Kapitel 4

Automatisierung des formalen Schließens

Im vorhergehenden Kapitel haben wir die intuitionistische Typentheorie schrittweise aus dem einfach getypten λ-Kalkül heraus entwickelt und damit einen mächtigen Formalismus aufgestellt, mit dem wir formale Schlüsse über alle Aspekte der Mathematik und Programmierung ziehen können. Diese Theorie liefert uns alles, was wir brauchen, um formale Beweise "von Hand" zu entwerfen. Ihr *praktischer* Nutzen liegt vor allem darin, daß durch die Formulierung von Inferenzregeln gezeigt wird, wie semantisches Schließen und die Verwendung mathematischer Erkenntnisse auf eine rein syntaktische Manipulation von Texten (Sequenzen) reduziert werden kann – also auf eine Aufgabe, für die der Einsatz von Computern geradezu prädestiniert ist.

Wir wollen uns in diesem Kapitel damit auseinandersetzen, auf welche Art eine sinnvolle maschinelle Unterstützung für die Entwicklung von Programmen und dem Beweis von Theoremen in der intuitionistischen Typentheorie geschaffen werden kann, also wie wir unserem ursprünglichen Ziel – der Automatisierung von Logik und Programmierung – durch den Entwurf von Beweissystemen für die Typentheorie näher kommen können. Welche grundsätzlichen Möglichkeiten bieten sich hierfür nun an?

Die elementarste Art der maschinellen Unterstützung für das formale Beweisen ist *Proof Checking* – die Überprüfung formal geführter Beweise durch einen Computer. Diese Vorgehensweise wurde erstmalig in voller Konsequenz innerhalb des AUTOMATH Projektes [Bruijn, 1980, van Benthem Jutting, 1977] verfolgt und bietet sich vor allem dann an, wenn eine sehr ausdrucksstarke formale Sprache zur Verfügung steht, die in der Lage ist, die für eine rigorose Formalisierung mathematischer Konzepte notwendigen Abstraktionen zu erfassen. Da Proof Checking sich nur auf ein Minimum algorithmischer Beweisführung stützt – nämlich nur auf die Kontrolle einer korrekten Regelanwendung, die in einem Beweisterm codiert ist – sind die entsprechenden Systeme sehr sicher und leicht zu programmieren. Für einen Benutzer sind sie allerdings nur sehr mühsam zu handhaben, da er praktisch alle formalen Informationen von Hand im Voraus bestimmen muß. Aufgrund der geringen maschinellen Unterstützung ist der *Verlustfaktor*[1] zwischen formalen und gewöhnlichen Beweisen extrem hoch, was formale Beweise sehr schwer zu lesen macht und den Umgang mit Proof Checkern wenig begeisternd erscheinen läßt.

Deutlich eleganter und genauso sicher und einfach zu programmieren sind Beweis*entwicklungs*systeme (*Proof editors*), bei denen die formalen Regeln des Kalküls nicht zur Überprüfung sondern zum Entwurf von Beweisen eingesetzt werden können. Der Benutzer wird hierbei davon entlastet, den Beweisterm im Voraus anzugeben. Stattdessen entwickelt er ihn in Kooperation mit dem System, indem er schrittweise die jeweils anzuwendende Regel angibt. Die Umwandlung der Regeln der einfachen Typentheorie in Regeln mit einer impliziten Darstellung der Beweisterme in Abschnitt 3.2.3.3 (Seite 117ff.) zielte genau auf diesen Vorteil ab. Auch hier ist das Maß an automatischer Unterstützung jedoch noch sehr gering, da ein Benutzer nur von dem Aufschreiben formaler Beweisterme entlastet wird, nicht aber davon, den gesamten Beweis selbst zu führen.

Das Gegenextrem zur Beweisüberprüfung bildet das *automatische Theorembeweisen*, das von verschiedenen Beweissystemen [Bledsoe, 1977, Bibel, 1987, Bibel *et.al.*, 1994, Bläsius *et.al.*, 1981, Boyer & Moore, 1979,

[1]Dieser Verlustfaktor drückt aus, um wieviel länger formale Beweise werden, wenn man sorgfältige 'normale' Beweise in die formale Sprache überträgt. In den meisten Proof Checkern liegt er zwischen 20 und 50. Bei der *Entwicklung* von Beweisen mit den 'reinen' Regeln der Typentheorie ist der Faktor geringer, aber immer noch zu hoch für ein praktisches Arbeiten.

Letz *et.al.*, 1992a, Letz *et.al.*, 1992b, Wos *et.al.*, 1984, Wos *et.al.*, 1990] angestrebt wird. Dieser Ansatz stützt sich auf die Tatsache, daß alle mit einem Kalkül beweisbaren Theoreme im Endeffekt durch eine vollständige Suche gefunden werden können.[2] Tatsächlich konnten durch automatische Theorembeweiser bereits eine Reihe bisher unbekannter Resultate nachgewiesen werden. Im allgemeinen ist Theorembeweisen jedoch sehr aufwendig, da die Gültigkeit mathematischer Sätze schon in der Prädikatenlogik erster Stufe nicht mehr entscheidbar ist. Die meisten Verfahren stützen sich daher auf maschinennahe Charakterisierungen der Gültigkeit von Sätzen der betrachteten Theorie und vor allem auf heuristische Suchstrategien, um auf diese Art die Effizienz der Suche zu steigern und eine größere Menge von Problemen in akzeptabler Zeit lösen zu können. Dies ist bisher jedoch nur für relativ "kleine" Theorien wie die Prädikatenlogik erster Stufe mit geringfügigen Erweiterungen möglich und macht eine Übertragung der Ergebnisse auf reichhaltigere Theorien nahezu unmöglich. Für die Typentheorie und andere formale Theorien mit einer ähnlich hohen Ausdruckskraft ist dieser Ansatz daher unbrauchbar.

Ein praktikabler Mittelweg zwischen reiner Beweisüberprüfung und heuristisch gesteuerten automatischen Beweisern ist, Beweisentwicklungssysteme um *Entscheidungsprozeduren* für gewisse einfache Teiltheorien zu erweitern. In derartigen Systemen beschränkt sich die automatische Unterstützung auf solche Probleme, die mit Hilfe von Algorithmen schnell erkannt und *entschieden* werden können. Dabei basieren die eingesetzten Entscheidungsprozeduren auf einer grundlegenden Analyse der Teiltheorie, für die sie eingesetzt werden, und auf einem komplexen, aber immer terminierenden Algorithmus, dessen Korrektheit nachgewiesen werden kann. Auf diese Art werden die Benutzer des Systems von vielen lästigen Teilaufgaben entlastet und die Zuverlässigkeit des Gesamtsystems bleibt gesichert.

Eine Möglichkeit die Stärken dieses Ansatzes mit denen der automatischen Beweiser zu verbindet, bildet das Konzept der Beweis*taktiken*, welches erstmals im LCF Project [Gordon *et.al.*, 1979] der University of Edinburgh untersucht wurde und sich im Laufe der Jahre als sehr leistungsfähig herausgestellt hat.[3] Die Schlüsselidee war dabei, ein flexibles System zum Experimentieren mit einer Vielfalt von Strategien zu entwickeln, indem einem Benutzer ermöglicht wird, den Inferenzmechanismus um eigene Methoden zu erweitern, ohne daß hierdurch die Sicherheit des Systems gefährdet werden kann. Dies kann dadurch geschehen, daß die Regeln des zugrundeliegenden Kalküls nach wie vor die einzige Möglichkeit zur Manipulation von Beweisen sind, aber ein Mechanismus geschaffen wird, ihre Anwendung durch (Meta-)Programme zu steuern. Dies verlangt allerdings eine Formalisierung der bis dahin nur informal vorliegenden Metasprache des Kalküls als eine interaktive Programmiersprache, in der alle objektsprachlichen Konzepte wie Terme, Sequenzen, Regeln und Beweise zu programmieren sind.

In diesem Kapitel wollen wir nun die grundsätzlichen Techniken vorstellen, die bei der Realisierung eines derartigen taktischen Theorembeweisers eine Rolle spielen können, und diese am Beispiel der konkreten Implementierung des NuPRL-Systems illustrieren.[4] Konkrete Inferenzmethoden zur Automatisierung der Beweisführung stehen in diesem Kapitel eher im Hintergrund, da es uns mehr um die Möglichkeiten dieser Techniken als solche geht. In Abschnitt 4.1 werden wir zunächst diskutieren, welche Grundbausteine notwendig oder sinnvoll sind um Systeme zu bauen, mit denen absolut korrekte Beweise interaktiv entwickelt werden können. In Abschnitt 4.2 werden wir zeigen, wie die Rechnerunterstützung bei der Beweisführung durch das Konzept der Taktiken gesteigert werden kann und in Abschnitt 4.3 werden wir am Beispiel zweier erfolgreicher Entscheidungsprozeduren des NuPRL Systems beschreiben, wie man Teiltheorien der Typentheorie vollautomatisch entscheiden kann.

[2]Das ist der aus der theoretischen Informatik bekannte Zusammenhang zwischen ***Beweisbarkeit*** und ***Aufzählbarkeit***.

[3] Das Taktik-Konzept wurde vor allem von Beweissystemen aufgegriffen, mit denen sehr komplexe Aufgaben gelöst werden sollten. Neben dem NuPRL-System, das wir hier detaillierter vorstellen werden, sind dies vor allem Cambridge LCF (Schließen über funktionale Programme) [Paulson, 1987], HOL (Logik höherer Ordnung) [Gordon., 1985, Gordon., 1987], λ-Prolog (Prolog mit einer Erweiterung um λ-Terme) [Felty & Miller, 1988, Felty & Miller, 1990], OYSTER (ein Planungssystem für Programmsynthese) [Bundy, 1989, Bundy *et.al.*, 1990], ISABELLE (ein universeller (generischer) Beweiser) [Paulson, 1989, Paulson, 1990] und KIV (ein Verifikationssystem für imperative Programme) [Heisel *et.al.*, 1988, Heisel *et.al.*, 1990], LEGO (ein Beweissystem für ECC) [Luo & Pollack, 1992, Pollack, 1994], ALF (ein Beweissystem für Martin-Löf's Typentheorie) [Altenkirch *et.al.*, 1994].

[4]Es sei angemerkt, daß die in diesem Kapitel besprochenen Methoden im Prinzip nicht von der Typentheorie abhängen sondern sich genauso mit anderen Objekttheorien realisieren lassen.

4.1 Grundbausteine interaktiver Beweisentwicklungssysteme

Im Gegensatz zu vollautomatischen Beweissystemen, bei denen die Effizienz der Beweisführung im Vordergrund steht, kommt es bei interaktiven Beweissystemen vor allem darauf an, Beweise und Programme in einer für Menschen verständlichen Form entwickeln zu können und in einer Art aufzuschreiben, wie dies auch in einem mathematischen Lehrbuch üblich ist. Das bedeutet, daß ein Benutzer in der Lage sein muß, Definitionen einzuführen, Theoreme aufzustellen und mit Hilfe des Systems zu beweisen (wobei dieses die Korrektheit des Beweises garantiert), Programme aus Beweisen zu extrahieren und auszuführen und seine Ergebnisse in einer "Bibliothek" zu sammeln, die praktisch einem Buch entspricht.

Um dies zu unterstützen, braucht ein praktisch nutzbares Beweisentwicklungssystem neben einer Implementierung der objektsprachlichen Konzepte eine Reihe von Mechanismen, welche eine Interaktion mit einem Benutzer unterstützen.[5]

- Eine *Bibliothek* (*library*), in der verschiedene vom Benutzer eingeführte Objekte wie Definitionen, Sätze, Kommentare etc. enthalten sind.
- Eine *Kommandoebene*, mit der Objekte der Bibliothek erzeugt, gelöscht oder anderweitig manipuliert werden können (dies schließt den Aufruf geeigneter Editoren ein) und andere Interaktionen mit dem System – wie das Laden, Sichern oder Aufbereiten von Bibliotheken – gesteuert werden können.
- Ein *Beweiseditor*, mit dem die Behauptungen eines Theorems aufgestellt und bewiesen werden können. Dieser hat vor allem die Aufgabe, die Korrektheit von Beweisen sicherzustellen und dem Anwender unnötige Schreibarbeit zu ersparen. Zu dem Beweiseditor gehört auch ein *Extraktionsmechanismus*, mit dem die implizit in einem Beweis enthaltenen Programme extrahiert werden können.
- Einen *Text- und Termeditor*, der die Erstellung syntaktisch korrekter Terme unterstützt.
- Einen *Definitionsmechanismus*, welcher konservative Erweiterungen der zugrundeliegenden Theorie mit einer flexiblen Darstellungsform unterstützt.
- Ein *Programmevaluator*, mit dem die generierten Programme auch innerhalb des Systems überprüft und ausgetestet werden können.

Im folgenden werden wir die wichtigsten Aspekte dieser Komponenten und einer Implementierung der objektsprachlichen Konzepte diskutieren. Zuvor werden wir kurz auf die Formalisierung der Metasprache eingehen, die wir für die Implementierungsarbeiten und eine Integration des Taktik-Konzeptes benötigen.

4.1.1 ML als formale Beschreibungssprache

Zur Beschreibung der Konzepte, die beim Aufbau eines formalen Kalküls eine Rolle spielen, hatten wir uns bisher einer halbformalen Metasprache bedient, die gemäß unserer Vereinbarung in Abschnitt 2.1.4 aus der natürlichen Sprache, Bestandteilen der Objektsprache und sogenannten syntaktischen Metavariablen bestand. Wenn wir nun beschreiben wollen, wie diese Konzepte innerhalb von Beweisunterstützungssystemen durch Algorithmen und Datenstrukturen zu realisieren sind, dann liegt es nahe, die Metasprache stärker zu formalisieren und als Ausgangspunkt einer Implementierung zu verwenden. Diese formale Metasprache hat den

[5] Prinzipiell könnte man auch ohne die hier genannten Konzepte auskommen und sich allein auf die Implementierung der Objektsprache, die Kommandoebene und ein Taktik-Konzept konzentrieren. Ein Verzicht auf den Beweiseditor hätte jedoch zur Folge, daß ein Benutzer seinen Beweis komplett von außen programmieren müßte. Der Verzicht auf einen flexiblen Definitionsmechanismus macht formale Theoreme nahezu unlesbar. Ohne den Termeditor müßte die Flexibilität der Darstellung von Termen auf dem Bildschirm begrenzt werden und ein Benutzer müßte sich die korrekte Syntax aller Terme merken. Aus diesem Grunde sollte man in einem praktisch verwendbaren System nicht darauf verzichten.

Vorteil, daß sie aus intuitiv verständlichen mathematischen Konzepten aufgebaut ist, die keiner ausführlichen Erklärung bedürfen, und dennoch formal genug ist, um als Programmiersprache verwendbar zu sein.

Da man im allgemeinen davon ausgehen kann, daß die Idee einer Funktion intuitiv klar ist, bietet es sich an, diese Metasprache aus bekannten einfachen Funktionen aufzubauen und diese um einfache Strukturierungskonzepte zu ergänzen. Dieser Gedanke wurde erstmals im Rahmen des LCF-Projects [Gordon *et.al.*, 1979] verfolgt und führte zur Entwicklung der formalen Metasprache ML (MetaLanguage), die ebenfalls bei der Implementierung von NuPRL eingesetzt wird.[6] Drei wichtige Charakteristika machen ML für diese Zwecke besonders geeignet.

- ML ist – wie der λ-Kalkül – eine funktionale Programmiersprache *höherer Stufe*: es gibt keine prinzipiellen Restriktionen an die Argumente einer Funktion.
- ML besitzt eine *erweiterbare und polymorphe Typdisziplin* mit sicheren (abstrakten) Datentypen. Als Kontrollinstrument dient eine erweiterte Form des Typechecking Algorithmus von Hindley und Milner.
- ML besitzt einen Mechanismus um *Ausnahmen* (*exceptions*) zu erzeugen und zu verarbeiten.

Da ML im wesentlichen die übliche mathematische Notation verwendet, wollen wir uns in diesem Abschnitt auf die zentralenen Grundkonstrukte und Besonderheiten von ML beschränken. Eine ausführlichere Beschreibung findet man in [Gordon *et.al.*, 1979], [Constable *et.al.*, 1986, Kapitel 6&9] und [Jackson, 1993a]. Um ML-Konstrukte von eventuell gleichlautenden Konstrukten der Objektsprache zu unterscheiden, werden wir sie `unterstrichen` darstellen.

4.1.1.1 Funktionen

Grundlage aller ML-Programme ist die Definition und Applikation von Funktionen. In ML werden diese entweder als *Abstraktion*

```
let divides = \x.\y.((x/y)*y = x);;
```

oder als definitorische Gleichung

```
let divides x y = ((x/y)*y = x);;
```

eingeführt. Beide Formen definieren die gleiche Funktion `divides`, welche eine ganze Zahl in eine Funktion von den ganzen Zahlen in Boole'sche Werte abbildet.[7] \ ist ein Abstraktionsoperator, der eine ASCII-Repräsentation des vertrauteren λ ist. Die definitorische Gleichung ist jedoch etwas allgemeiner als die Abstraktionsform, da sie auch benutzt werden kann, um rekursive Funktionen auf elegante Art einzuführen, wie in

```
letrec MIN f start = if f(start)=0 then start else MIN f (start+1).
```

Die Funktion `MIN` ist hierbei eine Funktion *höherer Ordnung*. Sie nimmt eine Funktion `f` $\in$ `int->int` als Argument und bildet sie in eine Funktion von den ganzen Zahlen in ganze Zahlen ab. In ML dürfen beliebige Funktionen als Argumente von anderen Funktionen vorkommen, solange sie typisierbar sind.

[6] Diese Sprache wurde später standardisiert und zu einer echten funktionalen Programmiersprache ausgebaut, die mittlerweile bei der Implementierung von Systemen, in denen Symbolverarbeitung eine wichtige Rolle spielt, weltweit Verbreitung gefunden hat und die zuvor dominierende Sprache LISP zu verdrängen beginnt. Die wichtigsten ML-Dialekte, die in der Praxis eingesetzt werden, sind CAML (Categorial Abstract Machine Language) [Cousineau & Huet, 1990, Mauny, 1991, Weis *et.al.*, 1990] und SML (Standard ML).

Funktionale Programmiersprachen haben gegenüber den imperativen Programmiersprachen den generellen Vorteil, daß der Programmieraufwand relativ gering ist, wenn man bereits eine präzise Beschreibung des Problems kennt. Was die Geschwindigkeit angeht, sind sie mittlerweile genauso effizient wie imperative Sprachen, solange nicht nur ständig einzelne Werte in komplexen Datenstrukturen verändert werden. In Kauf nehmen muß man allerdings einen relativ großen Speicherverbrauch, was in Anbetracht der heutigen Hardware allerdings kein Problem mehr ist.

[7] Die obige Gleichung wird vom ML-Interpreter zunächst auf ihre Typisierbarkeit überprüft. Ist eine Typisierung möglich, so wird die Funktion samt ihres Typs in die "Welt" von ML aufgenommen und es erscheint die Kontrollmeldung

```
divides = - :(int -> int -> bool)
```

Andernfalls erscheint eine Fehlermeldung und der Name `divides` gilt weiterhin als unbekannt, falls er zuvor unbekannt war.

Konstanten werden wie Funktionen durch eine definitorische Gleichung deklariert. Zwischen nullstelligen Funktionen (`let f () =` *ausdruck*) und Konstanten (`let f =` *ausdruck*) besteht jedoch ein Unterschied, da der Wert einer Konstanten zur Zeit der Deklaration berechnet wird, während ein Funktionskörper erst bei einer Applikation ausgewertet wird.[8]

4.1.1.2 Typen

In ML wird jedem Objekt, auch den Funktionen, ein *Typ* zugeordnet, zu dem es gehören soll. Dies ermöglicht es, Typeinschränkungen der Argumente und Ergebnisse von Funktionen auszudrücken und zu erzwingen. Die Basistypen von ML sind ganze Zahlen, Boole'sche Werte, Token und Strings (`int`, `bool`, `tok`, `string`) und ein einelementiger Datentyp `unit`. Token und Strings unterscheiden sich durch ihren Verwendungszweck und werden dadurch unterschieden, daß ein Token durch `‘`*token-quotes*`‘` umgeben wird. Komplexere Datentypen können hieraus durch die Typkonstruktoren `->`, `#`, `+` und `list` (Funktionenraum, Produkt, disjunkte Vereinigung und Listen) gebildet werden. Zugunsten einer automatischen Typisierbarkeit von ML-Ausdrücken sind abhängige Typkonstruktoren kein Bestandteil von ML.

Um eine höhere Flexibilität und Klarheit bei der Programmierung komplexerer Algorithmen zu erreichen, darf das Typsystem durch *anwenderdefinierte Datentypen* konservativ erweitert werden. Dies kann auf zwei Arten geschehen. Durch eine Deklaration

```
lettype intervals = int#int
```

wird einfach ein neuer Name für den Typ `int#int` eingeführt, der ab sofort als Abkürzung verwendet wird. Einen besonderen Unterschied zwischen `intervals` und `int#int` gibt es ansonsten nicht.

Darüber hinaus erlaubt ML aber auch die Deklaration *abstrakter Datentypen*, in denen die interne Darstellung der Elemente nach außen hin unsichtbar bleibt und Zugriffe nur über Funktionen möglich sind, die innerhalb der Deklaration des abstrakten Datentyps definiert wurden. Durch diese *Datenkapselung* kann man verhindern, daß Anwenderprogramme in unerwünschter Weise – zum Beispiel durch direkte Manipulation einer Komponente – auf die Daten zugreifen. Diese Eigenschaft ist besonders wichtig, wenn Systeme mit sensiblen Datenstrukturen wie Terme der Typentheorie oder Beweise programmiert werden sollen, die zugunsten einer Korrektheitsgarantie nur kontrollierte Veränderungen der Daten zulassen. Ein einfaches Beispiel für einen solchen abstrakten Datentyp ist der Datentyp `time`:

```
abstype time = int # int
 with maketime(hrs,mins) = if  hrs<0 or 23<hrs or mins<0 or 59<mins
                               then fail
                               else abs_time(hrs,mins)
 and hours t   = fst(rep_time t)
 and minutes t = snd(rep_time t);;
```

Diese Deklaration erklärt den Datentyp `time` zusammen mit drei Funktionen `maketime`, `hours` und `minutes`. Die Funktionen `abs_time` und `rep_time` sind nur innerhalb dieser Deklaration bekannt und stellen Konversionen von der expliziten zur abstrakten Repräsentation bzw. umgekehrt dar, die bei der Programmierung der mit `time` assoziierten Funktionen benötigt werden. Durch die abstrakte Deklaration wird sichergestellt, daß `time`-Objekte nur durch `maketime` verändert und nur durch `hours` und `minutes` analysiert werden können.

Typen dürfen auch *rekursiv* definiert werden, was unbedingt erforderlich ist, um Konstrukte wie Bäume, Graphen, Terme oder Beweise beschreiben zu können. Ebenso ist es möglich *generische* Datentypen zu erzeugen, also Datentypen, die einen *Typparameter* enthalten. So könnte man zum Beispiel Binärbäume über einem beliebigen Datentyp wie folgt deklarieren.

[8] Als ein Zugeständnis an die Effizienz bietet ML auch imperative Konzepte wie globale Variablen an. Diese sind explizit mit `letref ...` als solche zu deklarieren und dürfen dann Werte *zugewiesen* bekommen.


```
absrectype * bintree = * + (* bintree) # (* bintree)
  with mk_tree(s1,s2) = abs_bintree (inr(s1,s2) )
  and left s          = fst ( outr(rep_bintree s) )
  and right s         = snd ( outr(rep_bintree s) )
  and atomic s        = isl(rep_bintree s)
  and mk_atom a       = abs_bintree(inl a)
;;
```

Bei der Deklaration einer Funktion ist es normalerweise nicht erforderlich, den Datentyp der Argumente bzw. des Ergebnisses mit anzugeben, da ML jedem Term automatisch einen *Typ* zuordnet, sofern dies möglich ist. Hierzu wird eine erweiterte Form des *Typechecking Algorithmus* von Hindley und Milner (siehe Abschnitt 2.4.4 auf Seite 75 und [Hindley, 1969, Milner, 1978, Damas & Milner, 1982]) eingesetzt. Dabei kann der Typ einer Funktion auch *polymorph* sein, was bedeutet, daß in der Typisierung *Typvariablen* (üblicherweise `*`, `**`, `***` etc.) auftreten können, für bei einer Anwendung der Funktion beliebige konrekte Datentypen eingesetzt werden dürfen. So erhält zum Beispiel die Funktion

```
\x.x
```

den Datentyp `(* -> *)`, was besagt, daß der Ergebnistyp von `\x.x` identisch mit dem Typ des Argumentes sein muß, aber sonst keinerlei Beschränkungen existieren. `\x.x` kann somit als Identitätsfunktion auf beliebigen Datentypen eingesetzt werden. Ein weiteres Beispiel ist die oben deklarierte Funktion `mk_atom`, deren Datentyp `(* -> * bintree)` polymorph[9] ist, weil sie als Bestandteil eines generischen Datentyps deklariert wurde.

4.1.1.3 Vordefinierte Operationen

Die meisten der vordefinierten ML-Funktionen verwenden Bezeichnungen, die in der Mathematik geläufig sind. Auf `int` gibt es die üblichen Operationen `+,-,*,/,<,>,=`. Boole'sche Operationen sind `not, &, or`. Paare werden durch Kommata wie in `1,2` gebildet und durch `fst` und `snd` analysiert. Für die disjunkte Vereinigung verwendet man `inl, inr, outl, outr` und `isl` und für Listen `[], null, hd, tl` sowie die Punktnotation `a.liste`, um ein Element vor eine Liste zu hängen. Eine explizite Auflistung schreibt man in eckige Klammern durch Semikolon getrennt wie in `[1;2;3;4;5]`. Klammern sind einzusetzen, wenn die Eindeutigkeit es erfordert. Über diese Grundoperationen hinaus gibt es eine große Menge weiterer vordefinierter Funktionen. Für Details verweisen wir auf [Jackson, 1993a, Kapitel 6 & 7].

4.1.1.4 Abstraktionen

Um zu vermeiden, daß komplexe Teilausdrücke mehrmals explizit in einem Term genannt und ausgewertet werden müssen, kann man abkürzende Bezeichnungen einführen, die nur lokale Gültigkeit haben.

```
let x = 2*y*y+3*y+4 in x*x
```

bedeutet zum Beispiel, daß der Wert des Teilausdrucks `2*y*y+3*y+4` nur einmal bestimmt wird und dann alle Vorkommen von `x` im Ausdruck `x*x` durch diesen Wert ersetzt werden. Dieses Konstrukt kommt häufig innerhalb von Funktionsdeklarationen vor. Dabei dürfen durchaus auch (rekursive) Funktionen als Abkürzungen eingeführt werden wie zum Beispiel in

```
let upto from to   = letrec aux from to partial_list =
                          if to < from then partial_list
                                       else aux from (to-1) (to.partial_list)
                     in
                          aux from to []
                          ;;
```

[9] Man beachte, daß der Begriff der *Polymorphie* ("vielgestaltig") in der Informatik z.T. auch eine weitergehende Bedeutung bekommen hat, der im Zusammenhang mit den Konzepten Vererbung und dynamischem Binden der objektorientierten Programmierung steht.

Diese Funktion berechnet die Liste `[from;...;to]` indem sie diese schrittweise aus einer partiellen Liste aufbaut, die mit `[]` initialisiert wird.

Eine Besonderheit von ML ist, daß auf der linken Seite von Deklarationen und Abstraktionen auch zusammengesetzte Ausdrücke stehen dürfen. ML versucht dann, die Komponenten der linken Seite gegen den Wert der rechten Seite zu *matchen*[10] und die auf der linken Seite vorkommenden Variablen entsprechend zu belegen. Der Ausdruck auf der linken Seite darf aus Variablen, einer Dummy-Variablen `()`, und den *Konstruktoren* für Tupel (`,`) und Listen (`.` bzw. `[ ; ; ; ]`) aufgebaut sein. Dies erspart die Verwendung von Destruktoren wie `fst, snd, hd, tl` und ermöglicht sehr elegante Deklarationen wie zum Beispiel

```
let x.y.rest = upto 1 5 in x,y.
```

Hier wird `x` mit `1` und `y` mit `2` belegt und das Paar `1,2` zurückgegeben.

4.1.1.5 Ausnahmen

ML besitzt einen wohldurchdachten Mechanismus zur Behandlung von *Ausnahmesituationen* (*exceptions*). Einige Funktionen wie zum Beispiel die Division `/` oder die Funktion `hd` liefern bei der Eingabe bestimmter Argumente einen Laufzeitfehler (*failure*), da sie hierfür nicht sinnvoll definiert werden können. Ebenso können beim Matching in Deklarationen Fehler entstehen – zum Beispiel, wenn `let [x;y] = L in ...` ausgewertet werden soll, aber `L` die leere Liste ist. ML bietet nun die Möglichkeit an, derartige Ausnahmesituationen abzufangen (*failure catching*) und zu einem wohldefinierten Ende zu bringen. Auf diese Art kann ein unkontrollierter Abbruch des Programms innerhalb dessen der Fehler auftrat, vermieden werden.

Hierzu steht es ein spezieller Operator `?` zur Verfügung, der folgenden Effekt hat: ein Ausdruck e_1 `?` e_2 liefert als Ergebnis das Resultat der Auswertung von e_1, sofern diese keine Ausnahme erzeugt, und ansonsten das Ergebnis der Auswertung von e_2. So liefert zum Beispiel

```
2/2 ? 1000
```

den Wert `1`, während

```
2/0 ? 1000
```

den Wert `1000` liefert. Diesen Mechanismus kann man sich immer dann zunutze machen, wenn in einer Funktion eine andere Funktion benutzt wird, die einen Fehler erzeugen könnte. Durch die Deklaration

```
let divides x y = ((x/y)*y = x) ? false;;
```

wird zum Beispiel vermieden, daß `divides x 0` zu einem Fehler führt. Stattdessen wird das gewünschte Ergebnis `false` zurückgegeben.

Es ist auch möglich, Ausnahmen mit Hilfe des Ausdrucks `fail` gezielt zu erzeugen, um ein unerwünschtes Verhalten – besonders in rekursiven Funktionen – gezielt beenden zu können. Dadurch erspart man es sich, die Auswertung der Funktion zuende laufen lassen zu müssen und dabei ständig eine Fehlermeldung mitzuführen, die dann am Ende ausgegeben werden kann.

Der Ausnahmebehandlungsmechanismus hat gegenüber anderen Fehlerbehandlungsmöglichkeiten den Vorteil, daß man nicht von Anfang an alle Eingaben abfangen muß, die *möglicherweise* einen Fehler erzeugen. Er ist – sorgfältig eingesetzt – die effizienteste und eleganteste Art der Fehlerbehandlung, kann allerdings auch zu einem undurchsichtigen Programmierstil mißbraucht werden.

4.1.2 Implementierung der Objektsprache

Die Entwicklung der Programmiersprache ML als Formalisierung einer Metasprache, die bei der Beschreibung formaler Kalküle benutzt wurde, ermöglicht es, die Implementierung der objektsprachlichen Konzepte

[10] Im Deutschen gibt es hierfür kein einheitlich anerkanntes Wort. Manchmal wird der Begriff *mustern* verwendet.

unmittelbar an die in Abschnitt 3.2 gegebenen Definitionen von Termen, Sequenzen, Regeln und Beweisen anzulehnen. Wir müssen hierzu nur die informalen Definitionen in abstrakte Datentypen übertragen und dabei Funktionen für einen Zugriff auf Elemente dieser Datentypen einführen. Da wir in diesen Definitionen bereits eine strikte Trennung zwischen der allgemeinen Struktur von Termen und Regeln und den konkreten Bestandteilen der Typentheorie vorgenommen haben, erhalten wir eine sehr flexible und leicht zu wartende Implementierung, die auch für eine Realisierung anderer formaler Theorien verwendbar ist. Die Implementierung einer konkreten Theorie kann dann durch Einträge in separaten Tabellen (bzw. durch Objekte der Bibliothek) durchgeführt werden.

4.1.2.1 Terme

Der Datentyp `term` ist ein rekursiver abstrakter Datentyp, der die Definition 3.2.5 auf Seite 104 wiederspiegeln soll. Hierzu müssen wir Terme und gebundene Terme simultan definieren. Man beachte, daß sich die Darstellungsform von Termen und gebundenen Termen von ihrer internen Repräsentation unterscheidet.

```
abstype var  = tok
  with mkvar t = abs_var t
  and  dvar  v = rep_var v
;;
abstype level_exp  = tok + int
  with mk_var_level_exp t     = abs_level_exp (inl t)
  and  mk_const_level_exp i   = abs_level_exp (inr i)
  and  dest_var_level_exp l   = outl (rep_level_exp l)
  and  dest_const_level_exp l = outr (rep_level_exp l)

  and    ⋮
;;
abstype parm = int + tok + string + var + level_exp + bool
  with mk_int_parm i      = abs_parm (inl i)
  and  mk_tok_parm t      = abs_parm (inl (inr t))
  and  mk_string_parm s   = abs_parm (inl (inr (inr s)))
  and  mk_var_parm v      = abs_parm (inl (inr (inr (inr v))))
  and  mk_level_parm l    = abs_parm (inl (inr (inr (inr (inr l))))
  and  mk_bool_parm b     = abs_parm (inr (inr (inr (inr b))))
  and  dest_int_parm p    = outl (rep_parm p)
  and  dest_tok_parm p    = outl (outr (rep_parm p))
  and  dest_string_parm p = outl (outr (outr (rep_parm p)))
  and  dest_var_parm p    = outl (outr (outr (outr (rep_parm p))))
  and  dest_level_parm p  = outl (outr (outr (outr (outr (rep_parm p)))))
  and  dest_bool_parm p   = outr (outr (outr (outr (outr (rep_parm p)))))
;;
absrectype  term = (tok # parm list) # bterm list
and         bterm = var list # term
  with mk_term (opid,parms) bterms = abs_term((opid,parms),bterms)
  and  dest_term t                 = rep_term t
  and  mk_bterm vars t             = abs_bterm(vars,t)
  and  dest_bterm bt               = rep_bterm bt
;;
```

Im abstrakten Datentyp `parm` sind die verschiedene Parametertypen aus Abbildung 3.1 (Seite 104) direkt repräsentiert. Zur Bildung von Level Expressions gibt es noch weitere Möglichkeiten als die hier direkt angegebenen. Terme werden gebildet, indem man ihren Operatornamen, ihre Parameterliste und ihre Teilterme angibt. So wird zum Beispiel der Term **U**`{1:l}()` erzeugt durch:

```
mk_term (`universe`,[mk_level_parm (mk_const_level_exp 1)]) []
```
[11]

[11] Es sei angemerkt, daß in NuPRL 4.0 für alle Terme der Typentheorie bereits spezialisierte Funktionen wie `mk_universe_term`, `mk_function_term` etc. vordefiniert sind, welche Ausdrücke der obigen Art abkürzen.

Die Mechanismen zur Darstellung von NuPRL-Termen, die wir in Abschnitt 4.1.7.2 kurz ansprechen werden, sorgen dafür, daß dieser Term normalerweise das Erscheinungsbild $\mathsf{U_1}$ hat, sofern nicht explizit etwas anderes gefordert wird.

4.1.2.2 Regeln und Beweise

Beweise werden gemäß Definition 3.2.27 auf Seite 120 als Bäume dargestellt, deren Knoten aus Sequenzen und Beweisregeln bestehen. Unvollständige Beweise enthalten Blätter, die nur aus einer Sequenz bestehen. Eine Sequenz wiederum besteht aus einer Liste von Deklarationen und einer Konklusion (ein Term), wobei Deklarationen aus Variablen und Termen aufgebaut sind. Die Formulierung der entsprechenden Datentypen und Zugriffsfunktionen ist verhältnismäßig naheliegend, zumal die Definition der Beweise bereits in rekursiver Form vorliegt.

Durch eine abstrakte Definition des Datentyps `proof` kann jede unbefugte Manipulation von Beweisen unterbunden und somit die gewünschte Sicherheit des gesamten Beweisentwicklungssystems garantiert werden. Auf die Komponenten eines Beweises kann nur durch Selektorfunktionen `hypotheses`, `conclusion`, `refinement` und `children` zugegriffen werden. Veränderungen eines Beweises sind nur durch Erzeugung eines unbewiesenen Beweisziels mittels `mk_proof_goal` und durch Anwendung einer Regel auf einen Beweisknoten mit Hilfe der Funktion `refine` möglich.[12]

Die Regeln, mit denen Beweise manipuliert werden dürfen, repräsentieren die konkrete formale Theorie, welche durch das Beweissystem verarbeitet werden kann. In Definition 3.2.26 auf Seite 119 hatten wir definiert, daß eine Regel eine Sequenz – also einen unvollständigen Beweis – in eine Liste von Teilbeweisen abbildet und als Validierung angibt, wie die Extraktterme der Teilziele zu einem Extraktterm der Originalsequenz zusammenzusetzen sind. Im Kontext formaler Beweise muß die Rolle der Validierung jedoch etwas abstrakter gesehen werden sein, als nur einen Extraktterm zu generieren. Sie soll, wie der Name bereits andeutet, eine Evidenz konstruieren, *warum* die ursprüngliche Sequenz ein gültiges Urteil repräsentiert, wenn dies für die Teilziele gilt. Mit anderen Worten, sie soll beliebige Beweise der Teilziele – seien sie nun vollständig oder unvollständig – in einen Beweis des ursprünglichen Ziels zusammensetzen können. Das bedeutet, daß der tatsächliche Beweis*baum* durch die Validierung aufgebaut wird und nicht etwa durch die Regel selbst. Die Konstruktion des Extraktterms ist implizit in der Validierung enthalten, da dieser im wesentlichen als eine Term-Darstellung des konstruierten Beweises betrachtet werden kann.

Wozu ist dieser zusätzliche Aufwand nun nötig? Man könnte sicherlich darauf verzichten, wenn man Beweise ausschließlich mit Hilfe der elementaren Regeln des Kalküls konstruieren will, da diese alle Informationen enthalten, welche für die Dekomposition eines Beweiszieles und die Konstruktion eines Extrakttermes als Evidenz nötig sind. Dies reicht jedoch nicht mehr aus, wenn man mehrere Regeln zu einer Beweisregel zusammensetzen oder Beweise durch Meta-Programme von außen steuern will. In diesem Falle muß man nämlich berechnen können, welche unbewiesenen Teilziele übrigbleiben (das ist einfach) und welche Evidenz aus den Evidenzen der übriggebliebenen Teilziele entstehen soll. Letzteres aber würde bedeuten, objektsprachliche Terme zusammenzusetzen und hierzu auch auf Informationen aus Zwischenzielen zuzugreifen, die nicht mehr übrigbleiben. Im Endeffekt ist dies dasselbe wie einen Beweis aus einer Liste von Teilbeweisen zusammenzusetzen. Es ist somit einfacher und natürlicher, Validierungen als Funktionen von `proof list` nach `proof` zu beschreiben und ihnen auch die tatsächliche Erzeugung der Beweisknoten zu überlassen.

Diese Sichtweise auf Validierungen, die erstmalig im Rahmen des LCF-Konzepts der *Beweistaktiken* entstanden ist [Gordon *et.al.*, 1979] und in Abschnitt 4.2 vertieft wird, führt dazu, daß Regeln als spezielle Instanz von Beweistaktiken betrachtet werden, in die sie mit Hilfe der Funktion `refine` umgewandelt werden. Taktiken wiederum sind Funktionen, welche einen Beweis in eine Liste von Teilbeweisen und eine Validierung abbilden. Die Anwendung der Validierung auf die Liste der erzeugten Teilbeweise generiert schließlich den neuen Beweisknoten.[13] Diese Betrachtungen führen zu der folgenden Repräsentation von Beweisen durch abstrakte ML-Datentypen.

```
abstype  declaration = var # term
  with mk_assumption v t = abs_declaration(v,t)
  and  dest_assumption d = rep_declaration d
;;
lettype  sequent = declaration list # term;;
abstype  rule    = .....

absrectype  proof = (declaration list # term) # rule # proof list
  with mk_proof_goal decs t = abs_proof((decs,t), ◇,[])
  and  refine r p   = let children  = deduce_children r p
                      and validation= deduce_validation r p
                      in
                         children, validation
  and  hypotheses p = fst (fst (rep_proof p))
  and  conclusion p = snd (fst (rep_proof p))
  and  refinement p = fst (snd (rep_proof p))
  and  children   p = snd (snd (rep_proof p))
;;
lettype  validation = proof list -> proof;;
lettype  tactic     = proof -> (proof list # validation);;
```

In dieser Darstellung ist $\diamond$ eine Abkürzung für eine interne "Nullregel", die keinerlei Aktionen auslöst und nur als Platzhalter dient. Die genaue Struktur der Regeln, auf die wir hier nicht im Detail eingehen wollen, enthält eine schematische (bei komplexeren Regeln wie `arith` auch eine algorithmische) Beschreibung, wie Teilziele und Validierungen erzeugt werden sollen. Die Umwandlung dieser Beschreibungen in eine Liste von Beweisen und eine Validierung geschieht innerhalb der Funktion `refine` mit Hilfe der internen Funktionen `deduce_children` und `deduce_validation`. Dabei sind diese Funktionen so ausgelegt, daß Anwendung dieser Validierung auf die Liste der Teilbeweise einen Beweis der Gestalt

```
abs_proof( (hypotheses p, conclusion p), r, deduce_children r p)
```

generiert. Teilbeweise, die vor Anwendung der Regel in `p` enthalten waren, werden somit überschrieben. Die Funktion `refine` erzeugt eine Ausnahmesituation, wenn die Regel nicht anwendbar ist. Diese Ausnahme kann vom Beweiseditor aufgefangen werden und in eine Fehlermeldung umgewandelt werden. Für weitere Details verweisen wir auf [Constable *et.al.*, 1986, Kapitel 9.2].

Insgesamt hängt die Korrektheit eines maschinell geführten Beweises also nur von einer korrekten Repräsentation der theoretisch vorgegebenen Regeln und einer fehlerfreien Implementierung der Funktion `refine` ab. Alle anderen Komponenten des Systems beeinflussen die Eleganz des Umgangs mit dem System, haben aber keinen Einfluß auf seine Sicherheit.

4.1.3 Bibliothekskonzepte

Die Bibliothek eines Beweisentwicklungssystems ist das formale Gegenstück zu einem mathematischen Lehrbuch, in dem alle Definitionen, Sätze, Beweise, Methoden und Anmerkungen zu einem bestimmten Gebiet in linearer Reihenfolge gesammelt werden. Sie besteht aus *Objekten*, welche Terme, Beweise oder Definitionen

[12] Normalerweise müßte hierzu neben dem Namen der Regel auch die Position des zu modifizierenden Knotens im Beweis angegeben werden. In der NuPRL-Implementierung kann hierauf verzichtet werden, da der Beweiseditor (siehe Abschnitt 4.1.6) interaktive Bewegungen im Beweis und somit auch lokale Veränderungen von Beweisknoten ermöglicht.

[13] In NuPRL wird diese Anwendung der Validierung auf die Teilziele automatisch vom Beweiseditor ausgelöst. In Systemen ohne derartige Interaktionsmöglichkeiten würde die Beweiskonstruktion erheblich komplizierter und undurchsichtiger. So wurden zum Beispiel in LCF zuerst alle Regeln zu *einer* Taktik zusammengesetzt und dann geschlossen auf den Beweis angewandt.

Es sei angemerkt, daß alle in diesem Skript angegebenen Regeln im NuPRL-System bereits als Taktiken repräsentiert sind, welche aus den tatsächlichen, meist gleichnamigen Regeln durch die Funktion `refine` (und einen Mechanismus zur Generierung der notwendigen Variablennamen) entstanden sind. Dies erspart weitere Konversionen, wenn mehrere Regeln zu einer aufwendigeren Taktik zusammengesetzt (Siehe Abschnitt 4.2.4) werden sollen.

der konkreten formalen Theorie oder auch allgemeine mathematische Methoden (also Taktiken) oder Texte (Kommentare) enthalten. Zu jedem dieser Objekte gehört ein Name, eine Bezeichnung der Art des Objektes, sein Status (*vollständig, unvollständig, fehlerhaft, leer* – gekennzeichnet durch `*`,`#`,`-` und `?`) und sein Position in der Bibliothek.

Die einfachste Art, derartige Bibliotheken zu repräsentieren ist eine lineare Liste, wobei man aus Effizienzgründen einen Mechanismus für einen schnellen Zugriff über den Namen eines Objektes hinzufügen sollte. Die wichtigsten Operationen auf einer Bibliothek sind

- Erzeugen (`create`) eines neuen (leeren) Objektes durch Angabe eines Namens, der Art (`thm, abs, disp, ml, ....`) und einer Position in der Bibliothek (dem Namen des Objektes, *vor* dem es erscheinen soll) – jeweils als String.
- Löschen (`delete`) eines Objektes durch Angabe seines Namens.
- Editieren (`view`) eines Objektes durch Angabe seines Namens. Hierdurch wird der zum Objekt passende Editor aufgerufen, also ein Beweiseditor für Theoreme und ein Text-/Termeditor in allen anderen Fällen.

Darüber hinaus gibt es eine Reihe anderer nützlicher Operationen wie das Verschieben oder Umbenennen eines Objektes, eine Überprüfung des Inhaltes, das Laden von Teilbibliotheken von einer Datei, das Ablegen einer Reihe von Objekten in einer Datei, die Aufbereitung einer Reihe von Objekten für eine textliche Darstellung (z.B. in LaTeX), der Zugriff auf einzelne Komponenten eines Objektes (wie den in einem Theorem-Objekt enthaltenen Beweis oder Extrakt-Term) usw. In NuPRL ist die Bibliothek, auf die sich alle aktuellen Kommandos beziehen, als globale Variable vom Typ `library` deklariert. Dies Bibliothek wird ständig in einem speziellen Library-Fenster angezeigt und kann mit speziellen Befehlen (und Mausoperationen) durchgeblättert werden. Andere Bibliotheken können im Hintergrund gesichert und bei Bedarf zur aktuellen Bibliothek erklärt werden.

Die Existenz einer Bibliothek macht auch die in Abbildung 3.28 auf Seite 176 angegebenen Inferenzregeln `lemma` und `extract` zu einem sinnvollen Bestandteil des Inferenzsystems. Aus theoretischer Sicht kann man die Menge aller bewiesenen Theoreme einer Bibliothek als zusätzliche Hypothesen einer Beweissequenz betrachten, auf die man über den Namen des Theorems zugreifen kann. Damit sind diese beiden Regeln nichts anderes als eine besondere Form der Regeln `hypothesis` und `hypEq`, die unmittelbar auf den Hypothesen arbeiten.

Da die Darstellung einer Bibliothek als lineare Liste von Objekten eigentlich zu wenig Struktur enthält, um ein echtes Gegenstück zu einem Buch zu sein, welches aus Kapiteln, Unterkapiteln und Abschnitten besteht, gibt es in NuPRL einen simplen Mechanismus, eine Bibliothek in *Theorien* zu unterteilen. Dies geschieht durch Einfügen spezieller Kommentarobjekte, welche den Anfang und das Ende einer Theorie kennzeichnen sowie durch die Verwendung von Tabellen (globale ML-Variablen), in denen die Abhängigkeiten der Theorien untereinander und die zu einer Theorie assoziierten Filenamen enthalten sind. Die speziellen Details dieses Mechanismus sowie die wichtigsten Operationen zur Manipulation einer Bibliothek sind in [Jackson, 1993b, Kapitel 3] zu finden.

4.1.4 Die Kommandoebene

Die Kommandoebene stellt das zentrale Interface zwischen einem Beweisentwicklungssystem und seinem Benutzer dar. Sie dient dazu, das Bibliotheks-Fenster zu kontrollieren, Theorien und Teilbibliotheken zu laden und abzulegen, Editoren für Objekte der Bibliothek zu starten, mit Funktionen der Metasprache und Termen der Objektsprache zu experimentieren und externe ML-Dateien in das System hineinzuladen.

In den meisten Beweisentwicklungssystemen ist die Kommandoebene identisch mit einem Interpreter der Metasprache, mit dem die anstehenden Aufgaben in eleganter und unkomplizierter Weise gesteuert werden können, und läuft in einer Shell des Betriebssystems ab. In NuPRL hat man sich zugunsten einer Möglichkeit,

sichtbar mit Termen der Objektsprache zu experimentieren, dazu entschieden, die Kommandoebene in ein spezielles *Term-Editor* Fenster zu integrieren, innerhalb dessen sowohl Texte der Metasprache (die durch den ML-Parser kontrolliert werden) als auch Terme der Objektsprache in ihrer Display-Form editiert werden können.[14]

4.1.5 Der Text- und Termeditor

Die Verwendung verständlicher Notation innerhalb eines formalen Systems zur "Implementierung" mathematischer Theorien ist ein aktives Forschungsgebiet seit der Entwicklung der ersten Programmiersprachen. Das wesentliche Problem ist hierbei, daß einerseits zugunsten einer eleganten Handhabung durch menschliche Benutzer eine möglichst freie Syntax zur Verfügung stehen muß, andererseits die formale Sprache aber auch durch einen Computer decodierbar bleiben muß, wobei die Zeit zur Wiedererkennung objektsprachlicher Ausdrücke im Verhältnis zu den eigentlich durchzuführenden Berechnungen sehr gering sein muß.

Aus der Theorie der formalen Sprachen ist bekannt, daß eine Sprache im wesentlichen *kontextfrei* sein muß, um effizient mit Hilfe von Parsern decodiert werden zu können. Während man sich bei Programmiersprachen mittlerweile an derartige Einschränkungen in der Freiheit der Ausdrucksweise gewöhnt hat, ist dies zur Darstellung mathematischer Konzepte völlig unakzeptabel. Das Verständnis mathematischer Texte hängt zu einem Großteil von einer klaren und leicht zu merkenden Notation ab, die normalerweise viel zu komplex ist, um durch ASCII Text oder gar eine kontextfreie Syntax beschrieben werden zu können.

Beweisentwicklungssysteme, bei denen eine Interaktion mit einem Benutzer vorgesehen ist – und sei es nur für die Eingabe des Problems selbst – sind darauf angewiesen, einen Großteil der mathematischen Notation auch auf dem Bildschirm wiedergeben zu können. Aus diesem Grunde gibt es Bemühungen, durch *ausgefeiltere Parser* die Syntax formaler Sprachen flexibler zu gestalten. Dieser Ansatz ist jedoch problematisch, da die mathematische Notation voller Zweideutigkeiten steckt, die nur aus dem allgemeinen Kontext heraus richtig interpretiert werden können. So kann zum Beispiel die Juxtaposition `x y` einmal als Multiplikation zweier Zahlen oder als Applikation der Funktion `x` auf das Argument `y` interpretiert werden. Um derartige Zweideutigkeiten (also "*overloading*" in der Denkweise der Programmiersprachen) aufzulösen, muß man den Parser mit einem Type-Checker koppeln, was bei einer reichhaltigen Typstruktur sehr ineffizient werden kann.

Wesentlich effizienterer und langfristig auch vielseitiger als eine ständige Erweiterung eines Parsers ist es, die Eingabe von Termen der Objektsprache durch einen *Struktureditor* zu kontrollieren, der in der Lage ist, die Baumstruktur eines Termes direkt zu generieren, auf dem Bildschirm aber die Darstellungsform der Terme zu präsentieren. Der Vorteil dieser Vorgehensweise ist, daß

- überhaupt kein Parser mehr erforderlich ist, weil die Baumstruktur nach dem Editieren vorliegt,
- Mehrdeutigkeiten nicht mehr beachtet werden müssen, weil diese nur in der textlichen Präsentation, nicht aber intern vorkommen und auf Wunsch leicht aufgelöst werden können (man lasse sich die interne Form zeigen),
- keinerlei Beschränkung für die textliche Darstellungsform besteht – man kann alle Möglichkeiten von Textverarbeitungssystemen wie LaTeX ausschöpfen,
- ein einheitlicher Wechsel der Notation extrem einfach wird,
- Formatierung sich direkt an dem zur Verfügung stehenden Platz orientiert, und
- der Benutzer die genaue Syntax nicht kennen muß sondern nur den Namen des darzustellenden Terms.

Der größte Nachteil solcher Struktureditoren ist, daß sie ein Umdenken erfordern, wenn man gewohnt ist, die Syntax (wie in Emacs) direkt einzugeben, und daß sie bisher noch nicht so ausgereift sind wie gewöhnliche

[14] Beispiele für die Verarbeitung von Kommandos in diesem "ML Top Loop" findet man in [Jackson, 1993b, Kapitel 2&3]

Texteditoren. Dieser Nachteil ist jedoch akzeptabel, wenn man bedenkt, daß Struktureditoren bisher der einzige Weg sind, die flexible Notation mathematischer Textbücher auf formale Beweissysteme zu übertragen.

In den Texteditor des NuPRL Systems ist deshalb ein strukturierter Termeditor integriert, welcher durch ein spezielles Kommando (CONTROL-O) in einem bestimmten Bereich der Eingabe aktiviert werden kann. Dieser integrierte Text- und Termeditor steht im ML-Top Loop und beim Editieren aller Objekte mit Ausnahme der Beweisbäume zur Verfügung. Wir wollen die typische Arbeitsweise an einem Beispiel illustrieren.

Beispiel 4.1.1

Bei der Erzeugung eines Beweisziels mit dem Beweiseditor (siehe Abschnitt 4.1.6) ist der Editor automatisch im Term-Modus. Wenn wir nun einen existentiell quantifizierten Term eingeben wollen, so müssen wir nur den Namen dieses Terms, also `exists` eintippen und die RETURN taste betätigen. Danach erscheint im Display

∃[var]:[type]. [prop]

und der Cursor steht im `var`-Feld. Die Bezeichner `[var]`, `[type]` und `[prop]` dienen als Platzhalter für ein Eingabefeld und deuten an, welche Art von Information eingegeben werden soll. Sie verschwinden, sobald ein Stück Text und RETURN eingegeben wurde. Dabei wird wieder überprüft, ob der Text Name eines Terms ist, sofern wir in einem *Term-Slot* sind. Nach Eingabe von `x` RETURN haben wir

∃x:[type]. [prop]

und der Cursor steht im `type`-Feld. Wir geben `nat` RETURN ein, was dazu führt, daß der Term IN eingetragen wird, und anschließend `eqi` und erhalten

∃x:IN. [int]=[int]

wobei der Cursor im ersten `int`-Feld steht. Nach Eingabe von `x` RETURN und `4` RETURN haben wir als Endergebnis

∃x:IN. x=4.

Will man diesen Term verändern, so braucht man nur mit dem Cursor über den gewünschten Teilterm zu fahren und diesen zu ändern. Man beachte jedoch, daß nur die Inhalte der Slots oder der gesamte (Teil-)Term verändert werden können.

Erfahrungsgemäß dauert es nicht lange, bis man sich an die Vorteile eines Struktureditors gewöhnt hat und Bewegungen in Texten und Termbäumen nicht mehr miteinander verwechselt. Details über diesen Editor – vor allem die Befehle und Tastenbelegungen – findet man in [Jackson, 1993b, Kapitel 4].

4.1.6 Der Beweiseditor

Im Prinzip reichen Texteditor und Kommandoebene für das Arbeiten mit einem Beweisentwicklungssystem völlig aus. Es ist durchaus möglich, Beweise durch ML Kommandos zu manipulieren und sich das Ergebnis wieder anzeigen zu lassen. Dennoch ist diese Vorgehensweise äußerst unpraktisch, wenn man Beweise interaktiv entwickeln möchte. Man müßte ständig die unbearbeiteten Beweisknoten eruieren und ihren Inhalt ansehen, bevor man weiterarbeiten kann. Wesentlich sinnvoller ist daher, die Manipulation von Beweisen durch einen speziellen Beweiseditor zu unterstützen, mit dem man sich quasi graphisch durch den Beweisbaum bewegen und Knoten ansehen und modifizieren kann. Dabei beschränken sich die Manipulationsmöglichkeiten natürlich auf die Eingabe des initialen Beweisziels und der Regeln, mit denen man die Sequenz eines Knotens verfeinern möchte, sowie auf Bewegungen innerhalb des Beweisbauems. Abgesehen davon, daß man Beweisknoten verständlich darstellen muß, ist ein solcher Beweis- oder Verfeinerungseditor eine einfache Programmieraufgabe und daher der Steuerung von Beweisen durch Kommandos vorzuziehen.

Der Beweiseditor des NuPRL Systems, der aufgerufen wird, wann immer ein Theorem-Objekt mit `view` betrachtet wird, arbeitet knotenorientiert. Man sieht also nie den gesamten Beweisbaum, sondern nur einen

Name des Theorems	EDIT THM intsqrt
Status, Position relativ zur Wurzel	# top 1
Erste Hypothese des Beweisziels	1. x:IN
Konklusion	⊢ ∃y:IN. y^2≤x ∧ x<(y+1)^2
Regel	BY natE 1
Erstes Teilziel – Status, Konklusion	1# ⊢ ∃y:IN. y^2≤0 ∧ 0<(y+1)^2
Zweites Teilziel – Status,	2# 2. n:IN
neue Hypothesen	3. 0<n
	4. v:∃y:IN. y^2≤n-1 ∧ n-1<(y+1)^2
Konklusion	⊢ ∃y:IN. y^2≤n ∧ n<(y+1)^2

Abbildung 4.1: Darstellung eines Beweisknotens im Editorfenster

speziellen Knoten, dessen Position relativ zur Wurzel des Beweises im Editorfenster angezeigt wird. Angezeigt werden außerdem das aktuelle Beweisziel und – sofern vorhanden – die angewandte Regel sowie die erzeugten Unterziele. Abbildung 4.1 zeigt ein typisches Beispiel für die Darstellung eines Beweisknotens.

Beim Aufruf befindet man sich im Wurzelknoten und kann sich mit Hilfe von Maus und speziellen Tastenkombinationen durch den Beweis bewegen.[15] Die lokale Sicht auf den Beweis unterstützt das Arbeiten mit dem Sequenzenkalkül, dessen großer Vorteil ja gerade die lokale Behandlung von Beweiszielen ist. Durch den Editor wird sichergestellt, daß nur das Beweisziel des Wurzelknotens und die jeweiligen Regeln des Beweises verändert werden können. Alle anderen Veränderungen werden (durch Aufruf der Funktion `refine`) automatisch bestimmt. Wir wollen die typische Arbeitsweise des Beweiseditors an einem Beispiel erläutern.

Beispiel 4.1.2

Um ein Programm zur Berechnung von Integerquadratwurzeln (vergleiche Beispiel 3.4.9 auf Seite 154) mit Hilfe eines formalen NuPRL-Beweises zu generieren, erzeugen wir zunächst mit `create` ein geeignetes Theorem-Objekt und rufen dann mit `view` den Editor auf. Es erscheint das folgende Fenster

```
EDIT THM intsqrt
 ? top
 <main proof goal>
```

Nun wird (z.B. mit der Maus) das Feld des Beweiszieles selektiert und somit der Termeditor aufgerufen. Mit diesem Editor erzeugt man das Beweisziel und schließt es dann wieder. Dabei wird das Beweisziel auf syntaktische Korrektheit überprüft und in das entsprechende Feld übernommen.[16] Der Status, der vorher *leer* ("?") war, wird in *unvollständig* ("#") verändert. Das Editorfenster zeigt nun folgende Inhalte an.

```
EDIT THM intsqrt
 # top
 ⊢ ∀x:IN.∃y:IN. y²≤x ∧ x<(y+1)²

 BY <refinement rule>
```

Um dieses Theorem nun zu beweisen, müssen wir für das Regel-Feld des Beweisknotens mit Hilfe des Termeditors eine Regel (oder eine Taktik) angeben. In diesem speziellen Fall wird die Regel `all_i` im Termeditor angegeben und dieser wieder geschlossen. Dies löst die folgenden internen Schritte aus.

1. Eine globale Variable `prlgoal` vom Typ `proof` wird mit der aktuellen Beweissequenz, also mit ∀x:IN.∃y:IN. y^2≤x ∧ x<(y+1)^2 assoziiert. Bisher bestehende Teilziele und Regeln eines eventuell schon vorhandenen Teilbaumes werden ignoriert.

[15] Es gibt Überlegungen, durch eine separate graphische Darstellung des Beweisbaums die Bewegungen zu beschleunigen und unbewiesene Teilziele leichter überschaubar zu machen.

[16] Ein eventuell vorher vorhandenes Beweisziel und frühere Regeln gehen hierdurch verloren.

2. Die Regel all_i und wird zusammen mit ihren (in diesem Fall nicht vorhandenen) Argumenten mit Hilfe von refine in eine Taktik umgewandelt und auf die Variable prlgoal angewandt, was zu einer (möglicherweise leeren, hier einelementigen) Liste von Teilzielen und einer Validierung führt.
3. Die Validierung wird auf die Teilziele angewandt und erzeugt einen Beweisbaum der Tiefe 1.
4. Der Beweisbaum wird in den Beweis des aktuellen Theorems integriert, wobei insbesondere der Name der Regel in das Regelfeld übernommen wird.
5. Der Inhalt des Beweiseditorfensters (also Status und Teilzieldarstellung) wird neu berechnet und angezeigt (ist die Regel nicht anwendbar, so wird der Status auf *fehlerhaft* gesetzt und eine Fehlermeldung ausgegeben).

Nach Eingabe von all_i erhalten wir somit die folgende Darstellung des resultierenden Beweisknotens.

```
EDIT THM intsqrt
 # top
 ⊢ ∀x:ℕ.∃y:ℕ. y²≤x ∧ x<(y+1)²

 BY all_i

 1# 1. x:ℕ
    ⊢ ∃y:ℕ. y²≤x ∧ x<(y+1)²
```

Um nun den nächsten Schritt ausführen zu können, müssen wir uns in den nächsttieferen Beweisknoten bewegen, indem wir das Feld des ersten (und einzigen) Teilziels selektieren. Wir erhalten

```
EDIT THM intsqrt
 # top 1
 1# 1. x:ℕ
    ⊢ ∃y:ℕ. y²≤x ∧ x<(y+1)²

 BY <refinement rule>
```

und können nun schrittweise und auf ähnliche Art wie zuvor den Beweis zuende führen.

Es sei angemerkt, daß sich auch bei einer interaktiven Unterstützung eines formalen Beweises eine gewisse Vorausplanung des Beweisgangs empfiehlt, da das System einem nur die lästige Schreibarbeit und die Korrektheitsüberprüfung, nicht aber die Beweisideen abnehmen kann. Eine genaue Beschreibung des aktuellen Beweiseditors von NuPRL findet man in [Jackson, 1993b, Kapitel 7]. Verbesserungsvorschläge werden gerne entgegengenommen.

Extraktionsmechanismus

In den Beweiseditor integriert ist ein Extraktionsmechanismus, mit dem die implizit in einem Beweis enthaltenen Programme extrahiert werden können, sobald der Beweis vollständig vorliegt. Dabei werden die Extraktterme der einzelnen im Beweis enthaltenen Regeln schrittweise gemäß der rekursiven Definition auf Seite 120 zusammengesetzt. Der gesamte Extraktterm wird zusammen mit dem Beweis im Theorem-Objekt abgelegt. Im ML Top Loop kann hierauf dann mittels extract_of_thm_object *name* zugegriffen werden.

Der Beweiseditor und der zugehörige Programmgenerator basieren auf den allgemeinen Grundkonzepten des *Cornell Program Synthesizer Generator* [Teitelbaum & Reps, 1981, Reps & Teitelbaum, 1984] und speziellen Extraktionsmechanismen, die in [Bates, 1981, Sasaki, 1986, Stansifer, 1985] dokumentiert sind.

4.1.7 Definitionsmechanismus

Bereits in Abschnitt 2.1.5 auf Seite 14 haben wir über die Notwendigkeit gesprochen, einen formalen Kalkül durch definitorische Abkürzungen konservativ erweitern zu können. Definitorische Abkürzungen sind ein essentieller Bestandteil aller mathematischen Theorien, denn sie erlauben es, lange und komplexe Ausdrücke durch

eine kurze, prägnante Notation zu ersetzen und somit die Darstellung der Aussagen der Theorie verständlich zu halten. Es ist offensichtlich, daß ein gutes Beweisentwicklungssystem diese Vorgehensweise unterstützen muß, also einen *Definitionsmechanismus* benötigt, welcher erlaubt, konservative Erweiterungen der zugrundeliegenden Theorie auf elegante Weise einzuführen.

Ein sehr einfacher Mechanismus (der auch in früheren Versionen von NuPRL benutzt wurde) ist die Verwendung von *Textmacros*, bei denen ein langer formaler Text auf dem Bildschirm durch ein abkürzendes Macro repräsentiert wird. Dies hat aber den Nachteil, daß das System zwischen dem abkürzenden Text und der ausführlichen Form nicht unterscheiden kann. Der Gedankengang der Abstraktion, also die Einführung eines neuen Begriffs, wird dadurch nicht adäquat erfaßt. Sinnvoller ist daher, einen *Abstraktionsmechanismus* zu entwickeln, mit dem neue Terme der formalen Sprache deklariert werden, die nur durch Auffalten (die fold-Regel) in den Ausdruck überführt werden können, den sie abkürzen.[17] Entsprechend unserem Prinzip 3.2.4 auf Seite 103 wollen wir hierbei die einheitliche Term-Syntax verwenden und die Darstellung dieser Terme auf dem Bildschirm separat definieren.

4.1.7.1 Abstraktion

Innerhalb eines Beweisentwicklungssystems werden konservative Erweiterungen der Theorie durch Abstraktionsobjekte der Bibliothek eigeführt. Diese enthalten Definitionen der Form

lhs == *rhs*,

wobei *lhs* den neu deklarierten Term beschreibt und *rhs* den Ausdruck, der durch *lhs* abgekürzt werden soll. Beide Seiten sind Termschemata mit eventuell frei vorkommenden (Meta-)variablen, die implizit allquantifiziert sind. Derartige Definitionen haben wir im vorigen Kapitel an vielen Stellen benutzt wie zum Beispiel bei der Einführung von Logik-Operatoren in Definition 3.3.3 auf Seite 134:

$$\begin{array}{ll} \textbf{and}\{\}(A;B) & == \ A\times B \\ \textbf{exists}\{\}(T;x.P) & == \ x{:}T\times P \\ \textbf{prop}\{i{:}\mathtt{l}\}() & == \ \mathbb{U}_i \end{array}$$

Hier sind A, B, T und P Platzhalter (oder *Metavariablen*) für Terme, x Platzhalter für eine Variable, die in P frei vorkommen darf und i Platzhalter für einen Parameter. Beim Auffalten einer konkreten Instanz der linken Seite werden diese Platzhalter durch Substitutionen an konkrete Terme oder Parameter gebunden und entsprechend auf der rechten Seite ersetzt. Um diesen Mechanismus zu realisieren, reichen gewöhnliche Substitutionen erster Stufe und der zugehörige Matching-Algorithmus jedoch nicht mehr aus, wie das folgende Beispiel zeigt.

Beispiel 4.1.3

Wenn wir versuchen, eine Substitution zu finden, die das Schema $\exists x{:}T.P$ in die spezielle Instanz $\exists$y:ℕ.y<5 überführt, so werden wir feststellen, daß die Substitution $[$y,y<5,ℕ $/x,P,T]$ hierfür nicht ausreicht, da ihre Anwendung auf das Schema gemäß Definition 3.2.7 (Seite 105) zu einer Umbenennung der Variablen y, also zu einem Term der Gestalt $\exists$y':ℕ.y<5 führen würde, in dem der gewünschte Zusammenhang zwischen der quantifizierten Variablen und der freien Variablen des Terms nicht mehr besteht. Auf die Umbenennungsvorschrift kann aber nicht verzichtet werden, da ansonsten auch andere Variablen unbeabsichtigt in den Bindungsbereich eines Quantors geraten können.

Ein Blick auf die interne Darstellung der in diesem Beispiel benutzten Terme zeigt, wo das Problem liegt. In **exists**$\{\}(T;x.P)$ repräsentiert die Metavariable P einen Term, in die Bindungsvariable x frei vorkommen kann. P ist in Wirklichkeit also eine Metavariable *zweiter Stufe* der Stelligkeit 1. Sie darf nicht durch einen einfachen Term substituiert werden, sondern nur durch einen Term, in dem ein Platzhalter für einen anderen Term vorkommt, der dann durch x zu ersetzen ist. Derartige Terme nennt man *Terme zweiter Stufe*. Sie entsprechen den gebundenen Termen im Sinne von Definition 3.2.5.

[17] Der Unterschied ist vergleichbar mit dem Unterschied zwischen lettype und abstype in ML.

Das Substitutionsverfahren ist also komplizierter als bei Substitutionen erster Stufe. Eine Substitution zweiter Stufe hat die Gestalt $[x_1, ..., x_m.t_{x_1,...,x_m}/P]$, wobei P eine Variable zweiter Stufe und $x_1, ..., x_m.t_{x_1,...,x_m}$ ein Term zweiter Stufe ist. Die Anwendung dieser Substitution auf eine konkrete Instanz $P[a_1, ..., a_n]$ der Variablen P liefert den Term $t_{a_1,...,a_n}$. Die konkreten freien Variablen innerhalb der Instanz P ersetzen also die Platzhalter $x_1, ..., x_m$ in y. Wenn wir also eine Substitution suchen, die $\exists x{:}T.P$ in $\exists$y:IN.y<5 überführt, so benötigen wir die Substitution zweiter Stufe

[y.y<5, IN /P, T].

Die Anwendung dieser Substitution auf $\exists x{:}T.P$ führt dann zu dem Term $\exists x{:}$IN.x<5, der α-konvertibel zu dem gewünschten Term ist.[18]

Alle anderen Platzhalter, also Metavariablen erster Stufe und Platzhalter für Parameter sind unproblematisch. Sie können mit einem Verfahren behandelt werden, welches Matching und Substitutionsanwendung erster Stufe entspricht. Die Details des Abstraktionsmechanismus von NuPRL, insbesondere die konkreten Methoden zur Kennzeichnung von Variablen zweiter Stufe und von Meta-Parametern findet man in [Jackson, 1993b, Kapitel 5] beschrieben.

4.1.7.2 Termdarstellung

In Ergänzung des Abstraktionsmechanismus sorgt ein spezieller Display-Mechanismus für eine elegante und leicht lesbare Darstellung von mathematischen Texten auf dem Bildschirm. Neben der textlichen Präsentation eines Terms in einer nahezu beliebigen Syntax kann man mit diesem Mechanismus eine Reihe von anderen Kontrollmöglichkeiten über die Darstellung ausüben, wie zum Beispiel die folgenden:

- Formatierung des Textes in einem Fenster in Abhängigkeit von der Größe dieses Fensters.
- Automatische Klammerung von Termen niedrigerer Priorität, wenn diese als Teilterme eines anderen Terms auftreten. So wird **add**{}(**mul**{}(4,5),6) als 4*5+6 , **mul**{}(4,**add**{}(5,6)) aber als 4*(5+6) dargestellt, ohne daß der Benutzer dies erzwingen muß.
- Automatische Iteration von Operatoren wie z.B. die Verwendung von λx y.x+y anstelle von λx.λy.x+y.
- Automatische Anwendung sonstiger Verkürzungen wie z.B. 4=5 anstelle von 4=5 $\in \mathbb{Z}$, sobald diese anwendbar sind.

Dies erhöht die Eleganz und Flexibilität der Darstellung formaler Texte und erleichtert somit den Übergang von informal präsentierten mathematischen Theorien zu der in Beweissystemen verwendeten Form.

Diese Mechanismen wurden vor allem durch die Hypertext-Technik möglich gemacht und sind noch im Stadium der Weiterentwicklung. Den gegenwärtigen Stand findet man in [Jackson, 1993b, Kapitel 6].

4.1.8 Der Programmevaluator

Die Terme der intuitionistischen Typentheorie beschreiben im Endeffekt eine funktionale Programmiersprache deren Berechnungsvorschriften wir im vorigen Kapitel ausführlich besprochen haben. Neben der Möglichkeit, mit dem Beweissystem über das Resultat dieser Berechnungen zu *schließen*, bietet NuPRL auch einen Mechanismus an, Programme – seien sie nun aus Theoremen extrahiert oder direkt geschrieben – auch innerhalb des Systems laufen zu lassen und praktisch auszutesten. Hierzu wird die Funktion evaluate_term eingesetzt, die den ohnehin in das System integrierten Auswertungsmechanismus aus Abbildung 3.3 (Seite 108) benutzt.

[18] In NuPRL ist der Standardalgorithmus für die Anwendung von Substitutionen zweiter Stufe zugunsten der besseren Lesbarkeit so abgewandelt worden, daß die tatsächlichen Namen der bindenden Variablen erhalten bleiben.

4.1.9 Korrektheit der Implementierung

Die Korrektheit von Beweisen, die mit Hilfe eines interaktiven Beweissystems geführt werden, kann wegen der abstrakten Definition des Datentyps proof in Abschnitt 4.1.2 relativ leicht sichergestellt werden. Veränderungen von Beweisen sind ausschließlich mit Hilfe der Funktion refine möglich, die wiederum auf die vordefinierten Inferenzregeln zurückgreifen muß. Unbefugte oder irrtümliche Manipulationen von Beweisen sind somit gänzlich ausgeschlossen. Die Zuverlässigkeit des interaktiven Beweissystems hängt somit ausschließlich von der korrekten Implementierung der Funktion refine und der einzelnen Inferenzregeln (und natürlich von der korrekten Arbeitsweise der Implementierungssprache ML und des Betriebsystems) ab.

Für die Implementierung von Inferenzregeln bietet sich eine schematische Darstellung an, die sich an die im vorigen Kapitel gegebene textliche Repräsentation von Regeln anlehnt und Metavariablen und -parameter gesondert kennzeichnet. Auf diese Art ist eine Übereinstimmung der implementierten Regeln mit dem formalen Theorie leicht zu überprüfen.

Die einzige Funktion, die wirklich verifiziert werden muß, ist die Funktion refine. Sie muß im wesentlichen ein Regelschema (samt Extraktterm) für ein gegebenes Beweisziel korrekt instantiieren und hieraus die Nachfolgerknoten und die Validierung generieren. Ihre Korrektheit folgt somit aus der Korrektheit der eingebauten Matching- und Substitutionsalgorithmen, die ebenfalls nicht schwer zu beweisen ist.

4.2 Taktiken – programmierte Beweisführung

Mit den bisher vorgestellten Komponenten des Beweisentwicklungssystems sind wir in der Lage, formale Beweise interaktiv zu entwickeln und hieraus Programme zu extrahieren. Als Benutzer des Systems brauchen wir nur unsere Theoreme zu formulieren und zum Beweis die jeweiligen Regeln anzugeben. Die Ausführung der Regeln, die Korrektheitsüberprüfung und vor allem die Schreibarbeit bei der Erzeugung der Teilziele bleibt dem System überlassen. Hierdurch wird die Entwicklung formaler Beweise mit garantierter Korrektheit bereits erheblich erleichtert.

Nichtsdestotrotz zeigt sich schon bei relativ einfachen Problemstellungen, daß eine formale Beweisführung immer noch zu kompliziert ist, da die Beweise zu viele Details enthalten, die sie unübersichtlich werden lassen. Aus diesem Grunde ist es notwendig, eine Rechnerunterstützung anzubieten, die über die Möglichkeiten interaktiver Beweissysteme hinausgeht, und Techniken bereitzustellen, mit denen die Beweisführung zumindest zum Teil automatisiert und übersichtlicher gestaltet werden kann. Hierzu könnte man nun das Inferenzsystem um fest eingebaute Beweisprozeduren ergänzen, die in derLage sind, Teilprobleme vollautomatisch zu lösen. In einigen Anwendungsbereichen (siehe Abschnitt 4.3) ist dies sicherlich ein sinnvoller Weg. Wegen der großen Ausdruckskraft der Typentheorie und der Vielfalt der Anwendungsmöglichkeiten ist diese Vorgehensweise jedoch nicht flexibel genug, da sie größere Eingriffe in das Beweisentwicklungssystem verlangen würde, wann immer eine neuartige Problemstellung damit bearbeitet werden soll.

Aus diesem Grunde ist es wichtig, Mechanismen bereitzustellen, die den Benutzern des Systems ermöglichen, das Inferenzsystem um eigene Beweisstrategien zu ergänzen und damit zu experimentieren, und dabei gleichzeitig die größtmögliche Sicherheit gegenüber fehlerhaften Beweisen zu bieten. Diese Idee der *benutzerdefinierten Erweiterung* von Beweissystemen, die erstmals innerhalb des LCF Projektes [Gordon *et.al.*, 1979] aufkam, läßt sich auf einfache Weise realisieren, wenn die Metasprache des Systems als Programmiersprache – in unserem Fall ML – formalisiert ist, da in diesem Fall ein Benutzer den Aufruf von Beweisregeln und die Bestimmung der nötigen Parameter *programmieren* kann. Erlaubt man einem Benutzer also, metasprachliche Programme in Beweisen zu verwenden, so erhält man die gewünschte Flexibilität in der Beweisführung, wobei die Darstellung der Datentypen proof und rule dafür sorgt, daß keine fehlerhaften Beweise entstehen können.

Durch einen Zugriff auf die Metasprache wird ein Benutzer in die Lage versetzt, Beweise im Voraus zu *planen* und entsprechend zu programmieren, anstatt sie rein interaktiv auszuführen. Darüber hinaus kann er aber auch Beweisstrategien entwickeln, die versuchen, einen Beweis automatisch zu finden, und somit die Beweisführung von Teilproblemen entlasten, deren Beweis naheliegend ist aber voller mühsamer Details steckt. Die Programmierung auf der Metaebene des Kalküls ermöglicht also eine *taktische* Vorgehensweise in Ergänzung zu der interaktiven Beweisführung. *Taktiken* – also Metaprogramme, die auf Beweisen operieren – können eingesetzt werden, um nach Beweisen zu *suchen*, bestehende Beweise zu *modifizieren*, uninteressante Beweis*details zu verstecken*, komplexe Beweise zu *strukturieren* und – zusammen mit dem Abstraktionsmechanismus – die formale Sprache und das Inferenzsystem des zugrundeliegenden Kalküls um *benutzerdefinierte Konzepte* beliebig zu erweitern. Das Prinzip des *taktischen Theorembeweisens* liefert einen einfachen Weg, die Präzision von Computern mit dem Einfallsreichtum des menschlichen Benutzers zu koppeln und so einen einfachen Proof Checker in ein mächtiges Werkzeug für den Beweis von Theoremen, die Konstruktion von Programmen und die Entwicklung formaler mathematischer Theorien zu verwandeln.

In diesem Abschnitt wollen wir nun die wesentlichen Aspekte einer Realisierung des Taktik-Konzeptes am Beispiel der wichtigsten NuPRL Taktiken diskutieren.

4.2.1 Grundkonzepte des taktischen Beweisens

Das Konzept der taktischen Beweisführung basiert im wesentlichen auf der Methode des heuristischen Problemlösens, die schon von den altgriechischen Mathematikern eingesetzt, von Polya [Polya, 1945] systematisiert und erstmalig im Logic Theorist von Newell, Shaw und Simon [Newell *et.al.*, 1963] programmiert wurde. In dieser Denkweise ist ein Problem (oder Beweisziel) nichts anderes als "eine Menge möglicher Lösungskandidaten zusammen mit einem Testverfahren, welches überprüft ob ein gegebenes Element dieser Menge tatsächlich eine Lösung für das Problem ist" [Minsky, 1963]. In LCF [Gordon *et.al.*, 1979] wurde dieser Gedanke noch verallgemeinert und das Testverfahren durch den Begriff des *Erreichens* (englisch: *achievement*) ersetzt, der eine mögliche Beziehung zwischen einem Ziel (*goal*) und einem (Lösungs-)*Ereignis* (*event*) herstellt. In diesem Sinne können "viele Situationen des Problemlösens als spezielle Instanzen der drei Konzepte Ziel, Ereignis und Erreichen" verstanden werden.

Diese allgemeine Sicht auf Problemlöseprozesse erlaubt es, universelle Techniken zu entwickeln, die sich in allen Bereichen des Problemlösens – also nicht nur innerhalb eines festen Kalküls – einsetzen lassen und mittlerweile in vielen Systemen (siehe Fußnote 3 auf Seite 182) Anwendung gefunden haben. Taktiken sind dabei nichts anderes als eine Formulierung der Idee einer zielorientierten (top-down) Heuristik in dieser Denkweise. Eine Taktik zerlegt ein Beweis*ziel* G in eine endliche Liste von Teilzielen $G_1, \ldots, G_n$ sowie eine Validierung v. Wenn nun die *Ereignisse* e_i die Teilziele g_i *erreichen*, dann dient die Validierung v dazu, ein Ereignis $e = v(e_1, \ldots, e_n)$ zu konstruieren, welches das ursprüngliche Ziel G erfüllt. Bei dieser Vorgehensweise wird also im Top-Down Verfahren nach einer Lösung des Problems gesucht und diese dann bottom-up (zu einem erreichenden Ereignis) zusammengesetzt.

In der Sprache von Beweiskalkülen können wir ein Ziel als ein Theorem (also eine Sequenz) ansehen und ein Ereignis als seinen Beweis, welcher eine Evidenz für die Gültigkeit des Theorems sein soll. Beweise dürfen dabei unvollständig sein und Sequenzen können als degenerierte unvollständige Beweise angesehen[19] werden. Dieses Verständnis führt dann genau zu den in Abschnitt 4.1.2 angegebenen Datentypen. Der Begriff des Erreichens ist nun recht einfach umzusetzen: ein Beweis p erreicht ein Ziel G, wenn die Sequenz in seiner Wurzel genau die Sequenz G (also die Wurzelsequenz des degenerierten Initialbeweises) ist.[20]

Eine Taktik löst also einen Teil der Problemstellung und hinterläßt Teilziele, die sie nicht vollständig beweisen konnte und eine Validierung, welche Beweise für diese Teilziele in einen Beweis des Ausgangsziels umwandelt. Wenn diese Validierung nur auf vollständige Beweise angewandt wird – insbesondere also, wenn überhaupt keine Teilziele übrigbleiben, dann entsteht ein vollständiger Beweis des ursprünglichen Ziels.

[19] Wenn Taktiken in einem interaktiven System aufgerufen werden sollen, müssen sie auch unvollständige Beweise generieren dürfen, ohne fehlzuschlagen. Die Identifizierung von Sequenzen und Beweisen wird besonders bei Transformationstaktiken wichtig.

[20] In der NuPRL-Implementierung des Taktik-Konzeptes wird sichergestellt, daß Taktiken, wenn sie überhaupt anwendbar sind, nur Beweise erzeugen können, deren Wurzelsequenz das Ausgangsziel ist. Auf diese Art braucht der Begriff des Erreichens nicht gesondert überprüft zu werden.

Der Vorteil dieser etwas kompliziert anmutenden Vorgehensweise, die uns bereits in Abschnitt 4.1.2 im Zusammenhang mit der Implementierung von Beweisregeln begegnet ist, wird deutlich, wenn wir Taktiken zusammensetzen wollen. Da eine Taktik eine explizite Liste von Teilzielen generiert, können wir andere Taktiken einfach hierauf anwenden und somit das Beweisziel schrittweise zerlegen. Der Aufbau des eigentlichen Beweises wird dabei automatisch durch die entsprechenden Validierungen durchgeführt. Dies macht es sehr einfach, spezialisierte Taktiken zu programmieren, die nur in wenigen Fällen Anwendung finden (siehe Abschnitt 4.2.5) und Hilfsmittel für eine leichte Komposition von Taktiken bereitzustellen.

Es gibt zwei grundsätzliche Klassen von Taktiken: *Verfeinerungstaktiken* und *Transformationstaktiken*.

4.2.2 Verfeinerungstaktiken

Verfeinerungstaktiken können in erster Näherung als *abgeleitete Inferenzregeln* betrachtet werden. Sie werden angewandt, indem im Beweiseditor anstelle einer elementaren Kalkülregel der Name einer Taktik eingegeben wird. Dieser Name wird auch als Regelname im Editorfenster erscheinen, wenn die Taktik erfolgreich ausgeführt werden kann. Als Teilziele erscheinen dann diejenigen Ziele, die von der Taktik nicht vollständig bewiesen werden konnten. Alle zwischenzeitlich berechneten Ziele werden zwar (zugunsten einer effizienten Extraktion) im Beweis gespeichert, bleiben aber für den Anwender des Systems unsichtbar.

Beispiel 4.2.1

Ein typisches Beispiel für eine Verfeinerungstaktik ist die Taktik `cases`, welche eine Fallunterscheidung ausführt. Dies geschieht normalerweise dadurch, daß man zuerst eine Disjunktion $A \vee B$ über die Schnittregel einführt und dann diese Disjunktion eliminiert, um die Fälle A und B einzeln betrachten zu können. Im Beweiseditor würde man also die folgenden Schritte ausführen

EDIT THM cases
`# top` $\vdash T$ `BY cut 1` $A \vee B$ `1#` $\vdash A \vee B$ `2# 1.` $A \vee B \quad \vdash T$

und danach

EDIT THM cases

`# top 2`
 `1.` $A \vee B \quad \vdash T$
 	
 `BY or_e 1`
 	
 `1# 1.` $A \quad \vdash T$
 	
 `2# 1.` $B \quad \vdash T$

Eigentlich sind diese beiden Schritte jedoch eine Einheit und müßten zusammengefaßt werden. Dies genau geschieht durch die Taktik `cases`, die wir in Beispiel 4.2.4 auf Seite 204 ausprogrammieren werden. Sie liefert in einem Schritt folgendes Resultat.

EDIT THM cases
`# top` $\vdash T$ `BY cases` A B `1#` $\vdash A \vee B$ `2# 1.` $A \quad \vdash T$ `3# 1.` $B \quad \vdash T$

Der Zwischenschritt wird hierbei zwar wie oben ausgeführt, bleibt aber unsichtbar.

Im Prinzip führt der Beweiseditor beim Aufruf einer Verfeinerungstaktik dieselben Schritte aus wie bei der Anwendung einer elementaren Beweisregel (vergleiche unsere Beschreibung in Beispiel 4.1.2 auf Seite 194).

1. Die globale Variable `prlgoal` wird mit der aktuellen Beweissequenz assoziiert. Bisher bestehende Ziele und Regeln eines eventuell schon vorhandenen Teilbaumes werden ignoriert.

2. Die Taktik wird auf die Variable `prlgoal` angewandt, was zu einer (möglicherweise leeren) Liste von Teilzielen und einer Validierung führt.

3. Die Validierung wird auf die Teilziele angewandt und erzeugt einen Beweisbaum.
4. Der Beweisbaum wird zusammen mit dem Namen der Taktik als Verfeinerungsregel des aktuellen Beweises eingetragen. Die von der Taktik generierten Teilziele werden die Teilziele des Verfeinerungsschrittes.
5. Der Inhalt des Beweiseditorfensters wird neu berechnet und angezeigt, wobei nur der Name der Taktik als Verfeinerungsregel erscheint. Ist die Taktik nicht anwendbar, so wird der Status auf *fehlerhaft* gesetzt und eine Fehlermeldung ausgegeben.

Man beachte, daß der eigentliche Beweis erst durch die Validierung aufgebaut wird, nachdem alle Teilziele berechnet wurden. Dadurch entfällt die Notwendigkeit, einen Teil des erzeugten Beweises wieder rückgängig zu machen, wenn die Taktik nach vielen Schritten im Endeffekt fehlschlagen sollte.

4.2.3 Transformationstaktiken

Transformationstaktiken sind von ihrem Typ her identisch mit Verfeinerungstaktiken, unterscheiden sich von diesen aber durch die Art der Anwendung. Ihr Zweck ist nicht die Verfeinerung eines Beweisziels sondern die Transformation eines bestehenden Beweises in einen anderen. Daher betrachten sie üblicherweise auch nicht nur das Beweisziel, sondern den gesamten Beweisbaum, der in dem aktuellen Knoten des Beweises beginnt. Aus Benutzersicht erzeugen sie auch nicht nur neue Teilziele sondern einen ganzen Beweisbaum. Nach erfolgreicher Anwendung erscheint ihr Name normalerweise nicht im entstandenen Beweis. Transformationstaktiken werden üblicherweise dazu eingesetzt, Beweise sichtbar zu vervollständigen, zu expandieren, oder Beweise zu konstruieren, die in einem gewissen Sinn analog zu bestehenden Beweisen sind.

Beispiel 4.2.2

Ein typisches Beispiel für die Anwendung von Transformationstaktiken ist das Taktik-Paar `Mark` und `Copy`, die dazu benutzt werden können, das gleiche Argument an mehreren Stellen eines Beweises anzuwenden, ohne hierzu ein Lemma zu formulieren. Hierzu wird in einem ersten Schritt der in einem Knoten beginnende Beweis mit `Mark` ‘*name*‘ in einer globalen Variablen abgespeichert. Anschließend bewegt man sich zu dem Beweisknoten (ggf. auch in einem anderen Theorem-Objekt), in dem dasselbe Argument benutzt werden soll und benutzt `Copy` ‘*name*‘, um diesen Beweis erneut auszuführen. Dabei müssen alle Regeln des gespeicherten Beweises ohne Abänderung anwendbar sein, um eine analoge Kopie des Beweises zu erzeugen.

Eine Transformationstaktik wird innerhalb des Beweiseditors durch Aufruf des Termeditors mithilfe eines speziellen Befehls generiert. Bei ihrer Ausführung werden folgende internen Schritte durchgeführt.

1. Die Variable `prlgoal` wird mit dem gesamten Beweis unterhalb der aktuellen Beweissequenz assoziiert.
2. Die Taktik wird auf `prlgoal` angewandt und liefert eine Liste von Teilzielen und eine Validierung.
3. Die Validierung wird auf die Teilziele angewandt und erzeugt einen Beweisbaum.
4. Der gesamte erzeugte Beweisbaum wird in den Beweis des aktuellen Theorems anstelle des vorherigen Teilbeweises integriert.
5. Der Inhalt des Beweiseditorfensters wird neu berechnet und angezeigt. Ist die Taktik nicht anwendbar, so bleibt der Beweis unverändert und eine Fehlermeldung wird ausgegeben.

Eine Transformationstaktik schreibt insbesondere auch die Namen aller angewandten Regeln in die entsprechenden Regel-Felder des Beweiseditors und zeigt dem Anwender somit alle Details des ausgeführten Beweises. Daher ist es durchaus auch sinnvoll, Taktiken, die ursprünglich als Verfeinerungstaktiken geschrieben wurden, als Transformationstaktiken ausführen zu lassen. In diesem Fall wird nicht, wie sonst üblich, nur das Endergebnis der Beweisführung angezeigt, sondern alle einzelnen Schritte des Beweises sichtbar gemacht. Wir wollen diesen Unterschied an einem Beispiel erläutern.

Beispiel 4.2.3

Die Taktik `simple_prover`, deren Implementierung wir in Abbildung 4.4 auf Seite 207 beschreiben werden, ist in der Lage, einfache logische Schlüsse automatisch auszuführen, solange hierzu keine komplexen Entscheidungen zu treffen sind wie etwa die Angabe eines Terms für die Ausflösung eines Quantors, oder die Auswahl einer zu beweisenden Alternative in einer Disjunktion. Von ihrer Konzeption her ist sie eine typische Verfeinerungstaktik. Sie kann zum Beispiel benutzt werden, um das Ziel

$$\forall x{:}\mathbb{N}.\ \forall y{:}\mathbb{N}.\ x<y\ \wedge\ (\neg x<y\ \vee\ 4=x)\ \Rightarrow\ 4=x$$

vollständig zu beweisen. Um sie als Verfeinerungstaktik anzuwenden, müssen wir sie als Regel des entsprechenden Beweiszieles angeben:

```
EDIT THM or_test
 # top
 ⊢ ∀x:ℕ. ∀y:ℕ. x<y ∧ (¬x<y ∨ 4=x) ⇒ 4=x

 BY <refinement rule>
```

```
EDIT Rule of or_test

 simple_prover
```

Nachdem wir das Editorfenster für den Regelnamen geschlossen haben, wird die Taktik `simple_prover` als Verfeinerungstaktik ausgeführt und liefert in einem Schritt den vollständigen Beweis.

```
EDIT THM or_test
 * top
 ⊢ ∀x:ℕ. ∀y:ℕ. x<y ∧ (¬x<y ∨ 4=x) ⇒ 4=x

 BY simple_prover
```

An diesem Beweis kann man nur erkennen, daß die Taktik `simple_prover` in der Lage war, den Beweis alleine auszuführen. Solange man sich für die Details dieses Beweises nicht interessiert, ist diese Anwendungsform die beste Vorgehensweise, zumal sie immer noch den vollständigen Extraktterm (λ`x`.λ`y`. . . . liefert. Es ist allerdings auch möglich, `simple_prover` so ablaufen zu lassen, daß man jeden einzelnen Schritt, den diese Taktik ausgeführt hat, zu sehen bekommt.[21] Dies geschieht durch Aufruf des Editors für Transformationstaktiken

```
EDIT THM or_test
 # top
 ⊢ ∀x:ℕ. ∀y:ℕ. x<y ∧ (¬x<y ∨ 4=x) ⇒ 4=x

 BY <refinement rule>
```

```
EDIT Transformation Tactic

 simple_prover
```

Nachdem das Editorfenster geschlossen ist, wird `simple_prover` als Transformationstaktik ausgeführt. Dies liefert ebenfalls einen vollständigen Beweis, zeigt im Fenster aber nur den ersten Schritt an.

```
EDIT THM or_test
 * top
 ⊢ ∀x:ℕ. ∀y:ℕ. x<y ∧ (¬x<y ∨ 4=x) ⇒ 4=x

 BY all_i

 1* 1. x:ℕ
    ⊢ ∀y:ℕ. x<y ∧ (¬x<y ∨ 4=x) ⇒ 4=x
```

Den vollständigen Beweis kann man nun beim Durchlaufen des Beweises mit dem Beweiseditor ansehen oder mit der Transformationstaktik `PrintTexFile` *filename* in der vertrauten Kurzform (siehe Abbildung 4.2) als LaTeX-File darstellen lassen.

[21] Durch eine spezielle Funktion `Run`, welche Taktiken in elementare Regeln umwandelt, kann man allerdings verhindern, daß eine Taktik im Transformationsmodus aufgelöst wird. Dies ist sinnvoll, wenn man eine Taktik wie `simple_prover` nur in ihre Hauptbestandteile (die logischen "Regeln") zerlegen lassen will.

```
⊢ ∀x:IN. ∀y:IN. x<y ∧ (¬x<y ∨ 4=x) ⇒ 4=x
BY all_i
 \
  x:IN ⊢ ∀y:IN. x<y ∧ (¬x<y ∨ 4=x) ⇒ 4=x
  BY all_i
   \
    x:IN, y:IN  ⊢ x<y ∧ (¬x<y ∨ 4=x) ⇒ 4=x
    BY implies_i
     \
      x:IN, y:IN,  x<y ∧ (¬x<y ∨ 4=x) ⊢ 4=x
      BY and_e 3
       \
        x:IN, y:IN, x<y, ¬x<y ∨ 4=x  ⊢  4=x
        BY or_e 4
        |\
        | x:IN, y:IN, x<y, ¬x<y ⊢ 4=x
        | BY not_e 4
        |  \
        |   x:IN, y:IN, x<y, ¬x<y ⊢ x<y
        |   BY hypothesis 3
         \
          x:IN, y:IN, x<y, 4=x  ⊢ 4=x
          BY hypothesis 4
```

Abbildung 4.2: Von der Taktik `simple_prover` generierter Beweis

4.2.4 Programmierung von Taktiken

Wir wollen nun untersuchen, wie einfache Taktiken in ML geschrieben werden können, ohne daß man hierzu alle Details des NuPRL Systems verstehen muß.

Die Grundlage aller Taktiken ist die Funktion `refine`, mit der man die elementaren Regeln des NuPRL Systems in Taktiken umwandeln kann. Da alle im vorigen Kapitel vorgestellten Regeln der intuitionistischen Typentheorie bereits mittels `refine` in "Regel-Taktiken" umgewandelt wurden, ist eine explizite Verwendung der Funktion `refine` (und die damit verbundene Handhabung der Argumente einer Regel) für den normalen Anwender von NuPRL nicht erforderlich.[22] Er kann stattdessen davon ausgehen, daß die Inferenzregeln der Typentheorie als Taktiken vorliegen, und auf dieser Basis dann Spezialtaktiken zusammensetzen, die für seine Anwendungen besonders geeignet sind.

Auch hierfür ist es normalerweise nicht erforderlich, in die Details der Programmiersprache ML einzusteigen. Stattdessen können Anwender von NuPRL eine Reihe von vordefinierten *tacticals* benutzen, um bestehende Taktiken in neue Taktiken umzuwandeln. Tacticals sind nichts anderes als ML Funktionen des Typs `tactic -> tactic`, deren besonderer Zweck darin liegt, die imperative Denkweise bei der Anwendung von Regeln und Taktiken in einer funktionalen Programmiersprache auszudrücken.

Die häufigste Art, Taktiken zu kombinieren, ist die *Hintereinanderausführung*: um ein Ziel zu beweisen, wendet man eine gewisse Regel an, dann eine zweite auf die entstehenden Teilziele, dann eine dritte auf deren Resultate usw. Dieser Effekt kann mit dem Tactical `THEN` erreicht werden, welches als Funktion in *Infix-*

[22] Bei dieser Umwandlung wird insbesondere die automatische Umbenennung von Variablennamen durchgeführt, die in der Regel nicht direkt enthalten ist. Stattdessen verlangt die Regel die Angabe des Variablennamens als Parameter. Mithilfe einer Funktion `new : tok -> proof -> tok`, welche überprüft, ob eine bestimmte Variable bereits im Beweis deklariert ist und diese dann ggf. umbenennt, kann man die Regel-Taktik `lambdaI` dann z.B. wie folgt implementieren.

```
let lambdaI pf =
    let [id,S;(),T] = bound_terms_of_term (conclusion pf)
    in
      refine (make_primitive_rule `lambdaFormation` [make_level_expression_argument level; new id pf]) pf
;;
```

Da das Schema der Umwandlung bei allen Regeln gleich ist, enthält die tatsächliche Implementierung aller Regeltaktiken einige Hilfsfunktionen, welche die eben beschriebene Umwandlung eleganter durchführen.

- t_1 `THEN` t_2: *"Wende t_2 auf alle von t_1 erzeugten Teilziele an"*
  t `THENL` $[t_1; t_2; ..t_n]$: *"Wende t_i auf das i-te von t erzeugte Teilziel an"*
- t_1 `ORELSE` t_2: *"Wende t_1 an. Falls dies fehlschlägt, wende t_2 an".*
- `Repeat` t: *"Wiederhole die Taktik t bis sie fehlschlägt"*
- `Complete` t: *"Wende t nur an, wenn hierdurch der Beweis vollständig wird"*
- `Progress` t: *"Wende t nur an, wenn ein Fortschritt erzielt wird"*
- `Try` t: *"Wende t an; falls dies fehlschlägt, lasse das Ziel unverändert"*

Abbildung 4.3: Wichtige vordefinierte Tacticals

Notation vordefiniert ist. Eine Variante von `THEN` ist das Tactical `THENL`, die eine individuellere Handhabung der entstandenen Teilziele ermöglicht, indem man zu jedem der entstehenden Teilziele eine eigene Regel, insgesamt also eine Liste von Taktiken angibt. Neben der Programmierung neuer Taktiken unterstützen diese beiden Tacticals besonders auch Beweise, in denen man den Effekt der nächsten Schritte vorausplanen und zu einem Schritt zusammenfassen will. Da man hierbei oft auch nur einige der Teilziele automatisch weiterverarbeiten will (z.B. nur Wohlgeformtheitsziele), wurde eine spezielle Taktik `Id` vordefiniert, die ein Teilziel unverändert läßt. Ihre Implementierung ist denkbar einfach, da nur die Typstruktur verändert werden muß.

```
let Id (pf:proof) = [pf],hd;;
```

Beispiel 4.2.4 (Kombination von Taktiken durch Hintereinanderausführung)

Ein typisches Beispiel für eine Taktik, die nur durch Hinterausführung von Taktiken (Regeln) programmiert werden kann, ist die in Beispiel 4.2.1 auf Seite 200 angesprochene Taktik `cases`. Sie besteht darin, erst eine Disjunktion per `cut` einzuführen und dann das zweite Teilziel mit `or_e` zu zerlegen:

```
let cases A B =
        cut (-1) (make_or_term A B)
    THENL [Id ; or_e (-1)]
;;
```

Für ein mehr experimentelles Vorgehen ist das (Infix-)Tactical `ORELSE` da. Es ermöglicht, die Anwendung einer Taktik t_1 zu versuchen und eine zweite Taktik t_2 zu starten, wenn die Anwendung der ersten fehlschlägt. Dies wird insbesondere dann eingesetzt, wenn man sich über die genaue Struktur der zu erwartenden Ziele nicht ganz im klaren ist, sondern nur weiß, das eine der angegebenen Taktiken anwendbar sein wird. `ORELSE` ist somit für das weitestgehend automatische Suchen nach Beweisen ein entscheidendes Hilfsmittel.

Das Tactical `Repeat` wendet eine Taktik t solange an, bis sie fehlschlägt. Die Ausnahmen, die durch das Fehlschlagen der Taktik t auf einigen Unterzielen entstehen, werden dabei abgefangen. Hiermit kann man zum Beispiel eine einfache Taktik `repeatedIntro` schreiben, welche logische Einführungsregeln solange anwendet, bis das eigentlich interessante Teilziel offengelegt ist.

```
let repeatedIntro =
    Repeat (        all_i
             ORELSE imp_i
             ORELSE not_i
             ORELSE and_i
            )
;;
```

Diese Taktik übernimmt also die langwierigen trivialen Schritte eines Beweises, die ohnehin naheliegend sind.

Die wichtigsten weiteren einfachen Tacticals, die sich als nützlich für die Erstellung neuer Taktiken herausgestellt haben, sind `Complete`, `Progress` und `Try`. Abbildung 4.3 faßt ihre Bedeutung kurz zusammen. Weitere interessante Tacticals sind in [Jackson, 1993b, Kapitel 8.3] zusammengestellt. Die Implementierung dieser Tacticals ist nicht sehr schwierig, wie das folgende Beispiel zeigt.

Beispiel 4.2.5

Die Tacticals `THENL` und `ORELSE` müssen als Infix-Operatoren vereinbart werden (wozu die Funktion `ml_curried_infix` bereitsteht) und werden aus Gründen der Übersichtlichkeit groß geschrieben.

Die Programmierung von `THENL` ist nichts anderes als ein sorgfältiges Zusammensetzen von Teilzielen und Validierungsfunktionen. t `THENL` $[t_1;t_2;..t_n]$ muß als Teilziele die Taktiken t_i auf die Resultate der Anwendung von t anwenden, wozu eine elementare Listenfunktion `map_apply` benutzt werden kann. Das Resultat ist eine Liste von Listen von Beweiszielen, die "flach" gemacht werden muß. Die Validierungen der t_i finden eine solche flache Liste vor und müssen nun auf die jeweils passende Teilliste angewandt werden um eine Liste von Beweisen zu erzeugen, auf die dann die Validierung von t angewandt wird.

`ORELSE` wird mit dem Ausnahmebehandlungsmechanismus von ML programmiert. Hierbei ist allerdings zu beachten, daß nur die *Anwendung* einer Taktik eine Ausnahme erzeugt, nicht aber die Taktik selbst. Statt t_1 `?` t_2 muß man daher `\pf.` t_1 `pf ?` t_2 `pf` schreiben. Für die Programmierung von `Complete` muß man nur testen, ob nach der Anwendung der Taktik noch zu beweisende Teilziele vorhanden sind. In diesem Fall muß eine Ausnahme ausgelöst werden.

```
ml_curried_infix `THENL` ;;
ml_curried_infix `ORELSE` ;;

let $THENL (tac: tactic) (tac_list : tactic list) (pf:proof) =
  let subgoals, val = tac pf
  in
     if not length tac_list = length subgoals
        then fail
        else let subgoalLists, valList = map_apply tac_list subgoals
             in
                (flatten subgoalLists),
                \proofs. val ( (mapshape (map length subgoalLists) valList) proofs)
;;
let $ORELSE (t1:tactic) (t2:tactic) pf =  t1 pf ? t2 pf ;;

let Complete (tac:tactic) (pf:proof) = let subgoals, val = tac pf
                                       in
                                          if null subgoals
                                             then subgoals, val
                                             else fail
                                       ;;
```

Die Programmierung der anderen in Abbildung 4.3 genannten Tacticals ist eine einfache Übung.

Natürlich steht es einem Anwender auch frei, Taktiken durch wesentlich komplexere ML-Programme zu erzeugen als nur durch die Anwendung von Tacticals. Letztere erleichtern eine übersichtliche Programmierung jedoch ungemein, so daß auch trickreichere Taktiken zu einem Teil auf Tacticals zurückgreifen werden. Auch hierzu wollen wir ein Beispiel geben.

Beispiel 4.2.6

Die Taktik `hypcheck` überprüft, ob ein Beweisziel unmittelbar aus einer der Hypothesen folgt. Dies kann geschehen, indem man die Regel `hypothesis` auf alle Hypothesen des Beweises anwendet. Da die bisher angegebenen Tacticals hierfür nicht weiterhelfen, muß hierfür eine rekursive Taktik geschrieben werden.

```
let hypcheck (pf:proof)  =
  let n = length (hypotheses pf)
  in
      letrec TryHyp pos =
        if pos<=n  then hypothesis pos ORELSE TryHyp (pos+1)
                   else Id
      in
         TryHyp 1 pf
;;
```


Dieses Durchprobieren aller Hypothesen auf Anwendbarkeit einer Taktik ist auch in anderen Fällen sinnvoll. Aus diesem Grund lohnt es sich, die obige Implementierung zu einem neuen Tactical `TryAllHyps (tac: int -> tactic) (pf:proof)` zu verallgemeinern, welches anstelle der Regel `hypothesis` eine beliebige Taktik anwendet. Dieses Tactical findet zum Beispiel bei der Programmierung der Taktik `simple_prover` in Abbildung 4.4 mehrfach Verwendung.

Nur mit den vordefinierten Tacticals, den (umgewandelten) NuPRL Regeln und wenigen vordefinierten Taktiken kann ein Anwender des NuPRL Systems bereits bemerkenswert leistungsfähige Taktiken programmieren. Ein sehr interessantes Beispiel hierfür ist die Taktik `simple_prover`, die in der Lage ist, viele einfache logische Schlüsse selbständig auszuführen.

Beispiel 4.2.7

Die Taktik `simple_prover`, deren ML-Implementierung in Abbildung 4.4 angegeben ist, spiegelt die in Abschnitt 2.2.9 gegebenen Richtlinien für das Führen logischer Beweise wieder, welche die Prioritäten der Regeln nach den Kosten ihrer Anwendung sortiert.

Zuerst wird versucht, Taktiken anzuwenden, die keinerlei Teilziele erzeugen und somit das Beweisziel abschließen oder sofort fehlschlagen. Die Elimination von Konjunktionen und Existenzquantoren sorgt nur für eine feinere Auflösung der Hypothesenliste und ist somit gefahrlos (schlimmstenfalls hätte die Hypothese in einem *späteren* Teilziel als ganzes verwendet werden können und muß dann wieder zusammengebaut werden). Die Einführung von Allquantor, Implikation, Negation und Konjunktion ist ebenfalls nur mit geringen Kosten verbunden. Eine Auflösung einer Disjunktion in einer Hypothese erzeugt zwei Teilziele und verdoppelt die Beweislast (was teuer ist, wenn die Disjunktion keine Rolle spielt). Rückwärtsschließen über Negationen und Implikationen wird nur mit einer gewissen Vorsicht einzusetzen sein.

Die Einführung von Existenzquantoren und die Elimination universell quantifizierter Formeln erfordert eine gewisse Vorausschau, welcher Term zur Instantiierung der Variablen benötigt wird, und wird deshalb nicht mit aufgenommen. Aus einem ähnlichen Grund wird auf die Regeln `or_i1` und `or_i2` verzichtet, da diese den Beweis in eine falsche Richtung führen können und nur bewußt eingesetzt werden sollten.

Auch die Programmierung von echten Transformationstaktiken ist nicht sehr aufwendig. Wir wollen dies am Beispiel der Taktiken `Mark` und `Copy` illustrieren, die dazu benutzt werden können, Teilbeweise zu sichern und zu kopieren.

Beispiel 4.2.8

Die Taktik `Mark` legt eine direkte Kopie des aktuellen Teilbeweises unter einem angegebenen Namen in einer Tabelle `saved_proofs` ab, die als globale Variable vom Typ `(tok#proof) list` deklariert ist. Neue Einträge in diese Liste geschehen mit der Funktion `add_saved_proof` und mit `get_saved_proof` kann man den unter einem Namen abgelegten Teilbeweis wieder auffinden.

Da das Ablegen der Kopie alleine keine Taktik darstellt, muß zugunsten der Typkorrektheit im Anschluß daran die leere Taktik `Id` angewandt werden. Hierzu benutzt `Mark` ein imperatives Merkmal – die sequentielle Auswertung von Funktionen.

`Copy` durchläuft den abgesicherten Beweis rekursiv und benutzt die jeweilige `refinement` Komponente, um den Namen der Regel des Beweises zu erzeugen. Da dies fehlschlagen wird, wenn der abgelegte Teilbeweis unvollständig war, muß vorher abgefragt werden, ob überhaupt eine `refinement` Komponente existiert. Ist dies nicht der Fall, so ist die Taktik `Id` aufzurufen.

```
let Mark name pf = add_saved_proof name pf; Id pf ;;

letrec copy_pattern pattern pf =
  if is_refined pattern
    then Try ( refine (refinement pattern) THENL (map copy_pattern (children pattern)) ) pf
    else Id pf
;;
let Copy name  = copy_pattern (get_saved_proof name)
```

```
%-----------------------------------------------------------------------+
| PREDEFINED TACTICALS                                                  |
|                                                                       |
|  TryAllHyps: (int -> tactic) -> tactic[23]                            |
|            Try to apply a given tactic with to the hypotheses of the proof|
|                                                                       |
|  Chain: (* -> tactic) -> tactic -> tactic                             |
|            Try to apply a given tactic repeatedly to all the hypotheses |
|            of the proof or apply a given basetactic. Chaining is limited |
|            to a certain number of steps and must complete a proof.     |
|                                                                       |
|  Run:      convert a tactic into a rule to make it primitive. This    |
|            prevents transformation tactics from showing unwanted details. |
|                                                                       |
+-----------------------------------------------------------------------+
|                                                                       |
|   hypcheck:          check whether the goal follows from a hypothesis |
|   contradiction:     try to find a contradictory hypothesis           |
|   conjunctionE:      eliminate all conjunctions in hypotheses         |
|   disjunctionE:      eliminate all disjunctions in hypotheses         |
|   impChain:          chain several hypotheses in order to completely  |
|                      prove the goal                                   |
|                                                                       |
|   simple_prover:     A tactic trying to apply simple steps in increasing |
|                      order of costs                                   |
|                                                                       |
+-----------------------------------------------------------------------%

let hypcheck          = TryAllHyps hypothesis             ;;
let contradiction     = TryAllHyps false_e                ;;
let conjunctionE      = TryAllHyps and_e                  ;;
let existentialE      = TryAllHyps ex_e                   ;;
let disjunctionE      = TryAllHyps or_e                   ;;
let impChain          = Chain impE hypcheck               ;;
let notChain          = TryAllHyps not_e    THEN impChain ;;

let nondangerousI pf  = let termkind = opid (conclusion pf)
                        in
                           if member termkind [`all`; `not`; `implies`;
                                                `rev_implies`; `iff`; `and`]
                              then Run (termkind ^ `_i`) pf
                              else fail
                        ;;

let simple_prover    = Repeat
                        (       hypcheck
                         ORELSE contradiction
                         ORELSE conjunctionE
                         ORELSE existentialE
                         ORELSE nondangerousI
                         ORELSE disjunctionE
                         ORELSE notChain
                         ORELSE impChain
                        );;
```

Abbildung 4.4: Implementierung der Taktik `simple_prover`

Die tatsächliche Implementierung von `simple_prover` wurde mittlerweile noch weiter verfeinert und im Hinblick auf Effizienz optimiert. Das Tactical `TryALlHyps` nimmt nun zum Beispiel eine Vorselektierung vor, bei der Hypothesen ignoriert werden, auf welche die genannte Taktik prinzipiell nicht anwendbar ist. Diese Selektion ist effizienter als eine fehlschlagende Taktik abzufangen.

4.2.5 Korrektheit taktisch geführter Beweise

Wir haben bisher an vielen Beispielen die Vielfalt der Einsatzmöglichkeiten von Taktiken im automatischen Beweisen illustriert. Da einem Benutzer von taktisch gesteuerten Beweisentwicklungssystemen zur Programmierung von Taktiken der gesamte Sprachumfang einer Turing-mächtigen Programmiersprache (in unserem Fall der von ML) zur Verfügung steht, gibt es praktisch keinerlei Restriktionen beim Schreiben von Taktiken. Dies wirft natürlich die Frage auf, wie zuverlässig taktisch geführte Beweise bei derartigen Freiheitsgraden noch sein können. Solange das Beweissystem rein interaktiv war, folgte die Zuverlässigkeit eines Beweisers aus der korrekten Implementierung der Funktion `refine` und der Korrektheit der einzelnen Regeln. Es ist aber auch bekannt, daß bei *automatischen* Theorembeweisern die gesamte Beweisprozedur verifiziert werden muß, wenn man garantieren will, daß ein maschinell geführter Beweis auch tatsächlich korrekt ist. Ist dies nun für Beweistaktiken ebenfalls nötig?

Zum Glück kann diese Frage eindeutig mit "nein" beantwortet werden, da der Freiheitsgrad beim Programmieren von Taktiken in einem Punkt doch eingeschränkt ist, nämlich im Bezug auf die Möglichkeiten, einen Beweis zu manipulieren. Da Beweise gemäß Abschnitt 4.1.2 Elemente eines *abstrakten* Datentyps sind, können sie ausschließlich mithilfe der Funktion `refine` verändert werden. Im Endeffekt müssen daher alle Taktiken auf die Funktion `refine` und die vordefinierten Regeln des Beweissystems zurückgreifen. Eine andere Möglichkeit, Beweise zu manipulieren, gibt es nicht. Somit wird auf eine sehr einfache Art der Implementierung sichergestellt, daß jeder Beweis, der durch eine Taktik – sei es nun eine Verfeinerungstaktik oder eine Transformationstaktik – erzeugt wird, auch korrekt ist.

Satz 4.2.9

> *Das Resultat einer Taktik-Anwendung auf ein Beweisziel ist immer ein gültiger Beweis des zugrundeliegenden logischen Kalküls.*[24]

Dieser Satz erlaubt dem Anwender eines taktisch gestützten Beweisssystems, das Inferenzsystem des zugrundeliegenden Kalküls nach Belieben um eigene Beweisverfahren zu ergänzen, *ohne sich dabei Gedanken über die Korrektheit seiner Implementierungen zu machen*. Die Tatsache, daß Taktiken im Endeffekt auf den Regeln des Kalküls aufbauen, macht die benutzerdefinierten Erweiterungen automatisch konsistent mit dem Rest der Theorie. Das Schlimmste, was passieren könnte, wäre daß eine Taktik kein Resultat liefert. Da aber jedes durch Taktiken erzielte Ergebnis in jedem Falle korrekt ist, unterstützen Taktiken in besonderem Maße das Experimentieren mit neuen Ideen und Strategien in einer Art, die keinerlei Eingriffe in das eigentliche Beweissystem verlangt. In einem gewissen Sinne sind Taktiken also *konservative Erweiterungen* des Systems der Inferenzregeln und bieten zusammen mit Abstraktionen einen sicheren Weg, die praktische Ausdruckskraft des Kalküls beliebig zu erweitern.

4.2.6 Erfahrungen im praktischen Umgang mit Taktiken

Der Taktik-Mechanismus hat sich im Laufe der Jahre als ein sehr erfolgreiches Hilfsmittel zur Einbettung einfacher Beweisstrategien in ein interaktives Beweisentwicklungssystem erwiesen. Viele Anwender des NuPRL Systems haben eine Reihe universeller (siehe vor allem [Howe, 1988] und [Jackson, 1993b, Kapitel 8]) und anwendungsspezifischer Taktiken geschrieben und dazu benutzt, um bestimmte mathematische Teilgebiete formal zu verifizieren [Howe, 1986, Howe, 1987, Howe, 1988, Kreitz, 1986, Basin, 1989, de la Tour & Kreitz, 1992]. Der Taktik-Mechanismus hat hierbei ermöglicht, in wenigen Tagen die gleichen Strategien zu realisieren, deren Implementierung in anderen Systemen mehrere Wochen intensive Arbeit gekostet hatte, weil Eingriffe in das eigentliche Beweissystem vorgenommen werden mußten. Taktiken sind leichter zu verstehen, da sie nahezu direkt auf der Ebene der Objekttheorie operieren, und können somit auch leichter modifiziert und zusammengesetzt werden als direktere Implementierungen von Strategien.

[24] Dies setzt natürlich voraus, daß die Regeln des Kalküls und die Funktion `refine` korrekt implementiert wurden. Wenn dies aber nicht der Fall wäre, dann wäre das gesamte System nutzlos.

Besonders wichtig ist, daß Taktiken bei aller Flexibilität ein hohes Maß an Sicherheit bieten und sich somit für Experimente mit neuen strategischen Ideen besonders eignen. Insbesondere öffnen sie eine Möglichkeit, alle bekannten Verfahren des automatischen Beweisens und der Programmsynthese in *ein einziges Beweissystem* zu integrieren. Diese Verfahren können nämlich als eine Art *Beweisplaner* verstanden werden, deren Resultate dann zur Steuerung der Beweisregeln verwandt werden kann.[25] Der wachsende Erfolg taktischer Beweissysteme (siehe Fußnote 3 auf Seite 182), insbesondere der des universellen generischen Systems ISABELLE demonstriert in besonderem Maße die praktischen Vorteile des Taktik-Konzeptes gegenüber Systemen mit festeingebauten Strategien.

Im Prinzip gibt es keine Grenzen für das, was man mit Taktiken programmieren kann. Die Versuchung, Taktiken zu schreiben, die möglichst alle Arbeiten automatisch durchführen, ist daher sehr groß. Dennoch sollte man sich darauf beschränken, Taktiken zu programmieren, deren Effekte überschaubar und kontrollierbar[26] sind, da sehr universelle Taktiken oft Irrwege einschlagen und Beweise erheblich größer werden lassen, als dies der Fall sein müßte. Es ist ratsam, stattdessen kleine und effiziente Taktiken mit einem sehr begrenzten Anwendungsbereich zu schreiben und diese gezielt einzusetzen. Hierbei ist jedoch auf eine sinnvolle und leicht verständliche Namensgebung zu achten, da ansonsten eine unüberschaubare Fülle von Taktiken die Auswahl einer guten Taktik in einer konkreten Situation erschwert.

Bei komplexen Anwendungen kann die Anwendung von Taktiken recht ineffizient werden, da sie alle elementaren Beweisschritte einzeln ausführen müssen. In diesem Falle erweist es sich als sinnvoll, die wichtigsten Eigenschaften der betrachteten Objekte zunächst als Lemmata (Theoreme der Bibliothek) zu verifizieren und dann in Taktiken auf diese Lemmata zurückzugreifen. Damit reduziert sich die Beweislast zur Laufzeit der Taktik auf eine Instantiierung universell-quantifizierter Variablen des Lemmas, während die Hauptlast ein für alle Mal bei dem Beweis des Lemmas durchgeführt wurde. Diese Vorgehensweise steigert nicht nur die Effizienz der Beweisführung sondern auch die Verständlichkeit der erzeugten Beweise, da diese nun in großen konzeptionellen Schritten strukturiert werden können.[27] Somit kann der Taktik Mechanismus, gekoppelt mit Abstraktionen und Theoremen der Bibliothek dazu benutzt werden, das Niveau des logischen Schließens schrittweise von den Grundbestandteilen des zugrundeliegenden Kalküls auf die typische Denkweise der zu bearbeitenden Probleme anzuheben und somit die formal korrekte Lösung komplexerer Anwendungen erheblich zu erleichtern.

4.3 Entscheidungsprozeduren – automatische Beweisführung

Im formalen Schließen tauchen relativ oft Teilprobleme auf, die im Prinzip leicht zu beweisen sind, weil sie auf bekannten mathematischen Erkenntnissen beruhen. Dennoch ist ihr formaler Beweis selbst beim Einsatz von Taktiken oft sehr aufwendig obwohl er keinerlei algorithmische Bedeutung besitzt (also im wesentlichen Axiom als Extraktterm liefert). Das Hauptinteresse bei der Bearbeitung solcher Probleme liegt also darin, herauszufinden, ob die aufgestellte Behauptung wahr ist oder nicht, während die einzelnen Beweisschritte keine signifikante Bedeutung[28] besitzen.

Es ist daher durchaus sinnvoll, für manche Problemstellungen schnelle Algorithmen zu entwerfen, welche diese in einer Weise entscheiden, die für einen normalen Benutzer nicht unbedingt einsichtig ist. Derartige *Entscheidungsprozeduren* beruhen meist auf einer maschinennahen Charakterisierung für die Gültigkeit der

[25] Dieses Konzept wird sehr erfolgreich von dem System OYSTER [Bundy, 1989, Bundy *et.al.*, 1990] verfolgt.

[26] Nicht alle in Taktiken, die derzeit in NuPRL integriert sind, erfüllen diese Bedingungen. Zugunsten einer einfacheren Handhabung enthält das System einige Taktiken, die Wohlgeformtheitsziele möglichst vollständig abhandeln. Wenn diese allerdings fehlschlagen, findet der Benutzer oftmals recht unverständliche Teilziele vor.

[27] Die Taktiken zur Instantiierung von Lemmata in [Jackson, 1993b, Kapitel 8.5] und die in [Jackson, 1993b, Kapitel 8.8]) beschriebenen Taktiken zur automatischen Konversion von Beweiszielen mit Gleichheitslemmata unterstützen diese Methodik.

[28] Sie tragen weder zum Extraktterm des Gesamtbeweises bei noch liefern sie neue mathematische Einsichten, da es sich um ein längst erforschtes Teilgebiet handelt.

Behauptung, die durch eine grundlegenden allgemeinen Analyse der Problemstellung zustandegekommen ist[29] und sich leichter überprüfen läßt, als die ursprüngliche Behauptung. Nach außen hin erscheinen sie als eine einzige ausgeklügelte Inferenzregel, die Axiom als Extraktterm liefert, aber ansonsten nicht mehr durch ein einfaches Schema beschrieben werden kann.

Es ist offensichtlich, daß eine solche Entscheidungsprozedur nur dann zu einem Beweissystem hinzugenommen werden darf, wenn ihre Konsistenz mit den anderen Regeln der zugrundeliegenden Theorie bewiesen werden kann. Da die Typentheorie und viele darin enthaltene Teiltheorien (wie die Prädikatenlogik oder das Induktionsbeweisen) nachweislich unentscheidbar sind, müssen wir uns auf Entscheidungsprozeduren für Teiltheorien beschränken, die bekanntermaßen entscheidbar sind und eine wichtige Rolle in der Praxis des mathematischen Schließens spielen. Da Entscheidungsprozeduren aufgrund ihrer Verwendung innerhalb von interaktiven Beweissystemen sehr schnell arbeiten sollen, muß außerdem sehr leicht feststellbar sein, ob die Prozedur überhaupt anwendbar ist und welche Hypothesen für sie relevant sind. Dies schränkt die Menge der Teiltheorien, für die es sinnvoll ist, Entscheidungsprozeduren zu entwerfen, relativ stark ein.

In diesem Abschnitt werden wir zwei Entscheidungsprozeduren vorstellen, die erfolgreich zur Typentheorie von NuPRL hinzugefügt werden konnten und daran illustrieren, welche Schritte bei der Entwicklung einer Entscheidungsprozedur eine Rolle spielen.

4.3.1 Eine Entscheidungsprozedur für elementare Arithmetik

Die Erfahrung im Umgang mit Beweissystemen hat gezeigt, daß praktische Arbeiten ohne eine arithmetische Entscheidungsprozedur so gut wie undurchführbar sind. Mangels einer solchen Prozedur ist das LCF Projekt [Gordon *et.al.*, 1979], welches sich das Schließen über Berechnungen zum Ziel gesetzt hatte, ebenso in seinen Anfängen steckengeblieben wie das Projekt AUTOMATH [Bruijn, 1980, van Benthem Jutting, 1977], in dem mathematische Beweise formalisiert und automatisch überprüft werden sollten: es dauerte über 5 Jahre intensiver Arbeit bis man die Ringaxiome und einige andere elementare Eigenschaften der reellen Zahlen beweisen konnte, nur weil für jeden dieser Beweise eine extrem große Anzahl von Beweissschritten nötig war.

Warum nun ist eine Entscheidungsprozedur die Arithmetik so bedeutend? Im wesentlichen kann man vier Ursachen dafür finden.

1. In fast allen praktisch relevanten Beweisen spielt arithmetisches Schließen eine Rolle.
2. Ein Großteil der arithmetischen Schlüsse ist absolut trivial, was die gewonnenen Erkenntnisse angeht.
3. Ein und dieselbe arithmetische Aussage kann in einer unglaublichen Vielfalt von Erscheinungsformen auftauchen. So sind zum Beispiel `x+1<y, 0<t` $\vdash$ `(x+1)*t < y*t` und `x<y, 0<t` $\vdash$ `x*t < y*t` syntaktisch völlig verschiedene Behauptungen obwohl die zweite Aussage eine einfache Variante der ersten ist.
4. Der formale Beweis einer simplen arithmetischen Aussage mit Hilfe der elementaren Inferenzregeln der ganzen Zahlen ist keineswegs so einfach, wie es auf den ersten Blick erscheinen mag. So ist die Aussage

   Wenn drei ganze Zahlen sich jeweils um maximal 1 unterscheiden, dann sind zwei von ihnen gleich

   intuitiv besehen absolut einsichtig, kann formal aber nur relativ aufwendig bewiesen werden.

Es gibt also gute Gründe dafür, eine Entscheidungsprozedur für die Arithmetik zu entwerfen, die derartige Probleme in *einem* Inferenzschritt lösen kann und nach außen wie eine einfache Inferenzregel erscheint. Bevor wir dies tun können, müssen wir allerdings die theoretischen Grenzen einer solchen Entscheidungsprozedur untersuchen und eine maschinennahe Charakterisierung für die Gültigkeit arithmetischer Aussagen aufstellen

[29] So kann man die Gültigkeit einer arithmetischen Aussage auf ein graphentheoretisches Problem zurückführen (siehe Abschnitt 4.3.1, und die Gültigkeit einer logischen Aussage durch eine Matrixcharakterisierung [Bibel, 1983, Bibel, 1987] beschreiben. Auch für geometrische Probleme gibt es eine sehr effiziente Charakterisierung [Wu, 1986, Chou & Gao, 1990].

und als korrekt nachweisen. Auch der Algorithmus selbst muß schließlich als korrekt nachgewiesen werden, da wir sonst keinerlei Veranlassung mehr hätten, einem mechanisch geführten Beweis zu vertrauen. Wir werden in diesem Abschnitt eine relativ kurze und wenig formale Beschreibung der Entscheidungsprozedur `arith` geben, die in das NuPRL-System integriert ist und verweisen auf [Constable *et.al.*, 1982][30] für weitere Details.

Welche Art von Problemen kann man mit `arith` lösen und welche nicht? Wir wollen hierzu ein paar Beispiele geben.

Beispiel 4.3.1
Die folgenden Problemstellungen können wie folgt mit `arith` gelöst werden.

Γ, i: `0<x`, Δ	$\vdash$ `0<x+2`	**by** `arith`
Γ, i: `0<x`, Δ	$\vdash$ `0<x*x`	**by** `arith` i `*` i
Γ, i: `x+y`$\leq$`z`, Γ', j: `y`$\geq$`1`, Δ	$\vdash$ `x<z`	**by** `arith` i `-` j
Γ, i: `x`$\leq$`y`, Γ', j: `x`$\neq$`y`, Δ	$\vdash$ `x<y`	**by** `arith`
Γ	$\vdash$ `x-5 < x+10`	**by** `arith`
Γ, i: `x<x*x`, Γ', j: `x`$\neq$`0` , Δ	$\vdash$ `x`$\geq$`2` $\vee$ `x<0`	**by** `arith` i $\div$ j
Γ, i: `x<y`, Γ', j: `0<z`, Δ	$\vdash$ `x*z<y*z`	**by** `arith` i `*` i
Γ, i: `x+y>z`, Γ', j: `2*x`$\geq$`z`, Δ	$\vdash$ `3*x+y` $\geq$ `2*z-1`	**by** `arith` i `+` i

All diese Probleme wären ohne `arith` nur sehr aufwendig zu beweisen und zeigen die praktischen Vorteile dieser Prozedur. Es gibt jedoch auch Grenzen für die Einsatzmöglichkeiten arithmetischer Entscheidungsprozeduren, da nicht alle arithmetischen Probleme entscheidbar sind.

Beispiel 4.3.2 (Das 10. Hilbertsche Problem)
Ist f eine *beliebige* berechenbare Funktion auf den ganzen Zahlen, so gibt es keine Möglichkeit zu entscheiden, ob f eine Nullstelle besitzt oder nicht. Das Problem

$$\exists \mathtt{x}_1,\ldots,\mathtt{x}_n{:}\mathbb{Z}.\ \mathtt{f}(\mathtt{x}_1,\ldots,\mathtt{x}_n)\mathtt{=0}$$

ist nicht durch ein universelles Verfahren (das keine Spezialkenntnisse über f besitzt) zu entscheiden.

Dieses Beispiel zeigt, daß wir eine Entscheidungsprozedur für die gesamte Arithmetik nicht konstruieren können. Wir können also bestenfalls erwarten, eine *eingeschränkte Theorie der Arithmetik* entscheiden zu können, die aber immer noch die meisten realen Probleme erfassen kann. Arithmetische Probleme, die mit den Mitteln dieser eingeschränkten Theorie nicht mehr beschrieben werden können, müssen auf andere Weise (z.B. durch eine Taktik) behandelt werden. Dies ist aus praktischer Hinsicht aber kein gravierender Nachteil, da wir vor allem ja die trivialen Fälle durch eine Entscheidungsprozedur lösen wollen und nicht etwa anstreben, einen universellen Beweiser zu konstruieren. Wir werden im folgenden also eine eingeschränkte Theorie der Arithmetik ohne Induktion aufstellen, die mit Sicherheit entscheidbar ist. Für diese Theorie werden wir eine Charakterisierung der Gültigkeit aufstellen und auf dieser Basis dann eine Entscheidungsprozedur angeben.

Abbildung 4.5 beschreibt die komplette Syntax und Semantik einer eingeschränkten arithmetischen Theorie, die wir im folgenden als *Theorie der elementaren Arithmetik* $\mathcal{A}$ bezeichnet. Zur Vereinfachung haben wir die Theorie als *quantorenfreie Arithmetik* formuliert und gehen davon aus, daß alle vorkommenden Variablen all-quantifiziert und vom Typ $\mathbb{Z}$ sind.[31] Es ist leicht einzusehen, daß $\mathcal{A}$ eine Teiltheorie der intuitionistischen Typentheorie ist und somit von NuPRL verarbeitet werden kann.

[30] Die `arith`-Prozedur ist 1976 im Zusammenhang mit dem PL/CV System entstanden und erfolgreich in einer früheren Version von PRL (λ-PRL) eingesetzt worden, die noch nicht auf Typentheorie basierte. Eine ausführliche Beschreibung und einen Korrektheitsbeweis liefert der Artikel "*An algorithm for checking PL/CV arithmetic inferences*" von Tat-hung Chan, der im Appendix von [Constable *et.al.*, 1982] wiedergegeben ist. Die Einbettung in NuPRL verlangte eine Reihe kleinerer Anpassungen, die in diesem Artikel nicht erwähnt sind, aber keine signifikante Rolle spielen.

Man beachte, daß alle Beweise, die mit `arith` gelöst werden können, im Prinzip auch mit den normalen Regeln der Typentheorie bewiesen werden könnten, wobei man natürlich spezielle Regeln für Konstantenarithmetik und alle anderen explizit definierten Ausdrücke benötigen würde (die ja nicht mehr auf Nachfolger und Induktion abgestützt sind). Die Hinzunahme von `arith` anstelle eines "konventionellen" Regelsatzes geschieht aus rein pragmatischen Gründen und verändert die Typentheorie als solche überhaupt nicht.

[31] In der `arith`-Prozedur wirkt sich das so aus, daß alle symbolischen Ausdrücke, die bei der Verarbeitung wie Variablen behandelt werden, nach erfolgreicher Ausführung von `arith` als vom Typ $\mathbb{Z}$ nachgewiesen werden müssen.

Formale Sprache: Die formale Sprache von $\mathcal{A}$ besteht aus elementar-arithmetischen Formeln:

- *Terme* sind NuPRL-Terme, die nur aus ganzzahligen Konstanten, Variablen und den Operatoren `+`, `-`, `*` aufgebaut sind.
- *Atomare Formeln* sind Terme der Form $t_1\,\rho\,t_2$, wobei t_1 und t_2 Terme sind und ρ einer der arithmetischen Vergleichsoperatoren $=,\neq,<,>,\leq,\geq$ ist.
- *elementar-arithmetische Formeln* sind alle Ausdrücke, die sich aus atomaren Formeln und den aussagenlogischen Konnektiven $\neg$, $\wedge$, $\vee$ und $\Rightarrow$ aufbauen lassen.

Semantik: Die Semantik von $\mathcal{A}$ wird durch (eingeschränkte) Gleichheitsaxiome und die gewöhnlichen Axiome der Zahlentheorie charakterisiert. Im einzelnen sind dies:

Gleichheitsaxiome. Für alle ganzen Zahlen `x,y,z` gilt:

1. `x=x` (Reflexivität)
2. `x=y` $\Rightarrow$ `y=x` (Symmetrie)
3. `x=y` $\wedge$ `y=z` $\Rightarrow$ `x=z` (Transitivität)
4. `x=y` $\wedge$ `x`$\,\rho\,$`z` $\Rightarrow$ `y`$\,\rho\,$`z` `x=y` $\wedge$ `z`$\,\rho\,$`x` $\Rightarrow$ `z`$\,\rho\,$`y` (eingeschränkte Substitutivität)[a]

Axiome der Konstantenarithmetik wie `1+1=2, 2+1=3, 3+1=4, ...`[b]

Ringaxiome der ganzen Zahlen. Für alle ganzen Zahlen `x,y,z` gilt:

1. `x+y=y+x` `x*y=y*x` (Kommutativgesetze)
2. `(x+y)+z=x+(y+z)` `(x*y)*z=x*(y*z)` (Assoziativgesetze)
3. `x*(y+z)=(x*z)+(y*z)` (Distributivgesetz)
4. `x+0=x` `x*1=x` (Neutrale Elemente)
5. `x+(-x)=0` (Inverses Element der Addition)
6. `x-y=x+(-y)` (Definition der Subtraktion)

Axiome der diskreten linearen Ordnung. Für alle ganzen Zahlen `x,y,z` gilt:

1. $\neg$`(x<x)` (Irreflexivität)
2. `x<y` $\vee$ `x=y` $\vee$ `y<x` (Trichotomie)
3. `x<y` $\wedge$ `y<z` $\Rightarrow$ `x<z` (Transitivität)
4. $\neg$`(x<y` $\wedge$ `y<x+1)` (Diskretheit)

Definition von Ordnungsrelationen und Ungleichheiten. [c] Für alle ganzen Zahlen `x,y,z` gilt:

1. `x`$\neq$`y` $\Leftrightarrow$ $\neg$`(x=y)`
2. `x>y` $\Leftrightarrow$ `y<x`
3. `x`$\leq$`y` $\Leftrightarrow$ `x<y` $\vee$ `x=y`
4. `x`$\geq$`y` $\Leftrightarrow$ `y<x` $\vee$ `x=y`

Monotonieaxiome. Für alle ganzen Zahlen `x,y,z,w` gilt:

1. `x`$\geq$`y` $\wedge$ `z`$\geq$`w` $\Rightarrow$ `x+z`$\geq$`y+w` (Addition)
2. `x`$\geq$`y` $\wedge$ `z`$\leq$`w` $\Rightarrow$ `x-z`$\geq$`y-w` (Subtraktion)
3. `x`$\geq$`0` $\wedge$ `y`$\geq$`z` $\Rightarrow$ `x*y`$\geq$`x*z` (Multiplikation)
4. `x>0` $\wedge$ `x*y`$\geq$`x*z` $\Rightarrow$ `y`$\geq$`z` (Faktorisierung)

Sind `z` und `w` Konstanten, dann werden die Monotonoieaxiome der Addition und Subtraktion auch als *triviale Monotonien* bezeichnet.

Abbildung 4.5: Die Theorie $\mathcal{A}$ der elementaren Arithmetik

[a] Es sind also nur *geschlossene* Substitutionen zugelassen: aus `x=z` und `x`$\neq$`x*y` folgt `z`$\neq$`x*y`, nicht aber `z`$\neq$`z*y`.
[b] Entsprechende Gesetze mit größeren Konstanten wie `4+9=13` und `4*9=36` folgen hieraus mit den Ringaxiomen.
[c] In der NuPRL-Version von `arith` werden anstelle der Relationen $\neq$, $\leq$, $\geq$, `>` die entsprechenden rechten Seiten der Definition 3.4.5 auf Seite 145 verarbeitet.

Addition

	z>w	z≥w	z=w	z≠w
x>y	x+z≥y+w+2	x+z≥y+w+1	x+z≥y+w+1 x+w≥y+z+1	-----
x≥y	x+z≥y+w+1	x+z≥y+w	x+z≥y+w x+w≥y+z	-----
x=y	x+z≥y+w+1 y+z≥x+w+1	x+z≥y+w y+z≥x+w	x+z=y+w x+w=y+z	x+z≠y+w x+w≠y+z
x≠y	-----	-----	x+z≠y+w x+w≠y+z	-----

Multiplikation

	y≥z	y>z	y=z	y≠z
x>0	x*y≥x*z	x*y>x*z	x*y=x*z	x*y≠x*z
x≥0	xy*≥x*z	x*y≥x*z	x*y=x*z	-----
x=0	x*y=x*z x*y=0	x*y=x*z x*y=0	x*y=x*z x*y=0	x*y=x*z x*y=0
x≤0	x*y≤x*z	x*y≤x*z	x*y=x*z	-----
x<0	x*y≤x*z	x*y<x*z	x*y=x*z	x*y≠x*z
x≠0	-----	x*y≠x*z	x*y=x*z	x*y≠x*z

Subtraktion

	z>w	z≥w	z=w	z≠w
x>y	x-w≥y-z+2	x-w≥y-z+1	x-w≥y-z+1 x-z≥y-w+1	-----
x≥y	x-w≥y-z+1	x-w≥y-z	x-w≥y-z x-z≥y-w	-----
x=y	x-w≥y-z+1 y-w≥x-z+1	x-w≥y-z y-w≥x-z	x-w=y-z y-w=x-z	x-w≠y-z x-z≠y-w
x≠y	-----	-----	x-w≠y-z x-z≠y-w	-----

Faktorisierung

	x*y>x*z	x*y≥x*z	x*y=x*z	x*y≠x*z
x>0	y>z	y≥z	y=z	y≠z
x<0	y<z	y≤z	y=z	y≠z
x≠0	y≠z	-----	y=z	y≠z

Abbildung 4.6: Varianten der Monotoniegesetze von $\mathcal{A}$

Weiterhin gilt, daß die klassische und konstruktive Interpretation der elementar-arithmetischen Formeln übereinstimmen, da nur einfache arithmetische Operatoren (+, -, *), die elementaren Vergleichsoperatoren $=,\neq,<,>,\leq,\geq$ und die aussagenlogischen Konnektive ($\neg$, $\wedge$, $\vee$, $\Rightarrow$) zugelassen sind. Durch eine Induktion über die Termstruktur kann man daher beweisen, daß alle atomaren Formeln in $\mathcal{A}$ entscheidbar sind. Hieraus folgt wiederum die Entscheidbarkeit aller elementar-arithmetische Formeln, da Entscheidbarkeit von Formeln unter $\neg$, $\wedge$, $\vee$ und $\Rightarrow$ abgeschlossen ist. Insgesamt gilt also der folgende Satz.

Satz 4.3.3

Die Theorie der elementaren Arithmetik $\mathcal{A}$ ist entscheidbar.

Der (formal etwas aufwendige) Beweis von Satz 4.3.3 liefert im Prinzip bereits ein Verfahren, mit dem die Gültigkeit aller elementar-arithmetischen Aussagen entschieden werden kann, das allerdings sehr ineffizient ist. Um eine wirklich effiziente Entscheidungsprozedur für elementar-arithmetischen Aussagen zu entwickeln, ist es nötig, viele Varianten der Axiome von $\mathcal{A}$ und die üblichen Berechnungsverfahren für die Auswertung elementar-arithmetischer Terme direkt in den Entscheidungsalgorithmus einzubetten. Diese Varianten (modulo der Ringaxiome) sind in den Tabellen von Abbildung 4.6 aufgeführt, deren Zeilen und Spalten durch *Termschemata* indiziert sind. Jeder Tabelleneintrag beschreibt die Schlußfolgerungen, die sich aus den in den Indizes dargestellten Hypothesen ergeben.

Eine weitere wesentliche Vereinfachung ergibt sich daraus, daß aufgrund der Entscheidbarkeit von $\mathcal{A}$ jede elementar-arithmetische Formel auf eine konjunktive Normalform gebracht werden kann.

Lemma 4.3.4 (Konjunktive Normalform elementar-arithmetischer Formeln)

Zu jeder elementar-arithmetischen Formel F gibt es eine äquivalente elementar-arithmetische Formel G der Gestalt $(G_{11}\vee\ldots\vee G_{1n_1})\ \wedge\ \ldots\wedge\ (G_{m1}\vee\ldots\vee G_{1n_m})$, wobei alle G_{ij} atomare Formeln in $\mathcal{A}$ sind.

Beweis: Aufgrund der Entscheidbarkeit von $\mathcal{A}$ kann jede elementar-arithmetische Formel mit dem aus der klassischen Aussagenlogik bekannten Verfahren in eine Formel der Gestalt $(F_{11}\vee\ldots\vee F_{1n_1})\wedge\ldots\wedge(F_{m1}\vee\ldots\vee F_{1n_m})$ umgewandelt werden, wobei die F_{ij} atomare Formeln oder negierte atomare Formeln sind. Negierte atomare Formeln lassen sich wiederum durch Verwendung der 'negierten' Vergleichsoperatoren in atomare Formeln umwandeln (also z.B. ¬(x≤y) in x>y). □

Da nun eine Formel der Gestalt $(G_{11}\vee\ldots\vee G_{1n_1})\ \wedge\ \ldots\wedge\ (G_{m1}\vee\ldots\vee G_{1n_m})$ genau dann gültig ist, wenn jede einzelne Klausel dieser Formel allgemeingültig ist, können wir jede dieser Klauseln separat behandeln. Es reicht daher aus, daß die Entscheidungsprozedur in der Lage ist, Beweisziele der Form

$$\Gamma\ \vdash\ G_1\vee\ldots\vee G_n$$

zu verarbeiten, wobei die G_i elementar-arithmetische Formeln sind und der Sonderfall $n=0$ bedeutet, daß die Konklusion Λ ist (da keine Alternative gegeben wird).

Im folgenden wollen wir die Entscheidungsprozedur `arith` schrittweise anhand eines Beispiels entwickeln und anschließend die hierbei verwendeten Erkenntnisse als Satz festhalten. Wir betrachten das Beweisziel

$$\Gamma,\ i\ \texttt{:x+y>z},\ \Gamma',\ j\ \texttt{:2*x}\geq\texttt{z},\ \Delta\ \vdash\ \texttt{3*x+y}\geq\texttt{2*z-1}$$

In einem ersten vorverarbeitenden Schritt[32] können neue Hypothesen ergänzt werden, die sich aus bestehenden Hypothesen aufgrund der Monotoniegesetze ergeben. Dieser Schritt muß durch den Benutzer gesteuert werden, der angibt, wie die neue Hypothese durch arithmetische Operationen aus den bereits bestehenden zu erzeugen ist. Dabei kann man zwei Hypothesen aufaddieren, subtrahieren, multiplizieren und dividieren (beschränkt auf Faktorisierung), sofern hierdurch Schlußfolgerungen aufgrund der Monotoniegesetze in Abbildung 4.6 gezogen werden können. Hypothesen, die sich aufgrund *trivialer Monotonien* aus anderen Hypothesen ergeben, brauchen nicht explizit generiert zu werden, da dies im Verlaufe von `arith` automatisch geschieht.

Definition 4.3.5

Eine <u>triviale Monotonie</u> ist die Anwendung eines Monotonieaxioms für Addition oder Subtraktion auf zwei Prämissen, von denen eine die Gestalt $n\,\rho\,m$ hat, wobei m und n ganzzahlige Konstanten und ρ ein arithmetischer Vergleichsoperator ist. Jede andere Anwendung von Monotonieaxiomen ist <u>nichttrivial</u>.

Intuitiv betrachtet ist die Anwendung einer trivialen Monotonie also das Aufaddieren von Konstanten auf beiden Seiten eines elementaren arithmetischen Vergleichs, sofern danach noch ein Vergleich gezogen werden kann. So können wir zum Beispiel aus x=y folgern, daß x+2≠y+4 gilt, wissen aber nichts über das Verhältnis von x+2 and y+4, falls x≠y gilt.

Beispiel 4.3.6

1. Im Falle des Beweisziels $\Gamma,\ i\ \texttt{:x+y>z},\ \Gamma',\ j\ \texttt{:2*x}\geq\texttt{z},\ \Delta\ \vdash\ \texttt{3*x+y}\geq\texttt{2*z-1}$ gibt es zunächst nur nichttriviale Monotonien, von denen wir die Addition der Hypothesen i und j ausnutzen wollen. Auf diese Art erhalten wir eine Hypothese, in der – nach Vereinfachungen – links 3*x+y und rechts 2*z vorkommen wird. Um dies zu erreichen, muß die Entscheidungsprozedur `arith` mit dem Parameter i+j aufgerufen werden. Entsprechend der Additionstabelle in Abbildung 4.6 erhalten wir somit als Ausgangspunkt für das eigentliche Entscheidungsverfahren

   $$\Gamma,\ i\ \texttt{:x+y>z},\ \Gamma',\ j\ \texttt{:2*x}\geq\texttt{z},\ \Delta,\ \texttt{x+y+2*x}\geq\texttt{z+z+1}\ \vdash\ \texttt{3*x+y}\geq\texttt{2*z-1}$$

2. Da in der elementaren Arithmetik klassisches und intuitionistisches Schließen übereinstimmt, ist es legitim, auf eine *negative Darstellung des Problems* überzugehen, d.h. anzunehmen, daß die Konklusion falsch wäre und hieraus einen Widerspruch zu folgern. Dieser Wechsel der Darstellung ist ein Standardtrick im Theorembeweisen, denn er reduziert das ursprüngliche Problem auf die Frage, eine sich selbst widersprechende Menge von Hypothesen zu finden. Dieser Schritt, dessen Rechtfertigung in Theorem 4.3.7.1 gegeben wird, führt zu der Sequenz

   $$\Gamma,\ \texttt{x+y>z},\ \Gamma',\ \texttt{2*x}\geq\texttt{z},\ \Delta,\ \texttt{x+y+2*x}\geq\texttt{z+z+1},\ \neg\texttt{(3*x+y}\geq\texttt{2*z-1)}\ \vdash\ \Lambda$$

3. Hypothesen, die keine elementar-arithmetischen Formeln enthalten, können problemlos ignoriert werden, da sie auf keinen Fall zu einem (arithmetischen) Beweis beitragen werden. Dies führt dazu, daß Γ, Γ' und Δ aus der Hypothesenliste verschwinden.

[32] In der derzeitigen Implementierung ist dieser Vorverarbeitungsschritt nicht in der eigentlichen `arith`-Prozedur enthalten sondern muß separat durchgeführt werden.

4. Im nächsten Schritt werden die Komparanden (d.h. die Terme links und rechts von einem Vergleichsoperator) jeder Hypothese auf eine Normalform gebracht. Hierzu verwendet man die Standarddarstellung von Polynomen als Summen von Produkten $a{*}x_1^n{*}x_2^n\ldots{+}\ldots$. Diese symbolische Normalisierung ist eine einfache arithmetische Umformung, welche die Gültigkeit einer Sequenz nicht beeinflußt, aber dafür sorgt, daß semantisch gleiche Terme auch die gleiche syntaktische Gestalt erhalten. In unserm Beispiel führt dies zu

   $$x{+}y>z,\ 2{*}x\geq z,\ 3{*}x{+}y\geq 1{+}2{*}z,\ \neg(3{*}x{+}y\geq(-1){+}2{*}z)\ \vdash\ \Lambda$$

5. Nun werden alle Komparanden in *monadische lineare Polynome* transformiert, also in Ausdrücke der Form $c\ +\ u_i$, wobei u_i eine (neue) Variable ist. Dies bedeutet, daß ab jetzt alle nichtkonstanten Komponenten eines Terms wie eine Variable behandelt werden, wobei gleiche Komponenten natürlich durch die gleiche Variable repräsentiert werden. Dieser Schritt basiert auf Theorem 4.3.7.2 (was relativ mühsam zu beweisen ist) und führt zu

   $$u_0>z,\ u_1\geq z,\ u_2\geq 1{+}u_3\ \neg(u_2\geq(-1){+}u_3)\ \vdash\ \Lambda$$

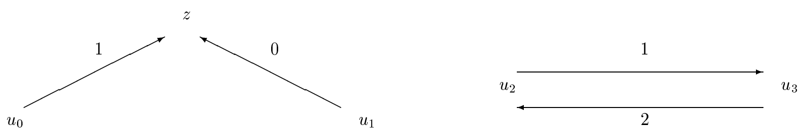
   Hierbei wurde $u_0\equiv x{+}y$, $u_1\equiv 2{*}x$, $u_2\equiv 3{*}x{+}y$ und $u_3\equiv 2{*}z$ gewählt.

6. Die nächste Normalisierung betrifft die Vergleichsoperatoren. Alle Hypothesen werden transformiert in Formeln der Gestalt $t_1\geq t_2$, wobei die linke Seite t_1 entweder eine Variable u_i oder die Konstante 0 ist und t_2 ein monadisches lineares Polynom. Dies läßt sich leicht durch Verwendung von Konversionstabellen[33] erreichen, kann allerdings dazu führen, daß neue Disjunktionen atomarer Formeln entstehen und somit mehrere Alternativen betrachtet werden müssen. In unserem Fall lautet das Resultat

   $$u_0\geq z{+}1,\ u_1\geq z,\ u_2\geq 1{+}u_3\ u_3\geq u_2{+}2\ \vdash\ \Lambda$$

   Wenn nach diesem Schritt Disjunktionen atomarer Formeln in den Hypothesen auftreten, so ist es nötig, diese zu zerlegen und jeden dieser Fälle separat zu betrachten. Im schlimmsten Fall kann dies dazu führen, daß im nächsten Schritt exponentiell viele Alternativen (gemessen an der Anzahl der ursprünglich vorhandenen Ungleichheiten) untersucht werden müssen.[34]

7. Im vorletzten Schritt wird die Sequenz in einen Graphen umgeformt, welcher die Ordnungsrelation zwischen den Variablen beschreibt. Jeder Knoten dieses Graphen repräsentiert eine Variable oder eine Konstante und eine Kante $u_i \xrightarrow{c} u_j$ repräsentiert die Ungleichung $u_i\geq u_j{+}c$. In unserem Beispiel führt dies zu folgendem Graphen



   Dieser Graph besitzt (gemäß Theorem 4.3.7.3) einen positiven Zyklus, wenn die ursprüngliche Formelmenge widersprüchlich war. Mit einem graphentheoretischen Standardalgorithmus (siehe zum Beispiel [Constable *et.al.*, 1982, Seite 241–242]) läßt sich nun leicht überprüfen, ob dies der Fall ist.

   In obigem Graphen gibt es einen Zyklus mit den Knoten u_2 und u_3. Dies bedeutet, daß die Hypothesenmenge des letzten Schrittes widersprüchlich und somit die ursprüngliche Sequenz allgemeingültig war. Die Prozedur `arith` schließt damit die Überprüfung der Formel erfolgreich ab und hinterläßt als Teilziele, daß `x, y` und `z` als Elemente von $\mathbb{Z}$ nachzuweisen sind.

[33] Typische Konversionen, die hierbei zum Tragen kommen werden, sind zum Beispiel $\neg x\geq y \Leftrightarrow x<y$, $x>y \Leftrightarrow x\geq y{+}1$, $x{=}y \Leftrightarrow x\geq y \wedge y\geq x$, $x\neq y \Leftrightarrow x\geq y{+}1 \vee y\geq x{+}1$, etc. sowie das beidseitige Aufaddieren von Konstanten, um negative Konstanten in Polynomen zu eliminieren.

[34] In der Praxis ist diese miserable Komplexitätsschranke allerdings nicht so bedeutend, da `arith` in den meisten Fällen nur eine relativ kleine Anzahl arithmetischer Hypothesen verarbeiten muß und von diesen nur sehr wenige auch Ungleichheiten sind, die zu einer Aufblähung in Alternativen führen. Schlüsse, die ausschließlich auf Ungleichheiten beruhen, sind auch für den Menschen keineswegs trivial.

Gegeben sei ein Beweisziel der Gestalt $\Gamma \vdash G_1\vee\ldots\vee G_n$ (Λ, falls $n=0$) wobei die G_i elementar-arithmetische Formeln sind und Γ beliebige Hypothesen enthält.
Der Algorithmus für die Regel `arith` *i op j*, wobei *op* entweder `+, -, *` oder $\div$ ist, geht wie folgt vor.

1. Führe die geforderten Monotonieschritte mit den Hypothesen i und j aus und erzeuge eine neue Hypothese gemäß den Einträgen in den Tabellen aus Abbildung 4.6 aus.
2. Transformiere die resultierende Sequenz $\Gamma' \ \vdash\ G_1\vee\ldots\vee G_n$ in $\Gamma,\ \neg G_1,\ldots,\neg G_n\ \vdash\ \Lambda$.
3. Zerlege alle Konjunktionen entsprechend der Regel `and_e` in neue Einzelhypothesen.
4. Transformiere alle Ungleichungen der Form $x\neq y$ in die Disjunktion $x\geq y{+}1 \vee y\geq x{+}1$
5. Zerlege alle Disjunktionen in den Hypothesen entsprechend der Regel `or_e`. Alle entstehenden Beweisziele werden im folgenden separat betrachtet und müssen zum Erfolg führen.
6. Entferne alle Hypothesen, die keine atomaren elementar-arithmetischen Formeln sind.
7. Ersetze Teilterme der Komparanden, welche nicht aus `+, -, *` oder ganzzahligen Konstanten aufgebaut sind, durch (neue) Variablen, wobei gleiche Terme durch die gleiche Variable ersetzt werden.
8. Transformiere alle Komparanden einer Hypothese in die Standarddarstellung von Polynomen.
9. Transformiere alle Komparanden in monadische lineare Polynome $c\ +\ u_i$, indem jedes nichtkonstante Polynom p durch eine (neue) Variable u_i ersetzt wird.
10. Konvertiere alle Hypothesen in Ungleichungen der Gestalt $t_1\geq t_2$, wobei t_1 eine Variable u_i oder die Konstante 0 und t_2 ein monadisches lineares Polynom ist.
11. Erzeuge den Ordnungsgraphen der entstandenen Formelmenge. Erzeuge Knoten für jede Variable und Konstante und eine Kante $u_i \xrightarrow{c} u_j$ für die Ungleichung $u_i\geq u_j{+}c$.
12. Teste, ob der Ordnungsgraph einen positiven Zyklus hat oder nicht. Im Erfolgsfall generiere Wohlgeformtheitsziele für jeden durch eine Variable repräsentierten Teilterm. Andernfalls erzeuge eine Ausnahme mit einer entsprechenden Fehlermeldung.

Abbildung 4.7: Die Entscheidungsprozedur `arith`

Über die eben beschriebenen Fähigkeiten hinaus ist die arithmetische Entscheidungsprozedur des NuPRL Systems in der Lage, auch nichtelementare arithmetische Ausdrücke zu verarbeiten, also arithmetische Vergleiche zu handhaben, deren Komparanden nicht nur aus `+, -, *` aufgebaut sind. Derartige Teilausdrücke werden von `arith` wie atomare Terme behandelt, die nicht weiter analysiert werden. Da `arith` keinerlei Wohlgeformtheitsüberprüfungen durchführt, müssen alle atomaren Terme – seien es nun Konstanten, Variablen oder komplexere nichtelementare arithmetische Ausdrücke – nach einer erfolgreichen Überprüfung der Gültigkeit einer Sequenz als Elemente von $\mathbb{Z}$ nachgewiesen werden. Entsprechende Behauptungen werden als Teilziele der `arith`-Regel generiert.

Abbildung 4.7 beschreibt das allgemeine Verfahren, welches beim Aufruf der `arith`-Regel ausgeführt wird. Zugunsten einer effizienteren Verarbeitung wird die Relation $\neq$ schon relativ früh aufgelöst und Disjunktionen frühzeitig als alternative Fälle behandelt. Es wird gerechtfertigt durch die Erkentnisse des folgenden Satzes.

Satz 4.3.7

1. *Sind $G_1,\ \ldots,\ G_n$ entscheidbare Aussagen so ist die Sequenz $\Gamma\ \vdash\ G_1\vee\ldots\vee G_n$ genau dann allgemeingültig, wenn $\Gamma,\ \neg G_1,\ldots,\neg G_n\ \vdash\ \Lambda$ gültig ist.*
2. *Es seien $e_1,\ldots,e_k$ elementar-arithmetische Terme und $u_1,\ldots,u_k$ Variablen.*
   Eine Menge $F_1[e_1,..,e_k\,/\,u_1,..,u_k],\ \ldots,\ F_n[e_1,..,e_k\,/\,u_1,..,u_k]$ von (instantiierten) elementar-arithmetischen Formeln ist genau dann widersprüchlich, wenn die Menge $F_1,\ \ldots,\ F_n$ widersprüchlich ist.
3. *Es sei $\Gamma = v_1\geq u_1+c_1,\ \ldots,\ v_n\geq u_n+c_n$ eine Menge von atomaren arithmetischen Formeln, wobei die v_i und u_i Variablen (oder die Konstante 0) und die c_i nichtnegative Konstanten sind, und $\mathcal{G}$ der Graph, welcher die Ordnungsrelation zwischen den Variablen von Γ beschreibt.*
   Dann ist Γ genau dann widersprüchlich, wenn $\mathcal{G}$ einen positiven Zyklus besitzt.

Es sei angemerkt, daß in der derzeitigen NuPRL Implementierung von `arith` der erste Schritt – die Monotonieoperation – nicht enthalten ist. Ausschließlich die trivialen Monotonien kommen im Verlaufe der im Schritt 10 beschriebenen Konversion zum Tragen. Es sind allerdings einige Erweiterungen von `arith` als Taktiken implementiert, die Definitionen von Teiltypen von $\mathbb{Z}$ auflösen und die entsprechenden Bereichsinformationen ebenfalls verarbeiten können. Weitere Informationen hierzu findet man in [Jackson, 1993b, Kapitel 8.7].

4.3.2 Schließen über Gleichheit

Schließen über Gleichheit ist das Problem zu verifizieren, daß eine Gleichheit in der Konklusion eines Beweisziels aus einer Reihe bekannter Gleichheiten in den Hypothesen folgt, also daß zum Beispiel

`f(f(a,b),b) = a` aus `f(a,b) = a`

oder

`g(a) = a` aus `g(g(g(a))) = a` und `g(g(g(g(g(a))))) = a`

folgt. Nahezu alle Problem, die in der Praxis auftauchen, und insbesondere ein formales Schließen über Programme und die von ihnen berechneten Werte verlangt ein solches Schließen über Gleichheiten.

Wir hatten in Abschnitt 2.2.7 bereits angesprochen, daß Schließen über Gleichheiten im wesentlichen aus einer Anwendung der (in Abbildung 3.7 auf Seite 115 formalisierten) Regeln der Reflexivität, Symmetrie, Transitivität und der Substitution besteht. Formale Beweise, die sich jedoch ausschließlich auf diese Regeln stützen, sind im Allgemeinfall sehr aufwendig, obwohl die dahinterstehenden Einsichten sehr gering sind. So ist für einen Menschen sehr leicht einzusehen, warum die obengenannten Gleichheitsschlüsse wirklich gültig sind, während der formale Beweis mehrere trickreiche Substitutionen enthält. Da mittlerweile bekannt ist, daß die spezielle Theorie der Gleichheit – die nur aus den Axiomen Reflexivität, Symmetrie, Transitivität und Substitution besteht – entscheidbar ist [Ackermann, 1954][35] ist es sinnvoll, eine Entscheidungsprozedur zum Schließen über Gleichheiten zu entwickeln, welche ein vielfaches Anwenden der elementaren Gleichheitsregeln *in einem einzigen Schritt* durchführt, und diese als Inferenzregel in das Beweisentwicklungssystem mit aufzunehmen.

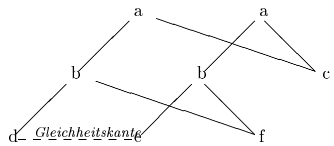
Es gibt eine Reihe guter Algorithmen, die für diesen Zweck eingesetzt werden können. Im folgenden werden wir ein Verfahren vorstellen, welches auf Arbeiten von Greg Nelson and Derek Oppen [Nelson & Oppen, 1979, Nelson & Oppen, 1980] basiert und als Kern der `equality`-Regel (siehe Abbildung 3.28 auf Seite 176) implementiert wurde. Die Schlüsselidee ist hierbei, in einem Graphen die transitive Hülle einer Relation zu konstruieren und hieraus dann abzuleiten, ob zwei Elemente in der gewünschten Relation zueinander stehen. Im Prinzip ist dieses Verfahren nicht nur zur Behandlung von Gleichheitsfragen geeignet, sondern kann ebenfalls dazu benutzt werden, um über Listenstrukturen und ähnliche Problemstellungen zu schließen. Wir wollen den Algorithmus zunächst an einem einfachen Beispiel erklären.

Beispiel 4.3.8

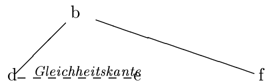
Wir wollen zeigen, daß `a(b(d,f),c) = a(b(e,f),c)` aus der Gleichheit `d=e` folgt, wobei `a,b,c,d,e` und `f` beliebige Terme sind.

Dazu repräsentieren wir Term-Ausdrücke in der üblichen Baumdarstellung und die Gleichheit von Term-Ausdrücken als Gleichheitskanten ("*equality links*") zwischen den Knoten dieser Bäume, wobei identische Teilausdrücke als gemeinsame Teilbäume der verschiedenen Termbäume behandelt werden. Für die obigen Formeln bedeutet dies, daß `c` und `f` gemeinsame Knoten sind, während `d` und `e` durch eine Gleichheitskante verbunden werden.

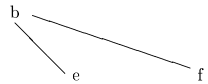
[35] Diese Form des Gleichheitsschließens darf nicht verwechselt werden mit dem aus dem automatischen Beweisen bekannten Problem, logische Schlüsse unter der Verwendung von Gleichheiten zu ziehen. Letzteres ist erheblich komplizierter, da hier Schließen über Gleichheiten und prädikatenlogisches Schließen, das bekanntlich alleine schon unentscheidbar ist, miteinander gekoppelt werden. Für derartige Fragestellungen sind daher wesentlich aufwendigere Verfahren wie Paramodulation [Robinson & Wos, 1969] oder modifizierte logische Verfahren wie E-Resolution [Morris, 1969, Anderson, 1970] oder Gleichheitskonnektionen [Bibel, 1987, Bibel, 1992] nötig.



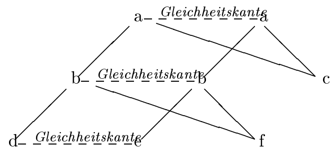
Der Beweis, daß zwei Ausdrücke gleich sind, ist nun dasselbe wie der Beweis, daß eine Gleichheitskante zwischen den Wurzelknoten ihrer Baumdarstellung konstruiert werden kann. Im Algorithmus geschieht dieser Nachweis dadurch, daß Knoten, die mit einer Gleichheitskante verbunden sind, miteinander identifiziert werden, was im Endeffekt dasselbe ist, als ob die Knoten zusammengelegt worden wären. In unserem Fall bedeutet dies, daß `d` mit `e` identifiziert wird.



ist identisch mit



Im nächsten Schritt suchen wir nun nach Teilbäumen, in denen `d` oder `e` vorkommen und die in allen anderen Knoten identisch sind. Zwischen den Wurzelknoten dieser Teilbäumen kann nun eine Gleichheitskante konstruiert werden. Wir können also `b(`$\boxed{\texttt{d}}$`,f)` mit `b(`$\boxed{\texttt{e}}$`,f)` verbinden und im Anschluß daran `a(`$\boxed{\texttt{b(d,f)}}$`,c)` with `a(`$\boxed{\texttt{b(e,f)}}$`,c)`, wodurch die Gleichheit der beiden Terme nachgewiesen ist.



Diese Lösung des Problems ist erheblich eleganter und effizienter als ein Beweis, der sich nur auf die Regeln der Reflexivität, Symmetrie, Transitivität und der Substitution stützen kann.

Der Algorithmus konstruiert also Gleichheitskanten zwischen Teilbäumen, bis das Beweisziel bewiesen ist oder alle verbundenen Teilbäume ohne weiteren Fortschritt bearbeitet wurden. Aus technischer Sicht ist diese Methode zum Schließen über Gleichheiten verwandt mit Verfahren zur Bestimmung gemeinsamer Teilausdrücke in optimierenden Compilern.[36] Aus mathematischer Sicht bedeutet die Konstruktion von Gleichheitskanten dasselbe wie die Berechnung der transitiven Hülle (auch *Kongruenzabschluß*, englisch *congruence closure*) der Gleichheitsrelation innerhalb des Termgraphen. Um den Algorithmus präzise beschreiben zu können, führen wir ein paar graphentheoretische Begriffe ein.

Definition 4.3.9

Es sei $G = (V, E)$ ein gerichteter Graph mit markierten Knoten und R eine Relation auf V.

1. *$\underline{l(v)}$ bezeichnet die Markierung (label) des Knoten v in G*
2. *$\underline{\delta(v)}$ bezeichnet die Anzahl der von v ausgehenden Kanten.*
3. *Für $1 \leq i \leq \delta(v)$ bezeichnet $\underline{v[i]}$ den i-ten Nachfolgerknoten von v.*[37]
4. *u ist ein <u>Vorgänger</u> von v, wenn $v = u[i]$ für ein i ist.*
5. *Zwei Knoten u und v sind <u>kongruent unter R</u>, wenn $l(u) = l(v)$, $\delta(u) = \delta(v)$ und $(u[i], v[i]) \in R$ für alle $1 \leq i \leq \delta(u)$ gilt.*
6. *R ist <u>abgeschlossen unter Kongruenzen</u>, wenn für alle Knoten u und v, die unter R kongruent sind, die Relation $(u, v) \in R$ gilt.*
7. *Der <u>Kongruenzabschluß</u> R^* von R ist die eindeutige minimale Erweiterung von R, die abgeschlossen unter Kongruenzen und eine Äquivalenzrelation ist.*

Gegeben sei ein gerichteter Graph $G = (V, E)$, eine unter Kongruenzen abgeschlossene Äquivalenzrelation R und zwei Knoten u, v aus V.

1. Wenn u und v in der gleichen Äquivalenzklasse von R liegen, dann ist R der Kongruenzabschluß von $R \cup \{(u, v)\}$. Lasse R unverändert und beende die Berechnung.

2. Andernfalls sei P_u die Menge aller Vorgänger von Knoten aus der Äquivalenzklasse von u und P_v die Menge aller Vorgänger von Knoten aus der Äquivalenzklasse von v.

3. Modifiziere R durch Verschmelzung der Äquivalenzklassen von u und v.

4. Wiederhole für alle $x \in P_u$ und $y \in P_v$

   Falls die Äquivalenzklassen von x und y verschieden sind, aber x und y kongruent sind, so modifiziere R durch Aufruf von MERGE(R,x,y). Andernfalls lasse R unverändert.

Ausgabe sei die modifizierte Relation R.

Abbildung 4.8: Der MERGE Algorithmus

Der Kongruenzabschluß einer Menge von Gleichheiten s_1=t_1, ..., s_n=t_n läßt sich mit Hilfe einiger Standardalgorithmen auf Graphen relativ einfach berechnen. Der Kongruenzabschluß der *Identitätsrelation*, bei der Knoten nur in Relation mit sich selbst sind, ist wieder die Identitätsrelation und mit Hilfe des Algorithmus MERGE, den wir in Abbildung 4.8 angegeben haben, können wir schrittweise Gleichungen zu dieser Relation hinzunehmen und jeweils den Kongruenzabschluß berechnen. Aus Effizienzgründen stellen wir dabei eine Äquivalenzrelation durch die entsprechende Menge der Äquivalenzklassen dar. Die Eigenschaften von MERGE beschreibt der folgende Satz.

Satz 4.3.10

Es sei $G = (V, E)$ ein gerichteter Graph mit markierten Knoten, R eine Äquivalenzrelation auf V, die unter Kongruenzen abgeschlossen ist, und u, v beliebige Knoten aus V.

Dann berechnet MERGE(R,u,v)[38] *den Kongruenzabschluß der Relation $R \cup \{(u, v)\}$.*

Wir wollen die Arbeitsweise des MERGE-Algorithmus an einem Beispiel erläutern.

Beispiel 4.3.11

Wir wollen den Kongruenzabschluß von `g(g(g(a)))=a` und `g(g(g(g(g(a)))))=a` berechnen. Da alle vorkommenden Terme Teilterme von `g(g(g(g(g(a)))))` sind, genügt es, die Baumdarstellung dieses Terms zu konstruieren und alle Kongruenzen in diesem Graphen zu bestimmen. Wir beginnen mit der Äquivalenzrelation $R = \{\{v_1, v_6\}, \{v_2\}, \{v_3\}, \{v_4\}, \{v_5\}\}$ (also `g(g(g(g(g(a)))))=a`), die offensichtlich unter Kongruenzen abgeschlossen ist, nehmen als neue Knoten u=v_3 (d.h. `g(g(g(a)))`) sowie v=v_6 (`a`) und rufen MERGE(R,u,v) auf.

$$\begin{array}{ll} g & (v_1) \\ \downarrow & \\ g & (v_2) \\ \downarrow & \\ g & (v_3) \\ \downarrow & \\ g & (v_4) \\ \downarrow & \\ g & (v_5) \\ \downarrow & \\ a & (v_6) \end{array}$$

Die Vorgänger von u ist v_2 und der von v ist v_5. Da der zu v äquivalente Knoten v_1 keinen Vorgänger besitzt erhalten wir $P_u = \{v_2\}$ und $P_v = \{v_5\}$.

Nun verschmelzen wir die Äquivalenzklassen von u und v und erhalten $\{\{v_1, v_3, v_6\}, \ldots\}$.

Aus den Klassen P_u bzw. P_v wählen wir nun $x = v_2$ und $y = v_5$. Da die Nachfolger von x und y äquivalent sind, sind x und y kongruent, gehören aber zu verschiedenen Äquivalenzklassen. Wir rufen also MERGE(R,v_2,v_5) auf.

[36] Hierzu siehe [Nelson & Oppen, 1980] und die Dissertation von Scott Johnson [Johnson, 1983].

[37] Es ist zulässig, daß mehrere von v ausgehende Kanten auf den gleichen Knoten zeigen, also daß $v[i] = v[j]$ für $i \neq j$ gilt.

[38] Man beachte, daß R Eingabe- *und* Ausgabeparameter von MERGE ist und hierbei verändert wird.

Gegeben seien Hypothesen s_1=t_1, ..., s_n=t_n und eine Konklusion s=t.

1. Konstruiere den Graphen G, der aus den Baumdarstellungen der Terme $s, s_1, \ldots, s_n, t, t_1, \ldots, t_n$ besteht, wobei identische Teilausdrücke jeweils durch einen Teilbaum dargestellt werden.

   Wähle R als Identitätsrelation auf den Knoten von G

2. Wende schrittweise die Prozedur MERGE(R,$\tau(s_i)$,$\tau(t_i)$) für $1 \leq i \leq n$ an.

3. Wenn $\tau(s)$ in der gleichen Äquivalenzklasse wie $\tau(t)$ liegt, dann sind s und t gleich.

   Andernfalls folgt s=t nicht aus s_1=t_1, ..., s_n=t_n

Dabei bezeichne $\tau(u)$ den Wurzelknoten der Baumdarstellung des Terms u in G.

Abbildung 4.9: Entscheidungsalgorithmus zum Schließen über Gleichheiten

Dies führt dazu, daß nun in R die Äquivalenzklasse $\{v_2, v_5\}$ gebildet wird und v_1 und v_4 untersucht werden, die sich nun ebenfalls als kongruente Elemente verschiedener Äquivalenzklassen herausstellen.

Beim rekursiven Aufruf von MERGE(R,v_1,v_4) werden nun die Klassen von v_1 und v_4 zu $\{v_1, v_3, v_4, v_6\}$ verschmolzen und die Vorgänger von $\{v_1, v_3, v_6\}$ bzw. von v_4 betrachtet.

Unter diesen sind v_2 und v_3 kongruente Elemente verschiedener Klassen, was zu MERGE(R,v_2,v_3) führt und $\{v_1, v_2, v_3, v_4, v_6\}$ liefert. Die nachfolgende Betrachtung von v_5 und v_3 liefert über MERGE(R,v_5,v_3) schließlich die Äquivalenzklasse $\{v_1, v_2, v_3, v_4, v_5, v_6\}$.

Da nun alle Knoten äquivalent sind stoppt der Algorithmus und liefert die Relation $\{v_1, v_2, v_3, v_4, v_5, v_6\}$ als Kongruenzabschluß der Ausgangsgleichungen. Alle Teilterme von `g(g(g(g(g(a)))))` sind äquivalent.

Quantorenfreies Schließen über Gleichheiten mit uninterpretierten Funktionssymbolen und Variablen kann nun relativ leicht auf das Problem der Bestimmung des Kongruenzabschlusses der Gleichheitsrelation reduziert werden. Um zu entscheiden, ob s=t aus den Hypothesen s_1=t_1, ..., s_n=t_n folgt, genügt es, schrittweise mit dem MERGE-Algorithmus den Kongruenzabschluß der Gleichheiten s_1=t_1, ..., s_n=t_n zu berechnen und dann zu testen, ob s und t in der gleichen Äquivalenzklasse liegen. Dieser Algorithmus, den wir in Abbildung 4.9 beschrieben haben, ist die Grundlage der `equality`-Regel von NuPRL.[39]

4.3.3 Andere Entscheidungsverfahren

Neben den hier vorgestellten Entscheidungsverfahren für Arithmetik und Schließen über Gleichheiten gibt es auf dem Gebiet des automatischen Beweisens noch weitere Entscheidungsprozeduren für gewisse mathematische Teilgebiete. So läßt sich zum Beispiel die Bildung des Kongruenzabschlusses auch für daß Schließen über *Listenstrukturen* und ähnlich gelagerte Problemstellungen einsetzen.[40] Für bestimmte *geometrische Probleme* gibt es effiziente Entscheidungsverfahren [Wu, 1986, Chou & Gao, 1990] genauso wie für die (klassische) *quantorenfreie (Aussagen-)logik mit uninterpretierten Funktionssymbolen* [Davis & Putnam, 1960, Prawitz, 1960].

[39] Man beachte hierbei, daß die Entscheidungsprozedur – aufgrund der einheitlichen Syntax von Termen – in der Lage ist, beliebige Terme zu verarbeiten und nicht etwa nur solche, die aus Funktionsapplikationen aufgebaut sind. Die Rolle der Funktionssymbole wird hierbei von den Operatoren (siehe Definition 3.2.5 auf Seite 104) übernommen. Allerdings befaßt sich der Entscheidungsalgorithmus ausschließlich mit der Frage der Gleichheit von Termen und kann deshalb keine Teilziele behandeln, in denen neben der reinen Gleichheitsüberprüfung auch noch Typbestimmungen nötig sind. Dies schränkt die Einsatzmöglichkeiten der Prozedur `equality` innerhalb von typentheoretischen Beweisen wieder etwas ein. `equality` kann nur auf den Wurzelknoten eines Terms operieren und darf nicht in Teilterme hineingehen, ohne deren Typ zu bestimmen. Ohne eine zusätzliche Unterstützung durch Taktiken sind die Fähigkeiten der `equality`-Regel von NuPRL daher etwas schwächer als die des allgemeinen Algorithmus.

[40] Diese Prozeduren wurden – ebenso wie die unten angesprochene Integration von Entscheidungsprozeduren – in Vorläufern des NuPRL Systems, die noch nicht auf Typentheorie basierten, erfolgreich eingesetzt. Eine Erweiterung auf typentheoretische Konzepte wurde bisher noch nicht durchgeführt, da der Taktik-Ansatz für diese Zwecke vielversprechender erscheint.

Auch für die *klassische Prädikatenlogik* gibt es eine Vielfalt von Beweisverfahren, die üblicherweise eine effiziente *Suchstrategie* [Robinson, 1965, Andrews, 1971, Bibel, 1987] beinhalten. Diese Verfahren bergen allerdings immer die Gefahr in sich, viel Rechenzeit und Speicherplatz zu verbrauchen und dennoch erfolglos zu arbeiten, also keine Entscheidung zu liefern. Aufgrund der Unentscheidbarkeit der Prädikatenlogik können sie daher nur dazu dienen, Beweise zu finden, wenn es welche gibt. *Entscheidungsprozeduren für die Prädikatenlogik kann es jedoch nicht geben.*

Üblicherweise liefert eine Entscheidungsprozedur wie `arith` oder `equality` einen vollständigen Beweis oder sie schlägt fehl, weil die Konklusion innerhalb der eingeschränkten Theorie dieser Prozedur nicht gültig ist. Es gibt jedoch eine Möglichkeit, mehrere Entscheidungsprozeduren miteinander zu kombinieren, so daß sie Informationen über die entdeckten Relationen untereinander austauschen[41] und insgesamt eine größere Theorie bearbeiten können. Durch eine derartige *Kooperation von Entscheidungsprozeduren* kann die Leistungsfähigkeit des Beweissystems erheblich gesteigert werden. Ein Algorithmus, mit dem man verschiedene Entscheidungsprozeduren für quantorenfreie Theorien miteinander kombinieren kann, ist in [Nelson & Oppen, 1979] beschrieben und erfolgreich in λ-PRL (einem Vorläufer von ν-PRL) eingesetzt worden.

4.3.4 Grenzen der Anwendungsmöglichkeiten von Entscheidungsprozeduren

In Beweissystemen, deren zugrundeliegende Logik weniger ausdruckstark ist als die intuitonistische Typentheorie, kann eine Kooperation verschiedener festeingebauter Entscheidungsprozeduren für Arithmetik, Gleichheit, Listenstrukturen, quantorenfreie Logik etc. sehr erfolgreich eingesetzt werden. In Systemen mit einer reichhaltigeren Typstruktur entstehen hierbei jedoch Probleme mit der Typisierung der auftauchenden Teilterme[42] und deshalb können nur wenige dieser Entscheidungsprozeduren sauber in die Theorie eingebettet werden. Aus diesem Grunde stößt das Konzept der Entscheidungsprozeduren als Mittel zur Automatisierung des logischen Schließens sehr schnell an seine Grenzen, denn es wird immer wieder Anwendungsgebiete für logisch-formales Schließen geben, die nicht mehr in einer entscheidbaren Theorie formuliert werden können.

Aber auch aus praktischen Gesichtspunkten macht es wenig Sinn, ein Beweissystem mit einer fest eingebauten Beweisstrategie ständig um Entscheidungsprozeduren zu erweitern, wann immer ein Benutzer eine Problemstellung findet, die von dem bestehenden System nicht mehr gelöst werden kann.[43] Denn hierzu ist immer ein Eingriff in das eigentliche Beweissystem notwendig, was dazu führt, daß dieses im Laufe der Zeit immer weniger verständlich wird und eine ursprünglich vorhandene saubere Systemstruktur irgendwann verloren geht. Zudem müßte jede Beweisprozedur vor ihrer Integration in das System als korrekt und konsistent mit dem Rest des Systems nachgewiesen werden, was zeitaufwendig und üblicherweise kaum durchführbar ist und somit das Vertrauen in die Zuverlässigkeit des Gesamtsystems erheblich belastet.

Entscheidungsprozeduren sind also hilfreich, um Probleme einer kleinen Anzahl wohlverstandener Schlüsseltheorien automatisch zu lösen. Als allgemeine Vorgehensweise, die dazu geeignet ist, das logische Schließen in allen Bereichen der Mathematik und Programmierung zu automatisieren, können sie jedoch nicht angesehen werden. Hierfür ist das Konzept der Taktiken erheblich besser geeignet, da es bei geringen Effizienzverlusten erheblich mehr Flexibilität und Sicherheit bei der Implementierung von Beweisstrategien liefert.

[41] Diese Technik nennt man *equality propagation*.

[42] Da die Typzugehörigkeitsrelation unter anderem auch die volle Prädikatenlogik, die Rekursionstheorie und ein Teilmengenkonzept beinhaltet, ist Typisierbarkeit im allgemeinen unentscheidbar. Das wesentliche Problem ist dabei die Bestimmung der Definitionsbereiche von Funktionen. Soll zum Beispiel `f(0)`$\in$`T` bewiesen werden, so ist es oft nicht möglich, den Typ von `f` zu bestimmen, der für den Nachweis `f(0)`$\in$`T` erforderlich ist. Ist `f` auf der ganzen Menge $\mathbb{Z}$ definiert, oder nur auf einer Teilmenge davon, welche die Null enthält? Hierfür gibt es keine algorithmische Lösung, da sonst das Halteproblem entscheidbar wäre.
Eine ausführliche Diskussion dieser Problematik findet man in [Harper, 1985].

[43] Diese Vorgehensweise findet man leider im automatischen Theorembeweisen immer noch relativ häufig vor, da diese sich zum Ziel gesetzt haben, alle praktischen Probleme *vollautomatisch*, also ohne Interaktion mit einem Benutzer zu lösen.

4.4 Diskussion

Wir haben uns in diesem Kapitel mit den wesentlichen Techniken für eine Implementierung zuverlässiger interaktiver Beweissysteme und den prinzipiellen Möglichkeiten der Automatisierung der Beweisführung befaßt. Dabei ging es uns weniger um die Fähigkeiten *konkreter Strategien*, mit denen Beweise geführt und und Programme konstruiert werden können, als um die Techniken, mit denen solche Strategien auf eine zuverlässige und effiziente Weise innerhalb eines Beweisentwicklungssystems realisiert werden können.

Grundsätzlich empfiehlt es sich, Beweissysteme für ausdruckstarke formale Theorien wie die intuitionistische Typentheorie, als *interaktiv gesteuerte Systeme* anzulegen, da vollautomatische Systeme aufgrund der vielen Unentscheidbarkeiten ausdruckstarker Formalismen nicht sinnvoll sind. Stattdessen bietet es sich an, eine Kooperation mit einem Benutzer anzustreben, dem hierfür ein verständliches Interface zu der formalen, intern verarbeiteten Theorie und gewisse Möglichkeiten zur Automatisierung der Beweisführung angeboten werden müssen. Für wohlverstandene entscheidbare Teiltheorien des zugrundeliegenden formalen Kalküls sind *Entscheidungsprozeduren* ein sehr wichtiges Hilfsmittel, um einen Benutzer bei der Interaktion von einer Fülle trivialer Schritte zu entlasten. Das Konzept der *Taktiken* bietet wiederum die Möglichkeit einer benutzerdefinierten Erweiterung des Inferenzsystems, die eine sehr flexible und dennoch absolut sichere Einbettung beliebiger Beweisstrategien (bei verhältnismäßig geringen Effizienzverlusten gegenüber einem direkten – ungesicherten – Eingriff in das eigentliche Beweissystem) in das System ermöglicht.

Damit haben wir nun die Grundlagen für die Implementierung eines flexiblen *universellen Basissystems* geschaffen, mit dem Schlußfolgerungen über alle Bereiche der Mathematik und Programmierung formal gezogen und nach Belieben automatisiert werden können. Dieses kann nun durch ausgefeiltere Strategien, die mit den in diesem Kapitel besprochenen Techniken implementiert werden, zu einem leistungsfähigen semiautomatischen Inferenzsystem für ein beliebiges Anwendungsgebiet – wie etwa die Verifikation mathematischer Teiltheorien oder die rechnergestützte Entwicklung von Software – ausgebaut werden, ohne daß hierzu an dem eigentlichen System noch etwas verändert werden muß. Ein Großteil der weiteren Forschungen auf dem Gebiet der Automatisierung von Logik und Programmierung wird sich daher mit der *Anwendung* und der *praktischen Nutzbarmachung* dieses Systems befassen.

Andere Forschungsrichtungen zielen auf Techniken für elegantere Handhabung der Theorie als solche. Hierzu gehören einerseits Arbeiten an einer Verbesserung der in das System integrierten Mechanismen wie Abstraktion, Display und vor allem die Strukturierung der mathematischen Bibliothek, andererseits aber auch die Erforschung von mathematische Schlußfolgerungsmethoden, die von Menschen benutzt werden, bisher aber von der Theorie nicht erfaßt werden konnten. Hierzu zählt vor allem die Fähigkeit, Schlußfolgerungen über das Beweisen als solches zu ziehen, also das sogenannte *Meta-Schließen*. Dies bedeutet zum Beispiel, Aussagen über das Resultat einer Taktik-Anwendung zu treffen [Knoblock, 1987, Constable & Howe, 1990a] oder *Analogien* zwischen bestimmten Problemstellungen für eine Beweiskonstruktion auszunutzen [de la Tour & Kreitz, 1992]. Diese Forschungsrichtung wird besonders interessant, wenn man versucht, das Meta-Schließen durch *Reflektion* wieder in die eigentliche Theorie zu integrieren, also innerhalb der Theorie Schlüsse über Beweise der Theorie zu ziehen [Allen *et.al.*, 1990, Giunchiglia & Smaill, 1989] .[44] Hierdurch könnten Beweise noch effizienter, kürzer und eleganter gestaltet werden und, da sie sich noch mehr an eine menschliche Denkweise anlehnen, dazu genutzt werden, die Kooperation zwischen Mensch und Computersystem weiter verbessern.

4.5 Ergänzende Literatur

Das 1986 erschienene Buch über das Beweisentwicklungssystem NuPRL [Constable *et.al.*, 1986] kann als Referenzbuch für dieses Kapitel angesehen werden, obwohl es in manchen Implementierungsdetails mittlerweile nicht mehr ganz auf dem neuesten Stand ist. Die wichtigsten Änderungen und Ergänzungen sind in drei

[44] Da die Metasprache von NuPRL bereits als mathematische Programmiersprache `ML` formuliert wurde, ist eine weitere Formalisierung der Metatheorie und ihrer Gesetze prinzipiell möglich.

technischen Manuals [Jackson, 1993a, Jackson, 1993b, Jackson, 1993c] dokumentiert, die derzeit allerdings – genauso wie das NuPRL System selbst – ständig ergänzt werden.

Die Grundideen des taktischen Beweisens werden in [Gordon *et.al.*, 1979] ausführlich diskutiert. Die Ausarbeitung dieses Konzepts für typtentheoretische Beweiser wird in [Constable *et.al.*, 1985] beschrieben. Wertvolle Hinweise findet man auch in den Beschreibungen der in Fußnote 3 auf Seite 182 erwähnten Systeme.

Vertiefende Informationen zu den hier vorgestellten Entscheidungsprozeduren liefert [Constable *et.al.*, 1982], dessen Anhang "*An algorithm for checking PL/CV arithmetic inferences*" von Tat-hung Chan) besonders zu empfehlen ist. Die Arbeiten von Nelson und Oppen [Nelson & Oppen, 1979, Nelson & Oppen, 1980] liefern ebenfalls viele lohnenswerte Informationen.