Mechanization of Program Construction in Martin-Löf's Theory of Types

Andrew Ireland

Department of Computing Science
University of Stirling
1989

Submitted to the University of Stirling
in partial fulfilment of the requirements
for the Degree of Doctor of Philosophy

Abstract

The constructive approach to the problem of program correctness dates back to the late 1960's. During the early 1970's interest developed in the application of constructive logics to the derivation of provably correct programs. Martin-Löf's Theory of Types was devised as a formalisation of constructive mathematics. His theory also integrates the processes of program construction and verification within a single deductive system. This thesis is concerned with the application of Martin-Löf's theory to the task of program construction. In particular, the mechanisation of this task is investigated. We begin with a comparative study of current implementations of constructive type theory. The aim of this study is to assess the suitability of the implementations in the role of programming assistant. A proposal for a more effective programming assistant is presented. A principal difficulty in constructing correct programs is the problem of scale. Computer assistance plays an essential role in alleviating this problem. Experience in performing formal proof provides a better understanding of this problem and is, therefore, an important aid to the development of computer assistance. For this reason the formal derivation of a generalised table look-up function was undertaken. This exercise in program construction revealed that a disproportionate amount of the overall proof effort was taken up with proving negations. A proof of a negation has no computational content; it contributes only to the correctness of the synthesised program. With the aim of freeing the programmer from these proof obligations a decision method for negation was developed.

This decision method exploits, and thereby demonstrates, the uniform structure of Martin-Löf's theory. This uniformity is further utilized in a scheme for automatically deriving primitive recursive functions. The scheme enables the formal introduction of definitions during the course of a proof which satisfy the constraints of primitive recursion.

Declaration

I hereby declare that this thesis has been composed by myself, that the work reported has not been presented for any university degree before, and that the ideas that I do not attribute to others are due to myself.

Andrew Ireland
June 1989

Acknowledgements

I would like to thank my supervisor, Alan Hamilton, for his constant support and guidance throughout this project. I am indebted to him as both my teacher and colleague. I am also grateful to Charles Rattray for his encouragement during the course of my studies. In addition, I would like to thank Roland Backhouse and the members of the Groningen Wednesday Morning Club who provided useful feed-back on earlier aspects of my research. I am also grateful to the Computer Science Department at Heriot-Watt University, and in particular Stuart Anderson, for providing computing facilities during the early stages of this project. I also thank the Department of Artificial Intelligence at Edinburgh University for the time and facilities which enabled me to complete the writing of this thesis. My research was funded by a scholarship from *The Carnegie Trust for the Universities of Scotland*, to whom I owe a debt of gratitude. Finally, I thank Maria McCann for her encouragement, and for her tireless assistance in the preparation of this manuscript.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Program correctness

In the late 1960's Dijkstra[Dijkstra 68] questioned the verification approach to the problem of program correctness. The traditional verification paradigm was exemplified at the time by the work of [Floyd 67] [McCarthy & Painter 66] [Naur 66]. In his paper, *A Constructive Approach to the Problem of Program Correctness*, Dijkstra describes the verification paradigm in this way:

> "Given an algorithm and given specifications of its desired dynamic behaviour, prove then that the dynamic behaviour of the given algorithm meets the given specifications."

The constructive approach turns the problem of correction on its head:

> "Given the specifications of the desired dynamic behaviour, how do we derive from these an algorithm meeting them in its dynamic behaviour?"

Integrating the synthesis of the program with the proof of its correctness has practical value as well as intuitive appeal. For example, coding errors may be

identified sooner than would be the case if the tasks were separated. Furthermore, the constructive approach makes explicit the correspondence between inductive proof and recursive programs. In general, the constructive approach exploits the relationship which exists between formal proof and program development. A relationship which is absent from the verification paradigm, a fact noted by Scherlis and Scott[Scherlis & Scott 83]:

> "...this style of proof requires programmers to rediscover the insights that went into the original development of the program and express them in a formal logical language."

In contrast, the development of the program and the proof go "hand–in–hand" within the constructive approach. Gries[Gries 81] states this as a principle of program development.

A consequence of the constructive approach is that formal proof becomes a creative activity, since the structure of a correctness proof will effect the structure of the synthesised program. This is in contrast to the verification paradigm where the emphasis is placed on the existence of a proof.

Computer assistance has been developed for both approaches to the problem of program correctness. The Edinburgh LCF project[Gordon *et al* 79] and the Boyer-Moore theorem prover[Boyer & Moore 79] exemplify the verification paradigm. LCF is an interactive proof assistant. The logic of LCF, PPlambda, was developed by Scott in the late 1960's as a basis for reasoning about computable partial functions. Proof strategies are expressed within a metalanguage (ML). The significance of LCF is that it provides a general methodology for computer assisted proof[Milner 85]. Proof assistants have been derived from the

LCF project[Gordon 83] [Paulson 83] [Petersson 82]. A survey of the LCF family of proof assistants is presented in [Paulson 85]. In contrast, the Boyer-Moore theorem prover is an automated system, designed for proving properties about recursive programs. Their logic is based on a form of recursive function theory which excludes existential quantification. Two basic proof techniques are employed: rewrite rules and induction. Boyer and Moore make use of failure in the application of rewrite rules to guide the selection of an appropriate induction scheme. In the case of both LCF and the Boyer-Moore theorem prover, the properties to be proven are separated from the program definition. An alternative approach, adopted by the Gypsy verification environment[Good 85], is to incorporate verification conditions within the program text. A similar approach is used by the Stanford Pascal verifier[Igarashi *et al* 75].

Manna and Waldinger's[Manna & Waldinger 80] deductive approach to program synthesis follows the constructive approach to program correctness. Their system provides automatic synthesis of recursive programs through the repeated transformation of an initial specification. They combine techniques of theorem proving and transformation systems[Burstall & Darlington 77] within a single deductive system.

During the early 1970's interest developed in using constructive logics as a basis for deriving provably correct programs. Constable, motivated by Bishop's work on constructive analysis[Bishop 67], investigated the relationship between constructive mathematics and programming[Constable 71]. This work led to the development of a programming refinement logic embodied in the NuPRL

interactive proof assistant[Constable *et al* 86], which is reviewed in chapter 3.

Independently, the Swedish logician Per Martin-Löf developed a constructive theory of types[Martin-Löf 75][Martin-Löf 82]. While Constable was motivated by mathematics and computing science, Martin-Löf was predominately interested in foundational issues of constructive mathematics. It is the programming methodology group in Göteborg, and more recently Roland Backhouse *et al* in Groningen, who have promoted Martin-Löf's work within the computing science community. Coquand and Huet's Theory of Constructions[Coquand & Huet 85] is in the style of Matin-Löf's theory, but their interest is mathematical. Implementations of Martin-Löf's theory exist[Hamilton 85] [Paulson 86] [Petersson 82] and are reviewed in chapter 3. Much more work on computer assisted proof is necessary, however, if the constructive logics are to provide an effective basis for tackling the problem of program correctness. This thesis is concerned with the development of such computer assistance within Martin-Löf's theory of constructive types.

1.2 Programming as constructive proof

The development of constructive logic was pioneered by the Dutch mathematician L.E.J. Brouwer[Brouwer 75] [Brouwer 81] under the name of "intuitionism". Constructive logics are characterised by the rejection of the *a priori* notion of truth associated with classical logic. Classically, a proposition, by definition, is ascribed a Boolean truth value independently of whether an affirmation or denial can be exhibited. As a direct consequence, the law of the excluded middle,

expressed by the proposition

$$P \vee \neg P$$

is classically true, irrespective of whether either disjunct can be justified. In contrast, the constructive notion of truth is identified with the proof of a proposition. Constructively, a proposition is defined by prescribing what counts as a proof. In proving a proposition a proof is made explicit. What counts as a constructive proof of the classical tautology given above is either a proof of P, or a proof of $\neg P$, or an effective method which yields such a proof. The notion of an effective or computable method is fundamental to the explanation of constructive logics. It is for this reason that constructive proofs are identified with programs, and propositions with program specifications. As a result, constructive truth can be interpreted as satisfiability in the sense of a program meeting a specification. In terms of program construction the rejection of the law of the excluded middle makes sense. To allow the law of the excluded middle would lead, in general, to programs which are not executable. For a comprehensive introduction to constructive logics the reader is referred to [Beeson 85] [Dummett 77] [Heyting 56].

1.3 Mechanisation of constructive proof

Formal proof is a rigorous activity demanding precise and detailed argument. Computer assistance is required in order to manage the complexity and to ensure correctness in performing formal proof. The application of constructive logic to the task of program development places an extra burden on the role of formal proof, as is noted by Petersson and Smith[Petersson & Smith 86]. Firstly, in

comparison to conventional verification proofs, the level of detail required is greater since a proof has computational content. Secondly, control over the search for a proof is necessary, since the *shape* of a proof is reflected in the structure of the synthesised program. This additional burden makes the need for computer assistance more acute. The change in emphasis away from simply establishing the existence of a proof, to the search for a particular proof, has consequences for the role of the computer in performing formal proof. Instead of automation there is a greater need for interaction. Milner[Milner 87] refers to the "dialogue between a user and a machine in building or performing a formal proof". In what is an exploratory activity, the machine should, as Milner says, "amplify the user's power to discover proofs". This emphasis on the need for "interactive tools", as opposed to complete automation, is also promoted by Scherlis and Scott[Scherlis & Scott 83]. The techniques of automatic proof, such as Boyer and Moore's work on induction, should not be dismissed. They should, however, be applied selectively and form part of a "set of tools" provided within the context of an interactive proof system.

1.4 Thesis overview

Aspects of Martin-Löf's theory of types are presented in chapter 2. The theory is characterised by the proposition-as-types principle. We emphasize the fundamental role this principle plays in the identification of programs with proofs. Underlying the theory of types is a general theory of expressions. This theory is outlined. The judgement forms and deductive system are presented. A

note on derivations is included in which the distinction between the notions of propositional and judgemental proof is clarified.

In chapter 3 a comparative study of current type theory implementations is presented. The objective of this study is to assess the suitability of current implementations in the role of programming assistant. The conclusions drawn from the study form the basis of a proposal for a more effective programming assistant.

Scale is a principal difficulty in deriving provably correct programs. Computer assistance plays an important role in overcoming this problem. With the aim of identifying areas where computer assistance may be useful, the derivation of a generalised table look-up function was undertaken. The details and conclusions drawn from this programming exercise are presented in chapter 4.

The exercise in program construction revealed difficulties in proving negations which led to the development of a decision method for negation. The design of the decision method and its application to the negations arising from the programming exercise are documented in chapter 5.

The decision method exploits the uniform structure of Martin-Löf's theory. This uniformity is further utilised in chapter 6, where a scheme is presented for automatically deriving a primitive recursive program from a specification expressed as a type. Chapter 7 summarises the work presented in this thesis and suggests topics for future research.

Chapter 2

Aspects of Type Theory

2.1 Introduction

Martin-Löf's constructive theory of types is characterised by the identification of a proposition with the type of its proofs. This *propositions-as-types* principle was first noted by Curry and Feys[Curry & Feys 58]. During the late 1960's and early 1970's this idea was extended by [Howard 80] [Martin-Löf 75] [Scott 70]. In the case of Martin-Löf's work, a type can be viewed as an extension to the conventional programming notion of a type, in which a programming task can be completely specified. A type is a representation of a proposition and at the same time it specifies a computation. Unlike Bishop's[Bishop 67] formalisation of constructive logic, where the computational content of a proof is left implicit, Martin-Löf makes the constructive justification for a proposition explicit. A constructive justification is expressed in terms of an object language. In general, an object expression will have computational content. The relationship between

types and objects is formalised at the level of judgements. Assertions are made within the logic through the judgement forms. In terms of program construction, the judgement form relates a program to its specification. Program derivation and the derivation of a judgement are, therefore, one and the same.

The remainder of this chapter expands on the aspects of type theory outlined above. The notions of propositions–as–types and constructive justification are examined in section 2.2. Object and type expressions are introduced formally in section 2.3. Type theory is underpinned by a general theory of expressions, the details of which are outlined in section 2.3.1. This theory shows how type and object expressions may be built. The notion of evaluation, on which the explanation of judgement forms is based, is outlined in section 2.3.2. The judgement forms are presented in section 2.4 and the deductive system is described in section 2.5. In section 2.6 an example derivation is presented and the distinction between propositional and judgemental proof is clarified.

2.2 Propositions as types

Constructively, a proposition is defined by laying down what constitutes a proof. This means that the logical constants and quantifiers are given new constructive interpretations. It is these constructive interpretations which give rise to the identification of propositions with types.

Logicals

We begin with logical conjunction. A justification of the proposition

$$A \wedge B$$

takes the form of a pair $\langle a, b \rangle$, where a and b denote justifications of propositions A and B respectively. As noted in section 2.1, a proposition is represented by the type of its proofs. Therefore, the above conjunction is represented by the type

$$A \times B$$

and consequently logical conjunction is identified with the cartesian product type.

In the case of logical disjunction:

$$A \vee B$$

a justification takes the form of an injection. Given a justification of A, denoted by a, then $inl(a)$ is a justification of the disjunction. Similarly, given b, a justification of B, then the right injection, $inr(b)$, denotes a justification. Logical disjunction is identified, therefore, with the disjoint union of two types, and is expressed in this case by the type

$$A + B$$

What counts as a proof of logical implication:

$$A \Rightarrow B$$

is a function which maps a justification of A into a justification of B. Such a

mapping is represented by the λ-expression

$$\lambda x.b(x)$$

This is the object level representation of a function, where x is the input argument and $b(x)$, the body of the function, is an expression in which x may or may not occur free. Implication is identified with the function type:

$$A \rightarrow B$$

Negation is not a primitive of the theory; it is defined in terms of the function and empty types. For instance, the proposition $\neg A$ is represented by the type

$$A \rightarrow \emptyset$$

A justification of this negation is a function which maps an arbitrary object in A into an object in $\emptyset$, the type with no members. The empty type is identified with *falsehood*, the proposition with no proofs. For this reason such a construction is referred to as an *absurdity* proof.

Quantifiers

The power of type theory as a specification language comes from the notion of a dependent type which is required in order to support quantification.

A justification of an existentially quantified statement:

$$(\exists x : A)B(x)$$

is a pair $\langle a, b \rangle$, where a denotes a justification for A, and b denotes a justification for $B(a)$. Existential quantification introduces a dependency between

justifications. The justification of $B(a)$ depends upon a, the justification of A (the chosen existential witness). The existential statement is represented by the disjoint union of a family of types:

$$(\Sigma x : A)B(x)$$

Here B denotes a family of types which is indexed by the elements of A. In his *Notes on Structured Programming* Dijkstra[Dijkstra 72] states that he prefers to regard a program "not so much as an isolated object, but rather as a member of a family of related programs". The disjoint union of a family of types formalises this idea. An instance of the disjoint union of a family of types specifies a family of related programs. A particular member of the family of programs is selected by refinement of the specification.

A justification of an universally quantified statement:

$$(\forall x : A)B(x)$$

is a function which maps a, an arbitrary justification of A, into a justification for $B(a)$. Universal quantification is represented by the cartesian product of a family of types:

$$(\Pi x : A)B(x)$$

This type specifies a function space in which the range type is dependent upon the particular element of the domain type supplied to the function. For instance, if a denotes an object in A, and a is supplied to the function, then the domain type is $B(a)$. This dependent function space is a more general notion of function than is supported by conventional programming languages. An object in the

cartesian product of a family of types takes the form

$$\lambda x.b(x)$$

Data structures

The constructive interpretation of the logicals and quantifiers presented above leads to a degree of duality. For instance, while the cartesian product

$$A \times B$$

represents logical conjunction, it is also identified with the notion of a record data structure. Furthermore, the disjoint union type is analogous to the notion of the variant record data structure. In his *Notes on Data Structuring* Hoare[Hoare 72] points out the close relationship that exists between the logicals and data structures. Martin-Löf's theory makes provision for other data types traditional to programming languages: the type of natural numbers N, and the finite types denoted by $\{\ldots\}$ in which enumeration types may be defined. The transfinite induction, or W-type, in which lists and trees can be defined, is also included. The W-type, as pointed out by Martin-Löf[Martin-Löf 82], has no direct analogy with current programming languages.

Program structures

The constructive justifications presented so far have been restricted to *canonical* forms; objects which evaluate to themselves (data elements). Each type is associated with a *noncanonical* form; a method for reducing an arbitrary justification to its canonical form. For instance, the inductively defined type of natural

numbers N has two canonical constructors 0 and $'$ (successor). The associated noncanonical constructor (or selector) *nrec* allows us to justify a proposition $P(x)$, where x denotes an arbitrary element of type N. The associated proof rule defines induction over N. The close relationship which exists between induction and recursion means that a justification constructed using *nrec* is a primitive recursive program. For example, the justification of the proposition $P(x)$ takes the form

$$nrec(x, b, [u, v]d(u, v))$$

This noncanonical form corresponds to the following more traditional primitive recursive definition

$$\begin{cases} f(0) = b \\ f(n') = d(n, f(n)) \end{cases}$$

Informally, the noncanonical expression is evaluated by first evaluating x. If the result of this evaluation is 0, then b is evaluated. Otherwise x must take the form n' and evaluation proceeds with d in a context in which u is bound to n and v is bound to $nrec(n, b, [u, v]d(u, v))$, the value of f at n.

A consequence of the close relationship between induction and recursion, which is reflected in Martin-Löf's type structure, is that general recursion is not available. The provision for higher-order functions makes primitive recursion less of a restriction. This is demonstrated by Smith[Smith 83].

Propositional equality

Given that a and b are justifications of a proposition A, then the proposition

$$a = b$$

is true only if a and b evaluate to the same canonical value. The associated justification e carries no computational significance. It simply maintains the uniform structure of the theory. Propositional equality is represented by the equality type:

$$Eq(A, a, b)$$

This type is interpreted as: "a and b are equal elements of type A".

Universes

Type theory provides for a cumulative hierarchy of universes $U_1, U_2, \ldots$. This hierarchy is required to enable the notion of a "type of types" to be expressed. The first universe, the type of small types, contains as canonical elements the types defined above. Similarly, the canonical elements of U_n are the types formed from the canonical types in U_{n-1}, together with U_{n-1} itself.

A note on decidability

The rich type structure described above provides an expressive specification language. The price paid for this expressiveness, however, is the loss of decidable type checking. Given an arbitrary object a, it is not possible to determine its type. Furthermore, given an object a and a type A, it is not decidable, in general, whether a is a member of A. The well-formedness of the equality type depends upon type membership. Consequently, the well-formedness of an arbitrary type expression is also not decidable.

2.3 Formal expressions in Type Theory

In the preceding section types were informally introduced as a representation for propositions and objects for constructive justifications. This section introduces object and type expressions more formally.

2.3.1 A theory of expressions

Underlying type theory is a general theory of expressions. The theory of expressions prescribes how object and type expressions may be constructed. A full account of the theory is presented in [Nordström *et al* 86]. Expressions are built up by means of abstraction and application. For instance, consider an expression e which contains free variables $x_1, \ldots, x_n$. Abstracting on the free variables $x_1, \ldots, x_n$ gives rise to an expression of the form

$$[x_1, \ldots, x_n]e$$

The application of the e to the expressions $e_1, \ldots, e_n$ is written as

$$e(e_1, \ldots, e_n)$$

Definitional equality between expressions is provided and involves α-, β- and η-reduction. For instance, the application of the abstraction given above to the free variables $x_1, \ldots, x_n$ is definitionally equal to e. This equality relationship is expressed as

$$([x_1, \ldots, x_n]e)(x_1, \ldots, x_n) \equiv e$$

Expressions which can be applied to other expressions are said to be *unsaturated*. For instance, the expression

$$[x]x + 1$$

is *unsaturated* whereas the application

$$([x]x + 1)(2)$$

denotes a *fully saturated* expression which is definitionally equal to $2 + 1$. An arity, which is a primitive form of typing, is associated with an expression to ensure correct application.

2.3.2 Type and object expressions

Type and object level expressions may be built up by abstraction and application, as described in the preceding section. The types and object expressions presented in section 2.2 are, however, abbreviations for object and type expressions. For instance, the expression

$$(\Pi x : A)B(x)$$

is an abbreviation for the type expression

$$\Pi(A, [x]B(x))$$

which is formed by the application of the Π operator (type constructor) to the type expression A and the abstraction $[x]B(x)$. Similarly, the expression

$$\lambda x.b(x)$$

is an abbreviation for the object expression

$$\lambda([x]b(x))$$

Expression	*Abbreviation*
$\times(A, B)$	$A \times B$
$\Sigma(A, [x]B(x))$	$(\Sigma x : A)B(x)$
$pair(a, b)$	$\langle a, b \rangle$
$+(A, B)$	$A + B$
$\rightarrow(A, B)$	$A \rightarrow B$
$\Pi(A, [x]B(x))$	$(\Pi x : A)B(x)$
$\lambda([x]b)$	$\lambda x.b$
$apply(f, a)$	$f[a]$
$W(A, [x]B(x))$	$(W x : A)B(x)$
$sup(a, b)$	$\langle\|a, b\|\rangle$
$succ(x)$	x'
$cons(u, v)$	$u :: v$

Table 2.1: Notational abbreviations

which is formed by the application of the λ operator to the expression obtained by abstracting for all free occurrences of x in b. These abbreviations simplify the presentation of formulae. A summary of the abbreviations which are used throughout this thesis is given in table 2.1.

As noted in section 2.2, the computational nature of constructive proof is reflected at the object level, where an object expression is either canonical (data) or noncanonical (program). A consequence of the abstraction mechanism is that expressions cannot be evaluated from within since, in general, the body of an abstraction may contain uninstantiated variables. To overcome this difficulty a strategy of lasy evaluation is employed. Therefore, an expression is judged to be canonical by its outer structure. For example, the expression $\lambda x.b$ is canonical irrespective of whether b can be evaluated further. For this reason a distinction between evaluated and fully evaluated expressions is necessary. An expression is

said to be *fully evaluated* if all its saturated subexpressions are fully evaluated. For instance, the expression

$$\lambda x.nrec(x', x, [u, v]u)$$

is canonical, but not fully evaluated, whereas

$$\lambda x.x$$

is both canonical and fully evaluated.

2.4 The judgement forms

Assertions are expressed in the theory as judgements, of which there are four:

- A is a type (A *type*)
- A and B are equal types ($A = B$)
- a is an object of type A ($a : A$)
- a and b are equal objects of type A ($a = b : A$)

A consequence of the propositions-as-types interpretation is that the judgements are open to the following more mathematical oriented readings:

- A is a proposition
- A and B are equal propositions
- a is a justification of the proposition A
- a and b are equal justifications of the proposition A

Similarly, the types-as-specifications interpretation gives rise to the following more programming oriented readings:

A is a specification

A and B are equal specifications

a is a program satisfying the specification A

a and b are equal programs satisfying the specification A

What follows are descriptions, paraphrased from [Martin-Löf 82], of the conditions under which the four judgements can be made:

A is a type (A *type*)

> "A canonical type A is defined by prescribing how a canonical object of type A is formed as well as how two equal canonical objects of type A are formed."

The only limitation Martin-Löf places on this definition is that equality between canonical objects of type A is reflexive, symmetric and transitive.

A and B are equal types ($A = B$)

> "Two canonical types A and B are equal if a canonical object of type A is also a canonical object of type B and, moreover, equal canonical objects of type A are also equal canonical objects of type B, and vice versa."

a is an object of type A ($a : A$)

Assuming that A is a type, then

> "...a judgement of the form
>
> $$a : A$$
>
> means that a has a canonical object of the canonical type denoted by A as value."

a and b are equal objects of type A ($a = b : A$)

Assuming that A is a type, and that a and b are objects of type A, then

> "...a judgement of the form
>
> $$a = b : A$$
>
> means that a and b have equal canonical objects of the canonical type denoted by A as values."

In general, a judgement may depend upon assumptions. An assumption takes the form

$$x : A$$

where x is a variable and A denotes a type. We adopt a notion of scoping, introduced by Backhouse[Backhouse 86a], to represent hypothetical judgements. For instance, the informal judgement: "$A(x)$ is a type assuming that x is in B" is expressed formally as

$$||[x : B \triangleright\ A(x)\ type]||$$

Note that A may or may not depend upon x. The ||[and]|| delimit the scope of assumptions and the "$\triangleright$" symbol separates assumptions from the conclusion. Collectively, the assumptions denote a context. The order in which assumptions appear within a context is important, since dependencies may exist. For example, the judgement

$$||[x : A;\ y : B(x) \triangleright\ C(x, y)\ type]||$$

expresses the assertion: "C is a type assuming that x is in type A and y is in type $B(x)$". A complete explanation of hypothetical judgements is presented in [Martin-Löf 82].

2.5 The deductive system

The rules of type theory are presented in a natural deduction style [Gentzen 69]. The general form of a rule is

$$\frac{P_1 \ \ldots \ P_n}{C}$$

where the premises $P_1, \ldots, P_n$ and the conclusion C denote judgements. A type is defined by four sets of rules:

- formation
- introduction
- elimination
- computation

These rules exhibit a rich structure. The rules of Martin-Löf's deduction system are presented in appendix A. To illustrate the structure of the deductive system the rules for the type of lists are presented. Although not included as a primitive in [Martin-Löf 82], rules for lists are presented in [Martin-Löf 84] in order to demonstrate the uniformity which the inductive definition of a type provides:

> "We can follow the same pattern used to define natural numbers to introduce other inductively defined sets[1]."

It is this uniform pattern of the type theory rules which Backhouse[Backhouse 86b] exploits in his scheme for automatically inferring elimination and computation rules from the formation and introduction rules. This uniformity owes much

[1] Note that the use of *type* in [Martin-Löf 82] it is replaced in [Martin-Löf 84] by the use of *set*.

to the theory of expressions which underlies it. To convey this uniformity, the presentation of the *List* data type includes rule schemas. These schemas are generalisations of the four kinds of rules. For each formation, introduction and elimination rule presented, an associated rule defining equality also exists. We, however, omit the equality versions here. This presentation owes much to Backhouse's explanation of the type theory rules.

Formation rules

A formation rule prescribes how to construct well-formed types. The formation rule for lists takes the form

$$\frac{A\ type}{List(A)\ type}\ List\text{-formation}$$

Given that A is a well-formed type, then this rule licences the construction of the type of lists with base type A. Using Backhouse's terminology, A denotes a formation variable. In general, a formation rule is given by the rule schema

$$\frac{A_1\ type \cdots A_m\ type}{\Theta(A_1, \ldots, A_m)\ type}\ \Theta\text{-formation}$$

where Θ denotes an arbitrary type constructor, and $A_1, \ldots, A_m$ denote the formation variables. By way of a notational short-hand, we let $\vec{A}$ denote the list of formation variables $A_1, \ldots, A_m$. Consequently, $\Theta(A_1, \ldots, A_m)$ may be written as $\Theta(\vec{A})$. If Θ is a nullary type constructor, then the schema presented above reduces to an axiom.

Introduction rules

An introduction rule prescribes how canonical (data) objects are constructed. The *List* type has two introduction rules defining the empty list and compound lists:

$$\frac{A\ type}{nil : List(A)}\ List\text{-introduction}_{nil}$$

$$\frac{a : A \qquad b : List(A)}{a :: b : List(A)}\ List\text{-introduction}_{::}$$

In the case of a nullary constructor, such as *nil*, the premises are the same as for the formation rule. Given that Θ denotes an arbitrary type constructor with k introduction rules, and assuming that the i^{th} canonical constructor is nullary, then the i^{th} introduction rule is given by the schema

$$\frac{A_1\ type \quad \cdots \quad A_m\ type}{\theta_i : \Theta(A_1, \ldots, A_m)}\ \Theta\text{-introduction}_{\theta_i}$$

A non-nullary object constructor has associated introduction variables, as described Backhouse. In the case of *List*-introduction$_{::}$ two introduction variables exist, a and b. The corresponding rule schema takes the form

$$\frac{\begin{array}{ll} & \langle \text{well-formedness premises} \rangle \\ 1 & b_{i1} : B_{i1} \\ & \vdots \\ n_i & b_{in_i} : B_{in_i} \end{array}}{\theta_i(b_{i1}, \ldots, b_{in_i}) : \Theta(\bar{A})}\ \Theta\text{-introduction}_{\theta_i}$$

where the j^{th} premise ($1 \leq j \leq n_i$) either takes the form

$$b_{ij} : A$$

where $A \in \{A_1, \ldots, A_m\}$, or

$$b_{ij} : \Theta(\vec{A})$$

In the latter case, the j^{th} premise is said to be recursive, where b_{ij} denotes a recursive introduction variable. In the absence of a recursive premise, well-formedness premises are required in addition to the main premises. For each member of $\{A_1, \ldots, A_m\}$ for which there does not exist a premise of the form $a : A$, a well-formedness premise is included. When it is appropriate to distinguish between introduction variables we let u_{ij} $(1 \leq j \leq p_i)$ range over the non-recursive variables, and v_{ij} $(1 \leq j \leq q_i)$ range over the recursive variables. Using this convention Θ-introduction$_{\theta_i}$ may be written as

$$\frac{\begin{array}{l}\langle\text{well-formedness premises}\rangle \\ u_{i1} : A_{i1} \\ \vdots \\ u_{ip_i} : A_{ip_i} \\ v_{i1} : \Theta(\vec{A}) \\ \vdots \\ v_{iq_i} : \Theta(\vec{A})\end{array}}{\theta_i(a_i, a_i) : \Theta(\vec{A})} \; \Theta\text{-introduction}_{\theta_i}$$

Elimination rules

An elimination rule prescribes how noncanonical (program) objects are constructed. Each type constructor has an unique most general eliminator, which in the case of the *List* type is *listrec*. The *listrec* operator is defined as follows

$$\frac{\begin{array}{l} l : \mathit{List}(A) \\ b : D(\mathit{nil}) \\ [\![u : A;\ v : \mathit{List}(A);\ w : D(v) \\ \triangleright\quad d(u, v, w) : D(u :: v) \\ ]\!] \end{array}}{\mathit{listrec}(l, b, d) : D(l)}\ \mathit{List}\text{-elimination}$$

Note that d is definitionally equal to the abstraction $[u, v, w]d(u, v, w)$. This rule defines induction over lists, where w denotes the inductive hypothesis. Backhouse calls this the "elimination variable". For each recursive introduction variable there exists an elimination variable. For an arbitrary type constructor Θ, with k associated introduction rules, the elimination rule will have to deal with k cases. The general elimination rule schema, therefore, takes the form

$$\frac{\begin{array}{ll} 0 & x : \Theta(\lambda) \\ 1 & [\![C_1 \\ & \triangleright\ s_1(\alpha_1, \sigma_1, \omega_1) : D(\theta_1(\alpha_1, \sigma_1)) \\ &]\!] \\ & \vdots \\ k & [\![C_k \\ & \triangleright\ s_k(\alpha_k, \sigma_k, \omega_k) : D(\theta_k(\alpha_k, \sigma_k)) \\ &]\!] \end{array}}{\Theta\mathit{rec}(x, s_1, \ldots, s_k) : D(x)}\ \Theta\text{-elimination}$$

where $C_1, \ldots, C_k$ denote contexts. Note that s_i is definitionally equal to the abstraction $[\alpha_i, \sigma_i, \omega_i]s_i(\alpha_i, \sigma_i, \omega_i)$. The construction of the i^{th} premise $(1 \leq i \leq k)$ is as follows: If θ_i is a nullary constructor then C_i is an empty context and the i^{th} premise takes the form

$$s_i : D(\theta_i)$$

Alternatively, if θ_i is a non-nullary constructor, and Θ-introduction$_{\theta_i}$ has no recursive premises, then the i^{th} premise takes the form

$$[\![u_{i1} : A_{i1}; \ldots; u_{ip_i} : A_{ip_i} \triangleright\ s_i(\vec{a}_i) : D(\theta_i(\vec{a}_i))]\!]$$

Finally, in the case where Θ-introduction$_{\theta_i}$ has recursive premises, then the i^{th} premise takes the form

$$\begin{array}{l}[\![u_{i1} : A_{i1}; \ldots; u_{ip_i} : A_{ip_i} \\ ; v_{i1} : \Theta(\vec{A}); \ldots; v_{iq_i} : \Theta(\vec{A}) \\ ; w_{i1} : D(v_{i1}); \ldots; w_{iq_i} : D(v_{iq_i}) \\ \triangleright\ s_i(\vec{a}_i, \vec{a}_i, \vec{w}_i) : D(\theta_i(\vec{a}_i, \vec{a}_i)) \\ ]\!]\end{array}$$

Computation rules

A computation rule prescribes how an arbitrary object is reduced to a canonical form. In terms of programming, a computation rule defines the execution of the program construct introduced by the elimination rule. The eliminator only permits the construction of total functions, so a separate computation rule is required for each canonical form. The *List* type, therefore, has two computation rules:

$$\frac{\begin{array}{l}b : D(nil) \\ [\![u : A;\ v : List(A);\ w : D(v) \\ \triangleright\quad d(u, v, w) : D(u :: v) \\ ]\!]\end{array}}{listrec(nil, b, d) = b : D(nil)}\ List\text{-computation}_{nil}$$

$$\frac{\begin{array}{l} h : A \\ t : List(A) \\ b : D(nil) \\ [\![u : A;\ v : List(A);\ w : D(v) \\ \triangleright\quad d(u, v, w) : D(u :: v) \\ ]\!] \end{array}}{listrec(h :: t, b, d) = d(h, t, listrec(t, b, d)) : D(h :: t)}\ List\text{-computation}_{::}$$

As a result of the one–to–one correspondence between introduction and computation rules, an arbitrary type constructor Θ with k introduction rules has k associated computation rules. Assuming that the i^{th} constructor θ_i $(1 \leq i \leq k)$ has p_i associated non-recursive introduction variables denoted by $\bar{a}$, and q_i recursive introduction variables denoted by $\bar{b}$, then the i^{th} computation rule takes the form

$$\frac{\begin{array}{l} \langle \text{well-formedness premises} \rangle \\ a_{i1} : A_{i1} \\ \vdots \\ a_{ip_i} : A_{ip_i} \\ b_{i1} : \Theta(\bar{A}) \\ \vdots \\ b_{iq_i} : \Theta(\bar{A}) \\ [\![C_1 \\ \triangleright\ z_1(\bar{a}_1, \bar{c}_1, \bar{w}_1) : D(\theta_1(\bar{a}_1, \bar{c}_1)) \\ ]\!] \\ \vdots \\ [\![C_k \\ \triangleright\ z_k(\bar{a}_k, \bar{c}_k, \bar{w}_k) : D(\theta_k(\bar{a}_k, \bar{c}_k)) \\ ]\!] \end{array}}{\begin{array}{l} \Theta rec(\theta_i(\bar{a}, \bar{b}), z_1, \ldots, z_k) \\ \qquad = z_i(\bar{a}, \bar{b}, \Theta rec(b_{i1}, z_1, \ldots, z_k) \\ \qquad\qquad \vdots \\ \qquad\qquad \Theta rec(b_{iq_i}, z_1, \ldots, z_k) : D(\theta_i(\bar{a}, \bar{b})) \end{array}}\ \Theta\text{-computation}_{\theta_i}$$

The construction of the contexts $C_1, \ldots, C_k$ in the above schema is the same as for the elimination rule.

2.6 A note on derivations

The scoping mechanism introduced by Backhouse provides a uniform notation for expressing hypothetical judgements and derivations. For instance, consider the following derivation of a length function for lists:

Derivation
0.0 $\quad\|l : \mathit{List}(A)$
0.1.0 $\quad\triangleright\quad\|u : A;\ v : \mathit{List}(A);\ w : N$
$\quad\quad\triangleright\quad\{0.1.0_2\ N\text{-intr}'\}$
0.1.1 $\quad\quad\quad w' : N$
$\quad\quad\quad\|$
$\quad\quad\quad\{0.0, N\text{-intr}_0, 0.1\ \mathit{List}\text{-elim}\}$
0.2 $\quad\mathit{listrec}(l, 0, [u, v, w]w') : N$
$\quad\quad\|$

The combination of indentation and nested contexts reflects the introduction and discharge of assumptions. Individual steps in the derivation are indicated using braces. The numbering serves two purposes: It enables deductions to be referenced and conveys the depth of context nesting. Note that the deduction which gives rise to step 0.1.1 uses a subscript to indicate the particular assumption within line 0.1.0 which is required in the application of N-introduction$'$.

A confusing aspect of type theory is that *proof* has two meanings. On the one hand there is the propositional notion of proof; a proposition is true if a proof object can be demonstrated to belong to the associated type. On the other hand, there is the judgemental notion of proof; the demonstration that a certain formula is well-formed or a particular object belongs to a type. These notions of proof are not unrelated. A proof object summarises the constructive component of the derivation of the judgement in which it appears. Where the intended meaning is ambiguous we will talk instead about the *justification* of a

proposition and the *derivation* of a judgement.

Chapter 3

Aspects of the Implementation of Type Theory

In this chapter we compare and contrast current implementations of constructive type theory. The aim of the study is to evaluate the suitability of current implementations in the role of programming assistant. Desirable objectives of a programming assistant are set out in section 3.1. These objectives form the basis of a review of current implementations presented in section 3.2. The conclusions drawn from this review are presented in section 3.3 and form the basis for a proposal for an implementation which more closely meets the objectives set out in section 3.1.

3.1 Objectives of a programming assistant

We categorise desirable objectives of a programming assistant under the four headings of programming:

- logic
- framework
- tools
- interface

Each category is discussed in the following sections.

3.1.1 Programming logic

The central idea which underpins programming as constructive proof is the identification of propositions with types (specifications) and proofs with objects (programs). In mathematics emphasis is placed on the existence of proofs. The method of proof is of secondary importance. If our objective is program construction, the significance lies not in the existence of a proof, but in its structure. Therefore, it seems natural that the significance of proof objects should be reflected in the logic by giving equal prominence to both object and types within the assertion form.

As noted by Schmidt[Schmidt 84], the task of proof development involves completing a partial proof tree:

> "Both the leaves (assumptions) and root (conclusion) of the tree are known, and the proof is obtained by filling in the interior nodes of the tree in any order desired."

To achieve this style of deduction it is essential that the logical system accommodates both forwards and backwards inference.

The logic should reflect the needs of the programmer by providing a suitably rich set of data types. No collection of data types, however, can be complete. Application dependent data structures will arise. Deriving new data types in terms of existing ones is a possible approach. Khamiss[Khamiss 86] has investigated the use of Martin-Löf's W-type as a basis for representing parse trees. Although this is a sound approach, in practise the specialisation of a general data type leads to redundancy and, as a result, increases the complexity of formal proof. An alternative approach is to introduce new type constructors directly. This approach is exemplified by the work of [Chisholm 87] [Dyckhoff 85] [Nordström 85] and is advocated by Backhouse[Backhouse 86b]. Extending a theory by-hand in this way, however, may unknowingly lead to the introduction of inconsistencies. Therefore, it is desirable that the core logic is uniform in structure, in order to ease the process of introducing extensions to the type structure while maintaining its integrity.

A consequence of the richness of Martin-Löf's theory is that the well-formedness of an arbitrary formula is not decidable. Therefore, it is important that these proof obligations are minimised.

Notational economy may simplify the implementation and formalisation of meta-theoretical results. Such minimalism, however, has consequences for the way in which formal proof is conducted. If our objective is an interactive system, then it seems important that clarity should take precedence over minimalism.

3.1.2 Programming framework

Two styles of program (proof) construction exist. Firstly, working from the axioms and assumptions, the desired program (proof) may be constructed by the application of the inference rules. This is known as *forwards proof* and lends itself more naturally to verification than proof discovery. An alternative approach is to begin with the desired goal (specification) and refine it repeatedly until it is reduced to the level of axioms and known theorems. The similarity between this *backwards* style of proof and general goal-seeking activities is noted by Milner[Milner 85]. Milner develops a theory of goal-seeking in which a *goal* is satisfied by an *event*. Furthermore, a relationship of *achievement* is defined between events and goals:

$$\text{achieves} \subseteq \text{event} \times \text{goal}$$

Goal refinement is carried out by partial functions known as *tactics*:

$$\text{tactic} = \text{goal} \rightharpoonup \text{goal list} \times \text{procedure}$$

It is the *procedure* generated by the application of a tactic to a goal which defines the achievement relationship. The procedure maps achievements of the subgoals into an achievement of the main goal:

$$\text{procedure} \subseteq \text{event list} \rightharpoonup \text{event}$$

This *goal-directed* proof methodology is embedded within the Edinburgh LCF Proof System. Having a goal to achieve gives us a handle on the problem. Therefore, goal-directed proof is more appropriate for program construction, while forwards proof is oriented more towards program verification.

Goal-directed proof is analogous to the *step-wise* approach to program construction. It is an approach which involves progressive refinements of an initial problem specification. Commitment to design decisions can be delayed until the problem has been analysed further. This structured approach to program construction, advocated by Dijkstra[Dijkstra 76] and Gries[Gries 81], is illustrated in [Dijkstra 72] where the benefits are described by Dijkstra as follows:

> "...in the step-wise approach it is suggested that even in the case of a well-defined task, certain aspects of the given problem statement are ignored at the beginning. That means that the programmer does not regard the given task as an isolated thing to be done, but is invited to view the task as a member of a whole family; he is invited to make the suitable generalisations of the given problem statement. By successively adding more detail he eventually pins his algorithm down to a solution for the given problem."

The requirement to allow delayed commitment to design decisions can be illustrated in terms of type theory by considering the following existentially quantified goal

$$(\Sigma x : A)B(x)$$

To satisfy this goal we require a particular object v in A, such that $B(v)$ is a non-empty type. Refinement corresponds to the inverse of the Σ-introduction rule:

$$\frac{a : A \qquad b : B(a)}{\langle a, b\rangle : (\Sigma x : A)B(x)}$$

To be true to the step-wise approach to problem solving, a mechanism for delaying instantiation of a, the existential witness, is required. A *schema* variable makes this possible by acting as a place-holder for the eventual value.

The ability to explore different proof strategies is an essential requirement of an interactive proof system. Incorporating this flexibility will affect both the

choice of proof representation and the strategy employed for instantiating schema variables. A persistent proof representation and a lazy instantiation strategy will reduce the pruning involved in backtracking. Furthermore, it will increase the user's freedom to explore different refinements.

Inference rules provide the basis for formal proof. A secure and efficient representation is, therefore, important.

In the context of an interactive system, the ability to store work for future use is essential and should provide for definitions, theorems, derivations and proof strategies.

3.1.3 Programming tools

The application of rules and tactics requires extra information which is generally not automatically deducible. Consider the refinement of the goal $C(x)$ by the Σ-elimination rule:

$$\frac{x : (\Sigma\, a : A)B(a) \quad ||u : A; v : B(u) \triangleright\ d(u,v) : C((u,v))||}{split(x,[u,v]d(u,v)) : C(x)}$$

In general, it will not be possible to determine the abstraction from which $C(x)$ is constructed. Additionally, the type of the object expression x may not be decidable. This information must be supplied to ensure the constructibility of the two subgoals. In terms of forwards proof there are similar problems. Given the theorems corresponding to the premises, then in order to apply Σ-introduction it is necessary that the abstraction $[z]C(z)$, and assumptions corresponding to u and v in the rule schema, are supplied to ensure the constructibility of the conclusion. An objective in developing a "tool set" would be to minimise the

information the user must supply in order to apply rules and tactics.

Inference rules provide the basis of forwards proof. The ability to derive new rules from existing ones is an important tool, as it enables the number of steps in a proof to be reduced. Support for derived rules is a desirable objective of a programming framework. It was noted in the preceding section that efficiency is important. For this reason the justification for a derived rule and its representation should be kept separate.

As mentioned earlier, tactics are the basic building blocks for goal-directed proof. Tactic combinators, known as tacticals, were first introduced by the Edinburgh LCF project to enable more sophisticated tactics (proof strategies) to be built. In an interactive proof environment it is desirable that support is provided for the development of proof strategies.

Uniform methods should be developed for general programming problems. A particular class of types which are suited to the application of general methods are those which carry no computational content. The correctness of the overall proof depends upon being able to demonstrate that such types are non-empty. The manner in which the proof is obtained is not important, since it does not have any bearing on the computational aspect of the proof.

3.1.4 Programming interface

There are two aspects to a programming interface: the presentation of information and the specification of interactions. Proofs are complex structures and the integration of programs and proofs adds to this complexity. Providing a

mechanism for supporting abstraction is, therefore, an important objective in designing a programming interface. Abbreviations provide abstraction at the level of formulae. Abstraction at the level of derivations is also important. More sophisticated techniques are necessary to achieve abstraction over derivations. The interface should ease the task of the user by minimising the amount of detail required in specifying interactions. The interface should be as flexible as possible, allowing the user to specify as much or as little detail as they wish at each step in the proof process.

3.2 Current implementations

In this section we review four implementations of constructive type theory:

- Göteborg Type Theory System
- Type Theory Proof Assistant
- NuPRL
- Isabelle

Although these systems are at different levels of development, each reflects a distinct approach to at least one of the four categories of objectives outlined in section 3.1.

3.2.1 Göteborg Type Theory System

The Göteborg Type Theory System (GTTS) developed by Petersson[Petersson 82] is an experimental system for proof development in Martin-Löf Type Theory.

Programming logic

GTTS is basically an implementation of [Martin-Löf 82]. The logic differs slightly in three minor ways: Firstly, Petersson introduces the pair ($\times$) and function ($\rightarrow$) types explicitly, whereas Martin-Löf defines these in terms of the Σ and Π types respectively. Secondly, the type of lists is included, which is absent from [Martin-Löf 82]. Thirdly, for reasons of presentation Martin-Löf leaves certain well-formedness premises implicit in his formalisation which Petersson makes explicit. The convention Martin-Löf uses, which is documented in [Martin-Löf 82], is as follows:

> "...in those rules whose conclusion has one of the forms $a \in A$[1] and $a = b \in A$, only those premises will be explicitly shown which have these very same forms."

For instance, consider Martin-Löf's presentation of the rule for Π-introduction:

$$\frac{b \in B\ (x \in A)}{(\lambda x)b \in (\Pi x \in A)B}$$

To be complete, the well-formedness of the domain type A must also appear as a premise:

$$\frac{A\ type \quad b \in B\ \ (x \in A)}{(\lambda x)b \in (\Pi x \in A)B}$$

Note that using Backhouse's notation the judgement "$b \in B\ \ (x \in A)$" is written as $||[x : A \triangleright b : B]||$. In mechanising the logic, it is essential that GTTS makes these extra premises explicit. As pointed out by Harper[Harper 85], however, Petersson's formulation of the type theory elimination rules is incomplete. Consider, for instance the arbitrary family of types defined by $C(x)$, where x is of

[1] While Martin-Löf uses "$\in$" to denote membership, we adopt Petersson's use of ":".

type $\Theta(\bar{\lambda})$. A proof of $C(x)$ may follow from an application of Θ-elimination. According to Peterssen's formulation of Θ-elimination, a proof of $C(x)$ is dependent on being able to establish that for each canonical form defined by the Θ, $C(x)$ is a well-formed type. Assuming $\Theta(\bar{\lambda})$ to be N, then two cases arise: Firstly, when x is 0 an object in $C(0)$ is required. Secondly, when x is constructed by an application of the successor function an object in $C(u')$ is required, where u is in N. Peterssen's N-elimination rule takes the form

$$\frac{x : N \quad c : C(0) \quad d(u,v) : C(u') \quad [u : N;\ v : C(u)]}{rec(x, c, (u,v)[d(u,v)]) : C(x)}$$

This formulation relies on the closure property for type N, which is expressed by the theorem

$$(\Pi x : N)(Eq(N, x, 0) + (\Sigma n : N)Eq(N, n', x))$$

where $(u,v)[d(u,v)]$ corresponds to the abstraction $[u,v]d(u,v)$. Harper argues that the closure property on its own does not ensure the soundness of Peterssen's elimination rules. Harper's justification rests on the distinction which exists between equality and identity within the theory. The closure states that all objects of type N are equal, but not identical, to either 0 or an application of the successor function. Consequently, an object x, of type N, may exist such that $C(x)$ is not well-formed. If such an object is constructed, then a theorem could be derived which is not well-formed. Harper, however, admits that he does not know how to prove or disprove the existence of such an object. For reasons of "safety" he prefers to include an extra premise corresponding to

$$||x : X \triangleright C(x)\ type||$$

in each elimination rule of his logics. Incorporating this into Peterson's N-elimination rule would give

$$\frac{x : N \quad C(z)\ type\ [z : N] \quad c : C(0) \quad d(u, v) : C(u')\ [u : N;\ v : C(u)]}{rec(x, c, (u, v)[d(u, v)]) : C(x)}$$

It is worth noting that in both [Martin-Löf 82] and [Martin-Löf 84] these extra well-formedness premises are not present.

Programming framework

GTTS is a member of the LCF family of proof systems. The most significant consequence is that it inherits the metalanguage ML, which is a functional programming language which supports a polymorphic type scheme. The polymorphic type structure provides a secure framework for proof construction. To illustrate this, consider the following definition of the pair selector *fst*

```
let fst p = let p1,p2 = p in p1;;
fst = - : * # ** -> *
```

ML evaluates a definition and determines the most general type scheme, which ensures that *fst* is only applied to pair data objects. Since a function is not printable, its value is denoted by a hyphen. The asterisk notation denotes an arbitrary type. All object and type expressions are represented by the type *term*. For instance, the application *succ*(0) is constructed using the ML function

```
mkapp : term # term -> term
```

Similarly, an operator is provided for constructing abstractions. The type *form* represents the four judgements forms, excluding assumptions which are also of

type *form*. For instance, assumption free judgements of the form $a : A$ may be built using the ML function

```
mkelem : term * term -> form
```

For each constructor an associated selector is provided, together with projections and predicates. Except for the lack of arities, the abstract operations outlined above implement the theory of expressions described in chapter 2. While the constructors and selectors ensure well-formedness, the soundness of the logical system is maintained by the abstract type *thm* (theorem). The type *thm* is associated with the axioms

$$N\ type \quad 0 : N \quad U_1\ type \quad \{\}\ type$$

and is preserved by the rules of the theory.

Working directly with these abstract operations leads to rather complex expressions. Consider, for instance, the sentence $succ(n) : N$. The corresponding construction takes the form

```
mkelem(mkapp((mkconst 'succ'),(mkvar 'n')),(mkconst 'N'))
```

This abstract syntax is cumbersome to use. To alleviate this problem a more natural proper syntax is provided. Using the proper syntax, the above goal sentence may be input as `"succ(n):N"`. Quotation marks are used to delimit proper syntax.

For ease of construction and manipulation of formal expressions, a prefix notation is adopted uniformly throughout GTTS. The subset of Petersson's con-

Thesis	*GTTS*
$A \to B$	`->(A,B)`
$\lambda x.b(x)$	`lambda((x)[b(x)])`
$A \times B$	`#(A,B)`
$\langle a,b \rangle$	`pair(a,b)`
$split(c,[x,y]d(x,y))$	`split(c,(x,y)[d(x,y)])`
$\emptyset$	`{}`
$U_1, U_2, \ldots$	`U1,U2,...`

Table 3.1: GTTS notation

structors, which is used in the remainder of this review, is presented together with the corresponding thesis notation in table 1.1.

Inference rules are represented as functions. For each rule of inference an associated ML function exists; a mapping from theorems to theorem. In order to demonstrate this, consider the pair introduction rule:

$$\frac{a : A \quad b : B}{\langle a,b \rangle : A \times B}$$

This rule is implemented by the ML function

```
PAIRintr : thm -> thm -> thm
```

Program construction in GTTS is by forwards proof and is achieved by function application. For example, consider the derivation:

$$\dfrac{\dfrac{}{N\ type}\ N\text{-form}}{\dfrac{\dfrac{\dfrac{}{N\ type}\ N\text{-form} \quad \dfrac{\dfrac{}{N\ type}\ N\text{-form}}{x : N\ [x : N]}\ VAR\text{-intr} \quad \dfrac{\dfrac{}{N\ type}\ N\text{-form}}{y : N\ [y : N]}\ VAR\text{-intr}}{\langle x,y \rangle : N \times N\ [x : N;\ y : N]}\ \times\text{-intr}}{\lambda y.\langle x,y \rangle : N \to N \times N\ [x : N]}\ \to\text{-intr}}{\lambda x.\lambda y.\langle x,y \rangle : N \to N \to N \times N}\ \to\text{-intr}$$

In GTTS this derivation is constructed by the following function application

```
FUNintr "x" Nform
#       (FUNintr "y" Nform
#                (PAIRintr
#                     (VARintr Nform "x")
#                     (VARintr Nform "y")))
;;
"lambda((x)[lambda((y)[pair(x,y)])]) : ->(N,->(N,#(N,N)))"
: thm
```

Exploiting the functional nature of ML, a derivation can be easily generalised to give a derived rule of inference. By replacing the occurrences of the axiom Nform, the above derivation may be generalised for arbitrary data objects. The corresponding ML function is

```
\th1.\th2.
#   (FUNintr "x" th1
#            (FUNintr "y" th2
#                     (PAIRintr
#                         (VARintr th1 "x")
#                         (VARintr th2 "y"))))
;;
- : thm -> thm -> thm
```

Programming tools

The richness of Martin-Löf's theory means that the structure of the conclusion to a rule, in general, is not directly deducible from its premises. For example, consider the application of the pair elimination rule:

$$\frac{a : A \times B \qquad d(u,v) : C(\langle u,v\rangle)\ [u : A;\ v : B]}{split(a,(u,v)[d(u,v)]) : C(a)}$$

to theorems of the form

$$a : A \times B$$

$$d : C(\langle u,v\rangle)$$

The problem is in determining which occurrences of $\langle u, v \rangle$ within C should be replaced by a in the conclusion of the rule. Petersson resolves the problem of the constructibility of a conclusion by forcing the user to specify additional information. In the case of pair elimination, the user must provide an abstraction as well as the variables to be discharged. This is reflected in the implementation of the rule:

```
PAIRelim : term -> thm -> (term # term # thm) -> thm
```

where the first argument takes the form $(s)[C(s)]$. The addition of these extra arguments increases the burden on the user in the construction of a derivation. The onus is on the user to introduce suitably named variables and to resolve ambiguities.

GTTS gives mechanised assistance in the construction of programs through three decision procedures: a simplifier, a type checker and a decision method for equality. The simplifier is implemented by the ML function

```
evalthm : thm -> thm
```

Given a theorem of the form $t : T$, evalthm evaluates t, generating a theorem of the form $t = t' : T$. For instance, applying evalthm to the theorem

```
"split(pair(a,b),(p,q)p) : A [A:U1; a:A; b:A]" : thm
```

generates the theorem

```
"split(pair(a,b),(p,q)p) = a : A [A:U1; a:A; b:A]" : thm
```

The type checker is implemented by the ML function

```
typecheck : term -> thm
```

and is based on Milner's type check algorithm[Milner 78]. For example, typecheck can automatically determine the type of the identity function:

```
typecheck "lambda((x)x)";;
"lambda((x)x) : ->(A,A) [A : U1]" : thm
```

Type checking is not, in general, decidable in type theory. Consequently, Peterson's implementation excludes expressions which require the equality or universe types to be checked. The final decision procedure for equality, implemented by the ML function

```
eqprove : thm list -> form -> thm
```

embodies a simple equality reasoning algorithm. Given a list of known equalities and a goal equality, eqprove attempts to construct a proof using the symmetry and transitivity of equality. Assuming an appropriate definition for plus, and given the theorems

```
"plus(0,plus(y,z)) = plus(y,z):N [y : N; z : N]" : thm
"plus(plus(0,y),z) = plus(y,z):N [y : N; z : N]" : thm
```

denoted by th1 and th2 respectively, then a proof of

```
"plus(0,plus(y,z)) = plus(plus(0,y),z):N"
```

is a achieved by the following application of eqprove

```
eqprove [th1;th2] "plus(0,plus(y,z)) = plus(plus(0,y),z):N";;
"plus(0,plus(y,z)) = plus(plus(0,y),z):N [y : N; z : N]" : thm
```

A limitation of eqprove is that it does not make use of the assumption lists to deduce type information when trying to construct a proof.

Programming interface

ML provides the interface to GTTS. As a consequence, interaction takes the form of a *read-eval-print* loop. A definition mechanism is provided which enables new object level constants to be defined. All folding/unfolding of definitions is done automatically by the system.

3.2.2 Type Theory Proof Assistant

The Type Theory Proof Assistant (TTPA) developed by Hamilton[Hamilton 85] is a proof editing facility based upon GTTS. Unlike GTTS, however, TTPA supports goal-directed proof and an explicit proof representation.

Programming logic

As an extension to GTTS, Hamilton's proof assistant inherits Peterson's formalisation of Martin-Löf Type Theory.

Programming framework

TTPA provides a facility for managing the creation and modification of a goal tree, and its subsequent conversion into a proof tree. A goal is represented by the ML type

```
goal = form # (form list)
```

where the first component denotes the judgement type and the second component denotes the type of the assumption list. The initial goal takes the form

$$t : T \quad W$$

where t denotes a schema variable, T is the specification type and W a possibly empty list of assumptions. Proof is goal-directed; an initial goal is repeatedly refined. This process generates a goal tree and gradually builds up an instantiation for the schema variable t. Both goal and proof trees are represented by the type

```
prooftree = node + (node # (prooftree list))
```

The distinction between a goal and a theorem is made at the level of the node:

```
node = validation # (thm + goal) # (subst list)
```

If a node belongs to a proof tree, then its second component will be a left injection. The value of the injection is a theorem; an object of type thm. In the case of a goal node a right injection is present; the value of the injection being of type goal. A goal tree is complete once all subgoals have been refined to the level of axioms or known theorems. Converting a complete goal tree into a proof tree is achieved by propagating theorem-hood back up through the goal tree. The first component of the node, the validation

```
validation = (thm list) -> thm
```

provides the basis for this conversion. A validation is a function which constructs a theorem corresponding to the current node from theorems corresponding to its immediate subgoals. As noted earlier, during the refinement of a goal tree schema variables become instantiated. Even so, a choice exists between whether instantiations are made immediately, or delayed until the goal tree is complete. During the course of a derivation errors may occur in the choice of refinement taken.

If a strategy of eager instantiation is adopted, then the amount of backtracking required will be more than if instantiation is delayed. TTPA delays instantiation for this reason, allowing the user to explore different refinements with greater ease. The third component of TTPA's node representation, the substitution list, provides the mechanism by which delayed instantiation is achieved. Substitution lists record instantiations for schema variables. A substitution is represented by the type

```
subst = term # term
```

where the first component is an object expression and the second component a schema variable.

Goal tree refinement, mentioned above, is achieved by LCF style tactics. For each GTTS rule of inference, an associated TTPA tactic (rule inverse) exists which is implemented by a ML function of the type

```
tactic = goal -> (validation # (goal list) # (subst list))
```

Tactics are applied to leaf nodes. A successful refinement results in the goal tree being extended. An extension generates a leaf for each new subgoal in which a copy of the substitution list is placed. The validation generated by the refinement is stored in the current node. At any point during the editing session the proof state is represented by a global variable of type state; a proof tree and an integer list which determines the position of the current proof node:

```
state = prooftree # (int list)
```

Proof states can be stacked. This enables a proof to be suspended while a subsidiary proof is undertaken. In addition, theorems can be stored for future use. After each refinement the editor checks to see whether any of the new subgoals match the set of known theorems, if so, then the corresponding subgoal is completed automatically.

Programming tools

It was noted in the preceding section that the basic building block of a proof in TTPA is the tactic. Given a goal it will not, in general, be possible to automatically deduce the subgoals from the application of a rule inverse. To ensure the constructibility of subgoals, it is necessary for certain tactics to be supplied with extra information. For instance, consider again the rule for pair elimination:

$$\frac{a : A \times B \quad d(u,v) : C(\langle u,v \rangle) \quad [u : A;\ v : B]}{split(a, (u,v)[d(u,v)]) : C(a)}$$

The corresponding tactic is implemented by the ML function

```
PAIRelimtac : term -> term -> term -> term -> tactic
```

The first argument is an abstraction $(x)[C(x)]$ which defines any dependency there exists between the goal type and the principal argument of the eliminator. The next three arguments indicate the principal argument and its associated base types respectively. This formulation covers the most general case. Nevertheless, in certain situations it will be possible to deduce certain arguments automatically. For instance, consider a goal of the form

$$v1 : A \quad [x : A \times B]$$

where v1 denotes a schema variable. Assuming we wish to eliminate on x, then the following application of PAIRelimtac is necessary

```
(PAIRelimtac "(x)A" "x" "A" "B")
```

It would be sufficient, however, to indicate that an elimination on x is required. Given this information, it is deducible from the context that the pair elimination tactic is required, and that the third and fourth arguments of PAIRelimtac take the form "A" and "B" respectively. Finally, since the goal type is a constant, no dependency on the principal argument exists.

As mentioned earlier, tacticals allow tactics to be combined in different ways in order to produce more sophisticated tactics. TTPA supports LCF style tactical reasoning. This approach to building proof strategies gives rise to the problem of referencing assumptions introduced during the application of the derived tactic. To illustrate this problem, consider the following goal

$$A \times B \rightarrow B \times A$$

Refinement by function introduction gives rise to two subgoals:

$$v1 : B \times A \; [b1 : A \times B]$$

$$A \times B \; type$$

To ensure the constructibility of the first subgoal, FUNintrtac requires a bound variable identifier to be supplied, in this case b1. The problem is to ensure the uniqueness of the identifier. One approach would be to allow complete freedom in choosing a variable name. The onus would then be on the system to check for consistency. Although potentially a sound approach, it would lead to a tried and

tested strategy failing in certain situations because of name conflicts. TTPA's approach to the problem is to use relative referencing. Each bound variable introduced within a strategy is denoted by an integer preceded by an underscore, the first being "_1". An application of the function introduction tactic takes the form

```
(FUNintrtac ["_1"])
```

TTPA incorporates a mechanism for generating unique variables which enables relative references to be translated into unique identifiers at the time the strategy is applied.

Programming interface

Interaction with TTPA takes place through a command oriented editing facility. Three categories of commands are provided:

- information
- navigation
- action

Information commands provide the user with information about a derivation, while navigation commands enable the user to move around a derivation. Action commands provide the mechanism by which a derivation is constructed and modified. For example, the initialisation of a goal tree is achieved by the ML function

```
proofedit : term -> asslist -> .
```

as illustrated below:

```
# proofedit "->(#(A,B),#(B,A))" ["A:U1";"B:U1"] ();;
"v10 : ->(#(A,B),#(B,A)) [A : U1; B : U1]"
(goal)
Prooftree initialised.
```

A node can either be unproven or proven. This is indicated by the status flags goal and theorem respectively. A goal node is refined by an application of the ML function

```
ebt : tactic -> .
```

which stands for *extend-by-tactic*. For example, the application of FUNintrtac to the goal tree initialised above gives rise to the following display:

```
# ebt (FUNintrtac []) ();;
1
"#(A,B) type [A : U1; B : U1]"
(goal)() : .
```

An application of ebt extends the goal tree automatically, making the left most subgoal the current node, which is indicated here by the 1 preceding the goal.

3.2.3 NuPRL

NuPRL is an interactive proof development system based on a constructive theory of types. Proof development is conducted in a goal-directed style. Once a proof is complete a corresponding program is obtained by an automatic process known as *extraction*. NuPRL grew out of the PRL system developed by Bates[Bates 79]. PRL stands for *program refinement logic*. The PRL system supports goal-directed proof in a first-order constructive logic of integers and lists.

This logic was found, however, not to be expressive enough for the needs of a programming logic. The shortcomings of the logic are discussed in [Constable 85] [Constable & Zlatin 84] where a richer logic is proposed which is influenced by the work of [de Bruijn 80] [Martin-Löf 75] [Martin-Löf 82] [Scott 70]. This work lead to the definition of the NuPRL logic[Bates & Constable 83].

Programming logic

Although NuPRL's constructive logic has similarities to Martin-Löf's theory, it also exhibits significant differences which are presented in this section.

A fundamental difference between the two logics is that Martin-Löf's theory is underpinned by a general theory of expressions. The NuPRL logic is based on no such theory.

An assertion in NuPRL is represented as a sequent, which is a hypothesis list followed by a goal type:

$$H \gg T$$

The hypothesis list, H, provides the context in which T, the goal type, is to be proved. A hypothesis list is built up from declarations of the form

$$x : X$$

where x is a variable bound to the type X. The NuPRL notion of assertion condenses Martin-Löf's four judgement forms into a single judgement form. Harper gives two main reasons for this minimalism: "notational economy" and "to simplify the formalisation of the theory". This economy is achieved through the

equality type which takes the form

$$a = b \text{ in } A$$

Apart from the superficial difference in syntax, the NuPRL equality type differs from Martin-Löf's formulation in that reflexivity is embedded within the type structure. In Martin-Löf's theory, reflexivity is defined separately through a general rule:

$$\frac{a : A}{a = a : A}$$

Reflexivity in NuPRL is automatic, the judgement

$$a \text{ in } A$$

is an abbreviation for

$$a = a \text{ in } A$$

To explain the unification of Martin-Löf's four judgement forms we consider each in turn:

- The type judgement, A *type*, is replaced by membership over an arbitrary universe, which is represented by the type, A in U_i, which is an abbreviation for $A = A$ in U_i.
- The type equality judgement, $A = B$, is replaced by an equality over an arbitrary universe, which is represented by the equality type $A = B$ in U_i.
- The membership judgement, $a : A$, is replaced by the equality type a in A, which is an abbreviation for $a = a$ in A.

- The equality membership judgement, $a = b : A$, is replaced by the equality type $a = b$ in A.

This unification of the judgement forms is too restrictive, however, in the context of goal-directed proof. For instance, consider a program specification denoted by the type P. To construct a program satisfying P, our initial goal must take the form of the membership judgement

$$\gg p \text{ in } P$$

This means that our initial goal also includes a program satisfying P. In effect, program construction is reduced to program verification. To overcome this restriction NuPRL includes the notion of an *implicit* judgement. This means a judgement in which the proof object is suppressed. The *implicit* proof object is known as the *extract term* and is represented notationally as

$$\gg T \text{ ext } t$$

where t, the *extract term*, is a justification for T. So to construct a program satisfying P, the initial goal takes the form

$$\gg P$$

From a derivation of P, an *extract term* p is obtained by a recursive process known as *extraction*, where p is an implementation of P. The *extract term* can only be obtained once the computational component of a goal type has been achieved. Emphasis is on demonstrating that a type is non-empty. Therefore, the logic is oriented towards demonstrating the truth of propositions; the actual demonstration seems to be of secondary importance.

As a programming assistant, NuPRL's type structure reflects the needs of the programmer. Here we present the aspects of the type structure which differ from Martin-Löf's theory presented in chapter 2. Character strings may be modelled directly in the theory through the type of atoms. This provision seems to be the reason for the exclusion of the more general finite types. Consequently, the empty type is given greater prominence through the constant type void. The type of integers, int, is included in NuPRL together with the associated operators $+$, $-$, $*$, $/$ and mod. The $<$ relation on integers is built into the theory as a separate type called less. Information hiding is supported through the subset type

$$\{x : A | B(x)\}$$

Given an object a in A, then a is also a member of this subset type, if an object in $B(a)$ can be constructed. The subset type is related to the dependent product type

$$x : A \# B$$

Note that the dependent product type corresponds to Martin-Löf's disjoint union of a family of types. The difference between the two types is that the justification for $B(a)$, denoted by b, is part of the constructive justification of the dependent product type, but is dropped from the justification of the subset type. The quotient type constructor is the most innovative component of the NuPRL logic. The quotient type introduces the notion of equivalence relations. By allowing a stronger equality relation over an existing type to be defined, the quotient operator enables new types to be constructed. For instance, given the base type

A, and an equivalence relation on A defined by the type E, then

$$A//E$$

is a quotient type. The objects in $A//E$ and A are the same. However, two objects x and y in A are equal in $A//E$, if the type $E(x,y)$ is shown to be non-empty. NuPRL incorporates an inductive type constructor *rec*, which permits the definition of new inductively defined types. A partial function space operator $\rightsquigarrow$ is proposed, permitting the definition of recursive partial functions. Both *rec* and $\rightsquigarrow$ are described in detail in [Constable & Mendler 85], while in [Constable & Smith 87] a partial object type theory is presented.

NuPRL's top-down style of proof construction is reflected in the structure of its rules. Known as refinement rules, they are presented in a linear style with prominence given to the conclusion. Indentation is used to distinguish the premises from the conclusion. A significant feature of the organisation of the NuPRL rules is that the introduction rules operate on the goal type, while the elimination rules operate on the hypothesis list. For example, consider the product introduction rule:

$$\begin{array}{lll} H & \gg & A\#B \text{ ext } \langle a,b\rangle \text{ by intro} \\ & \gg & A \text{ ext } a \\ & \gg & B \text{ ext } b \end{array}$$

The *ext* clause indicates the form the *extract term* will take, but it is not actually displayed by the system, except through the extraction mechanism. The product introduction rule is applicable where the product constructor is the principal constructor of the goal type. A valid application of the product introduction rule

generates two subgoals corresponding to the components of the product. Note that the *extract term* for the conclusion takes the form of a pair, the components of which are the *extract terms* corresponding to the subgoals. For each introduction rule there exists a version in which the extract term is made explicit through the equality type. The explicit version of the product introduction rule takes the form

$$\begin{array}{rl} H & \gg \ \langle a, b\rangle \text{ in } A\#B \text{ by intro} \\ & \gg \ a \text{ in } A \\ & \gg \ b \text{ in } B \end{array}$$

Similarly, there are implicit and explicit elimination rules. The implicit product elimination rule takes the form

$$\begin{array}{c} H, z : A\#B, H' \gg T \text{ ext } \mathit{spread}(z; u, v.t) \text{ by elim } z \text{ new } u, v \\ u : A, v : B, z = \langle u, v\rangle \text{ in } A\#B \ \gg \ T[\langle u, v\rangle/z] \text{ ext } t \end{array}$$

where $T[\langle u, v\rangle/z]$ denotes the substitution of $\langle u, v\rangle$ for z in T. This version of the product elimination rule is applicable when the hypothesis list contains a declaration of the form

$$z : A\#B$$

There may exist more than one declaration of the correct form, so the rule is parameterised by an integer which indicates the position of the hypothesis. The additional parameters, u and v, indicate names for new declarations which are introduced in the subgoal. In this way implicit elimination rules support forwards inference. The corresponding explicit version takes the form

$H \gg spread(e; x, y.t)$ in $T[e/z]$ by intro [over $z.T$] using $A\#B$[new u, v]
$\gg e$ in $A\#B$
$u : A, v : B, e =< u, v >$ in $A\#B \gg t[u, v/x, y]$ in $T[< u, v > /z]$

Note that explicit elimination rules break the pattern of eliminators applying only to hypotheses. As mentioned earlier, working with goals of the form of the equality type corresponds to program verification. In order to apply the above elimination rule, a fully instantiated *spread* is required. The principal argument of the eliminator, e, has to belong to the product $A\#B$. In contrast to the implicit elimination rule, where the principal argument of the noncanonical constructor must be a variable, no such restriction is imposed.

Both the implicit and explicit versions of the product elimination rule presented above are examples of what Dyckhoff[Dyckhoff 87] refers to as "strong elimination rules". NuPRL's formulation is stronger than Martin-Löf's Σ-elimination rule because certain premises are weakened by the inclusion of an additional hypothesis of the form

$$e = \langle u, v \rangle \text{ in } A\#B$$

which is discharged in the conclusion. Dyckhoff states that these stronger elimination rules overcome the problem of spurious λ-abstractions and applications noted by Backhouse[Backhouse 87] in his investigations into the application Martin-Löf's theory to program construction.

In regard to the issue of well-formedness, NuPRL minimises the number of additional proof obligations. This minimisation is achieved by unifying, as far as possible, the well-formedness obligations with the constructive obligations. A

notable feature of the NuPRL logic, with respect to well-formedness, is the way in which hypotheses are introduced. NuPRL has an axiom of the form

$$H, x : A, H' \gg A' \text{ ext } x \text{ by hyp } x$$

where A' is α -convertible to A. A direct consequence of adopting this approach is that the initial goal type must be self-contained. This means that the initial hypothesis list must be empty:

$$\gg T$$

This constraint is achieved by means of the dependent function space. The dependent function space corresponds to Martin-Löf's cartesian product of a family of types. For example, the initial goal

$$A : U_1,\ B : U_1,\ a : A,\ b : B \gg A \# B$$

is represented by the sequent

$$\gg A : U_1 \rightarrow (B : U_1 \rightarrow (a : A \rightarrow (b : B \rightarrow A \# B)))$$

This sequent is refined by the repeated application of the dependent function introduction rule. Note that the well-formedness premise of the dependent function rule takes care of the well-formedness of the hypotheses.

Programming framework

NuPRL provides a facility for creating and managing proof trees. A NuPRL proof tree is a recursive data type where each proof node has four components:

- List of declarations (hypotheses)
- Term (goal type)
- Refinement rule (initially unspecified)
- List of proofs/subgoals (initially empty)

Formal proof is supported by the metalanguage Cambridge ML, a dialect of the original Edinburgh ML. A proof is extended by the application of a tactic to a proof. Given a rule of inference, the ML function

```
refine : rule -> tactic
```

generates a tactic

```
tactic =  proof -> proof list # validation
```

Applying `refine` to a rule and a proof generates a proof list and an validation:

```
validation = proof list -> proof
```

The proof list is determined by the given refinement rule and the current proof node. A proof of the current node is obtained by the application of the validation to the achievements of the proof list. Note that the NuPRL type proof encompasses both proofs and partial proofs.

As well as providing a proof editing facility, NuPRL also supports a library module. A library is a linear structure for the storage and retrieval of definitions, theorems and proof strategies.

Programming tools

As indicated in the preceding section, tactics are the basic building blocks of NuPRL proofs. NuPRL provides two kinds of tactics: refinement and transformation. Refinement tactics closely resemble LCF style tactics in that they operate at the level of the unrefined goal. Any intermediate proof steps resulting from the application of a refinement tactic remain hidden. In contrast, transformation tactics operate on proofs. As a result, a transformation tactic is not restricted to unrefined nodes, which in turn enables the transformation of a complete subproof. A significant feature of transformation tactics is that proof refinement may be explicit. This means that the intermediate proof structure is maintained.

NuPRL also provides assistance with respect to the question of the constructibility of subgoals. Consider the goal sequent

$$A : U_1, B : U_1, A\#B \gg B$$

Proof requires the implicit product elimination rule presented earlier, which is implemented by the ML function

```
product_elim : int -> tok -> tok -> rule
```

The first argument denotes the hypothesis to eliminate on, while the remaining arguments specify new variables names which appear in the subgoal. The tactic corresponding to the required instance of the product elimination rule is constructed by the following application of `refine`

```
refine(product_elim 3 'nil' 'nil');;
```

Much of the information made explicit in this application is unnecessary. The only detail which is essential is the fact that an elimination is to be performed on the third hypothesis. The structure of the hypothesis determines which elimination rule is appropriate. NuPRL exploits this by allowing the above refinement to be specified by the string "elim 3". This feature is overshadowed, however, by a serious constructibility problem, noted by Harper[Harper 85], which concerns the introduction rule associated with the dependent product type:

$$\begin{array}{rll} H & \gg & x : A\#B \text{ ext } \langle a, b\rangle \text{ by intro at } U_i \; a \; [\text{new } y] \\ & \gg & a \text{ in } A \\ & \gg & B[a/x] \text{ ext } b \\ y : A & \gg & B[y/x] \text{ in } U_i \end{array}$$

The dependence between the subgoals means that the first subgoal must be solved, that is, an a must be constructed before the second subgoal can be fully instantiated. Harper discusses three solutions to the problem:

1. Parameterise the rule by a.

2. Parameterise the rule by a derivation of "$\gg a$ in A".

3. Allow derivations to "pass through inconsistent states".

The simplest solution of parameterising the rule by a forces the user to opt for a particular solution prematurely. Since the second premise of the product elimination rule constrains the form of the existential witness, it would seem desirable that a choice is delayed until the second subgoal is further refined. As there is no guarantee that the proof object supplied by the user is actually in A,

program construction is reduced to program verification. Harper also points out that this approach leads to a violation of his refinement principle:

> "...each refinement step ought not have its correctness depend on further independent development of the derivation tree."

The second solution suffers from the same problems as the first, except that the chosen value for the existential witness is guaranteed to belong to A. The final solution allows the choice of a value for the existential witness to be delayed until both subgoals are further refined. NuPRL is forced to adopt the first approach because it makes no provision for schema variables.

Like TTPA, NuPRL supports LCF style tactical reasoning. This approach to building proof strategies raises the problem of referencing hypotheses introduced during the application of a derived tactic. For example, consider the goal sequent

$$H \gg A\#B \rightarrow B\#A$$

refinement by function introduction generates two subgoals:

$$H, A\#B \gg B\#A$$

$$H \gg A\#B \text{ in } U_1$$

Refinement of the first subgoal requires reference to the hypothesis $A\#B$. As noted previously, hypotheses are referenced by their position within the hypothesis list. This leads to a rather clumsy approach to referencing, which involves computing the length of the hypothesis list. For example, by product introduction the first subgoal is reduced to

$$H, A\#B \gg B$$

$$H, A\#B \gg A$$

These subgoals are satisfied by decomposing the hypothesis $A\#B$. Refinement of the first subgoal by product elimination gives rise to a subgoal of the form

$$H, A\#B, A, B \gg B$$

The product elimination rule is implemented by the ML function

```
product_elim : int -> tok -> tok -> rule
```

The position of the required hypothesis corresponds to the length of the hypotheses list. Remembering that a tactic is of type

```
proof -> proof list # validation
```

then the required tactic is given by the ML function

```
\p.(refine (product_elim (length (hypotheses p)) 'u' 'v') p)
```

where the selector hypotheses returns the current hypotheses list. This construction is rather cumbersome and detracts from the intuitive ideas which underpin tactical reasoning. It would seem desirable that the strategy builder should be able to reference hypotheses without having knowledge of the abstract representation of proofs. More seriously, this approach to referencing hypotheses also reduces the generality of tactics. For instance, a tactic which is successful on one goal may fail on another which is identical except for the length of the hypothesis list.

NuPRL provides three decision methods to support proof construction:

- arith
- compute
- equality

A restricted form of arithmetic reasoning is provided by the arith proof procedure. It was developed by Tat-hung Chan. An account of the proof of correctness of arith appears in [Constable *et al* 82]. The second procedure, compute, supports the reduction of subterms which occur within a subgoal, without the need for the subterm to be explicitly isolated. Given an instance of the equality type

$$H \gg t = t' \text{ in } T$$

equality attempts to construct a justification making use of the hypothesis list H. The procedure relies on hypotheses which take the form of equalities over T or T', where $T = T'$ is deducible using reflexivity, commutativity and transitivity.

Finally, a special tactic, auto-tac, provides a level of automatic proof. auto-tac is a transformation tactic which is invoked after each refinement. The user is permitted to define the tactics which auto-tac will apply. By default, auto-tac is only invoked if it can complete a subgoal. However, this can be modified to allow for partial proofs.

Programming interface

From the point of view of the user NuPRL is divided into five subsystems:

- command module
- library module
- text editor
- refinement editor
- evaluation module

Each subsystem is assigned a window. Interaction with the various components of NuPRL is co-ordinated through the command module. The library module supports the storage and retrieval of the formal objects, as noted earlier. The text editor is general purpose and provides a basic screen editing interface. A definition mechanism is provided. The folding/unfolding of definitions during the course of a proof is automatic.

The refinement editor is the interface to the proof. The user is presented with a window onto the proof structure. The window provides the user with a view of the current proof node. For example, consider the following proof node:

```
.-----------------------------------.
|EDIT TERM swap                     |
|-----------------------------------|
|# top 1 1                          |
|1. A:U1                            |
|2. B:U1                            |
|>> (A#B)->B#A                      |
|                                   |
|BY <refinement rule>               |
'-----------------------------------'
```

Hypotheses are numbered for identification purposes. A proof node is associated with a status mark. In the above example the status mark # indicates that the proof is incomplete. Next to the node status mark there is an address. This gives the relative position of the current node with respect to the root node which is denoted by the label top. Once a refinement is specified, the proof editor window is updated showing the subgoals generated. For instance, refinement of the above goal by function introduction is achieved as follows:

```
|EDIT THM swap                          |
|---------------------------------------|
|# top 1 1                              |
|1. A:U1                                |
|2. B:U1                                |
|>> (A#B)->B#A                          |
|                                       |
|BY intro                               |
|                                       |
|1# 1. A:U1                             |
|   2. B:U1                             |
|   3. A#B                              |
|   >> B#A                              |
|                                       |
|2# 1. A:U1                             |
|   2. B:U1                             |
|   >> A#B in U1                        |
|                                       |
'---------------------------------------'
```

Navigation around the proof tree is achieved by either mouse or keyboard interactions.

Finally, the evaluation module provides an evaluator for executing extract terms.

3.2.4 Isabelle

Isabelle is a general purpose interactive theorem prover developed by Paulson [Paulson 86] which provides a framework in which the user can define the logic in which they work. Goal-directed proof is supported. Emphasis is placed on the role of the rule in proof construction. A theorem in Isabelle corresponds to a rule with no premises. Paulson has implemented a subset of Martin-Löf's theory in Isabelle.

Overview

In this section we present an overview of Isabelle. A general representation for logical syntax is provided by Isabelle's framework, which is based on a version of Martin-Löf's theory of expressions extended to allow for more than one atomic type. Internally, rules are represented as Horn-clauses. Proof construction corresponds to rule composition which is achieved by unification. At each step in its construction a proof is completely characterized by a derived rule. For example, the goal G is represented by the trivial rule

$$\frac{G}{G}$$

The refinement of this goal by the inference rule

$$\frac{P_1 \ \ldots \ P_n}{Q}$$

is achieved by unifying the premise G with the conclusion Q, giving rise to the derived rule

$$\frac{P_1' \ \ldots \ P_n'}{Q'}$$

At each step in a proof we are dealing with a valid rule of inference. This does not mean that we will obtain a theorem, but it ensures soundness in the same way as LCF validations. The difference between the two approaches is that in LCF the processes of goal refinement and proof construction are separated. A goal tree is constructed by the application of tactics. Each tactic is associated with a validation which embodies a rule of inference. It is the composition of the validations which produce the proof. In Isabelle goal refinement and validation are performed by the rule.

The process of rule composition may instantiate occurrences of schema variables in both the goal and the rule. For example, consider the derivation of an object in the type $A \to B \to A \times B$. The initial derivation takes the form

$$\frac{\mathtt{a} : A \to B \to A \times B}{\mathtt{a} : A \to B \to A \times B}$$

where a denotes a schema variable. The function type is defined in terms of the Π type:

$$\lambda \to B \equiv \Pi(\lambda, B)$$

Refinement is achieved by composing the premise of the initial derived rule with the conclusion of the Π-introduction rule:

$$\frac{\lambda\ type \quad \mathtt{b}(x) : B(x)\ [\Gamma, x : A]}{\mathtt{lambda}(\mathtt{b}) : \Pi(\lambda, B)} \qquad x \text{ not free in } \Gamma, \mathtt{b}, B.$$

Paulson uses the lambda operator to denote function objects, reserving λ to represent abstraction within the theory of expressions. The resulting derivation takes the form

$$\frac{A\ type \quad \mathtt{b}(x) : B \to A \times B\ [x : A]}{\mathtt{lambda}(\mathtt{b}) : A \to B \to A \times B}$$

Note that the schema variable $\bar{a}$ has been instantiated to a function object, where $\bar{b}$ is a representation for the abstraction $[x]b(x)$, or in Paulson's notation $\lambda(x)b(x)$. Further refinement by Π-introduction produces a derived rule of the form

$$\frac{A\ type \quad B\ type \quad \bar{c}(x,y) : A \times B\ [x : A,\ y : B]}{\mathbf{lambda}(x)\mathbf{lambda}(\bar{c}) : A \to B \to A \times B}$$

In this way the initial schema variable $\bar{a}$ is gradually instantiated to a program satisfying the initial specification.

Paulson accommodates higher-order logics, such as type theory, by providing higher-order schema variables. This enables certain short-hands to be introduced. For example, compare the Π-introduction rule given above with the version formulated by Petersson's[Petersson 82]:

$$\frac{A\ type \quad b(x) : B(x)\ [x : A]}{\mathbf{lambda}((x)[b(x)]) : \Pi(A, (x)[B(x)])}$$

Note that the abstractions $(x)[b(x)]$ and $(x)[B(x)]$ are replaced by higher-order schema variables $\bar{b}$ and $\bar{B}$ respectively. Rule composition now corresponds to higher-order unification. Note that in the preface to [Martin-Löf 84] such "higher-level variables" are introduced.

Programming logic

Paulson's implementation of type theory is similar to that of Petersson. Only a subset of the theory is implemented. The list and well-ordering types are not included, nor are the universes.

Programming framework

It was noted in the overview that a derivation is represented as a derived rule, where the premises denote the current subgoals and the conclusion is the top-level goal. Consequently, no internal proof structure is maintained. Isabelle has a goal-stack architecture. Each element of the goal-stack contains a snap-shot of a derivation. Rule composition is achieved by higher-order unification. As a consequence, an unbounded stream of unifiers may result. Isabelle deals with this by computing alternative unifiers lasily. Backtracking enables alternative unifiers to be tried. For each step in a proof, a new rule of inference is derived and pushed onto the goal-stack together with a possibly empty stream of unifiers. Error recovery is achieved by simply popping off the required number of goal-stack entries until the appropriate point in the derivation is reached. A set of operations upon rules, known as *inference principles*, form the basis of proof development in Isabelle. For instance, rule composition is achieved by two such operations:

- standardise
- compose

The first performs *standardising apart* on a rule, while the second attempts to unify a particular premise of a given rule with the conclusion of another. From the primitive inference principles more sophisticated operations may be derived. For example, a composite inference principle, *resolve*, follows from the primitives, *standardise* and *compose*. One of the most appealing aspects of Isabelle's proof

representation is that at any point during the construction of a proof a derived rule can be extracted for future use.

Programming tools

Isabelle's goal-stack architecture, described above, enforces a goal-directed style of proof. Proof construction is achieved by the application of tactics. Isabelle's notion of tactic is significantly different from those of LCF. An LCF tactic maps a goal onto a list of goals, while an Isabelle tactic is a function from rule to rules. Isabelle tactics build upon the inference principles discussed earlier. As a consequence of Isabelle's proof representation, tactics operate on a complete derivation. This is in contrast to LCF style tactics which are applied to a single subgoal. For instance, consider again the derivation of an object in the type $A \to B \to A \times B$. The complete derivation is as follows:

$$\dfrac{A\ type \quad \dfrac{\dfrac{A\ type}{x : A\ [x : A]}\,Assum\text{-}intr \quad \dfrac{B\ type}{y : B\ [y : B]}\,Assum\text{-}intr}{\langle x, y\rangle : A \times B\ [x : A,\ y : B]}\,\times\text{-}intr}{\mathbf{lambda}(y)\langle x, y\rangle : B \to A \times B\ [x : A]}\,{\to}\text{-}intr}{\mathbf{lambda}(x)\mathbf{lambda}(y)\langle x, y\rangle : A \to B \to A \times B}\,{\to}\text{-}intr$$

This derivation is represented in Isabelle by the following derived rule

$$\dfrac{A\ type \quad B\ type \quad A\ type \quad B\ type}{\mathbf{lambda}(x)\mathbf{lambda}(y)\langle x, y\rangle : A \to B \to A \times B}$$

Note the duplication in the well-formedness premises. Isabelle provides the tactic *merge-premises* to remove such duplication

$$\dfrac{A\ type \quad B\ type}{\mathbf{lambda}(x)\mathbf{lambda}(y)\langle x, y\rangle : A \to B \to A \times B}$$

Such global operations are useful when applying formation and introduction rules, since the structure of a subgoal completely determines which rule to apply. This kind of operation is less useful, however, when applying elimination rules. Elimination rules are related to the deeper structure of a goal. Consequently, more than one eliminator may be applicable to a given subgoal. To overcome this problem Isabelle's basic tactics are supplied with a subgoal number. This has consequences for tactical proof. Tacticals enable tactics to be composed to form proof strategies. If individual tactics are identified with particular subgoal numbers, then generality is lost. Paulson makes provision for a *round-robin* style of proof development which removes this problem. Each subgoal is taken in turn and an attempt is made to resolve it against a given list of rules. This approach, however, introduces the problem of deciding which subgoals should be developed first and which rules should be applied. Paulson uses a branching limit to deal with the potential combinatorial explosion. If the number of resolvants for a particular goal exceeds the branching limit, the goal is not developed. In this way the branching limit delays the decomposition of overly flexible goals. This approach to controlling the search for a proof does not take into account, however, the structure of the goal explicitly. As indicated above, the conventional tacticals are not appropriate for the style of proof development supported by Isabelle. Paulson, however, incorporates breadth first and depth first tacticals. Both tacticals are supplied with a predicate which determines which resolvant is selected at each step in the application of the tactic. The use of a predicate, rather than a branching limit, enables the search for a proof to be guided by the

structure of the goal.

Related to this is the problem of dealing with multiple unifiers. As mentioned earlier, the problem of the constructibility of subgoals is replaced by the task of determining which unifier is most appropriate. Consider, for example, that we wish to establish the associativity of addition over the natural numbers. This goal is expressed by the type

$$Eq(N, plus(x, plus(y, z)), plus(plus(x, y), z))$$

where x, y and z are of type N. Proof is by induction over the natural numbers. The composition of the initial goal with the rule for induction over natural numbers is achieved by higher-order unification. The unification process generates fourteen abstractions:

$\{(n)Eq(N, plus(x, plus(y, z)), n)$
$(n)Eq(N, plus(x, plus(y, n)), plus(plus(x, y), n))$
$(n)Eq(N, plus(x, plus(y, z)), plus(plus(x, y), n))$
$(n)Eq(N, plus(x, plus(y, z)), plus(n, z))$
$(n)Eq(N, plus(x, plus(n, z)), plus(plus(x, n), z))$
$(n)Eq(N, plus(x, plus(y, z)), plus(plus(x, n), z))$
$(n)Eq(N, plus(n, plus(y, z)), plus(plus(n, y), z))$
$(n)Eq(N, plus(x, plus(y, z)), plus(plus(n, y), z))$
$(n)Eq(N, n, plus(plus(x, y), z))$
$(n)Eq(N, plus(x, n), plus(plus(x, y), z))$
$(n)Eq(N, plus(x, plus(y, n)), plus(plus(x, y), z))$
$(n)Eq(N, plus(x, plus(n, z)), plus(plus(x, y), z))$
$(n)Eq(N, plus(n, plus(y, z)), plus(plus(x, y), z))$
$(n)Eq(N, plus(x, plus(y, z)), plus(plus(x, y), z))\}$

The required proof is achieved by induction on z which corresponds to the seventh abstraction. The other thirteen abstractions are of no use. In general, therefore, the selection of the appropriate unifier is a non-trivial problem.

Programming interface

The presentation of a proof in Isabelle directly reflects its internal representation. At each step in a proof the user is presented with an inverted derived rule of the form

Level *n*
⟨*conclusion*⟩
1. ⟨*subgoal*⟩
⋮
m. ⟨*subgoal*⟩

Consider, for example, the proposition

$$A \wedge B \Rightarrow B \wedge A$$

In the Isabelle implementation of type theory, this goal is represented by the trivial rule:

```
Level 1
?a : ?A * ?B ==> ?B * ?A
  1. ?a : ?A * ?B ==> ?B * ?A
> () : unit
```

Schema variables are preceded by a "?" symbol. The logical connectives * and ==> are defined in terms of the Σ and Π types. Definitions are folded and unfolded by the request of the user. For example, the definitions of * and ==> are unfolded in the above goal as follows:

```
- expand (unfold_goal_tac ["*","==>"]);
Level 2
?a : ?A * ?B ==> ?B * ?A
  1. ?a : PROD z:(SUM x:?A. ?B). SUM za:?B. ?A
> () : unit
```

The user is able to control whether a definition is unfolded throughout the derivation or, as above, restricted to just the subgoals.

An appealing aspect of Isabelle's rule representation is that rules are printable. For example, the Π-introduction rule is displayed by the following application of print utility print_rule:

```
- print_rule  Prod_intr;
Prod_intr
?A type [ ?H ]  ?b'(w) : ?B'(w) [ ?H, w: ?A ]
-----------------------------------------------
lambda(?b') : Prod(?A,?B')  [ ?H ]
w> ?H ?B' ?b'
> () : unit
```

Note that the rule presentation includes the restriction on the parameter w. For each refinement a new level in the goal stack is created. Refining the initial goal by Prod_intr gives rise to a new derivation. The associated derived rule takes the form:

```
-  expand (resolve_tac [Prod_intr] 1);
Level 3
lambda(?b'1) : A * B ==> B * A
  1. SUM z:A. B type
  2. ?b'1(w1) : SUM z:B. A [ w1: SUM z:A. B ]
> () : unit
```

Note that this refinement has the effect of instantiating the schema variable ?a in the goal to a function object.

3.3 Conclusion

This section is divided into two parts. A comparative study of the implementations of type theory reviewed in section 3.2 is presented, followed by a proposal for a more practical implementation. The proposal attempts to incorporate the best features of each implementation in order to attain the objectives for a programming assistant set out in section 3.1.

3.3.1 Comparative study

A comparative study of the four implementations of type theory is presented. The study is structured according to the four objectives of a programming assistant set out in section 3.1.

Programming logic

The four implementations reviewed are based upon two versions of constructive type theory. One due to Martin-Löf and the other to Constable *et al.* The notion of programs as proofs is central to the application of constructive methods to the task of program development. As a consequence, our intuitions as programmers and theorem provers can be utilised. It is for this reason that both objects and types should be given equal prominence in the theory. This is the case in Martin-Löf's formalisation, while in NuPRL proof objects are treated as a by-product of a derivation.

NuPRL's unification of Martin-Löf's four judgement forms seems to be directly related to the need for an extraction mechanism. If the extraction mech-

anism is a consequence of this unification, then the notational economy gained must justify this decision by enhancing the logic in some way. By reflecting the equality judgement in the type structure, however, we feel that the clarity of the system is not enhanced, for the simple reason that a type constructor can no longer be understood in isolation. The meaning of a particular type constructor will presuppose our understanding of the equality type.

When the type structure of each logic is compared, NuPRL seems, at first sight, to be better adapted to the needs of the programmer. However, no theory of data types can be considered as being complete. New applications demand new data structures. The NuPRL approach would be to define new data types in terms of the existing types using the quotient and inductive type constructors. Experience has shown that such an approach introduces redundancy which adds to the complexity of formal proof. An alternative approach is to permit disciplined extensions to the theory. This approach relies on the theory having a uniform structure. Backhouse[Backhouse 86b] claims that Martin-Löf's theory possesses such a structure; a structure in which the elimination and computation rules follow from the formation and introduction rules. NuPRL rules do form a pattern. There are five categories of rules:

- formation
- introduction
- elimination
- equality

- computation

It is less clear, however, to see how this pattern can be exploited, since there are also exceptions to this categorisation. For example, the atom type has no noncanonical form while the less, quotient, subset and function types are the only types with separate equality rules. A more subtle exception is that only non-recursive data types have "strong-elimination" rules. It is noted in [Malcolm & Chisholm 88] that "strong-elimination" rules for recursive data types lead either to inconsistency, or to an elimination rule with which recursive functions cannot be defined. An aspect of NuPRL which is desirable is the support for forward inference provided by its implicit elimination rules.

Both versions of type theory attempt to minimise well-formedness constraints. The oversight in Martin-Löf's formulation of his elimination rules, pointed out by Harper, does not seem significant for two reasons: Firstly, it is not clear whether the "hole" can actually be exploited, and secondly, it is easily filled. Indeed, Dyckhoff (personal communication) has shown how Petersson's formulation of Martin-Löf's rules should be modified. A more significant difference is how each logic introduces assumptions (hypotheses). In Petersson's formalization, the rule for introducing assumptions has a single well-formedness premise:

$$\frac{A\ \mathit{type}}{a : A\ \ [a : A]}$$

while in NuPRL the well-formedness obligation is absent:

$$H, x : A, H' \gg A'\ \text{ext}\ x\ \text{by hyp}\ x$$

As a result, NuPRL is restricted to an initial goal with an empty hypothesis

list. Hypotheses are expressed through the dependent function type. The well-formedness obligation is not removed; it simply arises as a premise of the dependent function introduction rule. The only advantage which seems to stem from this approach is that the well-formedness of hypotheses need only be proven once during the course of a proof. However, this approach leads to a loss of clarity in the initial goal sequent. An alternative approach would be to have the proof assistant store well-formedness proofs for future use. We believe that this approach is preferable since this aspect of the well-formedness problem is relevant to the programming environment, rather than the logic. Adopting the latter approach, however, raises the question of which theorems should be stored.

Programming framework

A prerequisite of step-wise refinement is goal-directed proof. For this reason GTTS can only be viewed as a verification tool. Goal-directed proof alone, however, is not a sufficient condition for refinement. As pointed out earlier, to be true to the approach of step-wise refinement provision must be made for schema variables. The "bottle-neck" which arises from NuPRL's dependent product introduction rule is a direct consequence of the absence of schema variables in its design. Both TTPA and Isabelle make provision for schema variables.

Formal program development is an exploratory activity. The manner in which schema variables are implemented effects the ease with which different refinements can be explored. Two instantiation strategies are possible: eager and lazy. Isabelle adopts an eager instantiation strategy which can lead to deep pruning when backtracking. Pruning in itself is not expensive, since it simply involves

popping off elements in a stack. TTPA's schema variables are instantiated lasily. Consequently, backtracking is easier. NuPRL's lack of schema variables makes backtracking potentially expensive.

The choice of proof representation also effects the ease with which different refinements can be explored. In the case of Isabelle, where no internal proof structure is maintained, a coarse style of pruning is enforced which, in general, results in a larger area of proof being undone than is actually necessary. Whereas with TTPA and NuPRL there is scope for less severe pruning, since the internal proof representation is maintained.

Finally, turning to the issue of storage, it is difficult to compare implementations. On the one hand NuPRL represents a *complete* system, whereas GTTS, TTPA and Isabelle are *experimental* implementations. The issue of storage becomes more important when dealing with realistic problems involving many theorems, definitions, proof strategies and possibly multiple derivations.

Programming tools

An important issue in the mechanisation of goal-directed proof is the problem of the constructibility of subgoals. TTPA ensures constructibility by parameterising tactics accordingly. NuPRL improves on this by providing assistance to minimise the user input. However, this assistance does not extend to the application of strategies. A more serious flaw, with respect to the constructibility of subgoals, is NuPRL's dependent product introduction rule. As mentioned earlier, the user is forced to supply the existential witness, reducing program construction to program verification. In Isabelle the problem of the constructibility of subgoals

is replaced by the choice of an appropriate unifier which, in general, is non-trivial.

A related problem is the constructibility of a conclusion which occurs in the context of forwards proofs. Only GTTS and Isabelle support forwards proof. GTTS represents one extreme, where any additional information is supplied by the user. As forwards and backwards proof are unified within Isabelle, the problem is again reduced to choosing the most appropriate unifier.

The ability to define new proof strategies in terms of the basic tactics is essential for goal-directed proof to be effective. NuPRL, TTPA and Isabelle all support a tactical style of reasoning. Nevertheless, experience has shown that as a tool for building proof strategies, tacticals are less useful than might at first appear. The problems are two-fold: Firstly, the lack of control over the application of strategies, and secondly, the process of building a strategy is undertaken in isolation from the actual proof process.

Derived rules allow large proof steps to be taken. To support derived rules seriously it is important that the justification and representation are kept separate. This is the case with Isabelle, where the rule is of primary importance. In GTTS rules are functions. Rule composition, therefore, corresponds to function application. Consequently, each time a derived rule is used it is, in effect, rejustified.

Finally, decision methods are important for reducing the burden of both forwards and backwards proof. GTTS provides decision methods for evaluation, type checking and simple equality reasoning. Similar tools are provided by NuPRL, which also includes a powerful procedure for reasoning about a restricted

form of arithmetic. A useful area of application for decision methods arises where the justification of a goal carries no computational content: proving equalities and negations are the obvious examples. Harper[Harper 85] has developed a decision method for equality based on Milner's type check algorithm[Milner 78] and a process known as *annotation* which builds typed analogs within a logic of typed terms. In chapter 5 a decision method for negation is presented based on the refutation proof technique.

Programming interface

The experimental nature of the systems reviewed makes it difficult to make direct comparisons. There are, however, several general observations which may be made. Firstly, all the systems which support goal-directed proof present the user with a fixed size of window onto the derivation. TTPA is the most constrained. It allows the user to view a single node at a time. NuPRL improves on this by presenting the user with the current node, together with the immediate subgoals. Isabelle provides a view of the complete derivation. However, no internal proof structure is maintained. It seems desirable that the user should be given freedom to choose how much of the derivation should be displayed. To achieve this it is necessary that the internal structure of a proof is maintained. It is difficult then to see how Isabelle could accommodate such a facility.

A related problem is the presentation of assumptions. Each system reviewed presents all assumptions. In the case of NuPRL and Isabelle this leads to assumptions being duplicated in the presentation. Such duplication clutters the presentation of the proof and should be avoided.

Isabelle's rule representation has benefits in terms of presentation. Since Isabelle's rules are data, they are printable. This is not the case with GTTS, TTPA and NuPRL where rules and tactics are functions. Therefore, only the effects of a rule or tactic application are visible.

3.3.2 Towards a more effective programming assistant

A proposal for a more practical implementation of type theory is presented. Based on the insights gained by the preceding review, this proposal attempts to combine the best features of these implementations.

Programming logic

Martin-Löf's theory, we believe, provides the most suitable programming logic. There are two reasons for this belief: Firstly, Martin-Löf's theory gives equal prominence to both objects and types. As noted earlier, if our intuitions as programmers as well as theorem-provers are to be exploited to the full, then proof objects must be given the same status as types within the logic. Secondly, the uniform type structure which Martin-Löf's theory possesses is necessary if our goal of permitting disciplined extensions is to be realized. NuPRL is not without merit. Forward inference, which NuPRL's implicit elimination rules support, is a desirable feature which could be incorporated uniformly within Martin-Löf's theory.

Programming framework

As indicated in section 3.1.2, an implementation of type theory must incorporate the following features within its proof representation for it to be practical:

1. goal-directed proof.
2. schema variables.
3. lazy instantiation of schema variables.
4. persistent proof representation.
5. efficient representation for derived rules.
6. storage of formal objects.

NuPRL incorporates 1,4 and 6, while Isabelle includes 1,2,5 and 6, and TTPA encompasses 1,2,3,4 and 6. Isabelle's eager instantiation strategy for schema variables and the derived rule approach to proof representation excludes the third and fourth objectives. In the case of TTPA it is an efficient rule representation which is missing.

The framework which is proposed here attempts to incorporate the benefits of a persistent proof representation and an efficient representation of derived rules. The solution we outline achieves this objective by giving less prominence to Isabelle's notion of derived rule. Instead of representing the proof by a derived rule, the basis for its construction is distributed over a tree structure. The architecture is very similar to TTPA, in that proof construction is divided into two phases: the construction of a goal tree and its conversion into a proof tree.

However, the notion of validation, which is achieved by function application in TTPA, is replaced by rule composition. A goal is now refined by unifying it with a given rule. The unification determines the form of the subgoals. A derivation is represented by a tree structure where a node contains three components:

- goal
- rule identifier
- stream of unifiers

By keeping the rule and the unifiers separate, the benefits of lazy instantiation demonstrated by TTPA are achieved. As an example, consider the goal G and the rule r:

$$\frac{P_1 \ldots P_n}{Q}$$

Refinement of G by r is achieved by the unification of the goal G with the conclusion Q. Unification generates a possibly infinite stream of unifiers. Denoting the first unifier by Φ, such that:

$$\Phi(G) = \Phi(Q)$$

and the remaining stream of unifiers by σ then refinement by r generates a derived rule of the form

$$\frac{P'_1 \ \ldots \ P'_n}{G'}$$

Here $G' = \Phi(G)$ and $P'_i = \Phi(P_i)$, where i is in the range $1 \leq i \leq n$. The node corresponding to the refinement of goal G takes the form

$$\langle G, r, \langle \Phi, \sigma \rangle \rangle$$

Note that it is G and not G' which is stored in the goal node. The substitutions Φ are used in calculating the subgoals $P'_1, \ldots, P'_n$. In this way instantiation of schema variables is delayed. The refinement is justified by the rule and the unifier. By maintaining the internal structure of a proof, it is possible to extract subderivations as rules. This proof representation is not without disadvantages. By opting for a tree representation, the ease with which subgoals can be merged, as demonstrated by Isabelle, is lost.

Finally, as mentioned previously, a storage facility for formal objects is essential for a practical implementation. In particular, a mechanism for managing collections of related theorems would be necessary. Sannella and Burstall have developed such a mechanism for LCF[Sannella & Burstall 83].

Programming tools

Schema variables provide a useful refinement tool. However, the explosion of schema variables resulting from certain unifications can obscure the goal being tackled. The use of unification in ensuring the constructibility of subgoals should, therefore, not preclude the user from volunteering information in order to constrain the unification process.

One of the advantages of Isabelle's architecture, which is lost, is the ease with which a complete goal tree can be refined. NuPRL transformation tactics provide an alternative, but less efficient, approach in the context of our chosen proof representation.

A restriction mechanism is required for tactics based on the form of a goal, rather than its flexibility or position. Schmidt[Schmidt 84] provides such a mech-

anism. To illustrate its operation consider that a tactic t is to be restricted to goals of the form $||H \triangleright_\tau \ a : A||$. Note that the symbol "$\triangleright_\tau$" is used to separate assumptions from the required conclusion and provides a way of distinguishing between a goal and a theorem. A tactic t', corresponding to the restricted version of t, may be expressed as

$$t' = t \upharpoonright ||H \triangleright_\tau \ a : A||$$

As noted in the comparative study, there is a need for support in building proof strategies. The problem when developing strategies is to determine the form and number of subgoals generated by each refinement. Experience has shown that the development of non-trivial proof strategies involves first sketching a proof on paper. Such a proof sketch is then used as a guide to the encoding of the strategy. We envisage a tactic editor driven by Schmidt's restriction mechanism which eliminates the need for the proof sketch. Schmidt's notion of restriction is extended so tactics have associated *pre-* and *post*-conditions. An initial restriction is presented to the editor which responds with a prompt for a tactic. Each time a tactic is supplied the editor checks that it is appropriate. This is achieved by ensuring the *pre*-condition of the tactic subsumes the current *post*-condition. Once a tactic is accepted the user is prompted for a tactical. The tactic editor responds with a new set of *post*-conditions. This approach would not only ease the process of building strategies, but it would also produce *pre-* and *post*-conditions for proof strategies. These conditions provide a useful documentation device and enable the editor to use proof strategies as well as the primitive tactics of the

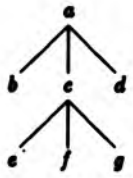
system.

Uniform procedures for general programming problems are essential, especially where the application carries no computational content. Obvious examples are the tasks of proving equalities and negations.

Programming interface

A definition mechanism is an essential component of any implementation of type theory. From our experience of NuPRL, GTTS and Isabelle, an explicit folding/unfolding mechanism seems the most practical approach.

All the systems reviewed which support top-down refinement only provide the user with a fixed view of the structure of a proof. The approach we promote gives greater flexibility by applying the idea of a *folding editor* to the presentation of derivations. A fold in the context of a derivation has the effect of hiding a subderivation. To illustrate the idea, consider the following schematic derivation:



and a corresponding linear representation:

```
a
1.    b
2.    c
3.        e
4.        f
5.        g
6.    d
```

Indentation is used to indicate subderivations. An application of `fold 2` would have the following effect:

a
1. *b*
2. — *c*
3. *d*

where the "—" symbol indicates a fold. The inverse command `unfold` would also be provided. The ability to focus on a particular subderivation is desirable. We envisage commands such as `focus` and `unfocus` to achieve this effect. To illustrate their use consider the following derivation achieved by the application of `focus 2` to the preceding derivation:

c
1. *e*
2. *f*
3. *g*

Note that `focus` automatically invokes `unfold` if required. The converse command, `unfocus`, requires the introduction of an additional command `root`. An application of `unfocus` first invokes `root`, which returns to the immediate root node, then `fold` is invoked. The ability to elide steps from a derivation would also enhance presentation. The important point is that the underlying proof representation is maintained allowing, as a result, these various views of a derivation.

It was mentioned earlier that one draw-back of both the NuPRL and Isabelle presentations is the duplication of assumptions. By exploiting the indentation which distinguishes subderivations, assumptions at a higher level can be subsumed by lower levels. To illustrate this idea, consider the following derivation:

```
a [a1, a2, a3, a4]
1.      b [a1, a2, a3, a4]
2.      c [a1, a2, a3, a4, a5, a6]
3.          e [a1, a2, a3, a4, a5, a6, a7]
4.          f [a1, a2, a3, a4, a5, a6]
5.          g [a1, a2, a3, a4, a5, a6]
6.      d [a1, a2, a3, a4]
```

This derivation would be transformed as follows:

```
a [a1, a2, a3, a4]
1.      b [...]
2.      c [... a5, a6]
3.          e [... a7]
4.          f [...]
5.          g [...]
6.      d [...]
```

where an node inherits the assumptions above it. Dots are used to indicate where assumptions have been elided. Provision should be made for expanding assumption lists on request.

Finally, a mouse and menu-driven interface is envisaged. This approach to specifying refinements significantly reduces the effort on the part of the user in building derivations. The benefits of a mouse and menu-driven interface in the context of interactive proof are demonstrated by the Edinburgh IPE project[Burstall & Ritchie 86], Dyckhoff's Machine Assisted Logic Teaching project [Dyckhoff 88] and by the work of Hamilton[Hamilton 89] in the context of TTPA.

Chapter 4

An Exercise in Program Construction

As noted by Dijkstra[Dijkstra 72], "scale" is one of the principal difficulties in constructing correct programs. The techniques employed in developing small "demonstration" programs do not, in general, scale up to "life-size" programs. To investigate ways in which machine assistance can alleviate the problem of scale, it is essential to gain experience in performing formal proof. For this reason the formal derivation of a generalised table look-up function was undertaken. Although far from being a "life-size" problem, the benefits of this exercise lie in the fact that the derivation was formalised completely. In chapter 3 we reviewed current implementations of type theory and evaluated their suitability in the role of programming assistant. The programming exercise presented here is formalised using TTPA. We concern ourselves, however, only with the general issues arising from the application of the theory.

In section 4.1 we formally specify a generalised table look-up function in type theory. An informal derivation of a program satisfying this specification is presented in section 4.2. The formal derivation in TTPA is discussed in section 4.3. Formalising the program derivation in TTPA revealed difficulties in proving negations. These difficulties are discussed in section 4.4. The work presented in this chapter is summarised in section 4.5.

4.1 Program specification

In this section we formally specify a generalised table look-up function in type theory. The problem of searching a table for a particular item is a very common programming task. We chose the simplest of representations for a *table*: a list of pairs where the first component is the *key* and the second is the *data* item. We assume that A and B are types, where A denotes the *key* type, and B the type of the *data* items. The searching process involves comparing *keys*. Therefore, we make the additional assumption that equality on A is decidable. Formally, the derivation of the table look-up function takes place in the context of the following assumptions

$$A : U_1$$
$$B : U_1$$
$$equ : (\Pi x : A)(\Pi y : A)Eq(A, x, y) + \neg Eq(A, x, y)$$

which we shall refer to as context C_1. Goals and theorems are distinguished by the symbol "$\triangleright_T$", as mentioned in chapter 3. We chose to specify the searching task implicitly, as this gives greatest freedom in selecting an implementation. We begin by introducing a definition for *table* membership:

$$Member \equiv [a, l, A, B](\Sigma h : List(A \times B))$$
$$(\Sigma t : List(A \times B))$$
$$(\Sigma b : B)$$
$$Eq(List(A \times B), append(h, \langle a, b \rangle :: t), l)$$

The intuitive idea behind this definition is as follows: If a denotes a *key* in the *table* l, then there exists *table* segments h and t, and a *data* item b for which the proposition

$$append(h, \langle a, b \rangle :: t) = l$$

is true. Using *Member*, the specification of the table look-up task is represented by the type

$$(\Pi a : A)$$
$$(\Pi l : List(A \times B))$$
$$Member(a, l, A, B) + \neg Member(a, l, A, B)$$

A program satisfying this definition is a function which maps a *key* and a *table* into an object in the type

$$Member(a, l, A, B) + \neg Member(a, l, A, B)$$

Assuming that the given *key* is present in the *table*, then the resulting output is a left injection, where the injected value takes the form

$$\langle x, \langle y, \langle z, e \rangle \rangle \rangle$$

Here x and y are *table* segments and z the indexed *data*. The e is a justification of the proposition

$$append(x, \langle a, z \rangle :: y) = l$$

If the *key* is not present in the *table*, then the result is a right injection. A right injection corresponds to an error condition in the conventional programming sense.

The injected value being a function which maps objects in $Member(a, l, A, B)$ into an object in the empty type.

This specification is somewhat unnatural in that the output from a program satisfying this specification includes not only the required data item, but also verification information. The problem is with the specification of *Member*, or rather the Σ type on which it is based. A constructive justification of the Σ type includes a demonstration of the validity of the exhibited existential witness. The subset type was introduced into the theory independently by Constable [Constable 85] and Nordström and Petersson [Nordström & Petersson 83], in order to alleviate this redundancy. Using the subset type, the definition of *Member* becomes

$$\begin{aligned} Member \equiv [a, l, A, B] \\ \{b : B | (\Sigma h : List(A \times B)) \\ (\Sigma t : List(A \times B)) \\ Eq(List(A \times B), append(h, \langle a, b \rangle :: t), l)\} \end{aligned}$$

An object in this type also belongs to B. The subset type was not used in the programming exercise presented here because it was not supported by the version of GTTS used in formalising the proof.

4.2 Informal program construction

In this section we present an informal derivation of a program satisfying the table look-up specification. Linear search of the table is implemented by list recursion which is achieved by induction on l. The base case corresponds to searching an empty *table* and is, therefore, achieved by a right injection, where the injected value is an *absurdity* proof. For the inductive step we let l denote

$u :: v$, assuming that u belongs to $A \times B$, v belongs to $List(A \times B)$, and where the inductive hypothesis is given by the assumption

$$w : Member(a, v, A, B) + \neg Member(a, v, A, B)$$

The inductive step determines whether or not a, the given *key*, is a member of the current table segment. In the case where a and $fst(u)$ are equal, then the constructed object takes the form of a left injection. The value of the left injection is

$$\langle nil, \langle v, \langle snd(u), e \rangle \rangle \rangle$$

Alternatively, if a is not equal to $fst(u)$, then the search process is applied recursively to v, the remainder of the *table*. The value of the recursive step is denoted by w, the inductive hypothesis. The value of w is an injection, therefore, analysis by cases is required. If a indexes an entry in v, then w is a left injection. By letting x denote the injected value, the derived program fragment is

$$\langle u :: fst(x), \langle fst(snd(x)), \langle fst(snd(snd(x))), e \rangle \rangle \rangle$$

If a is not a member of v, then w denotes a right injection, where the injected value is an *absurdity* proof.

We now present a slightly more rigorous account of the program derivation. A top-down style of presentation is used. Consequently, a schematic representation for proof objects is required. We adopt the convention of using the symbol "$?_n$" as a place-holder for the proof object associated with subgoal n. The initial goal, therefore, takes the form

(1) $\|\, C_1$
$\triangleright_T \;\; ?_1 \; : (\Pi a : A)$
$\qquad\qquad (\Pi l : List(A \times B))$
$\qquad\qquad\quad Member(a, l, A, B) + \neg Member(a, l, A, B)$
$\|$

Refinement of 1

By Π-introduction the quantification associated with 1 is stripped off giving rise to a subgoal of the form

(2) $\|\, C_1$
$;\; a : A$
$;\; l : List(A \times B)$
$\triangleright_T \;\; ?_2 \; : Member(a, l, A, B) + \neg Member(a, l, A, B)$
$\|$

The corresponding program fragment takes the form

$$\lambda a.\lambda l.\; ?_2$$

Refinement of 2

Refinement proceeds by induction on l giving rise to two cases:

(2.1) $\|\, C_2$
$\triangleright_T \;\; ?_{2.1} \; : Member(a, nil, A, B) + \neg Member(a, nil, A, B)$
$\|$

(2.2) $\|\, C_2$
$;\; u : A \times B$
$;\; v : List(A \times B)$
$;\; w : Member(a, v, A, B) + \neg Member(a, v, A, B)$
$\triangleright_T \;\; ?_{2.2} \; : Member(a, u :: v, A, B) + \neg Member(a, u :: v, A, B)$
$\|$

Where context C_2 is constructed from C_1 by the addition of the following assumptions

$$a : A$$
$$l : List(A \times B)$$

The program fragment resulting from the inductive argument takes the form

$$listrec(l, ?_{2.1}, [u, v, w]?_{2.2})$$

Refinement of 2.1

By the definition of table membership, the type

$$Member(a, nil, A, B)$$

is empty. As a result a right injection is required. The injected object represents a function which returns absurdity for any object in $Member(a, nil, A, B)$. Denoting such an object by $absurdity_1$, then subgoal 2.1 is achieved by the program

$$inr(absurdity_1)$$

Refinement of 2.2

Consider the non-empty *table*, denoted by $u :: v$. Two possibilities arise: Firstly, a and $fst(u)$ may be equal, in which case the search terminates. Otherwise the remaining table segment v must be searched. We initially assumed the existence of a program, denoted by *equ*, which determines whether two objects in A are equal. Applying *equ* to a and $fst(u)$ generates an object in the type

$$Eq(A, a, fst(u)) + \neg Eq(A, a, fst(u))$$

Refinement proceeds by analysing the structure of the constructed object. Two cases arise:

(2.2.1)
$$\begin{array}{l} [\![\, C_3 \\ ;\ p : Eq(A, a, fst(u)) \\ \triangleright_7 \quad ?_{2.2.1} : Member(a, u :: v, A, B) + \neg Member(a, u :: v, A, B) \\ ]\!] \end{array}$$

(2.2.2) $\|\, C_3$
$;\ q : \neg Eq(A, a, fst(u))$
$\triangleright_7 \quad ?_{2.2.2} : Member(a, u :: v, A, B) + \neg Member(a, u :: v, A, B)$
$\|$

Context C_3 is constructed from C_2 by the addition of the following assumptions

$$u : A \times B$$
$$v : List(A \times B)$$
$$w : Member(a, v, A, B) + \neg Member(a, v, A, B)$$

The program fragment derived by the above case analysis is

$$when(equ\ [a]\ [fst(u)], [p]\ ?_{2.2.1},\ [q]\ ?_{2.2.2})$$

Refinement of 2.2.1

In the case where a and $fst(u)$ are equal, the constructed program fragment is a left injection, where the injected value takes the form

$$\langle nil, \langle v, \langle snd(u), e \rangle \rangle \rangle$$

Here e denotes a justification for the proposition

$$append(nil, \langle a, snd(u) \rangle :: v) = u :: v$$

Refinement of 2.2.2

Alternatively, if a and $fst(u)$ are not equal, then the remainder of the table must be searched. Refinement proceeds by case analysis on the inductive hypothesis denoted by w. Two cases arise:

(2.2.2.1) $\|\, C_4$
$;\ x : Member(a, v, A, B)$
$\triangleright_7 \quad ?_{2.2.2.1} : Member(a, u :: v, A, B) + \neg Member(a, u :: v, A, B)$
$\|$

(2.2.2.2) $\|\, C_4$
$;\ y : \neg Member(a, v, A, B)$
$\triangleright_7 \quad ?_{2.2.2.2} : Member(a, u :: v, A, B) + \neg Member(a, u :: v, A, B)$
$\|$

Context C_4 is constructed from C_3 by the addition of the following assumption

$$q : \neg Eq(A, a, fst(u))$$

The resulting program fragment takes the form

$$when(w, [x]\ ?_{2.2.2.2.1}, [y]\ ?_{2.2.2.2.2})$$

Refinement of 2.2.2.2.1

Assuming the existence of a program x satisfying the specification

$$Member(a, v, A, B)$$

then a program satisfying the specification

$$Member(a, u :: v, A, B)$$

is constructed by adding the current table entry u onto the first component of x. The resulting program fragment is

$$\langle u :: fst(x), \langle fst(snd(x)), \langle fst(snd(snd(x))), e\rangle\rangle\rangle$$

from which subgoal 2.2.2.1 is achieved by constructing a left injection.

Refinement of 2.2.2.2.2

In the case where a does not index an entry in v, then we require an object in the type

$$\neg Member(a, u :: v, A, B)$$

The derivation of such an object is achieved by assuming $Member(a, u :: v, A, B)$ and showing that a contradiction follows. The required contradiction is based upon the following assumptions

$$q : \neg Eq(A, a, fst(u))$$
$$y : \neg Member(a, v, A, B)$$

Denoting the associated absurdity proof object by $absurdity_2(q, y)$, then subgoal 2.2.2.2 is achieved by a right injection of the form

$$inr(absurdity_2(q, y))$$

The derived judgement, or theorem, resulting from the above refinement process is

$$
\begin{array}{l}
\| \; C_1 \\
\triangleright \;\; \lambda a.\lambda l.\; listrec(l, inr(absurdity_1), \\
\qquad [u, v, w]when(equ[a]\,[fst(u)], \\
\qquad\qquad [p]inl(\langle nil, \langle v, \langle snd(u), a\rangle\rangle\rangle), \\
\qquad\qquad [q]when(w, [x]inl(\langle u :: fst(x), \\
\qquad\qquad\qquad\qquad \langle fst(snd(x)), \\
\qquad\qquad\qquad\qquad\quad \langle fst(snd(snd(x))), a\rangle\rangle\rangle), \\
\qquad\qquad\qquad\qquad [y]inr(absurdity_2(q, y))))) \\
\qquad : (\Pi a : A) \\
\qquad\quad (\Pi l : List(A \times B)) \\
\qquad\qquad Member(a, l, A, B) + \neg Member(a, l, A, B) \\
\|
\end{array}
$$

4.3 Formal program construction

The derivation of the table look-up function outlined in section 4.2 has been formalised using TTPA. We chose TTPA for two reasons: Firstly, because it supports step-wise refinement, and secondly, it was the system available at the time this work was undertaken.

Step-wise refinement is supported by TTPA, as described in chapter 3, through LCF style tactics. For each type theory rule a corresponding tactic is provided. Early experience with TTPA, however, revealed that the basic set of tactics were

too primitive to be practical. For this reason the basic set of LCF tacticals was implemented: THEN, THENL, ORELSE and REPEAT. Tacticals provide a mechanism for combining tactics to form more powerful proof strategies. However, as discussed in chapter 3, tacticals are less useful than they might first appear. The basic problem is that strategies are constructed independently from the proof. The strategies developed in the derivation of the table look-up function were found to be a useful aid for documenting the proof, but gave little support in the search for a proof. Nevertheless, it is possible to construct generally applicable strategies. These strategies, however, tend to be restricted to the application of formation and introduction rules, where the outer structure of the goal determines which tactic should be applied.

Early experience also revealed problems with the definition mechanism supported by GTTS. These problems involved the premature unfolding of definitions which made the goal presentation hard to read. Similar difficulties were experienced by Chisholm[Chisholm 87] in the derivation of a parsing function using GTTS. Chisholm used GTTS as a verification tool and dealt with the definition problem by altering the proof script generated by the system. The derivation of the table look-up function was produced interactively in a goal-directed manner. Consequently, to overcome the problem with definitions, new constants were introduced together with defining rules:

- *append*
- *fst*
- *snd*

The rules are given in appendix B.1. These rules represent an extension to the theory. In general, this is not a good approach since inconsistencies may be unknowingly introduced into the theory. Ideally, such rules should be derived within the theory, leaving the definition mechanism to take care of the abbreviations. The decision to extend the theory was made purely for pragmatic reasons in order to ease the goal presentation.

Space restrictions forced the proof to be conducted in three parts. The corresponding goals are given in appendix B.1. The derivation of the table look-up function is reflected in a hierarchy of proof strategies, as indicated in figure 4.1. The individual strategies are presented in appendix B.2. The proof script is too large to include in its entirety. Therefore we include, in appendix B.3, the theorems derived by the strategies highlighted in figure 4.1.

4.4 Proving negations

Negation is an example of what Backhouse[Backhouse 87] describes as the mismatch between programs and proofs. A proof of a negation has no computational content. In terms of our intuitions as programmers, a proof of a negation denotes an error state. Proof is necessary only to ensure the correctness of the overall program. For this reason we are only concerned with the existence of a proof. Two negations arose in the course of deriving the table look-up function, which we referred to as $absurdity_1$ and $absurdity_2$ in section 4.2. Proving a negation corresponds to establishing a contradiction. Formally identifying the contradictions which underlie $absurdity_1$ and $absurdity_2$ was found to be non-trivial. Proofs

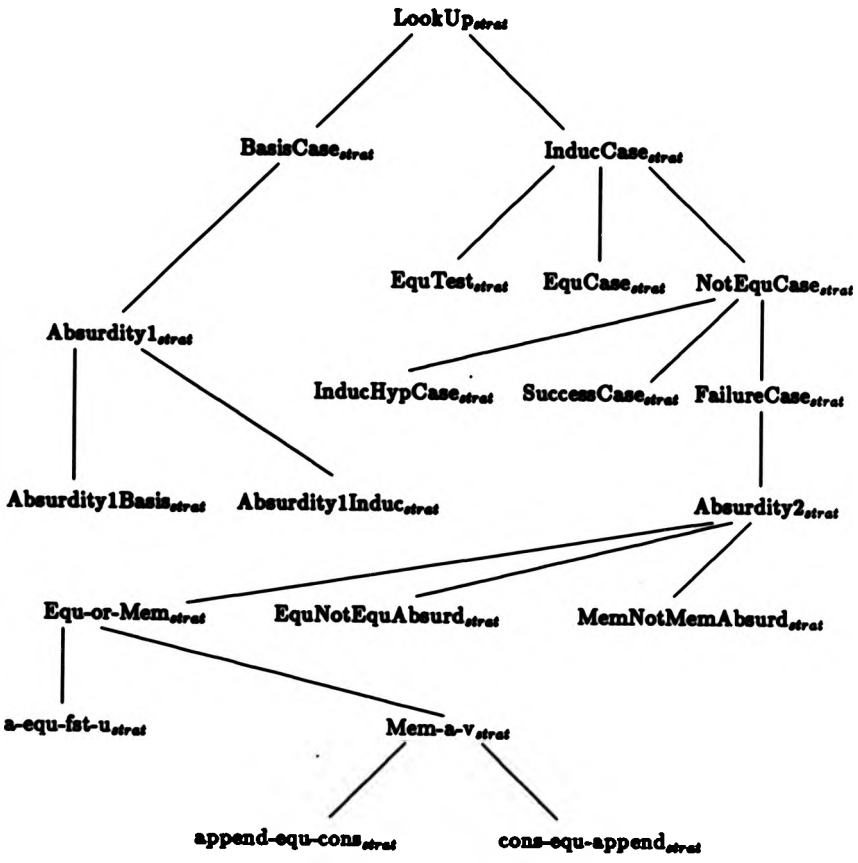


Figure 4.1: Proof strategy hierarchy

of $absurdity_1$ and $absurdity_2$ are outlined in sections 4.4.1 and 4.4.2 respectively. The difficulties surrounding the proofs of negations are discussed in section 4.4.3, where the case for mechanised assistance in proving negations is argued.

4.4.1 Proof of $absurdity_1$

Establishing $absurdity_1$ corresponds to constructing an object in the type

$$\neg Member(a, nil, A, B) \tag{4.1}$$

Proof is by contradiction. We assume the existence of an object in the type

$$Member(a, nil, A, B)$$

and derive an object in the empty type. The basis of the contradiction becomes apparent by inspecting the equality type which appears in the definition of *Member*:

$$\begin{aligned} Member \equiv [a, l, A, B](\Sigma h : List(A \times B)) \\ (\Sigma t : List(A \times B)) \\ (\Sigma b : B) \\ Eq(List(A \times B), append(h, \langle a, b \rangle :: t), l) \end{aligned}$$

Note that the *left-hand-side* of this equality reduces to a compound list, irrespective of the values chosen for the existential witnesses h, t and b. In the particular instance of *Member* corresponding to $absurdity_1$, the parameter l is *nil*. Consequently, $Member(a, nil, A, B)$ reduces to an equality between a compound list and the empty list.

Formally identifying this contradiction is less intuitive. By assuming

$$x : Member(a, nil, A, B) \tag{4.2}$$

our goal is reduced to constructing an object in $\emptyset$. From an object in $\emptyset$ an object in 4.1 is constructed by $\rightarrow$-introduction. Expressing the underlying contradiction in terms of assumption 4.2:

$$Eq(List(A \times B), append(fst(x), \langle a, fst(snd(snd(x)))\rangle :: fst(snd(x))), nil)$$

the problem is reduced, by $\rightarrow$-elimination, to showing that the types

$$Eq(List(A \times B), append(fst(x), \langle a, fst(snd(snd(x)))\rangle :: fst(snd(x))), nil) \rightarrow \emptyset \tag{4.3}$$

$$Eq(List(A \times B), append(fst(x), \langle a, fst(snd(snd(x)))\rangle :: fst(snd(x))), nil) \tag{4.4}$$

are non-empty. Constructing an object in 4.4 follows from assumption 4.2 by Σ-elimination. Establishing 4.3 is achieved by reducing the *left-hand-side* of the equality. Proof proceeds by induction on $fst(x)$. Here we consider only the base case, that is, when $fst(x)$ is nil:

$$Eq(List(A \times B), append(nil, \langle a, fst(snd(snd(x)))\rangle :: fst(snd(x))), nil) \rightarrow \emptyset \tag{4.5}$$

Proof is again by contradiction. Assuming

$$y : Eq(List(A \times B), append(nil, \langle a, fst(snd(snd(x)))\rangle :: fst(snd(x))), nil) \tag{4.6}$$

our goal is reduced to constructing an object in $\emptyset$. To complete the proof a method is required which maps assumption 4.6 into an object in $\emptyset$. Arguing forwards, given an object in the type

$$List(A \times B) \tag{4.7}$$

and a proof of the judgement

$$\mathit{List}(A \times B) = \mathbb{0} : U_1 \tag{4.8}$$

then an object in $\mathbb{0}$ is constructed by an application of the type equality rule. The canonical constant *nil* provides an object in 4.7. Satisfying 4.8 is less trivial and involves the *List* eliminator. Given proofs of

$$\mathit{listrec}(\mathit{nil}, \mathbb{0}, [u, v, w]\mathit{List}(A \times B)) = \mathbb{0} : U_1 \tag{4.9}$$

$$\begin{aligned}\mathit{listrec}(\langle a, \mathit{fst}(\mathit{snd}(\mathit{snd}(x)))\rangle :: \mathit{fst}(\mathit{snd}(x)), \mathbb{0}, [u, v, w]\mathit{List}(A \times B)) \\ = \mathit{List}(A \times B) : U_1\end{aligned} \tag{4.10}$$

$$\begin{aligned}\mathit{listrec}(\langle a, \mathit{fst}(\mathit{snd}(\mathit{snd}(x)))\rangle :: \mathit{fst}(\mathit{snd}(x)), \mathbb{0}, [u, v, w]\mathit{List}(A \times B)) \\ = \mathit{listrec}(\mathit{nil}, \mathbb{0}, [u, v, w]\mathit{List}(A \times B)) : U_1\end{aligned} \tag{4.11}$$

a proof of 4.8 follows by transitivity. Proofs of 4.9 and 4.10 are constructed using the computation rules for the *List* type. From proofs of

$$\langle a, \mathit{fst}(\mathit{snd}(\mathit{snd}(x)))\rangle :: \mathit{fst}(\mathit{snd}(x)) = \mathit{nil} : \mathit{List}(A \times B) \tag{4.12}$$

$$\begin{aligned}\mathit{listrec}(s, \mathbb{0}, [u, v, w]\mathit{List}(A \times B)) \\ = \mathit{listrec}(s, \mathbb{0}, [u, v, w]\mathit{List}(A \times B)) : U_1\end{aligned} \tag{4.13}$$

a proof of 4.11 follows by substitution. Assumption 4.6 provides the basis for the proof of 4.12 and is achieved using the rules for transitivity and *Eq*-elimination. Finally, given the assumption

$$s : \mathit{List}(A \times B)$$

a proof of 4.13 is constructed by *List*-elimination. The judgement resulting from the proof of $\mathit{absurdity}_1$ is

$$
\begin{array}{l}
[\![\, C_3 \\
\quad \triangleright \;\; \lambda x.(listrec(fst(x), \lambda y.nil, \\
\qquad\qquad\qquad\qquad [u,v,w]\lambda z.nil)\ [snd(snd(snd(x)))]) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad : \neg Member(a, nil, A, B) \\
[\!]
\end{array}
$$

4.4.2 Proof of *absurdity*$_2$

Establishing *absurdity*$_2$ corresponds to constructing an object in the type

$$\neg Member(a, u :: v, A, B) \tag{4.14}$$

Proof is by contradiction and takes place in the context of assumptions:

$$q : \neg Eq(A, a, fst(u)) \tag{4.15}$$

$$y : \neg Member(a, v, A, B) \tag{4.16}$$

We assume the existence of an object in

$$Member(a, u :: v, A, B)$$

and derive objects in

$$Eq(A, a, fst(u)) \tag{4.17}$$

$$Member(a, v, A, B) \tag{4.18}$$

Intuitively, from an object in 4.17, a contradiction follows as a consequence of assumption 4.15. Similarly, given an object in 4.18, a contradiction follows by assumption 4.16. To formalise this intuitive argument we begin by assuming

$$c_1 : Member(a, u :: v, A, B) \tag{4.19}$$

Our goal is reduced to constructing an object in $\emptyset$. Given an object in $\emptyset$, an object in 4.14 is established by $\rightarrow$-introduction. The underlying contradiction is expressed by the disjunction

$$Eq(A, a, fst(u)) + Member(a, v, A, B) \tag{4.20}$$

Given an arbitrary object in 4.20, a method is required which yields an object in $\emptyset$. Proof is by case analysis requiring an application of $+$-elimination. In the case of a left injection we assume

$$c_1 : Eq(A, a, fst(u))$$

and by $\rightarrow$-elimination a contradiction is established by assumption 4.15. Similarly, for a right injection we assume

$$c_2 : Member(a, v, A, B)$$

and derive a contradiction by assumption 4.16. To complete the proof we require a method which yields an object in 4.20, given an object in 4.19. Expressing the underlying contradiction in terms of assumption 4.19:

$$Eq(List(A \times B), append(fst(c_1), \langle a, fst(snd(snd(c_1)))\rangle :: fst(snd(c_1))), u :: v)$$

the problem is reduced, by $\rightarrow$-elimination, to showing that the types

$$\begin{aligned} &Eq(List(A \times B), \\ &\quad append(fst(c_1), \langle a, fst(snd(snd(c_1)))\rangle :: fst(snd(c_1))), u :: v) \\ &\qquad \rightarrow Eq(A, a, fst(u)) + Member(a, v, A, B) \end{aligned} \tag{4.21}$$

$$\begin{aligned} &Eq(List(A \times B), \\ &\quad append(fst(c_1), \langle a, fst(snd(snd(c_1)))\rangle :: fst(snd(c_1))), u :: v) \end{aligned} \tag{4.22}$$

are not empty. Construction of an object in 4.22 follows from assumption 4.19 by Σ-elimination. A proof of 4.21 gives rise to a function which produces an injection. The form of the injection depends upon the structure of the initial table segment given by $fst(c_1)$. If $fst(c_1)$ is *nil*, then a left injection is generated. Conversely, a right injection results if $fst(c_1)$ is a compound list. Proof proceeds, therefore, by induction on $fst(c_1)$. Here we deal only with the base step:

$$\begin{aligned} &Eq(List(A \times B), \\ &\quad append(nil, \langle a, fst(snd(snd(c_1)))\rangle :: fst(snd(c_1))), u :: v) \\ &\qquad \rightarrow Eq(A, a, fst(u)) + Member(a, v, A, B) \end{aligned} \tag{4.23}$$

Assuming

$$\begin{aligned} c_4 : &Eq(List(A \times B), \\ &append(nil, \langle a, fst(snd(snd(c_1)))\rangle :: fst(snd(c_1))), u :: v) \end{aligned} \tag{4.24}$$

then the base step is reduced by $\rightarrow$-introduction to constructing an object in the type

$$Eq(A, a, fst(u)) + Member(a, v, A, B) \tag{4.25}$$

By $+$- and Eq-introduction 4.25 is reduced to an equality of the form

$$a = fst(u) : A \tag{4.26}$$

Proof of 4.26 rests upon assumption 4.24. Transitivity reduces our goal to

$$fst(\langle a, fst(snd(snd(c_1)))\rangle) = fst(u) : A$$

which is further refined, by $\times$-elimination$_{\#4}$, giving rise to

$$\langle a, fst(snd(snd(c_1)))\rangle = u : A \tag{4.27}$$

The next step involves showing that 4.27 follows from a proof of

$$\langle a, \mathit{fst}(\mathit{snd}(\mathit{snd}(c_1)))\rangle :: \mathit{fst}(\mathit{snd}(c_1)) = u :: v : \mathit{List}(A \times B)$$

List induction is required. By transitivity, 4.27 is reduced to

$$\begin{aligned}&\mathit{listrec}(\langle a, \mathit{fst}(\mathit{snd}(\mathit{snd}(c_1)))\rangle :: \mathit{fst}(\mathit{snd}(c_1))\\ &\qquad\qquad \langle a, \mathit{snd}(c_1)\rangle, [d_1, d_2, d_3]d_1)\\ &\qquad\qquad\quad = \mathit{listrec}(u :: v, \langle a, \mathit{snd}(c_1)\rangle, [d_1, d_2, d_3]d_1) : A \times B\end{aligned}$$

Proof proceeds by *List*-elimination. The base step is completed using the rules for transitivity, *List*-computation$_{nil}$, *Eq*-elimination and assumption 4.24. The judgement resulting from the proof of $\mathit{absurdity}_2$ is

$$\begin{aligned}&[\!| C_3\\ &;\ q : \neg \mathit{Eq}(A, a, \mathit{fst}(u))\\ &;\ y : \neg \mathit{Member}(a, v, A, B)\\ &\triangleright\ \lambda c_1.\mathit{when}(\mathit{listrec}(\mathit{fst}(c_1), \lambda c_4.\mathit{inl}(e),\\ &\qquad\qquad [c_5, c_6, c_7]\\ &\qquad\qquad\quad \lambda c_8.\mathit{inr}(\langle c_8, \langle \mathit{fst}(\mathit{snd}(c_1)),\\ &\qquad\qquad\qquad\qquad \langle \mathit{fst}(\mathit{snd}(\mathit{snd}(c_1))), e\rangle\rangle\rangle\rangle)\\ &\qquad\qquad [\mathit{snd}(\mathit{snd}(\mathit{snd}(c_1)))],\\ &\qquad [c_2](q[c_2]),\\ &\qquad [c_3](y[c_3]))\\ &\qquad\qquad : \neg \mathit{Member}(a, u :: v, A, B)\\ &|\!]\end{aligned}$$

4.4.3 Discussion

In our presentation of the informal proofs of $\mathit{absurdity}_1$ and $\mathit{absurdity}_2$ in sections 4.4.1 and 4.4.2, we neglected many of the proof obligations generated by the subgoaling process, in particular the respective inductive steps. Our motivation, however, is not to convince the reader of correctness, but rather to demonstrate the difficulties arising from the formulation of the relatively intuitive arguments

involved. These difficulties mean that a disproportionate amount of the overall proof effort is taken up with identifying contradictions. The proof objects convey this point quite well. Comparing the proof objects resulting from the *absurdity* proofs with the computationally significant component of the proof presented in section 4.2, we find that more of the derivation relates to proving negations than to the synthesis of the program. A similarly disproportionate amount of effort occurs in Chisholm's[Chisholm 87] derivation of a parsing algorithm. Three causes for these difficulties were identified.

The first is related to the level at which uniqueness properties are established in the theory. The proof of $absurdity_1$ rests upon the uniqueness property associated with the *List* type:

$$(\Pi x : A)(\Pi y : List(A))\neg Eq(List(A), x :: y, nil)$$

The uniqueness property states that the empty list, *nil*, and compound lists formed by the :: constructor are distinct. Formally establishing this property requires the construction of a method which maps the judgement

$$x :: y = nil : List(A)$$

into the equality

$$\emptyset = List(A) : U_1$$

Once established, subsequent proofs can appeal to the existence of such a uniqueness property. It would seem desirable, therefore, to be able to generate uniqueness properties mechanically for an arbitrary data type constructor. This idea is pursued in chapter 5.

The second difficulty concerns the complexity involved in establishing a contradiction which rests upon multiple assumptions. This is illustrated by $absurdity_2$, where the contradiction rests upon three assumptions:

$$c_1 : Member(a, u :: v, A, B)$$
$$q : \neg Eq(A, a, fst(u))$$
$$y : \neg Member(a, v, A, B)$$

Establishing $absurdity_2$ involves recognising that the assumption denoted by c_1 implies the disjunction

$$Eq(A, a, fst(u)) + Member(a, v, A, B)$$

Proof of this implication requires a function which maps an arbitrary object in

$$Member(a, u :: v, A, B)$$

into an object in $Eq(A, a, fst(u))$, or an object in $Member(a, v, A, B)$. This kind of function involves a recursive argument requiring an inductive proof.

The third difficulty is the added burden the natural deduction system places on reasoning with equalities. Consider the judgement

$$e : Eq(A, a, b)$$

Before the rules for transitivity and symmetry can be applied, the judgement must be converted into an instance of the equality judgement:

$$a = b : A$$

Evaluation presents similar problems. Consider the judgement

$$e : Eq(A, a(t), b)$$

where t denotes a saturated expression which is not fully evaluated. Since type theory computation rules evaluate from the outside in, evaluation of the *left-hand-side* of the given equality cannot be achieved directly. The subexpression t must be evaluated in isolation. Assuming t is of type T, then the evaluation of t is achieved by the computation rule associated with type T, giving rise to a judgement of the form

$$t = t' : T$$

Combining this equality with the initial equality involves the substitution and type equality rules. A judgement of the form

$$||x : T \triangleright \quad Eq(A, a(x), b) = Eq(A, a(x), b)||$$

is constructed from which the judgement

$$Eq(A, a(t), b) = Eq(A, a(t'), b)$$

follows by substitution. Taken in conjunction with the initial judgement, the type equality rule gives rise to the required conclusion:

$$e : Eq(A, a(t'), b)$$

4.5 Summary

In this chapter we presented the derivation of a generalised table look-up function. This exercise was undertaken to gain insight into the problems of scale arising from the application of the theory to the task of programming. Proving negations was identified as being non-trivial and was found to absorb a disproportionate amount of the overall proof effort. The proof of a negation is constructed

by identifying a contradiction. Three areas of difficulty were identified in establishing contradictions. The first is concerned with the level of detail required in constructing uniqueness properties. The second relates to the complexity in establishing a contradiction which is based upon multiple assumptions. The third arises from the burden the natural deduction system places on reasoning with equalities. In view of the fact that negations have no computational content, it seems a desirable object to develop mechanised assistance in establishing such proof obligations. This idea is explored further in chapter 5, where a decision method for negation is presented.

Chapter 5

A Decision Method for Negation

Formally deriving a program which satisfies a type theory specification is a non-trivial task. This is demonstrated in chapter 4 where the derivation of a generalised table look-up function is presented. In particular, this exercise revealed difficulties in proving negations. This is reflected in the corresponding derivation, in which a significant proportion of the proof effort is associated with two negations. A negation is proved by establishing a contradiction. It is the task of formalising contradictions in type theory which is lengthy.

Negations arise in the specification of most programming tasks. Indeed the specification of the table look-up function corresponds to a particular instance of the following general specification schema

$$(\Pi x_1 : X_1) \ldots (\Pi x_n : X_n)$$
$$P(x_1 \ldots x_n) + \neg P(x_1 \ldots x_n)$$

Backhouse's formulation of the Boyer-Moore Majority-Vote algorithm [Backhouse 85] [Backhouse *et al* 89] and Chisholm's derivation of a parsing algorithm [Chisholm 87] also correspond to this specification schema. The inputs are represented by the

bound variables $x_1 \ldots x_n$, while the result is specified by the disjunction. The left disjunct specifies valid results and the erroneous states are captured by the right disjunct. As the initial specification is refined, subspecifications arise corresponding to specific instances of the disjunction. Such subspecifications are achieved, either by constructing a left or right injection, or by constructing a method for computing the appropriate injection. Proving negations, like proving equalities, is an essential part of the programming task. However, such proofs have secondary status in that they contribute only to the correctness of the derived program. Computationally they have no content. A specification can, therefore, be viewed as having *creative* and *uncreative* components. Creative in the sense that the proof has computational content. It seems a desirable objective to have a system which deals automatically with the *uncreative* components, allowing the user to concentrate on the *creative* parts of the programming task. It has been suggested that nonconstructive methods should be employed to deal with these problems[Smith 87]. Whether such an approach is feasible remains to be seen. Our objective is to see how far the practical problems in constructing programs in Martin-Löf's theory of types can be solved.

In this chapter a decision method for dealing with negations is presented. As mentioned previously, the task of satisfying a negated specification corresponds to formally identifying contradictions. The method is based upon the refutation proof technique. The search for refutation is based upon analysis at the level of the equality type.

The richness of type theory excludes the possibility of a complete decision

method. Even the task of classifying goals to which a decision method is applicable seems counter-productive, since a mechanism which determines whether or not the method is applicable would itself constitute a decision method.

The chapter is structured as follows. An overview is presented in section 5.1 where the notion of negation and the refutation proof technique are outlined. Section 5.2 describes methods for mechanically deriving general properties of data types. This work forms the basis for the refutation algorithm presented in section 5.3. Aspects of the implementation of the decision method are dealt with in section 5.4. The work presented in this chapter is summarized in section 5.5.

5.1 Overview

In this section the notion of negation is discussed in the context of type theory, together with an outline of an algorithm for proof by refutation.

5.1.1 Proving negations

Negation is not a primitive of type theory. It is defined in terms of the function and empty types:

$$\neg A \equiv A \rightarrow \emptyset$$

A proof of $\neg A$ is a function which maps an arbitrary object in A into an object in the empty type. To prove $\neg A$, it is sufficient to show that the type A is contradictory. This means that if we could construct an object in A, then we could construct an object in the empty type. Contradictions arise in type theory either at the level of propositions or through the surrounding context.

Propositions are based upon the equality type. We shall refer to a contradiction arising from an equality as a *direct* contradiction. Establishing a direct contradiction corresponds to demonstrating that an equality type is empty. Formally, this means reducing both sides of an equality to the level of distinct canonical expressions, from which a contradiction follows by way of an uniqueness property. Consider, for instance, the following contradictory instance of the equality type

$$Eq(N, 0, 0')$$

The formal identification of this contradiction relies on the uniqueness property for type N. This uniqueness property states that the constant 0, and any expression constructed using $'$, are distinct canonical objects. Given the rules defining an arbitrary type constructor Θ, then the uniqueness properties for the canonical constructors introduced by Θ can be mechanically derived. A method for achieving this is presented in section 5.2.1.

Contradictions can also arise from the surrounding context. We refer to such contradictions as *indirect*. For instance, consider the task of proving $\neg A$ in the context:

```
0.0      ||f : A → B
0.1.0    ▷  ||g : B → ∅
               ⋮
               ]|
         ]|
```

A proof of $\neg A$ relies not on the structure of A, but rather on the assumptions denoted by 0.0 and 0.1.0. By extending the context with the assumption $a : A$, an object belonging to the empty type can be derived. From this contradiction

the required proof follows by an application of $\rightarrow$-introduction. The complete derivation is as follows:

Derivation
0.0 $\quad [\![f : A \rightarrow B$
0.1.0 $\quad \triangleright \quad [\![g : B \rightarrow \emptyset$
0.1.1.0 $\quad \triangleright \quad [\![a : A$
$\quad \triangleright \quad \{0.0,0.1.1.0 \rightarrow\text{-elim}\}$
0.1.1.1 $\quad f[a] : B$
$\quad \{0.1.0,0.1.1.1 \rightarrow\text{-elim}\}$
0.1.1.2 $\quad g[f[a]] : \emptyset$
$\quad]\!]$
$\quad \{0.1.1.0,0.1.1.2 \rightarrow\text{-intr}\}$
0.1.2 $\quad \lambda a.g[f[a]] : A \rightarrow \emptyset$
$\quad]\!]$
$\quad]\!]$

A proof may depend on a combination of assumptions and on both *direct* and *indirect* contradictions. A decision method for negation must, therefore, deal with these possibilities.

5.1.2 Refutation in Type Theory

Given a proposition $\neg P$, a proof can be constructed by assuming P and showing that a contradiction follows. This general technique is known as *proof by refutation*. In this section we outline a refutation algorithm for type theory, the details of which will be presented in section 5.3.

Negated equality form

For a given negation there may exist many ways it can be expressed. Consider, for instance, the judgement

$$[\![\ \ldots \triangleright \quad v : \neg(\Sigma x : A)(\Sigma y : A)Eq(A, x, y)]\!]$$

which may be transformed using the logical system into the following judgements:

$$||\ \ldots \triangleright\ \ v_1 : (\Pi x : A)(\Pi y : A)\neg Eq(A, x, y)||$$

$$||[x : A \ldots \triangleright\ \ v_2 : (\Pi y : A)\neg Eq(A, x, y)]||$$

$$||[x : A; y : A \ldots \triangleright\ \ v_3 : \neg Eq(A, x, y)]||$$

In order to have a standard way of talking about negations, we introduce the *negated equality form* which is a negated instance of the equality type:

$$\neg Eq(T, \alpha, \beta)$$

The refutation algorithm we present here is restricted to goals of the negated equality form. In order to generalise the applicability of the algorithm, goal transformations are provided which push negations through the quantifiers and logicals.

Identifying contradictions

The search for refutation is based upon the analysis of the equality type. For example, in order to prove the following negation

$$||C \triangleright\ r : \neg Eq(T, \alpha, \beta)||$$

we assume $Eq(T, \alpha, \beta)$, in the context C, and by analysis of the α and β, show that a contradiction follows, thereby establishing $\neg Eq(T, \alpha, \beta)$. Schematically, such a proof corresponds to:

Derivation
0.0 $\|C$
0.1.0 $\triangleright$ $\|p : Eq(T, \alpha, \beta)$
$\triangleright$
$\vdots$
{by analysis of α and β}
0.1.1 $z : \emptyset$
$\|$
{0.1.0,0.1.1 $\rightarrow$-intr}
0.2 $\lambda p.z : \neg Eq(T, \alpha, \beta)$
$\|$

The α and β are analysed by a process of *reduction* and *decomposition*: noncanonical forms are reduced to canonical form, while non-atomic canonical forms are decomposed into atomic canonical forms. Consider, for example, the following contradictory instance of the equality type

$$Eq(N, plus(m, n')', 0')$$

where m and n are variables of type N. Informally, the analysis process proceeds as follows. Since both sides of the equality are non-atomic canonical forms, a decomposition is performed giving rise to

$$Eq(N, plus(m, n'), 0)$$

Now we have a noncanonical form on the *left-hand-side*, so analysis proceeds by reduction. Because the first argument of the *plus* is a variable, a reduction cannot be achieved by direct evaluation. Instead, the variable m must first be analysed by cases giving rise to two equalities:

$$Eq(N, plus(0, n'), 0) \tag{5.1}$$

$$Eq(N, plus(x', n'), 0) \tag{5.2}$$

where x is a variable of type N. Further reduction can now be performed by evaluating the *left-hand-side* of each equality. Evaluation of 5.1 gives rise to

$$Eq(N, n', 0)$$

which is contradictory by way of the uniqueness property for type N. Similarly, the evaluation of 5.2 gives rise to a contradiction directly:

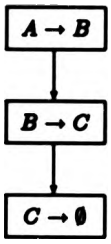
$$Eq(N, plus(x, n')', 0)$$

This analysis is based upon three kinds of derived rules:

- case analysis
- evaluation
- decomposition

Case analysis and evaluation provide the basis for reductions. For each data type constructor there exists an associated set of derived analysis rules. The structure of these rules is dealt with in detail in section 5.2.4, where methods for constructing such rules for an arbitrary data type are described.

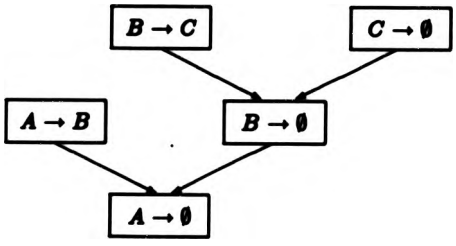
Proof by refutation is based upon the construction of a chain of implications which lead to a contradiction. Given the proposition $\neg A$, if by assuming A a chain of implications can be derived which give rise to $\emptyset$, then it follows that $A \rightarrow \emptyset$ is true:



The justification for this relies on a form of the *modus tollens* rule of inference:

$$\frac{\begin{array}{l} X \rightarrow Y \\ \neg Y \end{array}}{\neg X} \quad \text{modus tollens}$$

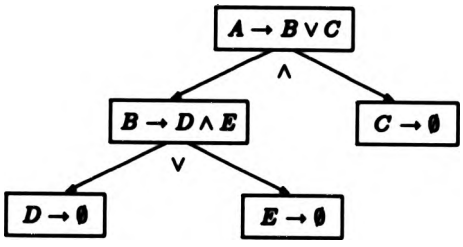
which enables the contradiction $C \rightarrow \emptyset$ to be propagated back through the chain of implications to give the required proof of $A \rightarrow \emptyset$:



In general, formulae may have logical structure. Consider, for instance, the implication

$$A \rightarrow B \vee C$$

In order to establish that the proposition $B \vee C$ is contradictory, it is necessary to analyse both disjuncts. A linear representation, therefore, is not appropriate. In order to represent logical conjunction and disjunction a tree structure is required. This can be seen in the following chain of implications:



This tree structure represents a proof of $A \rightarrow \emptyset$. The refutation algorithm presented here is based upon this notion of a tree of implications, where $\wedge$-nodes correspond to case analysis and $\vee$-nodes correspond to decomposition. A successful analysis is reflected in the construction of a tree structure which embodies a justification for the required negation. The final step in the proof process involves the extraction of the justification. This is achieved by propagating the negations held at the leaf nodes back up through the tree structure to obtain a proof of the original goal.

5.2 Deriving properties of data types

As noted by Backhouse[Backhouse 86b], Martin-Löf's theory betrays a rich structure. Backhouse proposes a scheme for introducing user-defined extensions to the theory which exploits this rich structure. The scheme is based on a mechanism for computing elimination and computation rules automatically, given the formation and introduction rules for an arbitrary type constructor. Here we are concerned with using the structure of the theory to derive properties of data type constructors. In particular, we present methods for generating uniqueness, universal closure and cancellation properties in sections 5.2.1, 5.2.2 and 5.2.3 respectively. Although of general use, this work was motivated by the development of the decision method for negation. Uniqueness properties are necessary for formalising direct contradictions. As mentioned earlier, our decision method is based upon derived rules for case analysis, evaluation and decomposition, within the context of the equality type. In section 5.2.4 we present methods for deriving these rules for arbitrary data type constructors. The method for constructing a case analysis rule assumes the existence of closure properties, while cancellation properties provide the basis for the construction of decomposition rules.

5.2.1 Uniqueness properties

For an arbitrary data type constructor Θ, there exist uniqueness properties which show that expressions, formed using the constructors defined by the Θ introduction rules, are distinct if the constructors are distinct. Consider, for instance, type N which has two introduction rules defining the constructors 0 and $'$. As-

sociated with type N is a single uniqueness property expressed by the type

$$(\Pi x : N)\neg Eq(N, x', 0)$$

An object in this type can be constructed according to the following derivation:

Derivation

0.0 $\quad [\![x : N$

0.1.0 $\quad \triangleright \quad [\![y : Eq(N, x', 0)$

0.1.1.0 $\quad \triangleright \quad [\![u : N; v : U_1$

$\quad \triangleright \quad \{U_1\text{-intr}_{Unit}\}$

0.1.1.1 $\quad Unit : U_1$

$\quad]\!]$

$\quad \{U_1\text{-intr}_{\emptyset}, 0.1.1\ N\text{-comp}_0\}$

0.1.2 $\quad nrec(0, \emptyset, [u, v]\, Unit) = \emptyset : U_1$

0.1.3.0 $\quad [\![u : N; v : U_1$

$\quad \triangleright \quad \{U_1\text{-intr}_{Unit}\}$

0.1.3.1 $\quad Unit : U_1$

$\quad]\!]$

$\quad \{0.0, U_1\text{-intr}_{\emptyset}, 0.1.3\ N\text{-comp}'\}$

0.1.4 $\quad nrec(x', \emptyset, [u, v]\, Unit) = Unit : U_1$

$\quad \{0.1.4\ \text{sym}\}$

0.1.5 $\quad Unit = nrec(x', \emptyset, [u, v]\, Unit) : U_1$

$\quad \{0.1.0\ Eq\text{-elim}\}$

0.1.6 $\quad x' = 0 : N$

0.1.7.0 $\quad [\![w : N$

$\quad \triangleright \quad \{0.1.7.0\ \text{reflex}\}$

0.1.7.1 $\quad w = w : N$

$\quad \{U_1\text{-intr}_{\emptyset}\ \text{reflex}\}$

0.1.7.2 $\quad \emptyset = \emptyset : U_1$

0.1.7.3.0 $\quad [\![u : N; v : U_1$

$\quad \triangleright \quad \{U_1\text{-intr}_{Unit}\ \text{reflex}\}$

0.1.7.3.1 $\quad Unit = Unit : U_1$

$\quad]\!]$

$\quad \{0.1.7.1, 0.1.7.2, 0.1.7.3\ N\text{-elim}\}$

0.1.7.4 $\quad nrec(w, \emptyset, [u, v]\, Unit) = nrec(w, \emptyset, [u, v]\, Unit) : U_1$

$\quad]\!]$

$\quad \{0.1.6, 0.1.7.4\ \text{subst}\}$

0.1.8 $\quad nrec(x', \emptyset, [u, v]\, Unit) = nrec(0, \emptyset, [u, v]\, Unit) : U_1$

$\quad \{0.1.5, 0.1.8\ \text{trans}\}$

0.1.9 $\quad Unit = nrec(0, \emptyset, [u, v]\, Unit) : U_1$

$\quad \{0.1.9, 0.1.2\ \text{trans}\}$

0.1.10 $\quad Unit = \emptyset : U_1$

$\quad \{0.1.10\ U_1\text{-elim}\}$

0.1.11 $\quad Unit = \emptyset$

$\{Unit\text{-intr}_{un}, 0.1.11\ \text{Eqtype}\}$

0.1.12 $un : \mathbb{1}$

$\|$

$\{0.1.0, 0.1.12 \rightarrow\text{-intr}\}$

0.2 $\lambda y.un : Eq(N, x', 0) \rightarrow \mathbb{1}$

$\|$

$\{0.0, 0.2\ \Pi\text{-intr}\}$

1 $\lambda x.\lambda y.un : (\Pi x : N)(Eq(N, x', 0) \rightarrow \mathbb{1})$

Note that the *Unit* type, with single element un, may be defined in terms of the finite types. The structure of this derivation is completely determined by the rules defining N. In the next section we demonstrate that the uniqueness properties associated with an arbitrary type constructor Θ may be generated mechanically.

Deriving uniqueness properties in general

Let Θ denote an arbitrary type constructor, with k introduction rules defining canonical constructors $\theta_1, \ldots, \theta_k$. For each pair of constructors θ_i and θ_j, where $i \neq j$, there exists an uniqueness property defined by

$$(\Pi p_{i1} : B_{i1}) \ldots (\Pi p_{in_i} : B_{in_i})$$
$$(\Pi p_{j1} : B_{j1}) \ldots (\Pi p_{jn_j} : B_{jn_j})$$
$$\neg Eq(\Theta(\vec{A}), \theta_i(\vec{p_i}), \theta_j(\vec{p_j}))$$

Constructing an object in this type relies on the rules which define Θ. Working in a forwards direction, we begin by introducing an assumption of the form

$$p_{ir} : B_{ir}$$

where r ranges over the introduction variables defined by Θ-introduction$_{\theta_i}$. We shall refer to these assumptions collectively as the context S_1. Similarly, we

introduce a context S_2 for the introduction variables defined by Θ-introduction$_{\theta_j}$. Within these contexts we require an object in the function type

$$Eq(\Theta(\bar{\lambda}), \theta_i(\bar{p}_i), \theta_j(\bar{p}_j)) \rightarrow \emptyset \tag{5.3}$$

Proof is by contradiction and proceeds by assuming

$$z : Eq(\Theta(\bar{\lambda}), \theta_i(\bar{p}_i), \theta_j(\bar{p}_j)) \tag{5.4}$$

from which the equality judgement

$$\theta_i(\bar{p}_i) = \theta_j(\bar{p}_j) : \Theta(\bar{\lambda}) \tag{5.5}$$

follows by Eq-elimination. The contradiction which will allow us to establish that θ_i and θ_j are distinct is formally identified by deriving

$$Unit = \emptyset : U_1 \tag{5.6}$$

A method is required which maps 5.5 onto 5.6. Such a mapping is derived by an application of Θ-elimination:

$$\begin{array}{ll}
0 & z = \bar{z} : \Theta(\bar{\lambda}) \\
1 & \|C_1 \\
 & \triangleright \quad s_1(\bar{\delta}_1, \bar{w}_1) = \bar{s}_1(\bar{\delta}_1, \bar{w}_1) : D(\theta_1(\bar{\delta}_1)) \\
 & \| \\
 & \vdots \\
k & \|C_k \\
 & \triangleright \quad s_k(\bar{\delta}_k, \bar{w}_k) = \bar{s}_k(\bar{\delta}_k, \bar{w}_k) : D(\theta_k(\bar{\delta}_k)) \\
 & \| \\
 & \hline \\
 & \Theta rec(z, s_1, \ldots, s_k) = \Theta rec(\bar{z}, \bar{s}_1, \ldots, \bar{s}_k) : D(z)
\end{array} \quad \Theta\text{-elimination}$$

Premise 0 is established by a reflexive instance of the assumption

$$a : \Theta(\bar{\lambda}) \tag{5.7}$$

Premise i $(1 \leq i \leq k)$ is established within a context C_i. Construction of C_i is as follows. Firstly, introduce assumptions of the form

$$b_{ir} : B_{ir}$$

where r ranges over the introduction variables defined by Θ-introduction$_{\theta_i}$. Secondly, introduce assumptions of the form

$$w_{is} : U_1$$

where s ranges over the recursive introduction variables defined by Θ-introduction$_{\theta_i}$. Remembering that the objective is to derive 5.6, then premise i is completed by a reflexive instance of U_1-introduction$_\theta$

$$||C_i \rhd \theta = \theta : U_1||$$

and premise j is completed by a reflexive instance of U_1-introduction$_{Unit}$

$$||C_j \rhd \ Unit = Unit : U_1||$$

where C_j is constructed in a similar manner to C_i. Construction of the remaining $k-2$ premises is required only to ensure well-formedness, therefore, any object in U_1 will suffice. The application of Θ-elimination discharges contexts $C_1, \ldots, C_k$, thereby establishing

$$\Theta rec(a, z_1, \ldots, z_k) = \Theta rec(a, z_1, \ldots, z_k) : U_1$$

where z_i denotes the abstraction $[\delta_i, \varpi_i]\theta$ and z_j denotes the abstraction $[\delta_j, \varpi_j]\,Unit$. Taken in conjunction with 5.5, an application of the substitution rule discharges assumption 5.7, giving rise to the judgement

$$\Theta rec(\theta_i(\bar{p}_i), z_1, \ldots, z_k) = \Theta rec(\theta_j(\bar{p}_j), z_1, \ldots, z_k) : U_1$$

Evaluating both sides of this equality establishes 5.6. The required evaluation is achieved by transitivity and symmetry, given the equalities:

$$\Theta rec(\theta_i(\vec{p}_i), z_1, \ldots, z_h) = \emptyset : U_1 \tag{5.8}$$

$$\Theta rec(\theta_j(\vec{p}_j), z_1, \ldots, z_h) = \Theta(\vec{\lambda}) : U_1 \tag{5.9}$$

Both 5.8 and 5.9 are constructed by Θ-computation. Since un belongs to *Unit*, the required contradiction follows from 5.6 by an application of the type equality rule. An application of $\neg$-introduction discharges assumption 5.4, thereby establishing 5.3. Finally, by Π-introduction contexts S_1 and S_2 are discharged to give the required judgement:

$$\begin{array}{l}\lambda p_{i1}. \ldots \lambda p_{im_i}. \\ \quad \lambda p_{j1}. \ldots \lambda p_{jm_j}. \lambda x. un \\ \quad\quad : (\Pi p_{i1} : B_{i1}) \ldots (\Pi p_{im_i} : B_{im_i}) \\ \quad\quad\quad (\Pi p_{j1} : B_{j1}) \ldots (\Pi p_{jm_j} : B_{jm_j}) \\ \quad\quad\quad\quad \neg Eq(\Theta(\vec{\lambda}), \theta_i(\vec{p}_i), \theta_j(\vec{p}_j))\end{array}$$

5.2.2 Closure properties

For an arbitrary data type constructor Θ, there exists an universal closure property which expresses completely the structure of the canonical objects defined by the Θ introduction rules. An universal closure property can be expressed, in general, as a universally quantified disjunction, where each disjunct corresponds to a distinct canonical constructor. Consider, for instance, type N which has two introduction rules defining the object level constructors 0 and $'$. The universal closure property for type N is expressed by the type

$$(\Pi x : N)(Eq(N, 0, x) + (\Sigma n : N) Eq(N, n', x))$$

An object in this type can be constructed according to the following derivation:

Derivation

0.0 $\|[x : N$

$\triangleright$ {N-intr$_0$ reflex}

0.1 $0 = 0 : N$

{0.1 Eq-intr}

0.2 $e : Eq(N, 0, 0)$

{0.2 $+$-intr$_{inl}$}

0.3 $inl(e) : Eq(N, 0, 0) + (\Sigma n : N)Eq(N, n', 0)$

0.4.0 $\|[u : N; v : Eq(N, 0, u) + (\Sigma n : N)Eq(N, n', u)$

$\triangleright$ {0.4.0$_1$ reflex}

0.4.1 $u = u : N$

{0.4.1 N-intr$'$}

0.4.2 $u' = u' : N$

{0.4.2 Eq-intr}

0.4.3 $e : Eq(N, u', u')$

{0.4.0$_1$,0.4.3 Σ-intr}

0.4.4 $\langle u, e\rangle : (\Sigma n : N)Eq(N, n', u')$

{0.4.4 $+$-intr$_{inr}$}

0.4.5 $inr(\langle u, e\rangle) : Eq(N, 0, u') + (\Sigma n : N)Eq(N, n', u')$

$]|$

{0.0,0.3,0.4 N-elim}

0.5 $nrec(x, inl(e), [u, v]inr(\langle u, e\rangle))$

$: Eq(N, 0, x) + (\Sigma n : N)Eq(N, n', x)$

$]|$

{0.5 Π-intr}

1 $\lambda x.nrec(x, inl(e), [u, v]inr(\langle u, e\rangle))$

$: (\Pi x : N)(Eq(N, 0, x) + (\Sigma n : N)Eq(N, n', x))$

The structure of this derivation reflects the structure of the introduction and elimination rules associated with type N. In the next section we demonstrate that the derivation of a closure property exhibits a general structure. We show that given the rules defining an arbitrary type constructor Θ, the associated closure property may be generated mechanically.

Deriving closure properties in general

For the arbitrary type constructor Θ, there exists a closure property expressed by the type

$$(\Pi x : \Theta(\bar{A}))P(x)$$

where the family of types, denoted by P, is determined uniquely by the introduction rules associated with Θ. Assuming that Θ has k introduction rules then, in general, P takes the form of a disjunction with k disjuncts, where the i^{th} disjunct $(1 \leq i \leq k)$ expresses the structure of the canonical constructor defined by the i^{th} introduction rule. When $k = 1$, however, this general structure breaks down. We present the method of construction for the general case only, since the special case when $k = 1$ follows as a simplification.

Assuming $k > 1$ then the property we wish to prove is expressed by the type

$$(\Pi x : \Theta(\bar{A}))(P_1(x) + \cdots + P_k(x))$$

An object in this type can be constructed by an application of Π-introduction, given an object in the disjunction

$$P_1(x) + \cdots + P_k(x)$$

Since x denotes an arbitrary object in $\Theta(\bar{A})$, proof is by induction on x giving rise to k cases. If y_i $(1 \leq i \leq k)$ denotes an instance of the i^{th} canonical constructor, then a proof of the i^{th} case takes the form of an injection, where the injected value belongs to the type

$$P_i(y_i)$$

Denoting the i^{th} canonical constructor by θ_i, and assuming that θ_i is nullary, then $P_i(y_i)$ denotes the type

$$Eq(\Theta(\bar{\lambda}), \theta_i, \theta_i)$$

An object in this type is constructed using a reflexive instance of the Θ-introduction$_\theta$ rule composed with Eq-introduction. The resulting judgement takes the form

$$c : Eq(\Theta(\bar{\lambda}), \theta_i, \theta_i)$$

and corresponds to step 0.2 in the derivation of the closure property for type N presented earlier. Alternatively, θ_i may be a non-nullary constructor with associated introduction variables $b_{i1}, \ldots, b_{im_i}$. To maintain uniformity, $P_i(y_i)$ could be expressed in terms of the projections defined for Θ

$$Eq(\Theta(\bar{\lambda}), \theta_i(\Theta_{proj_{i1}}(\theta_i(\bar{b}_i)), \ldots, \Theta_{proj_{im_i}}(\theta_i(\bar{b}_i))), \theta_i(\bar{b}_i))$$

This approach is, however, problematic. Consider, the projections *fst* and *snd* associated with the type of pairs. The $\times$-elimination rule prescribes how to construct functions over objects in the pair type. The resulting functions are defined in terms of the *split* operator:

$$fst \equiv (x)split(x, [p, q]p)$$
$$snd \equiv (x)split(x, [p, q]q)$$

This approach is adequate for projections which are total functions. In general this is not the case. Consider, for instance, the projection *pred* associated with type N. *pred* is a partial function. It is not defined for 0. The N-elimination rule prescribes how to construct functions over type N. N-elimination only permits

the construction of total functions. Partial functions can be accommodated in two ways.

On the one hand, the range of the function can be extended to allow for inputs for which the function is not defined. This extension is achieved by the disjoint union type. In the case of *pred* the range becomes

$$N + Unit$$

where *Unit* is the type with singleton member *un*. An implementation of *pred*, which satisfies this extension, takes the form

$$pred \equiv [n]nrec(n, inr(un), [u, v]inl(u))$$

This definition, however, leads to a rather clumsy specification of the closure property since $pred(n)$ belongs to $N + Unit$ and not N.

Alternatively, the domain can be restricted to only those elements for which the function is defined. This restriction is achieved by the function and subset types. In the case of *pred* the specification becomes

$$\neg Eq(N, n, 0) \rightarrow \{x : N | Eq(N, x', n)\}$$

where n denotes an arbitrary object in N. A program satisfying this specification takes the form

$$pred \equiv [n]nrec(n, \lambda p.case(p[e]), [u, v]\lambda q.u)$$

Domain restriction, however, suffers from the problem encountered above, where the range type is extended using the disjoint union type.

An alternative to using partial functions is to introduce existential quantification. Adopting this approach means that the closure property is expressed by

the type

$$(\Pi x : N)(Eq(N, 0, x) + (\Sigma n : N)Eq(N, n', x))$$

Although this approach does not maintain the uniformity of the disjuncts, it does lead to a cleaner formulation of the closure. In general, therefore, if θ_i is non-nullary then $P_i(y_i)$ denotes the type

$$(\Sigma y_{i1} : B_{i1}) \ldots (\Sigma y_{in_i} : B_{in_i}) Eq(\Theta(\bar{\lambda}), \theta_i(y_{i1}, \ldots, y_{in_i}), \theta_i(b_{i1}, \ldots, b_{in_i}))$$

Construction of an object in this type takes place in a context C_i, which is generated as follows. Firstly, introduce assumptions of the form

$$b_{ir} : B_{ir}$$

where r ranges over the introduction variables associated with Θ-introduction$_{\theta_i}$. Secondly, introduce assumptions of the form

$$w_{is} : (\Sigma y_{i1} : B_{i1}) \ldots (\Sigma y_{in_i} : B_{in_i}) Eq(\Theta(\bar{\lambda}), \theta_i(y_{i1}, \ldots, y_{in_i}), b_{is})$$

where s ranges over the recursive introduction variables defined by Θ-introduction$_{\theta_i}$. These additional assumptions denote inductive hypotheses. From the context C_i, an equality of the form

$$e : Eq(\Theta(\bar{\lambda}), \theta_i(b_{i1}, \ldots, b_{in_i}), \theta_i(b_{i1}, \ldots, b_{in_i}))$$

is established by a reflexive instance of the Θ-introduction$_{\theta_i}$ rule composed with Eq-introduction. By introducing existential quantification for b_{ij} $(1 \leq j \leq n_i)$ on the *left-hand-side* of the equality, a judgement is constructed of the form

$$\begin{array}{l} \|[C_i \\ \triangleright \quad \langle b_{i1}, \ldots \langle b_{in_i}, e \rangle \ldots \rangle \\ \quad\; : (\Sigma y_{i1} : B_{i1}) \ldots (\Sigma y_{in_i} : B_{in_i}) Eq(\Theta(\bar{\lambda}), \theta_i(y_{i1}, \ldots, y_{in_i}), \theta_i(b_{i1}, \ldots, b_{in_i})) \\ ]\| \end{array}$$

This judgement corresponds to step 0.4.4 in the derivation of the closure property for type N presented earlier. By the appropriate applications of the $+$-introduction rule, an injection into the type

$$P_1(\theta_i(\delta_i)) + \cdots + P_k(\theta_i(\delta_i))$$

is obtained. Denoting the i^{th} injection by $s_i(\delta_i, w_i)$, then the process described above generates k judgements of the form

$$[\![C_i \triangleright\ s_i(\delta_i, w_i) : P_1(\theta_i(\delta_i)) + \cdots + P_k(\theta_i(\delta_i))]\!]$$

Assuming x is an arbitrary object in $\Theta(\lambda)$, then the property

$$P_1(x) + \cdots + P_k(x)$$

follows from the k judgements constructed above by an application of Θ-elimination:

$$\frac{\begin{array}{l} x : \Theta(\lambda) \\ [\![C_1 \\ \triangleright\ \ s_1(\delta_1, w_1) : D(\theta_1(\delta_1)) \\ ]\!] \\ \vdots \\ [\![C_k \\ \triangleright\ \ s_k(\delta_k, w_k) : D(\theta_k(\delta_k)) \\ ]\!] \end{array}}{\Theta rec(x, s_1, \ldots, s_k) : D(x)}\ \Theta\text{-elimination}$$

Finally, by Π-introduction the required closure property is derived:

$$\lambda x.\Theta rec(x, s_1, \ldots, s_k) : (\Pi x : \Theta(\lambda))P_1(x) + \cdots + P_k(x)$$

5.2.3 Cancellation properties

Cancellation properties express the fact that for two canonical objects to be equal implies that their component parts are also equal. Consider, for example,

type N which has the associated non-nullary constructor $'$ and the associated cancellation property expressed by the type

$$(\Pi x : N)(\Pi y : N)(Eq(N, x', y') \to Eq(N, x, y))$$

An object in this type can be constructed as follows:

Derivation

0.0 $||x : N$

0.1.0 $\triangleright$ $||y : N$

0.1.1.0 $\triangleright$ $||r : Eq(N, x', y')$

0.1.1.1.0 $\triangleright$ $||u : N; v : N$

0.1.1.1.1 $\triangleright$ $u : N$

$||$

{0.0, 0.0, 0.1.1.1 N-comp$'$}

0.1.1.2 $nrec(x', x, [u, v]u) = x : N$

{similarly}

0.1.1.3 $nrec(y', x, [u, v]u) = y : N$

0.1.1.4.0 $||w : N$

$\triangleright$ {0.1.1.4.0 reflex}

0.1.1.4.1 $w = w : N$

{0.0 reflex}

0.1.1.4.2 $x = x : N$

0.1.1.4.3.0 $||u : N; v : N$

$\triangleright$ {$0.1.1.4.3.0_1$ reflex}

0.1.1.4.3.1 $u = u : N$

$||$

{0.1.1.4.1, 0.1.1.4.2, 0.1.1.4.3 N-elim}

0.1.1.4.4 $nrec(w, x, [u, v]u) = nrec(w, x, [u, v]u) : N$

{0.1.1.0 Eq-elim}

0.1.1.4.5 $x' = y' : N$

$||$

{0.1.1.4.5, 0.1.1.4.4 subst}

0.1.1.5 $nrec(x', x, [u, v]u) = nrec(y', x, [u, v]u) : N$

{0.1.1.2 sym}

0.1.1.6 $x = nrec(x', x, [u, v]u) : N$

{0.1.1.6, 0.1.1.5 trans}

0.1.1.7 $x = nrec(y', x, [u, v]u) : N$

{0.1.1.7, 0.1.1.3 trans}

0.1.1.8 $x = y : N$

{0.1.1.8 Eq-intr}

0.1.1.9 $e : Eq(N, x, y)$

$||$

{0.1.1.0,0.1.1.9 $\to$-intr}

$$
\begin{array}{ll}
0.1.2 & \lambda r.e : Eq(N, x', y') \to Eq(N, x, y) \\
 & \quad \| \\
 & \{0.1.0, 0.1.2\ \Pi\text{-intr}\} \\
0.2 & \lambda y.\lambda r.e : (\Pi y : N)(Eq(N, x', y') \to Eq(N, x, y)) \\
 & \| \\
 & \{0.0, 0.2\ \Pi\text{-intr}\} \\
1 & \lambda x.\lambda y.\lambda r.e : (\Pi x : N)(\Pi y : N)(Eq(N, x', y') \to Eq(N, x, y))
\end{array}
$$

As was the case with uniqueness properties for type N, the structure of this derivation is completely determined by the rules defining type N. Generalising from this derivation, we obtain a method for mechanically generating cancellation properties for an arbitrary type constructor.

Deriving cancellation properties in general

Let Θ denote an arbitrary type constructor with k introduction rules defining canonical constructors $\theta_1, \ldots, \theta_k$. For θ_i $(1 \leq i \leq k)$ there exists n_i cancellation properties expressed by the type

$$
\begin{array}{l}
(\Pi p_{i1} : B_{i1}) \ldots (\Pi p_{in_i} : B_{in_i}) \\
\quad (\Pi q_{i1} : B_{i1}) \ldots (\Pi q_{in_i} : B_{in_i}) \\
\quad\quad Eq(\Theta(\vec{A}), \theta_i(\vec{p_i}), \theta_i(\vec{q_i})) \to Eq(B_{ij}, p_{ij}, q_{ij})
\end{array}
$$

where j ranges over the introduction variables defined by Θ-introduction$_{\theta_i}$. Constructing an object in this type is as follows. Working in a forwards direction, we begin by introducing n_i assumptions of the form

$$p_{ir} : B_{ir}$$

where r ranges over the introduction variables defined by Θ-introduction$_{\theta_i}$. We shall refer to these assumptions collectively as context S_1. Similarly, a context S_2 is constructed by introducing n_i assumptions of the form

$$q_{ir} : B_{ir}$$

Within contexts S_1 and S_2 an object is required in the function type

$$Eq(\Theta(\vec{\lambda}), \theta_i(\vec{p}_i), \theta_i(\vec{q}_i)) \rightarrow Eq(B_{ij}, p_{ij}, q_{ij}) \tag{5.10}$$

Proof proceeds by assuming

$$x : Eq(\Theta(\vec{\lambda}), \theta_i(\vec{p}_i), \theta_i(\vec{q}_i)) \tag{5.11}$$

from which the equality judgement

$$\theta_i(\vec{p}_i) = \theta_i(\vec{q}_i) : \Theta(\vec{\lambda}) \tag{5.12}$$

follows by Eq-elimination. In order to establish 5.10 we require a mapping from 5.12 onto the judgement

$$p_{ij} = q_{ij} : B_{ij} \tag{5.13}$$

Such a mapping is derived by an application of Θ-elimination:

$$\begin{array}{ll}
0 & x = \tilde{x} : \Theta(\vec{\lambda}) \\
1 & \|C_1 \\
 & \triangleright \quad z_1(\vec{b}_1, \vec{w}_1) = \tilde{z}_1(\vec{b}_1, \vec{w}_1) : D(\theta_1(\vec{b}_1)) \\
 & \| \\
 & \vdots \\
k & \|C_k \\
 & \triangleright \quad z_k(\vec{b}_k, \vec{w}_k) = \tilde{z}_k(\vec{b}_k, \vec{w}_k) : D(\theta_k(\vec{b}_k)) \\
 & \| \\
 & \hline \\
 & \Theta rec(x, z_1, \ldots, z_k) = \Theta rec(\tilde{x}, \tilde{z}_1, \ldots, \tilde{z}_k) : D(x)
\end{array} \quad \Theta\text{-elimination}$$

Premise 0 is established by a reflexive instance of the assumption

$$a : \Theta(\vec{\lambda}) \tag{5.14}$$

Premise i $(1 \leq i \leq k)$ is established within a context C_i. Construction of C_i is as follows. Firstly, introduce assumptions of the form

$$b_{ir} : B_{ir}$$

where r ranges over the introduction variables defined by Θ-introduction$_{\theta_i}$. Secondly, introduce assumptions of the form

$$w_{is} : B_{ij}$$

where s ranges over the recursive introduction variables defined by Θ-introduction$_{\theta_i}$. Remembering that the objective is to derive 5.13, then premise i is completed by a reflexive instance of the j^{th} assumption associated with the context C_i:

$$||C_i \triangleright\ b_{ij} = b_{ij} : B_{ij}||$$

Construction of the remaining $k-1$ premises is required only to ensure well-formedness. Therefore, any object in B_{ij} will do. The application of Θ-elimination discharges contexts $C_1, \ldots, C_k$, thereby establishing

$$\Theta rec(a, z_1, \ldots, z_k) = \Theta rec(a, z_1, \ldots, z_k) : B_{ij}$$

where z_i denotes the abstraction $[\vec{b_i}, \vec{w_i}]b_{ij}$. Taken in conjunction with 5.12, an application of the substitution rule discharges assumption 5.14, giving rise to the judgement

$$\Theta rec(\theta_i(\vec{p_i}), z_1, \ldots, z_k) = \Theta rec(\theta_i(\vec{q_i}), z_1, \ldots, z_k) : B_{ij}$$

Evaluating both sides of this equality judgement establishes 5.13. The required evaluation is achieved by transitivity and symmetry, given the equalities:

$$\Theta rec(\theta_i(\vec{p_i}), z_1, \ldots, z_k) = p_{ij} : B_{ij} \qquad (5.15)$$

$$\Theta rec(\theta_i(\vec{q_i}), z_1, \ldots, z_k) = q_{ij} : B_{ij} \qquad (5.16)$$

Both 5.15 and 5.16 are constructed by Θ-computation. 5.10 is established from 5.13 by an application of Eq-introduction followed by $\rightarrow$-introduction, which dis-

charges assumption 5.11. Finally, by Π-introduction the assumptions associated with contexts S_1 and S_2 are discharged to give the required judgement:

$$\begin{array}{l}\lambda p_{i1}.\ldots.\lambda p_{im_i}.\\ \quad\lambda q_{i1}.\ldots.\lambda q_{im_i}.\lambda x.e\\ \quad\quad : (\Pi p_{i1} : B_{i1})\ldots(\Pi p_{im_i} : B_{im_i})\\ \quad\quad\quad (\Pi q_{i1} : B_{i1})\ldots(\Pi q_{im_i} : B_{im_i})\\ \quad\quad\quad\quad Eq(\Theta(\lambda), \theta_i(\vec{p_i}), \theta_i(\vec{q_i})) \rightarrow Eq(B_{ij}, p_{ij}, q_{ij})\end{array}$$

5.2.4 Reasoning about the equality type

Non-trivial propositions are expressed in type theory using the equality type. In terms of program construction the equality type enables specifications to be expressed implicitly. Reasoning about equalities, therefore, is an important aspect of program construction in type theory. We present derived rules for reasoning about equalities. In particular, rules for case analysis, evaluation and decomposition are derived. We demonstrate that the structure of these derived rules generalises for arbitrary data type constructors.

Case analysis rules

We present a method for mechanically generating type theory rules which facilitate case analysis in the context of the equality type. Analysis by cases is expressed by the disjoint union type. For example, given the judgement

$$||x : N \triangleright\ r : Eq(T, \alpha(x), \beta)||$$

then analysis of x by cases is expressed by the type

$$Eq(T, \alpha(0), \beta) + (\Sigma n : N)Eq(T, \alpha(n'), \beta)$$

Constructing an object in this type relies on the universal closure property for type N, as presented in section 5.2.2. The details of the derivation are as follows:

Derivation

0 $x : N$

1 $r : Eq(T, \alpha(x), \beta)$
$\{N_{closure}, 0\ \Pi\text{-elim}\}$

2 $nrec(x, inl(e), [u, v]inr((u, e))) : Eq(N, 0, x) + (\Sigma n : N)Eq(N, n', x)$

3.0 $||y : Eq(N, x, 0)$
$\triangleright$ $\{0.3\ Eq\text{-elim}\}$

3.1 $x = 0 : N$

3.2.0 $||w : N$

3.2.1 $\triangleright$ $Eq(T, \alpha(w), \beta)\ type$
$\{3.2.1\ reflex\}$

3.2.2 $Eq(T, \alpha(w), \beta) = Eq(T, \alpha(w), \beta)$
$||$
$\{3.1, 3.2\ subst\}$

3.3 $Eq(T, \alpha(x), \beta) = Eq(T, \alpha(0), \beta)$
$\{1, 3.3\ Eqtype\}$

3.4 $r : Eq(T, \alpha(0), \beta)$
$\{3.4\ +\text{-intr}_{inl}\}$

3.5 $inl(r) : Eq(T, \alpha(0), \beta) + (\Sigma n : N)Eq(T, \alpha(n'), \beta)$
$||$

4.0 $||z : (\Sigma n : N)Eq(N, x, n')$
$\triangleright$ $\{4.0\ \Sigma\text{-elim}_{snd}\}$

4.1 $snd(z) : Eq(N, x, fst(z)')$
$\{4.1\ Eq\text{-elim}\}$

4.2 $x = fst(z)' : N$
$\{similar\ to\ steps\ 3.2, \ldots, 3.5\}$

4.3 $inr(\langle fst(z), r\rangle) : Eq(T, \alpha(0), \beta) + (\Sigma n : N)Eq(T, \alpha(n'), \beta)$
$||$
$\{2,3,4\ +\text{-elim}\}$

5 $when(nrec(x, inl(e), [u, v]inr((u, e))), [y]inl(r), [z]inr(\langle fst(z), r\rangle))$
$: Eq(T, \alpha(0), \beta) + (\Sigma n : N)Eq(T, \alpha(n'), \beta)$

By eliding the intermediate steps, a derived rule is obtained of the form

$$\frac{\begin{array}{l} x : N \\ ||w : N \\ \triangleright\ Eq(T, \alpha(w), \beta)\ type \\ || \\ r : Eq(T, \alpha(x), \beta) \end{array}}{ncase(x) : Eq(T, \alpha(0), \beta) + (\Sigma n : N)Eq(T, \alpha(n'), \beta)}\ N\text{-case}$$

We use *ncase* to abbreviate the proof object. Similarities exist with the derivation of the closure property for type N, in that its structure reflects the structure of the associated introduction and elimination rules. In the next section we demonstrate that given the formation and introduction rules for an arbitrary type constructor Θ, together with its closure property, then the $\Theta case$ rule can be derived mechanically.

Derivation of case analysis rules

We now present a generalisation of the example outlined above. The general form of the derived case analysis rule is

$$\begin{array}{ll} 0 & x : \Theta(\bar{A}) \\ 1 & [\![w : \Theta(\bar{A}) \\ & \quad \triangleright \ Eq(T, \alpha(w), \beta)\ type \\ &]\!] \\ 2 & r : Eq(T, \alpha(x), \beta) \\ \hline & \Theta case(x) : R_1 + \ldots + R_k \end{array} \quad \Theta case$$

Premise 0 specifies the case variable, while premise 1 postulates that $Eq(T, \alpha(w), \beta)$ is well-formed, where w ranges over objects in $\Theta(\bar{A})$. Premise 2 is the judgement within which the analysis is to be performed. We present the general scheme of construction working back from the required conclusion:

$$R_1 + \ldots + R_k$$

An object in this type corresponds to an injection. Consider the injection into the i^{th} disjunct. The injected value belongs to the type R_i, where the structure of R_i is determined by the i^{th} canonical constructor θ_i. If θ_i is a nullary constructor

then R_i denotes the type

$$Eq(T, \alpha(\theta_i), \beta)$$

Construction of an object in this type proceeds as follows. Firstly, we introduce an assumption of the form

$$y_i : Eq(\Theta(\bar{\lambda}), x, \theta_i)$$

Given the formation variables $A_1, \ldots, A_m$, then the structure of this assumption is completely determined by Θ-formation and Θ-introduction$_{\theta_i}$. By an application of Eq-elimination, a judgement is derived of the form

$$\begin{array}{l} [\![y_i : Eq(\Theta(\bar{\lambda}), x, \theta_i) \\ \triangleright \quad x = \theta_i : \Theta(\bar{\lambda}) \\ ]\!] \end{array}$$

Taken in conjunction with a reflexive instance of premise 1, a judgement of the form

$$\begin{array}{l} [\![y_i : Eq(\Theta(\bar{\lambda}), x, \theta_i) \\ \triangleright \quad Eq(T, \alpha(x), \beta) = Eq(T, \alpha(\theta_i), \beta) \\ ]\!] \end{array}$$

is achieved by substitution. Using premise 2, the required injected value is obtained by the rule for type equality:

$$\begin{array}{l} [\![y_i : Eq(\Theta(\bar{\lambda}), x, \theta_i) \\ \triangleright \quad r : Eq(T, \alpha(\theta_i), \beta) \\ ]\!] \end{array}$$

This judgement corresponds to step 3.4 of the derivation of N-case presented earlier. Alternatively, θ_i may be non-nullary with associated introduction variables $b_{i1}, \ldots, b_{im_i}$. Following the approach for deriving closure properties, R_i will denote the type

$$(\Sigma b_{i1} : B_{i1}) \ldots (\Sigma b_{im_i} : B_{im_i}) Eq(T, \alpha(\theta_i(b_{i1}, \ldots, b_{im})), \beta)$$

To construct an object in this type we introduce an assumption of the form

$$y_i : (\Sigma b_{i1} : B_{i1}) \ldots (\Sigma b_{in_i} : B_{in_i}) Eq(\Theta(\bar{\lambda}), x, \theta_i(b_{i1}, \ldots, b_{in}))$$

Stripping off the existential quantification gives rise to a judgement of the form

$$\begin{array}{l} ||y_i : (\Sigma b_{i1} : B_{i1}) \ldots (\Sigma b_{in_i} : B_{in_i}) Eq(\Theta(\bar{\lambda}), x, \theta_i(b_{i1}, \ldots, b_{in})) \\ \triangleright \quad x = \theta_i(fst(y_i), \ldots, fst(\ldots snd(y_i) \ldots)) : \Theta(\bar{\lambda}) \\ || \end{array}$$

Following the process described above for a nullary constructor, the following judgement is derived

$$\begin{array}{l} ||y_i : (\Sigma b_{i1} : B_{i1}) \ldots (\Sigma b_{in_i} : B_{in_i}) Eq(\Theta(\bar{\lambda}), x, \theta_i(b_{i1}, \ldots, b_{in_i})) \\ \triangleright \quad r : Eq(T, \alpha(\theta_i(fst(y_i), \ldots, fst(\ldots snd(y_i) \ldots))), \beta) \\ || \end{array}$$

Finally, by Σ-introduction the value of the required injection is derived:

$$\begin{array}{l} ||y_i : (\Sigma b_{i1} : B_{i1}) \ldots (\Sigma b_{in_i} : B_{in_i}) Eq(\Theta(\bar{\lambda}), x, \theta_i(b_{i1}, \ldots, b_{in_i})) \\ \triangleright \quad \langle fst(y_i), \langle \ldots \langle fst(\ldots snd(y_i) \ldots), r \rangle \ldots \rangle \rangle \\ \qquad\qquad : (\Sigma b_{i1} : B_{i1}) \ldots (\Sigma b_{in_i} : B_{in_i}) Eq(T, \alpha(\theta_i(b_{i1}, \ldots, b_{in_i})), \beta) \\ || \end{array}$$

From an object in R_i, the required injection is constructed by an application of $+$-introduction. Denoting the i^{th} injection by $w_i(y_i)$, and the type of y_i by $P_i(x)$, then the process described above generates k judgements of the form

$$||y_i : P_i(x) \triangleright \ w_i(y_i) : R_1 + \ldots + R_k||$$

In order to combine these k judgements, a method is required which yields an injection into

$$P_1(x) + \ldots + P_k(x)$$

given an arbitrary object x in $\Theta(\bar{\lambda})$. The universal closure property for Θ provides such a method and by Π-elimination this method is particularised to the

case variable specified by premise 0. Finally, by +-elimination$_k$ the required conclusion is established:

$$when_k(\Theta rec(x, z_1, \ldots, z_k), w_1, \ldots, w_k) : R_1 + \ldots + R_k$$

Evaluation rules

We now examine evaluation of object expressions in the context of the equality type. Again we begin with an example. Consider the judgement

$$r : Eq(T, \alpha(nrec(0, b, d)), \beta)$$

Evaluation corresponds to the construction of an object in the type

$$Eq(T, \alpha(b), \beta)$$

Constructing such an object relies on N-computation$_0$:

$$\frac{\begin{array}{l} b : D(0) \\ [u : N; v : D(u) \\ \triangleright \quad d(u, v) : D(u') \\ ] \end{array}}{nrec(0, b, d) = b : D(0)} \; N\text{-computation}_0$$

In the above example the expression to be evaluated occurs as a subexpression of α, so a strategy of evaluation from within is appropriate. Martin-Löf's computation rules are, however, formulated to evaluate from without. While the evaluation of the subexpression is achieved by an application of N-computation$_0$, a mechanism for isolating the subexpression is required. The rules for substitution and type equality provide such a mechanism. The details are as follows:

Derivation

0 $b : D(0)$

1.0 $[[a : N; v : D(a)$

1.1 $\triangleright$ $d(a, v) : D(a')$

$]]$

2 $r : Eq(T, \alpha(nrec(0, b, d)), \beta)$

3.0 $[[x : N; y : D(x)$

3.1 $\triangleright$ $Eq(T, \alpha(y), \beta)$ *type*

{3.1 reflex}

3.2 $Eq(T, \alpha(y), \beta) = Eq(T, \alpha(y), \beta)$

{0,1 N-comp$_0$}

3.3 $nrec(0, b, d) = b : D(0)$

$]]$

{3.3,3.2 subst}

4 $Eq(T, \alpha(nrec(0, b, d)), \beta) = Eq(T, \alpha(b), \beta)$

{2,4 Eqtype}

5 $r : Eq(T, \alpha(b), \beta)$

Steps 0 and 1 of this derivation correspond to the premises of N-computation$_0$, while step 2 denotes the initial equality. By eliding the intermediate steps in the derivation the required evaluation rule is obtained:

$$\dfrac{\begin{array}{l}[[x : N; y : D(x)\\ \triangleright\ Eq(T, \alpha(y), \beta)\ type\\ ]]\\ b : D(0)\\ [[a : N; v : D(a)\\ \triangleright\ d(a, v) : D(a')\\ ]]\\ r : Eq(T, \alpha(nrec(0, b, d)), \beta)\end{array}}{neval_0 : Eq(T, \alpha(b), \beta)}\ N\text{-eval}_0$$

where $neval_0$ is an abbreviation for the constructed proof object.

Derivation of evaluation rules

We now present a generalisation of the scheme outlined above for deriving evaluation rules in the context of the equality type. Let Θ denote an arbitrary type

constructor with k introduction rules defining canonical constructors $\theta_1, \ldots, \theta_k$. For each constructor there exists an associated evaluation rule, where the i^{th} $(1 \leq i \leq k)$ rule takes the form

$$
\begin{array}{ll}
0 & \|x : \Theta(\lambda); y : D(x) \\
 & \triangleright\ Eq(T, \alpha(y), \beta)\ type \\
 & \| \\
1 & b_{i1} : B_{i1} \ldots b_{im_i} : B_{im_i} \\
2 & \|C_1 \\
 & \triangleright\ s_1(a_1, \sigma_1, w_1) : D(\theta_1(a_1, \sigma_1)) \\
 & \| \\
 & \vdots \\
 & \|C_k \\
 & \triangleright\ s_k(a_k, \sigma_k, w_k) : D(\theta_k(a_k, \sigma_k)) \\
 & \| \\
3 & r : Eq(T, \alpha(\Theta rec(\theta_i(\vec{b}_i), s_1, \ldots, s_k)), \beta) \\
 & \hline \\
 & \Theta eval_{\theta_i} : Eq(T, \alpha(s_i), \beta)
\end{array}
\quad \Theta\text{-eval}_{\theta_i}
$$

The premises are divided into four parts. The first premise postulates that $Eq(T, \alpha(y), \beta)$ is a well-formed type for an arbitrary y in $D(x)$, where x is an element of $\Theta(\lambda)$. The second and third parts correspond to the premises of Θ-computation$_{\theta_i}$. The contexts $C_1, \ldots, C_k$ are constructed as described for the general computation rule schema presented in section 2.5. Premise 3 denotes the equality in which the evaluation is to be performed. The required conclusion is derived from these premises as follows: Firstly, by the application of Θ-computation$_{\theta_i}$ to parts two and three of the premises, the evaluation of the subexpression $\Theta rec(\theta_i(\vec{b}_i), s_1, \ldots, s_k)$ is achieved. The resulting judgement takes the form

$$\Theta rec(\theta_i(\vec{b}_i), s_1, \ldots, s_k) = s_i : D(\theta_i(\vec{b}_i)) \qquad (5.17)$$

Substituting 5.17 for y within a reflexive instance of premise 0 establishes

$$Eq(T, \alpha(\Theta rec(\theta_i(\vec{b}_i), s_1, \ldots, s_k)), \beta) = Eq(T, \alpha(s_i), \beta) \qquad (5.18)$$

From premise 3 and 5.18 the required conclusion is derived by an application of the type equality rule

$$r : Eq(T, \alpha(z_i), \beta)$$

Decomposition rules

Finally, we present a method for mechanically deriving type theory rules which facilitate decomposition in the context of the equality type. A decomposition, in general, is expressed through the cartesian product type and utilizes the cancellation properties discussed in section 5.2.3. We begin with an example. Consider the judgement

$$r : Eq(List(A), a :: b, c :: d)$$

Applying a decomposition to this judgement yields an object in the type

$$Eq(A, a, c) \times Eq(List(A), b, d)$$

The derivation of the decomposition rule is based upon the two cancellation properties associated with the *List* type. The first property is expressed by the type

$$\begin{array}{l}(\Pi a : A)(\Pi b : List(A)) \\ \quad (\Pi c : A)(\Pi d : List(A)) \\ \quad\quad Eq(List(A), a :: b, c :: d) \rightarrow Eq(A, a, c)\end{array}$$

while the second is expressed by

$$\begin{array}{l}(\Pi a : A)(\Pi b : List(A)) \\ \quad (\Pi c : A)(\Pi d : List(A)) \\ \quad\quad Eq(List(A), a :: b, c :: d) \rightarrow Eq(List(A), b, d)\end{array}$$

Abbreviating the respective judgements by $List\text{-cancel}_{hd}$ and $List\text{-cancel}_{tl}$, then the required decomposition is derived as follows:

0 $a : A$
1 $b : List(A)$
2 $c : A$
3 $d : List(A)$
4 $r : Eq(List(A), a :: b, c :: d)$
{$List$-cancel$_{a,d}$,0 Π-elim}
5 $hd[a] : (\Pi b : List(A))$
$(\Pi c : A)(\Pi d : List(A))$
$Eq(List(A), a :: b, c :: d) \rightarrow Eq(A, a, c)$
{similarly}
6 $hd[a][b][c][d][r] : Eq(A, a, c)$
{6 Eq-elim}
7 $a = c : A$
{7 Eq-intr}
8 $c : Eq(A, a, c)$
{similarly using $List$-cancel$_a$}
9 $c : Eq(List(A), b, d)$
{8,9 $\times$-intr}
10 $\langle c, c \rangle : Eq(A, a, c) \times Eq(List(A), b, d)$

By eliding the intermediate steps in the above derivation the required derived rule is obtained:

$$\frac{\begin{array}{l} a : A \\ b : List(A) \\ c : A \\ d : List(A) \\ r : Eq(List(A), a :: b, c :: d) \end{array}}{listdecomp : Eq(A, a, c) \times Eq(List(A), b, d)} \; List\text{-decomposition}$$

Derivation of decomposition rules

We now present a generalisation of the scheme outlined above. Given the cancellation properties associated with the arbitrary type constructor Θ, a decomposition rule can be derived of the form

$$\begin{array}{ll} 0 & b_{i1} : B_{i1} \\ & \vdots \\ & b_{in_i} : B_{in_i} \\ 1 & d_{i1} : B_{i1} \\ & \vdots \\ & d_{in_i} : B_{in_i} \\ 2 & r : Eq(\Theta(\vec{\lambda}), \theta_i(\vec{b_i}), \theta_i(\vec{d_i})) \end{array}$$

$$\frac{}{\Theta decomp : Eq(B_{i1}, b_{i1}, d_{i1}) \times \cdots \times Eq(B_{in_i}, b_{in_i}, d_{in_i})}\ \Theta\text{-decomposition}_{\theta_i}$$

The premises are divided into three parts. The first and second parts correspond to the premises of Θ-introduction$_{\theta_i}$. Premise 2 denotes the equality in which the decomposition is performed. As noted in section 5.2.3, if Θ-introduction$_{\theta_i}$ has n_i associated introduction variables, then there exists n_i cancellation properties. These cancellation properties, taken together with the premises of Θ-decomposition$_{\theta_i}$, give rise to n_i judgements of the form

$$e : Eq(B_{ij}, p_{ij}, q_{ij})$$

The required conclusion follows by an application of $\times$-introduction$_{n_i}$.

5.3 An algorithm for refutation

In this section we present the details of a refutation algorithm. The refutation algorithm incorporates an algorithm for analysing equalities and a framework for managing the search for a contradiction. A successful search is reflected in the generation of a tree of logical consequences which embodies the justification for the required negation. The analysis algorithm is based upon the methods described in section 5.2 for deriving properties of data types. The underlying idea behind the algorithm is that analysis of an equality at the level of the object

expressions may betray contradictions. The basis for this analysis is set out in section 5.3.1. The analysis algorithm is outlined in section 5.3.2, while the framework in which the analysis takes place is described in section 5.3.3. The extraction of the justification from the tree of implications constructed by the analysis process is given in section 5.3.4. In section 5.3.5 goal transformations are introduced which generalise the applicability of the refutation algorithm.

5.3.1 Analysis of the equality type

Section 5.2 described how certain general properties of data types may be derived mechanically. In particular, derived rules for case analysis, evaluation and decomposition were formulated. These rules provide the basis for the reductions and decompositions which underlie our analysis of the equality type.

Reduction

Reductions are applied to noncanonical terms. Two cases may arise. The first case is where part of a term is not fully evaluated. This may of course be the complete term. Consider, for instance, the following equality

$$Eq(N, plus(plus(0, n'), m), 0)$$

Here the subexpression $plus(0, n')$ is open to evaluation. The evaluation is achieved by an application N-eval, the evaluation rule for type N. This gives rise to the following equality

$$Eq(N, plus(n', m), 0)$$

The second case is where a noncanonical term occurs which is not fully saturated and, therefore, not open to evaluation. For instance, consider the following equality

$$Eq(N, plus(plus(x, n'), m), 0)$$

Here the occurrence of the variable x prevents reduction of the subexpression $plus(x, n')$ by direct evaluation. To establish that the equality is contradictory, it is necessary to show that the instantiation of x for each canonical form associated with type N gives rise to a contradiction. This is achieved by the application of N-case, the case analysis rule for type N, which gives rise to a disjunction of the form

$$Eq(N, plus(plus(0, n'), m), 0) + (\Sigma u : N)Eq(N, plus(plus(u', n'), m), 0)$$

Decomposition

A decomposition is possible when both sides of an equality are non-atomic canonical forms with matching outer structure. Consider, for example, the equality

$$Eq(N \times N, \langle a, b\rangle, \langle c, d\rangle)$$

By decomposition this equality is transformed into a conjunction of the form

$$Eq(N, a, c) \times Eq(N, b, d)$$

This decomposition is achieved by an application of $\times$-decomp.

```
analyse[t] =
    let ⟨T, α, β, C⟩ = extract from t
    in if noncanonical α ∧ saturated α  →  apply eval ⟨α, t, C⟩
       [] noncanonical α ∧ ¬saturated α →  apply case ⟨α, t, C⟩
       [] noncanonical β ∧ saturated β  →  apply eval ⟨β, t, C⟩
       [] noncanonical β ∧ ¬saturated β →  apply case ⟨β, t, C⟩
       [] canonical α ∧ canonical β     →  apply decomp ⟨T, t, C⟩
       fi
```

Figure 5.1: Outline of the *analyse* algorithm

5.3.2 Analysis algorithm

The application of reductions and decompositions is carried out by *analyse*, the analysis algorithm. An outline of the algorithm is given in figure 5.1.

An application of *analyse* operates on a theorem of the form

```
||C
▸ ||r : Eq(T, α, β)
  ▸ r : Eq(T, α, β)
  ||
||
```

This theorem is constructed from the initial goal type, which is of the negated equality form:

$$\neg Eq(T, \alpha, \beta)$$

The assumption, denoted by r, enables the required negation to be derived. This will be explained fully in section 5.3.3. Referring to this theorem as t, the analysis begins by extracting from t the quadruple

$$\langle T, \alpha, \beta, C \rangle$$

In a given situation, the selection of the appropriate analysis rule is based upon this extracted information. Analysis proceeds by checking whether α or β is non-

canonical. If either is noncanonical then a reduction is performed. As mentioned earlier, if a noncanonical form is not fully saturated, then case analysis is performed, otherwise a reduction is achieved directly by evaluation. For each data type there exists a set of derived analysis rules. In the case of the *left-hand-side* of an equality, the appropriate analysis rule is determined by the tuple

$$\langle \alpha, t, C \rangle$$

Firstly, the noncanonical form α is used to determine the data type upon which the evaluation or case analysis is to be performed. Secondly, t is required as a premise of the actual analysis rule. Finally C, the context, is used as a basis for constructing the additional premises required when applying an analysis rule. At any step in an analysis where a reduction is appropriate, there may exist a choice between which expression or subexpression is reduced. This choice is accommodated by considering each of the possible reductions. The details are given in section 5.3.3. If both α and β are non-atomic canonical forms, then a decomposition is performed. Again a particular derived rule is required. The appropriate decomposition is determined by the tuple

$$\langle T, t, C \rangle$$

where T, the base type of the equality, determines the type of the decomposition. Again t is required as a premise of the actual decomposition rule. The context C provides the basis for applying the appropriate cancellation properties.

5.3.3 Searching for refutation

The process of searching for refutation, which we shall refer to as *contra-chk*, builds upon the algorithm for analysing equalities presented in section 5.3.2. An application of *contra-chk* operates on a pair: an equality type and a context. A successful search generates a tree of logical implications which embodies a proof of the required negation.

Managing the analysis process

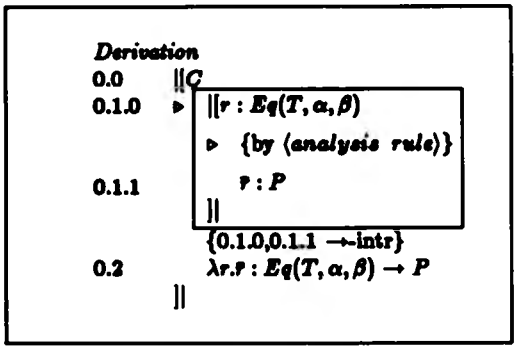
The search for a contradiction is recursive. An application of *contra-chk* invokes *analyse*. For the arbitrary equality $Eq(T, \alpha, \beta)$ and context C, *contra-chk* extends C by introducing an assumption for the given equality, which results in the construction of a derivation of the form:

Derivation
0.0 ||C
0.1.0 ▷ ||$r : Eq(T, \alpha, \beta)$
▷ $r : Eq(T, \alpha, \beta)$
||
||

As mentioned in section 5.3.2, this form of derivation provides the basis for an application of *analyse*, which in turn extends the derivation as follows:

Derivation
0.0 ||C
0.1.0 ▷ ||$r : Eq(T, \alpha, \beta)$
▷
⋮
{by ⟨*analysis rule*⟩}
0.1.1 $p : P$
||
||

The justification for a contradiction rests upon being able to construct a chain of logical implications based upon the initial equality which gives rise to a contradiction. To achieve this, *contra-chk* maintains a tree structure where each node denotes a derived implication constructed by an application of *analyse*. Consider again the derivation given above. This derivation corresponds to the result of an application of *analyse*. By discharging the initial assumption, the required implication is derived. Schematically an application of *contra-chk* corresponds to:



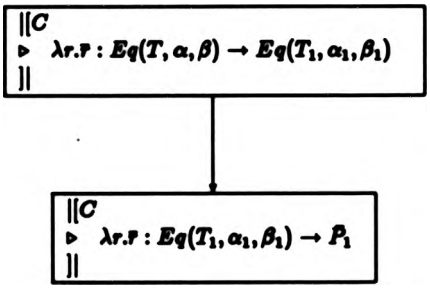
where the inner box represents the result of applying *analyse*. By eliding the intermediate step in the derivation, a node of the tree structure is established:

The search for a contradiction proceeds with P. Depending on the structure of P, the search may branch. P may correspond to one of four structures, each of

which requires different treatment. Firstly, P may take the form

$$Eq(T_1, \alpha_1, \beta_1)$$

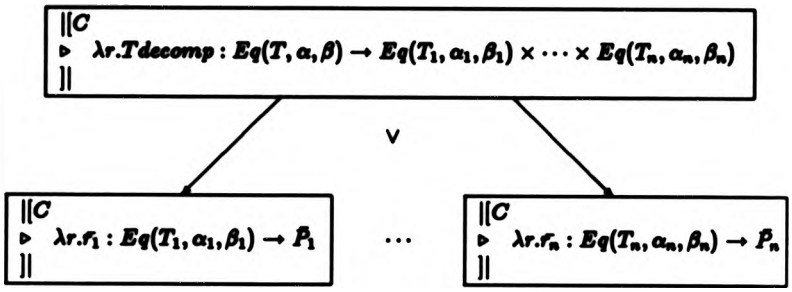
In such a situation the term corresponding to P is extracted from the derived implication and the *contra-chk* process is recursively applied using the initial context. Assuming that the analysis of $Eq(T_1, \alpha_1, \beta_1)$ gives rise to $\bar{P}_1$, then the analysis tree is extended as follows:



Secondly, P may take the form of a conjunction:

$$P_1 \times \cdots \times P_n$$

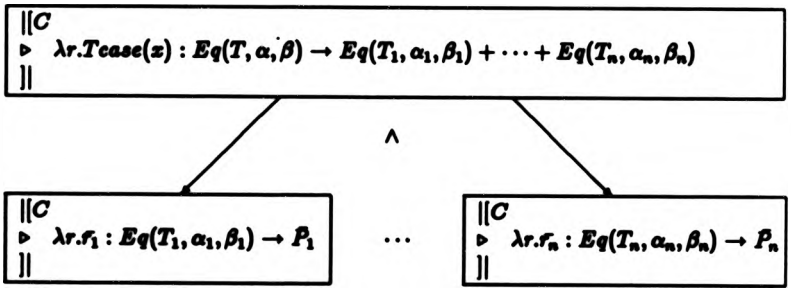
where each conjunct embodies an equality. A conjunction arises by decomposition. Assuming that P_i $(1 \leq i \leq n)$ is of the form $Eq(T_i, \alpha_i, \beta_i)$, and that the analysis of $Eq(T_i, \alpha_i, \beta_i)$ gives rise to $\bar{P}_i$, then the analysis tree is extended as follows:



In order for a contradiction to be established it is sufficient for only one conjunct to be shown to be contradictory. Alternatively, P may take the form of a disjunction:

$$P_1 + \cdots + P_n$$

where each disjunct embodies an equality. A disjunction arises from case analysis. In order for a contradiction to be established, each disjunct must be shown to be contradictory. Again assuming that P_i $(1 \leq i \leq n)$ is of the form $Eq(T_i, \alpha_i, \beta_i)$, and that the analysis of $Eq(T_i, \alpha_i, \beta_i)$ gives rise to P_i, then the analysis tree is extended as follows:



Note that either α or β may depend upon x. The analysis process may gener-

ate existentially quantified equalities. For instance, consider the analysis of the equality

$$Eq(T, f(x), g)$$

where x is of type $\Theta(\bar{\lambda})$. Analysis proceeds by cases on x. Assuming that the type constructor Θ has k associated object constructors $\theta_1, \ldots, \theta_k$, then the analysis will generate a disjunction with k disjuncts. The i^{th} $(1 \leq i \leq n)$ will take the general form

$$(\Sigma a_{i1} : B_{i1}) \ldots (\Sigma a_{in_i} : B_{in_i}) Eq(T, f(\theta_i(a_{i1}, \ldots, a_{in_i})), g)$$

The analysis of the i^{th} disjunct cannot be achieved directly because of the proof obligations introduced by the existential quantification. This problem is overcome by a *pre-analysis* step in which the existentially quantified equality is added to the context, allowing a judgement to be derived of the form

$$\begin{array}{l} \|C \\ \triangleright \quad \lambda v.snd(\ldots snd(v)\ldots) \\ \qquad : (\Pi v : (\Sigma a_{i1} : B_{i1}) \ldots (\Sigma a_{in_i} : B_{in_i}) Eq(T, f(\theta_i(a_{i1}, \ldots, a_{in_i})), g)) \\ \qquad\qquad\qquad Eq(T, f(\theta_i(fst(v), \ldots, fst(\ldots snd(v)\ldots))), g) \\ \| \end{array}$$

This dependent function removes the existential quantification. By extending the initial context with an assumption of the form

$$v : (\Sigma a_{i1} : B_{i1}) \cdots (\Sigma a_{in_i} : B_{in_i}) Eq(T, f(\theta_i(a_{i1}, \ldots, a_{in_i})), g)$$

the search for a contradiction proceeds by analysing

$$Eq(T, f(\theta_i(fst(v), \ldots, fst(\ldots snd(v)\ldots))), g)$$

The pre-analysis step is justified by the following derivation:

Derivation

0.0 $\|C$

0.1.0 $\triangleright$ $\|v : (\Sigma a_{i1} : B_{i1}) \ldots (\Sigma a_{in_i} : B_{in_i}) Eq(T, f(\theta_i(a_{i1}, \ldots, a_{in_i}), g)$

$\triangleright$ {0.1.0 Σ-elim$_{snd}$}

0.1.1 $snd(v) : (\Sigma a_{i2} : B_{i2}) \ldots (\Sigma a_{in_i} : B_{in_i}) Eq(T, f(\theta_i(fst(v), a_{i2}, \ldots, a_{in_i})), g)$

{similarly}

0.1.2 $snd(\ldots snd(v) \ldots) : Eq(T, f(\theta_i(fst(v), \ldots, fst(\ldots snd(v) \ldots))), g)$

$\|$

{0.1.0,0.1.2 Π-intro}

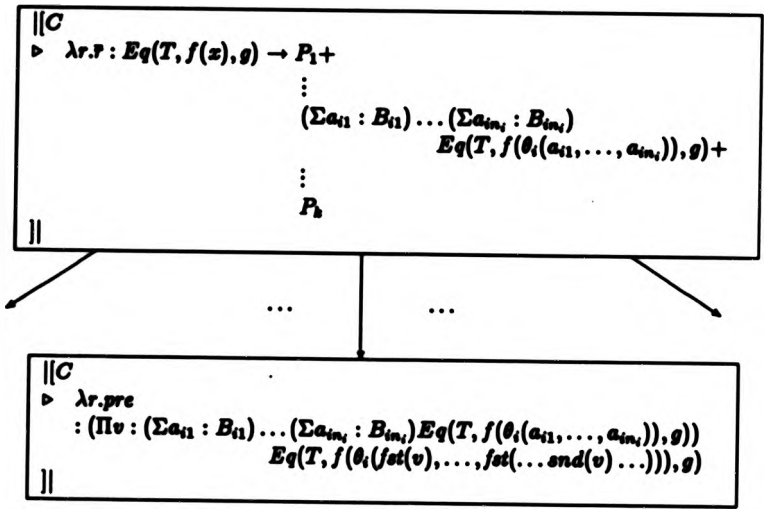
0.2 $\lambda v.snd(\ldots snd(v) \ldots)$

$: (\Pi v : (\Sigma a_{i1} : B_{i1}) \cdots (\Sigma a_{in_i} : B_{in_i}) Eq(T, f(\theta_i(a_{i1}, \ldots, a_{in_i})), g))$

$Eq(T, f(\theta_i(fst(v), \ldots, fst(\ldots snd(v) \ldots))), g)$

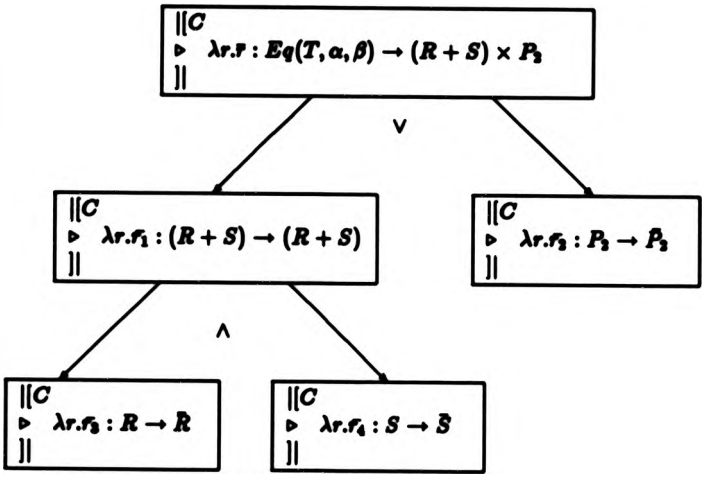
$\|$

The application of the pre-analysis step is reflected in the analysis tree by the introduction of an intermediate node:



As mentioned in section 5.3.2, a choice may exist in applying a reduction. The choice is accommodated within the framework by the introduction of a V-node as described above. However, if a disjunction arises then an additional intermediate

level in the tree structure is required. For example, if the equality $Eq(T, \alpha, \beta)$ reduces to P_1 and P_2, where P_1 denotes the disjunction $R+S$, then the associated analysis tree takes the form:



Termination

The *contra-chk* process may detect termination in two ways. Firstly, a contradiction may be identified. This corresponds to success. Secondly, an equality may be established, or alternatively, an equality may become too general to reason about. Either way this corresponds to to failure.

As noted earlier, a contradiction can arise either *directly* through the structure of the base type of an equality, or *indirectly* by means of the surrounding context. In the case of a *direct* contradiction, the basis of the proof is the uniqueness property associated with the base type of the equality. The basis of proof in the

case of an *indirect* contradiction is the surrounding context. A proof is established by a process of forward chaining from the assumptions.

In the case of failure two situations exist: we know an equality is valid or the equality has become too general. Firstly, consider the situation where an equality is valid. Such a situation directly mirrors that of a contradiction. An equality can arise either directly through the structure of the base type, or indirectly through the context. A direct equality corresponds to an equality between identical variables or canonical expressions. If the validity of an equality is implied by the context, then a proof is established by a process of forward chaining. The second situation in which failure occurs is where an equality becomes too general to reason about. Such a situation can arise if an equality between two unrelated variables, or a variable and a constant, is generated by the analysis process.

Worked example

We now present a worked example in order to illustrate the operation of *contra-chk*. The overview of the search for refutation, presented in section 5.1.2, includes an informal justification for the negation

$$\neg Eq(N, plus(m, n'), 0')$$

Given assumptions for m and n, the search for a contradiction proceeds as follows. The initial analysis performs a decomposition which generates a root node of the form:

$$
\begin{array}{l}
\|m : N;\ n : N \\
\triangleright\ \ \lambda r.ndecomp : Eq(N, plus(m, n')', 0') \\
\qquad\qquad\qquad\qquad \rightarrow Eq(N, plus(m, n'), 0) \\
\|
\end{array}
$$

Analysis proceeds by reduction of the *left-hand-side*. The required case analysis on m gives rise to a disjunction of the form

$$Eq(N, plus(0, n'), 0) + (\Sigma p : N)Eq(N, plus(p', n'), 0)$$

In order to establish that the initial equality is contradictory, both disjuncts must be shown to be contradictory. The search branches. The analysis of the *left-hand-branch* proceeds with the evaluation of $plus(0, n')$. The *right-hand-branch* involves existential quantification and is, therefore, not open to analysis directly. A pre-analysis step is required. The portion of the analysis tree corresponding to this branching takes the form:

$$
\begin{array}{l}
\|m : N; n : N \\
\triangleright\ \ \lambda r.ncase(m) : Eq(N, plus(m, n'), 0) \\
\qquad\qquad\qquad\qquad \rightarrow Eq(N, plus(0, n'), 0) + (\Sigma p : N)Eq(N, plus(p', n'), 0) \\
\|
\end{array}
$$

$$\wedge$$

$$
\begin{array}{l}
\|m : N; n : N \\
\triangleright\ \ \lambda r.neval \\
\qquad : Eq(N, plus(0, n'), 0) \\
\qquad\qquad\qquad \rightarrow Eq(N, n', 0) \\
\|
\end{array}
$$

$$
\begin{array}{l}
\|m : N; n : N \\
\triangleright\ \ \lambda r.pre \\
\qquad : (\Pi v : (\Sigma p : N)Eq(N, plus(p', n'), 0)) \\
\qquad\qquad\qquad Eq(N, plus(fst(v)', n'), 0) \\
\|
\end{array}
$$

Analysis of the *left-hand-branch* is complete. The required contradiction follows by the uniqueness property for type N. Similarly, by evaluation of $plus(fst(v)', n')$,

a contradiction is established in the *right-hand-branch*:

$$
\begin{array}{l}
||m : N; n : N \\
\triangleright \quad ||v : (\Sigma p : N)Eq(N, plus(p', n'), 0) \\
\qquad \triangleright \quad \lambda r.neval : Eq(N, plus(fst(v)', n'), 0) \rightarrow Eq(N, plus(fst(v), n')', 0) \\
\qquad || \\
||
\end{array}
$$

A proof of $Eq(N, plus(n, m')', 0') \rightarrow \emptyset$ is contained within the tree structure presented above. The extraction of the proof is dealt with in section 5.3.4.

5.3.4 Proof extraction

Given the arbitrary equality $Eq(T, \alpha, \beta)$ and the context C, a successful search for refutation generates a tree of implications which embodies a proof of

$$||C \triangleright \lambda x.x : Eq(T, \alpha, \beta) \rightarrow \emptyset||$$

In this section the process for extracting such a proof is described.

Propagation of contradictions

A successful search for a contradiction generates a tree in which every leaf directly descendent from a $\wedge$-node gives rise to a contradiction, and at least one leaf directly descendent from a $\vee$-node gives rise to a contradiction. The process of proof extraction involves propagating the contradictions held at these leaf nodes back up through the tree structure. Given the implication

$$Eq(T, \alpha, \beta) \rightarrow P$$

and assuming that by analysis of P a contradiction is identified

$$P \rightarrow \emptyset$$

then the required negation

$$Eq(T, \alpha, \beta) \to \emptyset$$

can be derived according to the following scheme:

Derivation

0.0 $\quad [\![f : A \to B$

0.1.0 $\quad \triangleright \quad [\![g : B \to \emptyset$

0.1.1.0 $\quad \triangleright \quad [\![a : A$

$\quad \triangleright \quad \{0.0, 0.1.1.0 \to\text{-elim}\}$

0.1.1.1 $\quad f[a] : B$

$\quad \{0.1.0, 0.1.1.1 \to\text{-elim}\}$

0.1.1.2 $\quad g[f[a]] : \emptyset$

$\quad]\!]$

$\quad \{0.1.1.0, 0.1.1.2 \to\text{-intr}\}$

0.1.2 $\quad \lambda a.g[f[a]] : A \to \emptyset$

$]\!]$

$]\!]$

By eliding the intermediate steps, a rule corresponding to a form of the *modus tollens* rule of inference is derived:

$$\frac{\begin{array}{l} A \text{ type} \\ f : A \to B \\ g : B \to \emptyset \end{array}}{\lambda x.x : A \to \emptyset}\ \text{mt}_1$$

Note that the derived proof object is replaced by the identity function $\lambda x.x$. If a negated type is non-empty, then the identity function will be present [Backhouse 86a]. P may also take the form of a conjunction or a disjunction. Firstly, if P takes the form of a conjunction

$$P_1 \times \cdots \times P_n$$

then an implication follows of the form

$$Eq(T, \alpha, \beta) \to P_1 \times \cdots \times P_n$$

In order to establish that $Eq(T, \alpha, \beta)$ is contradictory, it is sufficient to show that at least one of $P_1, \ldots, P_n$ is contradictory. Assuming that the analysis of P_i $(1 \leq i \leq n)$ gives rise to a contradiction

$$P_i \to \emptyset$$

then a proof of the negation

$$Eq(T, \alpha, \beta) \to \emptyset$$

can be derived according to the following scheme:

Derivation

0.0 $\quad [\![f : A \to B_1 \times \cdots \times B_n$
0.1.0 $\quad \triangleright \quad [\![g : B_i \to \emptyset$
0.1.1.0 $\quad \triangleright \quad [\![a : A$
$\quad \triangleright \quad \{0.0, 0.1.1.0 \to\text{-elim}\}$
0.1.1.1 $\quad f[a] : B_1 \times \cdots \times B_n$
0.1.1.2.0 $\quad [\![u_1 : B_1; \ldots; u_n : B_n$
0.1.1.2.1 $\quad \triangleright \quad u_i : B_i$
$\quad]\!]$
$\quad \{0.1.1.1, 0.1.1.2 \times\text{-elim}_n\}$
0.1.1.3 $\quad split_n(f[a], [u_1, \ldots, u_n]u_i) : B_i$
$\quad \{0.1.0, 0.1.1.3 \to\text{-elim}\}$
0.1.1.4 $\quad g[split_n(f[a], [u_1, \ldots, u_n]u_i)] : \emptyset$
$\quad]\!]$
$\quad \{0.1.1.0, 0.1.1.4 \to\text{-intr}\}$
0.1.2 $\quad \lambda a.(g[split_n(f[a], [u_1, \ldots, u_n]u_i)]) : A \to \emptyset$
$\quad]\!]$
$\quad]\!]$

Again by eliding the intermediate steps, a rule corresponding to a form of *modus tollens* is obtained:

$$\frac{\begin{array}{l} A \text{ type} \\ f : A \to B_1 \times \cdots \times B_n \\ g : B_i \to \emptyset \end{array}}{\lambda x.x : A \to \emptyset}\ \text{mt}_2$$

Alternatively, P may take the form of a disjunction

$$P_1 + \cdots + P_n$$

giving rise to an implication of the form

$$Eq(T, \alpha, \beta) \to P_1 + \ldots + P_n$$

Assuming that analysis of P_i $(1 \leq i \leq n)$ gives rise to n contradictions of the form

$$P_i \to \emptyset$$

then a proof of the negation

$$Eq(T, \alpha, \beta) \to \emptyset$$

can be derived according to the following scheme:

Derivation

0.0 $\quad [\![f : A \to B_1 + \cdots + B_n$

0.1.0 $\quad \triangleright \quad [\![g_1 : B_1 \to \emptyset; \ldots; g_n : B_n \to \emptyset$

0.1.1.0 $\quad \triangleright \quad [\![a : A$

$\quad \triangleright \quad \{0.0, 0.1.1.0 \to\text{-elim}\}$

0.1.1.1 $\quad f[a] : B_1 + \cdots + B_n$

0.1.1.2.0 $\quad [\![u_1 : B_1$

$\quad \triangleright \quad \{0.1.0_1, 0.1.1.2.0 \to\text{-elim}\}$

0.1.1.2.1 $\quad g_1[u_1] : \emptyset$

$\quad]\!]$

$\quad \{\text{similarly}\}$

0.1.1.3.0 $\quad [\![u_n : B_n$

$\quad \triangleright \quad \{0.1.0_n, 0.1.1.3.0 \to\text{-elim}\}$

0.1.1.3.1 $\quad g_n[u_n] : \emptyset$

$\quad]\!]$

$\quad \{0.1.1.1, 0.1.1.2, \ldots, 0.1.1.3 +\text{-elim}_n\}$

0.1.1.4 $\quad when_n(f[a], [u_1](g_1[u_1]), \ldots, [u_n](g_n[u_n])) : \emptyset$

$\quad]\!]$

$\quad \{0.1.1.0, 0.1.1.4 \to\text{-intr}\}$

0.1.2 $\quad \lambda a.when_n(f[a], [u_1](g_1[u_1]), \ldots, [u_n](g_n[u_n])) : A \to \emptyset$

$\quad]\!]$

$]\!]$

The corresponding derived rule takes the form

$$\frac{\begin{array}{l} A\ type \\ B_1\ type\ \ldots\ B_n\ type \\ f : A \to B_1 + \cdots + B_n \\ g_1 : B_1 \to \emptyset \\ \vdots \\ g_n : B_n \to \emptyset \end{array}}{\lambda x.x : A \to \emptyset}\ \mathrm{mt_e}$$

If P takes the form of a disjunction, then one or more of the disjuncts may involve existential quantification resulting in the application of a pre-analysis step. Our process for propagating contradictions must take account of this. As described earlier, if P denotes an instance of the Σ type, then the pre-analysis step constructs a dependent function type of the form

$$(\Pi u : P)P_1(u)$$

allowing the analysis to proceed with $P_1(v)$, where v belongs to P. Assuming that $P_1(v)$ gives rise to a contradiction, then a proof of $P \to \emptyset$ can be derived accordingly:

Derivation

0.0 $\|f : (\Pi u : P)P_1(u)$
0.1.0 ▸ $\|v : P$
0.1.1 ▸ $g : P_1(v) \to \emptyset$
$\|$
{0.1.0,0.1 Π-intr}
0.2 $\lambda v.g : (\Pi v : P)(P_1(v) \to \emptyset)$
0.3.0 $\|a : P$
▸ {0.2,0.3.0 Π-elim}
0.3.1 $(\lambda v.g)[a] : P_1(a) \to \emptyset$
{0.0,0.3.0 Π-elim}
0.3.2 $f[a] : P_1(a)$
{0.3.1,0.3.2 →-elim}
0.3.3 $((\lambda v.g)[a])[f[a]] : \emptyset$
$\|$

$\{0.3.0,0.3.3\ \rightarrow\text{-intr}\}$

0.4 $\lambda a.(((\lambda v.g)[a])[f[a]]) : P \rightarrow \emptyset$

$\|$

The corresponding derived rule takes the form

$$\frac{\begin{array}{l} P\ type \\ f : (\Pi u : P)P_1(u) \\ \|v : P \\ \triangleright\ \ g : P_1(v) \rightarrow \emptyset \\ \| \end{array}}{\lambda x.x : P \rightarrow \emptyset}\ \mathrm{mt}_4$$

Worked example revisited

In section 5.3.3 the contradictory equality

$$Eq(N, plus(m, n')', 0')$$

is used to illustrate the operation of *contra-chk*. We use the analysis tree generated in this example in order to demonstrate the application of the rules derived above for propagating contradictions. Starting with the *left-hand-branch*, from the derived implications

$$Eq(N, plus(0, n'), 0) \rightarrow Eq(N, n', 0) \qquad (5.19)$$

the negation of

$$Eq(N, plus(0, n'), 0)$$

is established by an application of mt_1 as follows:

Derivation

0.0 $\|n : N$

$\triangleright$ $\{N_{\mathrm{eml}_2}, 0.0\ \Pi\text{-elim}\}$

0.1 $\lambda x.x : Eq(N, n', 0) \rightarrow \emptyset$

$\{5.19, 0.1\ \mathrm{mt}_1\}$

0.2 $\lambda x.x : Eq(N, plus(0, n'), 0) \to \emptyset$

]|

Turning to the *right-hand-branch*, given the derived implications

$$Eq(N, plus(fst(v)', n'), 0) \to Eq(N, plus(fst(v), n')', 0) \tag{5.20}$$

$$(\Pi v : (\Sigma p : N)Eq(N, plus(p', n'), 0))Eq(N, plus(fst(v)', n'), 0) \tag{5.21}$$

the negation of

$$(\Sigma p : N)Eq(N, plus(fst(p)', n'), 0)$$

is established as follows:

Derivation

0.0 $\|m : N; n : N$

0.1.0 ▸ $\|v : (\Sigma p : N)Eq(N, plus(fst(p)', n'), 0)$

▸

⋮

0.1.1 $plus(fst(v), n') : N$

$\{N_{ent_2}, 0.1.1\ \Pi\text{-elim}\}$

0.1.2 $\lambda x.x : Eq(N, plus(fst(v), n')', 0) \to \emptyset$

$\{5.20, 0.1.2\ \text{mt}_3\}$

0.1.3 $\lambda x.x : Eq(N, plus(fst(v)', n'), 0) \to \emptyset$

]|

$\{5.21, 0.1.3\ \text{mt}_4\}$

0.2 $\lambda x.x : (\Sigma p : N)Eq(N, plus(p', n'), 0) \to \emptyset$

]|

Taken with the derived implication

$$\begin{aligned} &Eq(N, plus(m, n'), 0) \\ &\quad \to Eq(N, plus(0, n'), 0) + (\Sigma p : N)Eq(N, plus(fst(p)', n'), 0) \end{aligned} \tag{5.22}$$

the negations arising from both branches

$$Eq(N, plus(0, n'), 0) \to \emptyset \tag{5.23}$$

$$(\Sigma p : N)Eq(N, plus(fst(p)', n'), 0) \to \emptyset \tag{5.24}$$

are combined by an application of mt_6 to give the negation of

$$Eq(N, plus(m, n'), 0)$$

The details are as follows:

Derivation
0.0 $\|[m : N; n : N$
$\triangleright$ {5.22,5.23,5.24 mt_6}
0.1 $\lambda x.x : Eq(N, plus(m, n'), 0) \rightarrow \emptyset$
$]\|$

From the derived implication

$$Eq(N, plus(m, n')', 0') \rightarrow Eq(N, plus(m, n'), 0) \tag{5.25}$$

and the propagated contradiction

$$Eq(N, plus(m, n'), 0) \rightarrow \emptyset \tag{5.26}$$

the required negation

$$Eq(N, plus(m, n')', 0') \rightarrow \emptyset$$

is established by an application of mt_1:

Derivation
0.0 $\|[m : N; n : N$
$\triangleright$ {5.25,5.26 mt_1}
0.1 $\lambda x.x : Eq(N, plus(m, n')', 0') \rightarrow \emptyset$
$]\|$

5.3.5 Generalising the refutation algorithm

The algorithm for refutation described above operates at the level of equalities. In order to generalise the applicability of the algorithm, transformations are provided which convert a goal into the required negated equality form. These

transformations involve pushing negations through quantifiers and logicals. The transformations operate both at the level of the goal type and at level of the assumption types.

Goal type transformations

In general, a goal type G must be refined before *contra-chk* can be applied. This refinement process is carried out within the context of goal-directed proof and is achieved by the application of goal transformations. For each goal transformation there exists an associated validation; a derived rule of inference which ensures the soundness of the transformation. These derived rules correspond to a subset of DeMorgan's laws for the logicals and quantifiers. The rules are as follows:

$$\frac{\begin{array}{l} A\ type \\ [[w : A \\ \triangleright\ B(w)\ type \\ ]] \\ [[a : A \\ \triangleright\ b : B(a) \rightarrow \emptyset \\ ]] \end{array}}{\lambda x.x : ((\Sigma a : A)B(a)) \rightarrow \emptyset}\ \mathrm{bt}_1. \qquad \frac{\begin{array}{l} A\ type \\ a : A \\ [[w : A \\ \triangleright\ B(w) type \\ ]] \\ b : B(a) \rightarrow \emptyset \end{array}}{\lambda x.x : ((\Pi a : A)B(a)) \rightarrow \emptyset}\ \mathrm{bt}_2$$

$$\frac{\begin{array}{l} A\ type \\ B\ type \\ f : (A \rightarrow \emptyset) + (B \rightarrow \emptyset) \end{array}}{\lambda x.x : (A \times B) \rightarrow \emptyset}\ \mathrm{bt}_3 \qquad \frac{\begin{array}{l} A\ type \\ B\ type \\ f : A \rightarrow \emptyset \\ g : B \rightarrow \emptyset \end{array}}{\lambda x.x : (A + B) \rightarrow \emptyset}\ \mathrm{bt}_4$$

$$\frac{\begin{array}{l}a : A\\ f : B \to \emptyset\end{array}}{\lambda x.x : (A \to B) \to \emptyset}\ \mathrm{bt}_6$$

Assumption type transformations

Given a goal of the form

$$||a_1 : A_1; \ldots; a_n : A_n \rhd v : G||$$

then A_i $(1 \leq i \leq n)$ denotes an assumption type. Assumption type transformations represent forwards inference, and like the goal type transformations, correspond to a subset of DeMorgan's laws for the logicals and quantifiers. In general, however, the following implications are not constructively valid:

$$\neg(\Pi x : A)B(x) \to (\Sigma x : A)\neg B(x) \tag{5.27}$$

$$\neg(A \times B) \to (\neg A + \neg B) \tag{5.28}$$

Consequently, there exist only three forward transformations which correspond to the following derived rules:

$$\frac{\begin{array}{l}||x : ((\Sigma a : A)B(a)) \to \emptyset\\ \rhd\ \ v : G\\ ||\end{array}}{\begin{array}{l}||\ldots; z : (\Pi a : A)(B(a) \to \emptyset)\\ \rhd\ \ v : G\\ ||\end{array}}\ \mathrm{ft}_1 \qquad \frac{\begin{array}{l}||x : (A + B) \to \emptyset\\ \rhd\ \ v : G\\ ||\end{array}}{\begin{array}{l}||\ldots; z : (A \to \emptyset) \times (B \to \emptyset)\\ \rhd\ \ v : G\\ ||\end{array}}\ \mathrm{ft}_2$$

$$\frac{\begin{array}{l}\|s : (A \to B) \to \emptyset \\ \triangleright\ \ v : G \\ \|\end{array}}{\begin{array}{l}\|\ldots; z : B \to \emptyset \\ \triangleright\ \ v : G \\ \|\end{array}}\ \mathrm{ft}_3$$

The missing laws are not significant. For example, consider the following hypothetical judgement

$$\|f : (\Sigma a : A)\neg B(a) \triangleright\ v : G\|$$

Note that the assumption denoted by f corresponds to the consequent of 5.27. Such an assumption is too weak to be useful. It relates to a particular object v, for which $B(v)$ is contradictory, without specifying which object it is. Similarly, consider the hypothetical judgement

$$\|f : \neg A + \neg B \triangleright\ v : G\|$$

where the assumption denoted by f corresponds to the consequent of 5.28. Again this assumption is too weak to be useful. It does not provide a method for determining which disjunct holds. Consequently, the missing two laws are not significant as their presence would not enhance the generality of the decision method.

5.4 Implementation

The decision method presented above has been implemented in ML and incorporated into TTPA. As described in section 5.3.5, transformations operate in the

context of backwards proof, while the formal identification of contradictions at the level of the equality type is conducted in the context of forwards proof. This distinction is reflected in the implementation. On the one hand, the transformations are implemented as derived LCF style tactics. On the other hand, the analysis process is formalised using derived rules of inference. The transformation tactics are packaged up as two proof editor commands implemented by the ML functions

```
trans_goal_typ_strat : * -> .
trans_assl_type_strat : * -> .
```

The former deals with the transformation of goal types and the latter deals with assumption types. Both functions, as side effects, operate upon the TTPA proof state. The analysis algorithm is implemented by the ML function

```
analyse : thm -> thm
```

The search for a contradiction is managed by the ML function

```
contra_chk : (term # asslist) -> thm
```

The contra_chk function uses a depth first search strategy and incorporates the propagation of contradictions described in section 5.3.4. This part of the decision method is packaged up as a proof editor command implemented by the ML function

```
prove_neg_type : * -> .
```

The development of the decision method was motivated by the programming exercise documented in chapter 4, and in particular by the difficulties in proving

negations. We now consider the application of our decision method to the negated types arising from this exercise. Firstly, consider the negated type which we denoted as *absurdity*$_1$:

$$\neg Member(a, nil, A, B)$$

Unfolding the definition of *Member*, the corresponding proof editor session is initiated as follows:

```
# proofedit absi_goal absi_assl ();;
"v10
 : ->(Sigma(List(#(A,B)),
            (h)
             [Sigma(List(#(A,B)),
                    (t)
                     [Sigma(B,
                            (b)
                             [Eq(List(#(A,B)),
                                 append(h,cons(pair(a,b),t)),
                                 nil)])])]),
      {})
 [A : U1; B : U1; a : A]"
(goal)
Prooftree initialised.
() : .
```

On applying the goal transformations, fourteen subgoals are generated. All the subgoals, except subgoal 7, are well-formedness obligations:

```
# ebt trans_goal_typ_strat ();;
.
.
.
7
"v72 : ->(Eq(List(#(A,B)),append(h,cons(pair(a,b),t)),nil),{})
 [A : U1;
  B : U1;
  a : A;
  b10
  : Sigma(List(#(A,B)),
          (h)
           [Sigma(List(#(A,B)),
```

```
                    (t)
                     [Sigma(B,
                            (b)
                             [Eq(List(#(A,B)),
                                 append(h,cons(pair(a,b),t)),
                                 nil)])])]);
  h : List(#(A,B));
  b11
  : Sigma(List(#(A,B)),
          (t)
           [Sigma(B,(b)[Eq(List(#(A,B)),
                           append(h,cons(pair(a,b),t)),nil)])]);
  t : List(#(A,B));
  b12 : Sigma(B,(b)[Eq(List(#(A,B)),
                       append(h,cons(pair(a,b),t)),nil)]);
  b : B]"
(goal)
.
.
.
```

Subgoal 7 is of the required negated equality form to enable the application of the analysis algorithm:

```
# prove_neg_type ();;
7
"lambda((x)x)
 : ->(Eq(List(#(A,B)),append(h,cons(pair(a,b),t)),nil),{})
 [A : U1;
  B : U1;
  h : List(#(A,B));
  a : A;
  b : B;
  t : List(#(A,B));
  b12 : Sigma(B,(b)[Eq(List(#(A,B)),
                       append(h,cons(pair(a,b),t)),nil)]);
  b11
  : Sigma(List(#(A,B)),
          (t)
           [Sigma(B,(b)[Eq(List(#(A,B)),
                           append(h,cons(pair(a,b),t)),nil)])]);
  b10
  : Sigma(List(#(A,B)),
          (h)
           [Sigma(List(#(A,B)),
```

```
                    (t)
                     [Sigma(B,
                            (b)
                             [Eq(List(#(A,B)),
                                 append(h,cons(pair(a,b),t)),
                                 nil)])])])]"
(proved)() : .
```

Finally, the constructed justification is propagated back up through the goal tree converting it into a proof tree. The root node of the proof tree takes the following form:

```
# prove ();;
"lambda((x)x)
 : ->(Sigma(List(#(A,B)),
            (h)
             [Sigma(List(#(A,B)),
                    (t)
                     [Sigma(B,
                            (b)
                             [Eq(List(#(A,B)),
                                 append(h,cons(pair(a,b),t)),
                                 nil)])])]),
      {})
 [A : U1; B : U1; a : A]"
(proved)() : .
```

Next we consider *absurdity*$_2$, the second negation arising from the programming exercise:

$$\neg Member(a, cons(u10, u11), A, B)$$

Proof takes place in a context which includes the assumptions:

$$b11 : \neg Eq(A, a, fst(u10))$$
$$u14 : \neg Member(a, u11, A, B)$$

Again we unfold the definition of *Member*. The corresponding proof editor session is initiated as follows:

```
# proofedit abs2_goal abs2_ass1 ();;
"v10
 : ->(Sigma(List(#(A,B)),
            (h)
             [Sigma(List(#(A,B)),
                    (t)
                     [Sigma(B,
                            (b)
                             [Eq(List(#(A,B)),
                                 append(h,cons(pair(a,b),t)),
                                 cons(u10,u11))])])]),
      {})
 [A : U1;
  B : U1;
  a : A;
  u10 : #(A,B);
  u11 : List(#(A,B));
  b11 : ->(Eq(A,a,fst(u10)),{});
  u14
  : ->(Sigma(List(#(A,B)),
             (h)
              [Sigma(List(#(A,B)),
                     (t)
                      [Sigma(B,
                             (b)
                              [Eq(List(#(A,B)),
                                  append(h,cons(pair(a,b),t)),
                                  u11)])])]),
       {})]"
(goal)
Prooftree initialised.
() : .
```

Applying the goal type transformations gives rise to fourteen subgoals. Again we are only interested in subgoal 7:

```
7
"v72
 : ->(Eq(List(#(A,B)),append(h,cons(pair(a,b),t)),cons(u10,u11)),{})
 [A : U1;
  B : U1;
  a : A;
  u10 : #(A,B);
  u11 : List(#(A,B));
  b11 : ->(Eq(A,a,fst(u10)),{});
```

```
u14
: ->(Sigma(List(#(A,B)),
           (h)
            [Sigma(List(#(A,B)),
                   (t)
                    [Sigma(B,
                           (b)
                            [Eq(List(#(A,B)),
                                append(h,cons(pair(a,b),t)),
                                u11)])])]),
     {});
b12
: Sigma(List(#(A,B)),
        (h)
         [Sigma(List(#(A,B)),
                (t)
                 [Sigma(B,
                        (b)
                         [Eq(List(#(A,B)),
                             append(h,cons(pair(a,b),t)),
                             cons(u10,u11))])])]);
h : List(#(A,B));
b13
: Sigma(List(#(A,B)),
        (t)
         [Sigma(B,
                (b)
                 [Eq(List(#(A,B)),
                     append(h,cons(pair(a,b),t)),
                     cons(u10,u11))])]);
t : List(#(A,B));
b14
: Sigma(B,
        (b)[Eq(List(#(A,B)),
               append(h,cons(pair(a,b),t)),cons(u10,u11))]);
b : B]"
(goal)
```

We now apply the assumption type transformations:

```
# ebt trans_assl_type_strat ();;
.
.
.
7 10
"v103
```

```
: ->(Eq(List(#(A,B)),append(h,cons(pair(a,b),t)),cons(u10,u11)),{})
[A : U1;
 B : U1;
 a : A;
 u10 : #(A,B);
 u11 : List(#(A,B));
 b11 : ->(Eq(A,a,fst(u10)),{});
 u14
 : ->(Sigma(List(#(A,B)),
            (h)
             [Sigma(List(#(A,B)),
                    (t)
                     [Sigma(B,
                            (b)
                             [Eq(List(#(A,B)),
                                 append(h,cons(pair(a,b),t)),
                                 u11)])])]),
      {});
b12
: Sigma(List(#(A,B)),
         (h)
          [Sigma(List(#(A,B)),
                 (t)
                  [Sigma(B,
                         (b)
                          [Eq(List(#(A,B)),
                              append(h,cons(pair(a,b),t)),
                              cons(u10,u11))])])]);
h : List(#(A,B));
b13
: Sigma(List(#(A,B)),
         (t)
          [Sigma(B,
                 (b)
                  [Eq(List(#(A,B)),
                      append(h,cons(pair(a,b),t)),
                      cons(u10,u11))])]);
t : List(#(A,B));
b14
: Sigma(B,
         (b)[Eq(List(#(A,B)),
                append(h,cons(pair(a,b),t)),cons(u10,u11))]);
b : B;
b21
: Pi(List(#(A,B)),
     (h)
```

```
        [Pi(List(#(A,B)),
            (t)
             [Pi(B,
                 (b)
                  [->(Eq(List(#(A,B)),
                                append(h,cons(pair(a,b),t)),u11),{})])])])]"
(goal)
```

The goal and assumption types are now of the required form to enable an application of the analysis algorithm:

```
# prove_neg_type ();;
7 10
"lambda((x)x)
 : ->(Eq(List(#(A,B)),append(h,cons(pair(a,b),t)),cons(u10,u11)),{})
 [A : U1;
  B : U1;
  h : List(#(A,B));
  a : A;
  b : B;
  t : List(#(A,B));
  u10 : #(A,B);
  u11 : List(#(A,B));
  b21
  : Pi(List(#(A,B)),
       (h)
        [Pi(List(#(A,B)),
            (t)
             [Pi(B,
                 (b)
                  [->(Eq(List(#(A,B)),
                                append(h,cons(pair(a,b),t)),u11),{})])])]);`
  b14
  : Sigma(B,
          (b)[Eq(List(#(A,B)),
                       append(h,cons(pair(a,b),t)),cons(u10,u11))]);
  b13
  : Sigma(List(#(A,B)),
          (t)
           [Sigma(B,
                  (b)
                   [Eq(List(#(A,B)),
                       append(h,cons(pair(a,b),t)),
                       cons(u10,u11))])]);
  b12
```

```
  : Sigma(List(#(A,B)),
          (h)
           [Sigma(List(#(A,B)),
                  (t)
                   [Sigma(B,
                          (b)
                           [Eq(List(#(A,B)),
                               append(h,cons(pair(a,b),t)),
                               cons(u10,u11))])])]);
  u14
  : ->(Sigma(List(#(A,B)),
             (h)
              [Sigma(List(#(A,B)),
                     (t)
                      [Sigma(B,
                             (b)
                              [Eq(List(#(A,B)),
                                  append(h,cons(pair(a,b),t)),
                                  u11)])])]),
       {});
  b11 : ->(Eq(A,a,fst(u10)),{})]"
(proved)() : .
```

Finally, the constructed justification is propagated back up through the goal tree, converting it into a proof tree. The root node of the proof tree takes the following form:

```
# prove ();;
"lambda((x)x)
 : ->(Sigma(List(#(A,B)),
            (h)
             [Sigma(List(#(A,B)),
                    (t)
                     [Sigma(B,
                            (b)
                             [Eq(List(#(A,B)),
                                 append(h,cons(pair(a,b),t)),
                                 cons(u10,u11))])])]),
       {})
 [A : U1;
  B : U1;
  a : A;
  u10 : #(A,B);
  u11 : List(#(A,B));
```

```
u14
: ->(Sigma(List(@(A,B)),
           (h)
            [Sigma(List(@(A,B)),
                   (t)
                    [Sigma(B,
                           (b)
                            [Eq(List(@(A,B)),
                                append(h,cons(pair(a,b),t)),
                                u11)])])]),
     {});
 b11 : ->(Eq(A,a,fst(u10)),{})]"
(proved)() : .
```

5.5 Summary

In this chapter we have developed a decision method for negation. The method is based upon the derivation of data type properties and rules for analysing equalities. We have shown how the uniform structure of Martin-Löf's theory enables the required properties and analysis rules to be derived mechanically. The decision method has been incorporated into TTPA and successfully applied to the negations arising from the programming exercise documented in chapter 4. With completeness not an achievable goal, our criterion for evaluation must be based on this kind of empirical study. The decision method will be judged on its effectiveness as a programming tool. The work presented here represents a starting point; a framework and an algorithm which is both tunable and extendible.

Chapter 6

Primitive Recursive Definitions

Hoare emphasises the importance of abstraction in the design and development of computer programs[Hoare 72]. The need for abstraction is even more important within a framework which integrates the development and verification of programs. Definitions provide a limited, but useful, abstraction mechanism. Traditionally, definitions which have computational content are simply encoded directly at the object level. In the context of goal-directed proof an incorrectly encoded definition may go undetected, leading to a wasted proof attempt. We present a scheme for formally introducing definitions which satisfy the constraints of primitive recursion. A definition is specified as a type and an object level representation is derived automatically using the formal system. Exploiting the uniform structure of Martin-Löf's theory, we demonstrate that the scheme generalises for an arbitrary type constructor.

The structure of this chapter is as follows. The scheme is illustrated by way of an example in section 6.1. An specification schema for primitive recursive

definitions is presented in section 6.2. The general process for deriving an object level representation is described in section 6.3. Aspects of the implementation of the scheme are discussed in section 6.4.

6.1 An example

The scheme is best described by way of an example. Consider the following informal primitive recursive definition of the *length* function

$$\begin{cases} length(nil) = 0 \\ length(u :: v) = length(v)' \end{cases}$$

In type theory this definition of *length* is expressed in terms of *listrec*, the primitive recursive operator for lists:

$$length(l) \equiv listrec(l, 0, [u, v, w]w')$$

Defining *length* in this way corresponds, in effect, to writing a program. An alternative approach would be to specify the primitive recursive function as a type and use the formal system to construct an object in the type. It is this more formal approach which we have chosen. Assuming $List(A)$ to be a saturated type expression, then the primitive recursive definition of *length* is expressed by the type

(1) $$\begin{array}{l} (\Sigma f : List(A) \to N) \\ \quad Eq(N, f[nil], 0) \times \\ \quad (\Pi h : A) \\ \quad\quad (\Pi t : List(A)) \\ \quad\quad\quad Eq(N, f[h :: t], f[t]') \end{array}$$

An object in this type takes the form of a pair:

$$\langle \lambda l.listrec(l, 0, [u, v, w]w'), \langle e, \lambda h.\lambda t.e \rangle \rangle$$

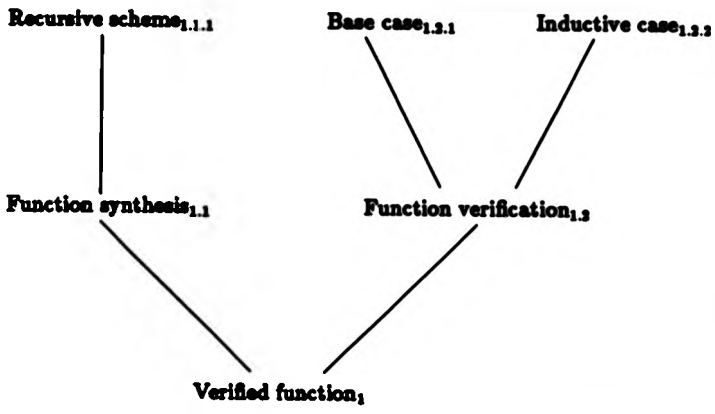


Figure 6.1: Schematic derivation of *length*

The first component, the existential witness, is a function. The second component demonstrates that the constructed function satisfies the recursion equations which define *length*. The task of constructing such an object can be viewed as two dependent tasks: function synthesis and function verification. This idea is expressed schematically in figure 6.1. The remainder of this section details the synthesis and verification of *length*. We begin with the synthesis of the *length* function, since the task of verification depends upon the existence of the function.

Synthesis of *length*

The close relationship between induction and recursion is reflected in the structure of Martin-Löf's elimination rules. Consider, for instance, the elimination rule for the *List* data type:

$$\frac{\begin{array}{l} l : \mathit{List}(A) \\ b : D(\mathit{nil}) \\ [\, u : A \\ ;\ v : \mathit{List}(A) \\ ;\ w : D(v) \\ \triangleright\ \ d(u, v, w) : D(u :: v) \\ ] \end{array}}{\mathit{listrec}(l, b, d) : D(l)}\ \mathit{List}\text{-elimination}$$

Taken at the level of the type expressions, this rule defines structural induction over the *List* data type. Alternatively, viewed at the object level, this rule enables us to define primitive recursive functions over lists. It is the association between the elimination rule and primitive recursion which we exploit here. In the case of *length*, inspection of the specification reveals that the required function object belongs to the type

$$\mathit{List}(A) \rightarrow N$$

This implies recursion over lists, which is achieved by an application of *List*-elimination. The first premise denotes the recursive argument and is established by introducing an assumption of the form

$$l : \mathit{List}(A) \qquad (6.1)$$

The second and third premises correspond to the base and the recursive steps respectively. The structure of each premise is uniquely determined by the types:

$$Eq(N, f[nil], 0) \tag{6.2}$$

$$Eq(N, f[h :: t], f[t]') \tag{6.3}$$

It follows from 6.2 that the base case is achieved by showing that 0 belongs to N. Similar inspection of 6.3 reveals that the recursive case is achieved by showing that w' belongs to N, where w denotes the inductive hypothesis, the value of f at t. The complete derivation is as follows:

Derivation of 1.1.1
0.0 $\quad ||l : List(A)$
0.1.0 $\quad \triangleright \quad ||u : A;\ v : List(A);\ w : N$
$\quad\quad \triangleright \quad \{0.1.0_3\ N\text{-intr}'\}$
0.1.1 $\quad\quad\quad w' : N$
$\quad\quad ||$
$\quad\quad \{0.0, N\text{-intr}_0, 0.1.1\ List\text{-elim}\}$
0.2 $\quad\quad listrec(l, 0, [u, v, w]w') : N$
$\quad ||$

By $\rightarrow$-introduction 6.1 is discharged, giving rise to the required function:

(1.1) $\qquad \lambda l.listrec(l, 0, [u, v, w]w') : List(A) \rightarrow N$

Verification of *length*

The result of the synthesis process described above is to instantiate f to

$$\lambda l.listrec(l, 0, [u, v, w]w')$$

By $\times$-introduction the task of verifying the correctness of this instantiation is reduced to constructing objects in the types:

(1.2.1) $Eq(N, \lambda l.listrec(l, 0, [u, v, w]w')[nil], 0)$

(1.2.2) $(\Pi\, h : A)$
$(\Pi\, t : List(A))$
$Eq(N, \lambda l.listrec(l, 0, [u, v, w]w')[h :: t],$
$\lambda l.listrec(l, 0, [u, v, w]w')[t]')$

Construction of an object in 1.2.1 is as follows:

Derivation of 1.2.1
$\{List\text{-intr}_{nil}$,derivation 1.1.1 $\rightarrow$-comp$\}$
0 $\lambda l.listrec(l, 0, [u, v, w]w')[nil] = listrec(nil, 0, [u, v, w]w') : N$
1.0 $||u : A;\ v : List(A);\ w : N$
$\triangleright$ $\{1.0_3\ N\text{-intr}'\}$
1.1 $w' : N$
$||$
$\{N\text{-intr}_0$,1.1 $List\text{-comp}_{nil}\}$
2 $listrec(nil, 0, [u, v, w]w') = 0 : N$
{0,2 trans}
3 $\lambda l.listrec(l, 0, [u, v, w]w')[nil] = 0 : N$
{3 Eq-intr}
4 $e : Eq(N, \lambda l.listrec(l, 0, [u, v, w]w')[nil], 0)$

while an object in 1.2.2 is constructed according to the following derivation:

Derivation of 1.2.2
0.0 $||h : A$
0.1.0 $\triangleright$ $||t : List(A)$
$\triangleright$ $\{0.0,0.1.0\ List\text{-intr}_{::}\}$
0.1.1 $h :: t : List(A)$
{0.1.1,derivation 1.1.1 $\rightarrow$-comp}
0.1.2 $\lambda l.listrec(l, 0, [u, v, w]w')[h :: t] = listrec(h :: t, 0, [u, v, w]w') : N$
0.1.3.0 $||u : A;\ v : List(A);\ w : N$
$\triangleright$ $\{0.1.3.0_3\ N\text{-inr}'\}$
0.1.3.1 $w' : N$
$||$
$\{0.0,0.1.0,N\text{-intr}_0,0.1.3.1\ List\text{-comp}_{::}\}$
0.1.4 $listrec(h :: t, 0, [u, v, w]w') = listrec(t, 0, [u, v, w]w')' : N$
{0.1.0,derivation 1.1.1 $\rightarrow$-comp}
0.1.5 $\lambda l.listrec(l, 0, [u, v, w]w')[t] = listrec(t, 0, [u, v, w]w') : N$
{0.1.5 N-intr$'$}
0.1.6 $\lambda l.listrec(l, 0, [u, v, w]w')[t]' = listrec(t, 0, [u, v, w]w')' : N$
{0.1.2,0.1.4 trans}
0.1.7 $\lambda l.listrec(l, 0, [u, v, w]w')[h :: t] = listrec(t, 0, [u, v, w]w')' : N$
{0.1.6 sym}

0.1.8 $listrec(t, 0, [u, v, w]w')' = \lambda l.listrec(l, 0, [u, v, w]w')[t]' : N$
{0.1.7,0.1.8 trans}

0.1.9 $\lambda l.listrec(l, 0, [u, v, w]w')[h :: t] = \lambda l.listrec(l, 0, [u, v, w]w')[t]' : N$
{0.1.9 *Eq*-intr}

0.1.10 $e : Eq(N, \lambda l.listrec(l, 0, [u, v, w]w')[h :: t],$
$\lambda l.listrec(l, 0, [u, v, w]w')[t]')$

||

{0.1.0,0.1.10 Π-intr}

0.2 $\lambda t.e$
$: (\Pi t : List(A))$
$Eq(N, \lambda l.listrec(l, 0, [u, v, w]w')[h :: t],$
$\lambda l.listrec(l, 0, [u, v, w]w')[t]')$

||

{0.0,0.2 Π-intr}

1 $\lambda h.\lambda t.e$
$: (\Pi h : A)$
$(\Pi t : List(A))$
$Eq(N, \lambda l.listrec(l, 0, [u, v, w]w')[h :: t],$
$\lambda l.listrec(l, 0, [u, v, w]w')[t]')$

Both derivations are based upon the recursion scheme specified for *length* and a strategy of evaluation which is achieved by the computation rules for the function and *List* types.

Finally, the synthesis and verification of *length* are combined by an application of Σ-introduction:

(1) $\langle \lambda l.listrec(l, 0, [u, v, w]w'), \langle e, \lambda h.\lambda t.e \rangle \rangle$
$: (\Sigma f : List(A) \rightarrow N)$
$Eq(N, f[nil], 0) \times$
$(\Pi h : A)$
$(\Pi t : List(A))$
$Eq(N, f[h :: t], f[t]')$

6.2 Specification schema

The informal definition of *length* presented earlier is a particular instance of a general primitive recursive scheme for lists:

$$\begin{cases} f(nil, c) = b(c) \\ f(u :: v, c) = d(u, v, c, f(v, c)) \end{cases}$$

The provision for an additional argument c of type C makes the scheme more general than is required to specify the *length* function. We shall build this generality into our specification schema. By exploiting the uniform structure of Martin-Löf's theory, it is possible to define a schema for specifying primitive recursive functions over an arbitrary data type. The details are as follows: Let $\Theta(\vec{A})$ denote a saturated expression, where the arbitrary type constructor Θ has k introduction rules defining canonical constructors $\theta_1, \ldots, \theta_k$. A primitive recursive function f belonging to the type

$$(\Pi\, x : \Theta(\vec{A}))C \to D(x)$$

is specifiable by the type schema

$$\begin{array}{l} (\Sigma f : (\Pi\, x : \Theta(\vec{A}))C \to D(x)) \\ \quad (\Pi c : C) \\ \quad\quad P_1(f, c) \times \cdots \times P_k(f, c) \end{array}$$

where C denotes the type of the arbitrary argument c. The structure of $P_i(f, c)$ $(1 \leq i \leq k)$ is determined by the constructor introduced by Θ-introduction$_{\theta_i}$. If θ_i is a nullary constructor, and $s_i(c)$ is the value of f applied to θ_i and c, then $P_i(f, c)$ denotes the type

$$Eq(D(\theta_i), f[\theta_i][c], s_i(c))$$

Alternatively, if θ_i is a non-nullary object constructor, then $P_i(f,c)$ denotes the type

$$(\Pi u_{i1} : A_{i1}) \ldots (\Pi u_{ip_i} : A_{ip_i})$$
$$(\Pi v_{i1} : \Theta(\vec{\lambda})) \ldots (\Pi v_{iq_i} : \Theta(\vec{\lambda}))$$
$$Eq(D(\theta_i(\vec{a}_i, \vec{a}_i)), f[\theta_i(\vec{a}_i, \vec{a}_i)][c], z_i(\vec{a}_i, \vec{a}_i, c, f[v_{i1}][c], \ldots, f[v_{iq_i}][c]))$$

where $u_{i1}, \ldots, u_{ip_i}$ and $v_{i1}, \ldots, v_{iq_i}$ denote the non-recursive and recursive introduction variables associated with θ_i respectively.

6.3 Method of construction

In this section we describe a method for satisfying the specification schema presented in section 6.2:

$$(\Sigma f : (\Pi x : \Theta(\vec{\lambda}))C \to D(x))$$
$$(\Pi c : C)$$
$$P_1(f,c) \times \cdots \times P_b(f,c)$$

The construction of an object in this type, as indicated in the derivation of the *length* function, decomposes into two dependent tasks: the synthesis of a function and its verification. The synthesis task corresponds to constructing an object in the type

(1.1) $\quad (\Pi x : \Theta(\vec{\lambda}))C \to D(x)$

Such an object takes the general form

$$\lambda a.\lambda y.\Theta rec(a, z_1(y), \ldots, z_b(y))$$

Verification is achieved by constructing an object in the type

(1.2) $\quad (\Pi c : C)$
$$P_1(\lambda a.\lambda y.\Theta rec(a, z_1(y), \ldots, z_b(y)), c) \times$$
$$\vdots$$
$$P_b(\lambda a.\lambda y.\Theta rec(a, z_1(y), \ldots, z_b(y)), c)$$

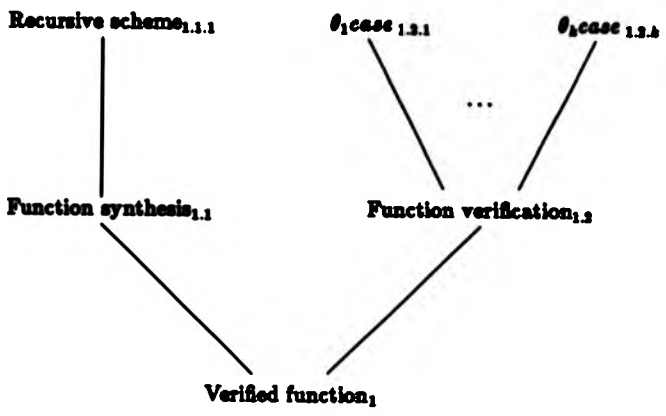


Figure 6.2: Generalised schematic derivation

The general form of such an object is

$$\lambda c.\langle p_1\langle \ldots \langle p_{k-1}, p_k\rangle\rangle\rangle$$

where p_i $(1 \leq i \leq k)$ belongs to the type

$$P_i(\lambda a.\lambda y.\Theta rec(a, s_1(y), \ldots, s_k(y)), c)$$

As demonstrated in section 6.1, the results of the synthesis and verification tasks are combined by an application of Σ-introduction. A schematic view of this general process is presented in figure 6.2.

6.3.1 Function synthesis

The task of function synthesis corresponds to constructing an object in the type

$$(\Pi\, x : \Theta(\bar{\lambda}))C \rightarrow D(x)$$

By convention, $\Theta(\lambda)$ denotes the type over which recursion will be performed. Consequently, the required function is derived by an application of Θ-elimination:

$$
\begin{array}{ll}
0 & x : \Theta(\lambda) \\
1 & \|C_1 \\
 & \quad \triangleright \ s_1(\varpi_1, \sigma_1, \omega_1) : D(\theta_1(\varpi_1, \sigma_1)) \\
 & \| \\
 & \vdots \\
k & \|C_k \\
 & \quad \triangleright \ s_k(\varpi_k, \sigma_k, \omega_k) : D(\theta_k(\varpi_k, \sigma_k)) \\
 & \| \\
 & \hrulefill \quad \Theta\text{-elimination} \\
 & \Theta rec(x, s_1, \ldots, s_k) : D(x)
\end{array}
$$

Premise 0 follows directly from an initial context containing the assumption

$$a : \Theta(\lambda)$$

Premises $1, \ldots, k$ are constructed within a context containing the assumption

$$y : C$$

which is nested within the initial context. In addition, construction of the i^{th} premise ($1 \leq i \leq k$) takes place in a context C_i. The construction of C_i is as described for the general elimination rule schema presented in section 2.5. If θ_i is a nullary constructor, then premise i is completed by establishing

$$s_i(y) : D(\theta_i)$$

Alternatively, if θ_i is a non-nullary object constructor and $\Theta\text{-introduction}_{\theta_i}$ has no recursive premises, then premise i is completed by establishing

$$s_i(\varpi_i, y) : D(\theta_i(\varpi_i))$$

Finally, consider the case when Θ-introduction$_i$ has recursive premises. Note that the result of applying f to $\theta_i(a_i, a_i)$ and y is of the form

$$z_i(a_i, a_i, y, f[v_{i1}][y], \ldots, f[v_{ia_i}][y])$$

By substituting w_{ij} $(1 \leq j \leq a_i)$ for $f[v_{ij}][y]$, the i^{th} premise is completed by establishing

$$z_i(a_i, a_i, w_i, y) : D(\theta_i(a_i, a_i))$$

The application of Θ-elimination gives rise to the judgement

(1.1.1)
$$\begin{array}{l} [\![a : \Theta(\bar{A}) \\ \quad \blacktriangleright \ [\![y : C \\ \quad\quad \blacktriangleright \ \Theta rec(a, z_1(y), \ldots, z_b(y)) : D(a) \\ \quad\quad]\!] \\ ]\!] \end{array}$$

By $\rightarrow$- and Π-introduction the synthesis of the function is complete:

(1.1)
$$\begin{array}{l} \lambda a.\lambda y.\Theta rec(a, z_1(y), \ldots, z_b(y)) \\ \qquad\qquad : (\Pi x : \Theta(\bar{A}))C \rightarrow D(x) \end{array}$$

6.3.2 Function verification

To prove that the synthesized function satisfies the verification condition, it is necessary to construct an object in the type

(1.2)
$$\begin{array}{l} (\Pi\, c : C) \\ \quad P_1(\lambda a.\lambda y.\Theta rec(a, z_1(y), \ldots, z_b(y)), c) \times \\ \quad\quad \vdots \\ \quad P_b(\lambda a.\lambda y.\Theta rec(a, z_1(y), \ldots, z_b(y)), c) \end{array}$$

By Π- and $\times$-introduction 1.2 is reduced to showing that k types of the form

(1.2.i) $\quad P_i(\lambda a.\lambda y.\Theta rec(a, z_1(y), \ldots, z_b(y)), c)$

are non-empty. Here we only consider the case where θ_i $(1 \leq i \leq k)$ has recursive introduction variables since this is the most general case. Given an object in the equality type

$$\begin{aligned} &Eq(D(\theta_i(\alpha_i, \alpha_i)), \\ &\quad \lambda a.\lambda y.\Theta rec(a, s_1(y), \ldots, s_h(y))[\theta_i(\alpha_i, \alpha_i)][c], \\ &\quad s_i(\alpha_i, \alpha_i, c, \lambda a.\lambda y.\Theta rec(a, s_1(y), \ldots, s_h(y))[v_{i1}][c], \\ &\qquad\qquad \vdots \\ &\qquad \lambda a.\lambda y.\Theta rec(a, s_1(y), \ldots, s_h(y))[v_{iq_i}][c])) \end{aligned} \tag{6.4}$$

a proof of 1.2.i follows by Π-introduction. Proof of 6.4 takes place in the context of the following assumptions

$$\begin{aligned} &e : C \\ &u_{i1} : A_{i1}\,; \ldots;\; u_{ip_i} : A_{ip_i} \\ &v_{i1} : \Theta(\lambda)\,; \ldots;\; v_{iq_i} : \Theta(\lambda) \end{aligned}$$

The required proof is established by showing that the following object expressions are equal elements in type $D(\theta_i(\alpha_i, \alpha_i))$:

$$\lambda a.\lambda y.\Theta rec(a, s_1(y), \ldots, s_h(y))[\theta_i(\alpha_i, \alpha_i)][c]$$

$$\begin{aligned} &s_i(\alpha_i, \alpha_i, c, \lambda a.\lambda y.\Theta rec(a, s_1(y), \ldots, s_h(y))[v_{i1}][c], \\ &\qquad\qquad \vdots \\ &\qquad \lambda a.\lambda y.\Theta rec(a, s_1(y), \ldots, s_h(y))[v_{iq_i}][c]) \end{aligned}$$

To achieve this, both expressions must be shown to have the same value, which is

$$\begin{aligned} &s_i(\alpha_i, \alpha_i, c, \Theta rec(v_{i1}, s_1(c), \ldots, s_h(c)), \\ &\qquad\qquad \vdots \\ &\qquad \Theta rec(v_{iq_i}, s_1(c), \ldots, s_h(c))) \end{aligned}$$

By transitivity and symmetry 6.4 is reduced to proving:

$$\lambda a.\lambda y.\Theta rec(a, s_1(y), \ldots, s_h(y))[\theta_i(\mathfrak{C}_i, \mathfrak{G}_i)][c] \tag{6.5}$$
$$= s_i(\mathfrak{C}_i, \mathfrak{G}_i, c, \Theta rec(v_{i1}, s_1(c), \ldots, s_h(c)),$$
$$\vdots$$
$$\Theta rec(v_{iq_i}, s_1(c), \ldots, s_h(c))) : D(\theta_i(\mathfrak{C}_i, \mathfrak{G}_i))$$

$$s_i(\mathfrak{C}_i, \mathfrak{G}_i, c, \lambda a.\lambda y.\Theta rec(a, s_1(y), \ldots, s_h(y))[v_{i1}][c], \tag{6.6}$$
$$\vdots$$
$$\lambda a.\lambda y.\Theta rec(a, s_1(y), \ldots, s_h(y))[v_{iq_i}][c])$$
$$= s_i(\mathfrak{C}_i, \mathfrak{G}_i, c, \Theta rec(v_{i1}, s_1(c), \ldots, s_h(c)),$$
$$\vdots$$
$$\Theta rec(v_{iq_i}, s_1(c), \ldots, s_h(c))) : D(\theta_i(\mathfrak{C}_i, \mathfrak{G}_i))$$

A proof of 6.5 is established by showing that the *left-hand-side* evaluates to the *right-hand-side*. This is achieved by the rules for $\rightarrow$-computation, Θ-computation$_i$ and transitivity. A proof of 6.6 follows in a similar manner. The method of proof is complicated, however, by the fact that subexpressions on the *left-hand-side* must be evaluated. Following the evaluation outlined above, q_i equality judgements are constructed:

$$\lambda a.\lambda y.\Theta rec(a, s_1(y), \ldots, s_h(y))[v_{i1}][c]$$
$$= \Theta rec(v_{i1}, s_1(c), \ldots, s_h(c)) : D(v_{i1})$$
$$\vdots$$
$$\lambda a.\lambda y.\Theta rec(a, s_1(y), \ldots, s_h(y))[v_{iq_i}][c]$$
$$= \Theta rec(v_{iq_i}, s_1(c), \ldots, s_h(c)) : D(v_{iq_i})$$

Substituting these equalites for $x_1, \ldots, x_h$ in

$$Eq(D(\theta_i(\mathfrak{C}_i, \mathfrak{G}_i)), s_i(\mathfrak{C}_i, \mathfrak{G}_i, c, x_1, \ldots, x_h), s_i(\mathfrak{C}_i, \mathfrak{G}_i, c, \Theta rec(v_{i1}, s_1(c), \ldots, s_h(c))$$
$$\vdots$$
$$\Theta rec(v_{iq_i}, s_1(c), \ldots, s_h(c))))$$
$$= Eq(D(\theta_i(\mathfrak{C}_i, \mathfrak{G}_i)), s_i(\mathfrak{C}_i, \mathfrak{G}_i, c, x_1, \ldots, x_h), s_i(\mathfrak{C}_i, \mathfrak{G}_i, c, \Theta rec(v_{i1}, s_1(c), \ldots, s_h(c))$$
$$\vdots$$
$$\Theta rec(v_{iq_i}, s_1(c), \ldots, s_h(c))))$$

and given an object in

$$Eq(D(\theta_i(a_i, a_i)), z_i(a_i, a_i, c, \Theta rec(v_{i1}, z_1(c), \ldots, z_h(c)) \\ \vdots \\ \Theta rec(v_{iq_i}, z_1(c), \ldots, z_h(c))), \\ z_i(a_i, a_i, c, \Theta rec(v_{i1}, z_1(c), \ldots, z_h(c)) \\ \vdots \\ \Theta rec(v_{iq_i}, z_1(c), \ldots, z_h(c))))$$

then a proof of 6.6 follows by the type equality rule.

6.4 Implementation

The scheme described in this chapter for deriving a primitive recursive function from a type theory specification is implemented by the ML function

```
prs : (term # asslist) -> thm
```

The first component of the input argument specifies the recursive function and the second component denotes the assumptions on which it is based. The implementation deals with recursion over lists and natural numbers. However, as demonstrated in section 6.3, the scheme generalises for any data type defined by primitive recursion. The implementation makes use of a version of Milner's[Milner 78] type check algorithm due to Petersson[Petersson 82]. To aid the type checker, an initial environment is supplied which is constructed by applying the type checker recursively to the initial list of assumptions. Consider again the *length* function discussed earlier. Applying prs to the *length* specification:

```
prs ("Sigma(Pi(List(A),(x)[N]),(f)[
            #(Eq(N,apply(f,nil),zero),
              Pi(A,(x)
                 Pi(List(A),(y)[Eq(N,apply(f,cons(x,y)),
                                   succ(apply(f,y)))])))])",
    ["A:U1"]);;
```

generates the following theorem:

```
"pair(lambda((z)[listrec(z,zero,(x,y,b13)[succ(b13)])]),
     pair(e,lambda((x)[lambda((y)e)])))
 : Sigma(Pi(List(A),(z)N),
         (f)
          [#(Eq(N,apply(f,nil),zero),
             Pi(A,
                (x)
                 [Pi(List(A),(y)[Eq(N,apply(f,cons(x,y)),
                                    succ(apply(f,y)))])])])
 [A : U1]"
: thm
```

prs is combined with the GTTS definition mechanism (def) by the ML function

```
prs_def : tok -> (term # asslist) -> tok
```

An application of prs_def applies prs to its second argument. From the resulting theorem the required object level definition is extracted and supplied with the first argument to def. As an example, consider the traditional recursive definition of *mult*:

$$\begin{cases} mult(0,y) = y \\ mult(x',y) = plus(mult(x,y),y) \end{cases}$$

which is defined in terms of *plus*:

$$\begin{cases} plus(0,y) = y \\ plus(x',y) = plus(x,y)' \end{cases}$$

We begin by introducing a definition for *plus*:

```
prs_def 'plus' ("Sigma(Pi(N,(x)[->(N,N)]),(f)[
                  Pi(N,(y)[
                      #(Eq(N,apply(apply(f,zero),y),y),
                        Pi(N,(x)[Eq(N,apply(apply(f,succ(x)),y),
                                    succ(apply(apply(f,x),y)))])])])])",
                []);;
'plus' : tok
```

By specifying *mult* in terms of *plus*, an object level representation for *mult* is generated as follows:

```
prs ("Sigma(Pi(N,(x)[->(N,N)]),(f)[
          Pi(N,(y)[
             #(Eq(N,apply(apply(f,zero),y),zero),
                  Pi(N,(x)[Eq(N,apply(apply(f,succ(x)),y),
                                  plus(apply(apply(f,x),y),y))])])])])",
      []);;
"pair(lambda((x)[lambda((y)[rec(x,zero,(x,b15)[plus(b15,y)])])]),
      lambda((y)[pair(e,lambda((x)e))]))
 : Sigma(Pi(N,(x)[->(N,N)]),
         (f)
          [Pi(N,
              (y)
               [#(Eq(N,apply(apply(f,zero),y),zero),
                  Pi(N,
                     (x)
                      [Eq(N,
                          apply(apply(f,succ(x)),y),
                          plus(apply(apply(f,x),y),y))])])])])"
: thm
```

6.5 Summary

In this chapter we have presented a type schema for specifying primitive recursive functions which exploits the uniformity of Martin-Löf's theory. We have demonstrated that for a particular instance of the specification schema, the derivation of an implementation is automatic. The scheme has been implemented in ML

for lists and natural numbers. Combined with the GTTS definition mechanism, our scheme provides a tool for automatically generating an object level definition given a specification expressed by the type schema.

Chapter 7

Conclusion

7.1 What has been accomplished?

In this thesis we have considered aspects of the mechanization of program construction in Martin-Löf's Theory of Types.

We began by briefly introducing the problem of program correctness. The constructive approach to this problem was presented and contrasted with the verification paradigm. The kind of machine assistance which has been developed for both approaches was outlined. In particular, the interest in constructive logics was noted. The relationship between programming and constructive proof was highlighted and machine assistance in performing formal proof was discussed. The need for an interactive programming assistant, as opposed to a fully automated system, was emphasized.

In our introduction to Martin-Löf's theory we emphasized the propositions-as-types principle. The general theory of expressions which underpins type the-

ory was outlined. In our explanation of the judgement forms we introduced Backhouse's context notation. Our presentation of the deductive system owes much to Backhouse's investigations into the structure of the type theory rules. The uniform presentation of judgements and derivations which Backhouse's notation allows was noted, and the distinction between propositional and judgemental proof was clarified.

With the objective of assessing current implementations of type theory in the role of programming assistant, a comparative study was undertaken. Four implementations were reviewed: GTTS, TTPA, NuPRL and a version developed using Isabelle. Assessment was based upon the four broad categories of programming: logic, framework, tools and interface. The conclusions of the study form the basis of a proposal for a more effective programming assistant.

The study looked at two versions of constructive type theory. One due to Martin-Löf, the other to Constable *et al.* Two main concerns in the assessment of a programming logic were that the logic should be uniform in structure and exploit the intuitions of the programmer as well as the theorem prover. Martin-Löf's theory satisfies both these conditions. The theory of expressions contributes to the uniformity of the theory of types, as does the close relationship which exists between the formation and introduction rules and the elimination and computation rules. The equal prominence given to objects and types within the judgement forms satisfies the second condition. NuPRL's logic does not meet these requirements to the same degree. Firstly, the NuPRL logic lacks the uniform structure which Martin-Löf's theory exhibits. This may, in part,

be attributed to the fact that the NuPRL logic is not underpinned by a general theory of expressions, which is the case with Martin-Löf's theory. Secondly, the computational content of a NuPRL proof is given secondary status. This is reflected in the requirement of an extraction mechanism which is applied once a proof is complete in order to obtain the program. Martin-Löf's theory, therefore, was judged to be more appropriate as a programming logic.

Step-wise refinement is a principal technique in developing structured programs. The integration of program derivation with the proof of its correctness requires support for step-wise refinement. Two key ingredients were identified: goal-directed proof and schema variables. The integration described above leads to an exploratory activity in which the freedom to experiment is important. For this reason the persistent proof representation, exemplified by TTPA and NuPRL, is preferred to Isabelle's goal stack framework. Inference rules are the basic building blocks of proofs. By adopting Paulson's Horn clause representation, in preference to the LCF functional style, forwards and backwards proof are unified. Furthermore, an efficient representation for derived rules is obtained. A framework is proposed which incorporates the features outlined above. Goal refinement is achieved by unification, as is the case with Isabelle. The internal structure of the derivation, however, is maintained. Instantiation of schema variables is delayed, which increases the flexibility in exploring different refinements.

We emphasise the need for an interactive system which provides a set of programming tools. A strategy editor is envisaged which represents an extension to Schmidt's notion of tactic restriction. The need for decision methods where

proofs have no computational content is stressed.

A principal failure of all the systems reviewed was the inability to vary the users view of a derivation. To overcome this problem we propose the development of a folding editor. Such an editor would allow the user to focus on a particular area of interest.

Computer assistance plays an essential part in overcoming the problem of scale associated with the construction of provably correct programs. Practical experience is an important aid in developing such computer assistance. For this reason the formal derivation of a generalised table look-up function was undertaken. Although at first sight this seems a trivial problem, the benefits of this exercise lie in the fact that it was completely formalised. Indeed, this exercise revealed that a disproportionate amount of the overall proof effort was taken up with proving negations. This observation may have been overlooked if the derivation had been constructed on an informal or selective basis.

Motivated by the programming exercise, we developed a decision method for negation. In so doing, the notions of direct and indirect contradictions were identified. The decision method is based upon the proof technique known as refutation. The search for refutation takes place at the level of the equality type. The process is generalised by providing goal transformations which push negation through the logicals and quantifiers. Analysis of equalities is achieved by a process of reduction and decomposition. A successful search for refutation results in the construction of a tree structure which embodies a justification for the required negation. Extraction of the justification is automatic. Analysis takes

place in the context of forwards proof. For each data type constructor there are associated derived rules of inference which perform the required reductions and decompositions. These derived rules of inference provide the basis for reasoning about equalities and are, therefore, useful outwith the context of proving negations. Utilising the uniform structure of Martin-Lof's theory, we have shown that the reductions and decompositions associated with an arbitrary data type constructor are completely determined by the definition of the constructor. As a consequence, new data type constructors can be incorporated uniformly within the decision method.

By exploiting the uniformity of Martin-Löf's type structure, we have formulated a schema for specifying primitive recursive functions. Furthermore, we have demonstrated that the process of deriving an object in a particular instance of the schema is automatible. This work provides practical assistance with respect to the introduction of definitions which satisfy the constraints of primitive recursion.

7.2 Topics for future research

We have identified three topics for future research:

1. The implementation of the proposal for a more practical type theory programming proof assistant, outlined in chapter 3, would be undertaken. This implementation would incorporate both the decision method for negation, documented in chapter 5, and the scheme for introducing primitive recursive definitions, demonstrated in chapter 6.

2. Our decision method for negation could be developed in two ways. Firstly, as mentioned in chapter 5, a choice may exist with respect to the application of reductions. The decision method could be improved by exploiting this choice in the search for refutation. For instance, instead of simply imposing a depth first or breadth first search, some form of "heuristic" guidance based on the structure of the goal may be possible. Secondly, our experience of program construction in type theory has shown that lemmas and corollaries which arise within the creative parts of a proof are also useful in proving the uncreative parts, such as negations. A possible line of future research will be to determine whether such properties can be automatically identified and utilised by our decision method.

3. As experiments into program derivation become more ambitious, the need for assistance in managing the complexities of formal proof will become more acute. Programming paradigms [Floyd 79] [Green & Barstow 78] may provide the level of abstraction necessary in order to alleviate some of these problems. The identification and formulation of the "paradigms of programming" in type theory, therefore, seems a useful direction for future research.

Appendix A

Deductive System

General Rules

Reflexivity

$$\frac{a : A}{a = a : A} \qquad \frac{A \text{ type}}{A = A}$$

Symmetry

$$\frac{a = b : A}{b = a : A} \qquad \frac{A = B}{B = A}$$

Transitivity

$$\frac{a = b : A \quad b = c : A}{a = c : A} \qquad \frac{A = B \quad B = C}{A = C}$$

Equality of types

$$\frac{a : A \quad A = B}{a : B} \qquad \frac{a = b : A \quad A = B}{a = b : B}$$

Substitution

$$\frac{a : A \quad [\![x : A \triangleright\ B(x)\ type]\!]}{B(a)\ type} \qquad \frac{a = c : A \quad [\![x : A \triangleright\ C(x) = D(x)]\!]}{C(a) = D(c)}$$

$$\frac{a : A \quad [\![x : A \triangleright\ b(x) : B(x)]\!]}{b(a) : B(a)} \qquad \frac{a = c : A \quad [\![x : A \triangleright\ b(x) = d(x) : B(x)]\!]}{b(a) = d(c) : B(a)}$$

Assumption

$$\frac{A\ type}{[\![a : A \triangleright\ a : A]\!]}$$

Finite types

$\{i_1, \ldots, i_n\}$***-formation***

$$\{i_1, \ldots, i_n\}\ type \qquad \text{where } n \geq 0$$

$\{i_1, \ldots, i_n\}$***-introduction***

$$i_j : \{i_1, \ldots, i_n\} \qquad \text{where } 1 \leq j \leq n$$

$$i_j : \{i_1, \ldots, i_n\} = i_j : \{i_1, \ldots, i_n\} \qquad \text{where } 1 \leq j \leq n$$

$\{i_1, \ldots, i_n\}$***-elimination***

$$\frac{a : \{i_1, \ldots, i_n\} \quad b_1 : C(i_1) \ \ldots\ b_n : C(i_n)}{case(a, b_1, \ldots, b_n) : C(a)}$$

$$\frac{a = c : \{i_1, \ldots, i_n\} \quad b_1 = d_1 : C(i_1) \ \ldots\ b_n = d_n : C(i_n)}{case(a, b_1, \ldots, b_n) = case(b, d_1, \ldots, d_n) : C(a)}$$

$\{i_1, \ldots, i_n\}$***-computation***

$$\frac{b_1 : C(i_1) \ \ldots\ b_n : C(i_n)}{case(i_j, b_1, \ldots, b_n) = b_j : C(i_j)} \qquad \text{where } 1 \leq j \leq n$$

Natural numbers

N-formation

$$\frac{}{N\ type}$$

N-introduction

$$\frac{}{0 : N} \qquad \frac{n : N}{n' : N} \qquad \frac{n = m : N}{n' = m' : N}$$

N-elimination

$$\frac{n : N \quad b : C(0) \quad [[u : N;\ v : C(u) \triangleright\ d(u,v) : C(u')]]}{nrec(n, b, [u,v]d(u,v)) : C(n)}$$

$$\frac{n = m : N \quad b = c : C(0) \quad [[u : N;\ v : C(u) \triangleright\ d(u,v) = g(u,v) : C(u')]]}{nrec(n, b, [u,v]d(u,v)) = nrec(m, c, [u,v]g(u,v)) : C(n)}$$

N-computation$_0$

$$\frac{b : C(0) \quad [[u : N;\ v : C(u) \triangleright\ d(u,v) : C(u')]]}{nrec(0, b, [u,v]d(u,v)) = b : C(0)}$$

N-computation$'$

$$\frac{n : N \quad b : C(0) \quad [[u : N;\ v : C(u) \triangleright\ d(u,v) : C(u')]]}{nrec(n', b, [u,v]d(u,v)) = d(n, nrec(n, b, [u,v]d(u,v))) : C(n')}$$

Lists

List-formation

$$\frac{A\ type}{List(A)\ type}$$

List-introduction$_{nil}$

$$\frac{A\ type}{nil : List(A)}$$

List-introduction$_{::}$

$$\frac{a : A \quad b : List(A)}{a :: b : List(A)} \qquad \frac{a = c : A \quad b = d : List(A)}{a :: c = b :: d : List(A)}$$

List-elimination

$$\frac{\begin{array}{l} l : List(A) \\ b : C(nil) \\ [[u : A;\ v : List(A);\ w : C(v) \triangleright\ d(u,v,w) : C(u :: v)]] \end{array}}{listrec(l, b, [u,v,w]d(u,v,w)) : C(l)}$$

$$\frac{\begin{array}{l} m = n : List(A) \\ b = c : C(nil) \\ [[u : A;\ v : List(A);\ w : C(v) \triangleright\ d(u,v,w) = g(u,v,w) : C(u :: v)]] \end{array}}{listrec(m, b, [u,v,w]d(u,v,w)) = listrec(n, c, [u,v,w]g(u,v,w)) : C(m)}$$

List-computation$_{nil}$

$$\frac{\begin{array}{l} b : C(nil) \\ [[u : A;\ v : List(A);\ w : C(v) \triangleright\ d(u,v,w) : C(u :: v)]] \end{array}}{listrec(nil, b, [u,v,w]d(u,v,w)) = b : C(nil)}$$

List-computation$_{::}$

$$\frac{\begin{array}{l} h : A \\ t : List(A) \\ b : C(nil) \\ [[u : A;\ v : List(A);\ w : C(v) \triangleright\ d(u,v,w) : C(u :: v)]] \end{array}}{listrec(h :: t, b, [u,v,w]d(u,v,w)) = d(h, t, listrec(t, b, [u,v,w]d(u,v,w))) : C(h :: t)}$$

Cartesian product of a family of types

Π-*formation*

$$\frac{A\ type \quad [\![x : A \triangleright\ B(x)\ type]\!]}{(\Pi x : A)B(x)\ type} \qquad \frac{A = C \quad [\![x : A \triangleright\ B(x) = D(x)]\!]}{(\Pi x : A)B(x) = (\Pi x : C)D(x)}$$

Π-*introduction*

$$\frac{A\ type \quad [\![x : A \triangleright\ b(x) : B(x)]\!]}{\lambda x.b(x) : (\Pi x : A)B(x)} \qquad \frac{A\ type \quad [\![x : A \triangleright\ b(x) = d(x) : B(x)]\!]}{\lambda x.b(x) = \lambda x.d(x) : (\Pi x : A)B(x)}$$

Π-*elimination*

$$\frac{f : (\Pi x : A)B(x) \quad a : A}{f[a] : B(a)} \qquad \frac{f = g : (\Pi x : A)B(x) \quad a = b : A}{f[a] = g[b] : B(a)}$$

Π-*computation*

$$\frac{a : A \quad [\![x : A \triangleright\ b(x) : B(x)]\!]}{(\lambda x.b(x))[a] = b(a) : B(a)} \qquad \frac{f : (\Pi x : A)B(x)}{\lambda x.(f[x]) = f : (\Pi x : A)B(x)}$$

Disjoint union of a family of types

Σ-*formation*

$$\frac{A\ type \quad [\![x : A \triangleright\ B(x)\ type]\!]}{(\Sigma x : A)B(x)\ type} \qquad \frac{A = C \quad [\![x : A \triangleright\ B(x) = D(x)]\!]}{(\Sigma x : A)B(x) = (\Sigma x : C)D(x)}$$

Σ-*introduction*

$$\frac{a : A \quad b(a) : B(a)}{\langle a, b(a)\rangle : (\Sigma x : A)B(x)} \qquad \frac{a = c : A \quad b(a) = d(a) : B(a)}{\langle a, b(a)\rangle = \langle c, d(c)\rangle : (\Sigma x : A)B(x)}$$

Σ-*elimination*

$$\frac{p : (\Sigma x : A)B(x) \quad [\![u : A;\ v : B \triangleright\ d(u, v) : C(\langle u, v\rangle)]\!]}{split(p, [u, v]d(u, v)) : C(p)}$$

$$\frac{p = q : (\Sigma x : A)B(x) \quad [\![u : A;\ v : B \triangleright\ d(u, v) = g(u, v) : C(\langle u, v\rangle)]\!]}{split(p, [u, v]d(u, v)) = split(q, [u, v]g(u, v)) : C(p)}$$

Σ-computation

$$\frac{a : A \quad b(a) : B(a) \quad [\![u : A;\ v : B \triangleright\ d(u,v) : C(\langle u,v\rangle)]\!]}{split(\langle a,b(a)\rangle,[u,v]d(u,v)) = d(a,b(a)) : C(\langle a,b(a)\rangle)}$$

Function types

→-formation

$$\frac{A\ type \quad B\ type}{A \to B\ type} \qquad \frac{A = C \quad B = D}{A \to B = C \to D}$$

→-introduction

$$\frac{A\ type \quad [\![x : A \triangleright\ b(x) : B]\!]}{\lambda x.b(x) : A \to B} \qquad \frac{A\ type \quad [\![x : A \triangleright\ b(x) = d(x) : B]\!]}{\lambda x.b(x) = \lambda x.d(x) : A \to B}$$

→-elimination

$$\frac{f : A \to B \quad a : A}{f[a] : B} \qquad \frac{f = g : A \to B \quad a = b : A}{f[a] = g[b] : B}$$

→-computation

$$\frac{a : A \quad [\![x : A \triangleright\ b(x) : B]\!]}{\lambda x.b(x)[a] = b(a) : B} \qquad \frac{f : A \to B}{\lambda x.(f[x]) = f : A \to B}$$

Cartesian product of two types

×-formation

$$\frac{A\ type \quad B\ type}{A \times B\ type} \qquad \frac{A = C \quad B = D}{A \times B = C \times D}$$

×-*introduction*

$$\frac{a : A \quad b : B}{\langle a, b\rangle : A \times B} \qquad \frac{a = c : A \quad b = d : B}{\langle a, b\rangle = \langle c, d\rangle : A \times B}$$

×-*elimination*

$$\frac{p : A \times B \quad [[u : A;\ v : B \triangleright\ d(u, v) : C(\langle u, v\rangle)]]}{split(p, [u, v]d(u, v)) : C(p)}$$

$$\frac{p = q : A \times B \quad [[u : A;\ v : B \triangleright\ d(u, v) = g(u, v) : C(\langle u, v\rangle)]]}{split(p, [u, v]d(u, v)) = split(q, [u, v]g(u, v)) : C(p)}$$

×-*computation*

$$\frac{a : A \quad b : B \quad [[u : A;\ v : B;\ \triangleright\ d(u, v) : C(\langle u, v\rangle)}{split(\langle a, b\rangle, [u, v]d(u, v)) = d(a, b) : C(\langle a, b\rangle)}$$

Disjoint union of two types

+-*formation*

$$\frac{A\ type \quad B\ type}{A + B\ type} \qquad \frac{A = C \quad B = D}{A + B = C + D}$$

+-*introduction*$_{inl}$

$$\frac{a : A \quad B\ type}{inl(a) : A + B} \qquad \frac{a = c : A \quad B\ type}{inl(a) = inl(c) : A + B}$$

+-*introduction*$_{inr}$

$$\frac{A\ type \quad b : B}{inr(b) : A + B} \qquad \frac{A\ type \quad b = d : B}{inr(b) = inr(d) : A + B}$$

+-*elimination*

$$\frac{p : A + B \quad [[u : A \triangleright\ c(u) : C(inl(u))]] \quad [[v : B \triangleright\ d(v) : C(inr(v))]]}{when(p, [u]c(u), [v]d(v)) : C(p)}$$

$$\frac{p = q : A + B \quad [[u : A \triangleright\ c(u) = d(u) : C(inl(u))]] \quad [[v : B \triangleright\ f(v) = g(v) : C(inr(v))]]}{when(p, [u]c(u), [v]f(v)) = when(q, [u]d(u), [v]g(v)) : C(p)}$$

+-computation$_{inl}$

$$\frac{a : A \quad [u : A \triangleright c(u) : C(inl(u))] \quad [v : B \triangleright d(v) : C(inr(v))]}{when(inl(a), [u]c(u), [v]d(v)) = c(a) : C(inl(a))}$$

+-computation$_{inr}$

$$\frac{b : B \quad [u : A \triangleright c(u) : C(inl(u))] \quad [v : B \triangleright d(v) : C(inr(v))]}{when(inr(b), [u]c(u), [v]d(v)) = d(b) : C(inr(b))}$$

Well-orderings

W-formation

$$\frac{A \; type \quad [x : A \triangleright B(x) \; type]}{(Wx : A)B(x) \; type} \qquad \frac{A = C \quad [x : A \triangleright B(x) = D(x)]}{(Wx : A)B(x) = (Wx : C)D(x)}$$

W-introduction

$$\frac{a : A \quad b : B(a) \to (Wx : A)B(x)}{\langle |a, b| \rangle : (Wx : A)B(x)} \qquad \frac{a = c : A \quad b = d : B(a) \to (Wx : A)B(x)}{\langle |a, b| \rangle = \langle |c, d| \rangle : (Wx : A)B(x)}$$

W-elimination

$$\frac{\begin{array}{l} s : (Wx : A)B(x) \\ [u : A \\ ; v : B(u) \to (Wx : A)B(x) \\ ; w : (\Pi y : B(u))C(v(y)) \\ \triangleright \; b(u, v, w) : C(\langle |u, v| \rangle) \\ ] \end{array}}{wrec(s, [u, v, w]b(u, v, w)) : C(s)}$$

$$\frac{\begin{array}{l} s = t : (Wx : A)B(x) \\ [u : A \\ ; v : B(u) \to (Wx : A)B(x) \\ ; w : (\Pi y : B(u))C(v(y)) \\ \triangleright \; b(u, v, w) = d(u, v, w) : C(\langle |u, v| \rangle) \\ ] \end{array}}{wrec(s, [u, v, w]b(u, v, w)) = wrec(t, [u, v, w]d(u, v, w)) : C(s)}$$

W-computation

$$\begin{array}{l} a : A \\ b : B(a) \to (W x : A)B(x) \\ [\![u : A \\ ; v : B(u) \to (W x : A)B(x) \\ ; w : (\Pi y : B(u))C(v(y)) \\ \triangleright\ b(u,v,w) : C(\langle\!|u,v|\!\rangle) \\ ]\!] \end{array}$$

$$\overline{wrec(\langle\!|a,b|\!\rangle, [u,v,w]d(u,v,w)) = d(a,b,\lambda x.wrec(b[x],[u,v,w]d(u,v,w))) : C(\langle\!|a,b|\!\rangle)}$$

Equality types

Eq-formation

$$\frac{A\ type \quad a : A \quad b : A}{Eq(A,a,b)\ type} \qquad \frac{A = C \quad a = c : A \quad b = d : A}{Eq(A,a,b) = Eq(C,c,d)}$$

Eq-introduction

$$\frac{a = b : A}{e : Eq(A,a,b)} \qquad \frac{a = b : A}{e = e : Eq(A,a,b)}$$

Eq-elimination

$$\frac{e : Eq(A,a,b)}{a = b : A}$$

Universes

U_n-formation

$$\frac{}{U_n\ type}$$

U_n-introduction

$$\{i_1,\ldots,i_m\} : U_n \qquad \{i_1,\ldots,i_m\} = \{i_1,\ldots,i_m\} : U_n$$

$$N : U_n \qquad N = N : U_n$$

$$\frac{A : U_n}{List(A) : U_n} \qquad \frac{A = B : U_n}{List(A) = List(B) : U_n}$$

$$\frac{A : U_n \quad [\![x : A \triangleright B(x) : U_n]\!]}{(\Pi x : A)B(x) : U_n} \qquad \frac{A = C : U_n \quad [\![x : A \triangleright B(x) = D(x) : U_n]\!]}{(\Pi x : A)B(x) = (\Pi x : C)D(x) : U_n}$$

$$\frac{A : U_n \quad [\![x : A \triangleright B(x) : U_n]\!]}{(\Sigma x : A)B(x) : U_n} \qquad \frac{A = C : U_n \quad [\![x : A \triangleright B(x) = D(x) : U_n]\!]}{(\Sigma x : A)B(x) = (\Sigma x : C)D(x) : U_n}$$

$$\frac{A : U_n \quad B : U_n}{A \to B : U_n} \qquad \frac{A = C : U_n \quad B = D : U_n}{A \to B = C \to D : U_n}$$

$$\frac{A : U_n \quad B : U_n}{A \times B : U_n} \qquad \frac{A = C : U_n \quad B = D : U_n}{A \times B = C \times D : U_n}$$

$$\frac{A : U_n \quad B : U_n}{A + B : U_n} \qquad \frac{A = C : U_n \quad B = D : U_n}{A + B = C + D : U_n}$$

$$\frac{A : U_n \quad [\![x : A \triangleright B(x) : U_n]\!]}{(Wx : A)B(x) : U_n} \qquad \frac{A = C : U_n \quad [\![x : A \triangleright B(x) = D(x) : U_n]\!]}{(Wx : A)B(x) = (Wx : C)D(x) : U_n}$$

$$\frac{A : U_n \quad a : A \quad b : A}{Eq(A,a,b) : U_n} \qquad \frac{A = C : U_n \quad a = c : A \quad b = d : A}{Eq(A,a,b) = Eq(C,c,d) : U_n}$$

U_n-elimination

$$\frac{A : U_n}{A\ type} \qquad \frac{A = B : U_n}{A = B}$$

$$\frac{A : U_n}{A : U_{n+1}} \qquad \frac{A = B : U_n}{A = B : U_{n+1}}$$

Appendix B

Programming Exercise

B.1 Definitions and problem specification

Definitions

Difficulties with the GTTS definition mechanism led to the introduction of the following rules defining the operators *fst*, *snd* and *append*:

Σ-*elimination*$_{fst}$

$$\frac{a : (\Sigma x : A)B(x)}{fst(a) : A} \qquad \frac{a = c : (\Sigma x : A)B(x)}{fst(a) = fst(c) : A}$$

$$\frac{a = c : (\Sigma x : A)B(x)}{fst(a) = split(c, [u, v]u) : A}$$

Σ-*elimination*$_{snd}$

$$\frac{a : (\Sigma x : A)B(x)}{snd(a) : B(fst(a))} \qquad \frac{a = c : (\Sigma x : A)B(x)}{snd(a) = snd(c) : B(fst(a))}$$

$$\frac{a = c : (\Sigma x : A)B(x)}{snd(a) = split(c, [u, v]v) : B(fst(a))}$$

Σ-*computation*$_{fst}$

$$\frac{a : A \quad b : B(a)}{fst(\langle a, b\rangle) = a : A}$$

Σ-*computation*$_{snd}$

$$\frac{a : A \quad b : B(a)}{snd(\langle a, b\rangle) = b : B(\langle a, b\rangle)}$$

$\times$-*elimination*$_{fst}$

$$\frac{a : A \times B}{fst(a) : A} \qquad \frac{a = c : A \times B}{fst(a) = fst(c) : A}$$

$$\frac{a = c : A \times B}{fst(a) = split(c, [u, v]u) : A}$$

$\times$-*elimination*$_{snd}$

$$\frac{a : A \times B}{snd(a) : B} \qquad \frac{a = c : A \times B}{snd(a) = snd(c) : B}$$

$$\frac{a = c : A \times B}{snd(a) = split(c, [u, v]v) : B}$$

$\times$-*computation*$_{fst}$

$$\frac{a : A \quad b : B}{fst(\langle a, b\rangle) = a : A}$$

$\times$-*computation*$_{snd}$

$$\frac{a : A \quad b : B}{snd(\langle a, b\rangle) = b : B}$$

$\times$-*computation*$_{()}$

$$\frac{x : A \times B}{\langle fst(x), snd(x)\rangle = x : A \times B}$$

List-elimination$_{append}$

$$\frac{a : List(A) \quad b : List(A)}{append(a, b) : List(A)} \qquad \frac{a = b : List(A) \quad c = d : List(A)}{append(a, c) = append(b, d) : List(A)}$$

$$\frac{a = b : List(A) \quad c = d : List(A)}{append(a, c) = listrec(b, d, [u, v, w]u :: w) : List(A)}$$

$\mathit{List\text{-}computation}_{\mathit{append}_{nil}}$

$$\frac{b : \mathit{List}(B)}{\mathit{append}(\mathit{nil}, b) = b : \mathit{List}(B)}$$

$\mathit{List\text{-}computation}_{\mathit{append}_{ll}}$

$$\frac{h : B \quad t : \mathit{List}(B) \quad b : \mathit{List}(B)}{\mathit{append}(h :: t, b) = h :: \mathit{append}(t, b) : \mathit{List}(B)}$$

```
def 'Member'
    "(a,l,A,B)Sigma(List(#(A,B)),(h)[
                Sigma(List(#(A,B)),(t)[
                  Sigma(B,(b)[
                    Eq(List(#(A,B)),append(h,cons(pair(a,b),t)),l)])])])";;

def 'Not' "(A)->(A,{})";;
```

Specifications

The size of the proof resulted in the sub-division of the problem into three parts.

The corresponding subspecifications are presented below:

Specification of absurdity 1

```
let abs1_goal = "Not(Member(a,nil,A,B))"
and abs1_assl = ["A : U1";
                 "B : U1";
                 "a : A"]
;;
```

Specification of absurdity 2

```
let abs2_goal = "Not(Member(a,cons(u10,u11),A,B))"
and abs2_assl = ["A : U1";
                 "B : U1";
                 "a : A";
                 "u10 : #(A,B)";
                 "u11 : List(#(A,B))";
                 "b11 : Not(Eq(A,a,1st(u10)))";
                 "u14 : Not(Member(a,u11,A,B))"]
;;
```

Specification of the main problem

```
let main_goal = "Pi(A,(a)[
                    Pi(List(#(A,B)),(l)[
                       +(Member(a,l,A,B),Not(Member(a,l,A,B)))])])"
and main_assl = ["A:U1";"B:U1";
                 "equ:Pi(A,(x)[
                         Pi(A,(y)[
                            +(Eq(A,x,y),Not(Eq(A,x,y)))])])"]
;;
```

B.2 Proof strategies

```
let
VARintrUistrat =
    (Uielimtac THEN VARintrtac)
;;

let
PAIRformstrat =
    (PAIRformtac THENL
        [VARintrUistrat;
         VARintrUistrat]
    )
;;

let
LISTformstrat =
    (LISTformtac THEN
         PAIRformstrat)
;;

let
FORMstrat =
    (PAIRformstrat    ORELSE
    (LISTformstrat    ORELSE
     VARintrUistrat))
;;

let
VARintrstrat =
    (VARintrtac THEN FORMstrat)
;;

let
LISTintrnilstrat =
     (LISTintrniltac THEN LISTformstrat)
;;

let
LISTintrconsstrat =
    (LISTintrconstac THEN
         VARintrstrat)
;;
```

```
let
INTRstrat =
    (VARintrstrat         ORELSE
    (LISTintrailstrat     ORELSE
     LISTintrconstrat ))
;;

let
EQformstrat =
    (EQforutac THENL
        [LISTformstrat;
         (LISTelimappendtac THENL
             [INTRstrat;
              (LISTintrconstac THENL
                  [(PAIRintrtac THEN
                       VARintrstrat);
                   VARintrstrat])]);
          INTRstrat])
;;

let
Memberformstrat  =
    (REPEAT (SIGMAforutac      ORELSE
            (FORMstrat         ORELSE
             EQformstrat))
    )
;;

let
Solve_h_strat q =
    ((SIGMAelim1tac "(h)[Sigma(List(@(A,B)),(t)[
                              Sigma(B,(b)[
                                 Eq(List(@(A,B)),
                                         append(h,cons(pair(a,b),t)),
                                         "q)])])]" ) THEN
      (VARintrtac THEN Memberformstrat))
;;

let
Bind_h_strat p q =
    ((SIGMAelim2tac p
                    "List(@(A,B))"
                    "(h)[Sigma(List(@(A,B)),(t)[
                            Sigma(B,(b)[
                               Eq(List(@(A,B)),
                                       append(h,cons(pair(a,b),t)),
                                       "q)])])]") THEN
       (INST_auto_tac THEN
            (VARintrtac THEN Memberformstrat)) )
```

```
;;

let
Solve_t_strat p q =
    ((SIGMAelim1tac "(t)[Sigma(B,(b)[
                                Eq(List(@(A,B)),
                                   append(fst(^p),
                                          cons(pair(a,b),t)),
                                   ^q)])]") THEN
        (Bind_h_strat p q))
;;

let
Bind_t_strat p q =
    ((SIGMAelim2tac "snd(^p)"
                    "List(@(A,B))"
                    "(t)[Sigma(B,(b)[
                                Eq(List(@(A,B)),
                                   append(fst(^p),
                                          cons(pair(a,b),t)),
                                   ^q)])]") THEN
        (Bind_h_strat p q))
;;

let
Solve_b_strat p q =
    ((SIGMAelim1tac "(b)[Eq(List(@(A,B)),
                            append(fst(^p),
                                   cons(pair(a,b),
                                        fst(snd(^p)))),
                            ^q)]") THEN
        (Bind_t_strat p q))
;;

let
Bind_b_strat p q =
    ((SIGMAelim2tac "snd(snd(^p))"
                    "B"
                    "(b)[Eq(List(@(A,B)),
                               append(fst(^p),
                                      cons(pair(a,b),
                                           fst(snd(^p)))),
                               ^q)]") THEN
        (Bind_t_strat p q))
;;

let
Solve_cons_strat p q =
    (LISTintrconstac THENL
```

```
          [(PAIRintrtac THENL
               [VARintrstrat;
                (Solve_b_strat p q)]);
           (Solve_t_strat p q)])
;;

let
EQformnilstrat p q =
    (EQformtac THENL
          [LISTformstrat;
           (LISTelimappendtac THENL
               [LISTintrnilstrat;
                (Solve_coms_strat p q)]);
           (LISTintrnilstrat ORELSE
            LISTintrcomsstrat)])
;;

let
EQformcomsstrat p q =
    (EQformtac THENL
       [LISTformstrat;
        (LISTelimappendtac THENL
            [LISTintrcomsstrat;
             (Solve_coms_strat p q)]);
        (LISTintrnilstrat ORELSE
         LISTintrcomsstrat)])
;;

let
UiintrLISTstrat =
    (UiintrLISTtac THEN
        (UiintrPAIRtac THEN VARintrtac))
;;

let
AbsurdityiBasisstrat p  =
 ((FUNintrtac []) THENL
    [(EQformnilstrat p nill);
       ((EQTYPEtac "List(#(A,B))") THENL
          [((SPECtac "nil") THEN
            LISTintrnilstrat);
           (Uielimtac THEN
            ((TRANStac "listrec(cons(pair(a,fst(snd(snd(^p)))),
                                     fst(snd(^p))),
                               {},
                               (z1,z2,z3)[List(#(A,B))])") THENL
```

```
[(SYMtac THEN
   ((LISTeqconstac "(u)[U1]" "@(A,B)") THENL
     [(PAIRintrtac THENL
         [VARintrstrat;
          (Solve_b_strat p nill)]);
         (Solve_t_strat p nill);
         IDtac;
         U1intrLISTstrat]));
      ((TRANStac "listrec(nil,
                          {},
                          (z1,z2,z3)[List(@(A,B))])") THENL
         [((SUBSTtac "List(@(A,B))"
                     ["(z)[listrec(z,
                                   {},
                                   (z1,z2,z3)[List(@(A,B))])]";
                      "(z)[listrec(z,
                                   {},
                                   (z1,z2,z3)[List(@(A,B))])]";
                      "(z)[U1]"]
                     ["cons(pair(a,
                                 fst(snd(snd(^p)))),
                            fst(snd(^p)))";
                      "nil"]) THENL
            [((TRANStac "append(nil,
                                cons(pair(a,fst(snd(snd(^p)))),
                                     fst(snd(^p))))") THENL
               [(SYMtac THEN
                   (LISTeqappendniltac THEN
                      (Solve_cons_strat p nill)));
                (EQelimtac THEN
                   (INST_auto_tac THEN
                      (VARintrtac THEN
                         (EQformnilstrat p nill)))));
                (REFLtac THEN
                   ((LISTelimtac "(z)[U1]"
                                 "z"
                                 "@(A,B)"
                                 [])    THENL
                      [VARintrstrat;
                       IDtac;
                       U1intrLISTstrat])))]);
      ((LISTeqniltac "(u)[U1]"
                     "@(A,B)") THENL
          [IDtac;
           U1intrLISTstrat])])
 ])))))
;;
```

```
let
Absurdity1Inductstrat p x y =
    ((FUNintrtac []) THENL
         [(EQformconstrat p nil);
          ((EQTYPEtac "List(#(A,B))") THENL
               [((SPECtac "nil") THEN
                   LISTintrnilstrat);
                (Uielimtac THEN
                   ((TRANStac "listrec(cons(^x,
                                            append(^y,
                                              cons(pair(a,
                                                        fst(snd(snd(^p)))),
                                                   fst(snd(^p))))),
                                       {},
                                       (z1,z2,z3)[List(#(A,B))])") THENL
                     [(SYMtac THEN
                        ((LISTeqconstac "(u)[U1]" "#(A,B)") THENL
                           [VARintrstrat;
                            (LISTelimappendtac THENL
                              [VARintrstrat;
                               (Solve_cons_strat p nil)]);
                            IDtac;
                            UiintrLISTstrat]));
                      ((TRANStac "listrec(nil,
                                          {},
                                          (z1,z2,z3)[List(#(A,B))])") THENL
                        [((SUBSTtac "List(#(A,B))"
                                    ["(x)[listrec(x,
                                                  {},
                                                  (z1,z2,z3)[List(#(A,B))])]";
                                     "(x)[listrec(x,
                                                  {},
                                                  (z1,z2,z3)[List(#(A,B))])]";
                                     "(x)[U1]"]
                                    ["cons(^x,append(^y,
                                                     cons(pair(a,
                                                               fst(snd(snd(^p)))),
                                                          fst(snd(^p)))))";
                                     "nil"]) THENL
                          [((TRANStac "append(cons(^x,^y),
                                              cons(pair(a,fst(snd(snd(^p)))),
                                                   fst(snd(^p))))") THENL
                             [(SYMtac THEN
                                 (LISTeqappendconstac THENL
                                    [VARintrstrat;
                                     VARintrstrat;
                                     (Solve_cons_strat p nil)]));
                              (EQelimtac THEN
                                 (INST_auto_tac THEN
```

```
                                        (VARintrtac THEN
                                           (EQformcomsstrat p nill)))))]);
                                (REFLtac THEN
                                   ((LISTelimtac "(u)[U1]"
                                                 "x"
                                                 "@(A,B)"
                                                 [])    THENL
                                      [VARintrstrat;
                                       IDtac;
                                       UiintrLISTstrat])));
                           ((LISTeqmiltac "(u)[U1]"
                                          "@(A,B)") THENL
                                [IDtac;
                                 UiintrLISTstrat])])
                    ])))])
;;

let
Absurdity1strat =
   \g.((FUNintrtac ["_1"]) THENL
         [Memberformstrat;
          ((FUNelimtac [] "Eq(List(@(A,B)),
                              append(fst(^p),
                                 cons(pair(a,fst(snd(snd(^p)))),
                                      fst(snd(^p)))),
                              nil)" ) THENL
              [((LISTelimtac "(z)[Not(Eq(List(@(A,B)),
                                         append(z,
                                            cons(pair(a,
                                               fst(snd(snd(^p)))),
                                            fst(snd(^p)))),
                                         nil))]"
                             "fst(^p)"
                             "@(A,B)"
                             ["_2";"_3";"_4"]) THENL
                  [(Solve_h_strat nill);
                   (Absurdity1Basisstrat p);
                   (Absurdity1Inducstrat p x y)]);
               (Bind_b_strat p nill)])]
               where p = (un "_1" g) and
                     x = (un "_2" g) and
                     y = (un "_3" g)) g
;;

let
a_eqn_fst_a_strat a v p =
    (EQintrtac THEN
        ((TRANStac "fst(pair(a,fst(snd(snd(^p)))))") THENL
        [(SYMtac THEN
```

```
    ((PAIReqltac "B") THENL
        [VARintrstrat;
         (Solve_b_strat p "cons(^u,^v)")]));
((PAIRelim1tac "B") THEN
     ((TRANStac "listrec(cons(pair(a,fst(snd(snd(^p)))),
                              fst(snd(^p))),
                         pair(a,snd(^u)),
                         (x1,y1,z1)[x1])") THENL
      [(SYMtac THEN
           ((LISTeqconstac "(z)[@(A,B)]" "@(A,B)") THENL
                [(PAIRintrtac THENL
                     [VARintrstrat;
                      (Solve_b_strat p "cons(^u,^v)")]);
                 (Solve_t_strat p "cons(^u,^v)");
                 (PAIRintrtac THENL
                     [VARintrstrat;
                      ((PAIRelim2tac "A") THEN
                          VARintrstrat)]);
                  VARintrstrat]));
          ((TRANStac "listrec(cons(^u,^v),
                              pair(a,snd(^u)),
                              (x2,y2,z2)[x2])") THENL
               [((LISTelimtac "(z)[@(A,B)]"
                              "cons(pair(a,fst(snd(snd(^p)))),
                                    fst(snd(^p)))"
                              "@(A,B)"
                              [] ) THENL
                  [((TRANStac "append(nil,
                                  cons(pair(a,
                                            fst(snd(snd(^p)))),
                                       fst(snd(^p))))") THENL
                     [(SYMtac THEN
                        (LISTeqappendniltac THEN
                           (Solve_cons_strat p "cons(^u,^v)")));
                      (EQelimtac THEN
                        (INST_auto_tac THEN
                           (VARintrtac THEN
                              (EQformmilstrat p "cons(^u,^v)")))))]);
                   (REFLtac THEN
                      (PAIRintrtac THENL
                         [VARintrstrat;
                          ((PAIRelim2tac "A") THEN
                             VARintrstrat)]));
                   (REFLtac THEN VARintrstrat)]);
                ((LISTeqconstac "(z)[@(A,B)]"
                                "@(A,B)") THENL
                    [VARintrstrat;
                     VARintrstrat;
                     (PAIRintrtac THENL
```

```
                                    [VARintrstrat;
                                     ((PAIRelim2tac "A") THEN
                                         VARintrstrat)]);
                                  VARintrstrat])])])])])
;;

let
append_equ_cons_strat p u v x y =
   (SYMtac THEN
     ((SUBSTtac "List(@(A,B))"
                ["(z)[listrec(cons(^x,z),
                              ^v,
                              (x3,y3,z3)[y3])]";
                 "(z)[z]";
                 "(z)[List(@(A,B))]"]
                ["append(^y,cons(pair(a,fst(snd(snd(^p)))),fst(snd(^p))))";
                 "append(^y,cons(pair(a,fst(snd(snd(^p)))),fst(snd(^p))))"])
        THENL [(REFLtac THEN
                  (LISTelimappendtac THENL
                     [VARintrstrat;
                      (Solve_cons_strat p "cons(^u,^v)")]));
               ((LISTeqconstac "(z)[List(@(A,B))]"
                               "@(A,B)") THEN VARintrstrat)]))
;;

let
cons_equ_append_strat p u v x y =
   ((TRANStac "listrec(cons(^u,^v),
                       ^v,
                       (x4,y4,z4)[y4])") THENL
          [((LISTelimtac "(z)[List(@(A,B))]"
                         "cons(^x,append(^y,
                                         cons(pair(a,fst(snd(snd(^p)))),
                                              fst(snd(^p)))))"
                         "@(A,B)"
                         []) THENL
                [((TRANStac "append(cons(^x,^y),
                                    cons(pair(a,fst(snd(snd(^p)))),
                                         fst(snd(^p))))")  THENL
                                    [(SYMtac THEN
                                        (LISTeqappendconstac THENL
                                           [VARintrstrat;
                                            VARintrstrat;
                                            (Solve_cons_strat
                                               p
                                               "cons(^u,^v)")]));
                                     (EQelimtac THEN
                                        (INST_auto_tac THEN
                                           (VARintrtac THEN
```

```
                                                    (EQformconsstrat
                                                      p
                                                      "cons(^u,^v)")))]]);
                                          (REFLtac THEN VARintrstrat);
                                          (REFLtac THEN VARintrstrat)]);
                                        ((LISTeqconstac "(x)[List(@(A,B))]"
                                                        "@(A,B)") THEN
                                            VARintrstrat) ])
;;

let
Mem_a_v_strat p u v x y =
      ((SIGMAintrtac [y]) THENL
         [VARintrstrat;
          ((SIGMAintrtac ["fst(snd(^p))"])  THENL
              [(Solve_t_strat p "cons(^u,^v)");
               ((SIGMAintrtac ["fst(snd(snd(^p)))"]) THENL
                   [(Solve_b_strat p "cons(^u,^v)");
                    (EQintrtac THEN
                       ((TRANStac "listrec(cons(^x,
                                        append(^y,
                                          cons(pair(a,fst(snd(snd(^p)))),
                                                    fst(snd(^p))))),
                                            ^v,
                                            (x3,y3,z3)[y3])") THENL
                          [(append_equ_cons_strat p u v x y);
                           (cons_equ_append_strat p u v x y)])])])])])
;;

let
EquKlemformstrat =
        (EQformtac THENL
           [VARintrUlstrat;
            VARintrstrat;
            ((PAIRelimtac "B") THEN
                VARintrstrat)])
;;

let
Equ_or_Mem_strat u v p =
    \g.((FUNelimtac [] "Eq(List(@(A,B)),
                          append(fst(^p),cons(pair(a,fst(snd(snd(^p)))),
                                              fst(snd(^p)))),
                                 cons(^u,^v))") THENL
        [((LISTelimtac
              "(z)[->(Eq(List(@(A,B)),
                          append(z,cons(pair(a,fst(snd(snd(^p)))),
                                        fst(snd(^p)))),
                          cons(^u,^v)),
```

```
                        +(Eq(A,a,fst(^u)),
                          Member(a,^v,A,B)))]"
                  "fst(^p)"
                  "@(A,B)"
                  ["_1";"_2";"_3"]) THENL
                [(Solve_h_strat "cons(^u,^v)");
                 ((FUNintrtac [])THENL
                      [(EQformulstrat p "cons(^u,^v)");
                        ((UNIONintrtac 1 ) THENL
                              [(a_equ_fst_u_strat u v p);
                               Memberformstrat])]);
                 ((FUNintrtac []) THENL
                     [(EQformconststrat p "cons(^u,^v)");
                      ((UNIONintrtac 2) THENL
                         [EquKlomformstrat;
                          (Mem_a_v_strat p u v  x y)])])]);
            (Bind_b_strat p "cons(^u,^v)")]

        where x = (un "_1" g)  and
              y = (un "_2" g)) g
;;

let
EquNotEquAbsurdstrat u =
                 ((FUNelimtac [] "Eq(A,a,fst(^u))") THENL
                    [(INST_auto_tac THEN
                        (VARintrtac THEN
                           (FUNformtac THENL [EquKlomformstrat;
                                              IDtac])));
                     (INST_auto_tac THEN
                        (VARintrtac THEN
                           EquKlomformstrat))])
;;

let
MemNotMemAbsurdstrat v =
                 ((FUNelimtac [] "Member(a,^v,A,B)") THENL
                    [(INST_auto_tac THEN
                        (VARintrtac THEN
                           (FUNformtac THENL [Memberformstrat;
                                              IDtac])));
                     (INST_auto_tac THEN
                        (VARintrtac THEN
                          Memberformstrat))])
;;

let
Absurdity2strat u v =
    \g.((FUNintrtac ["_1"]) THENL
```

```
            [Memberformstrat;
             ((UNIONelimtac "(z)[{}]"
                            []
                            "Eq(A,a,fst(^u))"
                            "Member(a,^v,A,B)"
                            []) THENL
                [(Equ_or_Mem_strat u v p);
                 (EquNotEquAbsurdstrat u);
                 (MemNotMemAbsurdstrat v)])]
          where p = (nu "_1" g))  g
;;

let
EquTeststrat u =
        ((PIelimtac "(y)[+(Eq(A,a,y),
                           Not(Eq(A,a,y)))]"
                    "fst(^u)" "A") THENL
          [((PIelimtac "(x)[Pi(A,(y)[+(Eq(A,x,y),
                                       Not(Eq(A,x,y)))])]"
                       "a" "A") THENL
            [(INST_auto_tac THEN
               (VARintrtac THEN
                (PIformtac THENL
                 [VARintrUistrat;
                  (PIformtac THENL
                   [VARintrUistrat;
                    (UNIONformtac THENL
                     [EquBasestrat;
                      (FUNformtac THENL [EquBasestrat;
                                         IDtac])])])])));
             VARintrstrat]);
            ((PAIRelimtac "B") THEN
                VARintrstrat)]
          where EquBasestrat =
                   (EQformtac THENL
                      [FORMstrat;
                       VARintrstrat;
                       VARintrstrat]))
;;

let
EquCasestrat u v =
        ((UNIONintrtac 1) THENL
          [((SIGMAintrtac [nil]) THENL
            [(LISTintrniltac THEN LISTformstrat);
             ((SIGMAintrtac [v]) THENL
                [(VARintrtac THEN LISTformstrat);
                 ((SIGMAintrtac ["cmd(^u)"]) THENL
                    [((PAIRelim2tac "A") THEN
```

```
                        (VARintrtac THEN PAIRformstrat));
                    (EQintrtac THEN
                       ((TRANStac "cons(pair(a,snd(^u)),^v)") THENL
                       [(LISTeqappendniltac THEN
                           (LISTintrconstac THENL
                              [ResolvePAIRstrat;
                               (VARintrtac THEN LISTformstrat)]));
                        ((SUBSTtac "@(A,B)"
                                   ["(z)[cons(z,^v)]";
                                    "(z)[cons(z,^v)]";
                                    "(z)[list(@(A,B))]"]
                                   ["pair(a,snd(^u))";"^u"]) THENL
                           [((TRANStac "pair(fst(^u),snd(^u))") THENL
                               [((SUBSTtac "A"
                                           ["(z)[pair(z,snd(^u))]";
                                            "(z)[pair(z,snd(^u))]";
                                            "(z)[@(A,B)]"]
                                            ["a";"fst(^u)"]) THENL
                                  [(EQelimtac THEN
                                      (INST_auto_tac THEN
                                         (VARintrtac THEN
                                            (EQformtac THENL
                                               [VARintrUistrat;
                                                VARintrstrat;
                                                ((PAIRelimitac "B") THEN
                                                    VARintrstrat)])))));
                                   (REFLtac THEN ResolvePAIRstrat)]);
                                (PAIReqpairtac THEN VARintrstrat)]);
                            (REFLtac THEN
                               (LISTintrconstac THENL
                                  [(VARintrtac THEN PAIRformstrat);
                                   (VARintrtac THEN
                                      LISTformstrat)])])])])])]));
          (FUNformtac THENL [Memberformstrat;
                             IDtac])]
        where ResolvePAIRstrat =
                (PAIRintrtac THENL
                    [VARintrstrat;
                    ((PAIRelim2tac "A") THEN
                        VARintrstrat)]))
;;

let
InducHypCasestrat =
       (VARintrtac THEN
          (UNIONformtac THENL
             [Memberformstrat;
              (FUNformtac THENL [Memberformstrat;
                                 IDtac])]))
```

```
;;

let
SuccessCasestrat u v n =
   ((UNIONintrtac 1) THENL
      [((SIGMAintrtac  ["cons(^u,fst(^n))"]) THENL
         [(LISTintrconstac THENL
                [VARintrstrat;(Solve_h_strat v)]);
          ((SIGMAintrtac ["fst(snd(^n))"]) THENL
                [(Solve_t_strat n v);
                 ((SIGMAintrtac ["fst(snd(snd(^n)))"]) THENL
                       [(Solve_b_strat n v);
                        (EQintrtac THEN
                           ((TRANStac "cons(^u,append(fst(^n),
                                                cons(pair(a,fst(snd(snd(^n)))),
                                                     fst(snd(^n)))))") THENL
                              [(LISTeqappendconstac THENL
                                 [VARintrstrat;
                                  (Solve_h_strat v);
                                  (LISTintrconstac THENL
                                     [(PAIRintrtac THENL
                                        [VARintrstrat;
                                         (Solve_b_strat n v)]);
                                      (Solve_t_strat n v)])]);
                               ((SUBSTtac "List(#(A,B))"
                                          ["(z)[cons(^u,z)]";
                                           "(z)[cons(^u,z)]";
                                           "(z)[List(#(A,B))]"]
                                          ["append(fst(^n),
                                              cons(pair(a,fst(snd(snd(^n)))),
                                                 fst(snd(^n))))";v]) THENL
                                  [(EQelimtac THEN
                                      (Bind_b_strat n v));
                                   (REFLtac THEN
                                      LISTintrconsstrat)])])])])]);
             (FUNformtac THENL [Memberformstrat;
                                IDtac])])
;;

let
FailureCasestrat  u v =
        ((UNIONintrtac 2) THENL
           [Memberformstrat;
            (Absurdity2strat u v)])
;;
```

```
let
NotEquCasestrat u v w =
     \g.((UNIONelimtac "(z)[+(Member(a,cons(^u,^v),A,B),
                              Not(Member(a,cons(^u,^v),A,B)))]"
                         [w]
                         "Member(a,^v,A,B)"
                         "Not(Member(a,^v,A,B))"
                         ["_1";"_2"]) THENL
                [InducHypCasestrat;
                 (SuccessCasestrat u v w);
                 (FailureCasestrat u v)]
         where
             a = (us "_1" g)) g
;;


let
BasisCasestrat  =
        ((UNIONintrtac 2) THENL
           [Memberformstrat;
            Absurdity1strat]
        )
;;


let
InducCasestrat u v w  =
        ((UNIONelimtac "(z)[+(Member(a,cons(^u,^v),A,B),
                              Not(Member(a,cons(^u,^v),A,B)))]"
                        []
                        "Eq(A,a,fst(^u))"
                        "Not(Eq(A,a,fst(^u)))"
                        []) THENL
          [(EquTeststrat u);
           (EquCasestrat u v);
           (NotEquCasestrat u v w)])
;;
```

```
let
LookUpstrat =
     \g.(PIintrtac THENL
           [VARintrUistrat;
            (PIintrtac THENL
               [LISTformstrat;
                ((LISTolintac "(1)[+(Member(a,1,A,B),Not(Member(a,1,A,B)))]"
                              "1"
                              "@(A,B)"
                              ["_1";"_2";"_3"]) THENL
                   [VARintrstrat;
                    BasisCasestrat;
                    (InducCasestrat u v w)])])]
                where u = (un "_1" g)
                and   v = (un "_2" g)
                and   w = (un "_3" g)) g
;;
```

B.3 Proof scripts

The proof script generated interactively in TTPA is too large to include in its entirety. Selective parts of the proof script are therefore included. These "snapshots" are theorems which correspond to the nodes in the strategy hierarchy presented in chapter 4 (figure 4.1). Difficulties with the GTTS definition mechanism led to the definitions for Not and Member being introduced by hand.

Proof script for absurdity 1

```
%-------------------------------------------------------
      Absurdity1strat
-------------------------------------------------------%
6 2
"lambda((u15)
         [apply(listrec(fst(u15),
                        lambda((b12)nil),
                        (u16,u17,u18)[lambda((b16)nil)]),
                and(and(and(u15)))])
 : Not(Member(a,nil,A,B))
 [A : U1; B : U1; a : A]"
(proved)() : .

%-------------------------------------------------------
      Absurdity1Basisstrat "u15"
-------------------------------------------------------%
6 2 3
"lambda((b12)nil)
 : Not(Eq(List(#(A,B)),
          append(nil,cons(pair(a,fst(and(and(u15)))),fst(and(u15)))),
          nil))
 [A : U1;
  B : U1;
  a : A;
  u15 : Member(a,nil,A,B)]"
(proved)() : .
```

```
%-------------------------------------------------------
      Absurdity1Inducstrat "u15" "u16" "u17"
-------------------------------------------------------%
6 2 4
"lambda((b16)nil)
 : Not(Eq(List(#(A,B)),
          append(cons(u16,u17),
                 cons(pair(a,fst(snd(snd(u15)))),fst(snd(u15)))),
          nil))
 [A : U1;
  B : U1;
  u16 : #(A,B);
  u17 : List(#(A,B));
  a : A;
  u15 : Member(a,nil,A,B)]"
(proved)() : .
```

Proof script for absurdity 2

```
%-------------------------------------------------------
      Absurdity2strat "u10" "u11"
-------------------------------------------------------%
7 3 3 2
"lambda((u19)
         [when(apply(listrec(fst(u19),
                             lambda((b22)[i(e)]),
                             (u20,u21,u22)
                              [lambda((b23)
                                       [j
                                         (pair
                                           (u21,
                                            pair
                                             (fst(snd(u19)),
                                              pair(fst(snd(snd(u19))),e)))))]])),
                     snd(snd(snd(u19)))),
               (b20)[apply(b11,b20)],
               (b21)[apply(u14,b21)]])]
 : Not(Member(a,cons(u10,u11),A,B))
 [A : U1;
  B : U1;
  a : A;
  u10 : #(A,B);
  u11 : List(#(A,B));
  b11 : Not(Eq(A,a,fst(u10)));
  u14 : Not(Member(a,u11,A,B))]"
(proved)() : .

%-------------------------------------------------------
     Equ_or_Mem_strat "u10" "u11" "u19"
-------------------------------------------------------%
7 3 3 2
"apply(listrec(fst(u19),
               lambda((b22)[i(e)]),
               (u20,u21,u22)
                [lambda((b23)
                         [j(pair(u21,
                                 pair(fst(snd(u19)),
                                      pair(fst(snd(snd(u19))),e)))))]]),
      snd(snd(snd(u19))))
 : +(Eq(A,a,fst(u10)),
     Member(a,u11,A,B))
 [A : U1;
  B : U1;
  a : A;
  u10 : #(A,B);
  u11 : List(#(A,B));
  u19 : Member(a,cons(u10,u11),A,B)]"
```

```
(proved)() : .

%-----------------------------------------------------------
      a_equ_fst_u_strat "u10" "u11" "u19"
-----------------------------------------------------------%
7 3 3 2 2 13
"e : Eq(A,a,fst(u10))
 [A : U1;
  a : A;
  B : U1;
  u10 : #(A,B);
  u11 : List(#(A,B));
  u19 : Member(a,cons(u10,u11),A,B);
  b22
  : Eq(List(#(A,B)),
       append(nil,cons(pair(a,fst(snd(snd(u19)))),fst(snd(u19)))),
       cons(u10,u11))]"
(proved)() : .

%-----------------------------------------------------------
      Mem_a_v_strat "u19" "u10" "u11" "u20" "u21"
-----------------------------------------------------------%
7 3 3 2 2 32
"pair(u21,pair(fst(snd(u19)),pair(fst(snd(snd(u19))),e)))
 : Member(a,u11,A,B)
 [A : U1;
  B : U1;
  u21 : List(#(A,B));
  a : A;
  u10 : #(A,B);
  u11 : List(#(A,B));
  u19 : Member(a,cons(u10,u11),A,B);
  u20 : #(A,B);
  b23
  : Eq(List(#(A,B)),
       append(cons(u20,u21),
              cons(pair(a,fst(snd(snd(u19)))),fst(snd(u19)))),
       cons(u10,u11))]"
(proved)() : .
```

```
%-----------------------------------------------------------
   append_equ_cons_strat "u19" "u10" "u11" "u20" "u21"
-----------------------------------------------------------%
7 3 3 2 2 32 5
"append(u21,cons(pair(a,fst(snd(snd(u19)))),fst(snd(u19))))
 = listrec(cons(u20,
                append(u21,cons(pair(a,fst(snd(snd(u19)))),fst(snd(u19))))),
           u11,
           (x3,y3,z3)y3)
 : List(#(A,B))
 [A : U1;
  B : U1;
  u21 : List(#(A,B));
  a : A;
  u10 : #(A,B);
  u11 : List(#(A,B));
  u19 : Member(a,cons(u10,u11),A,B);
  u20 : #(A,B)]"
(proved)() : .

%-----------------------------------------------------------
   cons_equ_append_strat "u19" "u10" "u11" "u20" "u21"
-----------------------------------------------------------%
7 3 3 2 2 32 6
"listrec(cons(u20,
              append(u21,cons(pair(a,fst(snd(snd(u19)))),fst(snd(u19))))),
         u11,
         (b27,b28,b29)b28)
 = u11
 : List(#(A,B))
 [A : U1;
  B : U1;
  u20 : #(A,B);
  u21 : List(#(A,B));
  a : A;
  u10 : #(A,B);
  u11 : List(#(A,B));
  u19 : Member(a,cons(u10,u11),A,B);
  b28
  : Eq(List(#(A,B)),
       append(cons(u20,u21),
              cons(pair(a,fst(snd(snd(u19)))),fst(snd(u19)))),
       cons(u10,u11))]"
(proved)() : .
```

```
%-----------------------------------------------------------
      EquNotEquAbsurdstrat "u10"
-----------------------------------------------------------%
7 3 3 2 3
"apply(b11,b20) : Null
 [A : U1;
  a : A;
  B : U1;
  u10 : @(A,B);
  b11 : Not(Eq(A,a,fst(u10)));
  b20 : Eq(A,a,fst(u10))]"
(proved)() : .

%-----------------------------------------------------------
      MemNotMemAbsurdstrat "u11"
-----------------------------------------------------------%
7 3 3 2 4
"apply(u14,b21) : Null
 [A : U1;
  B : U1;
  a : A;
  u11 : List(@(A,B));
  u14 : Not(Member(a,v11,A,B));
  b21 : Member(a,v11,A,B)]"
(proved)() : .
```

Proof script for main proof

```
%-------------------------------------------------------
      LookUpstrat
-------------------------------------------------------%
"lambda((a)
      [lambda((l)
            [listrec(l,
                     j(lambda((u15)
                              [apply
                                 (listrec
                                   (fst(u15),
                                    lambda((b12)nil),
                                    (u16,u17,u18)[lambda((b16)nil)]),
                                  snd(snd(snd(snd(u15)))))]),
                      (u10,u11,u12)
                       [when(apply(apply(equ,a),fst(u10)),
                             (b10)
                              [i(pair(nil,pair(u11,pair(snd(u10),e))))],
                             (b11)
                             [when
                                (u12,
                                (u13)
                                  [i
                                    (pair
                                      (cons(u10,fst(u13)),
                                       pair
                                       (fst(snd(u13)),
                                         pair(fst(snd(snd(u13))),e))))],
                                (u14)
                                 [j
                                    (lambda
                                      ((u19)
                                        [when
                                          (apply
                                            (listrec
                                              (fst(u19),
                                               lambda((b22)[i(e)]),
                                               (u20,u21,u22)
                                                [lambda
                                                  ((b23)
                                                    [j
                                                      (pair
                                                       (u21,
                                                         pair
                                                          (fst
                                                            (snd(u19)),
                                                           pair
                                                            (fst
                                                              (snd
```

```
                                                                    (snd
                                                                      (u10))),
                                                                  e)))))])]),
                                                    snd(snd(snd(u19)))),
                                                  (b20)[apply(b11,b20)],
                                                  (b21)[apply(u14,b21)])])])])])])])])
  : Pi(A,
       (a)
        [Pi(List(0(A,B)),
            (l)
             [+(Member(a,l,A,B),
                Not(Member(a,l,A,B)))])])
  [A : U1;
   B : U1;
   equ : Pi(A,(x)[Pi(A,(y)[+(Eq(A,x,y),Not(Eq(A,x,y)))])])]*
(proved)() : .

%-------------------------------------------------------
      BasicCasestrat
-------------------------------------------------------%
6
*j(lambda((u15)
           [apply(listrec(fst(u15),
                          lambda((b12)nil),
                          (u16,u17,u18)[lambda((b16)nil)]),
                  snd(snd(snd(u15))))])])
  : +(Member(a,nil,A,B),Not(Member(a,nil,A,B)))
  [A : U1; B : U1; a : A]*
(proved)() : .

%-------------------------------------------------------
      InducCasestrat "u10" "u11" "u12"
-------------------------------------------------------%
7
*when(apply(apply(equ,a),fst(u10)),
      (b10)[i(pair(nil,pair(u11,pair(snd(u10),e))))],
      (b11)
       [when(u12,
             (u13)
              [i(pair(cons(u10,fst(u13)),
                      pair(fst(snd(u13)),pair(fst(snd(snd(u13))),e))))],
             (u14)
              [j(lambda((u19)
                        [when(apply(listrec
                                    (fst(u19),
                                     lambda((b22)[i(e)]),
                                     (u20,u21,u22)
                                      [lambda
                                        ((b23)
```

```
                                                    [j
                                                      (pair
                                                        (u21,
                                                         pair
                                                          (fst(snd(u19)),
                                                           pair
                                                            (fst(snd(snd(u19))),
                                                             e)))))]]),
                                              snd(snd(snd(u19)))),
                                        (b20)[apply(b11,b20)],
                                        (b21)[apply(u14,b21)]])))))])
  : +(Member(a,cons(u10,u11),A,B),Not(Member(a,cons(u10,u11),A,B)))
  [A : U1;
   equ : Pi(A,(x)[Pi(A,(y)[+(Eq(A,x,y),Not(Eq(A,x,y)))])]);
   a : A;
   B : U1;
   u10 : #(A,B);
   u11 : List(#(A,B));
   u12 : +(Member(a,u11,A,B),Not(Member(a,u11,A,B)))]"
 (proved)() : .

%-------------------------------------------------------
      EquTeststrat "u10"
-------------------------------------------------------%
7 1
"apply(apply(equ,a),fst(u10))
 : +(Eq(A,a,fst(u10)),Not(Eq(A,a,fst(u10))))
 [A : U1;
  equ : Pi(A,(x)[Pi(A,(y)[+(Eq(A,x,y),Not(Eq(A,x,y)))])]);
  a : A;
  B : U1;
  u10 : #(A,B)]"
 (proved)() : .

%-------------------------------------------------------
      EquCasestrat "u10" "u11"
-------------------------------------------------------%
7 2
"i(pair(u11,pair(u11,pair(snd(u10),e))))
 : +(Member(a,cons(u10,u11),A,B),Not(Member(a,cons(u10,u11),A,B)))
 [A : U1;
  B : U1;
  u11 : List(#(A,B));
  u10 : #(A,B);
  a : A;
  b10 : Eq(A,a,fst(u10))]"
 (proved)() : .
```

```
%-------------------------------------------------------
      NotEquCasestrat "u10" "u11" "u12"
-------------------------------------------------------%
7 3
"when(u12,
      (u13)
       [i(pair(cons(u10,fst(u13)),
               pair(fst(snd(u13)),pair(fst(snd(snd(u13))),e)))))],
      (u14)
       [j(lambda((u19)
                  [when(apply(listrec(fst(u19),
                                      lambda((b22)[i(e)]),
                                      (u20,u21,u22)
                                       [lambda
                                         ((b23)
                                           [j
                                             (pair
                                               (u21,
                                                pair
                                                 (fst(snd(u19)),
                                                  pair
                                                   (fst(snd(snd(u19))),
                                                    e)))))]])),
                                snd(snd(snd(u19)))),
                        (b20)[apply(b11,b20)],
                        (b21)[apply(u14,b21)]])])])
 : +(Member(a,cons(u10,u11),A,B),Not(Member(a,cons(u10,u11),A,B)))
 [A : U1;
  B : U1;
  a : A;
  u11 : List(@(A,B));
  u12 : +(Member(a,u11,A,B),Not(Member(a,u11,A,B)));
  u10 : @(A,B);
  b11 : Not(Eq(A,a,fst(u10)))]"
(proved)() : .

%-------------------------------------------------------
      InducHypCasestrat
-------------------------------------------------------%
7 3 1
"u12
 : +(Member(a,u11,A,B),Not(Member(a,u11,A,B)))
 [A : U1;
  B : U1;
  a : A;
  u11 : List(@(A,B));
  u12 : +(Member(a,u11,A,B),Not(Member(a,u11,A,B)))]"
(proved)() : .
```

```
%-----------------------------------------------------------
      SuccessCasestrat "u10" "u11" "u13"
-----------------------------------------------------------%
7 3 2
"i(pair(cons(u10,fst(u13)),pair(fst(snd(u13)),pair(fst(snd(snd(u13))),e))))
 : +(Member(a,cons(u10,u11),A,B),Not(Member(a,cons(u10,u11),A,B)))
 [A : U1;
  B : U1;
  u10 : #(A,B);
  a : A;
  u11 : List(#(A,B));
  u13 : Member(a,u11,A,B)]"
(proved)() : .

%-----------------------------------------------------------
      FailureCasestrat "u10" "u11"
-----------------------------------------------------------%
7 3 3
"j(lambda((u19)
          [when(apply(listrec(fst(u19),
                              lambda((b22)[i(e)]),
                              (u20,u21,u22)
                               [lambda
                                 ((b23)
                                   [j(pair
                                       (u21,
                                        pair
                                         (fst(snd(u19)),
                                          pair(fst(snd(snd(u19))),e)))))]])),
                      snd(snd(snd(u19)))),
                (b20)[apply(b11,b20)],
                (b21)[apply(u14,b21)]])))
 : +(Member(a,cons(u10,u11),A,B),Not(Member(a,cons(u10,u11),A,B)))
 [A : U1;
  B : U1;
  a : A;
  u10 : #(A,B);
  u11 : List(#(A,B));
  b11 : Not(Eq(A,a,fst(u10)));
  u14 : Not(Member(u11))]"
(proved)() : .
```

References

[Backhouse 85] R.C. Backhouse. Algorithm Development in Martin-Löf's Type Theory. Dept. of Computer Science, University of Essex, 1985.

[Backhouse 86a] R.C. Backhouse. Notes on Martin-Löf's Theory of Types, parts 1 and 2. *FACS FACTS*, 1986.

[Backhouse 86b] R.C. Backhouse. *On the Meaning and Construction of the Rules in Martin-Löf's Theory of Types.* Computing Science Notes CS 8606, Dept. of Mathematics and Computing Science, University of Groningen, 1986.

[Backhouse 87] R.C. Backhouse. Overcoming the mismatch between programs and proofs. In P.Dybjer, B. Nordström, K. Petersson, and J.M. Smith, editors, *Proceedings of the Workshop in Programming Logic*, pages 116–122, Marstrand, June 1987.

[Backhouse *et al* 89] R.C. Backhouse, P. Chisholm, and G. Malcolm. Do–it–yourself Type Theory. *Formal Aspects of Computing*, 1(1), 1989.

[Bates & Constable 83] J.L. Bates and R.L. Constable. *The Nearly–Ultimate PRL.* Technical Report, TR 83 551, Dept. of Computer Science, Cornell University, Ithaca, NY, 1983.

[Bates 79] J.L. Bates. *A Logic for Correct Program Development.* PhD thesis, Dept. of Computer Science, Cornell University, 1979.

[Beeson 85] M.J. Beeson. *Foundations of Constructive Mathematics.* Springer-Verlag, 1985.

[Bishop 67] E. Bishop. *Foundations of Constructive Analysis.* McGraw-Hill, 1967.

[Boyer & Moore 79] R.S. Boyer and J.S. Moore. *A Computational Logic.* Academic Press, 1979.

[Brouwer 75] L.E.J. Brouwer. *Collected Works.* Editor, A. Heyting, Volume 1, North-Holland, Amsterdam, 1975.

[Brouwer 81] L.E.J. Brouwer. *Brouwer's Cambridge Lectures on Intuitionism.* Editor, D. van Dalen, Cambridge University Press, Cambridge, 1981.

[Burstall & Darlington 77] R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.

[Burstall & Ritchie 86] R. Burstall and B. Ritchie. Edinburgh Interactive Proof Editor. Dept. of Computer Science, University of Edinburgh, 1986.

[Chisholm 87] P. Chisholm. Derivation of a Parsing Algorithm in Martin-Löf's Theory of Types. *Science of Computer Programming*, 8:1–42, 1987.

[Constable & Mendler 85] R.L. Constable and N.P. Mendler. Recursive Definitions in Type Theory. In *Proceedings of Logics of Programs Conference*, pages 61–78, Springer-Verlag LNCS 193, 1985.

[Constable & Smith 87] R.L. Constable and S.F. Smith. Partial Objects in Constructive Type Theory. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 183–193, Computing Society Press of the IEEE, 1987.

[Constable & Zlatin 84] R.L. Constable and D.R. Zlatin. The Type Theory of PL/CV3. *ACM Transactions on Programming Languages and Systems*, 6(1):97–117, 1984.

[Constable 71] R.L. Constable. Constructive Mathematics and Automatic Program Writers. In *Proceedings of IFIP Congress*, pages 229–233, Ljubljana, 1971.

[Constable 85] R.L. Constable. Constructive Mathematics as a Programming Logic 1: Some Principles of Theory. *Annals of Discrete Mathematics*, 24:21–38, 1985.

[Constable *et al* 82] R.L. Constable, S.D. Johnson, and C.D. Eichenlaub. *Introduction to the PL/CV2 Programming Logic.* Springer-Verlag LNCS 135, 1982.

[Constable *et al* 86] R.L. Constable *et al.* *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[Coquand & Huet 85] T. Coquand and G. Huet. Constructions: a higher order proof system for mechanising mathematics. In *Proceedings of EUROCAL 85*, Linz, Austria, April 1985.

[Curry & Feys 58] H.B. Curry and R. Feys. *Combinatory Logic*. Volume 1, North-Holland, 1958.

[de Bruijn 80] N.G. de Bruijn. A survey of the project AUTOMATH. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606, Academic Press, 1980.

[Dijkstra 68] E.W. Dijkstra. A Constructive Approach to the Problem of Program Correctness. *Bit*, 8:174–186, 1968.

[Dijkstra 72] E.W. Dijkstra. Notes on Structured Programming. In *Structured Programming*, O.-J. Dahl, E.W. Dijkstra and C.A.R. Hoare, pages 1–82, Academic Press, 1972.

[Dijkstra 76] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[Dummett 77] M. Dummett. *Elements of Intuitionism. Oxford Logic Series*, Clarendon Press, Oxford, 1977.

[Dyckhoff 85] R. Dyckhoff. *Category Theory as an Extension of Martin-Löf Type Theory*. Technical Report CS/85/3, Dept. of Computational Science, University of St. Andrews, 1985.

[Dyckhoff 87] R. Dyckhoff. Strong elimination rules in Type Theory. In P.Dybjer, B. Nordström, K. Petersson, and J.M. Smith, editors, *Proceedings of the Workshop in Programming Logic*, pages 112–115, Marstrand, June 1987.

[Dyckhoff 88] R. Dyckhoff. MALT (Machine Assisted Logic Teaching). *Computerised Logic Teaching Bulletin*, 1(1), March 1988.

[Floyd 67] R.W. Floyd. Assigning Meanings to Programs. In *Mathematical Aspects of Computer Science*, pages 19–32, American Mathematical Society, 1967.

[Floyd 79] R.W. Floyd. The paradigms of programming. *Communications of the ACM*, 22(8):455–460, 1979.

[Gentzen 69] G. Gentzen. Investigations into logical deduction. In M.E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–213, North-Holland, Amsterdam, 1969.

[Good 85] D.I. Good. Mechanical proofs about computer programs. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, chapter 3, pages 55–75, Prentice-Hall, 1985.

[Gordon 83] M.J.C. Gordon. *LCF_LSM: A system for specifying and verifying hardware.* Report 41, Computer Laboratory, University of Cambridge, 1983.

[Gordon *et al* 79] M.J.C. Gordon, R. Milner, and C.P. Wadsworth. *Edinburgh LCF.* Springer-Verlag LNCS 78, 1979.

[Green & Barstow 78] C. Green and D.R. Barstow. On Program Synthesis Knowledge. *Artificial Intelligence*, 10:241–279, 1978.

[Gries 81] D. Gries. *The Science of Programming.* Springer-Verlag, 1981.

[Hamilton 85] A.G. Hamilton. *Program Construction in Martin-Löf Type Theory.* Technical Report, T.R. 24, Dept. of Computing Science, University of Stirling, June 1985.

[Hamilton 89] G.W. Hamilton. *A User Interface to a Proof Assistant using X-Windows.* B.Sc. Dissertation, Dept. of Computing Science, University of Stirling, May 1989.

[Harper 85] R. Harper. *Aspects of the Implementation of Type Theory.* PhD thesis, Dept. of Computer Science, Cornell University, 1985.

[Heyting 56] A. Heyting. *Intuitionism: an Introduction.* North-Holland, Amsterdam, 1956.

[Hoare 72] C.A.R. Hoare. Notes on Data Structuring. In *Structured Programming*, O.-J. Dahl, E.W. Dijkstra and C.A.R. Hoare, pages 83–174, Academic Press, 1972.

[Howard 80] W.A. Howard. The formulas-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus, and Formalism*, pages 479–490, Academic Press, 1980.

[Igarashi *et al* 75] S. Igarashi, R.L. London, and D.C. Luckham. Automatic Program Verification 1: A Logical Basis and its Implementation. *ACTA Informatica*, 4:145–182, 1975.

[Khamiss 86] A.M.A. Khamiss. *Program Construction in Martin-Löf's Theory of Types.* PhD thesis, Dept. of Computer Science, University of Essex, 1986.

[Malcolm & Chisholm 88] G. Malcolm and P. Chisholm. *Polymorphism and Information Loss in Martin-Löf's Type Theory.* Computing Science Notes, CS 8814, Dept. of Mathematics and Computing Science, University of Groningen, 1988.

[Manna & Waldinger 80] Z. Manna and R. Waldinger. A Deductive Approach to Program Synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[Martin-Löf 75] P. Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73*, pages 73–118, North-Holland, 1975.

[Martin-Löf 82] P. Martin-Löf. Constructive Mathematics and Computer Programming. In L.J. Cohen, J. Loś, H. Pfeiffer, and K-P. Podewski, editors, *Logic, Methodology and Philosophy of Science VI*, pages 153–175, North-Holland, 1982.

[Martin-Löf 84] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[McCarthy & Painter 66] J. McCarthy and J. Painter. *Correctness of a Compiler for Arithmetic Expressions.* Technical Report No. CS38, Dept. of Computer Science, Stanford University, April 1966.

[Milner 78] R. Milner. A Theory of Type Polymorphism in Programming Languages. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Milner 85] R. Milner. The use of machines to assist in rigorous proof. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, chapter 4, pages 77–88, Prentice-Hall, 1985.

[Milner 87] R. Milner. Dialogue with a Proof System. In *TAPSOFT '87*, pages 271–275, Springer-Verlag LNCS 250, Pisa, Italy, 1987.

[Naur 66] P. Naur. Proof of Algorithms by General Snapshots. *BIT*, 6:310–316, 1966.

[Nordström & Petersson 83] B. Nordström and K. Petersson. Types and Specifications. In R.E. Mason, editor, *IFIP '83*, pages 915–920, Elsevier Science Publishers, 1983.

[Nordström 85] B. Nordström. Multilevel functions in type theory. In N. Jones, editor, *Programs as Data Objects*, Springer-Verlag LNCS 217, 1985.

[Nordström *et al* 86] B. Nordström, K. Petersson, and J. Smith. *An Introduction to Martin-Löf's Theory of Types*. Technical Report, Programming Methodology Group, University of Göteborg/Chalmers, August 1986.

[Paulson 83] L.C. Paulson. *Tactics and tacticals in Cambridge LCF*. Report 36, Computer Laboratory, University of Cambridge, 1983.

[Paulson 85] L.C. Paulson. Lessons learned from LCF: A Survey of Natural Deduction Proofs. In D. Bjørner, editor, *Workshop on Formal Software Development: Combining Specification Methods*, Springer-Verlag, 1985.

[Paulson 86] L.C. Paulson. Natural Deduction as Higher-Order Resolution. *The Journal of Logic Programming*, 3:237–258, 1986.

[Petersson & Smith 86] K. Petersson and J.M. Smith. Program derivation in type theory: a partitioning problem. *Computing Languages*, 11(3/4):161–172, 1986.

[Petersson 82] K. Petersson. *A programming system for type theory*. LPM Memo 21, Department of Computer Science, Chalmers University of Technology, Göteborg, 1982.

[Sannella & Burstall 83] D.T. Sannella and R.M. Burstall. Structured theories in LCF. In *CAAP '83*, pages 377–391, Springer-Verlag LNCS 159, 1983.

[Scherlis & Scott 83] W. Scherlis and D. Scott. First Steps Towards Inferential Programming. In R.E. Mason, editor, *IFIP '83*, pages 199–212, Elsevier Science Publishers, 1983.

[Schmidt 84] D. Schmidt. A programming notation for tactical reasoning. In *Proceedings of Seventh International Conference on Automated Deduction*, pages 445–459, Springer-Verlag LNCS 170, 1984.

[Scott 70] D. Scott. Constructive Validity. In *Symposium on Automatic Demonstration*, pages 237–275, Springer-Verlag LNCS 125, 1970.

[Smith 83] J. Smith. The identification of propositions and types in Martin-Löf's Type Theory: a programming example. In *Foundations of Computation Theory*, pages 445–456, Springer-Verlag LNCS 158, 1983.

[Smith 87] J. Smith. On a Nonconstructive Type Theory and Program Derivation. In D.G. Skordev, editor, *Mathematical Logic and its Applications*, pages 331–340, Plenum Publishing Corporation, 1987.