) with \(\text{free variables}\) \(\to\) \(\text{term patterns}\)

The \(\text{free variables}\) must be the list of the free variables of the theorem in any order, written e.g. as \([x,y,z]\). Note that this is the same list as shown in the pop-up window, when applying the rule interactively, except that the order of the variables doesn’t matter. \(\text{term patterns}\) must be a list of term patterns of the same length. The $i$-th free variable of the first list is instantiated by the $i$-th term of the second list. Since the pattern will be instantiated with the substitution $\Theta$, you can use the meta variables of the sequent pattern.
APPENDIX E. THE PATTERNS HEURISTIC

**call left/right, split left/right**

Used interactively, you have to select the formulas to discard. Used as a module specific entry, you have to do the same. However, you can only select a formula that is matched by a formula pattern. A formula position is given by an indicator for the antecedent (call/split left) or the succedent (call/split right) and an index i. The index refers to the position of the formula pattern in the list of patterns. The first formula that matches an indicated formula pattern will be selected. If the sequent pattern is of the form

\[
\text{pattern: } < f#(x;\text{vary}) > \phi, \\
< g#(x;\text{vary}) > \phi \vdash ;
\]

the sequent is

\[
\langle g#(x:x_0) \rangle \text{ true}, \langle f#(x:x_0) \rangle \text{ true}, \langle g#(y:y_0) \rangle \text{ true}, \langle f#(y:y_0) \rangle \text{ true} \vdash
\]

and the rule is

**rule: call left**

the rule will be applied on \( \langle f#(x:x_0) \rangle \text{ true} \). If the rule is

**rule: call left 2**

the rule will be applied on \( \langle g#(x:x_0) \rangle \text{ true} \). The same holds for the rules split left/right.

**weakening**

Used interactively, you have to select the formulas to discard. Used as a module specific entry, you select a formula as described for the call left/right. Since it is possible to discard several formulas, the formula positions must be listed as

\[
\langle \text{formula positions} \rangle, \langle \text{formula positions} \rangle, \ldots
\]

Example: If the sequent pattern is of the form

\[
\text{pattern: } < f#(x;\text{vary}) > \phi, \\
< g#(x;\text{vary}) > \phi \vdash ;
\]

the sequent is

\[
\langle g#(x:x_0) \rangle \text{ true}, \langle f#(x:x_0) \rangle \text{ true}, \langle g#(y:y_0) \rangle \text{ true}, \langle f#(y:y_0) \rangle \text{ true} \vdash
\]

and the rule is

**rule: weakening left 2**

the formula \( \langle g#(x:x_0) \rangle \text{ true} \) will be discarded.

**cut formula**

The rule cut formula requires a formula as its argument. This has to be entered as

**rule: cut formula \( \text{(formula)} \);**

The formula may contain meta variables from the sequent pattern.
all left/exists right
The rules *all left* and *exists right* need the position of the quantified formula (again a formula position) and the substitution for the quantified variables (or a program). The formula position is ignored and computed by the module specific heuristic in the same manner as for *call left/right*. For example the this must be entered as

```plaintext
rule: all left 1 with (list of terms);
```

The list of terms and the program may contain meta variables of the sequent pattern.

insert spec-lemma
The rule *insert spec-lemma* needs the theorem to apply and a substitution. The theorem is specified by its name. However, since the theorem stems from another specification that was perhaps actualized several times, the unique name of the theorem consists of three parts: The specification name, the instance name, and the theorem name. The three names are the same as displayed when viewing the theorem with the command *View Spec Theorems*. If the specification was not actualized you can leave out the instance name. The substitution is entered in the same manner as for *insert lemma*, the complete rule has the form

```plaintext
rule: insert spec-lemma (theorem name) from specification 
     (specification name) [(instance name)]
     with (free variables) -> (instantiations);
```

or, if no instance name is needed,

```plaintext
rule: insert spec-lemma (theorem name) from specification 
     (specification name) with (free variables) -> (instantiations);
```

Apply Rule
As you saw in the section for the *concrete syntax* of an module specific entry, you can say how often a rule should be applied. The given parameter are 1, 11, once, unique, always.

If you want that the rule should be applied everytime the sequent pattern matches, you have to write

```plaintext
apply rule: always
```

If it should be applied only once, write

```plaintext
apply rule: once
```

or

```plaintext
apply rule: 1
```

if the rule should be applied once with the same substitution, write

```plaintext
apply rule: unique
```

or

```plaintext
apply rule: 11
```

If you omit the line *apply rule:...*, this has the same meaning as to write *apply rule: always*

Summary
The following list gives an overview over the rules and its corresponding arguments.
E.5 Matching

Matching the current goal against a module specific entry is based on finding a replacement for all metavariables such that applying the replacement on the formulas of the pattern gives a sequent that must be part of the current goal. The replacement is also applied on the forbidden sequent. This must result in a sequent without metavariables, and none of the resulting formulas may be present in the current goal for the match to be successful. The match instantiates metavariables, but also instantiates ordinary variables of the pattern (technically, this is done by replacing ordinary variables with metavariables in the pattern before attempting to compute the replacement. ‘View module specific’ shows the internally stored patterns which contain metavariables in place of ordinary variables). A variable can be instantiated with a variable only, if it does occur in the pattern as

- the left hand side of an assignment or let-binding
- a bound variable of a lambda or quantifier
- a reference parameter of a call

It can be instantiated with a term (no quantifiers, no dynamic logic), if it occurs somewhere else in a program (e.g. is the right hand side of an assignment or a value parameter). Otherwise it be instantiated with an arbitrary expression. Note that instantiating variables is not subject to bound renaming (this can be exploited to ensure that certain variables are identical). A pattern therefore should never contain a variable both bound and free, if it is not intended that both variables are the same in the goal.

Examples

1. "x = y and y = z" matches "phi and psi" by the substitution \{phi ← x = y, psi ← y = z\}
2. "x = z" matches "epsilon" by the substitution \{epsilon ← x = z\}
3. "< skip > true" matches "< alpha > true" by the substitution \{alpha ← skip\}
4. "add(a, b) = add(a, b)" matches "x = x" by the substitution \{x ← add(a,b)\}, if the type of x is the same as the one of add(a,b).
APPENDIX E. THE PATTERNS HEURISTIC

5. %"x := add(y,z)" matches %"<xvar := x" by the substitution {xvar ← x, x ← add(y,z)}.

6. %"all x. x = x" matches %"all \$vl. \$phi"
   by the substitution {\$vl ← (x), \$phi ← x = x}

7. %"all x. true" matches %"all xvar,\$vl. true"
   by the substitution {\$vl ← (x), xvar ← x}, if the type of xvar is the same as the type of x,
   but not %"all xvar,\$vl, yvar. true"

8. %"y = y and all x. true" does not match %"x = x and all x. true" since the same
   variable must instantiate all occurrences of x in the pattern (the pattern used interally is
   %"x = x and all x. true")

9. %"all x. x = z" does not match %"\$epsilon", since all x. x = z is not a boolean expression

10. %"<f#(x; y)> true" matches %"<f#(add(x,y); z)> true". If you want to match calls
    only, where the value parameter is a variable, a nice trick is to add %"all x. true" to the
    forbidden formulas. This will make sure that x can match variables in the sequent only (the
    forbidden formula itself will not appear in any meaningful sequent).
Bibliography


