ISAC — Interface for Developers of Math Knowledge and Tools for Experiments in Symbolic Computation

The ISAC-Team
isac-users@ist.tugraz.at

April 3, 2004
Contents

1 Introduction 3
  1.1 The scope of this document . . . . . . . . . . . . . . . . . . 3
  1.2 Related documents . . . . . . . . . . . . . . . . . . . . . . 3
  1.3 Getting started . . . . . . . . . . . . . . . . . . . . . . . . 4

I Experimental approach 5

2 Basics, terms and parsing 6
  2.1 The definition of terms . . . . . . . . . . . . . . . . . . . . 6
  2.2 Theories and parsing . . . . . . . . . . . . . . . . . . . . . 6
  2.3 Displaying terms . . . . . . . . . . . . . . . . . . . . . . . . 7
  2.4 Converting terms . . . . . . . . . . . . . . . . . . . . . . . . 9
  2.5 Theorems . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

3 Rewriting 11
  3.1 The arguments for rewriting . . . . . . . . . . . . . . . . . . 11
  3.2 The functions for rewriting . . . . . . . . . . . . . . . . . . 12
  3.3 Variants of rewriting . . . . . . . . . . . . . . . . . . . . . . . 15
  3.4 Rule sets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
  3.5 Calculate numeric constants . . . . . . . . . . . . . . . . . . 17

4 Term orders 19
  4.1 Examples for term orders . . . . . . . . . . . . . . . . . . . . 19
  4.2 Ordered rewriting . . . . . . . . . . . . . . . . . . . . . . . . 20

5 The hierarchy of problem types 23
  5.1 The standard-function for 'matching' . . . . . . . . . . . . . 23
  5.2 Accessing the hierarchy . . . . . . . . . . . . . . . . . . . . . 24
  5.3 Internals of the datastructure . . . . . . . . . . . . . . . . . . 26
  5.4 Match a formalization with a problem type . . . . . . . . . . 26
  5.5 Refine a problem specification . . . . . . . . . . . . . . . . . 27
## 6 Methods

6.1 The scripts’ syntax  
6.2 Control the flow of evaluation  

## 7 Do a calculational proof

7.1 Tactics for doing steps in calculations  

## 8 TSAC’s tactics

8.1 The functionality of the math engine  
8.2 Initialize the calculation  
8.3 The phase of modeling  
8.4 The phase of specification  
8.5 The phase of solving  
8.6 The final phase: check the post-condition  

## II Systematic description

## 9 The structure of the knowledge base

9.1 Tactics and data  
9.2 TSAC’s theories  
9.3 Data in *.thy- and *.ML-files  
9.4 Formal description of the problem-hierarchy  
9.5 Script tactics  

## III Authoring on the knowledge

9.6 Add a theorem  
9.7 Define and add a problem  
9.8 Define and add a predicate  
9.9 Define and add a method  
9.10  
9.11  
9.12  
9.13  


List of Tables

9.1 Atomic items of the KB . . . . . . . . . . . . . . . . . . . . . . 45
9.2 Which tactic uses which KB’s item ? . . . . . . . . . . . . . . . 46
9.3 Theories in $\mathcal{ZS4C}$-version I . . . . . . . . . . . . . . . . 47
9.4 Data in *.thy- and *.ML-files . . . . . . . . . . . . . . . . . . . . 48
Chapter 1

Introduction

1.1 The scope of this document

As a manual: This document describes the interface to IsaC’s kernel (KE), the interface to the mathematics engine (ME) included in the KE, and to the various tools like rewriting, matching etc.

IsaC’s KE is written in SML, the language developed in conjunction with predecessors of the theorem prover Isabelle. Thus, in this document we use the plain ASCII representation of SML code. The reader may note that the IsaC-user is presented a completely different view on a graphical user interface.

The document is selfcontained; basic knowledge about SML (as an introduction [?] is recommended), terms and rewriting is assumed.

Notation: SML code, directories, file names are written in ’tt’; in particular ML> is the KE prompt.

Give it a try! Another aim of this text is to give the reader hints for experiments with the tools introduced.

1.2 Related documents

Isabelle reference manual [?], also contained in the Isabelle distribution under ~/doc/.

The actual locations of files is being recorded in /software/services/isac/README and rigorously updated. In this

\footnote{The KEs current version is isac.020120-math/ which is based on the version Isabelle99 at http://isabelle.in.tum.de. The current locations at IST are
[isabelle] /software/sol26/Isabelle99/
isac-src /software/services/isac/src/ke/
isac-bin /software/services/isac/bin/ke/}
document we refer to the following directories

- Isabelle sources  
  [isabelle]/
- KE sources  
  [isac-src]/...version.../
- KE binary  
  [isac-bin]/...version.../

where ...version... stands for a directory-name containing information on the version.

1.3 Getting started

Change to the directory [isac-bin] where ISAC’s binary is located and type to the unix-prompt ‘>’ (ask your system administrator where the directory [isac-bin] is on your system):

```
> [isac-bin]/sml @SMLload=isac.020120-math
val it = false : bool
ML>
```

yielding the message val it = false : bool followed by the prompt of the KE. Having been successful so far, just type in the input presented below – all of it belongs to one session!
Part I

Experimental approach
Chapter 2

Basics, terms and parsing

Isabelle implements terms of the *simply typed lambda calculus* \cite{isabelle} defined in `~/src/Pure.ML`.

### 2.1 The definition of terms

There are two kinds of terms in Isabelle, 'raw terms' and 'certified terms'. \texttt{ISAC} works on raw terms, which are efficient but hard to comprehend.

```ml
datatype term =
  Const of string * typ
| Free of string * typ
| Var of indexname * typ
| Bound of int
| Abs of string * typ * term
| op $ of term * term;
```

```ml
datatype typ = Type of string * typ list
  | TFree of string * sort
  | TVar of indexname * sort;
```

where the definitions of sort and indexname is not relevant in this context. The type is being inferred during parsing. Parsing creates the other kind of terms, \texttt{cterm}. These \texttt{cterm}s are encapsulated records, which cannot be composed without the respective Isabelle functions (checking for type correctness), but which then are conveniently displayed as strings (using SML compiler internals – see below).

### 2.2 Theories and parsing

Parsing uses information contained in Isabelle’s theories `~/src/HOL`. The currently active theory is held in a global variable \texttt{thy}; theories can be accessed individually:

```ml
ML> thy;
```
val it = 
{ProtoPure, CPure, HOL, Ord, Set, subset, equalities, mono, Vimage, Fun, 
Prod, Lfp, Relation, Trancl, WF, NatDef, Gfp, Sum, Inductive, Nat, Arith, 
Divides, Power, Finite, Equip, IntDef, Int, Univ, Datatype, Numeral, Bin, 
IntArith, WF_Rel, Recdef, IntDiv, NatBin, List, Option, Map, Record, 
RelPow, Sexp, String, Calculation, SVC_Oracle, Main, Zorn, Filter, PNat, 
PReal, RealDef, RealOrd, RealInt, RealBin, HyperDef, Descript, ListG, 
Tools, Script, Typefix, Atools, RatArith, SqRoot, Differentiate, DiffApp, 
InsSort, Isac} : theory

ML> HOL.thy;
val it = {ProtoPure, CPure, HOL} : theory
ML> parse;
val it = fn : theory -> string -> cterm option
ML> parse thy "a + b * #3";
val it = Some "a + b * #3" : cterm option
ML> 
ML> val t = (term_of o the) it;
val t = Const (#,#) $ Free (#,#) $ (Const # $ Free # $ Free (#,#)) : term

where term_of and the are explained below. The syntax of the list of 
characters can be read out of Isabellas theories [?] [isabelle]/src/HOL/ and 
from theories developed with in ISAC at [isac-src]/knowledge/. Note 
that the syntax of the terms is different from those displayed at ISAC’s 
frontend after conversion to MathML.

2.3 Displaying terms

The print depth on the SML top-level can be set in order to produce output 
in the amount of detail desired:

ML> Compiler.Control.Print.printDepth;
val it = ref 4 : int ref
ML>
ML> Compiler.Control.Print.printDepth:= 2;
val it = () : unit
ML> t;
val it = # $ # $ (# $ #) : term
ML>
ML> Compiler.Control.Print.printDepth:= 6;
val it = () : unit
ML> t;
val it = 
Const ("op +",[RealDef.real, RealDef.real] => RealDef.real") $ 
Free ("#",RealDef.real") $ 
(Const ("op *",[RealDef.real, RealDef.real] => RealDef.real") $ 
Free ("#",RealDef.real") $ Free ("#",RealDef.real") : term

A closer look to the latter output shows that typ is output as a string 
like cterm. Other useful settings for the output are:

1Or you may use your internetbrowser to look at the files in [isabelle]/browser_info.
Anyway, the SML output of terms is not very readable; there are functions in the KE to display them:

ML> Compiler.Control.Print.printLength;
val it = ref 8 : int ref
ML> Compiler.Control.Print.stringDepth;
val it = ref 250 : int ref

where again the types are rendered as strings, but more elegantly by use of the information contained in thy..

Give it a try! The mathematics knowledge grows as it is defined in Isabelle theory by theory. Have a look by your internet browser to the hierarchy of those theories at [isabelle]/src/HOL/HOL.thy and its children available on your system. Or you may use your internet browser to look at the files in [isabelle]/browser_info.
ML> (*-5-*) parse Differentiate.thy "d_d x (a + x)"
val it = Some "d_d x (a + x)" : cterm option
ML>
ML> (*-6-*) parse Differentiate.thy "#2^^^#3"
val it = Some "#2 ^^^ #3" : cterm option

Don’t trust the string representation: if we convert (*-4-*) and (*-6-*) to terms ...

ML> (*-4-*) val thy = RatArith.thy;
ML> ((atomty thy) o term_of o the o (parse thy)) "d_d x (a + x)"
*** ------------
*** Free ( d_d, [real, real] => real)
*** . Free ( x, real)
*** . Const ( op +, [real, real] => real)
*** . . Free ( a, real)
*** . . Free ( x, real)
val it = () : unit
ML>
ML> (*-6-*) val thy = Differentiate.thy;
ML> ((atomty thy) o term_of o the o (parse thy)) "d_d x (a + x)"
*** ------------
*** Const ( Differentiate.d_d, [real, real] => real)
*** . Free ( x, real)
*** . Const ( op +, [real, real] => real)
*** . . Free ( a, real)
*** . . Free ( x, real)
val it = () : unit

... we see: in (*-4-*) we have an arbitrary function Free ( d_d, _) and in (*-6-*) we have the special function constant Const ( Differentiate.d_d, _) for differentiation, which is defined in Differentiate.thy and presumably is meant.

2.4 Converting terms

The conversion from cterm to term has been shown above:

ML> term_of;
val it = fn : cterm -> term
ML>
ML> the;
val it = fn : 'a option -> 'a
ML>
ML> val t = (term_of o the o (parse thy)) "a + b * #3";
val t = Const (#,#) $ Free (#,#) $ (Const # $ Free # $ Free (#,#)) : term

where the unwraps the term option — an auxiliary function from Larry Paulsons basic library at [isabelle]/src/Pure/library.ML, which is really worthwhile to study for any SML programmer.

The other conversions are the following, some of which use the signature sign of a theory:
2.5 Theorems

Theorems are a type, thm, even more protected than cterms: they are defined as axioms or proven in Isabelle. These definitions and proofs are contained in theories in the directory [isac-src]/knowledge/, e.g. the theorem diff_sum in the theory [isac-src]/knowledge/Differentiate.thy. Additionally, each theorem has to be recorded for ISAC in the respective *.ML, e.g. diff_sum in [isac-src]/knowledge/Differentiate.ML as follows:

```ml
ML> theorem' := overwritel (!theorem',
  ["diff_const", num_str diff_const]);
```

The additional recording of theorems and other values will disappear in later versions of ISAC.
Chapter 3

Rewriting

3.1 The arguments for rewriting

The type identifiers of the arguments and values of the rewrite-functions in 3.2 differ only in an apostroph: the apostrophed types are re-named strings in order to maintain readability.

\[ ML> \text{HOL.thy}; \]
\[ \text{val it = \{ProtoPure, CPure, HOL\} : theory} \]
\[ ML> \text{"HOL.thy" : theory'}; \]
\[ \text{val it = "HOL.thy" : theory'} \]
\[ ML> \text{sqrt_right}; \]
\[ \text{val it = fn : rew_ord (* term * term -> bool *)} \]
\[ ML> \text{"sqrt_right" : rew_ord'}; \]
\[ \text{val it = "sqrt_right" : rew_ord'} \]
\[ ML> \text{eval_rls}; \]
\[ \text{val it =} \]
\[ \text{Rls} \]
\[ \text{(preconds=[],rew_ord=\"sqrt_right\",fn),} \]
\[ \text{rules=[Thm \#,Thm \#,Thm \#,Thm \#,Thm \#,Thm \#,Thm \#,Thm \#,Thm \#,Thm \#,Thm \#,Calc \#,Calc \#,...]},} \]
\[ \text{scr=Script \{Free \#\}}: \text{rls} \]
\[ ML> \text{"eval_rls" : rls'}; \]
\[ \text{val it = "eval_rls" : rls'} \]
\[ ML> \text{diff_sum}; \]
\[ ML> \text{("diff_sum","") : thm'}; \]
\[ \text{val it = (\"diff_sum","\") : thm'} \]

where a thm’ is a pair, eventually with the string-representation of the respective theorem.
3.2 The functions for rewriting

Rewriting comes along with two equivalent functions, where the first is being actually used within the KE, and the second one is useful for tests:

```ml
ML> rewrite_; 
val it = fn 
  : theory 
    -> rew_ord 
      -> rls -> bool -> thm -> term -> (term * term list) option
ML>
ML> rewrite; 
val it = fn 
  : theory'
    -> rew_ord'
      -> rls' -> bool -> thm' -> cterm' -> (cterm' * cterm' list) option
```

The arguments are the following:

- **theory** the Isabelle theory containing the definitions necessary for parsing the term
- **rew_ord** the rewrite order [?] for ordered rewriting – see the section 4 below. For no ordered rewriting take `tless_true`, a dummy order yielding true for all arguments
- **rls** the rule set for evaluating the condition within `thm` in case `thm` is a conditional rule
- **bool** a flag which triggers the evaluation of the eventual condition in `thm`: if `false` then evaluate the condition and according to the result of the evaluation apply `thm` or not (conditional rewriting [?]), if `true` then don’t evaluate the condition, but put it into the set of assumptions
- **thm** the theorem used to try to rewrite `term`
- **term** the term eventually rewritten by `thm`

The respective values of `rewrite_` and `rewrite` are an option of a pair, i.e. `Some(_,_)` in case the `term` can be rewritten by `thm` w.r.t. `rew_ord` and/or `rls`, or `None` if no rewrite is found:

- **term** the term rewritten
- **term list** the assumptions eventually generated if the `bool` flag is set to true and `thm` is applicable.

Give it a try! ...rewriting is fun! many examples can be found in `[isac-src]/tests/...`. In `[isac-src]/tests/differentiate.sml` the following can be found:

```ml
ML> val thy' = "Differentiate.thy";
```
val thy' = "Differentiate.thy" : string
ML> val ct = "d_d x (x ^^ ^ ^ #2 + #3 * x + #4)"
val ct = "d_d x (x ^^ ^ ^ #2 + #3 * x + #4)" : cterm'
ML>
ML> val thm = ("diff_sum","");
val thm = ("diff_sum","") : thm'
ML> val Some (ct,_) = rewrite_inst thy' "tless_true" "eval_rls" true
[ ("bdv","x::real") ] thm ct;
val ct = "d_d x (x ^^ ^ ^ #2 + #3 * x) + d_d x #4" : cterm'
ML>
ML> val Some (ct,_) = rewrite_inst thy' "tless_true" "eval_rls" true
[ ("bdv","x::real") ] thm ct;
val ct = "d_d x (x ^^ ^ ^ #2) + d_d x (#3 * x) + d_d x #4" : cterm'
ML>
ML> val thm = ("diff_prod_const","");
val thm = ("diff_prod_const","") : thm'
ML> val Some (ct,_) = rewrite_inst thy' "tless_true" "eval_rls" true
[ ("bdv","x::real") ] thm ct;
val ct = "d_d x (x ^^ ^ ^ #2) + #3 * d_d x x + d_d x #4" : cterm'
ML>
ML> val thy' = "Isac.thy";
val thy' = "Isac.thy" : string
ML> val ct' = "#3 * a + #2 * (a + #1)"
val ct' = "#3 * a + #2 * (a + #1)" : cterm'
ML>
ML> val thm' = ("radd_mult_distrib2","?k * (?m + ?n) = ?k * ?m + ?k * ?n")
val thm' = ("radd_mult_distrib2","?k * (?m + ?n) = ?k * ?m + ?k * ?n")
: thm'
ML> (*1*) val Some (ct',_) = rewrite thy' "tless_true" "eval_rls" true thm' ct';
val ct' = "#3 * a + (#2 * a + #2 * #1)" : cterm'
ML>
ML> val thm' = ("radd_assoc_RS_sym","?m1 + (?n1 + ?k1) = ?m1 + ?n1 + ?k1")
val thm' = ("radd_assoc_RS_sym","?m1 + (?n1 + ?k1) = ?m1 + ?n1 + ?k1")
: thm'
ML> (*2*) val Some (ct',_) = rewrite thy' "tless_true" "eval_rls" true thm' ct';
val ct' = "#3 * a + #2 * a + #2 * #1" : cterm'
ML>
ML> val thm' = ("rcollect_right",
[| ?l is_const; ?m is_const |] ==> ?l * ?n + ?m * ?n = (?l + ?m) * ?n")
val thm' = ("rcollect_right",
[| ?l is_const; ?m is_const |] ==> ?l * ?n + ?m * ?n = (?l + ?m) * ?n")

You can look up the theorems in [isac-src]/knowledge/Differentiate.thy and try to apply them until you get the result you would expect if calculating by hand. 

Give a try! Conditional rewriting is a more powerful technique than ordinary rewriting, and is closer to the power of programming languages (see the subsequent 'try it out'!). The following example expands a term to polynomial form:

ML> val thy' = "Isac.thy";
val thy' = "Isac.thy" : string
ML> val ct' = "#3 * a + #2 * (a + #1)"
val ct' = "#3 * a + #2 * (a + #1)" : cterm'
ML>
ML> val thm' = ("radd_mult_distrib2","?k * (?m + ?n) = ?k * ?m + ?k * ?n")
val thm' = ("radd_mult_distrib2","?k * (?m + ?n) = ?k * ?m + ?k * ?n")
: thm'
ML> (*1*) val Some (ct',_) = rewrite thy' "tless_true" "eval_rls" true thm' ct';
val ct' = "#3 * a + (#2 * a + #2 * #1)" : cterm'
ML>
ML> val thm' = ("radd_assoc_RS_sym","?m1 + (?n1 + ?k1) = ?m1 + ?n1 + ?k1")
val thm' = ("radd_assoc_RS_sym","?m1 + (?n1 + ?k1) = ?m1 + ?n1 + ?k1")
: thm'
ML> (*2*) val Some (ct',_) = rewrite thy' "tless_true" "eval_rls" true thm' ct';
val ct' = "#3 * a + #2 * a + #2 * #1" : cterm'
ML>
ML> val thm' = ("rcollect_right",
[| ?l is_const; ?m is_const |] ==> ?l * ?n + ?m * ?n = (?l + ?m) * ?n")
val thm' = ("rcollect_right",
[| ?l is_const; ?m is_const |] ==> ?l * ?n + ?m * ?n = (?l + ?m) * ?n")

1Hint: At the end you will need val (ct,_) = the (rewrite_set thy' "eval_rls"
false "SqRoot_simplify" ct);
"[! ?l is Const; ?m is Const |] ==> ?l * ?n + ?m * ?n = (?l + ?m) * ?n"

ML> (*3*) val Some (ct',_) = rewrite thy' "tless_true" "eval_rls" true thm' ct';
val ct' = "(#3 + #2) * a + #2 * #1" : cterm'
ML>
ML> (*4*) val Some (ct',_) = calculate' thy' "plus" ct';
val ct' = "#5 * a + #2 * #1" : cterm'
ML>
ML> (*5*) val Some (ct',_) = calculate' thy' "times" ct';
val ct' = "#5 * a + #2" : cterm'

Note, that the two rules, radd_mult_distrib2 in (*1*) and rcollect_right in (*3*) would neutralize each other (i.e. a rule set would not terminate), if there would not be the condition is_const.

**Give it a try!** Functional programming can, within a certain range, modeled by rewriting. In [isac-src]/.../tests/InsSort.thy the following rules can be found, which are able to sort a list ('insertion sort'):

<table>
<thead>
<tr>
<th>Rule</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort_def</td>
<td>&quot;sort ls = foldr ins ls []&quot;</td>
</tr>
<tr>
<td>ins_base</td>
<td>&quot;ins [] a = [a]&quot;</td>
</tr>
<tr>
<td>ins_rec</td>
<td>&quot;ins (x#xs) a = (if x &lt; a then x#(ins xs a) else a#(x#xs))&quot;</td>
</tr>
<tr>
<td>foldr_base</td>
<td>&quot;foldr f [] a = a&quot;</td>
</tr>
<tr>
<td>foldr_rec</td>
<td>&quot;foldr f (x#xs) a = foldr f xs (f a x)&quot;</td>
</tr>
<tr>
<td>if_True</td>
<td>&quot;(if True then ?x else ?y) = ?x&quot;</td>
</tr>
<tr>
<td>if_False</td>
<td>&quot;(if False then ?x else ?y) = ?y&quot;</td>
</tr>
</tbody>
</table>

where # is the list-constructor, foldr is the well-known standard function of functional programming, and if_True, if_False are auxiliary rules.

Then the sort may be done by the following rewrites:

ML> val thy' = "InsSort.thy";
val thy' = "InsSort.thy" : theory'
ML> val ct = "sort [#1,#3,#2] []" : cterm'
ML>
ML> val thm = ("sort_def","");
ML> val (ct,_) = the (rewrite thy' "tless_true" "eval_rls" false thm ct);
val ct = "foldr ins [#1, #3, #2] []" : cterm'
ML>
ML> val thm = ("foldr_rec","");
ML> val (ct,_) = the (rewrite thy' "tless_true" "eval_rls" false thm ct);
val ct = "foldr ins [#3, #2] (ins [] #1)" : cterm'
ML>
ML> val thm = ("ins_base","");
ML> val (ct,_) = the (rewrite thy' "tless_true" "eval_rls" false thm ct);
val ct = "foldr ins [#3, #2] #1" : cterm'
ML>
ML> val thm = ("foldr_rec","");
ML> val (ct,_) = the (rewrite thy' "tless_true" "eval_rls" false thm ct);
val ct = "foldr ins [#2] (ins #1 #3)": cterm'
3.3 Variants of rewriting

Some of the above examples already used variants of `rewrite` all of which have the same value, and very similar arguments:

```plaintext
ML> val thm = ("ins_rec","";
ML> val (ct,_) = the (rewrite thy' "tless_true" "eval_rls" false thm ct);
val ct = "foldr ins [#2] (if #1 < #3 then #1 # ins [] #3 else [#3, #1])" : cterm'
ML> val (ct,_) = the (calculate' thy' "le" ct);
val ct = "foldr ins [#2] (if True then #1 # ins [] #3 else [#3, #1])" : cterm'
ML> val thm = ("if_True","(if True then ?x else ?y) = ?x";
ML> val (ct,_) = the (rewrite thy' "tless_true" "eval_rls" false thm ct);
val ct = "foldr ins [#2] (#1 # ins [] #3)" : cterm'
ML> ...
val ct = "sort [#1,#3,#2]" : cterm'
```
val it = fn
  : theory' -> rls' -> bool
    -> (cterm' * cterm') list
      -> rls' -> cterm' -> (cterm' * cterm' list) option

The variant \texttt{rewrite\_inst} substitutes \texttt{(term * term) list} in \texttt{thm} before rewriting,
the variant \texttt{rewrite\_set} rewrites with a whole rule set \texttt{rls} (instead with a \texttt{thm} only),
the variant \texttt{rewrite\_set\_inst} is a combination of the latter two variants. In order to watch how a term is rewritten theorem by theorem, there is a switch \texttt{trace\_rewrite}:

ML> toggle; val it = fn : bool ref -> bool
ML> toggle trace_rewrite; val it = true : bool
ML> toggle trace_rewrite; val it = false : bool

3.4 Rule sets

Some of the variants of \texttt{rewrite} above do not only apply one theorem, but a whole set of theorems, called a ‘rule set’. Such a rule set is applied as long one of its elements can be used for a rewrite (which can go forever, i.e. the rule set eventually does not ‘terminate’).

A simple example of a rule set is \texttt{rearrange\_assoc} which is defined in \texttt{knowledge/RatArith.ML} as:

val rearrange_assoc =  
  Rls{preconds = [], rew_ord = ("tless\_true",tless\_true),
    rules = [Thm ("radd\_assoc\_RS\_sym",num\_str (radd\_assoc RS sym)),
             Thm ("rmult\_assoc\_RS\_sym",num\_str (rmult\_assoc RS sym))],
    scr = Script ((term\_of o the o (parse thy))
                    "empty\_script")};

where

\texttt{preconds} are conditions which must be true in order to make the rule set applicable (the empty list evaluates to \texttt{true})

\texttt{rew\_ord} concerns term orders introduced below in 4

\texttt{rules} are the theorems to be applied – in principle applied in arbitrary order, because all these rule sets should be ‘complete’ [?] (and actually the theorems are applied in the sequence they appear in this list).
function `num_str` must be applied to theorems containing numeral constants (and thus is obsolete in this example). `RS` is an infix function applying the theorem `sym` to `radd_assoc` before storage (the effect see below).

`scre` is the script applying the ruleset; it will disappear in later versions of `TSAC`.

These variables evaluate to

```ml
ML> sym;
val it = "?s = ?t ==> ?t = ?s" : thm
ML> rearrange_assoc;
val it =
Rls
{preconds=[],rew_ord="tless_true",fn),
  rules=[Thm ("radd_assoc_RS_sym","?m1 + (?n1 + ?k1) = ?m1 + ?n1 + ?k1"),
         Thm ("rmult_assoc_RS_sym","?m1 * (?n1 * ?k1) = ?m1 * ?n1 * ?k1")],
  scr=Script (Free ("empty_script","RealDef.real"))} : rls
```

**Give it a try!** The above rule set makes an arbitrary number of parentheses disappear which are not necessary due to associativity of + and *

```ml
ML> val ct = (string_of_cterm o the o (parse RatArith.thy))
  ("a + (b * (c * d) + e) + f");
val ct = "a + ((b * (c * d) + e) + f)" : cterm'
ML> rewrite_set "RatArith.thy" "eval_rls" false "rearrange_assoc" ct;
val it = Some ("a + b * c * d + e + f",[]) : (string * string list) option
```

For achieving this result the rule set has to be surprisingly busy:

```ml
ML> toggle trace_rewrite;
val it = true : bool
ML> rewrite_set "RatArith.thy" "eval_rls" false "rearrange_assoc" ct;
## trying thm 'radd_assoc_RS_sym'
## rewrite_set: a + b * (c * d) + e
## trying thm 'radd_assoc_RS_sym'
## rewrite_set: a + b * c * d + e
## trying thm 'rmult_assoc_RS_sym'
## rewrite_set: a + b * c * d + e
## trying thm 'radd_assoc_RS_sym'
## rewrite_set: a + b * c * d + e
## trying thm 'rmult_assoc_RS_sym'
## rewrite_set: a + b * c * d + e
val it = Some ("a + b * c * d + e",[]) : (string * string list) option
```

### 3.5 Calculate numeric constants

As soon as numeric constants are in adjacent subterms (see the example on p.13), they can be calculated by the function

```ml
ML> calculate;
val it = fn : theory' -> string -> cterm' -> (cterm' * thm') option
ML> calculate_;
val it = fn : theory -> string -> term -> (term * (string * thm)) option
```
where the string in the arguments defines the algebraic operation to be calculated. The function returns the result of the calculation, and as second element in the pair the theorem applied. The following algebraic operations are available:

\[
\text{ML> calc_list;}
\]
\[
\text{val it = ref}
\]
\[
[(\"plus\", (\"op +\", fn)),
 (\"times\", (\"op *\", fn)),
 (\"cancel_\", (\"cancel\", fn)),
 (\"power\", (\"pow\", fn)),
 (\"sqrt\", (\"sqrt\", fn)),
 (\"Var\", (\"Var\", fn)),
 (\"Length\", (\"Length\", fn)),
 (\"Nth\", (\"Nth\", fn)),
 (\"power\", (\"pow\", fn)),
 (\"le\", (\"op <\", fn)),
 (\"leq\", (\"op <=\", fn)),
 (\"is_const\", (\"is\'_const\", fn)),
 (\"is_root_free\", (\"is\'_root\'_free\", fn)),
 (\"contains_root\", (\"contains\'_root\", fn)),
 (\"ident\", (\"ident\", fn))]
\]
\[
: (\text{string} \times (\text{string} \times (\text{string} \to \text{term} \to \text{theory} \to (\text{string} \times \text{term}) \text{ option})) \text{ list ref}
\]

These operations can be used in the following way.

\[
\text{ML> calculate' \"Isac.thy\" \"plus\" \"#1 + #2\";
val it = Some (\"#3\", (\"add\_#1\_#2\", \"#1 + #2 = #3\")) : (\text{string} \times \text{thm'}) \text{ option}
\]
\[
\text{ML> calculate' \"Isac.thy\" \"times\" \"#2 * #3\";
val it = Some (\"#6\", (\"mult\_#2\_#3\", \"#2 * #3 = #6\"))
: (\text{string} \times \text{thm'}) \text{ option}
\]
\[
\text{ML> calculate' \"Isac.thy\" \"power\" \"#2 ^^^ #3\";
val it = Some (\"#8\", (\"power\_#2\_#3\", \"#2 ^^^ #3 = #8\"))
: (\text{string} \times \text{thm'}) \text{ option}
\]
\[
\text{ML> calculate' \"Isac.thy\" \"cancel\_\" \"#9 // #12\";
val it = Some (\"#3 // #4\", (\"cancel\_#9\_#12\", \"#9 // #12 = #3 // #4\"))
: (\text{string} \times \text{thm'}) \text{ option}
\]
\[
\text{ML> ...}
\]
Chapter 4

Term orders

Orders on terms are indispensable for the purpose of rewriting to normal forms in associative - commutative domains [?], and for rewriting to normal forms necessary for matching models to problems, see sect.5.

4.1 Examples for term orders

It is not trivial to construct a relation $<$ on terms such that it is really an order, i.e. a transitive and antisymmetric relation. These orders are 'recursive path orders' [?]. Some orders implemented in the knowledgebase at [isac-src]/knowledge/..., e.g.

```ml
ML> sqrt_right;
val it = fn : bool -> theory -> term * term -> bool
ML> tless_true;
val it = fn : 'a -> bool

where the bool argument is a switch for tracing the checks on the respective subterms (and theory is necessary for displaying the (sub-)terms as strings in the case of 'true'). The order tless_true is a dummy always yielding true, and sqrt_right prefers a square root shifted to the right within a term:

```ml
ML> val t1 = (term_of o the o (parse thy)) "(sqrt a) + b";
val t1 = Const # $ (# $ #) $ Free (#,#) : term
ML> val t2 = (term_of o the o (parse thy)) "b + (sqrt a)";
val t2 = Const # $ Free # $ (Const # $ Free #) : term
ML> sqrt_right false SqRoot.thy (t1, t2);
val it = false : bool
ML> sqrt_right false SqRoot.thy (t2, t1);
val it = true : bool
```

The many checks performed recursively through all subterms can be traced throughout the algorithm in [isac-src]/knowledge/SqRoot.ML by setting the flag to true:

20
4.2 Ordered rewriting

Rewriting faces problems in just the most elementary domains, which are all associative and commutative w.r.t. + and * — the law of commutativity applied within a rule set causes this set not to terminate! One method to cope with this difficulty is ordered rewriting, where a rewrite is only done if the resulting term is smaller w.r.t. a term order (with some additional properties called 'rewrite orders' [?]).

Such a rule set ac_plus_times, called an AC-rewrite system, can be found in [isac-src]/knowledge/RathArith.ML:

```ml
val ac_plus_times =
  Rls{preconds = [], rew_ord = ("term_order",term_order),
```
rules = [Thm ("radd_commute",radd_commute),
Thm ("radd_left_commute",radd_left_commute),
Thm ("radd_assoc",radd_assoc),
Thm ("rmult_commute",rmult_commute),
Thm ("rmult_left_commute",rmult_left_commute),
Thm ("rmult_assoc",rmult_assoc)],
scr = Script ((term_of o the o (parse thy)) "empty_script")
];

val ac_plus_times =
Rls {preconds=[],rew_ord="term_order",fn,
rules=[Thm ("radd_commute","?m + ?n = ?n + ?m"),
Thm ("radd_left_commute","?x + (?y + ?z) = ?y + (?x + ?z)"),
Thm ("radd_assoc","?m + ?n + ?k = ?m + (?n + ?k)"),
Thm ("rmult_commute","?m * ?n = ?n * ?m"),
Thm ("rmult_left_commute","?x * (?y * ?z) = ?y * (?x * ?z)"),
Thm ("rmult_assoc","?m * ?n * ?k = ?m * (?n * ?k)")],
scr=Script (Free ("empty_script", RealDef.real"))))
};

Note that the theorems radd_left_commute and rmult_left_commute are really necessary in order to make the rule set 'confluent'!

**Give it a try!** Ordered rewriting is one technique to produce polynomial normal from from arbitrary integer terms:

ML> val ct' = "#3 * a + b + #2 * a";
 ML> val ct' = "#3 * a + b + #2 * a" : cterm

ML> (*-1-*) radd_commute; val thm' = ("radd_commute","") : thm';
 val it = "?m + ?n = ?n + ?m" : thm
 val thm' = ("radd_commute","") : thm'
 ML> val Some (ct',_) = rewrite thy' "tless_true" "eval_rls" true thm' ct';
 val ct' = "#2 * a + (#3 * a + b)" : cterm

ML> (*-2-*) rdistr_right_assoc_p; val thm' = ("rdistr_right_assoc_p","") : thm';
 val it = "?l * ?n + (?m * ?n + ?k) = (?l + ?m) * ?n + ?k" : thm
 val thm' = ("rdistr_right_assoc_p","") : thm'
 ML> val Some (ct',_) = rewrite thy' "tless_true" "eval_rls" true thm' ct';
 val ct' = "(#2 + #3) * a + b" : cterm

ML> (*-3-*)
 ML> val Some (ct',_) = calculate thy' "plus" ct';
val ct' = "#5 * a + b" : cterm

This looks nice, but if radd_commute is applied automatically in (*-1-*) without checking the resulting term to be 'smaller' w.r.t. a term order, then rewriting goes on forever (i.e. it does not 'terminate')...
Ordered rewriting with the above AC-rewrite system \texttt{ac\_plus\_times} performs a kind of bubble sort which can be traced:

\begin{verbatim}
ML> toggle trace_rewrite;
val it = true : bool
ML>
ML> rewrite_set "RatArith.thy" "eval_rls" false "ac\_plus\_times" ct;

### trying thm 'radd_commute'
### not: "a + (b * (c + d) + e)" > "b * (c + d) + e + a"
### rewrite_set_: a + (e + b * (c + d))
### trying thm 'radd_commute'
### not: "a + (e + b * (c + d))" > "e + b * (c + d) + a"
### not: "e + b * (c + d)" > "b * (c + d) + e"
### trying thm 'radd_left_commute'
### not: "a + (e + b * (c + d))" > "e + (a + b * (c + d))"
### trying thm 'radd_assoc'
### trying thm 'rmult_commute'
### not: "b * (c + d)" > "c * d * b"
### not: "c * d" > "d * c"
### trying thm 'rmult_left_commute'
### not: "b * (c + d)" > "c * (b + d)"
### trying thm 'rmult_assoc'
### trying thm 'radd_commute'
### not: "a + (e + b * (c + d))" > "e + b * (c + d) + a"
### not: "e + b * (c + d)" > "b * (c + d) + e"
### trying thm 'radd_left_commute'
### not: "a + (e + b * (c + d))" > "e + (a + b * (c + d))"
### trying thm 'radd_assoc'
### trying thm 'rmult_commute'
### not: "b * (c + d)" > "c * d * b"
### not: "c * d" > "d * c"
### trying thm 'rmult_left_commute'
### not: "b * (c + d)" > "c * (b + d)"
### trying thm 'rmult_assoc'

val it = Some ("a + (e + b * (c + d))",\[]) : (string * string list) option
\end{verbatim}

Notice that + is left-associative where the parentheses are omitted for \((a + b) + c = a + b + c\), but not for \(a + (b + c)\). Ordered rewriting necessarily terminates with parentheses which could be omitted due to associativity.
Chapter 5

The hierarchy of problem types

5.1 The standard-function for 'matching'

Matching [?] is a technique used within rewriting, and used by ISAC also for (a generalized) 'matching' a problem with a problem type. The function which tests for matching has the following signature:

```ml
ML> matches;
val it = fn : theory -> term -> term -> bool
```

where the first of the two term arguments is the particular term to be tested, and the second one is the pattern:

```ml
ML> val t = (term_of o the o (parse thy)) "#3 * x^^^#2 = #1";
val t = Const (#,#) $ (# $ # $ (# $ #)) $ Free (#1,"RealDef.real") : term
ML> val p = (term_of o the o (parse thy)) "a * b^^^#2 = c";
val p = Const (#,#) $ (# $ # $ (# $ #)) $ Free (c,"RealDef.real") : term
ML> atomt p;
*** ------------
*** Const ( op =)
*** . Const ( op *)
*** . . Free ( a, )
*** . . Const ( RatArith.pow)
*** . . . Free ( b, )
*** . . . Free ( #2, )
*** . . Free ( c, )
val it = () : unit
ML> free2var;
val it = fn : term -> term
ML> val pat = free2var p;
val pat = Const (#,#) $ (# $ # $ (# $ #)) $ Var ((#,#),"RealDef.real") : term
ML> Sign.string_of_term (sign_of thy) pat;
val it = "$?a * ?b ^^^ #2 = ?c" : cterm'
```
Note that the pattern `pat` contains so-called scheme variables decorated with a `?` (analogous to theorems). The pattern is generated by the function `free2var`. This format of the pattern is necessary in order to obtain results like these:

```ml
ML> matches thy t pat;
val it = true : bool
ML> val t2 = (term_of o the o (parse thy)) "x^^^#2 = #1";
val t2 = Const (#,#) $ (# $ # $ Free #) $ Free (#1,"RealDef.real") : term
ML> matches thy t2 pat;
val it = false : bool
ML> val pat2 = (term_of o the o (parse thy)) "?u^^^#2 = ?v";
val pat2 = Const (#,#) $ (# $ # $ Free #) $ Var ((#,#),"RealDef.real") : term
ML> matches thy t2 pat2;
val it = true : bool
```

### 5.2 Accessing the hierarchy

The hierarchy of problem types is encapsulated; it can be accessed by the following functions. `show_ptyps` retrieves all leaves of the hierarchy (here in an early version for testing):

```ml
ML> show_ptyps;
val it = fn : unit -> unit
ML> show_ptyps();
["e_pblID", "equation", "univariate", "linear", "equation", "univariate", "plain_square"],
"equation", "univariate", "polynomial", "degree_two", "pq_formula"],
"equation", "univariate", "polynomial", "degree_two", "abc_formula"],
"equation", "univariate", "squareroot"],
"equation", "univariate", "normalize"],
"equation", "univariate", "sqroot-test"],
"function", "derivative_of"],
"function", "maximum_of", "on_interval"],
"function", "make"],
"tool", "find_values"],
"functional", "inssort"]
val it = () : unit
```
The retrieve function for individual problem types is `get_pbt`. Note that its argument, the 'problem identifier' `pblID`, has the strings listed in reverse order w.r.t. the hierarchy, i.e. from the leave to the root. This order makes the `pblID` closer to a natural description:

```ml
ML> get_pbt; val it = fn : pblID -> pbt
ML> get_pbt ["squareroot", "univariate", "equation"]; val it = 
{met=[("SqRoot.thy","square_equation")],
 ppc=[("#Given",(Const (#,#).Free (#,#)))],
 ("#Where",(Const (#,#).Free (#,#))),
 ("#Find",(Const (#,#).Free (#,#))))
thy={ProtoPure, CPure, HOL, Ord, Set, subset, equalities, mono, Vimage, Fun, Prod, Lfp, Relation, Trancl, WF, NatDef, Gfp, Sub, Inductive, Nat, Arith, Divides, Power, Finite, Equiv, IntDef, Int, Univ, Datatype, Numeral, Bin, IntArith, WF_Rel, Recdef, IntDiv, NatBin, List, Option, Map, Record, RelPow, Sexp, String, Calculation, SVC_Oracle, Main, Zorn, Filter, PNat, P Rat, PReal, RealDef, RealOrd, RealInt, RealBin, HyperDef, Descript, ListG, Tools, Script, Typefix, Atools, RatArith, SqRoot},
where_=Const ("SqRoot.contains'_root","bool => bool") $
Free ("e_","bool")): pbt
```

where the records fields hold the following data:

- **thy**: the theory necessary for parsing the formulas
- **ppc**: the items of the problem type, divided into those `Given`, the precondition `Where` and the output item(s) `Find`. The items of `Given` and `Find` are all headed by so-called descriptions, which determine the type. These descriptions are defined in `isac-src/Isa99/Descript.thy`.
- **met**: the list of methods solving this problem type.

The following function adds or replaces a problem type (after having it prepared using `prep_pbt`)

```ml
ML> store_pbt; val it = fn : pbt * pblID -> unit
ML> store_pbt (prep_pbt SqRoot.thy
{["newtype","univariate","equation"]},
{["#Given",["equality e_","solveFor v_","errorBound err_"]],
 ("#Where",["contains_root (e_::bool)"]),
 ("#Find",["solutions v_i"])
},
{["SqRoot.thy","square_equation")]});
val it = () : unit
```

When adding a new type with argument `pblID`, an immediate parent must already exist in the hierarchy (this is the one with the tail of `pblID`).

---

1 A function providing better readable output is in preparation
5.3 Internals of the datastructure

This subsection only serves for the implementation of the hierarchy browser and can be skipped by the authors of math knowledge.

A problem type is described by the following record type (in the file \texttt{[isac-src]/globals.sml}, the respective functions are in \texttt{[isac-src]/ME/ptyps.sml}), and held in a global reference variable:\

```sml

    type pbt = 
        {thy : theory, (* the nearest to the root, which allows to compile that pbt *)
         where_: term list, (* where - predicates *)
         ppc : ((string * (* fields "#Given","#Find" *)
                  (term * (* description *)
                  term)) list), (* id *)
         met : metID list}; (* methods solving the pbt *)

datatype ptyp =
    Ptyp of string * pbt list * ptyp list;

e_Ptyp = Ptyp ("empty",[],[]);

type ptyps = ptyp list;

```

val ptyps = ref (\[e_Ptyp\]:ptyps);

The predicates in \texttt{where} (i.e. the preconditions) usually are defined in the respective theory in \texttt{[isac-src]/knowledge}. Most of the predicates are not defined by rewriting, but by SML-code contained in the respective *.ML file.

Each item is headed by a so-called description which provides some guidance for interactive input. The descriptions are defined in \texttt{[isac-src]/Isa99/Descript.thy}.

5.4 Match a formalization with a problem type

A formalization is matched with a problem type which yields a problem. A formal description of this kind of matching can be found in \\
\texttt{ftp://ft.ist.tugraz.at/projects/isac/publ/calculemus01.ps.gz}. A formalization of an equation is e.g.

```sml

ML> val fmz = ["equality (#1 + #2 * x = #0)",
                "solveFor x",
                "solutions L"] : fmz;

```

Given a formalization (and a specification of the problem, i.e. a theory, a problemtype, and a method) \texttt{ISAC} can solve the respective problem automatically. The formalization must match the problem type for this purpose:

```sml

ML> match_pbl;
val it = fn : fmz -> pbt -> match'
ML>

```
ML> match_pbl fmz (get_pbt ["univariate","equation"]);
val it = Matches’
{Find=[Correct "solutions L"],
Given=[Correct "equality (#1 + #2 * x = #0)"],
Relate=[],Where=[Correct "matches (?a = ?b) (#1 + #2 * x = #0)"]},With=[]} : match’

ML> match_pbl fmz (get_pbt ["linear","univariate","equation"]);
val it = Matches’
{Find=[Correct "solutions L"],
Given=[Correct "equality (#1 + #2 * x = #0)"],
Relate=[],
Where=[Correct "matches ( x = #0) (#1 + #2 * x = #0) |
matches (?b * x = #0) (#1 + #2 * x = #0) |
matches (?a + x = #0) (#1 + #2 * x = #0) |
matches (?a + ?b * x = #0) (#1 + #2 * x = #0)"]],
With=[]} : match’

ML> match_pbl fmz (get_pbt ["squareroot","univariate","equation"]);
val it = NoMatch’
{Find=[Correct "solutions L"],
Given=[Correct "equality (#1 + #2 * x = #0)"],
Relate=[],
Where=[False "contains_root #1 + #2 * x = #0 "]},With=[]} : match’

The above formalization does not match the problem type ["squareroot","univariate","equation"] which is explained by the tags:

- **Missing**: the item is missing in the formalization as required by the problem type
- **Superfl**: the item is not required by the problem type
- **Correct**: the item is correct, or the precondition (Where) is true
- **False**: the precondition (Where) is false
- **Incompl**: the item is incomplete, or not yet input.

### 5.5 Refine a problem specification

The challenge in constructing the problem hierarchy is, to design the branches in such a way, that problem refinement can be done automatically (as it is done in algebra system e.g. by an internal hierarchy of equations).

For this purpose the hierarchy must be built using the following rules:
Let $F$ be a formalization and $P$ and $P_i$, $i = 1 \cdots n$ problem types, where the $P_i$ are specialized problem types w.r.t. $P$ (i.e. $P$ is a parent node of $P_i$), then

1. for all $F$ matching some $P_i$ must follow, that $F$ matches $P$
2. an $F$ matching $P$ should not have more than one $P_i$, $i = 1 \cdots n - 1$ with $F$ matching $P$; (if there are more than one $P_i$, the first one will be taken)

3. for all $F$ matching some $P$ must follow, that $F$ matches $P_i$

Let us give an example for the point (1.) and (2.) first:

ML> refine;
val it = fn : fmz -> pblID -> match list
ML> val fmz = ["equality (sqrt(#9+#4*x)=sqrt x + sqrt(#5+x))",
  "solveFor x","errorBound (eps=#0)",
  "solutions L"];
ML> refine fmz ["univariate","equation"];
*** pass ["equation","univariate"]
*** pass ["equation","univariate","linear"]
*** pass ["equation","univariate","plain_square"]
*** pass ["equation","univariate","polynomial"]
*** pass ["equation","univariate","squareroot"]
val it =

[Matches
  ("univariate","equation"),
  (Find=[Correct "solutions L"],
   Given=[Correct "equality (sqrt (#9 + #4 * x) = sqrt x + sqrt (#5 + x))", Correct "solveFor x",Superfl "errorBound (eps = #0)"] ,Relate=[],
   Where=[Correct
   "matches (?a = ?b) (sqrt (#9 + #4 * x) = sqrt x + sqrt (#5 + x))"],
   With=[])],
NoMatch
  ("linear","univariate","equation"),
  (Find=[Correct "solutions L"],
   Given=[Correct "equality (sqrt (#9 + #4 * x) = sqrt x + sqrt (#5 + x))", Correct "solveFor x",Superfl "errorBound (eps = #0)"] ,Relate=[],
   Where=[False "(?a + ?b * x = #0) (sqrt (#9 + #4 * x)#",
   With=[])],
NoMatch
  ("plain_square","univariate","equation"),
  (Find=[Correct "solutions L"],
   Given=[Correct "equality (sqrt (#9 + #4 * x) = sqrt x + sqrt (#5 + x))", Correct "solveFor x",Superfl "errorBound (eps = #0)"] ,Relate=[],
   Where=[False
   "matches (?a + ?b * x ^^^ #2 = #0)"],
   With=[])],
NoMatch
  ("polynomial","univariate","equation"),
  (Find=[Correct "solutions L"],
   Given=[Correct "equality (sqrt (#9 + #4 * x) = sqrt x + sqrt (#5 + x))", Correct "solveFor x",Superfl "errorBound (eps = #0)"] ,Relate=[],
   Where=[False
   "is_polynomial_in sqrt (#9 + #4 * x) = sqrt x + sqrt (#5 + x) x"],
   With=[])],
Matches
  ("squareroot","univariate","equation"),
This example shows, that in order to refine an "univariate", "equation"], the formalization must match respective respective problem type (rule (1.)) and one of the descendants which should match selectively (rule (2.)).

If no one of the descendants of ["univariate", "equation"] match, rule (3.) comes into play: The last problem type on this level (Pₙ) provides for a special ‘problem type’ ["normalize"]. This node calls a method transforming the equation to a (or another) normal form, which then may match. Look at this example:

ML> val fmz = ["equality (x+#1=#2)",
  "solveFor x","errorBound (eps=#0)",
  "solutions L"]; [...]
ML> refine fmz ["univariate","equation"];
*** pass ["equation","univariate"]
*** pass ["equation","univariate","linear"]
*** pass ["equation","univariate","plain_square"]
*** pass ["equation","univariate","polynomial"]
*** pass ["equation","univariate","squareroot"]
*** pass ["equation","univariate","normalize"]
val it =
  [Matches
    ["univariate","equation"],
    {Find=[Correct "solutions L"],
     Given=[Correct "equality (x + #1 = #2)",Correct "solveFor x",
     Superfl "errorBound (eps = #0)"],Relate=[],
     Where=[Correct "matches (?a = ?b) (x + #1 = #2)"],With=[]},
    NoMatch
    ["linear","univariate","equation"],
    [...]}
  With=[]],
NoMatch
["squareroot","univariate","equation"],
 {Find=[Correct "solutions L"],
  Given=[Correct "equality (x + #1 = #2)",Correct "solveFor x",
  Correct "errorBound (eps = #0)"],Relate=[],
  Where=[False "contains_root x + #1 = #2 "],With=[]}),
Matches
["normalize","univariate","equation"],
{Find=[Correct "solutions L"],
Given=[Correct "equality (x + #1 = #2)",Correct "solveFor x",
Superfl "errorBound (eps = #0)"],Relate=[],Where=[],With=[]})
]: match list

The problem type Pₙ, ["normalize", "univariate", "equation"], will transform the equation x + #1 = #2 to the normal form #~1 + x = #0,
which then will match ["linear","univariate","equation"].

This recursive search on the problem hierarchy can be done within a proof state. This leads to the next section.
Chapter 6

Methods

A problem type can have one or more methods solving a respective problem. A method is described by means of another new program language. The language itself looks like a simple functional language, but constructs an imperative proof-state behind the scenes (thus liberating the programmer from dealing with technical details and also prohibiting incorrect construction of the proof tree). The interpreter of 'scripts' written in this language evaluates the script-expressions, and also delivers certain parts of the script itself for discussion with the user.

6.1 The scripts’ syntax

The syntax of scripts follows the definition given in Backus-normal-form:

\[
\text{script} ::= \text{Script id arg}^{*} = \text{body} \\
\text{arg} ::= \text{id} \mid ( ( \text{id} :: \text{type} ) ) \\
\text{body} ::= \text{expr} \\
\text{expr} ::= \text{let id = expr ( ; id = expr)* in expr} \\
\quad \mid \text{if prop then expr else expr} \\
\quad \mid \text{listexpr} \\
\quad \mid \text{id} \\
\quad \mid \text{seqex id} \\
\text{seqex} ::= \text{While prop Do seqex} \\
\quad \mid \text{Repeat seqex} \\
\quad \mid \text{Try seqex} \\
\quad \mid \text{seqex Or seqex} \\
\quad \mid \text{seqex @@ seqex} \\
\quad \mid \text{tac ( id | listexpr )*} \\
\text{type} ::= \text{id} \\
\text{tac} ::= \text{id}
\]

where \text{id} is an identifier with the usual syntax, \text{prop} is a proposition constructed by Isabelle’s logical operators (see [?] [isabelle]/src/HOL/HOL.thy),
listexpr (called list-expression) is constructed by Isabelle’s list functions like \texttt{hd}, \texttt{tl}, \texttt{nth} described in \texttt{[isabelle]/src/HOL/List.thy}, and \texttt{type} are (virtually) all types declared in Isabelle’s version 99.

Expressions containing some of the keywords \texttt{let}, \texttt{if} etc. are called script-expressions.

Tactics \texttt{tac} are (curried) functions. For clarity and simplicity reasons, \texttt{listexpr} must not contain a \texttt{tac}, and \texttt{tacs} must not be nested.

### 6.2 Control the flow of evaluation

The flow of control is managed by the following script-expressions called tacticals.

\begin{align*}
\texttt{while} & \ \texttt{prop} \ \texttt{Do} \ \texttt{expr} \ \texttt{id} \\
\texttt{if} & \ \texttt{prop} \ \texttt{then} \ \texttt{expr} \ \texttt{else} \ \texttt{expr}
\end{align*}

While the above script-expressions trigger the flow of control by evaluating the current formula, the other expressions depend on the applicability of the tactics within their respective subexpressions (which in turn depends on the proofstate)

\begin{align*}
\texttt{Repeat} & \ \texttt{expr} \ \texttt{id} \\
\texttt{Try} & \ \texttt{expr} \ \texttt{id} \\
\texttt{expr} \ \texttt{Or} & \ \texttt{expr} \ \texttt{id} \\
\texttt{expr} \ \texttt{@@} & \ \texttt{expr} \ \texttt{id}
\end{align*}

xxx
Chapter 7

Do a calculational proof

First we list all the tactics available so far (this list may be extended during further development of ISAc).

7.1 Tactics for doing steps in calculations
Chapter 8

ISACs tactics

Init_Proof_Hid (dialogmode, formalization, specification) transfers the arguments to the math engine, the latter two in order to solve the example automatically. The tactic is not intended to be used by the student; it generates a proof tree with an empty model.

Init_Proof generates a proof tree with an empty model.

Model_Problem problem determines a problemtype (eventually found in the hierarchy) to be used for modeling.

Add_Given, Add_Find, Add_Relation formula inputs a formula to the respective field in a model (necessary as long as there is no facility for the user to input formula directly, and not only select the respective tactic plus formula from a list).

Specify_Theory theory, Specify_Problem problem, Specify_Method method specifies the respective element of the knowledgebase.

Refine_Problem problem searches for a matching problem in the hierarchy below 'problem'.

Apply_Method method finishes the model and specification phase and starts the solve phase.

Free_Solve initiates the solve phase without guidance by a method.

Rewrite theorem applies 'theorem' to the current formula and transforms it accordingly (if possible – otherwise error).

Rewrite_Asm theorem is the same tactic as 'Rewrite', but stores an eventual assumption of the theorem (instead of evaluating the assumption, i.e. the condition)

Rewrite_Set ruleset similar to 'Rewrite', but applies a whole set of theorems ('ruleset').
**Rewrite.Inst** (substitution, theorem), **Rewrite.Set.Inst** (substitution, ruleset) similar to the respective tactics, but substitute a constant (e.g. a bound variable) in 'theorem' before application.

**Calculate operation** calculates the result of numerals w.r.t. 'operation' (plus, minus, times, cancel, pow, sqrt) within the current formula.

**Substitute substitution** applies 'substitution' to the current formula and transforms it accordingly.

**Take formula** starts a new sequence of calculations on 'formula' within an already ongoing calculation.

**Subproblem (theory, problem)** initiates a subproblem within a calculation.

**Function formula** calls a function, where 'formula' contains the function name, e.g. 'Function (solve 1 + 2x + 3x^2 = 0 x)'. In this case the modelling and specification phases are suppressed by default, i.e. the solving phase of this subproblem starts immediately.

**Split_And, Conclude_And, Split_Or, Conclude_Or, Begin_Trans, End_Trans, Begin_Sequ, End_Sequ, Split_Intersect, End_Intersect** concern the construction of particular branches of the prooftree; usually suppressed by the dialog guide.

**Check_elementwise assumptions** w.r.t. the current formula which comprises elements in a list.

**Or_to_List** transforms a conjunction of equations to a list of equations (a questionable tactic in equation solving).

**Check_postcond**: check the current formula w.r.t. the postcondition on finishing the respective (sub)problem.

**End_Proof** finishes a proof and delivers a result only if 'Check_postcond' has been successful before.

### 8.1 The functionality of the math engine

A proof is being started in the math engine me by the tactic 1 **Init.Proof**, and interactively promoted by other tactics. On input of each tactic the me returns the resulting formula and the next tactic applicable. The proof is finished, when the me proposes **End.Proof** as the next tactic.

We show a calculation (calculational proof) concerning equation solving, where the type of equation is refined automatically: The equation is given by the respective formalization ...

---

1In the present version a tactic is of type **mstep**.
... and the specification spec of a domain dom, a problem type pbt and a method met. Note that the equation is such, that it is not immediately clear, what type it is in particular (it could be a polynomial of degree 2; but, for sure, the type is some specialized type of a univariate equation). Thus, no method (no_met) can be specified for solving the problem.

Nevertheless this specification is sufficient for automatically solving the equation — the appropriate method will be found by refinement within the hierarchy of problem types.

### 8.2 Initialize the calculation

The start of a new proof requires the following initializations: The proof state is given by a proof tree ptree and a position pos'; both are empty at the beginning. The tactic Init_Proof is, like all other tactics, paired with an identifier of type string for technical reasons.

```ml
ML> val (mID,m) = ("Init_Proof",Init_Proof (fmz, (dom,pbt,met)));
val mID = "Init_Proof" : string
val m = Init_Proof
  ("equality ((x+#1)*(x+#2)=x^^^#2+#8)","solveFor x","errorBound (eps=#0)",
  "solutions L"),("SqRoot.thy","univariate","equation"),
  ("SqRoot.thy","no_met") : mstep
ML> val (p,_,f,nxt,_,pt) = me (mID,m) e_pos' c EmptyPtree;
val p = ([],Pbl) : mout
val f = Form' (PpcKF (0,EdUndef,0,Nundef,(#,#))) : mout
val nxt = ("Refine_Tacitly",Refine_Tacitly ["univariate","equation"])
  : string * mstep
val pt = Nd (PblObj {branch=#,cell=#,env=#,loc=#,meth=#,model=#,origin=#,ostate=#,probl=#,result=#,spec=#,[]}) : ptree
```

The mathematics engine me returns the resulting formula f of type mout (which in this case is a problem), the next tactic nxt, and a new proof state (ptree, pos').

We can convince ourselves, that the problem is still empty, by increasing Compiler.Control.Print.printDepth.
Recall, please, the format of a problem as presented in sect. 5.4 on p. 26; such an 'empty' problem can be found above encapsulated by several constructors containing additional data (necessary for the dialog guide, not relevant here).

In the sequel we will omit output of the me if it is not important for the respective context.

In general, the dialog guide will hide the following two tactics Refine_Tacitly and Model_Problem from the user.

8.3 The phase of modeling

comprises the input of the items of the problem; the me can help by use of the formalization tacitly transferred by Init_Proof. In particular, the me in general 'knows' the next promising tactic; the first one has been returned by the (hidden) tactic Model_Problem.

ML> Compiler.Control.Print.printDepth:=8; (*4 default*)
val it = () : unit
ML>
ML> f;
val it = 
Form' 
(PpcKF 
(0,EdUndef,0,Nundef, 
(Problem [], 
{Find=[Incompl "solutions []"], 
Given=[Incompl "equality",Incompl "solveFor"],Relate=[], 
Where=[False "matches (?a = ?b) e_"],With=[]))): mout

ML> nxt;
val it = ("Refine_Tacitly",Refine_Tacitly ["univariate","equation"])
 : string * mstep
ML>
ML> val (p,_,f,nxt,_,pt) = me nxt p [1] pt;
val nxt = ("Model_Problem",Model_Problem ["normalize","univariate","equation"])
 : string * mstep
ML>
ML> val (p,_,f,nxt,_,pt) = me nxt p [1] pt;

ML> nxt;
val it = 
("Add_Given",Add_Given "equality ((x + #1) * (x + #2) = x ^^ #2 + #8)")
 : string * mstep
ML>
ML> val (p,_,f,nxt,_,pt) = me nxt p [1] pt;
val nxt = ("Add_Given",Add_Given "solveFor x") : string * mstep
ML>
ML> val (p,_,f,nxt,_,pt) = me nxt p [1] pt;
val nxt = ("Add_Find","Add_Find "solutions L") : string * mstep
ML>
ML> val (p,_,f,nxt,_,pt) = me nxt p [1] pt;
val f = Form' (PpcKF (0,EdUndef,0,Nundefined,(#,#))) : mout

Now the problem is 'modeled', all items are input. We convince ourselves by increasing Compiler.Control.Print.printDepth once more.

ML> Compiler.Control.Print.printDepth:=8;
ML> f;
val it =
Form' (PpcKF
(0,EdUndef,0,Nundefined,
Problem [],
{Find=[Correct "solutions L"],
Given=[Correct "equality 
((x + #1) \times (x + #2) = x ^ {#2} + #8)",
Correct "solveFor x"],Relate=[],Where=[],With=[]})} : mout

8.4 The phase of specification

This phase provides for explicit determination of the domain, the problem type, and the method to be used. In particular, the search for the appropriate problem type is being supported. There are two tactics for this purpose: Specify_Problem generates feedback on how a candidate of a problem type matches the current problem, and Refine_Problem provides help by the system, if the user gets lost.

ML> nxt;
val it = ("Specify_Domain",Specify_Domain "SqRoot.thy") : string * mstep
ML>
ML> val (p,_,f,nxt,_,pt) = me nxt p [1] pt;
val nxt =
("Specify_Problem",Specify_Problem ["normalize","univariate","equation"])
: string * mstep
val pt =
Nd
(PblObj
{branch=#,cell=#,env=#,loc=#,meth=#,model=#,origin=#,ostate=#,probl=#,result=#,spec=#,},[]): ptree

The me is smart enough to know the appropriate problem type (transferred tacitly with Init_Proof). In order to challenge the student, the dialog guide may hide this information; then the me works as follows.

ML> val nxt = ("Specify_Problem",
Specify_Problem ["polynomial","univariate","equation"]);
ML> val (p,_,f,nxt,_,pt) = me nxt p [1] pt;
val f = Form' (PpcKF (0,EdUndef,0,Nundefined,(#,#))) : mout
val nxt =
("Refine_Problem",Refine_Problem ["normalize","univariate","equation"])
: string * mstep
ML>
ML> val nxt = ("Specify_Problem",

39
Again assuming that the dialog guide hide the next tactic proposed by the me, and the student gets lost, Refine_Problem always 'knows' the way out, if applied to the problem type "univariate", "equation".

ML> val nxt = ("Refine_Problem", Refine_Problem ("linear","univariate","equation")
ML> val (p,_,f,nxt,_,pt) = me nxt p [1] pt; val f = Problems (RefinedKF [NoMatch #]) : mout
ML> Compiler.Control.Print.printDepth:=9;f;Compiler.Control.Print.printDepth:=4; val f =

(RefinedKF [NoMatch [
("linear","univariate","equation"),
{Find=[Correct "solutions L"],
 Given=[Correct "equality ((x + #1) * (x + #2) = x ^^^ #2 + #8)"
 Correct "solveFor x"],Relate=[],
 Where=[False
 "matches (?a + ?b * x = #0) ((x + #1) * (x + #2) = x ^^^ #2 + #8)"],
 With=[[])]) : mout
ML>
ML> val nxt = ("Refine_Problem",Refine_Problem ("univariate","equation")
ML> val (p,_,f,nxt,_,pt) = me nxt p [1] pt; val f = Problems (RefinedKF [Matches #,NoMatch #,NoMatch #,NoMatch #,NoMatch #,Matches #]) : mout
ML> Compiler.Control.Print.printDepth:=9;f;Compiler.Control.Print.printDepth:=4; val f =

(RefinedKF [Matches [
("univariate","equation"),
{Find=[Correct "solutions L"],
 Given=[Correct "equality ((x + #1) * (x + #2) = x ^^^ #2 + #8)"
 Correct "solveFor x"],Relate=[],
 Where=[Correct
 With=[[]])},

The tactic Refine_Problem returns all matches to problem types along the path traced in the problem hierarchy (analogously to the authoring tool for refinement in sect. ?? on p. ??) — a lot of information to be displayed appropriately in the hierarchy browser!

8.5 The phase of solving

This phase starts by invoking a method, which achieves the normal form within two tactics, Rewrite rnorm_equation_add and Rewrite_SetSqRoot_simplify:

ML> nxt;
val it = ("Apply_Method",Apply_Method ("SqRoot.thy","norm_univar_equation")) : string * mstep
ML> val (p,_,f,nxt,_,pt) = me nxt p [1] pt;
val f = Form\' (FormKF ("1,EdUndef,1,Nundef,"(x + #1) * (x + #2) = x "^^' #2 + #8"))
val nxt = ("Rewrite", Rewrite ("rnorm_equation_add"," ?b =!= #0 ==> (?a = ?b) = (?a + #-1 * ?b = #0")

ML> val (p,_,f,nxt,_,pt) = me nxt p [1] pt;
val f = Form\' (FormKF ("1,EdUndef,1,Nundef,"(x + #1) * (x + #2) + #-1 * (x "^^' #2 + #8) = #0")) : mout
val nxt = ("Rewrite_Set",Rewrite_Set "SqRoot_simplify") : string * mstep
ML> val (p,_,f,nxt,_,pt) = me nxt p [1] pt;
val f = Form\' (FormKF ("1,EdUndef,1,Nundef,="#-6 + #3 * x = #0") : mout
val nxt = ("Subproblem",Subproblem ("SqRoot.thy","#1")) : string * mstep

Now the normal form \(#-6 + #3 * x = #0\) is the input to a subproblem, which again allows for specification of the type of equation, and the respective method:

ML> nxt;
As required, the tactic \texttt{Refine ["univariate","equation"]} selects the appropriate type of equation from the problem hierarchy, which can be seen by the tactic \texttt{Model\_Problem ["linear","univariate","equation"]} prosed by the system.

Again the whole phase of modeling and specification follows; we skip it here, and \texttt{IS\_AC}’s dialog guide may decide to do so as well.

### 8.6 The final phase: check the post-condition

The type of problems solved by \texttt{IS\_AC} are so-called 'example construction problems' as shown above. The most characteristic point of such a problem is the post-condition. The handling of the post-condition in the given context is an open research question.

Thus the post-condition is just mentioned, in our example for both, the problem and the subproblem:

\begin{verbatim}
val it = ("Check\_Postcond",Check\_Postcond ["linear","univariate","equation"])
ML>
ML> val (p,_,f,nxt,_,pt) = me nxt p [1] pt;
val f = Form' (FormKF (~1,EdUndef,1,Nundef,"[x = #2]") : mout
val nxt = ("Check\_Postcond",Check\_Postcond ["normalize","univariate","equation"])
ML>
ML> val (p,_,f,nxt,_,pt) = me nxt p [1] pt;
val nxt = ("End\_Proof'",End\_Proof') : string * mstep
\end{verbatim}

The next tactic proposed by the system, \texttt{End\_Proof'}, indicates that the proof has finished successfully.

\textit{The tactics proposed by the system need not be followed by the user; the user is free to choose other tactics, and the system will report, if this is applicable at the respective proof state, or not! The reader may try out!}
Part II

Systematic description
Chapter 9

The structure of the knowledge base

9.1 Tactics and data

First we view the ME from outside, i.e. we regard tactics and relate them to the knowledge base (KB). W.r.t. the KB we address the atomic items which have to be implemented in detail by the authors of the KB. The items are listed in alphabetical order in Tab.9.1 on p.45. The relations between tactics and data items are shown in Tab.9.2 on p.46.

9.2 \textit{ISAC}'s theories

\textit{ISAC}'s theories build upon Isabelle's theories for high-order-logic (HOL) up to the respective development of real numbers (HOL/Real). Theories have a special format defined in [?] and the suffix *.thy; usually theories are paired with SML-files having the same filename and the suffix *.ML.

\textit{ISAC}'s theories represent the deductive part of \textit{ISAC}'s knowledge base, the hierarchy of theories is structured accordingly. The *.ML-files, however, contain all data of the other two axes of the knowledge base, the problems and the methods (without presenting their respective structure, which is done by the problem browser and the method browser, see 5).

Tab.9.3 on p.47 lists the basic theories planned to be implemented in version \textit{ISAC}.1. We expect the list to be expanded in the near future, actually, have a look to the theory browser!

The first three theories in the list do not belong to \textit{ISAC}'s knowledge base; they are concerned with \textit{ISAC}'s script-language for methods and listed here for completeness.

\footnote{Some of these items are fetched by the tactics from intermediate storage within the ME, and not directly from the KB.}
9.3 Data in *.thy- and *.ML-files

As already mentioned, theories come in pairs of *.thy- and *.ML-files with the same respective filename. How data are distributed between the two files is shown in Tab.9.4 on p.48. The order of the data-items within the theories should adhere to the order given in this list.

9.4 Formal description of the problem-hierarchy

9.5 Script tactics

The tactics actually promote the calculation: they construct the prooftree behind the scenes, and they are the points during evaluation where the script-interpreter transfers control to the user. Here we only describe the syntax of the tactics; the semantics is described on p.?? below in context with the tactics the student uses (‘user-tactics’): there is a 1-to-1 correspondence between user-tactics and script-tactics.
### Table 9.1: Atomic items of the KB

<table>
<thead>
<tr>
<th>abbreviation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>calc_list</code></td>
<td>associationlist of the evaluation-functions <code>eval_fn</code></td>
</tr>
<tr>
<td><code>eval_fn</code></td>
<td>evaluation-function for numerals and for predicates coded in SML</td>
</tr>
<tr>
<td><code>eval_rls</code></td>
<td>ruleset <code>rls</code> for simplifying expressions with <code>eval_fns</code></td>
</tr>
<tr>
<td><code>fmz</code></td>
<td>formalization, i.e. a minimal formula representation of an example</td>
</tr>
<tr>
<td><code>met</code></td>
<td>a method, i.e. a datastructure holding all informations for the solving phase (rew_ord, scr, etc.)</td>
</tr>
<tr>
<td><code>metID</code></td>
<td>reference to a <code>met</code></td>
</tr>
<tr>
<td><code>op</code></td>
<td>operator as key to an <code>eval_fn</code> in a <code>calc_list</code></td>
</tr>
<tr>
<td><code>pbl</code></td>
<td>problem, i.e. a node in the problem-hierarchy</td>
</tr>
<tr>
<td><code>pblID</code></td>
<td>reference to a <code>pbl</code></td>
</tr>
<tr>
<td><code>rew_ord</code></td>
<td>rewrite-order</td>
</tr>
<tr>
<td><code>rls</code></td>
<td>ruleset, i.e. a datastructure holding theorems <code>thm</code> and operators <code>op</code> for simplification (with a <code>rew_ord</code>)</td>
</tr>
<tr>
<td><code>Rrls</code></td>
<td>ruleset for 'reverse rewriting' (an ISAC-technique generating stepwise rewriting, e.g. for cancelling fractions)</td>
</tr>
<tr>
<td><code>scr</code></td>
<td>script describing algorithms by tactics, part of a <code>met</code></td>
</tr>
<tr>
<td><code>norm_rls</code></td>
<td>special ruleset calculating a normalform, associated with a <code>thy</code></td>
</tr>
<tr>
<td><code>spec</code></td>
<td>specification, i.e. a tripel (<code>thyID</code>, <code>pblID</code>, <code>metID</code>)</td>
</tr>
<tr>
<td><code>subs</code></td>
<td>substitution, i.e. a list of variable-value-pairs</td>
</tr>
<tr>
<td><code>term</code></td>
<td>Isabelle term, i.e. a formula</td>
</tr>
<tr>
<td><code>thm</code></td>
<td>theorem</td>
</tr>
<tr>
<td><code>thy</code></td>
<td>theory</td>
</tr>
<tr>
<td><code>thyID</code></td>
<td>reference to a <code>thy</code></td>
</tr>
</tbody>
</table>
Table 9.2: Which tactic uses which KB’s item?

<table>
<thead>
<tr>
<th>tactic</th>
<th>input</th>
<th>norm</th>
<th>rew_rls</th>
<th>eval_rls</th>
<th>eval_calc</th>
<th>eval_eval</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>thy</td>
<td>scr</td>
<td>Rrls</td>
<td>thm</td>
<td>ord</td>
</tr>
<tr>
<td>Init_Proof</td>
<td>fmz</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>model phase</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add_#</td>
<td>term</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FormFK</td>
<td>model</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>specify phase</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specify_Theory</td>
<td>thyID</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specify_Problem</td>
<td>pblID</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Refine_Problem</td>
<td>pblID</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Specify_Method</td>
<td>metID</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Apply_Method</td>
<td>metID</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>solve phase</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rewrite,Inst</td>
<td>thm</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>met</td>
<td>met</td>
</tr>
<tr>
<td>Rewrite,Detail</td>
<td>thm</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>rls</td>
<td>rls</td>
</tr>
<tr>
<td>Rewrite,Detail</td>
<td>thm</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>Rrls</td>
<td>Rrls</td>
</tr>
<tr>
<td>Rewrite_Set,Inst</td>
<td>rls</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Calculate</td>
<td>op</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Substitute</td>
<td>subs</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SubProblem</td>
<td>spec</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fmz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 9.3: Theories in \texttt{ISAC}-version I

<table>
<thead>
<tr>
<th>theory</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListI.thy</td>
<td>assigns identifiers to the functions defined in \texttt{Isabelle2002/src/HOL/List.thy} and (intermediately ?) defines some more list functions</td>
</tr>
<tr>
<td>ListI.ML</td>
<td>\texttt{eval_fn} for the additional list functions</td>
</tr>
<tr>
<td>Tools.thy</td>
<td>functions required for the evaluation of scripts</td>
</tr>
<tr>
<td>Tools.ML</td>
<td>the respective \texttt{eval_fns}</td>
</tr>
<tr>
<td>Script.thy</td>
<td>prerequisites for scripts: types, tactics, tacticals,</td>
</tr>
<tr>
<td>Script.ML</td>
<td>sets of tactics and functions for internal use</td>
</tr>
<tr>
<td>Typefix.thy</td>
<td>an intermediate hack for escaping type errors</td>
</tr>
<tr>
<td>Describe.thy</td>
<td>descriptions for the formulas in \texttt{models} and \texttt{problems}</td>
</tr>
<tr>
<td>Atools</td>
<td>(re-)definition of operators; general predicates and functions for preconditions; theorems for the \texttt{eval_rls}</td>
</tr>
<tr>
<td>Float</td>
<td>floating point numerals</td>
</tr>
<tr>
<td>Equation</td>
<td>basic notions for equations and equational systems</td>
</tr>
<tr>
<td>Poly</td>
<td>polynomials</td>
</tr>
<tr>
<td>PolyEq</td>
<td>polynomial equations and equational systems</td>
</tr>
<tr>
<td>Rational.thy</td>
<td>additional theorems for rationals</td>
</tr>
<tr>
<td>Rational.ML</td>
<td>cancel, add and simplify rationals using (a generalization of) Euclids algorithm; respective reverse rulesets</td>
</tr>
<tr>
<td>RatEq</td>
<td>equations on rationals</td>
</tr>
<tr>
<td>Root</td>
<td>radicals; calculate normalform; respective reverse rulesets</td>
</tr>
<tr>
<td>RootEq</td>
<td>equations on roots</td>
</tr>
<tr>
<td>RatRootEq</td>
<td>equations on rationals and roots (i.e. on terms containing both operations)</td>
</tr>
<tr>
<td>Vect</td>
<td>vector analysis</td>
</tr>
<tr>
<td>Trig</td>
<td>trigonometry</td>
</tr>
<tr>
<td>LogExp</td>
<td>logarithms and exponential functions</td>
</tr>
<tr>
<td>Calculus</td>
<td>nonstandard analysis</td>
</tr>
<tr>
<td>Diff</td>
<td>differentiation</td>
</tr>
<tr>
<td>DiffApp</td>
<td>applications of differentiaten (maxima-minima-problems)</td>
</tr>
<tr>
<td>Test</td>
<td>(old) data for the test suite</td>
</tr>
<tr>
<td>Isac</td>
<td>collects all \texttt{ISAC}-theoris.</td>
</tr>
</tbody>
</table>
### Table 9.4: Data in \*.thy- and \*.ML-files

<table>
<thead>
<tr>
<th>file</th>
<th>data</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*.thy</td>
<td>consts</td>
<td>operators, predicates, functions, script-names (&quot;Script name ...arguments&quot;)</td>
</tr>
<tr>
<td></td>
<td>rules</td>
<td>theorems: \textsc{Isac} uses Isabelle’s theorems if possible; additional theorems analogous to such existing in Isabelle get the Isabelle-identifier attached an \textit{I}</td>
</tr>
<tr>
<td>*.ML</td>
<td>\texttt{theory’ :=}</td>
<td>the theory defined by the actual *.thy-file is made accessible to \textsc{Isac}</td>
</tr>
<tr>
<td></td>
<td>\texttt{eval_fn}</td>
<td>evaluation function for operators and predicates, coded on the meta-level (SML); the identifier of such a function is a combination of the keyword \texttt{eval} with the identifier of the function as defined in *.thy</td>
</tr>
<tr>
<td></td>
<td>\texttt{*_simplify}</td>
<td>the canonical simplifier for the actual theory, i.e. the identifier for this ruleset is a combination of the theories identifier and the keyword \texttt{*_simplify}</td>
</tr>
<tr>
<td></td>
<td>\texttt{norm_rls :=}</td>
<td>the canonical simplifier \texttt{*_simplify} is stored such that it is accessible for \textsc{Isac}</td>
</tr>
<tr>
<td></td>
<td>\texttt{rew_ord’ :=}</td>
<td>the same for rewrite orders, if needed outside of one particular ruleset</td>
</tr>
<tr>
<td></td>
<td>\texttt{ruleset’ :=}</td>
<td>the same for rulesets (ordinary rulesets, reverse rulesets and \texttt{eval_rls})</td>
</tr>
<tr>
<td></td>
<td>\texttt{calc_list :=}</td>
<td>the same for \texttt{eval_fns}, if needed outside of one particular ruleset (e.g. for a tactic \texttt{Calculate} in a script)</td>
</tr>
<tr>
<td></td>
<td>\texttt{store_pbl}</td>
<td>problems defined within this *.ML-file are made accessible for \textsc{Isac}</td>
</tr>
<tr>
<td></td>
<td>\texttt{methods :=}</td>
<td>methods defined within this *.ML-file are made accessible for \textsc{Isac}</td>
</tr>
</tbody>
</table>
Part III

Authoring on the knowledge
9.6 Add a theorem
9.7 Define and add a problem
9.8 Define and add a predicate
9.9 Define and add a method
9.10
9.11
9.12
9.13