ISAC

Experimente zur Computermathematik

und

Handbuch für Autoren der
Mathematik-Wissensbasis

Alexandra Hirn und Eva Rott
isac-users@ist.tugraz.at

August 5, 2010

Contents

1 Einleitung **4**
- 1.1 “Authoring” und “Tutoring” 4
- 1.2 Der Inhalt des Dokuments 4
- 1.3 Gleich am Computer ausprobieren! 5

I Experimentelle Annäherung **6**

2 Terme und Theorien **7**
- 2.1 Von der Formel zum Term 7
- 2.2 “Theory” und “Parsing“ 8
- 2.3 Details von Termen 10

3 ”Rewriting“ **13**
- 3.1 Was ist Rewriting? 13
- 3.2 Welche Wünsche kann man an Rewriting stellen? 14
- 3.3 Rule sets 15
- 3.4 Berechnung von Konstanten 16

4 Termordnung **18**
- 4.1 Beispiel für Termordnungen 18
- 4.2 Geordnetes Rewriting 18

5 Problem hierachy **19**
- 5.1 ”Matching“ 19
- 5.2 Zugriff auf die hierachy 21
- 5.3 Die passende ”formalization“ für den problem type 22
- 5.4 ”problem-refinement“ 22

6 ”Methods“ **24**
- 6.1 Der ”Syntax“ des script 24
- 6.2 Überprüfung der Auswertung 25

7 Befehle von ISAC **26**
7.1 Die Funktionsweise der mathematic engine 28
7.2 Der Beginn einer Rechnung 28
7.3 The phase of modeling 29
7.4 The phase of specification 30
7.5 The phase of solving 32
7.6 The final phase: Überprüfung der "post-condition" 33

II Handbuch für Autoren **34**

8 Die Struktur des Grundlagenwissens **35**
8.1 "tactics" und Daten 35
8.2 Die theories von ISAC 35
8.3 Daten in `*.thy` und `*.ML` 36
8.4 Formale Beschreibung der Hierarchie von Problemen 36
8.5 Skripttaktiken 36

III Authoring on the knowledge **41**
8.6 Add a theorem 42
8.7 Define and add a problem 42
8.8 Define and add a predicate 42
8.9 Define and add a method 42
8.10 42
8.11 42
8.12 42
8.13 42

List of Tables

8.1 Kleinste Teilchen des KB 37
8.2 Welche tactics verwenden die Teile des KB ? 38
8.3 theory von der ersten Version von ISAC 39
8.4 Daten in `*.thy`- und `*.ML`-files 40

Chapter 1

Einleitung

Dies ist die Übersetzung der dersten Kapitel einer englischen Version [1], auf den Stand von ISAC 2008 gebracht. Die Übersetzung und Anpassung erfolgte durch die Autorinnen im Rahmen einer Ferialpraxis am Institut für Softwaretechnologie der TU Graz.

Diese Version zeichnet sich dadurch aus, dass sie von "Nicht-Computer-Freaks" für "Nicht-Computer-Freaks" geschrieben wurde.

1.1 "Authoring" und "Tutoring"

TO DO Mathematik lernen – verschiedene Autoren – Isabelle Die Grundlage für ISAC bildet Isabelle. Dies ist ein "theorem prover", der von L. Paulson und T. Nipkow entwickelt wird und Hard- und Software prüft.

1.2 Der Inhalt des Dokuments

TO DO Als Anleitung: Dieses Dokument beschreibt das Kerngebiet (KE) von ISAC, das Gebiet der mathematics engine (ME) im Kerngebiet und die verschiedenen Funktionen wie das Umschreiben und der Vergleich.

ISAC und KE wurden in SML geschrieben, die Sprache in Verbindung mit dem Vorgänger des theorem Provers Isabelle entwickelt. So kam es, dass in diesem Dokument die Ebene ASCII als SML Code präsentiert wird. Der Leser wird vermutlich erkennen, dass der ISAC Benutzer eine vollkommen andere Sichtweise auf eine grafische Benutzeroberfläche bekommt.

Das Dokument ist eigenständig; Basiswissen über SML (für eine Einführung siehe [**?**]), Terme und Umschreibung wird vorrausgesetzt.

Hinweis: SML Code, Verzeichnis, Dateien sind `in 'tt' geschrieben`; besonders in `ML>` ist das Kerngebiet schnell.

[1] http://www.ist.tugraz.at/projects/isac/publ/mat-eng.pdf

Versuchen Sie es! Ein weiteres Anliegen dieses Textes ist, dem Leser Tipps für Versuche mit den Anwendungen zu geben.

1.3 Gleich am Computer ausprobieren!

TO DO screenshot Bevor Sie mit Ihren Versuchen beginnen, möchten wir Ihnen noch einige Hinweise geben:

- System starten
- Shell aufmachen und die Datei mat-eng-de.sml öffnen.
- $>$: Hinter diesem Zeichen ("Prompt") stehen jene, die Sie selbst eingeben bzw. mit Copy und Paste aus der Datei kopieren.
- Die Eingabe wird mit ";" und "Enter" abgeschlossen.
- Zeilen, die nicht mit Prompt beginnen, werden vom Computer ausgegeben.

Part I

Experimentelle Annäherung

Chapter 2

Terme und Theorien

Wie bereits erwähnt, geht es um Computer-Mathematik. In den letzten Jahren hat die "computer science" grosse Fortschritte darin gemacht, Mathematik auf dem Computer verständlich darzustellen. Dies gilt für mathematische Formeln, für die Beschreibung von Problemen, für Lösungsmethoden etc. Wir beginnen mit mathematischen Formeln.

2.1 Von der Formel zum Term

Um ein Beispiel zu nennen: Die Formel $a + b \cdot 3$ lässt sich in lesbarer Form so eingeben:

```
> "a + b * 3";
   val it = "a + b * 3" : string
```

"a + b * 3" ist also ein string (eine Zeichenfolge). In dieser Form weiss der Computer nicht, dass z.B. eine Multiplikation *vor* einer Addition zu rechnen ist. Isabelle braucht dazu eine andere Darstellung für Formeln. In diese kann man mit der Funktion `str2term` (string to term) umrechnen:

```
> str2term "a + b * 3";
   val it =
      Const ("op +", "[RealDef.real, RealDef.real] => RealDef.real") $
            Free ("a", "RealDef.real") $
         (Const ("op *", "[RealDef.real, RealDef.real] => RealDef.real") $ ... $
            ...) : Term.term
```

Diese Form heisst `term` und ist nicht für den Menschen zum lesen gedacht. Isabelle braucht sie aber intern zum Rechnen. Wir wollen sie mit Hilfe von `val` (value) auf der Variable `t` speichern:

```
> val t = str2term "a + b * 3";
   val t =
      Const ("op +", "[RealDef.real, RealDef.real] => RealDef.real") $
            Free ("a", "RealDef.real") $
         (Const ("op *", "[RealDef.real, RealDef.real] => RealDef.real") $ ... $
            ...) : Term.term
```

Von dieser Variablen `t` kann man den Wert jederzeit abrufen:

```
> t;
   val it =
      Const ("op +", "[RealDef.real, RealDef.real] => RealDef.real") $
            Free ("a", "RealDef.real") $
         (Const ("op *", "[RealDef.real, RealDef.real] => RealDef.real") $ ... $
            ...) : Term.term
```

Der auf `t` gespeicherte Term kann einer Funktion `atomty` übergeben werden, die diesen in einer dritten Form zeigt:

```
> atomty term;

***
*** Const (op +, [real, real] => real)
*** . Free (a, real)
*** . Const (op *, [real, real] => real)
*** . . Free (b, real)
*** . . Free (3, real)
***

val it = () : unit
```

Diese Darstellung nennt man "abstract syntax" und macht unmittelbar klar, dass man a und b nicht addieren kann, weil ein Mal vorhanden ist. Es gibt noch eine vierte Art von Term, den cterm. Er wird weiter unten verwendet, weil er sich als string lesbar darstellt.

2.2 "Theory" und "Parsing"

Der Unterschied zwischen *ISAC* und bisheriger Mathematiksoftware (GeoGebra, Mathematica, Maple, Derive etc.) ist, dass das mathematische Wissen nicht im Programmcode steht, sondern in sogenannten theories (Theorien). Dort wird das Mathematikwissen in einer für nicht Programmierer lesbaren Form geschrieben. Das Wissen von *ISAC* ist in folgenden Theorien entahlten:

```
> Isac.thy;
   val it =
      {ProtoPure, CPure, HOL, Set, Typedef, Fun, Product_Type, Lfp, Gfp,
      Sum_Type, Relation, Record, Inductive, Transitive_Closure,
      Wellfounded_Recursion, NatDef, Nat, NatArith, Divides, Power,
      SetInterval, Finite_Set, Equiv, IntDef, Int, Datatype_Universe,
      Datatype, Numeral, Bin, IntArith, Wellfounded_Relations, Recdef, IntDiv,
      IntPower, NatBin, NatSimprocs, Relation_Power, PreList, List, Map,
      Hilbert_Choice, Main, Lubs, PNat, PRat, PReal, RealDef, RealOrd,
      RealInt, RealBin, RealArith0, RealArith, RComplete, RealAbs, RealPow,
      Ring_and_Field, Complex_Numbers, Real, ListG, Tools, Script, Typefix,
      Float, ComplexI, Descript, Atools, Simplify, Poly, Rational, PolyMinus,
      Equation, LinEq, Root, RootEq, RatEq, RootRat, RootRatEq, PolyEq, Vect,
      Calculus, Trig, LogExp, Diff, DiffApp, Integrate, EqSystem, Biegelinie,
      AlgEin, Test, Isac} : Theory.theory
```

ProtoPure und CPure enthalten diese logischen Grundlagen, die in HOL und den nachfolgenden Theorien erweitert werden. *ISAC* als letzte Theorie beinhaltet das gesamte Wissen. Dass das Mal vor dem Plus berechnet wird, ist so festgelegt:

```
class plus =
fixes plus :: "'a \<Rightarrow> 'a \<Rightarrow> 'a" (infixl "+" 65)

class minus =
fixes minus :: "'a \<Rightarrow> 'a \<Rightarrow> 'a" (infixl "-" 65)

class uminus =
fixes uminus :: "'a \<Rightarrow> 'a" ("- _" [81] 80)

class times =
fixes times :: "'a \<Rightarrow> 'a \<Rightarrow> 'a" (infixl "*" 70)
```

infix gibt an, dass der Operator zwischen den Zahlen steht und nicht, wie in "abstract syntax", vorne oben. Die Zahlen rechts davon legen die Priorität fest. 70 für Mal ist grösser als 65 für Plus und wird daher zuerst berechnet.

Wollen Sie wissen, wie die einzelnen Rechengesetze aussehen, können Sie im Internet folgenden Link ansehen: http://isabelle.in.tum.de/dist/library/HOL/Groups.html

Der Vorgang, bei dem aus einem string ein Term entsteht, nennt man Parsing. Dazu wird Wissen aus der Theorie benötigt, denn str2term nimmt intern eine parse-Funktion, bei der immer das gesamte *ISAC*-Wissen verwendet wird. Bei dieser Funktion wird weiters festgelegt, aus welcher Theorie das Wissen genommen werden soll.

```
> parse Isac.thy "a + b";
   val it = Some "a + b" : Thm.cterm Library.option
```

Um sich das Weiterrechnen zu erleichtern, kann das Ergebnis vom Parsing auf eine Variable, wie zum Beispiel t gespeichert werden:

```
> val t = parse Isac.thy "a + b";
   val t = Some "a + b" : Thm.cterm Library.option
```

Some bedeutet, dass das nötige Wissen vorhanden ist, um die Rechnung durchzuführen. None zeigt uns, dass das Wissen fehlt oder ein Fehler aufgetreten ist. Daher sieht man im folgenden Beispiel, dass HOL.thy nicht ausreichend Wissen enthält:

```
> parse HOL.thy "a + b";
   val it = None : Thm.cterm Library.option
```

Anschliessend zeigen wir Ihnen noch ein zweites Beispiel, bei dem sowohl ein Fehler aufgetreten ist, als auch das Wissen fehlt:

```
> parse Isac.thy "a + ";
   *** Inner syntax error: unexpected end of input
   *** Expected tokens: "contains_root" "is_root_free" "q_" "M_b" "M_b'"
   *** "Integral" "differentiate" "E_" "some_of" "||" "|||" "argument_in"
```

```
   *** "filter_sameFunId" "I__" "letpar" "Rewrite_Inst" "Rewrite_Set"
   *** "Rewrite_Set_Inst" "Check_elementwise" "Or_to_List" "While" "Script"
   *** "\\" "\\" "\\" "CHR" "xstr" "SOME" "\\" "@"
   *** "GREATEST" "[" "[]" "num" "\\" "{)" "{.." "\\" "(|"
   *** "\\" "SIGMA" "()" "\\" "PI" "\\" "\\" "{" "INT"
   *** "UN" "{}" "LEAST" "\\" "0" "1" "-" "!" "?" "?!" "\\"
   *** "\\" "\\" "\\!" "THE" "let" "case" "~" "if" "ALL"
   *** "EX" "EX!" "!!" "_" "\\" "\\" "PROP" "[|" "OFCLASS"
   *** "\\" "op" "\\" "%" "TYPE" "id" "longid" "var" "..."
   *** "\\" "("
   val it = None : Thm.cterm Library.option
```

Das mathematische Wissen wächst mit jeder Theorie von ProtoPure bis Isac. In den folgenden Beispielen wird gezeigt, wie das Wissen wächst.

```
> (*-1-*);
> parse HOL.thy "2^^^3";
   *** Inner lexical error at: "^^^3"
   val it = None : Thm.cterm Library.option
```

"Inner lexical error" und "None" bedeuten, dass ein Fehler aufgetreten ist, denn das Wissen über `*` findet sich erst in der `theorie group`.

```
> (*-2-*);
> parse HOL.thy "d_d x (a + x)";
   val it = None : Thm.cterm Library.option
```

Hier wiederum ist noch kein Wissen über das Differenzieren vorhanden.

```
> (*-3-*);
> parse Rational.thy "2^^^3";
   val it = Some "2 ^^^ 3" : Thm.cterm Library.option

> (*-4-*);
> val Some t4 = parse Rational.thy "d_d x (a + x)";
   val t4 = "d_d x (a + x)" : Thm.cterm

> (*-5-*);
> val Some t5 = parse Diff.thy  "d_d x (a + x)";
   val t5 = "d_d x (a + x)" : Thm.cterm
```

Die letzen drei Aufgaben können schon gelöst werden, da `Rational.thy` über das nötige Wissen verfügt.

2.3 Details von Termen

Mit Hilfe der darunterliegenden Darstellung sieht man, dass ein cterm in einen Term umgewandelt werden kann.

```
> term_of;
   val it = fn : Thm.cterm -> Term.term
```

Durch die Umwandlung eines cterms in einen Term sieht man die einzelnen Teile des Terms. "Free" bedeutet, dass man die Variable ändern kann.

```
> term_of t4;
   val it =
      Free ("d_d", "[RealDef.real, RealDef.real] => RealDef.real") $ ... $
      ...: Term.term
```

In diesem Fall sagt uns das "Const", dass die Variable eine Konstante ist, also ein Fixwert, der immer die selbe Funktion hat.

```
> term_of t5;
   val it =
      Const ("Diff.d_d", "[RealDef.real, RealDef.real] => RealDef.real") $ ... $
      ... : Term.term
```

Sollten verschiedene Teile des "output" (= das vom Computer Ausgegebene) nicht sichtbar sein, kann man mit einem bestimmten Befehl alles angezeigt werden.

```
> print_depth;
   val it = fn : int -> unit
```

Zuerst gibt man den Befehl ein, danach den Term, der grösser werden soll. Dabei kann man selbst einen Wert für die Länge bestimmen.

```
> print_depth 10;
   val it = () : unit
> term_of t4;
      val it =
         Free ("d_d", "[RealDef.real, RealDef.real] => RealDef.real") $
             Free ("x", "RealDef.real") $
         (Const ("op +", "[RealDef.real, RealDef.real] => RealDef.real") $
             Free ("a", "RealDef.real") $ Free ("x", "RealDef.real"))
      : Term.term

> print_depth 10;
      val it = () : unit
> term_of t5;
      val it =
         Const ("Diff.d_d", "[RealDef.real, RealDef.real] => RealDef.real") $
             Free ("x", "RealDef.real") $
         (Const ("op +", "[RealDef.real, RealDef.real] => RealDef.real") $
             Free ("a", "RealDef.real") $ Free ("x", "RealDef.real"))
      : Term.term
```

Versuchen Sie es! Eine andere Variante um den Unterschied der beiden Terme zu sehen ist folgende:

```
> (*-4-*) val thy = Rational.thy;
   val thy =
      {ProtoPure, CPure, HOL, Set, Typedef, Fun, Product_Type, Lfp, Gfp,
      Sum_Type, Relation, Record, Inductive, Transitive_Closure,
```

```
      Wellfounded_Recursion, NatDef, Nat, NatArith, Divides, Power,
      SetInterval, Finite_Set, Equiv, IntDef, Int, Datatype_Universe,
      Datatype, Numeral, Bin, IntArith, Wellfounded_Relations, Recdef, IntDiv,
      IntPower, NatBin, NatSimprocs, Relation_Power, PreList, List, Map,
      Hilbert_Choice, Main, Lubs, PNat, PRat, PReal, RealDef, RealOrd,
      RealInt, RealBin, RealArith0, RealArith, RComplete, RealAbs, RealPow,
      Ring_and_Field, Complex_Numbers, Real, ListG, Tools, Script, Typefix,
      Float, ComplexI, Descript, Atools, Simplify, Poly, Rational}
  : Theory.theory
> ((atomty) 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


> (*-5-*) val thy = Diff.thy;
   val thy =
      {ProtoPure, CPure, HOL, Set, Typedef, Fun, Product_Type, Lfp, Gfp,
      Sum_Type, Relation, Record, Inductive, Transitive_Closure,
      Wellfounded_Recursion, NatDef, Nat, NatArith, Divides, Power,
      SetInterval, Finite_Set, Equiv, IntDef, Int, Datatype_Universe,
      Datatype, Numeral, Bin, IntArith, Wellfounded_Relations, Recdef, IntDiv,
      IntPower, NatBin, NatSimprocs, Relation_Power, PreList, List, Map,
      Hilbert_Choice, Main, Lubs, PNat, PRat, PReal, RealDef, RealOrd,
      RealInt, RealBin, RealArith0, RealArith, RComplete, RealAbs, RealPow,
      Ring_and_Field, Complex_Numbers, Real, Calculus, Trig, ListG, Tools,
      Script, Typefix, Float, ComplexI, Descript, Atools, Simplify, Poly,
      Equation, LinEq, Root, RootEq, Rational, RatEq, RootRat, RootRatEq,
      PolyEq, LogExp, Diff} : Theory.theory

> ((atomty) o term_of o the o (parse thy)) "d_d x (a + x)";

   ***
   *** Const (Diff.d_d, [real, real] => real)
   *** . Free (x, real)
   *** . Const (op +, [real, real] => real)
   *** . . Free (a, real)
   *** . . Free (x, real)
   ***

   val it = () : unit
```

Chapter 3

"Rewriting"

3.1 Was ist Rewriting?

Bei Rewriting handelt es sich um das Umformen von Termen nach vorgegebenen Regeln. Folgende zwei Funktionen sind notwendig:

```
> rewrite;
   val it = fn
   :
   theory' ->
   rew_ord' ->
   rls' -> bool -> thm' -> cterm' -> (string * string list) Library.option
```

Die Funktion hat zwei Argumente, die mitgeschickt werden müssen, damit die Funktion arbeiten kann. Das letzte Element `(cterm' * cterm' list) Library.option` im unteren Term ist das Ergebnis, das die Funktionen `rewrite` zurückgeben und die zwei vorhergehenden Argumente, `theorem` und `cterm`, sind die für uns wichtigen. `Theorem` ist die Rechenregel und `cterm` jene Formel auf die die Rechenregel angewendet wird.

```
> rewrite_inst;
   val it = fn
   :
   theory' ->
   rew_ord' ->
   rls' ->bool -> 'a -> thm' -> cterm' -> (cterm' * cterm' list) Library.option
```

Die Funktion `rewrite_inst` wird benötigt, um Gleichungen, Rechnungen zum Differenzieren etc. zu lösen. Dabei wird die gebundene Variable (bdv) instanziiert, d.h. es wird die Variable angegeben, nach der man differenzieren will, bzw. für die ein Wert bei einer Gleichung herauskommen soll. Um zu sehen wie der Computer vorgeht nehmen wir folgendes Beispiel, dessen Ergebnis offenbar 0 ist, was der Computer jedoch erst nach einer Reihe von Schritten herausfindet. Im Beispiel wird differenziert, wobei *ISAC*'s Schreibweise jene von Computer Algebra Systemen (CAS) anzugleichen: in CAS wird differenziert mit $\frac{d}{dx}\ x^2 + 3 \cdot x + 4$, in *ISAC* mit `d_d x (x ^^^ 2 + 3 * x + 4)`. Zuerst werden die einzelnen Werte als Variablen gespeichert:

```
> val thy' = "Diff.thy";
   val thy' = "Diff.thy" : string
> val ro = "tless_true";
   val ro = "tless_true" : string
> val er = "eval_rls";
   val er = "eval_rls" : string
> val inst = [("bdv","x::real")];
   val inst = [("bdv", "x::real")] : (string * string) list
> val ct = "d_d x (a + a * (2 + b))";
   val ct = "d_d x (a + a * (2 + b))" : string
```

Nun wird die Rechnung nach den Regeln ausgerechnet, wobei am Ende mehrere Dinge zugleich gemacht werden. Folgende Regeln werden benötigt: Summenregel, Produktregel, Multiplikationsregel mit einem konstanten Faktor und zum Schluss die Additionsregel.

```
> val Some (ct,_) = rewrite_inst thy' ro er true inst ("diff_sum","") ct;
   val ct = "d_d x a + d_d x (a * (2 + b))" : cterm'
> val Some (ct,_) = rewrite_inst thy' ro er true inst ("diff_prod","") ct;
   val ct = "d_d x a + (d_d x a * (2 + b) + a * d_d x (2 + b))" : cterm'
> val Some (ct,_) = rewrite_inst thy' ro er true inst ("diff_const","") ct;
   val ct = "d_d x a + (d_d x a * (2 + b) + a * 0) " : cterm'
> val Some (ct,_) = rewrite_inst thy' ro er true inst ("diff_const","") ct;
   val ct = "d_d x a + (0 * (2 + b) + a * 0)" : cterm'
> val Some (ct,_) = rewrite_inst thy' ro er true inst ("diff_const","") ct;
   val ct = "0 + (0 * (2 + b) + a * 0)" : cterm'
> val Some (ct,_) = rewrite_set thy' true "make_polynomial" ct;
   val ct = "0" : string
```

Was `rewrite_set` genau macht, finden Sie im nächsten Kapitel.
Dies wäre ein etwas ernsthafteres Beispiel zum Differenzieren:

```
> val ct = "d_d x (x ^^^ 2 + 3 * x + 4)";
> val Some (ct,_) = rewrite_inst thy' ro er true inst ("diff_sum","") ct;
```

Versuchen Sie es, diese Beispiel zu Ende zu führen! Die Regeln, die ISAC kennt und zum Umformen verwenden kann, finden Sie im Internet [1].

3.2 Welche Wünsche kann man an Rewriting stellen?

Es gibt verschiedene Varianten von Rewriting, die alle eine bestimmte Bedeutung haben. `rewrite_ set` wandelt Terme in ein ganzes rule set um, die normalerweise nur mit einem Theorem vereinfacht dargestellt werden. Hierbei werden auch folgende Argumente verwendet:

[1] http://www.ist.tugraz.at/projects/isac/www/kbase/thy/browser_info/HOL/HOL-Real/Isac/Diff.html

`theory`	Die Theory von Isabelle, die alle nötigen Informationen für das Parsing `term` enthält.
`rew_ord`	die Ordnung für das geordnete Rewriting. Für *no* benötigt das geordnete Rewriting `tless_true`, eine Ordnung, die für alle nachgiebigen Argumente true ist
`rls`	Das rule set; zur Auswertung von bestimmten Bedingungen in `thm` falls `thm` eine conditional rule ist
`bool`	ein Bitschalter, der die Berechnungen der möglichen condition in `thm` auslöst: wenn sie `false` ist, dann wird die condition bewertet und auf Grund des Resultats wendet man `thm` an, oder nicht; wenn `true` dann wird die condition nicht ausgewertet, aber man gibt sie in eine Menge von Hypothesen
`thm`	das theorem versucht den `term` zu überschreiben
`term`	`thm` wendet Rewriting möglicherweise auf den Term an

```
> rewrite_set;
   val it = fn : theory' -> bool -> rls' -> ...

> rewrite_set_inst;
   val it = fn : theory' -> bool -> subs' -> .
```

Wenn man sehen möchte wie Rewriting bei den einzelnen theorems funktioniert kann man dies mit `trace_rewrite` versuchen.

```
> trace_rewrite := true;
   val it = () : unit
```

3.3 Rule sets

Einige der oben genannten Varianten von Rewriting beziehen sich nicht nur auf einen theorem, sondern auf einen ganzen Block von theorems, die man als rule set bezeichnet. Dieser wird so lange angewendet, bis ein Element davon für Rewriting verwendet werden kann. Sollte der Begriff "terminate" fehlen, wird das Rule set nicht beendet und läuft weiter. Ein Beispiel für einen rule set ist folgendes:

```
???????????
```

```
> sym;
   val it = "?s = ?t ==> ?t = ?s" : Thm.thm
> rearrange_assoc;
   val it =
      Rls
         {id = "rearrange_assoc",
            scr = Script (Free ("empty_script", "RealDef.real")),
            calc = [],
```

```
            erls =
            Rls
               {id = "e_rls",
                  scr = EmptyScr,
                  calc = [],
                  erls = Erls,
                  srls = Erls,
                  rules = [],
                  rew_ord = ("dummy_ord", fn),
                  preconds = []},
            srls =
            Rls
               {id = "e_rls",
                  scr = EmptyScr,
                  calc = [],
                  erls = Erls,
                  srls = Erls,
                  rules = [],
                  rew_ord = ("dummy_ord", fn),
                  preconds = []},
            rules =
            [Thm ("sym_radd_assoc", "?m1 + (?n1 + ?k1) = ?m1 + ?n1 + ?k1"  [.]),
               Thm
                  ("sym_rmult_assoc",
                     "?m1 * (?n1 * ?k1) = ?m1 * ?n1 * ?k1"  [.])],
            rew_ord = ("e_rew_ord", fn),
            preconds = []} : rls
```

3.4 Berechnung von Konstanten

Sobald Konstanten in dem Bereich des Subterms sind, können sie von einer Funktion berechnet werden:

```
> calculate;
   val it = fn
   :
      theory' ->
      string *
      (
      string ->
      Term.term -> Theory.theory -> (string * Term.term) Library.option) ->
      cterm' -> (string * thm') Library.option

> calculate_;
   val it = fn
   :
      Theory.theory ->
      string *
      (
      string ->
      Term.term -> Theory.theory -> (string * Term.term) Library.option) ->
      Term.term -> (Term.term * (string * Thm.thm)) Library.option
```

Man bekommt das Ergebnis und das theorem bezieht sich darauf. Daher sind die folgenden mathematischen Rechnungen möglich:

```
> calclist;
   val it =
      [("Vars", ("Tools.Vars", fn)), ("matches", ("Tools.matches", fn)),
         ("lhs", ("Tools.lhs", fn)), ("plus", ("op +", fn)),
         ("times", ("op *", fn)), ("divide_", ("HOL.divide", fn)),
         ("power_", ("Atools.pow", fn)), ("is_const", ("Atools.is'_const", fn)),
         ("le", ("op <", fn)), ("leq", ("op <=", fn)),
         ("ident", ("Atools.ident", fn)), ("sqrt", ("Root.sqrt", fn)),
         ("Test.is_root_free", ("is'_root'_free", fn)),
         ("Test.contains_root", ("contains'_root", fn))]
   :
      (
      string *
      (
      string *
      (
      string ->
      Term.term -> Theory.theory -> (string * Term.term) Library.option))) list
```

Chapter 4

Termordnung

Die Anordnungen der Begriffe sind unverzichtbar für den Gebrauch des Umschreibens von normalen Funktionen und von normalen Formeln, die nötig sind um passende Modelle für Probleme zu finden.

4.1 Beispiel für Termordnungen

Es ist nicht unbedeutend, eine Verbindung zu Termen herzustellen, die wirklich eine Ordnung besitzen. Diese Ordnungen sind selbstaufrufende Bahnordnungen:

```
> sqrt_right;
   val it = fn : bool -> Theory.theory -> subst -> Term.term * Term.term -> b       ool
> tless_true;
   val it = fn : subst -> Term.term * Term.term -> bool
```

Das "bool" Argument gibt Ihnen die Möglichkeit, die Kontrolle zu den zugehörigen Unterordununngen zurück zu verfolgen, damit sich die Unterordnungen, die 'true' sind, als strings anzeigen lassen.

4.2 Geordnetes Rewriting

Beim Rewriting entstehen Probleme, die vom "law of commutativity" (= Kommutativgesetz) durch '+' und '*' verursacht werden. Diese Probleme können nur durch geordnetes Rewriting gelöst werden, da hier ein Term nur umgeschrieben wird, wenn ein kleinerer dadurch entsteht.

Chapter 5

Problem hierachy

5.1 "Matching"

Matching ist eine Technik von Rewriting, die von ISAC verwendet wird, um ein Problem und den passenden problem type dafür zu finden. Die folgende Funktion überprüft, ob Matching möglich ist:

```
> matches;
val it = fn : Theory.theory -> Term.term -> Term.term -> bool
```

Die folgende Gleichung wird in Operatoren und freie Variablen zerlegt.

```
> val t = (term_of o the o (parse thy)) "3 * x^^^2 = 1";
val t =
Const ("op =", "[RealDef.real, RealDef.real] => bool") $
(Const ("op *", "[RealDef.real, RealDef.real] => RealDef.real") $
Free ("3", "RealDef.real") $
(Const
("Atools.pow",
"[RealDef.real, RealDef.real] => RealDef.real") $
Free ("x", "RealDef.real") $ Free ("2", " RealDef.real"))) $
Free ("1", "RealDef.real") : Term.term
```

Nun wird ein Modell erstellt, das sich nicht auf bestimmte Zahlen bezieht, sondern nur eine generelle Zerlegung durchführt.

```
> val p = (term_of o the o (parse thy)) "a * b^^^2 = c";
val p =
Const ("op =", "[RealDef.real, RealDef.real] => bool") $
(Const ("op *", "[RealDef.real, RealDef.real] => RealDef.real") $
Free ("a", "RealDef.real") $
(Const
("Atools.pow",
"[RealDef.real, RealDef.real] => RealDef.real") $
Free ("b", "RealDef.real") $ Free ("2", "RealDef.real"))) $
Free ("c", "RealDef.real") : Term.term
```

Dieses Modell enthält sogenannte *scheme variables*.

```
> atomt p;
"*** -------------"
"*** Const (op =)"
"*** . Const (op *)""*** . . Free (a, )"
"*** . . Const (Atools.pow)"
"*** . . . Free (b, )"
"*** . . . Free (2, )"
"*** . Free (c, )"
"\n"
val it = "\n" : string
```

Das Modell wird durch den Befehl *free2var* erstellt.

```
> free2var;
val it = fn : Term.term -> Term.term
> val pat = free2var p;
val pat =
Const ("op =", "[RealDef.real, RealDef.real] => bool") $
(Const ("op *", "[RealDef.real, RealDef.real] => RealDef.real") $
Var (("a", 0), "RealDef.real") $
(Const
("Atools.pow",
"[RealDef.real, RealDef.real] => RealDef.real") $
Var (("b", 0), "RealDef.real") $
Free ("2", "RealDef.real"))) $ Var (("c", 0), "RealDef.real")
: Term.term
> Sign.string_of_term (sign_of thy) pat;
val it = "?a * ?b ^^^ 2 = ?c" : string
```

Durch *atomt pat* wird der Term aufgespalten und in eine Form gebracht, die für die weiteren Schritte benötigt wird.

```
> atomt pat;
"*** -------------"
"*** Const (op =)"
"*** . Const (op *)"
"*** . . Var ((a, 0), )"
"*** . . Const (Atools.pow)"
"*** . . . Var ((b, 0), )"
"*** . . . Free (2, )"
"*** . Var ((c, 0), )"
"\n"
val it = "\n" : string
```

Jetzt kann das Matching an den beiden vorigen Terme angewendet werden.

```
> matches thy t pat;
val it = true : bool
> val t2 = (term_of o the o (parse thy)) "x^^^2 = 1";
val t2 =
Const ("op =", "[RealDef.real, RealDef.real] => bool") $
(Const
("Atools.pow",
"[RealDef.real, RealDef.real] => RealDef.real") $
```

```
Free ("x", "RealDef.real") $ Free ("2", "RealDef.real")) $
Free ("1", "RealDef.real") : Term.term
> matches thy t2 pat;
val it = false : bool
> val pat2 = (term_of o the o (parse thy)) "?u^^^2 = ?v";
val pat2 =
Const ("op =", "[RealDef.real, RealDef.real] => bool") $
(Const
("Atools.pow",
"[RealDef.real, RealDef.real] => RealDef.real") $
Var (("u", 0), "RealDef.real") $ Free ("2", "RealDef.real")) $
Var (("v", 0), "RealDef.real") : Term.term
> matches thy t2 pat2;
val it = true : bool
```

5.2 Zugriff auf die hierachy

Man verwendet folgenden Befehl, um sich Zugang zur hierachy von problem type zu verschaffen.

```
> show_ptyps;
val it = fn : unit -> unit
> show_ptyps();
[
["e_pblID"],
["simplification", "polynomial"],
["simplification", "rational"],
["vereinfachen", "polynom", "plus_minus"],
["vereinfachen", "polynom", "klammer"],
["vereinfachen", "polynom", "binom_klammer"],
["probe", "polynom"],
["probe", "bruch"],
["equation", "univariate", "linear"],
["equation", "univariate", "root", "sq", "rat"],
["equation", "univariate", "root", "normalize"],
["equation", "univariate", "rational"],
["equation", "univariate", "polynomial", "degree_0"],
["equation", "univariate", "polynomial", "degree_1"],
["equation", "univariate", "polynomial", "degree_2", "sq_only"],
["equation", "univariate", "polynomial", "
 degree_2", "bdv_only"],
["equation", "univariate", "polynomial", "degree_2", "pqFormula"],
["equation", "univariate", "polynomial", "degree_2", "abcFormula"],
["equation", "univariate", "polynomial", "degree_3"],
["equation", "univariate", "polynomial", "degree_4"],
["equation", "univariate", "polynomial", "normalize"],
["equation", "univariate", "expanded", "degree_2"],
["equation", "makeFunctionTo"],
["function", "derivative_of", "named"],
["function", "maximum_of", "on_interval"],
["function", "make", "by_explicit"],
["function", "make", "by_new_variable"],
["function", "integrate", "named"],
```

```
["tool", "find_values"],
["system", "linear", "2x2", "triangular"],
["system", "linear", "2x2", "normalize"],
["system", "linear", "3x3"],
["system", "linear", "4x4", "triangular"],
["system", "linear", "4x4", "normalize"],
["Biegelinien", "
MomentBestimmte"],
["Biegelinien", "MomentGegebene"],
["Biegelinien", "einfache"],
["Biegelinien", "QuerkraftUndMomentBestimmte"],
["Biegelinien", "vonBelastungZu"],
["Biegelinien", "setzeRandbedingungen"],
["Berechnung", "numerischSymbolische"],
["test", "equation", "univariate", "linear"],
["test", "equation", "univariate", "plain_square"],
["test", "equation", "univariate", "polynomial", "degree_two", "pq_formula"],
["test", "equation", "univariate", "polynomial", "degree_two", "abc_formula"],
["test", "equation", "univariate", "squareroot"],
["test", "equation", "univariate", "normalize"],
["test", "equation", "univariate", "sqroot-test"]
]
val it = () : unit
```

5.3 Die passende "formalization" für den problem type

Eine andere Art des Matching ist es die richtige "formalization" zum jeweiligen problem type zu finden. Wenn eine solche vorhanden ist, kann ISAC selbstständig die Probleme lösen.

5.4 "problem-refinement"

Will man die problem hierachy (=) aufstellen, so ist darauf zu achten, dass man die verschiedenen Branches so konstruiert, dass das problem-refinement automatisch durchgeführt werden kann.

```
> refine;
val it = fn : fmz_ -> pblID -> SpecifyTools.match list
> val fmz = ["equality (sqrt(9 + 4 * x)=sqrt x
+ sqrt (5 + x))",
# "soleFor x","errorBound (eps=0)",
# "solutions L"];
val fmz =
["equality (sqrt(9 + 4 * x)=sqrt x + sqrt (5 + x))", "soleFor x",
"errorBound (eps=0)", ...] : string list
> refine fmz ["univariate","equation"];
*** pass ["equation","univariate"]
*** comp_dts: ??.empty $ soleFor x
Exception- ERROR raised
```

Wenn die ersten zwei Regeln nicht angewendet werden können, kommt die dritte zum Einsatz:

```
> val fmz = ["equality (x + 1 = 2)",
# "solveFor x","errorBound (eps=0)",
# "solutions L"];
val fmz = ["equality (x + 1 = 2)", "solveFor x", "errorBound (eps=0)", ...]
: string list
> refine fmz ["univariate","equation"];
*** pass ["equation","univariate"]
*** pass ["equation","univariate","linear"]
*** pass ["equation","univariate","root"]
*** pass ["equation","univariate","rational"]
*** pass ["equation","univariate","polynomial" ]
*** pass ["equation","univariate","polynomial","degree_0"]
*** pass ["equation","univariate","polynomial","degree_1"]
*** pass ["equation","univariate","polynomial","degree_2"]
*** pass ["equation","univariate","polynomial","degree_3"]
*** pass ["equation","univariate","polynomial","degree_4"]
*** pass ["equation","univariate","polynomial","normalize"]
val it =
[Matches
(["univariate", "equation"],
{Find = [Correct "solutions L"], With = [...], ...}),
NoMatch (["linear", "univariate", ...], {Find = [...], ...}),
NoMatch (["root", ...], ...), ...] : SpecifyTools.match list
```

Der problem type wandelt $x + 1 = 2$ in die normale Form $-1 + x = 0$ um. Diese Suche nach der jeweiligen problem hierachy kann mit Hilfe eines "proof state" durchgeführt werden (siehe nächstes Kapitel).

Chapter 6

"Methods“

Methods werden dazu verwendet, Probleme von type zu lösen. Sie sind in einer anderen Programmiersprache beschrieben. Die Sprache sieht einfach aus, betreibt aber im Hintergrund einen enormen Prüfaufwand. So muss sich der Programmierer nicht mit technischen Details befassen, gleichzeitig können aber auch keine falschen Anweisungen eingegeben werden.

6.1 Der "Syntax“ des script

Syntax beschreibt den Zusammenhang der einzelnen Zeichen und Zeichenfolgen mit den Theorien. Er kann so definiert werden:

script ::= `Script` id arg* = body
arg ::= id | ((id :: type))
body ::= expr
expr ::= `let` id = expr (; id = expr)* `in` expr
| `if` prop `then` expr `else` expr
| listexpr
| id
| seqex id
seqex ::= `While` prop `Do` seqex
| `Repeat` seqex
| `Try` seqex
| seqex `Or` seqex
| seqex `@@` seqex
| tac (id | listexpr)*
type ::= id
tac ::= id

6.2 Überprüfung der Auswertung

Das Kontrollsystem arbeitet mit den folgenden Script-Ausdrücken, die *tacticals* genannt werden:

`while` prop `Do` expr id

`if` prop `then` expr `else` expr

Während die genannten Befehle das Kontrollsystem durch Auswertung der Formeln auslösen, hängen die anderen von der Anwendbarkeit der Formel in den entsprechenden Unterbegriffen ab:

`Repeat` expr id

`Try` expr id

expr `Or` expr id

expr `@@` expr id

xxx

Chapter 7

Befehle von *ISAC*

In diesem Kapitel werden alle schon zur Verfügung stehenden Schritte aufgelistet. Diese Liste kann sich auf Grund von weiteren Entwicklungen von *ISAC* noch ändern.

Init_Proof_Hid (dialogmode, formalization, specifictaion) gibt die eingegebenen Befehle an die mathematic engine weiter, wobei die beiden letzten Begriffe die Beispiele automatisch speichern. Es ist nicht vorgesehen, dass der Schüler tactic verwendet.

Init_Proof bildet mit einem "proof tree" ein leeres Modell.

Model_Problem problem bestimmt ein problemtype, das womöglich in der "hierachy" gefunden wurde, und verwendet es für das Umformen.

Add_Given, Add_Find, Add_Relation formula fügt eine Formel in ein bestimmtes Feld eines Modells ein. Dies ist notwendig, solange noch kein Objekt für den Benutzer vorhanden ist, in dem man die Formel eingeben kann, und nicht die gewünschte tactic und Formel von einer Liste wählen will.

Specify_Theorie theory, Specify_Problem proble, Specify_Method method gibt das entsprechende Element des Basiswissens an.

Refine_Problem problem sucht nach einem Problem in der hierachy, das auf das vorhandene zutrifft.

Apply_Method method beendet das Modell und die Beschreibung. Danach wird die Lösungmeldung geöffnet.

Free_Solve beginnt eine Lösungsmeldung ohne die Hilfe einer method.

Rewrite theorem befördert ein theorem in die aktuelle Formel und wandelt es demenetsprechend um. Wenn dies nicht möglich ist, kommt eine Meldung mit "error".

Rewrite_Asm theorem hat die gleiche Funktion wie Rewrite, speichert jedoch eine endgültige Vorraussetzung des theorems, anstatt diese zu schätzen.

Rewrite_Set ruleset hat ähnliche Funktionen wie Rewrite, gilt aber für einen ganzen Satz von theorems, dem rule set.

Rewrite_Inst (substitution, theorem), Rewrite_Set_Inst (substitution, rule set) ist vergleichbar mit besonderen tactics, ersetzt aber Konstanten im theorem, bevor es zu einer Anwendung kommt.

Calculate operation berechnet das Ergebnis der Eingabe mit der aktuellen Formel (plus, minus, times, cancel, pow, sqrt).

Substitute substitution fügt der momentanen Formel `substitution` hinzu und wandelt es um.

Take formula startet eine neue Reihe von Rechnungen in den Formeln, wo sich schon eine andere Rechnung befindet.

Subproblem (theory, problem) beginnt ein subproblem innerhalb einer Rechnung.

Function formula ruft eine Funktion auf, in der der Name in der Formel enthalten ist. ???????

Split_And, Conclude_And, Split_Or, Conclude_Or, Begin_Trans, End_Trans, Begin_Sequ, End_Sequ, Split_Intersect, End_Intersect betreffen den Bau einzelner branches des proof trees. Normalerweise werden sie vom dialog guide verdrängt.

Check_elementwise assumption wird in Bezug auf die aktuelle Formel verwendet, die Elemente in einer Liste enthält.

Or_to_List wandelt eine Verbindung von Gleichungen in eine Liste von Gleichungen um.

Check_postcond überprüft die momentane Formel im Bezug auf die Nachbedinung beim Beenden des subproblem.

End_Proof beendet eine Überprüfung und gibt erst dann ein Ergebnis aus, wenn Check_postcond erfolgreich abgeschlossen wurde.

7.1 Die Funktionsweise der mathematic engine

Ein proof (= Beweis) wird in der mathematic engine me von der tactic `Init_Proof` gestartet und wird wechselwirkend mit anderen tactics vorangebracht. Auf den input (= das, was eingegeben wurde) einzelner tactics folgt eine Formel, die von der me ausgegeben wird, und die darauf folgende tactic gilt. Der proof ist beendet, sobald die me `End_Proof` als nächste tactic vorschlägt.
Im Anschluss werden Sie einen Rechenbeweis sehen, der von der Lösung einer Gleichung (= equation) handelt, bei der diese automatisch differenziert wird.

```
????????????????????????????????????????????????????????????????????????????????????????????

ML> val fmz = ["equality ((x+#1)*(x+#2)=x^^^#2+#8)","solveFor x",
                  "errorBound (eps=#0)","solutions L"];
   val fmz =
     ["equality ((x+#1)*(x+#2)=x^^^#2+#8)","solveFor x","errorBound (eps=#0)",
      "solutions L"] : string list
   ML>
   ML> val spec as (dom, pbt, met) = ("SqRoot.thy",["univariate","equation"],
                                ("SqRoot.thy","no_met"));
   val dom = "SqRoot.thy" : string
   val pbt = ["univariate","equation"] : string list
   val met = ("SqRoot.thy","no_met") : string * string
```

7.2 Der Beginn einer Rechnung

Der proof state wird von einem proof tree und einer position ausgegeben. Beide sind zu Beginn leer. Die tactic `Init_Proof` ist, wie alle anderen tactics auch, an einen string gekoppelt. Um einen neuen proof beginnen zu können, werden folgende Schritte durchgeführt:

```
????????????????????????????????????????????????????????????????????????????????????????????
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",[#,#],(#,#))) : mstep
   ML>
   ML>  val (p,_,f,nxt,_,pt) = me (mID,m) e_pos' c EmptyPtree;
   val p = ([],Pbl) : pos'
   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
```

Die mathematics engine gibt etwas mit dem type `mout` aus, was in unserem Fall ein Problem darstellt. Sobald mehr angezeigt wird, müsste dieses jedoch gelöst sein.

```
????????????????????????????????????????????????????????????????????????????????????????????????????????????
   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;
```

7.3 The phase of modeling

Dieses Kapitel beschäftigt sich mit dem input der Einzelheiten bei einem Problem. Die me kann dabei helfen, wenn man die formalization durch `Init_Proof` darauf hinweist. Normalerweise weiss die mathematics engine die nächste gute tactic.

```
????????????????????????????????????????????????????????????????????????????????????????????????????????????
   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,Nundef,(#,#))) : mout
```

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

7.4 The phase of specification

Diese phase liefert eindeutige Bestimmungen einer domain, den problem type und die method damit man sie verwenden kann. Für gewöhnlich wird die Suche nach dem richtigen problem type unterstützt. Dazu sind zwei tactics verwendbar: `Specify_Problem` entwickelt ein Feedback, wie ein problem type bei dem jetzigen problem zusammenpasst und `Refine_Problem` stellt Hilfe durch das System bereit, falls der Benutzer die Übersicht verliert.

```
????????????????????????????????????????????????????????????????????????????????????????????????????
   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
```

Die me erkennt den richtigen Problem type und arbeitet so weiter:

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

```
        (Problem ["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
```

Wir nehmen wieder an, dass der dialog guide die nächsten tactics, veranlasst von der mathematic engine, versteckt und der Schüler Hilfe benötigt. Dann muss `Refine_Problem` angewandt werden. Dieser Befehl findet immer den richtigen Weg, wenn man es auf den problem type bezieht [”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>
   ML>  Compiler.Control.Print.printDepth:=9;f;Compiler.Control.Print.printDepth:=4;
   val f =
     Problems
       (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>
   ML>
   ML>  Compiler.Control.Print.printDepth:=9;f;Compiler.Control.Print.printDepth:=4;
   val f =
     Problems
       (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=[]}),
           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=[]}),
NoMatch
  ...
  ...
Matches
  (["normalize","univariate","equation"],
   {Find=[Correct "solutions L"],
    Given=[Correct "equality ((x + #1) * (x + #2) = x ^^^ #2 + #8)",
           Correct "solveFor x"],Relate=[],Where=[],With=[]})]) : mout
```

Die tactic `Refine_Problem` wandelt alle matches wieder in problem types um und sucht in der problem hierachy weiter.

7.5 The phase of solving

Diese phase beginnt mit dem Aufruf einer method, die eine normale form innerhalb einer tactic ausführt: `Rewrite rnorm_equation_add` und `Rewrite_Set SqRoot_simplify`:

```
ML> nxt;
val it = ("Apply_Method",Apply_Method ("SqRoot.thy","norm_univar_equation"))
  : string * mstep
ML>
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>
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>
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",[#,#])) : string * mstep
```

Die Formel $-6+3\cdot x = 0$ ist die Eingabe eine subproblems, das wiederum gebraucht wird, um die Gleichungsart zu erkennen und die entsprechende method auszuführen:

```
ML> nxt;
val it = ("Subproblem",Subproblem ("SqRoot.thy",["univariate","equation"]))
ML>
ML> val (p,_,f,nxt,_,pt) = me nxt p [1] pt;
val f =
```

```
  Form' (FormKF
       (~1,EdUndef,1,Nundef,"Subproblem (SqRoot.thy, [univariate, equation])"))
  : mout
val nxt = ("Refine_Tacitly",Refine_Tacitly ["univariate","equation"])
ML>
ML> val (p,_,f,nxt,_,pt) = me nxt p [1] pt;
val nxt = ("Model_Problem",Model_Problem ["linear","univariate","equation"])
```

`Refine [''univariate'', ''equation'']` sucht die passende Gleichungsart aus der problem hierachy heraus, welche man mit `Model_Problem [''linear'', ''univariate'', ''equation'']` über das System ansehen kann. Nun folgt erneut die phase of modeling und die phase of specification.

7.6 The final phase: Überprüfung der "post-condition"

Die gezeigten problems, die durch *ISAC* gelöst wurden, sind so genannte 'example construction problems'. Das massivste Merkmal solcher problems ist die post-condition. Im Umgang mit dieser gibt es noch offene Fragen. Dadurch wird die post-condition im folgenden Beispiel als problem und subproblem erwähnt.

```
ML> nxt;
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 f = Form' (FormKF (~1,EdUndef,0,Nundef,"[x = #2]")) : mout
val nxt = ("End_Proof'",End_Proof') : string * mstep
```

Die tactic `End_Proof'` bedeutet, dass der proof erflogreich beendet wurde.

Versuchen Sie es! Die tactics, die vom System vorgeschlagen werden, müssen vom Benutzer nicht angewendet werden. Er kann selbstverständlich auch andere tactics verwenden und das System wird melden, ob dieser Befehl zutreffend ist oder nicht.

Part II

Handbuch für Autoren

Chapter 8

Die Struktur des Grundlagenwissens

8.1 "tactics" und Daten

Zuerst betrachten wir die me von aussen. Wir sehen uns tactics und an und verbinden sie mit unserem Grundwissen (KB). Im Bezug auf das KB befassen wir uns mit den kleinsten Teilchen, die von den Autoren des KB sehr genau durchgeführt werden müssen. Diese Teile sind in alphabetischer Anordnung in Tab.8.1 auf Seite 37 aufgelistet.

Die Verbindung zwischen tactics und Daten werden in Tab.8.2 auf Seite 38 dargestellt.

8.2 Die theories von *ISAC*

Die theories von *ISAC* basieren auf den theories für HOL und Real von Isabelle. Diese theories haben eine spezielle Form, die durch die Endung `*.thy` gekennzeichnet sind; normalerweise werden diese theories zusammen mit SML verwendet. Dann haben sie den selben Dateinamen, aber die Endung `*.ML`. Die theories von *ISAC* representieren den Teil vom Basiswissen von *ISAC*, die hierachy von den zwei theories ist nach diesen strukturiert. Die `*.ML` Dateien beinhalten *alle* Daten von den anderen zwei Hauptlinien des Basiswissens, die problems und methods (ohne ihre jeweilige Struktur, die von den problem Browsern und den method Browsern gemacht wird, zu präsentieren. Die Tab.8.3 auf Seite 39 listet die base theories auf, die geplant sind in der Version *ISAC* 1 angewendet zu werden. Wir erwarten, dass die Liste erweitert wird in näherer Zukunft, und wir werden uns auch den theorie Browser genauer ansehen. Die ersten drei theories auf der Liste gehören *nicht* zum Grundwissen von *ISAC*; sie beschäftigen sich mit der Skriptsprache für methods und ist hier nur zur Vollständigkeit angeführt.

8.3 Daten in `*.thy` und `*.ML`

Wie schon zuvor angesprochen, haben die Arbeiten die theories von *.thy und *.ML zusammen und haben deswegen den selben Dateiname. Wie diese Daten zwischen den zwei Dateien verteilt werden wird in der Tab.8.4 auf Seite 40 gezeigt. Die Ordnung von den Datenteilchen in den theories sollte an der Ordnung von der Liste festhalten.

8.4 Formale Beschreibung der Hierarchie von Problemen

8.5 Skripttaktiken

Tatsächlich sind es die tactics, die die Berechnungen vorantreiben: im Hintergrund bauen sie den proof tree und sie übernehmen die wichtigsten Aufgaben während der Auswertung bei der der "script-interpreter" zur Steuerung des Benutzers transferiert wird. Hier beschreiben wir nur den Syntax von tactics; die Semantik ist beschrieben etwas weiter unten im Kontext mit tactics, die die Benutzer/Innen dieses Programmes verwenden: Es gibt einen Schriftverkehr zwischen den user-tactics und den script tactics.

Table 8.1: Kleinste Teilchen des KB

Abkürzung	Beschreibung
calc_list	gesammelte Liste von allen ausgewerteten Funktionen
eval_fn	ausgewertete Funktionen für Zahlen und für Eigenschaften, die in SML kodiert sind
eval_rls	rule set *rls* für einfache Ausdrücke mit *eval_fn*s
fmz	Formalisierung, d.h. eine sehr geringe Darstellung von einem Beispiel
met	eine method d.h. eine Datenstruktur, die alle Informationen zum Lösen einer phase enthält (*rew_ord*, *scr*, etc.)
metID	bezieht sich auf *met*
op	ein Operator, der der Schlüssel zu *eval_fn* in einer *calc_list* ist
pbl	Problem d.h. der Knotenpunkt in der problem hierachy
pblID	bezieht sich auf *pbl*
rew_ord	Anordnung beim Rewriting
rls	rule set, d.h. eine Datenstruktur, die theorems *thm* und Operatoren *op* zur Vereinfachung (mit *rew_ord*) enthält
Rrls	rule set für das 'reverse rewriting' (eine ISAC-Technik, die schrittweise Rewriting entwickelt, z.B. für die zurückgenommenen Teile)
scr	script, das die Algorithmen durch Anwenden von tactics beschreibt und ein Teil von *met* ist
norm_rls	spezielles Regelwerk zum Berechnen von Normalformen, im Zusammenhang mit *thy*
spec	Spezifikation, z.B, ein Tripel (*thyID, pblID, metID*)
subs	Ersatz, z.B. eine Liste von Variablen und ihren jeweiligen Werten
Term	Term von Isabelle, z.B. eine Formel
thm	theorem
thy	theory
thyID	im Bezug auf *thy*

Table 8.2: Welche tactics verwenden die Teile des KB ?

tactic	Eingabe	norm_			rew_ rls			eval_	eval_	calc_	
		thy	scr	Rrls	rls	thm	ord	Rrls	fn	rls	list	dsc
Init_Proof	fmz	x			x							x
	spec											
model phase												
Add_*	Term	x			x							x
FormFK	model	x			x							x
specify phase												
Specify_Theory	thyID	x			x				x	x		x
Specify_Problem	pblID	x			x				x	x		x
Refine_Problem	pblID	x			x				x	x		x
Specify_Method	metID	x			x				x	x		x
Apply_Method	metID	x	x		x				x	x		x
solve phase												
Rewrite,_Inst	thm	x	x			x	met		x	met		
Rewrite, Detail	thm	x	x			x	rls		x	rls		
Rewrite, Detail	thm	x	x			x	Rrls		x	Rrls		
Rewrite_Set,_Inst	rls	x	x					x	x	x		
Calculate	op	x	x								x	
Substitute	subs	x			x							
SubProblem	spec	x	x		x				x	x		x
	fmz											

Table 8.3: theory von der ersten Version von ISAC

theory	Beschreibung
ListI.thy	ordnet die Bezeichnungen den Funktionen, die in `Isabelle2002/src/HOL/List.thy` sind, zu und (intermediatly ?) definiert einige weitere Listen von Funktionen
ListI.ML	`eval_fn` für die zusätzliche Listen von Funktionen
Tools.thy	Funktion, die für die Auswertung von Skripten benötigt wird
Tools.ML	bezieht sich auf `eval_fn`s
Script.thy	Vorraussetzung für script: types, tactics, tacticals
Script.ML	eine Reihe von tactics und Funktionen für den internen Gebrauch
Typefix.thy	fortgeschrittener Austritt, um den type Fehlern zu entkommen
Descript.thy	*Beschreibungen* für die Formeln von *Modellen* und *Problemen*
Atools	Neudefinierung von Operatoren; allgemeine Eigenschaften und Funktionen für Vorraussetzungen; theorems für `eval_rls`
Float	Gleitkommerzahlendarstellung
Equation	grundsätzliche Vorstellung für Gleichungen und Gleichungssysteme
Poly	Polynome
PolyEq	polynomiale Gleichungen und Gleichungssysteme
Rational.thy	zusätzliche theorems für Rationale Zahlen
Rational.ML	abbrechen, hinzufügen und vereinfachen von Rationalen Zahlen durch Verwenden von (einer allgemeineren Form von) Euclids Algorithmus; die entsprechenden umgekehrten Regelsätze
RatEq	Gleichung mit rationalen Zahlen
Root	Radikanten; berechnen der Normalform; das betreffende umgekehrte Regelwerk
RootEq	Gleichungen mit Wurzeln
RatRootEq	Gleichungen mit rationalen Zahlen und Wurzeln (z.B. mit Termen, die beide Vorgänge enthalten)
Vect	Vektoren Analysis
Trig	Trigonometrie
LogExp	Logarithmus und Exponentialfunktionen
Calculus	nicht der Norm entsprechende Analysis
Diff	Differenzierung 39
DiffApp	Anwendungen beim Differenzieren (Maximum-Minimum-Probleme)
Test	(alte) Daten für Testfolgen
Isac	enthält alle Theorien von ISAC

Table 8.4: Daten in `*.thy`- und `*.ML`-files

Datei	Daten	Beschreibung
`*.thy`	consts	Operatoren, Eigenschaften, Funktionen und Skriptnamen ('`Skript` Name ... `Argumente`')
	rules	theorems: *ISAC* verwendet theorems von Isabelle, wenn möglich; zusätzliche theorems, die jenen von Isabelle entsprechen, bekommen ein *I* angehängt
`*.ML`	`theory' :=`	Die theory, die abgegrenzt ist von der `*.thy`-Datei, wird durch *ISAC* zugänglich gemacht
	`eval_fn`	die Auswertungsfunktion für die Operatoren und Eigenschaften, kodiert im meta-Level (SML); die Bezeichnugn von so einer Funktion ist eine Kombination von Schlüsselwörtern `eval_` und einer Bezeichnung von der Funktion, die in in `*.thy` erklärt ist
	`*_simplify`	der automatisierte Vereinfacher für die tatsächliche Theorie, z.B. die Bezeichnung von diesem Regelwerk ist eine Kombination aus den Theorienbezeichnungen und dem Schlüsselwort `*_simplify`
	`norm_rls :=`	der automatisierte Vereinfacher `*_simplify` wird so aufgehoben, dass er über *ISAC* zugänglich ist
	`rew_ord' :=`	das Gleiche für die Anordnung des Rewriting, wenn es ausserhalb eines speziellen Regelwerks gebraucht wird
	`ruleset' :=`	dasselbe wie für Regelsätze (gewöhnliche Regelsätze, umgekehrte Regelsätze, und `eval_rls`)
	`calc_list :=`	dasselbe für `eval_fn`s, wenn es ausserhalb eines bestimmten Regelwerks gebraucht wird (wenn es ausserhalb eines bestimmten Regelwerks benötigt wird) (z.B. für eine tactic `Calculate` in einem Skript)
	`store_pbl`	Problems, die in `*.ML`-Dateien definiert sind, werden zugänglich für *ISAC*
	`methods :=`	methods, die in `*.ML`-Dateien definiert sind werden zugänglich für *ISAC*

Part III

Authoring on the knowledge

8.6 Add a theorem

8.7 Define and add a problem

8.8 Define and add a predicate

8.9 Define and add a method

8.10

8.11

8.12

8.13