Specification and implementation of a unification algorithm in Martin-Löf's type theory

Neil Leslie

Submitted for the Degree of Master of Science

University of St Andrews

Department of Mathematical and Computational Sciences,
Computational Science Division

September 1992

Abstract

Unification is one of the most important notions in computational science and artificial intelligence. Martin-Löf's type theory provides a framework for the specification, implementation and execution of provably correct algorithms. In this thesis we use Martin-Löf's type theory to specify, implement and prove correct a unification algorithm. We represent the terms to be unified using mutually recursive types. Our unification algorithm uses well-founded recursion. We address the issues that arise from using well-founded induction to prove the total correctness of a collection of mutually recursive functions in Martin-Löf's type theory.

I, Neil Leslie, hereby certify that this thesis has been composed by myself, that it is a record of my own work, and that it has not been accepted in partial or complete fulfilment of any other degree or professional qualification.

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker.

Acknowledgments

Notwithstanding the preceding declaration this thesis would not have been produced without Roy Dyckhoff and Mhairi McHugh. Roy introduced me to (as he would have it) Martin-Löf type theory. Without him this thesis would not have been started. Mhairi gave me support and encouragement throughout. Without her this thesis would never have been finished.

Contents

Chapter 1

Introduction

The software engineer is faced with a variety of criteria to meet when asked to produce a piece of software. Of paramount importance is that the software does what it is supposed to do. Concerns about speed, portability, re-usability and so on are secondary to this concern. This thesis investigates the construction of a correct unification algorithm.

Unification concerns answering the question of under what circumstances two terms are the same, and is one of the most important notions in computational science. It lies at the heart of many automated reasoning systems and is the core of many mechanical natural language systems. The unification algorithm for simple terms has technical interest as it is not naturally expressed in structurally recursive form. Hence we can expect the task of proving that the algorithm terminates to be non-trivial.

Formally we say that a program is *correct* if it meets its specification. For software development we see that we need a language to write programs in, a language to write specifications in and a logical system which will allow us to prove that programs meet their specifications (or, alternatively, that some specification can be met). There are a number of formal systems in which we might choose to do this. Most of them use three separate languages to express the algorithm, the specification and the correctness proof.

Martin-Löf's Type Theory (M-LTT) was developed as a contribution to the foundations of mathematics. The theory has been developed over a period of time and there are several variations which differ in small but important ways. We shall be using the theory as presented in [Mar84] which uses an extensional equality and lazy evaluation. The usual semantics for M-LTT is an operational one based on our understanding of the notion of a *method*. Because of this, and because the theory has been very carefully developed, we can use it to express functions, specifications and correctness proofs. These correspond to elements, types and proofs in the purely mathematical interpretation of the theory.

M-LTT is a theory in the intuitionistic (or constructive) tradition. Although we believe that constructivism has a privileged status amongst the differing philosophies of mathematics the utility of M-LTT for the computational scientist is not dependent on any such beliefs.

1.1 Overview of this thesis

This thesis develops in the following way.

Chapter 2 introduces the notion of unification and explains why it is of such great importance in computational science. Some basic notions relating to substitutions are introduced and some observations made about these.

Chapter 3 introduces M-LTT in more depth. We present the theory of expressions and the related notion of computation rules, the theory of judgements and the forms of judgement that are required in type theory. We present rules for some useful types and explain the relationship between M-LTT and computer programming.

In Chapter 4 we review work done by others on proving the correctness of unification algorithms.

In Chapter 5 we show how to represent terms in M-LTT. We do this in two ways: directly using two mutually recursive types and indirectly using tree types. We show that these are equivalent and make some comments about mutually recursive functions in M-LTT.

In Chapter 6 we investigate the use of well-founded recursion in M-LTT. In particular we investigate how to express mutually recursive functions using well-founded recursion. We present an ordering on the type of pairs of terms that we introduced in Chapter 5 and prove that it is a well-founded ordering.

In Chapter 7 we investigate different possible specifications for the algorithm. We investigate how specifications for mutually recursive functions must be written.

In Chapter 8 we investigate different unification algorithms. We look both at algorithms which have been presented elsewhere and at our own suggestions.

In Chapters 9 and 10 we use well-founded induction to prove that one of the algorithms presented in Chapter 8 meets one of the specifications presented in Chapter 7. In Chapter 9 we present some useful lemmas and prove that the algorithm terminates. In Chapter 10 we prove that the algorithm is correct.

Chapter 11 presents some comments on the work, with suggestions for how to improve and extend it.

Chapter 12 presents our conclusions.

1.2 Typesetting

We have attempted to place rules, proofs and functions in the text where they are referred to. Due to the size of some of the figures this

has not always been possible, and they have been placed at the end of the appropriate chapter. Some of the figures proved to be of a complexity which required that some details be suppressed.

Chapter 2

Unification

2.1 Introduction

Unification theory is concerned with solving the following problem:

> Suppose s and t to be a pair of terms. Under what circumstances are s and t the same term?

For example, suppose that a, b, c and d are constants, x and y variables. As we take the constants to be distinct we see that there are no circumstances under which a and b will be the same term. Suppose s and t are $a(x, b(d, x))$ and $a(c, y)$. We see that if we replace x by c and y by $b(d, c)$ in s and t then they both become $a(c, b(d, c))$. We call a substitution which makes two terms the same a *unifier* and we say that the terms are *unifiable*. We now start to think of the properties that a unifier might have: 'Is the unifier of two terms unique?', 'If unifiers are not unique, then is there a special unifier for unifiable terms?' Before we can start to answer these questions we must make clearer our notions of terms and of substitutions, which we do in this chapter. In Chapter 7 we will reformulate these notions in type theory, but for our current purposes we will leave some details unspecified. We also explain why answering the above question is so important to computer science, recap some history and see why the simple question that we started with is such an interesting one to answer.

2.2 Terms

A *term* is either a variable or the application of a functor to a (possibly empty) list of arguments, which are themselves terms. The variables and the functors are distinct. We will use the letters x, y, z and so on for variables, f, g, h and so on for the functors.

We think that two terms are the same if they are the same variable or if they are the same functor applied to the same list of terms. If we had a different notion of equality between terms we would have a different notion of unification. [Sie89] discusses unification in situations where various additions to the notion of equality are allowed. In unification based approaches to natural language, such as that presented in [Shi86] and [GM89], unification is thought of as being over open-ended directed, acyclic graphs (called 'feature structures') and hence the terms f and $f(x)$ are considered unifiable. Neither of the two mentioned texts makes clear the notion of substitution involved.

2.3 Substitutions

A *substitution* is a function from variables to terms. We will use the lower-case Greek letters σ, τ, υ and so on for substitutions. If x is a variable and t a term then we may write x --> t for the substitution which replaces x by t and leaves all other variables unchanged.

The *support* (or *domain*) of substitution is the set of variables that are changed by the substitution. All the substitutions that we shall be concerned with shall have finite support. The *range variables* of a substitution is the set of variables which occur in the set of terms generated by applying the substitution to its support.

If σ is a substitution then its *extension* σ^* is the function from terms to terms which replaces each occurrence of a variable in a term by σ applied to that variable.

The *composition* of two substitutions σ and τ, $\sigma \bullet \tau$, is $(x)(\tau^* (\sigma\ x))$.

One substitution τ is an *instance* of another σ if there is a substitution ν such than $\tau = \sigma \bullet \nu$. In this case σ is said to be *more general* than τ. Notice that, as two substitutions may be instances of each other, the phrase 'more general than' is slightly misleading.

A *unifier* of two terms s and t is a substitution σ such that $\sigma^* s = \sigma^* t$.

A *most general unifier* of two terms s and t is a unifier of s and t, of which all other unifiers are instances.

An *idempotent* substitution σ is one such that $\sigma = \sigma \bullet \sigma$. We will show later that the domain and range variables of an idempotent substitution are disjoint.

2.4 Initial observations

Clearly there is not a unique unifier for any pair of unifiable terms s and t. If σ is a unifier of s and t then so is any instance of σ.

Nor need there be a unique most general unifier. Any pair of variables, x and y, will be unified by the two substitutions x --> y and y--> x. These substitutions are instances of each other and are both most general unifiers of x and y.

We shall see later that the property of idempotence of unifiers is crucial. We point out here that a most general unifier need not be idempotent. Manna and Waldinger ([MW81]) give the example of the pair of terms $g(x, z)$ and $g(y, f(y))$. A most general unifier for these terms is z -->$f(z) \bullet x$ --> $z \bullet y$ --> z, which is not idempotent.

We now see that the original question can be reformulated as 'If a pair of terms is unifiable is there a most general unifier?' In fact when we come to try to answer this question we see that we are forced to answer the question 'If a pair of terms is unifiable is there a most general idempotent unifier?' Later we shall consider whether it is best to answer this question directly as stated or to answer some logically equivalent question.

2.5 The history and importance of unification

Robinson ([Rob92]) and Siekmann ([Sie89]) provide surveys of the history and importance of unification. The landmark paper in the field is Robinson's [Rob65]. Work before this paper may be called the pre-history of unification. The main contributions are those of Herbrand ([Her71]) and Prawitz ([Pra60]). The word 'unification' was coined by Robinson in [Rob65] for the procedure described in [Pra60].

In his thesis ([Her71], originally published in 1930) Herbrand presents an algorithm to find a proof of a proposition in first-order logic (in the case that such a proof exists, of course). In the course of this he introduces the notion of unification, as [Rob92] states "albeit only in a rather brief and obscure passage".

Prawitz ([Pra60]) discusses a mechanisable proof procedure for first order logic. Again as a part of this procedure the notion of unification is introduced. Unification allows us to reduce the number of possible cases of instantiation of variables that we must deal with, and therefore is a step towards making the problem tractable. Prawitz' algorithm was implemented on an electronic digital computer.

Davis and Putnam ([DP60]) drew attention to a clausal predicate logic. Robinson ([Rob65]) combined the ideas in [DP60] and [Pra60] and introduced a clausal logic with a single inference schema which combines the Prawitz procedure with Gentzen's Cut rule. This was called *resolution*.

Chang and Lee ([CL73]) discuss some of these procedures in more detail.

We see that the pre-history of unification was concerned with mechanisable theorem proving. The purpose of using unification was to improve the efficiency of proof procedures for first-order logic. Unification has continued to lie at the heart of mechanical theorem proving and has been described as 'the addition and subtraction of automated reasoning'. Unification is, however, important in many other contexts. Siekmann ([Sie89]) gives a listing of areas of computing science and artificial intelligence where unification problems arise which includes databases, natural language processing (NLP), expert systems, knowledge representation, logic programming and automated deduction systems. Much work on NLP (see, for example, [Shi86]) relies on unification as the sole information combining mechanism. Unification is also important to polymorphic type-checking algorithms used in modern functional programming languages ([Mil78], [Pey87]). Resolution theorem proving can be seen as the basis for the logic programming language Prolog (see, for example, [SS86]) which has, in

turn, been used as a vehicle to implement many other systems which exploit unification.

Given that unification is such an important procedure it is important that it be implemented correctly. It is worth noting that most Prolog implementations deliberately implement unification incorrectly for reasons of efficiency, specifically, the 'occurs check' is omitted. Many textbooks present a unification algorithm. Norvig ([Nor91]) points out that in at least seven LISP textbooks the presentation given is incorrect, failing to unify $f(x, y)$ and $f(y, x)$. The errors usually occur for reasons of efficiency (not connected with the omission of the 'occurs check') connected with the way that terms and substitutions are represented. Norvig points out that:

> Curiously, this note shows that for unification, the functional approach is error-prone, whereas the procedural state-modification approach tends to lead to a correct solution.

We agree that this is indeed a curiosity.

Chapter 3

Martin-Löf's Type Theory

3.1 Introduction

In this thesis Martin-Löf's theory of types (M-LTT) will be used as a framework for the specification, derivation and execution of algorithms. The theory was originally developed as a formalization of constructive mathematics. It is built from a number of components which are, more or less, independent of each other. The development of the theory, as presented in, for example, [NPS90], [BCMS88] and [Mar84] takes the following pattern. Firstly we must have a sensible theory of what a mathematical expression is. Then we introduce the related notions of judgement and rule of inference. Finally we present the judgements that are needed in the theory of types. When we have developed the theory to this stage we can then see how it is relevant to computer science.

The aim is to build a theory that naturally reflects the actual activities of mathematicians. As we develop the theory we will first give the motivation and then give a more formal account of what we are doing.

3.2 A theory of expressions

3.2.1 Informal motivation

From an inspection of common mathematical expressions we see that the primitive notions that we must try to capture are those of *application* and *abstraction*. (Later we will see how to handle *combination* and *selection*.)

We must take care as we do not want to allow expressions to be applied to each other unrestrictedly. For instance we do not want to think that *sin sin* is an expression, nor do we want to be faced with expressions like $(\lambda x.xx)(\lambda x.xx)$ *a.k.a.* Ω from the untyped lambda calculus. (See [Bar84]. Ω is not in normal form but it β-reduces to itself.) We avoid this problem by associating an *arity* with each expression. The arities tell us how we may combine expressions and, like the types of the simply-typed lambda calculus, allow us to define a normal form for expressions, and hence a decidable equality for expressions (The proof of this is analogous to the proof that normal forms exist for terms of the simply typed λ-calculus. See, for example, [Bar84].). This is cru-

cial. For example, we will want to be able to recognise when an inference rule is applicable. If we did not have such an equality on the expressions of our language we could not be certain of this, in general. We shall also need to have a mechanism to deal with abbreviatory definitions, as these are very useful for the ordering and presentation of our thoughts.

3.2.2 Formal development

Having seen what we desire from a theory of expressions we shall now give a formal presentation of Martin-Löf's theory of aritied expressions.

3.2.2.1 Arities

Arities are inductively defined as follows:

1) o is an arity, the arity of *saturated* expressions;

2) if α and β are arities then $\alpha \to \beta$ is an arity, the arity of *unsaturated* expressions.

The notation is similar to that which we will later use for function types. These two should not be confused.

We build up expressions from variables and constants and by the actions of abstraction and application. Each expression will have an arity associated with it.

3.2.2.2 Expressions

We can now inductively define expressions and their arities:

1) if x is a variable of arity αthen x is an expression of arity α;

2) if c is a constant of arity αthen c is an expression of arity α;

3) if x is a variable of arity α and e is an expression of arity β then $(x)e$ is an expression of arity $\alpha \to \beta$;

4) if e is an expression of arity $\alpha \to \beta$ and f is an expression of arity α then e f is an expression of arity β.

3.2.2.3 Definitions

We now introduce a mechanism for making abbreviatory definitions. If e is an expression without free variables and c is a new constant then we may make an abbreviatory definition:

$$c =_{def} e.$$

We call c the *definiendum* and e the *definiens*. We are now free to use c in exactly the same way as we could have used e. Hence we add a fifth clause to the definition of an aritied expression:

5) if e is an expression of arity α and c $=_{def}$ e then c is an expression of arity α

By taking such care with the introduction of definitions we can be sure that all instances of the definiendum can be replaced by instances of the definiens without changing the meaning of the expression. This is an essential property.

3.2.3 Computation rules

We shall now explain the form of the computation rules. We shall later use computation rules to give us an operational semantics for the type theory. For the moment, however, we are only treating these rules syntactically. The rules will look like this:

$$\frac{s_1 \text{ --> } a_1, \ldots, s_n \text{ --> } a_n}{t \text{ --> } e}$$

and may be read as "if s_1 computes to a_1, and ... and s_n computes to a_n then t computes to e". The term t will, in practice, depend on the s_i. Notice that this tells us nothing about what to do if the s_i evaluate to anything other than the values mentioned.

For reasons which will become apparent later we call a constant which, when it is the outermost part of a term, permits the term to be further evaluated a *non-canonical* constant

3.2.4 Concrete syntax

We have defined what we may call the abstract syntax. We shall now introduce what we will call the concrete syntax, that is the syntax that we will actually use when displaying expressions.

The first thing that we will do is introduce a notion of *fixity* of an operator. An operator is *prefix* if it occurs before its operand(s), *infix* if it occurs between its operands, *postfix* if it occurs after its operand(s) and *distfix* if it is distributed around its operands.

We shall also introduce the notion of the *associativity* of an operator. This allows us to dispense with brackets on many occasions. An operator * is right associative if A*B*C is to be read as A*(B*C) and left associative if A*B*C is to be read as (A*B)*C.

We introduce an ordering on the operators, *priority*. Again this is to allow us to dispense with some parentheses. If one operator * has a

higher priority than another % then $A * B \% C$ is to be read as $(A*B) \% C$.

We shall also *overload* some symbols in the concrete syntax so that they have two or more meanings, depending on the context they appear in.

For convenience, if f has arity $o \to (o \to (o \to o))$, and x, y, z have arity o we shall write $f(x, y, z)$ for $(((f\ x)\ y\ )z)$, $f(x, y)$ for $((f\ x)\ y)$, and so on. In this way we can deal with combination and selection.

Further we shall also relax our rules on making definitions to allow variables in the definiendum to be bound by those in the definiens.

These rules for the concrete syntax allow us to replace, for example,

$$\leftrightarrow =_{def} (A)(B)(\&\ (\to A\ B)\ (\to B\ A))$$

by

$$A \leftrightarrow B =_{def} (A \to B)\ \&\ (B \to A).$$

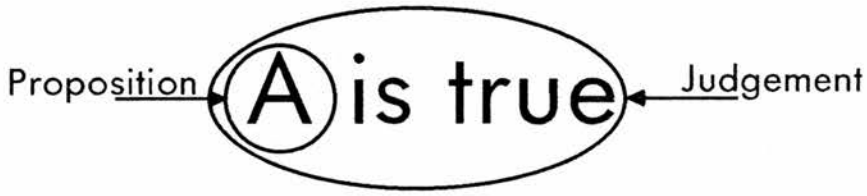
3.3 A theory of judgements

3.3.1 Semantics and proof theory

To grasp this material we need to leave the development of type theory and look at the semantic justifications for intuitionistic logic that have been presented by Dummett ([Dum75], [Dum77], [Dum91]), Prawitz ([Pra77]), and Martin-Löf himself ([Mar84], [Mar83], [Mar87]). Notice that the intuitionists do not consider logic to be a separate field from mathematics, and so this is distinction is rather artificial. Brouwer, in particular, had little interest in logic as such, and simply tries to justify intuitionistic mathematics (see, for example, [Bro81]). The intuitionist view is that mathematics consists of mental constructions that we (can, in principle) carry out ourselves. The meaning of a proposition is the mental construction that goes along with it, so to understand a proposition is to know which construction would go with it. Then we say that a proposition is true if we can make the construction. So in intuitionistic mathematics *proof* is a notion prior to *truth* and the theory of meaning is based on the idea that to know the *meaning* of a proposition is to know what a *canonical proof* of it would be. [Dum91] in particular covers this material.

In logic we deal with *propositions*, and traditionally, the *judgement* that we make is that we have a proof of a proposition, i.e. that the proposition is true. We need to make clear the distinction between propositions and judgements. Martin-Löf [Mar84] explains it thus:

> What we combine by means of the logical operators ($\wedge$, $\supset$, &, $\vee$, $\forall$, $\exists$) and hold to be true are propositions. When we hold a proposition to be true we make a judgement:



> In particular, the premisses and conclusion of a logical inference are judgements.

3.3.2 Hypothetical and categorical judgements

In general the judgements that we will make will be hypothetical, i.e. they will be made in the context of some assumptions. The meanings of hypothetical judgements can be easily induced from the meanings of the categorical judgements. When we are being formal we shall write judgements with a turnstile like this:

$$\vdash C$$

The hypothetical judgements are then written:

$$A_1 \ldots A_N \vdash C$$

However we shall often suppress the turnstile.

We explain how we may manipulate judgements by presenting a system of natural deduction (see [Gen36] and [Pra65]) rules. The rules will look like those of [Mar84]. In general they will look like:

$$[A_1 \ldots A_n] \quad \ldots \quad [B_1 \ldots B_n]$$

$$\frac{P \qquad \ldots \qquad Q}{C}$$

The premisses are written above the line and the conclusion below. Notice that, although we have suppressed the turnstiles, the conclusion and the premisses are *judgements*. The rule may *discharge* a number of the assumptions that the premisses rest on. We indicate this by enclosing these assumptions in square brackets. We may also have *hypothetical assumptions*, that is assumptions which are themselves dependent on some further assumptions. This notion is explained by

Schroeder-Heister in [Sc-H84]. We explain a rule by justifying the conclusion on the basis that the premisses are justified.

3.4 The judgements of type theory

We stated above that logic has traditionally dealt with the judgement that a proposition has a proof. We shall now show which forms of judgement we need to have for the theory of types and then show how we explain these judgements. The judgements that we need (following [Mar84]) are:

A is a type;

A and *B* are equal types;

a is an object of type *A*;

a and *b* are equal objects of type *A*.

3.4.1 The meaning of the judgement that *A* is a type

The meaning of the judgement that *A* is a type is that we know how to form the *canonical objects* of *A* and when two such are equal. The canonical objects are those which are directly introduced by the introduction rules. To justify the judgement that *A* is a type we must explain how to form the canonical objects of *A* and explain when they are equal.

Later we shall see that it is a property of the elimination and computation rules that any object of the type can be evaluated to a canonical object of the type.

3.4.2 The meaning of the judgement that *A* and *B* are equal types

The meaning of the judgement that *A* and *B* are equal types is that we know how to show that the canonical objects of *A* are (canonical) objects of *B* and *vice versa*. To justify the judgement that *A* and *B* are equal types we must explain how to show that the canonical objects of *A* are (canonical) objects of *B* and *vice versa*.

3.4.3 The meaning of the judgement that *a* is an object of the type *A*

For this judgement to make sense we must first know that *A* is a type. So we must know how to form the canonical elements of *A*. The meaning of the judgement that *a* is an object of *A* is that we know how to show that it evaluates to a canonical element of *A*. We can tell from its outermost form whether an object is canonical or not. To justify the

judgement that *a* is an object of *A* we must explain how to evaluate it to canonical form.

3.4.4 The meaning of the judgement that *a* and *b* are equal objects of type *A*

Again for this judgement to make sense we must know that *A* is a type, and hence how to introduce equal canonical objects. The meaning of this judgement is that *a* and *b* evaluate to equal canonical objects. To justify the judgement that *a* and *b* are equal objects of type *A* we must show that *a* and *b* evaluate to equal canonical objects of *A*.

3.5 On the computation rules and the elimination rules

As we introduce new types we associate with each type one *non-canonical* constant. The non-canonical constant for each type has an arity which depends on the number of clauses in the introduction rules for the type. For the enumerated types and the inductively defined types the non-canonical constant takes one more argument than there are introduction rules for the type. The choice of the non-canonical constant is justified by considering what it was that we must have known to obtain a canonical object of the type. For the enumerated types the non-canonical constants are *case* operators, for the inductively defined types they are the structural recursion operators and for mutually inductive types they are mutually, structurally recursive operators.

The non-canonical constant associated with a type allows us to compute with (i.e. to make use of) an object of that type. *Hence* it allows us to define the *elimination* rule for objects of the type. As there is only one non-canonical constant associated with each type there is strictly only one elimination rule for each type. The elimination rule for a type can be seen as a type-checking rule for the non-canonical constant. For the enumerated types and the inductively defined types there will be two more premisses to the elimination rule than there are introduction rules for the type, for *n* mutually inductive types, with a total of *m* introduction rules, the elimination rule for *each* type will have a total of $n + m + 1$ premisses. Because we have structural recursion operators as non-canonical constants the functions we construct are guaranteed to terminate.

The non-canonical constant associated with a type is a structural recursion operator. There is, however, no reason why we cannot give *computation* rules for other operators which act on values of the type. These operators will not, of course, have the special place in the type theory that the structural recursion operators have. For the inductively defined types we can define *case* operators which select cases of values of the type but do not involve recursion. The case operator is

weaker than the structural recursion operator for the type and the latter can be used to mimic the former.

3.6 Examples: some useful types

We shall now present some types which will be of use to us. These types are well-known (they are presented in, for example, [Mar84], [BCMS88], [NPS90], [Tho91]) and so we will not give full justifications. These types are needed before we can interpret (constructive) logic in Martin-Löf's type theory, and thereby draw the "propositions as types" analogy. We will appeal to this when we assert that M-LTT can be used as a specification language. The types that we present in this section should not be thought of as 'base' types. Because M-LTT is an open-ended theory we are free to add any types which we want (provided we can justify them, of course).

The following table presents some constants which will be used in the useful types presented below.

Constant	Arity
Π	$o \to (o \to o) \to o$
Σ	$o \to (o \to o) \to o$
+	$o \to o \to o$
EQ	$o \to o \to o \to o$
λ	$(o \to o) \to o$
×	$o \to o \to o$
Inl	$o \to o$
Inr	$o \to o$
apply	$o \to o \to o$
funsplit	$o \to ((o \to o) \to o) \to o$
e	o
split	$(o \to o \to o) \to o \to o$
when	$(o \to o) \to (o \to o) \to o \to o$

3.6.1 Cartesian product of a family of types

When we present the type of general trees later in this thesis we will need to make use of hypothetical assumptions. Therefore we shall illustrate the use of hypothetical assumptions when presenting the rules for the $\prod$ types.

The rule for the formation of the $\prod$ types is:

$$[x : A]$$

$$\frac{A \text{ type} \quad B(x) \text{ type}}{\Pi(y : A, B(y)) \text{ type}}$$

We also have a rule for forming equal $\prod$ types. The rule simply says that from equal arguments we get equal values. We do not present this rule, nor will we ever present the equivalent rules for the other types that we define.

We justify the above formation rule by presenting the rule for introducing values of a $\prod$ type and the rule for introducing equal values.

$$[x : A]$$

$$\frac{b(x) : B(x)}{\lambda b : \Pi(y: A, B(y))}$$

$$[x : A]$$

$$\frac{b(x) = d(x) : B(x)}{\lambda b = \lambda d: \Pi(y: A, B(y))}$$

We have the usual restriction on x not appearing free in any of the other assumptions.

We are now at the stage where we may construct the non-canonical constant associated with the $\prod$ types. It is called *funsplit*, and has the following computation rule:

$$\frac{f \dashrightarrow \lambda b \qquad a(b) \dashrightarrow b'}{\text{funsplit}(f, a) \dashrightarrow b'}$$

Now that we have the introduction rule and the computation rule we can form the elimination rule. This can be seen as a type-checking rule for *funsplit*. It is:

$$[c : \Pi(z : A, B(z))] \quad [y(x) : B(x) \; [x : A]]$$

$$\frac{f : \Pi(z : A, B(z)) \qquad C(c) \text{ type} \qquad d(y) : C(\lambda(y))}{\text{funsplit}(f, d) : C(f)}$$

This rule is justified directly from a consideration of the introduction rule and the computation rule for *funsplit*.

We have the following rule for *funsplit* equality, again justified from consideration of the introduction and computation rules:

$$[x : A] \quad [c : \Pi(z : A, B(z))] \quad [y(x) : B(x) \; [x : A]]$$

$$\frac{b(x) : B(x) \qquad C(c) \text{ type} \qquad d(y) : C(\lambda(y))}{\text{funsplit}(\lambda(b), d) = b(d) : C(\lambda(b))}$$

Using *funsplit* as the non-canonical constant associated with the Π types allows us, by using hypothetical assumptions, to cast the elimination rule in the same form as those for all the other types, however we will find it more convenient to use the constant *apply*, as defined below, in the remainder of the thesis.

$\text{apply}(f, a) =_{\text{def}} \text{funsplit}(f, (y)(y(a)))$

If A is a type and $B(x)$ is *not* dependent on x we can define:

$A \to B =_{\text{def}} \Pi(x : A, B(x))$

3.6.2 Disjoint union of a family of types

We repeat the above treatment for the Σ types. We give no justification for the rules.

The rule for forming $\Sigma(y : A, B(y))$ is:

$$[x : A]$$

$$\frac{A \text{ type} \quad B(x) \text{ type}}{\Sigma(y : A, B(y)) \text{ type}}$$

The rule for the introduction of a value of $\Sigma(y : A, B(y))$ is:

$$[x : A]$$

$$\frac{a : A \quad B(x) \text{ type} \quad b : B(a)}{\text{Pair}(a, b) : \Sigma(y : A, B(y))}$$

The rule for the introduction of equal values of $\Sigma(y : A, B(y))$ is:

$$\frac{a = c : A \quad b = d : B(a)}{\text{Pair}(a, b) = \text{Pair}(c, d) : \Sigma(y : A, B(y))}$$

The computation rule for *split* is:

$$\frac{p \dashrightarrow \mathrm{Pair}(a, b) \quad f(a, b) \dashrightarrow f'}{\mathrm{split}(f, p) \dashrightarrow f'}$$

Notice that the argument which we expect to be in the associated type is the last argument. We do this with most of the non-canonical constants (*apply* is one exception) because it makes definitions using them shorter and clearer.

The elimination rule is:

$$\frac{p : \Sigma(z : A, B(z)) \qquad C(c) \ \mathrm{type} \qquad f(x, y) : C(\mathrm{Pair}(x, y))}{\mathrm{split}(f, p) : C(p)}$$

$$[c : \Sigma(z : A, B(z))] \qquad \left[\begin{array}{l} x : A \\ y : B(x) \end{array}\right]$$

The rule for *split* equality is:

$$\frac{a : A \quad b : B(a) \qquad C(c) \ \mathrm{type} \qquad f(x, y) : C(\mathrm{Pair}(x, y))}{\mathrm{split}(f, \mathrm{Pair}(a, b)) = f(a, b) : C(\mathrm{Pair}(a, b))}$$

$$[c : \Sigma(z : A, B(z))] \qquad \left[\begin{array}{l} x : A \\ y : B(x) \end{array}\right]$$

We shall make some useful definitions:

$\mathrm{fst}(p) =_{\mathrm{def}} \mathrm{split}((a, b)a, p)$
$\mathrm{snd}(p) =_{\mathrm{def}} \mathrm{split}((a, b)b, p)$

If *A* is a type and *B(x)* is *not* dependent on *x* we can define:

$A \times B =_{\mathrm{def}} \Sigma(x : A, B(x))$
$A \ \mathrm{PAIR} =_{\mathrm{def}} A \times A$

3.6.3 Sum of two types

The rules for the + types are as follows.

The rule for forming a + type is:

$$\frac{A \ \mathrm{type} \quad B \ \mathrm{type}}{A + B \ \mathrm{type}}$$

The rules for the introduction of values of a + type are:

$$\frac{a : A}{\mathrm{inl}(a) : A + B}$$

$$\frac{b : B}{\mathrm{inr}(b) : A + B}$$

The rules for the introduction of equal values of a + type are:

$$\frac{a = c : A}{\mathrm{inl}(a) = \mathrm{inl}(c) : A + B}$$

$$\frac{b = d : B}{\mathrm{inr}(b) = \mathrm{inr}(d) : A + B}$$

The computation rules for *when* are:

$$\frac{c \dashrightarrow \mathrm{inl}(a) \qquad d(a) \dashrightarrow d'}{\mathrm{when}(d, e, c) \dashrightarrow d'}$$

$$\frac{c \dashrightarrow \mathrm{inr}(b) \qquad e(b) \dashrightarrow e'}{\mathrm{when}(d, e, c) \dashrightarrow e'}$$

The rule for + elimination is:

$$\frac{\begin{matrix}[c : A + B] & [a : A] & [b : B] \\ w : A + B & C(c)\ \mathrm{type} & d(a) : C(\mathrm{inl}(a)) \quad e(b) : C(\mathrm{inr}(b))\end{matrix}}{\mathrm{when}(d, e, w) : C(w)}$$

The rules for *when* equality are:

$$\frac{\begin{matrix}[c : A + B] & [x : A] & [b : B] \\ a : A & C(c)\ \mathrm{type} & d(x) : C(\mathrm{inl}(x)) \quad e(b) : C(\mathrm{inr}(b))\end{matrix}}{\mathrm{when}(d, e, \mathrm{inl}(a)) = d(a) : C(\mathrm{inl}(a))}$$

$$\frac{\begin{matrix}[c : A + B] & [a : A] & [x : B] \\ b : B & C(c)\ \mathrm{type} & d(a) : C(\mathrm{inl}(a)) \quad e(x) : C(\mathrm{inr}(x))\end{matrix}}{\mathrm{when}(d, e, \mathrm{inr}(a)) = e(b) : C(\mathrm{inr}(b))}$$

3.6.4 Equality

There are a number of informal notions of equality which should not be confused. So far we have encountered *identity* (sometimes called definitional equality) and the two judgements of equality. Following [Mar84] we introduce an equality *type*.

The rule for *EQ* formation is:

$$\frac{A\ \text{type}\quad a : A\quad b : A}{EQ(A,\ a,\ b)\ \text{type}}$$

The rule for *EQ* introduction is:

$$\frac{a = b : A}{e : EQ(A,\ a,\ b)}$$

A rule for EQ elimination is:

$$\frac{d : EQ(A,\ a,\ b)}{a = b : A}$$

We shall reason using equality informally later in the thesis.

We observe after [Mar84] and [Tho91] that we can use the equality type to define non-trivial dependent types, without reference to universes. In section 5.4.4.1 we will abuse a *case* operator to define a dependent type. We observe here that we could do this using *EQ* types, without mentioning universes. We choose not to do this as it only adds needless complexity.

3.6.5 Other types

We will assume that the types of lists, of booleans and of the natural numbers are well-known.

3.7 Logic and the "propositions as types" analogy

The constructive notion of a proposition is that it is something of which we would recognise a proof. Informally we may say that we understand a proposition by understanding what would count as a proof of it. Correctly we should say that to understand a proposition we must understand what would count as a direct or *canonical* proof of it. This brings to mind an analogy with types, noted by Curry ([How80]) and often called the Curry-Howard analogy. Part of the understanding that *A* is a type is to understand what a canonical object of *A* is. We shall show how to use the type forming constants to represent the proposi-

tion forming constants. The following table presents the proposition forming constants and states what count as canonical proofs of propositions so formed, following [Hey71].

Constant	Arity	Proposition formed	Canonical proof
Λ	o	The absurd proposition.	There are none.
&	$o \to (o \to o)$	$A \ \& \ B$, if A, B are propositions.	A pair consisting of a proof of A and a proof of B.
$\vee$	$o \to (o \to o)$	$A \vee B$, if A, B are propositions.	Either a proof of A or a proof of B, with an indication of which.
$\to$	$o \to (o \to o)$	$A \to B$, if A, B are propositions.	A method which takes any proof of A and returns a proof of B.
$\exists$	$o \to (o \to o) \to o$	$\exists(x : A, B(x))$, if x is a variable from some domain A and B is a propositional function over A	A pair consisting of a term a from the domain A and a proof of $B(a)$.
$\forall$	$o \to (o \to o) \to o$	$\forall(x : A, B(x))$, if x is a variable from some domain A and B is a propositional function over A.	A method which takes any term a from the domain A and returns a proof of $B(a)$.

Notice the similarity between $\&$ and $\exists$ and between $\to$ and $\forall$. For the analogy to be good the following correspondences must hold:

1) Λ corresponds to the type of which there are no values. Hence we identify Λ with the empty type.

2) $A \ \& \ B$ corresponds to some type $*(A, B)$, whose values are formed by pairing the values of A and B. B is independent of A. So we identify the proposition $A \ \& \ B$ with the type $A \times B$.

3) $A \vee B$ corresponds to some type $*(A, B)$, whose values are formed by marking the values of A and of B. So we identify the proposition $A \vee B$ with the type $A + B$.

4) $A \to B$ corresponds to some type $*(A, B)$, whose values are functions from values of A to values of B. B is independent of A. So we identify the proposition $A \to B$ with the type $A \to B$.

5) $\exists(x : A, B(x))$ corresponds to some type $*(A, B)$. Notice that B may be dependent on the particular value of A involved. Values of $*(A, B)$ are formed by pairing values of a of A and b of the corresponding $B(a)$. So we identify the proposition $\exists(x : A, B(x))$ with the type $\Sigma(x{:}A,B(x))$.

6) $\forall(x : A, B)$ corresponds to some type $*(A, B)$. B may be dependent on the particular value of A involved. Values of $*(A, B)$ are functions from values a of A to values of the corresponding $B(a)$. So we identify the proposition $\forall(x : A, B(x))$ with the type $\Pi(x : A, B(x))$.

3.8 M-LTT and computer programming

Constructive mathematics has an algorithmic content absent from classical mathematics. For advocacy of constructive mathematics and examples of its practice see, for example, [BB85], [Bro81], [Tro69] and [TD88]. For discussion of the philosophical basis of constructive mathematics, and a justification of the constructive view of mathematics is to be preferred over other views see, for example, [Dum75], [Dum77], [Dum90], [Dum91], [Hey71], [Mar83], [Mar87], [Pra77] and [Sun86b].

Although we agree that mathematics in general should have an algorithmic content which is often absent this belief is not important to the utility of M-LTT for the computer scientist. The relationship between constructive mathematics, particularly M-LTT, and computer programming has been explored in, for example, [BCMS88], [Chi88], [Mar82], [NPS90], [NS84], [Pet86], [SM87], [Smi83] and [Tho91]. A specification states that there is some relationship between the program's input and its output. As a program specification is a proposition then, by the "propositions as types" analogy it is a type. An object of this type is then a program which satisfies the specification. For example a specification for a program which sorts lists might be:

Π(as : α list, Π(ord : α -> (α -> bool), Σ(bs : α list, perm(as, bs) & ordered(bs, ord))))

Where *perm(as, bs)* is the proposition that *as* and *bs* are permutations of each other and *ordered(bs, ord)* is the proposition that *bs* is ordered by *ord*. An object of this type is then a function which takes an α *list*, a function of type α -> (α -> *bool*) and constructs a pair consisting of an α *list* and a proof that this list is an ordered permutation of the first list. We are interested in the list that is constructed, but will rarely, if ever, be interested in the proof that this list is an ordered permutation of the input list. Techniques, such as exploiting laziness, as described in [Tho91], for resolving this tension are beyond the scope of this thesis.

The remainder of this thesis is concerned with using M-LTT as a framework for the specification and implementation of a unification algorithm and with the problems that arise therefrom, particularly the use of mutually recursive types and well-founded induction in M-LTT.

Chapter 4

Related Work

In this chapter we review some previous work in this area, specifically that of Manna and Waldinger ([MW81], also in [MW90]), Eriksson ([Eri84]), Paulson ([Pau85b]) and Nardi ([Nar89]). A number of different approaches are taken in these papers. Manna and Waldinger are historically prior and all other authors in this area draw heavily on their work. They present a constructive proof of a theorem which expresses a specification. Eriksson synthesises a logic program. Paulson reports on his experiences of adapting and checking Manna and Waldinger's proof using the Cambridge LCF. Nardi presents a synthesis of a program using a deductive tableau method.

4.1 Manna and Waldinger

In [MW81] Manna and Waldinger present an informal (but rigourous) proof of the correctness of a unification algorithm. They view the constructing of a program which meets a specification as the task of proving (sufficiently constructively) a theorem of the form:

$(\forall a)(\exists z)(\text{if } P(a) \text{ then } R(a, z))$

where a is the input to the program, z the output, P the *input condition* and R describes the intended relation between the input and the output. The input condition 'expresses the class of legal inputs to which the program is expected to apply'. From the proof of the theorem a program can be extracted, which is presented in an informal applicative language.

They use the principle of well-founded induction, so the corresponding program will use general recursion.

They express terms by using *expressions* and *l-expressions*. There is an alphabet, S, of symbols, formed from three (disjoint) sets, representing constants, variables and functors. Each functor has an associated integer called its *arity*.

The expressions of S are:

constants;

variables;

functors of arity n applied to lists of expressions of length n.

The *l*-expressions of S are:

the expressions of *S*;

lists of *l*-expressions.

It is two *l*-expressions that are to be unified. A well-founded order is defined on *l*-expressions, and this is used to show that the algorithm terminates. The proof of termination relies on the most general unifier being constructed being idempotent.

Substitutions are represented by sets of *replacements*. A replacement consists of a variable and the *l*-expression which replaces it. Furthermore it is required that in a substitution all the variables are distinct and none of the *l*-expressions is one of the variables. The replacements are to be performed simultaneously.

The theorem that they prove (in their notation) is:

$$(\forall s)(\forall s')(\exists \theta)\left[\begin{array}{c} \theta \text{ is most-general, idempotent unifier of } s \text{ and } s' \\ \text{and } \theta \neq \text{nil} \\ \text{or} \\ s \text{ and } s' \text{ are not unifiable and } \theta = \text{nil} \end{array}\right]$$

where *s* and *s'* are *l*-expressions and *nil* is a special symbol, distinct from any substitution.

4.2 Eriksson

In [Eri84] Eriksson works in an untyped logic programming calculus. The logic that he uses to reason about the specification and the program is intuitionistic and is presented in a natural deduction style.

Datatypes are represented by predicates, e.g. lists are defined as:

$\forall w(\text{list}(w) \leftrightarrow w = 0 \vee \exists x \exists y(w = x.y \;\&\; \text{element}(x) \;\&\; \text{list}(y)))$.

Eriksson also changes the representation of the terms to be unified. He has a language of terms built from variables and from functors and predicates applied to term lists. He calls variables and functors applied to their arguments terms and predicate symbols applied to their arguments predicates. The program that he constructs unifies pairs of terms, pairs of term lists, or pairs of predicates.

Substitutions are represented as unambiguous lists of replacements.

He uses a separate induction schema for each type. The induction schema presented for terms is:

$\forall v(\text{variable}(v) \rightarrow P(v)) \;\&$
$\forall f \forall tl(\text{symbol}(f) \;\&\; \text{termlist}(tl) \;\&\; \forall t(t \in tl \rightarrow P(t)) \rightarrow P(\text{termfun}(f, tl))) \rightarrow$
$\forall z(\text{term}(z) \rightarrow P(z))$

This is a schema for structural induction. No schema for induction on predicates is presented. Such an induction schema is needed if a function which unifies pairs of predicates is to be defined.

The algorithm that Eriksson presents (with a minor change in syntax and our comments) is:

```
mgu(a, b, u) ← variable(a) & variable(b) & a = b & u = 0
mgu(a, b, u) ← variable(a) & variable(b) & ¬(a = b) & u = (a,b).0
mgu(a, b, u) ← variable(a) & variable(b) & ¬(a = b) & u = (b,a).0
mgu(a, b, u) ← variable(a) & function(b) &
      ¬occurs_in(a, b) & u = (a, b).0
mgu(a, b, u) ← function(a) & variable(b) &
      ¬occurs_in(b, a) & u = (b, a).0
mgu(a, b, u) ← function(a) & function(b) &
      a = termfun(s, as) & b = termfun(s, bs) & mgu(as, bs, u)

mgu(a, b, u) ← termlist(a) & termlist(b) &
      a = 0 & b = 0 & u = 0
mgu(a, b, u) ← termlist(a) & termlist(b) &
      a = ah.at & (* a has a head and a tail *)
      b = bh.bt & (* b has a head and a tail *)
      mgu(at, bt, ut) & (* the unifier of the tails is ut*)
      aprime = subst(ah, ut) & (* apply ut to the head of a *)
      bprime = subst(bh, ut) & (* apply ut to the head of b *)
      mgu(aprime, bprime, uh) & (* unify these terms*)
      u = substconc(ut, uh) (* compose the separate unifiers *)

mgu(a, b, u) ← predicate(a) & predicate(b) & a = predfun(s, as) &
b = predfun(s, bs) & mgu(as, bs, u)
```

In a typed programming system we would expect to see three functions defined: one pair of mutually recursive functions for unifying term or term list pairs and one function to unify predicates. In the case of failure to unify two terms, term lists or predicates this algorithm merely fails, rather than explicitly reporting failure.

Notice that although the term induction schema presented is a structural induction schema the algorithm uses well founded recursion. In the case when two term lists are to be unified their tails are unified and then the terms constructed by applying the unifier of the tails to the heads are unified. Hence this program could not be derived using the induction schema described.

Eriksson only claims to prove partial correctness. He does not specify that the most general unifier constructed by his program is also idempotent. We will later see that we require this property to prove both that our algorithm terminates and that it is correct. We think that a proof of the termination of Eriksson's algorithm would also rely on the idempotence of the constructed unifier. As we have stated Eriksson needs to provide a schema for well founded induction on his terms in order correctly to derive the algorithm that he presents. It is our experience that the idempotence of the constructed unifier will be required to prove that the recursive calls are made on values which are in the required well founded order.

4.3 Paulson

In [Pau85b] Paulson describes his experiences checking Manna and Waldinger's proof using the Cambridge LCF ([Pau85a], [Pau87]), an interactive theorem prover for reasoning about computable functions. Notice that the meta-language of LCF (ML) uses Milner's polymorphic type system ([Mil78]), the correctness of which depends on the correctness of a unification algorithm. In the course of attempting to check Manna and Waldinger's proof Paulson finds that he must make some changes. A significant change is made to the language of terms. He uses variables, constants and terms applied to terms. He states that this simplifies the proof 'without sacrificing generality'. There remains a burden of proof here however. There are terms (e.g. the application of the variable x to the variable y) in Paulson's language that are not terms under our intuitive notion. We shall call such terms *unacceptable* and terms that accord with out intuitive notion *acceptable*. We can expect the unification algorithm to unify unacceptable terms. We may fear that it will produce unifiers that, when applied to acceptable terms produce unacceptable terms. We may also fear that two acceptable terms may unify to an unacceptable term. We may also fear that two acceptable terms may have as their most general unifier an unacceptable substitution. A proof that this unification algorithm is safe is required for us to be convinced that Paulson's proof applies to the case of interest to us.

4.4 Nardi

Nardi [Nar89] presents the work of Manna and Waldinger, providing a formal proof using the deductive tableau method. Deductive tableaux are described in [MW90] and are a variant of the semantic tableaux described in [Sun86a]. It is to be noted that one of the rules used is a resolution rule which allows the derivation of a new line of a tableau 'if it is possible to unify two subsentences of two rows of the tableau'. Nardi adopts exactly Manna and Waldinger's notion of what a specification is and aims to synthesise an algorithm from the proof that the specification can be met. The logic used in the tableaux is classical:

> The distinction between assertions and goals reflects the usual distinction between hypothesis and theses; any goal can be moved into the assertion column by simply negating it (and vice versa), without affecting the meaning of the tableau.

It is therefore not clear exactly what the status of the proof is. On one hand it may be considered as a classical proof that a given algorithm meets a given specification. However, to assert that an algorithm can be synthesised (or extracted) from the classical proof requires further justification, even if this simply consists of asserting that none of the rules used was non-constructive.

Again, well-founded induction is used.

Nardi deviates from Manna and Waldinger by choosing to use Paulson's terms. The caveat above remains valid here too.

Chapter 5

Representing Terms

In this chapter we present two ways to represent our intuitive notion of a term using type theory. The first way is to use two mutually dependent types. The second way is to use the general trees that are presented in [NPS90].

5.1 Diversion on mutually recursive types

Recall that with the ordinary inductively defined types we must explain the conditions under which the type is well-formed, the conditions under which canonical and equal canonical elements of the type are formed, present the rules that will allow us to perform structurally recursive computations with the values of the type, state the consequences of knowing that we have a value of the type and finally present equality rules involving non-canonical elements formed using the recursion operator. In [BCMS88] Backhouse *et al* state that:

> Mutual recursion adds nothing substantially new to type theory. What innovations there are, reside in the elimination and computation rules.

We should like to add that the use of mutually recursive types forces us to take great care in the framing of specifications and of the well-founded order that we will later present for the type of terms. If we have a family of mutually recursive types $T_1 \ldots T_n$ then we cannot, in general, define a function on any one of them without simultaneously defining analogous functions on all the others. It should therefore be clear that we cannot specify a function on any one of them without, in general, providing an analogous specification for all the others.

5.1.1 Example

As an informal example we discuss the two mutually recursive types *ODD* and *EVEN*.

The rules for the introduction of canonical elements of the types *ODD* and *EVEN* are:

$$\frac{\mathsf{e} : \mathsf{EVEN}}{\mathsf{So(e)} : \mathsf{ODD}}$$

$$\frac{o : ODD}{Se(o) : EVEN}$$

$$\frac{}{Zero : EVEN}$$

We can define another type:

EOSUM $=_{def}$ ODD + EVEN

EOSUM is, of course, just another representation of the natural numbers. For any family of mutually recursive types we can always define a new type which contains all of them in this way.

We now start to consider what we must do to define functions over the types *ODD* and *EVEN*. There are a few functions, like the predecessor function, which we can define, in M-LTT, on the type *ODD*, without defining any function on the type *EVEN*. However, most functions that we will be interested in will require that we define two functions simultaneously. For example to add an odd number to an odd number we can expect to have to explain how to add an odd number to an even number. Defining the function over the type *EOSUM* implicitly requires the we define it over *ODD* and *EVEN*. For any family of mutually recursive types we can decide to work with their sum. Functions and specifications can be written for this, and the appropriate functions and specifications for the individual type extracted. We can view the type of general trees as a way of encoding this observation.

5.2 The type of terms

In this section we will give formal rules for the types *TERM* and *TERMS*. These types are mutually dependent. We can give a direct semantics for them in precisely the same way that we provide a semantics for ordinary types. We must make some changes to reflect the fact that we are defining two types simultaneously, but the justification of the rules concerning these types follows exactly the same pattern as the justification of the rules for any other types.

The types *VAR* and *CONST* are presumed to be known. They must each have a decidable equality.

We will often use the word 'term' to indicate informally an object either of type *TERM* or of type *TERMS*.

5.2.1 Formation rules

The formation rules for *TERM* and *TERMS* are:

$$\frac{}{TERM\ type} \qquad \frac{}{TERMS\ type}$$

We could have chosen to parameterise the types *TERM* and *TERMS* by the types of the variables and constants that we use. We would then have these formation rules:

$$\frac{V \text{ type} \quad C \text{ type}}{\text{TERM}(V,\ C) \text{ type}} \qquad \frac{V \text{ type} \quad C \text{ type}}{\text{TERMS}(V,\ C) \text{ type}}$$

This however would simply add needless complexity.

5.2.2 Introduction rules

The rules for the introduction of canonical elements of the types *TERM* and *TERMS* are:

$$\frac{v : \text{VAR}}{\text{Var}(v) : \text{TERM}}$$

$$\frac{c : \text{CONST} \quad ts : \text{TERMS}}{\text{App}(c,\ ts) : \text{TERM}}$$

$$\frac{}{\text{None} : \text{TERMS}}$$

$$\frac{t : \text{TERM} \quad ts : \text{TERMS}}{\text{Some}(t,\ ts) : \text{TERMS}}$$

The rules for the introduction of equal canonical elements of *TERM* and *TERMS* are:

$$\frac{w = v : \text{VAR}}{\text{Var}(w) = \text{Var}(v) : \text{TERM}}$$

$$\frac{d = c : \text{CONST} \quad ss = ts : \text{TERMS}}{\text{App}(d,\ ss) = \text{App}(c,\ ts) : \text{TERM}}$$

$$\frac{}{\text{None} = \text{None} : \text{TERMS}}$$

$$\frac{s = t : \text{TERM} \quad ss = ts : \text{TERMS}}{\text{Some}(s,\ ss) = \text{Some}(t,\ ts) : \text{TERMS}}$$

5.2.3 Computation rules

From these introduction rules we can construct the computation rules for the non-canonical constants associated with these types. The non-

canonical constant associated with a type allows us to perform structurally recursive computation on values of the type.

$$\frac{t \dashrightarrow Var(x) \quad d(x) \dashrightarrow d'}{termrec(d,\ e,\ f,\ g,\ t) \dashrightarrow d'}$$

$$\frac{t \dashrightarrow App(c,\ ts) \quad e(c,\ ts,\ termsrec(d,\ e,\ f,\ g,\ ts)) \dashrightarrow e'}{termrec(d,\ e,\ f,\ g,\ t) \dashrightarrow e'}$$

$$\frac{ts \dashrightarrow None \quad f \dashrightarrow f'}{termsrec(d,\ e,\ f,\ g,\ ts) \dashrightarrow f'}$$

$$\frac{ts \dashrightarrow Some(s,\ ss) \quad g(s,\ ss,\ termrec(d,\ e,\ f,\ g,\ s),\ termsrec(d,\ e,\ f,\ g,\ ss)) \dashrightarrow g'}{termsrec(d,\ e,\ f,\ g,\ ts) \dashrightarrow g'}$$

Notice that the computation rules for *termrec* and *termsrec* are mutually dependent. To evaluate expressions formed using *termrec* we can expect to evaluate expressions formed using *termsrec* and *vice versa*.

5.2.4 A single non-canonical constant for a group of mutually recursive types

At this point we would be free to define a new non-canonical constant *term_or_termsrec*, whose computation rules are as above, but with every occurrence of *termrec* and *termsrec* replaced with *term_or_termsrec*. This non-canonical constant can be used in place of the other two in the elimination rules and so on. We will return to this point later in this chapter after we have shown that we can represent our types *TERM* and *TERMS* using tree types and that we can define *termsrec* and *termrec* using *treerec*.

5.2.5 Elimination rules

Figure 5.5 is the rule for using an element of *TERM*.

Figure 5.6 is the rule for using an element of *TERMS*.

We have two mutually dependent elimination rules, which coincide on their minor premisses. As stated previously, for *m* mutually defined types, with a total of *n* introduction rules the elimination rules require *n + m* minor premisses: *m* well-formedness conditions and *n* premisses corresponding to the introduction rules. The elimination rules for *TERM* and *TERMS* must be justified together and their justification follows the usual pattern of carefully explaining what we could do if we had a value of each of the types.

Suppose we have an object *t* of type *TERM*. We wish to justify the judgement that *termrec(d, e, f, g, t)* has a value of some type, *P(t)*,

depending on *t*, so we must have as a premiss to the rule that *P(x)* is a type if *x* has type *TERM*.

t must take either *Var(v)* or *App(c, ts)* as its value.

Suppose *t* takes value *Var(v)*. From the computation rules we know that *termrec(d,e,f,g,Var(v))* has the same value as *d(v)*, so if *d(v):P(Var(v))* when *v : VAR* then *termrec(d,e,f,g,t):P(t)*. Therefore we need the third minor premiss.

Suppose *t* takes value *App(c,ts)*. From the computation rules we know that *termrec(d,e,f,g,App(c,ts))* has the same value as *e(c,ts,termsrec(d,e,f,g,ts))*. *termsrec(d,e,f,g,ts)* takes a value, say *q*, of type *Q(ts)*. For this to be sensible *Q(ts)* must be a type if *ts : TERMS*, hence we need the second minor premiss. If *e(c,ts,q):P(App(c,ts))* then *termrec(d,e,f,g,t):P(t)*.

We are still required to explain the conditions under which *termsrec(d, e, f, g, ts)* can take a value of type *Q(ts)*, given that *ts* is of type *TERMS*. Notice that this is *precisely* what we will be asked to do to explain the *TERMS* elimination rule.

ts must have either *None* or *Some(h, tl)* as its value.

Suppose *ts* takes value *None*. From the computation rules we know that *termsrec(d, e, f, g, None)* has the same value as *f*, so if *f:Q(None)* then *termsrec(d, e, f, g, None) : Q(None)* . Therefore we require the fifth minor premiss.

Suppose *ts* takes value *Some(hd, tl)* . From the computation rules we know that *termsrec(d,e,f,g,Some(hd,tl))* has the same value as

g(hd, tl, termrec(d, e, f, g, hd), termrec(d, e, f, g, tl)).

termrec(d,e,f,g,hd) and *termrec(d,e,f,g,tl)* will take values, say *r* and *s*, of types *P(hd)* and *Q(tl)* respectively. Hence if *g(hd,tl,r,s):Q(Some(hd,tl)* then

termsrec(d, e, f, g, Some(hd, tl)) : Q(Some(hd, tl)

Hence we require the sixth minor premiss.

We have now explained how we justify the conclusion of the *TERM* elimination rule on the basis of its premisses. The justification of the *TERMS* elimination rule follows a similar pattern, except that because the major premiss is that *ts : TERMS*, we first see the need for the second, fifth and sixth minor premisses and need to add the first, third and fourth when we explain how *termsrec(d, e, f, g, t)*, where *t:TERM* has a value of type *P(t)* as we explain the sixth minor premiss.

5.2.6 *Termrec* and *termsrec* equality

Figures 5.7, 5.8, 5.9 and 5.10 are the equality rules.

We justify figure 5.7 by observing that from the computation rules *termrec(d, e, f, g, Var(p))* takes the same value as *d(p)*. The minor premisses of the rule are required to ensure that this value is of an appropriate type, using an argument identical to that presented above for the elimination rules.

We justify figure 5.8 by observing that from the computation rules *termrec(d,e,f,g,App(a,b))* takes the same value as *e(a,b,termsrec(d, e, f, g, b))*. The minor premisses of the rule are required to ensure that this value is of an appropriate type, using an argument identical to that presented above for the elimination rules.

We justify figure 5.9 by observing that from the computation rules *termsrec(d, e, f, g, None)* takes the same value as *f*. The premisses of the rule are required to ensure that this value is of an appropriate type, using an argument identical to that presented above for the elimination rules.

We justify figure 5.10 by observing that from the computation rules *termsrec(d,e,f,g,Some(a,b))* takes the same value as *g(a,b,termrec(d, e, f, g, a), termsrec(d, e, f, g, b))*. The minor premisses of the rules are required to ensure that this value is of an appropriate type, using an argument identical to that presented above for the elimination rules.

5.2.7 Remarks

We have now presented the two types *TERM* and *TERMS*, and justified the rules for dealing with them in a direct fashion. In the next two sections we introduce these types and justify these rules in a different way, by utilising the general trees that are presented in [NPS90]. This is, strictly, unnecessary, but serves to underline the correctness of the approach taken above to the definition and use of mutually recursive types. M-LTT is an open-ended theory: we are free to add new types and forms of types (e.g. enumerated types, inductively defined types, mutually inductively defined types) provided we can justify them. We are not forced to define everything in terms of some special 'base' types. We should, however, be wary of adding new apparatus to the theory, as this provides us with an opportunity to make absurd extensions.

5.3 The type of general trees

We recap the section on general trees from [NPS90]. We do not give any justification of the rules here.

5.3.1 *Tree* formation

The rule for the formation of the tree type is:

$$[x : A] \quad \left[\begin{array}{c} x : A \\ y : B(x) \end{array}\right] \quad \left[\begin{array}{c} x : A \\ y : B(x) \\ z : C(x, y) \end{array}\right]$$

$$\frac{A \text{ type} \quad B(x) \text{ type} \quad C(x, y) \text{ type} \quad d(x,y,z) : A \quad a : A}{\mathrm{Tree}(A, B, C, d)(a) \text{ type}} \text{ Tree Form}$$

We write $\mathcal{T}(a)$ to abbreviate *Tree(A, B, C, d)(a)*. *A* is the index set for the trees, and is used to allow us to pick out definitions relating to a particular one of the mutually recursive types that we are encoding as trees. We re-iterate that we cannot in general do anything for one $\mathcal{T}(a)$ without explaining how to do something analogous for the other $\mathcal{T}(a)$.

5.3.2 *Tree* introduction

The rule for the introduction of a value of $\mathcal{T}(a)$ is:

$$[z : C(a, b)]$$

$$\frac{a : A \quad b : B(a) \quad c(z) : \mathcal{T}(d(a, b, z))}{\mathrm{tree}(a, b, c) : \mathcal{T}(a)} \text{ Tree Intro}$$

5.3.3 Computation rules

The non-canonical constant associated with the type of general trees is *treerec*, which has the following computation rule:

$$\frac{t \dashrightarrow \mathrm{tree}(a, b, c) \quad f(a, b, c, (x)\mathrm{treerec}(f, c(x))) \dashrightarrow f'}{\mathrm{treerec}(f, t) \dashrightarrow f'}$$

5.3.4 *Tree* elimination

The tree elimination rule is:

$$\frac{D(x,\ s)\ :\ \text{type}\quad a\ :\ A\quad t\ :\ \mathcal{T}(a)\qquad f(x,\ y,\ z,\ u)\ :\ D(x,\ \text{tree}(x,\ y,\ z))}{\text{treerec}(f,\ t)\ :\ D(a,\ t)}\ \text{Tree Elim}$$

where the premise $f(x, y, z, u) : D(x, \text{tree}(x, y, z))$ is under the hypotheses

$$\begin{bmatrix} x : A \\ s : \mathcal{T}(x) \end{bmatrix} \qquad \begin{bmatrix} x : A \\ y : B(x) \\ z(v) : \mathcal{T}(d(x, y, v))\ [v : C(x, y)] \\ u(v) : D(d(x, y, v), z(v))\ [v : C(x, y)] \end{bmatrix}$$

5.3.5 *Treerec* equality

Figure 5.11 is the rule for *treerec* equality.

5.4 Using the type of general trees to encode terms

Having presented the rules for the type of general trees we shall now show how to define the type of terms using the type of trees and proceed to justify the rules for terms on the basis of the rules for trees.

5.4.1 Some preliminary definitions

We make the following definitions, with *A*, *B*, *C*, *d* as above:

$A =_{def} \{TM, TMS\}$

$B(TM) =_{def} VAR + CONST$
$B(TMS) =_{def} \{N, S\}$
i.e $B =_{def} \text{case}_{\{TM,\ TMS\}}(VAR + CONST, \{N, S\})$

$C(TM, \text{Inl}(v)) =_{def} \{\}$
$C(TM, \text{Inr}(c)) =_{def} \{args\}$
$C(TMS, N) =_{def} \{\}$
$C(TMS, S) =_{def} \{hd, tl\}$
i.e. $C =_{def} (y)\ \text{case}_{\{TM,\ TMS\}}(\text{when}((v)\{\}, (c)\{args\}, y),$
$\text{case}_{\{N,\ S\}}(\{\}, \{hd, tl\}, y)$
$)$

$d(TM, \mathrm{Inl}(v), f) =_{def} \mathrm{case}_{\{\}}(f)$
$d(TM, \mathrm{Inr}(c), args) =_{def} TMS$
$d(TMS, N, f) =_{def} \mathrm{case}_{\{\}}(f)$
$d(TMS, S, hd) =_{def} TM$
$d(TMS, S, tl) =_{def} TMS$

i.e. $d =_{def} (y)(z)\ \mathrm{case}_{\{TM,\ TMS\}}(\mathrm{when}((v)\mathrm{case}_{\{\}}(z),$
$(c)\mathrm{case}_{\{args\}}(TMS, z), (*= (c)\ TMS*)$
y
$),$
$\mathrm{case}_{\{N,\ S\}}(\mathrm{case}_{\{\}}(z),$
$\mathrm{case}_{\{hd,\ tl\}}(TM, TMS, z),$
$y)$
$)$

5.4.2 Justifying the introduction rules

Now we shall present instances of the *Tree* introduction rule which, with appropriate definitions, can be used to justify the *TERM* and *TERMS* introduction rules.

5.4.2.1 Justifying the first *TERM* introduction rule

$$\frac{TM : \{TM,\ TMS\} \qquad \dfrac{v : VAR}{\mathrm{Inl}(v) : VAR + CONST} \qquad \overset{[f\ :\ \{\}]}{\mathrm{case}_{\{\}}(f) : \mathcal{T}(\mathrm{case}_{\{\}}(f))}}{\mathrm{tree}(TM,\ \mathrm{Inl}(v),\ \mathrm{case}_{\{\}}) : \mathcal{T}(TM)}$$

justifies (because the judgements of the first and third premisses are immediate):

$$\frac{v : VAR}{\mathrm{tree}(TM,\ \mathrm{Inl}(v),\ \mathrm{case}_{\{\}}) : \mathcal{T}(TM)}$$

which, given the definitions:

$TERM =_{def} \mathcal{T}(TM)$
$\mathrm{Var}(v) =_{def} \mathrm{tree}(TM, \mathrm{Inl}(v), \mathrm{case}_{\{\}})$

justifies:

$$\frac{v : VAR}{\mathrm{Var}(v) : TERM}$$

which is the first *TERM* introduction rule.

5.4.2.2. Justifying the second *TERM* introduction rule

$$\frac{TM : \{TM,\ TMS\} \qquad \dfrac{c : CONST}{Inr(c) : VAR + CONST} \qquad ts : \mathcal{T}(TMS)}{tree(TM,\ Inr(c),\ (z)ts) : \mathcal{T}(TM)}\ [z : \{args\}]$$

justifies (because the judgement of the first is immediate):

$$\frac{c : CONST \qquad ts : \mathcal{T}(TMS)}{tree(TM,\ Inr(c),\ (z)ts) : \mathcal{T}(TM)}$$

which, with the following definitions:

$TERMS =_{def} \mathcal{T}(TMS)$
$App(c, ts) =_{def} tree(TM, Inr(c), (z)ts)$

justifies:

$$\frac{c : CONST \qquad ts : TERMS}{App(c,\ ts) : TERM}$$

which is the second *TERM* introduction rule.

5.4.2.3 Justifying the first *TERMS* introduction rule

$$\frac{\begin{array}{c}[f : \{\}]\\ \vdots\\ TMS : \{TM,\ TMS\} \quad N : \{N,\ S\} \quad case_{\{\}}(f) : \mathcal{T}(case_{\{\}}(f))\end{array}}{tree(TMS,\ N,\ case_{\{\}}) : \mathcal{T}(TMS)}$$

justifies (because the judgements of all the premisses are immediate):

$$\frac{}{tree(TMS,\ N,\ case_{\{\}}) : \mathcal{T}(TMS)}$$

which, with the following definition:

$None =_{def} tree(TMS, N, case_{\{\}})$

justifies:

$$\frac{}{None : TERMS}$$

which is the first *TERMS* introduction rule.

5.4.2.4 Justifying the second *TERMS* introduction rule

Figure 5.12 justifies (because the judgements of the final three premisses are immediate):

$$\frac{\begin{array}{c}[t : \mathcal{T}(TM)]\\ \vdots\\ t : \mathcal{T}(TM) \quad ts : \mathcal{T}(TMS)\end{array}}{tree(TMS,\ S,\ case_{\{hd,\ tl\}}(t,\ ts)) : \mathcal{T}(TMS)}$$

which, with the following definition:

$Some(t, ts) =_{def} tree(TMS, S, case_{\{hd,\ tl\}}(t, ts))$

justifies:

$$\frac{t : TERM \quad ts : TERMS}{Some(t,\ ts) : TERMS}$$

which is the second *TERMS* Introduction rule.

We have now added the following definitions:

$TERM =_{def} Tree(\{TM, TMS\}, B, C, d, TM) =_{def} \mathcal{T}(TM)$
$TERMS =_{def} Tree(\{TM, TMS\}, B, C, d, TMS) =_{def} \mathcal{T}(TMS)$
$Var(x) =_{def} tree(TM, Inl(x), case_{\{\}})$
$App(c, ts) =_{def} tree(TM, Inr(c), (z)ts)$
$None =_{def} tree(TMS, N, case_{\{\}})$
$Some(t, ts) =_{def} tree(TMS, S, case_{\{hd, tl\}}(t, ts))$.

We have now justified the introduction rules for the types *TERM* and *TERMS* on the basis of the introduction rules for *TREE* and some suitable definitions.

5.4.3 Justifying the rules for the introduction of equal canonical elements

We shall not present justifications of the rules for the introduction of equal canonical elements of *TERM* and *TERMS*, but these can be justified in the same way as the introduction rules were justified from the rules for the introduction of equal canonical elements of *TREE* and the above definitions.

5.4.4 Justifying the elimination rules

Next we need to justify the elimination rules and define *termrec* and *termsrec* in terms of *treerec*. We justify the elimination rule for *TERM* by showing that, if the premisses of this rule are valid judgements then we may justify the conclusion of this rule by using the *TREE* elimination rule. We may then do the same for the elimination rule for *TERMS*.

5.4.4.1 Justifying *TERM* elimination

We shall justify the *TERM* elimination rule from the tree elimination rule. For convenience we shall recap the rule for *TERM* elimination in Figure 5.13, using new symbols to avoid clashing with those used in the *TREE* elimination rule.

The rule for using a value of type $\mathcal{T}(TM)$ is:

$$\left[\begin{array}{c} x : \{TM, TMS\} \\ s : \mathcal{T}(x) \end{array}\right] \qquad \left[\begin{array}{c} x : \{TM, TMS\} \\ y : B(x) \\ z(v) : \mathcal{T}(d(x, y, v)) \ [v : C(x, y)] \\ u(v) : D(d(x, y, v), z(v)) \ [v : C(x, y)] \end{array}\right]$$

$$\frac{D(x, s) : type \quad TM : \{TM, TMS\} \quad t : \mathcal{T}(TM) \qquad f(x, y, z, u) : D(x, tree(x, y, z))}{treerec(t, f) : D(TM, t)}$$

We shall treat the first premiss of the $\mathcal{T}(TM)$ elimination rule as the following pair of judgements:

$$[t : \mathcal{T}(TM)]$$

$$D(TM, t)\ \text{type}$$

and

$$[s : \mathcal{T}(TMS)]$$

$$D(TMS, s)\ \text{type}$$

The definitions:

$D(z, x) =_{def} \text{case}_{\{TM, TMS\}}(P(x), Q(x), z)$
$TERMS =_{def} \mathcal{T}(TMS)$
$TERM =_{def} \mathcal{T}(TM)$

where P and Q are as before, justify the first and second minor premisses of the TERM elimination rule.

In section 3.6.4 we observed that we could avoid this abuse of *case* by using EQ types, but choose not to do this as it only adds needless complexity.

The second premiss of the $\mathcal{T}(TM)$ elimination rule may be ignored, as the judgement of it is immediate.

The third premiss, given the definition $TERM =_{def} \mathcal{T}(TM)$, is justified by the major premiss of the *TERM* elimination rule.

The fourth premiss of the $\mathcal{T}(TM)$ elimination rule will be justified from the final four premisses of the *TERM* elimination rule. We shall treat the fourth premiss as the following four judgements:

Figure 5.1)

$$\left[\begin{array}{c} v : VAR \\ z(w) : \mathcal{T}(d(TM, Inl(v), w))\ [w : C(TM, Inl(v))] \\ u(w) : D(d(TM, Inl(v), w), z(w))\ [w : C(TM, Inl(v))] \end{array}\right]$$

$$f(TM, Inl(v), z, u) : D(TM, tree(TM, Inl(v), z))$$

Figure 5.2)

$$\left[\begin{array}{c} c : CONST \\ z(w) : \mathfrak{T}(d(TM,\ Inr(c),\ w))\ [w : C(TM,\ Inr(c))] \\ u(w) : D(d(TM,\ Inr(c),\ w),\ z(w))\ [w : C(TM,\ Inr(c))] \end{array}\right]$$

$$f(TM,\ Inr(c),\ z,\ u) : D(TM,\ tree(TM,\ Inr(c),\ z))$$

Figure 5.3)

$$\left[\begin{array}{c} z(w) : \mathfrak{T}(d(TMS,\ N,\ w))\ [w : C(TMS,\ N)] \\ u(w) : D(d(TMS,\ N,\ w),\ z(w))\ [w : C(TMS,\ N)] \end{array}\right]$$

$$f(TMS,\ N,\ z,\ u) : D(TMS,\ tree(TMS,\ N,\ z))$$

Figure 5.4)

$$\left[\begin{array}{c} z(w) : \mathfrak{T}(d(TMS,\ S,\ w))\ [w : C(TMS,\ S)] \\ u(w) : D(d(TMS,\ S,\ w),\ z(w))\ [w : C(TMS,\ S)] \end{array}\right]$$

$$f(TMS,\ S,\ z,\ u) : D(TMS,\ tree(TMS,\ S,\ z))$$

Now we can use the definitions above to reduce these judgements further.

Figure 5.1 becomes

$$\left[\begin{array}{c} v : VAR \\ z(w) : \mathfrak{T}(case_{\{\}}(f))\ [w : \{\}] \\ u(w) : D(case_{\{\}}(f),\ z(w))\ [w : \{\}] \end{array}\right]$$

$$f(TM,\ Inl(v),\ z,\ u) : D(TM,\ tree(TM,\ Inl(v),\ z))$$

If we take z to be $case_{[]}$ and $f(TM,\ Inl(v),\ z,\ u)$ to be $\alpha(v)$, where α is the function mentioned in the TERM elimination rule then we get the following:

$$[v : VAR]$$

$$\alpha(v) : P(Var(v))$$

which is justified by the third minor premiss of *TERM* elimination.

Figure 5.2 becomes

$$\left[\begin{array}{c} c : CONST \\ z(w) : \mathcal{T}(TMS)\ [w : \{args\}] \\ u(w) : D(TMS,\ z(w))\ [w : \{args\}] \end{array}\right]$$

$$f(TM,\ Inr(c),\ z,\ u) : D(TM,\ tree(TM,\ Inr(c),\ z))$$

If we assume *TS : TERMS*, take *z* to be *(x)TS* and *f(TM, Inr(c), z, u)* to be $\beta(c, z, u)$ this becomes:

$$\left[\begin{array}{c} c : CONST \\ z(w) : \mathcal{T}(TMS)\ [w : \{args\}] \\ u(w) : Q(z(w))\ [w : \{args\}] \end{array}\right]$$

$$\beta(c,\ z,\ u):\ P(App(c,\ TS))$$

which is justified by the fourth minor premiss of *TERM* elimination.

Figure 5.3 becomes:

$$\left[\begin{array}{c} z(w) : \mathcal{T}(case_{\{\}}(f))\ [w : \{\}] \\ u(w) : D(case_{\{\}}(f),\ z(w))\ [w : \{\}] \end{array}\right]$$

$$f(TMS,\ N,\ z,\ u) : D(TMS,\ tree(TMS,\ N,\ z))$$

If we take *f(TMS, N, z, u)* to be χ and *z* to be $case_{[]}$ this becomes:

$$\left[\begin{array}{c} z(w) : \mathcal{T}(case_{\{\}}(f))\ [w : \{\}] \\ u(w) : D(case_{\{\}}(f),\ z(w))\ [w : \{\}] \end{array}\right]$$

$$\chi : Q(None)$$

which is justified by the fifth minor premiss of *TERM* elimination.

Figure 5.4 becomes:

$$\left[\begin{array}{c} z(w) : \mathcal{T}(d(TMS,\ S,\ w))\ [w : \{hd,\ tl\}] \\ u(w) : D(d(TMS,\ S,\ w),\ z(w))\ [\ w : \{hd,\ tl\}] \end{array}\right]$$

$$f(TMS,\ S,\ z,\ u) : D(TMS,\ tree(TMS,\ S,\ z))$$

If we expand the hypothetical assumptions we get:

$$\begin{bmatrix} z(hd) : \mathcal{T}(TM) \\ z(tl) : \mathcal{T}(TMS) \\ u(hd) : P(z(hd)) \\ u(tl) : Q(z(tl)) \end{bmatrix}$$

$$f(TMS,\ S,\ z,\ u) : D(TMS,\ tree(TMS,\ S,\ z))$$

which is justified by the last minor premiss of the *TERM* elimination rule, if we assume:

T : TERM
TS : TERMS

and take *z* to be:

$case_{\{hd,\ tl\}}(T,\ TS)$

and *f(TMS, S, z, u)* to be:

$\delta(z(hd),\ z(tl),\ u(hd),\ u(tl))$

In this section we have justified the elimination rule for the type *TERM*. Much of the work has been done by expanding the final premiss of the rule for using a value of type $\mathcal{T}(TM)$. We now reverse this process to allow us to define *termrec* in terms of *treerec*.

We have now taken:

f(TM, Inl(v), z, u) to be $\alpha(v)$
f(TM, Inr(c), z, u) to be $\beta(c,\ z,\ u)$
f(TMS, N, z, u) to be χ
f(TMS, S, z, u) to be $\delta(z(hd),\ z(tl),\ u(hd),\ u(tl))$

The assumptions used in Figures 5.1 to 5.5 respectively ensure that these expressions are correctly typed.

Hence we have taken:

f(TM, v_or_c, z, u) to be $when(\alpha,\ (c)\beta(c,\ z(args),\ u(args)),\ v\_or\_c)$
f(TMS, ns, z, u) to be $case_{\{N,\ S\}}(\chi,\ \delta(z(hd),\ z(tl),\ u(hd),\ u(tl)),\ ns)$

Again, the types of these expressions are determined by the assumptions that we have made.

Hence we have taken:

f(x, y, z, u) to be
$case_{\{TM,\ TMS\}}(when(\alpha,\ (c)\ \beta(c,\ z(args),\ u(args)),\ y),$
$case_{\{N,\ S\}}(\chi,\ \delta(z(hd),\ z(tl),\ u(hd),\ u(tl)),\ y),\ x)$

On the assumptions made in the fourth premiss of the $\mathcal{T}(TM)$ elimination rule and given our earlier definition of *D* this expression is of type *D(x,tree(x,y,z))*.

So we may now make the definition:

termrec(α, β, χ, δ, t) $=_{def}$
treerec((b, z, u)case$_{\{TM, TMS\}}$(when(α, (c) β(c, z(args), u(args)), b),
case$_{\{N, S\}}$(χ, δ(z(hd), z(tl), u(hd), u(tl)), b))
),
t).

5.4.4.2 Justifying *TERMS* elimination

The above justification goes through *mutatis mutandis* to justify the *TERMS* elimination rule. We make the definition:

termsrec(α, β, χ, δ, ts) $=_{def}$
treerec((b, z, u)case$_{\{TM, TMS\}}$(when(α, (c) β(c, z(args), u(args)), b),
case$_{\{N, S\}}$(χ, δ(z(hd), z(tl), u(hd), u(tl)), b))
),
ts).

5.5 Conclusions

In this chapter we have shown that we may represent our terms in two ways in M-LTT. Firstly by extending the theory with mutually recursive types and using these, and secondly by using the type of trees. We have justified the rules that we gave for the mutually recursive types using the rules that we gave when using trees.

We notice that the definitions of *termsrec* and *termrec* that we gave using *treerec* are identical. This should not be a surprise: in general to define a function over the type *TERM* we must define a function over *TERMS* and *vice versa*. However we may expect there to be one non-canonical constant for each type. If our understanding of mutually recursive types is correct, or if our interpretation of our mutually recursive types as trees is correct then we have one non-canonical constant for every group of mutually recursive types. We may say that every type has only one non-canonical constant but one constant may be the non-canonical constant associated with a number of types. At a number of places later in the thesis we will use the identity of *termsrec* and *termrec*.

Figure 5.5) *TERM* Elim

$$\frac{t : \mathrm{TERM} \quad P(x)\ \mathrm{type} \quad Q(y)\ \mathrm{type} \quad d(v) : P(\mathrm{Var}(v)) \quad e(c, u, q) : P(\mathrm{App}(c, u)) \quad f : Q(\mathrm{None}) \quad g(w, z, r, s) : Q(\mathrm{Some}(w, z))}{\mathrm{termrec}(d, e, f, g, t) : P(t)}\ [x : \mathrm{TERM}]\ [y : \mathrm{TERMS}]\ [v : \mathrm{VAR}]\ \begin{bmatrix} c : \mathrm{CONST} \\ u : \mathrm{TERMS} \\ q : Q(u) \end{bmatrix}\ \begin{bmatrix} w : \mathrm{TERM} \\ z : \mathrm{TERMS} \\ r : P(w) \\ s : Q(z) \end{bmatrix}$$

Figure 5.6) *TERMS* Elim

$$\frac{ts : \mathrm{TERMS} \quad P(x)\ \mathrm{type} \quad Q(y)\ \mathrm{type} \quad d(v) : P(\mathrm{Var}(v)) \quad e(c, u, q) : P(\mathrm{App}(c, u)) \quad f : Q(\mathrm{None}) \quad g(w, z, r, s) : Q(\mathrm{Some}(w, z))}{\mathrm{termsrec}(d, e, f, g, ts) : Q(ts)}\ [x : \mathrm{TERM}]\ [y : \mathrm{TERMS}]\ [v : \mathrm{VAR}]\ \begin{bmatrix} c : \mathrm{CONST} \\ u : \mathrm{TERMS} \\ q : Q(u) \end{bmatrix}\ \begin{bmatrix} w : \mathrm{TERM} \\ z : \mathrm{TERMS} \\ r : P(w) \\ s : Q(z) \end{bmatrix}$$

Figure 5.7) A *termrec* = rule

$$\frac{p : \mathrm{VAR} \quad P(x)\ \mathrm{type} \quad Q(y)\ \mathrm{type} \quad d(v) : P(\mathrm{Var}(v)) \quad e(c, u, q) : P(\mathrm{App}(c, u)) \quad f : Q(\mathrm{None}) \quad g(w, z, r, s) : Q(\mathrm{Some}(w, z))}{\mathrm{termrec}(d, e, f, g, \mathrm{Var}(p)) = d(p) : P(\mathrm{Var}(p))}\ [x : \mathrm{TERM}]\ [y : \mathrm{TERMS}]\ [v : \mathrm{VAR}]\ \begin{bmatrix} c : \mathrm{CONST} \\ u : \mathrm{TERMS} \\ q : Q(u) \end{bmatrix}\ \begin{bmatrix} w : \mathrm{TERM} \\ z : \mathrm{TERMS} \\ r : P(w) \\ s : Q(z) \end{bmatrix}$$

Figure 5.8) The other *termrec* = rule

$$\frac{[x : \mathrm{TERM}]\ [y : \mathrm{TERMS}]\quad [v : \mathrm{VAR}]\quad \begin{bmatrix} c : \mathrm{CONST} \\ u : \mathrm{TERMS} \\ q : Q(u) \end{bmatrix} \qquad \begin{bmatrix} w : \mathrm{TERM} \\ z : \mathrm{TERMS} \\ r : P(w) \\ s : Q(z) \end{bmatrix}}{\begin{array}{c} a : \mathrm{CONST}\quad b : \mathrm{TERMS}\quad P(x)\ \mathrm{type}\quad Q(y)\ \mathrm{type}\quad d(v) : P(\mathrm{Var}(v))\quad e(c, u, q) : P(\mathrm{App}(c, u))\quad f : Q(\mathrm{None})\quad g(w, z, r, s) : Q(\mathrm{Some}(w, z)) \\ \hline \mathrm{termrec}(d, e, f, g, \mathrm{App}(a, b)) = e(a, b, \mathrm{termsrec}(d, e, f, g, b)) : P(\mathrm{App}(a, b)) \end{array}}$$

Figure 5.9) A *termsrec* = rule

$$\frac{[x : \mathrm{TERM}]\ [y : \mathrm{TERMS}]\quad [v : \mathrm{VAR}]\quad \begin{bmatrix} c : \mathrm{CONST} \\ u : \mathrm{TERMS} \\ q : Q(u) \end{bmatrix} \qquad \begin{bmatrix} w : \mathrm{TERM} \\ z : \mathrm{TERMS} \\ r : P(w) \\ s : Q(z) \end{bmatrix}}{\begin{array}{c} P(x)\ \mathrm{type}\quad Q(y)\ \mathrm{type}\quad d(v) : P(\mathrm{Var}(v))\quad e(c, u, q) : P(\mathrm{App}(c, u))\quad f : Q(\mathrm{None})\quad g(w, z, r, s) : Q(\mathrm{Some}(w, z)) \\ \hline \mathrm{termsrec}(d, e, f, g, \mathrm{None}) = f : Q(\mathrm{None}) \end{array}}$$

Figure 5.10) The other *termsrec* = rule

$$\frac{[x : \mathrm{TERM}]\ [y : \mathrm{TERMS}]\quad [v : \mathrm{VAR}]\quad \begin{bmatrix} c : \mathrm{CONST} \\ u : \mathrm{TERMS} \\ q : Q(u) \end{bmatrix} \qquad \begin{bmatrix} w : \mathrm{TERM} \\ z : \mathrm{TERMS} \\ r : P(w) \\ s : Q(z) \end{bmatrix}}{\begin{array}{c} a : \mathrm{TERM}\quad b : \mathrm{TERMS}\quad P(x)\ \mathrm{type}\quad Q(y)\ \mathrm{type}\quad d(v) : P(\mathrm{Var}(v))\quad e(c, u, q) : P(\mathrm{App}(c, u))\quad f : Q(\mathrm{None})\quad g(w, z, r, s) : Q(\mathrm{Some}(w, z)) \\ \hline \mathrm{termsrec}(d, e, f, g, \mathrm{Some}(a, b)) = g(a, b, \mathrm{termrec}(d, e, f, g, a), \mathrm{termsrec}(d, e, f, g, b)) : Q(\mathrm{Some}(a, b)). \end{array}}$$

Figure 5.11) The *treerec* = rule

$$\left[\begin{array}{c} x : A \\ t : \mathcal{T}(x) \end{array}\right] \qquad [z : C(a, b)] \qquad \left[\begin{array}{c} x : A \\ y : B(x) \\ z(v) : \mathcal{T}(d(x, y, v)) \ [v : C(x, y)] \\ u(v) : D(d(x, y, v), z(v)) \ [v : C(x, y)] \end{array}\right]$$

$$\frac{D(x, t) : \text{type} \quad a : A \quad b : B(a) \quad c(z) : \mathcal{T}(d(a, b, z)) \qquad f(x, y, z, u) : D(x, \text{tree}(x, y, z))}{\text{treerec}(f, \text{tree}(a, b, c)) = f(a, b, c, (x)\text{treerec}(f, c(x))) : D(a, \text{tree}(a, b, c))}$$

Figure 5.12)

$$[z : \{hd, tl\}]$$

$$\frac{t : \mathcal{T}(TM) \quad ts : \mathcal{T}(TMS) \quad TMS : \{TM, TMS\} \quad S : \{N, S\} \quad \text{case}_{\{hd, tl\}}(t, ts, z) : \mathcal{T}(\text{case}_{\{hd, tl\}}(TM, TMS, z))}{\text{tree}(TMS, S, \text{case}_{\{hd, tl\}}(t, ts)) : \mathcal{T}(TMS)}$$

Figure 5.13)

$$[x : TERM] \quad [y : TERMS] \quad [v : VAR] \quad \left[\begin{array}{c} c : CONST \\ d : TERMS \\ q : Q(d) \end{array}\right] \quad \left[\begin{array}{c} w : TERM \\ z : TERMS \\ r : P(w) \\ s : Q(z) \end{array}\right]$$

$$\frac{t : TERM \quad P(x) \text{ type} \quad Q(y) \text{ type} \quad \alpha(v) : P(Var(v)) \quad \beta(c, d, q) : P(App(c, d)) \quad \chi : Q(None) \quad \delta(w, z, r, s) : Q(Some(w, z))}{\text{termrec}(\alpha, \beta, \chi, \delta, t) : P(t)}$$

Chapter 6

Well-founded Recursion In Martin-Löf's Type Theory

In this chapter we discuss the use of well-founded recursion in M-LTT. The non-canonical constant for an inductively defined type is a structural recursion operator. Functions over the type are defined in terms of this constant. There is one outstanding feature of using the structural recursion operators: every well-typed expression strongly normalises, i.e. evaluation of well-typed expressions always terminates. Functions are total.

There are, however, many algorithms which are not most naturally expressed using structural recursion. In particular there are many functions that are naturally expressed using well-founded recursion. The unification algorithm that we shall present later is such an algorithm.

6.1 Describing Well-foundedness

There are several definitions in the literature of the well-foundedness of an ordering. The classical version is in terms of the non-existence of infinite descending sequences. We ignore this as not allowing constructive proofs by (well-founded) induction or constructions by (well-founded) recursion. There are several constructive versions:

i) Thompson presents one in §7.9 of [Tho91], formalising the usual idea of proof by well-founded induction;

ii) Paulson [Pau84] defines well-foundedness in terms of various rules 'holding' and gives an alternative which uses subsets (which we assume can be replaced by Σ types);

iii) Thompson states (Theorem 7.16) and Paulson states and proves the theorem that an ordering on a type is well-founded iff it is the inverse image of the canonical well-founded ordering on a well-ordering, where the definition of the canonical well-founded ordering on a well-ordering is as on page 36 of [Pau84];

iv) Nordström, in [Nor88], presents the type $Acc(A,<)$, the accessible elements of $<$ in A and defines well-foundedness in terms of this;

v) Saaman and Malcolm, in [SM87], extend the work of [Nor88] by internalising the membership relation and hence deriving new versions of the elimination rules.

The two which are of most interest to us are Paulson's characterisation of well-foundedness using well-orderings and Nordström's using *Acc* types. Nordström's use of *Acc* types is criticised both by Thompson and by Saaman and Malcolm. It is beyond the scope of this thesis to investigate these criticisms in detail. The rules that Nordström presents allow a relatively natural expression of an algorithm and are convenient to work with. We shall assume that Nordström and Paulson are describing the same concept and that we may make use of Nordström's rules even if we use what is essentially Paulson's characterisation of well-foundedness.

6.1.1 Defining well-foundedness using well-orderings

A *partial ordering* on a small type, *A*, is a small-type-valued binary function, $<_A$, on *A* for which the following types are inhabited:

$\prod(x : A, \neg(x <_A x))$
$\prod(x : A, \prod(y : A, \prod(z : A, x <_A y \,\&\, y <_A z \rightarrow x <_A z)))$

Well-orderings are described in [Mar84]. Paulson [Pau84] defines the canonical well-founded ordering on a well-ordering as:

$v' <_W sup(a, f) =_{def} \{y \in B(a) \mid v' =_W f(y)\}$

We replace the use of the subset type with a Σ type and make use of *wrec* to define the canonical well-founded ordering $<_W$ on a well-ordering *W(A, B)* as follows:

$<_W(v', v) =_{def} wrec((x, y, z)\Sigma(u : B(x), Eq(W(A, B), v', y(u))), v).$

We can now proceed to use Paulson's characterisation of an ordering on a type as well-founded iff it is the inverse image of the canonical well-founded ordering on a well-ordering:

$$
\begin{aligned}
WF(A, <_A) =_{def}\ & Order(A, <_A)\ \&\\
& \Sigma(C : U_0,\\
& \quad \Sigma(B : C \rightarrow U_0,\\
& \quad\quad \Sigma(f : A \rightarrow W(C, B),\\
& \quad\quad\quad \prod(a : A,\\
& \quad\quad\quad\quad \prod(a' : A,\\
& \quad\quad\quad\quad\quad (a <_A a') \leftrightarrow (f(a) <_W f(a'))\\
& \quad\quad\quad\quad)\\
& \quad\quad\quad)\\
& \quad\quad)\\
& \quad)\\
&)
\end{aligned}
$$

6.1.2 Nordström's rule

In [Nor88] Nordström presents the type $Acc(A,<)$, the accessible elements of $<$ in A, with introduction and elimination rules as follows:

$$\left[\begin{array}{l} y : A \\ y < a \end{array}\right]$$

$$\frac{a : A \qquad y : Acc(A, <)}{a : Acc(A, <)}\ \text{Acc Introduction}$$

$$\left[\begin{array}{c} x : Acc(A, <) \\ y(z) : C(z) \left[\begin{array}{l} z : A \\ z < x \end{array}\right] \end{array}\right]$$

$$\frac{p : Acc(A, <) \qquad e(x, y) : C(x)}{rec(e, p) : C(p)}\ \text{Acc Elimination}$$

The computation rule for *rec* is:

$$\frac{e(p, rec(e)) \dashrightarrow e'}{rec(e, p) \dashrightarrow e'}$$

In [Nor88] Nordström works in a type theory with propositions. In order to follow his presentation we have used the propositions $y<a$ and $z<x$ in the above rules.

Nordström states that a partial ordering $<$ on a type A is *well-founded* if A is the same as$Acc(A,<)$.

[Nor88] derives the following rule:

$$\left[\begin{array}{c} x : A \\ y(z) : C(z) \left[\begin{array}{l} z : A \\ z < x \end{array}\right] \end{array}\right]$$

$$\frac{\omega : Wellfnd(A, <) \quad p : A \qquad e(x, y) : C(x)}{rec(e, p) : C(p)}$$

This is the rule which we shall make use of in Chapters 9 and 10.

6.2 Lemmas about well-founded orderings

We require some lemmas about well-founded orderings.

6.2.1 Lemma 6.1

The *lexicographic product* $<_{1*2}$ of two partial orderings $<_1$, $<_2$ on A, A_1 resp. is defined as follows:

$$a <_{1*2} b =_{def} fst(a) <_1 fst(b) \vee (fst(a) = fst(b) \,\&\, snd(a) <_2 snd(b))$$

The lexicographic product of two well-founded orderings is itself a well-founded ordering.

Proof:

See [Pau84]. ■

6.2.2 Lemma 6.2

If $<_A$ is a well-founded ordering on A and we have a function *norm* of type B -> A then the relation $<_B$ on B defined as

$$b <_B b' =_{def} norm(b) <_A norm(b')$$

is a well-founded ordering called the *inverse image* of $<_A$ under *norm*.

Proof

See [Tho91] or [Pau84]. ■

6.3 A particular well-founded ordering

In the following sections we will present a binary relation on $\mathcal{T}(a)$ *PAIR* and prove that it is a well-founded ordering. We will use this well-founded ordering to prove the termination of our implementation of the unification algorithm. The particular well-founded ordering that we use was chosen after contemplation of the algorithm that we will present in Chapter 8 and the orderings used in [MW81] and [Pau85b].

We presume that the type of natural numbers has been defined and that *0*, *1*, and *+* have their usual meanings.

We begin by defining a function, *norm*:

norm(ts) $=_{def}$
Pair(num_vars_occing(fst(ts)) + num_vars_occing(snd(ts)), size(fst(ts)))

Where:

num_vars_occing $=_{def}$
(t)(listrec(0, (hd, tl, r)(if in(hd, tl) then r else r + 1), varsin(t))
size $=_{def}$ termsrec((v)1, (f, ts, a)(1 + a), 0, (h, tl, a, b)(a + b))
varsin $=_{def}$ termrec((v)[v], (f, ts, r)r, [], (t, ts, r, s)(r @ s))
in(x, l) $=_{def}$ listrec(false, (hd, tl, r)(if x = hd then true else r), l)
l1 @ l2 $=_{def}$ listrec(l2, (h, tl, r)(h :: r), l1)
if boolean then t else e $=_{def}$ boolrec(t, e, boolean)

Listrec is the non-canonical constant associated with the type of lists. We define *listrec* so that its third argument is a list. We present the computation rules for *listrec* in section 6.2.2 below. *Boolrec* is the non-canonical constant associated with the type of booleans. We define *boolrec* so that its third argument is a boolean.

num_vars_occing counts the number of variables which occur in a term. *size* is a measure of the size of term. *varsin* returns a list (including duplicates) of the variables which occur in a term. @ is the usual append operator. Notice that the definition of *in* relies on the fact that we have a decidable equality for the type involved, in this case *VAR*.

We define a well-founded ordering on *N*, the type of the natural numbers.

$n <_N m =_{def} \Sigma(p : N, m = n + Succ(p))$.

By lemma 6.1 the lexicographic product of $<_N$ and $<_N$ is a well-founded ordering on $N * N$.

By lemma 6.2 $<$, the inverse image under *norm* of the lexicographic product of $<_N$ and $<_N$ is a well-founded-ordering on $\mathcal{T}(a)$ *PAIR*.

6.4 Using the general recursion operator

[Nor88] states (Theorem 1):

> All iterating constructs in type theory can be reduced to pattern matching and the general recursion operator rec.

He proves this by illustrating it in the case of *natrec*, leaving *listrec* as an exercise for the reader. We shall show that this theorem holds for *listrec* and for the types of *TERM* and *TERMS* that we introduced in Chapter 5.

6.4.1 *Listrec* in terms of *listcases* and *rec*

The computation rules for *listcases* are:

$$\frac{l \dashrightarrow [] \qquad d \dashrightarrow d'}{\text{listcases}(d,\ e,\ l) \dashrightarrow d'}$$

$$\frac{l \dashrightarrow h::t \qquad e(h,\ t) \dashrightarrow e'}{\text{listcases}(d,\ e,\ l) \dashrightarrow e'}$$

The computation rules for *listrec* are:

$$\frac{l \dashrightarrow [] \qquad d \dashrightarrow d'}{\text{listrec}(d,\ e,\ l) \dashrightarrow d'}$$

$$\frac{l \dashrightarrow h::t \qquad e(h,\ t,\ \text{listrec}(t,\ d,\ e)) \dashrightarrow e'}{\text{listrec}(d,\ e,\ l) \dashrightarrow e'}$$

We define a new constant *listrec'* using *rec* and *listcases* and show that it has the same computational behaviour as *listrec*.

$\text{listrec}'(p, q, l) =_{\text{def}} \text{rec}((x, y)\text{listcases}(p, (z, w)q(z, w, y(w)), x), l)$

In the following let e be *(x, y)listcases(p, (z, w)q(z, w, y(w)), x)*.

We can use the computation rules for *rec* and *listcases* and the definitions e and of *listrec'* to show the following:

$$\cfrac{\cfrac{\cfrac{\cfrac{l \dashrightarrow [] \qquad p \dashrightarrow p'}{\text{listcases}(p,\ (z,\ w)q(z,\ w,\ \text{rec}(e,\ w)),\ l) \dashrightarrow p'}}{e(l,\ \text{rec}(e)) \dashrightarrow p'}}{\text{rec}((x,\ y)\text{listcases}(p,(z,\ w)q(z,w,y(w),x)),\ l) \dashrightarrow p'}}{\text{listrec}'(p,\ q,\ l) \dashrightarrow p'}$$

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{l \dashrightarrow h :: t \qquad q(h,\ t,\ \text{listrec}'(p,\ q,\ t)) \dashrightarrow q'}{q(h,\ t,\ \text{rec}(e,\ t)) \dashrightarrow q'}}{(z,\ w)q(z,\ w,\ \text{rec}(e,\ w))(h,\ t) \dashrightarrow q'}}{\text{listcases}(p,\ (z,\ w)q(z,\ w,\ \text{rec}(e,\ w)),\ l) \dashrightarrow q'}}{e(l,\ \text{rec}(e)) \dashrightarrow q'}}{\text{rec}((x,\ y)\text{listcases}(p,(z,\ w)q(z,w,y(w),x)),\ l) \dashrightarrow q'}}{\text{listrec}'(p,\ q,\ l) \dashrightarrow q'}$$

Thus we have shown that *listrec'* has the same computational behaviour as *listrec*. Notice that we cannot show, as we might like to, that *listrec(p, q, l) = listrec'(p, q, l)* using only the computation rules. To show that this is the case we need to have available to us the judgements which are the premisses of the elimination rules.

6.4.2 Example

[Nor88] presents a version of *quicksort*. We may express this in ML as:

```
(*
quick : ('a -> ('a -> bool)) -> (('a list) -> ('a list))
*)
fun quick anorder [] = []
  | quick anorder (hd :: tl) =
              (quick (less hd anorder tl))
            @ [hd]
            @ (quick (gteqs hd anorder tl));
```

Where:

> *less a < bs* returns those elements of *bs* less than *a* under the ordering < ;
> *gteqs a < bs* returns those elements of *bs* greater than or equal to *a* under the ordering < .

The function *quick* is not expressed in structurally recursive form as the recursive calls are not on the (structural) sub-parts of the list that the initial call is made on. These calls are, however, made on lists which are certain to be shorter than the initial list. The ordering of length on lists is well-founded and so we can be certain that evaluation of *quick theorder thelist* will terminate. The other functions used can easily be written in structurally recursive form.

We can now handle *quicksort* by defining (again in ML syntax for clarity):

```
(*
listcases : 'b -> (('a -> (('a list) ->'b)) -> (('a list) -> 'b))
*)
fun listcases d e [] = d
  | listcases d e (h::tl) = e h tl;

(*
nordrec : (('a * ('a -> 'b)) -> 'b) -> ('a -> 'b)
Name avoids clash with ML's own rec.
*)
fun nordrec e p = e(p, fn q => nordrec e q);
```

```
(*
wfquick : ('a -> ('a -> bool)) -> (('a list) -> ('a list))
*)
val wfquick =
            fn anorder =>
                  nordrec
                  fn q =>
                        listcases
                              []
                              (fn hd =>
                               fn tl =>
                                    (q (less hd anorder tl))
                                    @ [hd]
                                    @ (q (gteqs hd anorder tl)));
```

We shall later use the same technique to handle the well-founded recursion in the unification algorithm.

6.4.3 *Termsrec* and *termrec* in terms of a case operator and *rec*

To define *termrec* and *termsrec* in terms of *rec* and a case operator we must be a little bit more subtle.

The computation rules for *termrec* and *termsrec* are:

$$\frac{t \text{ --> } Var(x) \quad d(x) \text{ --> } d'}{termrec(d,\ e,\ f,\ g,\ t) \text{ --> } d'}$$

$$\frac{t \text{ --> } App(c,\ ts) \quad e(c,\ ts,\ termsrec(d,\ e,\ f,\ g,\ ts)) \text{ --> } e'}{termrec(d,\ e,\ f,\ g,\ t) \text{ --> } e'}$$

$$\frac{ts \text{ --> } None \quad f \text{ --> } f'}{termsrec(d,\ e,\ f,\ g,\ ts) \text{ --> } f'}$$

$$\frac{ts \text{ --> } Some(s,\ ss) \quad g(s,\ ss,\ termrec(d,\ e,\ f,\ g,\ s),\ termsrec(d,\ e,\ f,\ g,\ ss)) \text{ --> } g'}{termsrec(d,\ e,\ f,\ g,\ ts) \text{ --> } g'}$$

The obvious computation rules for *termcases* and *termscases* are:

$$\frac{t \text{ --> } Var(x) \quad d(x) \text{ --> } d'}{termcases(d,\ e,\ t) \text{ --> } d'}$$

$$\frac{t \text{ --> } App(c,\ ts) \quad e(c,\ ts) \text{ --> } e'}{termcases(d,\ e,\ t) \text{ --> } e'}$$

$$\frac{ts \text{ --> } None \quad f \text{ --> } f'}{termscases(f,\ g,\ ts) \text{ --> } f'}$$

$$\frac{ts \dashrightarrow Some(s,\ ss) \qquad g(s,\ ss) \dashrightarrow g'}{termscases(f,\ g,\ ts) \dashrightarrow g'}$$

However if we follow the above work on *listrec* and define:

putative_termrec'(p, q, t) $=_{def}$ rec((x, y)termcases(p, (z, w)q(z, w, y(w)), x), t)

then work as before, letting e stand for

(x, y)termcases(p, (z, w)q(z, w, y(w)), x)

we find that for the case that *t* has the value *App(f,ts)* the value of *putative_termrec'(p,q,t)* depends on the value of *q(f,ts,rec(e,ts))*. This is unfortunate as we do not know how to evaluate *rec(e,ts)*. We must instead define a case operator *term_or_termscases* which has the following computation rules:

$$\frac{t \dashrightarrow Var(x) \qquad d(x) \dashrightarrow d'}{term\_or\_termscases(d,\ e,\ f,\ g,\ t) \dashrightarrow d'}$$

$$\frac{t \dashrightarrow App(c,\ ts) \qquad e(c,\ ts) \dashrightarrow e'}{term\_or\_termscases(d,\ e,\ f,\ g,\ t) \dashrightarrow e'}$$

$$\frac{ts \dashrightarrow None \qquad f \dashrightarrow f'}{term\_or\_termscases(d,\ e,\ f,\ g,\ ts) \dashrightarrow f'}$$

$$\frac{ts \dashrightarrow Some(s,\ ss) \qquad g(s,\ ss) \dashrightarrow g'}{term\_or\_termscases(d,\ e,\ f,\ g,\ ts) \dashrightarrow g'}$$

We now make the following definitions:

```
termrec'(p, q, r, s, t) =def
        rec((x, y)term_or_termscases(
                                        p,
                                        (v, w)q(v, w, y(w)),
                                        r,
                                        (z, n)s(z, n, y(z), y(n)),
                                        x),
            t)
```

```
termsrec'(p, q, r, s, ts) =def
        rec((x, y)term_or_termscases(
                                        p,
                                        (v, w)q(v, w, y(w)),
                                        r,
                                        (z, n)s(z, n, y(z), y(n)),
                                        x),
            ts)
```

Again, notice that *termrec'* and *termsrec'* are definitionally equal.

We can now show that *termrec'* and *termsrec'* have the same computational behaviour as *termrec* and *termsrec*, respectively. Figures 6.1, 6.2, 6.3 and 6.4 are the proofs.

We have shown that *termrec'* and *termsrec'* have the same computational behaviour as *termrec* and *termsrec* respectively and hence any function that we can express using *termrec* and *termsrec* can be expressed using *rec* and *term_or_termscases*.

6.5 Summary

In this chapter we have presented some approaches to the use of well-founded recursion in Martin-Löf's type theory, in particular Nordström's rule for well-founded induction and his *rec* operator. We have shown that a particular ordering on $\mathcal{T}(a)$ *PAIR* is well-founded. Therefore we can define functions on $\mathcal{T}(a)$ *PAIR* using well-founded recursion. We have also shown how to use *rec* and a case operator to define the structural recursion operator on terms, thus allowing us to use only *rec* and the case operator to define functions, as we will do in Chapter 8.

In the following proofs let e be *(x, y)term_or_termscases(p, (v, w)q(v, w, y(w)), r, (z, n)s(z, n, y(z), y(n)), x)*. In the interests of clarity we have suppressed some of the terms, replacing them with ellipsis.

Figure 6.1)

$$\dfrac{\dfrac{\dfrac{\dfrac{t \text{ --> } Var(v) \qquad p(v) \text{ --> } p'}{\text{term\_or\_terms\_cases(p, ..., ..., ..., t) --> p'}}\text{term\_or\_terms\_cases computation}}{\text{e(t, rec(e)) --> p'}}\text{defn of e}}{\text{rec((x, y)term\_or\_termscases(p, ..., ..., ..., x), t) --> p'}}\text{rec computation}}{\text{termrec'(p, q, r, s, t) --> p'}}\text{defn of termrec'}$$

The steps in this proof all consist of re-writing expressions based on the appropriate definitions or the use of the computation rules for *rec* or *term_or_terms_cases*. The other proofs that we present follow the same pattern, but we will suppress the names of the rules used at each step.

Figure 6.2)

$$\dfrac{\dfrac{\dfrac{t \text{ --> } App(f, ts) \qquad q(f, ts, termsrec'(p, q,r, s, ts)) \text{ --> } q'}{\text{term\_or\_termscases(..., (v, w)q(v, w, termsrec'(p,q,r,s)(w)), ..., ..., t) --> q'}}}{\text{e(t, rec(e)) --> q'}}}{\text{rec((x, y)term\_or\_termscases(..., (v, w)q(v, w, y(w)), ..., ..., x), t) --> q'}}$$
$$\text{termrec'(p, q, r, s, t) --> q'}$$

Figure 6.3)

$$\dfrac{\dfrac{\dfrac{\dfrac{ts \text{ --> } None \qquad\qquad r \text{ --> } r'}{term\_or\_termscases(..., r, ..., ..., ts) \text{ --> } r'}}{e(ts, rec(e)) \text{ --> } r'}}{rec((x, y)term\_or\_termscases(..., ..., r, ..., x), ts) \text{ --> } r'}}{termsrec'(p, q, r, s, ts) \text{ --> } r'}$$

Figure 6.4)

$$\dfrac{\dfrac{\dfrac{\dfrac{t \text{ --> } Some(h, ts) \qquad\qquad s(h, ts, termsrec'(p, q,r, s, h), termsrec'(p, q,r, s, ts)) \text{ --> } s'}{term\_or\_termscases(..., ..., ..., X, t) \text{ --> } p'}}{e(t, rec(e)) \text{ --> } s'}}{rec((x, y)term\_or\_termscases(..., ..., ...,(v, w)s(v, w, y(v), y(w)), x), t) \text{ --> } s'}}{termrec'(p, q, r, s, t) \text{ --> } s'}$$

Where X is the expression $(v, w)s(v, w, termrec'(p, q, r, s)(v), termsrec'(p, q, r, s)(w))$.

Chapter 7

Formalization Of The Problem

In this chapter we formalise the notion of substitution and some of the other notions discussed informally above. We discuss some propositions which may be a suitable specification for the unification algorithm. We shall select one of these to be treated as the specification that we subsequently work with.

7.1 The type of substitutions

We can now formalise some material from Chapter 2. As we stated before a substitution is a function from variables to terms. Therefore we make the definition:

SUBST $=_{def}$ VAR -> TERM

7.2 Definitions

We can now make some definitions.

The *extension* of a substitution is defined as:

σ *tm $=_{def}$
termrec((v)apply(σ, v), (f, ts, u)App(f, u), None, (t, ts, r, u)Some(r, u))

σ *tms $=_{def}$
termsrec((v)apply(σ, v) ,(f, ts, u)App(f, u), None, (t, ts, r, u)Some(r, u))

Notice that, because *termrec* and *termsrec* are definitionally equal, so are **tm* and **tms*. We will use * when we are free to choose either.

The *composition* of two substitutions is defined as:

$\sigma \bullet \tau =_{def}$ (x)(τ * (apply(σ, x)))

A substitution may be an *instance* of another:

τ INSTANCE $\sigma =_{def} \Sigma(v : SUBST, EQ(SUBST, \tau, \sigma \bullet v))$

This allows us to define the notion of *generality* for substitutions:

MORE_GENERAL(σ, τ) $=_{def}$ τ INSTANCE σ

Idempotence is defined as usual:

IDEMPOTENT_SUBST(σ) $=_{def}$ EQ(SUBST, σ, $\sigma \bullet \sigma$)

We say what it means to be a *unifier* of either a *TERM PAIR* or of a *TERMS PAIR*:

UNIFIERTM(tmpr, σ) $=_{def}$
    EQ(TERM, σ *tm fst(tmpr), σ *tm snd(tmpr))

UNIFIERTMS(tmspr, σ) $=_{def}$
    EQ(TERMS, σ *tms fst(tmspr), σ *tms snd(tmspr))

We can generalise this definition:

UNIFIES(p, σ, a) $=_{def}$ EQ($\mathcal{T}$(a), σ * fst(p), σ * snd(p))

We can now define:

UNIFIER(p, σ) $=_{def}$ Σ(a : {TM, TMS}, UNIFIES(p, σ, a))

We say what it means to be a *most general unifier* of either a *TERM PAIR* or of a *TERMS PAIR*:

MGUTM(tmpr, σ) $=_{def}$
    UNIFIERTM(tmpr, σ) &
    Π(τ : SUBST, UNIFIERTM(tmpr, τ) -> MORE_GENERAL(σ, τ))
MGUTMS(tmspr, σ) $=_{def}$
    UNIFIERTMS(tmspr, σ) &
    Π(τ : SUBST, UNIFIERTMS(tmspr, τ) -> MORE_GENERAL(σ, τ))

And again this definition is generalisable:

MGUA(tpr, σ, a) $=_{def}$
    UNIFIES(tpr, σ, a) &
    Π(τ : SUBST, UNIFIES(tpr, τ, a) -> MORE_GENERAL(σ, τ))

And we can define:

MGU(tpr, σ) $=_{def}$ Σ(a : {TM, TMS}, MGUA(p, σ, a))

We define the notion of *most general, idempotent unifier* of a *TERM PAIR* or a *TERMS PAIR*:

MGIU(tpr, σ) $=_{def}$ MGU(tpr, σ) & IDEMPOTENT_SUBST(σ)

We define the notion that a *TERM PAIR* or a *TERMS PAIR* has a most general idempotent unifier:

$$\text{HASMGIU}(tpr) =_{\text{def}} \Sigma(\sigma : \text{SUBST}, \text{MGIU}(tpr, \sigma))$$

Finally we define the notion of a *TERM PAIR* or a *TERMS PAIR* not being unifiable:

$$\text{NU}(tpr) =_{\text{def}} \neg\Sigma(\tau : \text{SUBST}, \text{UNIFIER}(tpr, \tau))$$

7.3 Putative specifications

There are two informal specifications that appear intuitively to be suitable. For convenience we use 'term' to refer to a value either of the type *TERM* or of the type *TERMS*. They are:

Specification 1

Two terms either have a unifier or else they don't and if two terms have a unifier then they have a most general unifier;

Specification 2

Either two terms have a most general unifier or they are not unifiable.

If we treat these two specifications as propositions then we see that they are constructively equivalent (remembering that if two terms have a most general unifier then they have a unifier). Their canonical proofs are different. A canonical proof of the first specification would be a function which when presented with two terms produced a pair consisting of a proof of whether they were unifiable or not, and a function which took a unifier of the terms and produced a most-general unifier. A canonical proof of the second would be a function which, when presented with two terms, either produced a most general unifier or a proof that no unifier exists. Our work is divided up in different ways with these different specifications. Whether it is easier to find whether a unifier exists and then to construct a most-general unifier or simply to attempt to construct a most general unifier is not *a priori* obvious.

Compare with the task of finding a normal proof. We may find it easier to attempt to find a non-normal proof and then apply some mechanical normalisation procedure than to attempt to find a normal proof directly. It has been our experience that searching for most-general unifiers is no worse than searching for unifiers, and furthermore we know of no technique for converting an arbitrary unifier of two terms into a most general unifier, barring the trivial one of simply calculating a most general unifier in the usual fashion. So we choose to work with the second specification, both because we think that it is a type of which we wish to construct an inhabitant and because we think that its inhabitants are the sort of objects that we will find easier to construct.

Notice that both of these specifications will produce positive evidence in the case that that the two terms are not unifiable. We claim not

merely that there is no most-general unifier, but there is no unifier whatsoever.

The type that this specification corresponds to is:

$$\prod(a : \{TM, TMS\}, \prod(tpr : \mathcal{T}(a)\ PAIR, HASMGIU(tpr) + NU(tpr)))$$

The (canonical) proof term that we will construct will then be a function which takes a pair of terms and returns either a substitution, together with a proof that it is a most general idempotent unifier of the pair of terms or else a method which will show that the existence of any substitution which unifies the two terms is absurd.

Chapter 8

Different Possible Algorithms

In this chapter we look at some different possible algorithms for computing most general unifiers. These algorithms have been chosen either because they illustrate classic approaches or because there is some interesting feature about them which relates to proving properties of them. Each of these algorithms makes different assumptions about the representation of terms and of substitutions. We leave these assumptions unstated unless they are of interest. We shall combine one of the algorithms in this chapter with the specification discussed in the previous chapter.

8.1 Chang and Lee's unification algorithm

The first algorithm that we will inspect is that which is presented in [CL73]. This algorithm unifies sets (not pairs) of terms or declares them not to be unifiable. The process operates by explicitly finding where the terms *disagree* and attempting to eliminate this disagreement. In the case that the terms are unifiable a most general unifier is found (and the term of which the others are instances), in the case that they are not unifiable a set of terms that cannot be unified is explicitly constructed. The presentation in [CL73] is in an informal imperative style. We follow this.

The algorithm unifies a set of terms W. Note that W must have at least *two* members for the algorithm to make sense. The algorithm is presented as follows:

Step 1 Set $k = 0$, $W_k = W$ and $\sigma_k = \varepsilon$ (* the empty substitution *).

Step 2 If W_k is a singleton (* impossible for $k = 0$ *) then σ_k is a most general unifier of W. Stop. Otherwise find D_k, the disagreement set of W_k.

Step 3 If there is a term t_k in D_k and a variable v_k in D_k such that v_k does not occur in t_k then go to Step 4. Otherwise stop as W is not unifiable.

Step 4 Let $\sigma_{k+1} = \sigma_k \bullet (v_k \text{ --> } t_k)$ and $W_{k+1} = (v_k \text{ --> } t_k) * W_k$.

Step 5 Set $k = k + 1$ and go to Step 2.

The disagreement set D_k of W_k is found by

> locating the first symbol (counting from the left) at which not all the expressions in W_k have exactly the same symbol, and then extracting from each expression in W_k the subexpression that begins with the symbol occupying that position.

Notice that this algorithm is non-deterministic: we have a free choice at Step 3 if there is more than one suitable variable and term. Constructing the whole disagreement set in Step 2 will lead to failure quickly but may produce unnecessary effort in the case that the two terms do unify. The structure of the algorithm does not reflect the structure of terms and so is unlikely to be easy express or prove correct in Martin-Löf's type theory.

8.2 Apt's algorithm

The following algorithm is presented in [Apt90], and is based on that appearing in [Her71]. This algorithm unifies sets of equations between terms. We may think of these as sets of pairs of terms. A set of equations of the form $[x_1 = t_1, \ldots, x_n = t_n]$ is called a *solved* set if all the x_i are distinct and none of the x_i occurs in any of the t_i. We see that a solved set of equations determines a substitution. The algorithm proceeds by transforming the existing equations into simpler ones until either a solved set is found or no more transformations are possible. In the case of success the most general unifier is then read off the solved set. The algorithm is as follows:

Pick an equation from the set. In each of the following cases of the equation perform the action stated:

1) $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$. Replace this equation by $s_1 = t_1$, $\ldots$ $s_n = t_n$;

2) $f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m)$ where $f \neq g$. Halt with failure.

3) $x = x$. Delete the equation.

4) $t = x$ where t is not a variable. Replace with $x = t$.

5) $x = t$ where $x \neq t$ and x has another occurrence in the set of equations. If x occurs in t then halt with failure otherwise perform the substitution $x \rightarrow t$ in every other equation.

The algorithm halts either when failure is detected or when no further steps can be performed.

It is presumed that each functor is applied to the same number of arguments where ever it is used

Clause 5 needs to be treated with care. We must read '*x* has another occurrence in the set of equations' to include the possibility that the occurrence is in the term *t*.

An informal proof of termination is given based on the observation that either the number of variables appearing in the problem is reduced at each step, or this number stays constant and that the number of occurrences of functor symbols is reduced.

This algorithm is also unsuitable for a functional presentation and is therefore not of further interest to us.

8.3 A poor function and a better one

Now we shall look at some variants of unification algorithms written in a way which will be more suitable for our purposes. We shall unify pairs of terms. We shall construct substitutions directly. The algorithms will be written using case analysis on the structure of pairs of terms. For convenience we will initially express the algorithms in ML ([Wik87], [Pau91])

We make the following type declarations:

```
type VAR = string;
type CONST = string;

datatype TERM = Var of VAR | App of CONST * TERMS
and TERMS = None | Some of TERM * TERMS;

type SUBST = VAR -> TERM;
```

We can now (given definitions of some auxiliary functions) define the following function:

```
(*
mgu1 : TERM -> (TERM -> SUBST)
*)
fun mgu1 (Var x) (Var y) =
                  x --> Var y
  | mgu1 (Var x) (App(g, ss)) =
                  if occurs x ss          (* the occurs check *)
                        then raise occurs_1
                        else x --> App(g, ss)
  | mgu1 (App(f, ts)) (Var y) =
                  if occurs y ts
                        then raise occurs_2
                        else y --> App(f, ts)
  | mgu1 (App(f, None)) (App(g, None)) =
                  if f = g
                        then empty  (* the empty substitution *)
                        else raise functors
  | mgu1 (App(f, None)) (App(g, Some(s, ss))) =
                  raise arities_1
  | mgu1 (App(f, Some(t, ts))) (App(g, None)) =
                  raise arities_2
  | mgu1 (App(f, Some(t, ts))) (App(g, Some(s, ss))) =
                  if f = g
                  then  let sigma = (mgu1 t s)
                        in
                        sigma
                        composed_with (* composing substitutions*)
                        (mgu1
                           App("dummy", mapterms (sigma * ) ts)
                           App("dummy", mapterms (sigma * ) ss)
                        )
                        end
                  else raise functors;
```

The auxiliary functions that we presume to be defined are as follows:

```
(*
mapterms : (TERM -> TERM) -> (TERMS -> TERMS) maps a function over
TERMs over TERMSs

--> is an infix operator.
--> : (VAR * TERM) -> SUBST.

* is an infix operator.
* : (SUBST * TERM) -> TERM) is the extension of a substitution.

composed_with : (SUBST * SUBST) -> SUBST composes two
substitutions.

The exceptions functors, arities_1, arities_2, occurs_1 and
occurs_2 have been declared.
*)
```

This function is rather odd as we start to deal with pseudo-terms (formed with the pseudo-functor "dummy": we could, of course, have chosen to use one of the functors already in the problem as a pseudo-functor) in the middle of it. These pseudo-terms are only being used as holders for lists of terms and seem absurd. We also seem to perform a lot of un-needed checks on whether two functors are the same. The solution to both these problems is, of course, the same. We have written the wrong function.

As we saw in Chapter 6 we need to consider the two types of *TERM* and *TERMS* simultaneously. So we should expect to define mutually recursive functions.

The functions that we should have defined are:

```
(*
mgu2 : TERM -> (TERM -> SUBST)
and
mgu2all : TERMS -> (TERMS -> SUBST)
*)
fun mgu2 (Var x) (Var y) =
                  x --> Var y
  | mgu2 (Var x) (App(g, ss)) =
                  if occurs x ss          (* the occurs check *)
                        then raise occurs_1
                        else x --> App(g, ss)
  | mgu2 (App(f, ts)) (Var y) =
                  if occurs y ts
                        then raise occurs_2
                        else y --> App(f, ts)
  | mgu2 (App(f, ts)) (App(g, ss)) =
                  if f = g
                        then mgu2all ts ss
                        else raise functors

and mgu2all None None = empty
  | mgu2all (Some(t, ts)) None = raise arities_1
  | mgu2all None (Some(s, ss)) = raise arities_2
  | mgu2all (Some(t, ts)) (Some(s, ss)) =
                  let val sigma = (mgu2 t s)
                  in
                        sigma
                        composed_with
                        (mgu2all
                              (mapterms (sigma * ) ts)
                              (mapterms (sigma * ) ss)
                        )
                  end;
```

Now we have a pair of mutually recursive functions which reflect clearly the structure of the terms that we are unifying. Notice that there is no sleight of hand using a convenient constructor to 'coerce' some structure into the type that we wished it had been in. Notice also that this pair of functions allows us to view ourselves as unifying either a

pair of terms or a list of pairs of terms as we fancy. This algorithm will form the basis for that which we shall implement in M-LTT.

8.4 A function that unifies Paulson's terms

For comparison we present a function that is suggested to construct most general unifiers for terms like those of [Pau85b].

We define the following types:

```
datatype PAUTERM = Varp of VAR |
                   Constp of CONST |
                   Appp of PAUTERM * PAUTERM;

type PAUSUBST = VAR -> PAUTERM;
```

We take the liberty of reusing some of the symbols used above for functions involving *TERM*, *TERMS* and *SUBST* for the analogues using *PAUTERM* and *PAUSUBST*.

```
(*
mgupau : PAUTERM -> (PAUTERM -> PAUSUBST)
*)
fun mgupau (Varp x) (Varp y) = x --> Varp y
  | mgupau (Varp x) (Constp g) = x --> Constp g
  | mgupau (Varp x) (Appp(t, s)) =
                  if occurs x Appp(t, s)
                        then raise occurs_1
                        else x --> Appp(t, s)
  | mgupau (Constp f) (Varp x) = x --> Constp f
  | mgupau (Constp f) (Constp g) =
                  if f = g
                        then empty
                        else raise functors
  | mgupau (Constp f) (Appp(t, s)) = raise arities_1
  | mgupau (Appp(t, s)) (Varp x) =
                  if occurs x Appp(t, s)
                        then raise occurs_2
                        else x --> Appp(t, s)
  | mgupau (Appp(t, s)) (Constp g) = raise arities_1
  | mgupau (Appp(t, s)) (Appp(v, w)) =
                  let sigma = (mgupau t v)
                  in sigma
                     composed_with
                     mgupau (sigma * s) (sigma * w)
                  end;
```

We reiterate the comment that there is a certain burden of proof that this algorithm and these terms are an acceptable substitute for our intuitive notions. We also suggest that this algorithm does not look *a priori* substantially simpler to reason about than the pair of mutually recursive functions that we presented above.

8.5 Expressing our algorithm in M-LTT

We shall now explain how to express our preferred algorithm in M-LTT. We shall use the types of *TERM* and *TERMS* that we discussed in Chapter 5. We use the technique described in Chapter 6 which allows us to use pattern matching and well-founded recursion.

The functions which we have discussed above present us with only a substitution or failure. The function which we define now must present us with both a substitution and a proof that the substitution is a most general idempotent unifier of the terms concerned or with a proof that the terms are not unifiable. In the following definitions we will use lower case Greek letters α, β, χ, δ to represent the formal proof terms that witness the most general and idempotent property of the substitution. The proof that we present in Chapter 9 will provide us with the information required to construct these, but we will in fact never present these formal proof terms.

As the function in which we are interested is a large expression we shall define it in a number of stages. Firstly we shall assume that we are allowed to split the definition up into parts and that we can use recursive definitions. Then we shall put the parts together and finally use the *rec* operator to remove the recursion from the definition. We call the function *mgiu*. It should not be confused with the type *MGIU*.

8.5.1 First step in the definition of our mgiu function

We define some auxiliary functions:

s1 •' s2 $=_{def}$ when((g)Inl(Pair(fst(s1) • fst(g), δ(snd(s1), snd(g))), (b)Inr(b), s2)

v --> t $=_{def}$ λx.(if x = v then t else x)

occs(x, tms) $=_{def}$ termsrec((v)(x = v), (f, ts, r)r, false, (t, ts, r, s)(r or s), tms)

We have seen that *termrec* and *termsrec* are definitionally equal. Hence we could have defined *occs* as:

occs(x, tm) $=_{def}$ termrec((v)(x = v), (f, ts, r)r, false, (t, ts, r, s)(r or s), tm)

We will use *occs(x, tms)* to define a type *OCCURS(x, tms)* in Chapter 10.

We are attempting to construct an inhabitant of a type of the form $A+\neg B$. All elements of a type of form $\neg B$ can be shown to be equal. (See, for example, [Kha86]). We call the value inhabiting this type *Abort*. For our purpose we may be interested in why there is no unifier of the terms we are interested in. We therfore make the following definitions:

occurs $=_{def}$ Abort

type_error $=_{def}$ Abort

functors $=_{def}$ Abort

arities $=_{def}$ Abort.

First we define *mgiu0*. *Empty* is the empty substitution.

```
mgiu0(t1, t2) =def (* preliminary defn. yet to use rec, and splitting the defn. *)
        term_or_terms_cases(A(t2), B(t2), C(t2), D(t2), t1)
where
A(t2, v) =def
        term_or_terms_cases(
                (w)(Inl(Pair(v --> w, α))),
                (f2, ts2)(if occs(v, ts2)
                                        then Inr(occurs)
                                        else Inl(Pair(v --> App(f2, ts2), β))),
                Inr(type_error)
                (hd2, tl2)Inr(type_error),
        t2)

B(t2, f, ts) =def
        term_or_terms_cases(
                (w)(if occs(w, ts)
                                then Inr(occurs)
                                else Inl(Pair(v --> App(f, ts), δ))),
                (f2, ts2)(if f2 = f then mgiu0(ts, ts2) else Inr(functors)),
                Inr(type_error),
                (hd2, tl2)Inr(type_error),
        t2)

C(t2) =def
        term_or_terms_cases(
                (w)Inr(type_error),
                (f2, ts2)Inr(type_error),
                Inl(Pair(Empty, χ)),
                (hd2, tl2)Inr(arities),
        t2)

D(t2, hd, tl) =def
        term_or_terms_cases(
                (w)Inr(type_error),
                (f2, ts2)Inr(type_error),
                Inr(arities),
                (hd2, tl2)(when(
                                (g)(g •' mgiu0(tl, tl2)),
                                (b)Inr(b),
                        mgiu0(hd, hd2))),
        t2)
```

8.5.2 Second step in the definition of our mgiu function

Now we put *A*, *B*, *C* and *D* in place:

```
mgiu1(t1, t2) =def (* preliminary defn. yet to use rec. *)
        term_or_terms_cases (
                (v)(term_or_terms_cases(
                        (w)(Inl(Pair(v --> w, α))),
                        (f2)(ts2)(if occs(v, ts2)
                                        then Inr(occurs)
                                        else Inl(Pair(v --> App(f2, ts2), β))),
                        Inr(type_error)
                        (hd2)(tl2)Inr(type_error),
                        t2)),
                (f)(ts)(term_or_terms_cases(
                        (w)(if occs(w, ts)
                                        then Inr(occurs)
                                        else Inl(Pair(v --> App(f, ts), δ))),
                        (f2)(ts2)(if f2 = f then mgiu1(ts, ts2) else Inr(functors)),
                        Inr(type_error),
                        (hd2)(tl2)Inr(type_error),
                        t2)),
                (term_or_terms_cases(
                        (w)Inr(type_error),
                        (f2)(ts2)Inr(type_error),
                        Inl(Pair(Empty, χ)),
                        (hd2)(tl2)Inr(arities) ,
                        t2)),
                (hd)(tl)(term_or_terms_cases(
                        (w)Inr(type_error),
                        (f2)(ts2)Inr(type_error),
                        Inr(arities),
                        (hd2)(tl2)(when(
                                        (g)(g •' mgiu1(tl, tl2)),
                                        (b)Inr(b),
                                mgiu1(hd, hd2))),
                        t2)),
        t1)
```

8.5.3 Final step in the definition of our mgiu function

Finally we use the *rec* operator:

```
mgiu(t1, t2) =def
rec((p)(m)term_or_terms_cases(
            (v)(term_or_terms_cases(
                  (w)(Inl(Pair(v --> w, α))),
                  (f2)(ts2)(if occs(v, ts2)
                              then Inr(occurs)
                              else Inl(Pair(v –> App(f2, ts2), β))),
                  Inr(type_error),
                  (hd2)(tl2)Inr(type_error),
                  snd(p))),
            (f)(ts)(term_or_terms_cases(
                  (w)(if occs(w, ts)
                              then Inr(occurs)
                              else Inl(Pair(v –> App(f, ts), δ))),
                  (f2)(ts2)(if f2 = f then m(ts, ts2) else Inr(functors)),
                  Inr(type_error),
                  (hd2)(tl2)Inr(type_error),
                  snd(p))),
            (term_or_terms_cases(
                  (w)Inr(type_error),
                  (f2)(ts2)Inr(type_error),
                  Inl(Pair(Empty, χ)),
                  (hd2)(tl2)Inr(arities) ,
                  snd(p))),
            (hd)(tl)(term_or_terms_cases(
                  (w)Inr(type_error),
                  (f2)(ts2)Inr(type_error),
                  Inr(arities),
                  (hd2)(tl2)(when(
                              (g)(g •' m(tl, tl2)),
                              (b)Inr(b),
                            m(hd, hd2))),
                  snd(p))),
      fst(p)),
      Pair(t1, t2))
```

8.6 Summary

In this chapter we have looked at a number of different unification algorithms. We have presented an expression which we shall show in Chapter 9 to be an inhabitant of the desired type. We should make it clear that the expression which we presented above was, in fact, derived partly as we performed the proof presented in the next chapter and partly from reflection on the other unification algorithms discussed above.

Chapter 9

A Proof By Well-founded Induction

In this chapter and Chapter 10 we use Nordström's rule for well-founded induction, presented in Chapter 6, to show that the function that we presented at the end of Chapter 8 is an inhabitant of the type that we presented in Chapter 7. We split this task into three parts: first we present some lemmas, then we prove that the algorithm terminates and in Chapter 10 we show that the algorithm terminates correctly.

9.1 Instantiating Nordström's rule

Recall that *mgiu(p)*, as defined at the end of Chapter 8, has the form *rec(e, p)*. If we take *C(p)* to be *HASMGIU(p) + NU(p)* and use the ordering < which we presented in Chapter 6 we can instantiate Nordström's rule for well-founded induction as shown in Figure 9.4. We have left implicit a certain amount of type information in this rule.

9.2 Proof of the total correctness of *mgiu*

To find a function which meets our specification we see that we should justify the following judgement:

$$\begin{bmatrix} a : \{TM,\ TMS\} \\ p : \mathcal{T}(a)\ PAIR \end{bmatrix}$$

$$mgiu(p) : HASMGIU(p) + NU(p)$$

We use the rule presented in Figure 9.4.

We observe that the judgement:

$$[a : \{TM,\ TMS\}]$$

$$\omega : Wellfnd(\mathcal{T}(a)\ PAIR,\ <)$$

was established in Chapter 6.

We use the technique outlined in [Nor88]. This means that we separate the proofs of termination and of correctness. We will prove termination in this chapter and correctness in Chapter 10.

9.3 Some lemmas

In this section we present, sometimes without proof, some lemmas which will prove useful in the proof.

9.3.1 Lemma 9.1

Let σ be a substitution. Then σ is idempotent iff there is no variable which occurs in both its range and its domain.

Proof:

Suppose σ is idempotent. Then for any term t, $\sigma * t = (\sigma \bullet \sigma) * t$. So σ has no effect on $\sigma * t$, i.e. no variable occurs in $\sigma * t$ and in the domain of σ. The variables which may occur in $\sigma * t$ are those which occur in t, but not in the domain of σ, and those which occur in the range of σ. Hence if σ is idempotent then no variable may occur in both its domain and its range.

Suppose no variable occurs in both the domain and the range of σ. Then, for any term t, no variable occurs in $\sigma * t$ which occurs in the domain of σ. Hence $\sigma = \sigma \bullet \sigma$. ■

9.3.2 Lemma 9.2

If σ is a most general idempotent unifier of $Pair(t_1, t_2)$ then every variable in its domain occurs in $Pair(t_1, t_2)$.

Proof:

If there were a variable in the domain of σ which did not occur in $Pair(t_1, t_2)$ then σ could not be both idempotent and most general. ■

9.3.3 Lemma 9.3

If σ is a most general idempotent unifier of $Pair(t_1, t_2)$ then every variable in its range occurs in $Pair(t_1, t_2)$.

Proof:

If there were a variable in the range of σ which did not occur in $Pair(t_1, t_2)$ then σ could not be both idempotent and most general. ■

9.3.4 Lemma 9.4

The composition of substitutions is associative:

$$\frac{\alpha : \text{SUBST} \qquad \beta : \text{SUBST} \qquad \chi : \text{SUBST}}{\delta : \text{EQ(SUBST, } \alpha \bullet (\beta \bullet \chi), (\alpha \bullet \beta) \bullet \chi)}$$

Proof:

The proof proceeds by induction, exploiting the fact that we are using a theory with an extensional equality. We suppress the informal proof here. ■

9.3.5 Lemma 9.5

α is a unifier of *Pair(Some(t, ts), Some(s, ss))* iff

1) α is a unifier of *Pair(t, s)* and

2) α is a unifier of *Pair(ts, ss)*.

Proof:

From the definition of the notion of a unifier α * *Some(t, ts)* is the same *TERMS* as α * *Some(s, ss)*. From the definition of * we see that *Some*(α * *t*, α * *s*) is the same *TERMS* as *Some*(α * *s*, α * *ss*). From the rules for equality of values of type *TERMS* we see that this means that α * *t* and α * *s* are the same *TERM* and that α * *ts* and α * *ss* are the same *TERMS*. From the definition of unifier we see that this means that α is a unifier of *Pair(t, s)* and also of *Pair(ts, ss)*.

If α is a unifier of *Pair(t, s)* and α is a unifier of *Pair(ts, ss)* then it is apparent for the rules for the equality of TERMS that it is also a unifier of *Pair(Some(t, ts), Some(s, ss))*. ■

9.3.6 Lemma 9.6

This lemma is crucial. It is stated without proof in [MW81] and [MW90].

If σ is a unifier of *p* then σ is a most general idempotent unifier of *p* iff for every unifier τ of *p* $\tau = \sigma \bullet \tau$.

9.3.7 Lemma 9.7

If σ is a most general idempotent unifier of *Pair(s, t)* then it is also a most general idempotent unifier of *Pair(t, s)*.

Proof:

Any unifier of *Pair(s, t)* is also a unifier of *Pair(t, s)* and *vice versa*. Hence the unifiers of *Pair(t, s)* are the same as those of *Pair(s, t)*. σ is a most general member of these. σ is idempotent by assumption. Hence σ is a most general idempotent unifier of *Pair(t, s)*. ■

Notice that if α is a formal proof that σ is a most general idempotent unifier of *Pair(s, t)* then it is not (in general) a proof that σ is a most general idempotent unifier of *Pair(t, s)*.

9.3.8 Lemma 9.8

If no unifier of *Pair(s, t)* exists then no unifier of *Pair(t, s)* exists.

Proof:

If it is absurd that a substitution which makes *s* and *t* the same exists, then by the symmetry of equality, it is absurd that a substitution that makes *t* and *s* the same exists. ■

9.4 Proof of termination

To prove termination we must show that the recursive calls are made on values which lie below the current one in the order which we presented before. From an inspection of the algorithm we see that we have the task of justifying the following judgements (we have left some of the witnesses uninstantiated):

Figure 9.1)

$$\begin{bmatrix} ts : TERMS \\ ss : TERMS \\ f : CONST \\ g : CONST \end{bmatrix}$$

$$\alpha : Pair(ts, ss) < Pair(App(f, ts), App(g, ss))$$

Figure 9.2)

$$\begin{bmatrix} t_1 : \text{TERM} \\ t_2 : \text{TERM} \\ ts_1 : \text{TERMS} \\ ts_2 : \text{TERMS} \end{bmatrix}$$

$$\beta : \text{Pair}(t_1, t_2) < \text{Pair}(\text{Some}(t_1, ts_1), \text{Some}(t_2, ts_2))$$

Figure 9.3)

$$\begin{bmatrix} t_1 : \text{TERM} \\ t_2 : \text{TERM} \\ ts_1 : \text{TERMS} \\ ts_2 : \text{TERMS} \\ \alpha : \text{HASMGIU}(s, \text{Pair}(t_1, t_2)) \end{bmatrix}$$

$$\chi(\alpha) : \text{Pair}(\text{fst}(\alpha) * ts_1, \text{fst}(\alpha) * ts_2) < \text{Pair}(\text{Some}(t_1, ts_1), \text{Some}(t_2, ts_2))$$

We will not construct the witnesses explicitly. As we have stated before their details are of little interest. We will, however, present informal proofs which would allow us, if we took sufficient care, to construct these witnesses. We will adopt the same policy when we come to prove correctness.

9.4.1 Justification of 9.1

We observe that *Pair(ts, ss)* contains precisely those occurrences of variables which *Pair(App(f, ts), App(g, ss))* does and that *ts* is smaller than *App(f, ts)*. Hence *Pair(ts, ss) < Pair(App(f, ts), App(g, ss))*. ■

9.4.2 Justification of 9.2

It is decidable whether any variables occur in $Pair(ts_1, ts_2)$ which do not occur in $Pair(t_1, t_2)$. If any do then there are more variables in $Pair(Some(t_1, ts_1), Some(t_2, ts_2))$ than in $Pair(t_1, t_2)$ and hence $Pair(t_1, t_2) < Pair(Some(t_1, ts_1), Some(t_2, ts_2))$. If there are no such variables then $Pair(t_1, t_2) < Pair(Some(t_1, ts_1), Some(t_2, ts_2))$ because t_1 is smaller than $Some(t_1, ts_1)$. Hence $Pair(t_1, t_2) < Pair(Some(t_1, ts_1), Some(t_2, ts_2))$. ■

9.4.3 Justification of 9.3

Informally, if σ is a most general idempotent unifier of $Pair(t_1, t_2)$ then $Pair(\sigma * ts_1, \sigma * ts_2) < Pair(Some(t_1, ts_1), Some(t_2, ts_2))$.

It is decidable whether $Pair(\sigma * ts_1, \sigma * ts_2)$ is the same pair of terms as $Pair(ts_1, ts_2)$.

If they are the same then either there is some variable which occurs in $Pair(t_1, t_2)$ but not in $Pair(ts_1, ts_2)$ or there is no such variable. In either case, ts_1 is smaller than $Some(t_1, ts_1)$. Hence, if $Pair(\sigma*ts_1,\sigma*ts_2)$ is the same pair of terms as $Pair(ts_1, ts_2)$ then $Pair(\sigma * ts_1, \sigma * ts_2) < Pair(Some(t_1, ts_1), Some(t_2, ts_2))$.

If $Pair(\sigma * ts_1, \sigma * ts_2)$ and $Pair(ts_1, ts_2)$ differ then all the occurrences of some variable x in one or both of ts_1 and ts_2 have been replaced by some term. Lemmas 9.2 and 9.3 tell us that x must also occur in $Pair(t_1, t_2)$. The only variables which may occur in the replacing term must also occur in $Pair(t_1, t_2)$ and, by lemma 9.1, differ from x. Hence x does not occur in $\sigma * ts_1$ or in $\sigma * ts_2$. Hence fewer variables occur in $Pair(\sigma*ts_1,\sigma*ts_2)$ than occur in $Pair(Some(t_1,ts_1),Some(t_2,ts_2))$.

Hence $Pair(\sigma*ts_1,\sigma ts_2)<Pair(Some(t_1, ts_1), Some(t_2, ts_2))$.

Hence, whether or not $Pair(\sigma*ts_1,\sigma*ts_2)$ is the same as $Pair(ts_1,ts_2)$,

$$Pair(\sigma * ts_1, \sigma * ts_2) < Pair(Some(t_1, ts_1), Some(t_2, ts_2))$$

and hence the judgement is justified. ■

9.5 Conclusion

We have now shown that evaluation of our *mgiu* algorithm terminates. In the next chapter we will show that it terminates correctly, that is that it terminates with a value of the type which is the specification.

Figure 9.4)

$$\left[b : \{TM,\ TMS\}\right] \qquad \left[\begin{array}{c} x : \mathcal{T}(c)\ PAIR \qquad \left[c : \{TM,\ TMS\}\right] \\ y(z) : HASMGIU(z) + NU(z) \left[\begin{array}{c} z : \mathcal{T}(d)\ PAIR \qquad \left[d : \{TM,\ TMS\}\right] \\ z < x \end{array}\right] \end{array}\right]$$

$$\frac{\omega : Wellfnd(\mathcal{T}(b)\ PAIR,\ <) \quad a : \{TM,\ TMS\} \quad p : \mathcal{T}(a)\ PAIR \qquad\qquad e(x,\ y) : HASMGIU(x) + NU(x)}{rec(e,\ p) : HASMGIU(p) + NU(p)}$$

Chapter 10

Proof of Correctness

In this chapter we complete the proof started in Chapter 9.

Our task is now to justify the final premiss of figure 9.4. We recall our definition of *mgiu(p)* in the form *rec(e,p)* from Chapter 8 and expand *e(x,y)* where it occurs in the premiss. We also expand the hypothetical assumptions. We then have a number of judgements to justify. These fall into a group of base cases and some induction steps. Justifying these judgements shows us how, in principle, to construct the proof witnesses that we left unspecified when we presented the *mgiu* algorithm in Chapter 8.

10.1 Base cases

We left the proof terms which witness the correctness of the algorithm in Chapter 8 unspecified. In this section we provide informal proofs which would allow us, if we took sufficient care, to construct these witnesses. In figures 10.1 to 10.6' α, β, χ, δ, *functors*, *occurs* and *arities* are as in the definition of *mgiu* in §8.5.3. We must show:

Figure 10.1)

$$\text{Pair(EMPTY, } \chi\text{) : HASMGIU(Pair(None, None))}$$

Figure 10.2)

$$\begin{bmatrix} x : \text{VAR} \\ y : \text{VAR} \end{bmatrix}$$

$$\text{Pair(x --> Var(y), } \alpha\text{) : HASMGIU(Pair(Var(x), Var(y)))}$$

Figure 10.3)

$$\begin{bmatrix} x : \text{VAR} \\ f : \text{CONST} \\ ts : \text{TERMS} \\ \kappa : \neg\text{OCCURS(x, ts)} \end{bmatrix}$$

$$\text{Pair(x --> App(f, ts), } \beta\text{) : HASMGIU(Pair(Var(x), App(f, ts)))}$$

Figure 10.3')

$$\begin{bmatrix} y : VAR \\ f : CONST \\ ts : TERMS \\ \kappa : \neg OCCURS(x, ts) \end{bmatrix}$$

$$Pair(y \dashrightarrow App(f, ts), \delta) : HASMGIU(Pair(App(f, ts), Var(y)))$$

Figure 10.4)

$$\begin{bmatrix} y : VAR \\ f : CONST \\ ts : TERMS \\ \kappa : OCCURS(x, ts) \end{bmatrix}$$

$$occurs : NU(Pair(Var(x), App(f, ts)))$$

Figure 10.4')

$$\begin{bmatrix} y : VAR \\ f : CONST \\ ts : TERMS \\ \kappa : OCCURS(x, ts) \end{bmatrix}$$

$$occurs : NU(Pair(App(f, ts), Var(x)))$$

Figure 10.5)

$$\begin{bmatrix} f : CONST \\ ts : TERMS \\ g : CONST \\ ss : TERMS \\ \kappa : \neg EQ(CONST, f, g) \end{bmatrix}$$

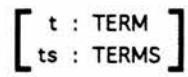
$$arities : NU(Pair(App(f, ts), App(g, ss)))$$

Figure 10.6)

$$\begin{bmatrix} t : TERM \\ ts : TERMS \end{bmatrix}$$

$$arities : NU(None, Some(t, ts))$$

Figure 10.6')



arities : NU(Some(t, ts), None)

Where, given the definition of *occs(x,tms)* in §8.5.1:

OCCURS(x, tms) $=_{def}$ EQ(BOOL, occs(x, tms), true)

10.1.1 Justification of 10.1

Any substitution σ is a unifier of *Pair(None,None)*. As every substitu - tion is an instance of the empty substitution the empty substitution is the most general unifier of *Pair(None,None)*. The empty substitution is idempotent. Hence it is the most general idempotent unifier of *Pair(None,None)*. ■

10.1.2 Justification of 10.2

The substitution *x --> Var(y)* is a unifier of *Pair(Var(x),Var(y))*. We show that for any unifier σ of *Pair(Var(x),Var(y))*,

$\sigma = (x \rightarrow \text{Var}(y)) \bullet \sigma$

and hence by lemma 9.6 that *x --> Var(y)* is a most general idempotent unifier of *Pair(Var(x),Var(y))*. Suppose σ is a unifier of *Pair(Var(x),Var(y))*. Consider, for an arbitrary variable *z*, *apply(((x--> Var(y)) • σ),z)*. From the definition of • this term is the same as $\sigma^*((x \text{ --> } (Var(y))\ z)$. If *z* differs from *x* then this is the same term as σ * *Var(z)*, which is the same term as *apply(σ,z)*. If *z* is the same variable as *x* then σ * *(apply((x --> Var(y)),z))* is the same term as σ^* *Var(y)*, which, from the definition of *, is the same terms as *apply(σ,y)*. But σ is a unifier of *Pair(Var(x),Var(y))*, so this is the same term as *apply(σ, x)*. We assumed that *x* = *z* so this is the same term as *apply(σ, x)*. So:

apply(σ, z) = apply((x --> Var(y)) • σ), z), for all variables z.

Hence by lemma 9.6, *x --> Var(y)* is a most general idempotent unifier of *Pair(Var(x),Var(y))*. ■

10.1.3 Justification of 10.3 and 10.3'

Lemmas 9.7 and 9.8 allow us to deal with 10.3 and 10.3' simultaneously. We choose to deal with 10.3. This proof follows much the same pattern as the proof above.

Because *x* does not occur in *ts*, the substitution *x --> App(f,ts)* is a unifier of *Pair(Var(x),App(f,ts))*. We show that for any unifier σ of *Pair(Var(x),App(f,ts))* σ = (*x --> App(f,ts)*) • σ, and hence by lemma 9.6 that *x --> App(f,ts)* is a most general idempotent unifier of *Pair(Var(x),App(f,ts))*.

Consider the term *apply((x --> App(f,ts)) • σ), z)*, for an arbitrary variable *z*. From the definition of • we see that this is the same as *σ*apply(x --> App(f,ts),z)*.

Suppose *z* differs from *x*. Then *apply(x --> App(f,ts),z)* is just *z* and so σ * *apply(x --> App(f,ts),z)* is σ * *z*, which is, by definition, *apply(σ,Var(z))*.

Suppose *z* = *x*. Then *apply(x --> App(f,ts),z)* is *App(f,ts)* and so σ* *apply(x --> App(f,ts),z)* is σ * *App(f,ts)*. If *z* = *x* then *apply(σ,z)* = *apply(σ,x)* which is, by definition, σ * *Var(x)*. By assumption, σ is a unifier of *Pair(Var(x),App(f,ts))* and so *σ*Var(x)= σ * App(f,ts)*.

Hence, whether *z* is the same as *x* or not,

apply(σ, z) = apply((x –> App(f, ts)) • σ, z).

As we have an extensional equality σ = (*x --> App(f,ts)*) • σ, for any unifier of *Pair(Var(x),App(f,ts))*, and hence, by lemma 9.6 the substitution *x --> App(f,ts)* is a most general idempotent unifier of *Pair(Var(x),App(f,ts))*. ■

Notice that the 'occurs check' was necessary to show the *correctness* of the algorithm. We normally think of the 'occurs check' in connection with the termination of the algorithm.

10.1.4 Justification of 10.4 and 10.4'

Lemmas 9.7 and 9.8 allow us to deal with 10.4 and 10.4' simultaneously. We deal with 10.4. As *x* occurs in *ts* it is absurd that there should be a substitution which makes *Var(x)* and *App(f,ts)* the same term. ■

10.1.5 Justification of 10.5

As *f* and *g* differ there is no substitution which can make *App(f,ts)* and *App(g,ss)* the same, irrespective of what *ts* and *ss* are. ■

10.1.6 Justification of 10.6 and 10.6'

There is no substitution which can make *Some(t,ts)* the same as *None*. ■

10.2 Induction

We must justify the judgements (recalling the definitions of TERM and TERMS, and taking *mgiu(p)* to be *rec(e,p)* and *C(p)* to be *HASMGIU(p) + NU(p)*, as above):

$$\left[\begin{array}{c} \text{tp : TERM PAIR} \\ \text{y(z) : C(z) } [\text{z} < \text{tp}] \end{array}\right]$$

$$\text{e(tp, y) : C(tp)}$$

and

$$\left[\begin{array}{c} \text{tsp : TERMS PAIR} \\ \text{y(z) : C(z) } [\text{z} < \text{tsp}] \end{array}\right]$$

$$\text{e(tsp, y) : C(tsp)}$$

We have a number of cases to consider, depending on the canonical form that *tp* or *tsp* may have. In each of the above judgements the second assumption tells us that we may assume that $y(z)$ meets the specification for *every* z below the current *tp* or *tsp*. We use this assumption for those z on which the function is recursively called. From an inspection of e and C we see that for the above judgements to be valid we require to justify the following, where the terms η and ι have been left uninstantiated in figures 10.7 and 10.8, respectively:

Figure 10.7)

$$\left[\begin{array}{c} f : CONST \\ g : CONST \\ \xi : EQ(CONST,\ f,\ g) \\ ts : TERMS \\ ss : TERMS \\ \psi : HASMGIU(Pair(ts,\ ss)) \end{array}\right]$$

$$Pair(fst(\psi),\ \eta(snd(\psi))) : HASMGIU(Pair(App(f,\ ts),\ App(g,\ ss)))$$

Figure 10.8)

$$\left[\begin{array}{c} t_1 : TERM \\ t_2 : TERM \\ ts_1 : TERMS \\ ts_2 : TERMS \\ \xi : HASMGIU(Pair(t_1,\ t_2)) \\ \psi : HASMGIU(Pair(fst(\xi)\ *\ ts_1,\ fst(\xi)\ *\ ts_2)) \end{array}\right]$$

$$Pair(fst(\xi)\ \bullet\ fst(\psi),\ \iota(snd(\xi),\ snd(\psi))) : HASMGIU(Pair(Some(t_1,\ ts_1),\ Some(t_2,\ ts_2)))$$

Figure 10.9)

$$\left[\begin{array}{c} ss : TERMS \\ ts : TERMS \\ f : CONST \\ g : CONST \\ \xi : NU(Pair(ts,\ ss)) \end{array}\right]$$

$$\xi : NU(Pair(App(f,\ ts),\ App(g,\ ss)))$$

Figure 10.10)

$$\left[\begin{array}{c} t_1 : \text{TERM} \\ t_2 : \text{TERM} \\ ts_1 : \text{TERMS} \\ ts_2 : \text{TERMS} \\ \xi : \text{NU}(\text{Pair}(t_1,\ t_2)) \end{array}\right]$$

$$\xi : \text{NU}(\text{Pair}(\text{Some}(t_1,\ ts_1),\ \text{Some}(t_2,\ ts_2)))$$

Figure 10.11)

$$\left[\begin{array}{c} t_1 : \text{TERM} \\ t_2 : \text{TERM} \\ ts_1 : \text{TERMS} \\ ts_2 : \text{TERMS} \\ \xi : \text{HASMGIU}(\text{Pair}(t_1,\ t_2)) \\ \psi : \text{NU}(\text{Pair}(\text{fst}(\xi) * ts_1,\ \text{fst}(\xi) * ts_2)) \end{array}\right]$$

$$\psi : \text{NU}(\text{Pair}(\text{Some}(t_1,\ ts_1),\ \text{Some}(t_2,\ ts_2)))$$

10.2.1 Justification of 10.7

Informally, we are asked to show that a most general idempotent unifier σ of *Pair(ts,ss)* is a most general idempotent unifier of *Pair(App(f,ts),App(g,ss))*, where *f* and *g* are equal.

Proof:

Suppose σ is a most general idempotent unifier of *Pair(ts,ss)* and that *f* and *g* are equal. We show that σ is a most general unifier of *Pair(App(f,ts),App(g,ss))*. Suppose τ is a unifier of *Pair(App(f,ts),App(g,ss))*. Then τ is also a unifier of *Pair(ts,ss)*. As σ is a most general unifier of *Pair(ts,ss)* σ is more general than τ. Hence σ is a most general unifier of *Pair(App(f,ts),App(g,ss))*. By assumption σ is idempotent. Hence σ is a most general idempotent unifier of *Pair(App(f,ts),App(g,ss))*. ■

10.2.2 Justification of 10.8

Informally this judgement tells us that if σ is a most general idempotent unifier of $Pair(t_1,t_2)$ and τ is a most general idempotent unifier of $Pair(\sigma * ts_1,\sigma * ts_2)$ then $\sigma \bullet \tau$ is a most general idempotent unifier of $Pair(Some(t_1,ts_1),Some(t_2,ts_2))$.

Proof:

We show that for an arbitrary unifier δ of $Pair(Some(t_1,ts_1),Some(t_2,ts_2))$, $\delta = (\sigma \bullet \tau) \bullet \delta$. Lemma 9.6 then shows that $\sigma \bullet \tau$ is a most general idempotent unifier of $Pair(Some(t_1,ts_1),Some(t_2,ts_2))$.

Lemma 9.5 tells us that δ is a unifier of $Pair(t_1,t_2)$. From the assumption that σ is a most general unifier of $Pair(t_1,t_2)$ and lemma 9.6 we conclude that $\delta = \sigma \bullet \delta$.

Lemma 9.5 tells us that δ is a unifier of $Pair(ts_1,ts_2)$. We have just shown that $\delta = \sigma \bullet \delta$. Hence $(\sigma \bullet \delta)$ is a unifier of $Pair(ts_1,ts_2)$. Hence, from the definition of $\bullet$, δ is a unifier of $Pair(\sigma * ts_1,\sigma * ts_2)$. From the assumption that τ is a most general idempotent unifier of $Pair(\sigma * ts_1,\sigma * ts_2)$ and lemma 9.6 we conclude $\delta = \tau \bullet \delta$. As $\delta=\sigma\bullet \delta$, $\delta = \sigma \bullet (\tau \bullet \delta)$. By the associativity of the composition of substitutions we see that $\delta = (\sigma \bullet \tau) \bullet \delta$. We now appeal to lemma 9.6 and conclude that $(\sigma \bullet \tau)$ is a most general idempotent unifier of $Pair(Some(t_1,ts_1),Some(t_2,ts_2))$. ■

Figures 10.7 and 10.8 constitute the 'positive' part of the proof, that is they tell us that the substitution that we construct is, indeed, a most general idempotent unifier of the terms concerned. Figures 10.9, 10.10 and 10.11 constitute the 'negative' part of the proof, that is they tell us that there is no unifier of the terms concerned. We have arranged the algorithm so that we are told part of the reason why there is no unifier: we chose to be told that either the functors of some application terms to be unified differed or that the arities of some application terms to be unified differed or that unification was not possible because some variable occurred in a term that it was to be unified with. We have suppressed a certain amount of information here: strictly we should say that it is absurd that two terms unify because if they were to unify then two differing constants would be the same or two terms of differing sizes should be the same. We have appealed to our experience of using unification algorithms here as to what information should be included and what suppressed.

10.2.3 Justification of 10.9

We now justify the judgement that if there is no unifier of $Pair(ts,ss)$ then there is no unifier of $Pair(App(f,ts),App(g,ss)$.

Proof:

Lemma 9.5 tells us that if we have a unifier of $Pair(App(f,ts),App(g,ss))$ then we have a unifier of $Pair(ts,ss)$. The premiss is that it is absurd to have have a unifier of $Pair(ts,ss)$.

Hence it is also absurd to have a unifier of *Pair(App(f,ts),App(g,ss))*. ■

10.2.4 Justification of 10.10

We must now justify the judgement that if there is no unifier of $Pair(t_1,t_2)$ then there is no unifier of $Pair(Some(t_1,ts_1), Some(t_2,ts_2))$.

Proof:

Lemma 9.5 tells us that if we have a unifier of $Pair(Some(t_1,ts_1),Some(t_2,ts_2))$ then we have a unifier of $Pair(t_1,t_2)$. The premiss is that it is absurd to have have a unifier of $Pair(t_1,t_2)$. Hence it is also absurd to have a unifier of $Pair(Some(t_1,ts_1),Some(t_2,ts_2))$. ■

10.2.5 Justification of 10.11

Finally we wish to justify the judgement that if σ is a most general unifier of $Pair(t_1,t_2)$, and there is no unifier of $Pair(\sigma * ts_1,\sigma * ts_2)$ then there is no unifier of $Pair(Some(t_1,ts_1),Some(t_2,ts_2))$.

Proof:

Suppose σ is a most general unifier of $Pair(t_1,t_2)$. Suppose we have a unifier τ of $Pair(Some(t_1,ts_1),Some(t_2,ts_2))$. We show that this is absurd. $\sigma \bullet \tau$ is also a unifier of $Pair(Some(t_1,ts_1),Some(t_2,ts_2))$, so τ is a unifier of $Pair(Some(\sigma*t_1,\sigma*ts_1),Some(\sigma*t_2,\sigma*ts_2))$. Lemma 9.5 then tells us that τ is a unifier of $Pair(\sigma*ts),\sigma*ts_2)$. But this contradicts the second premiss and hence it is absurd that τ is a unifier of $Pair(Some(t_1,ts_1),Some(t_2,ts_2))$. ■

Hence we have proved total correctness for *mgiu*. ■

10.3 Comments on the proof

When proving both termination and correctness we relied heavily on the idempotence of the substitutions involved. We knew that we should have to specify that the most general unifier should also be idempotent because Manna and Waldinger made this observation in [MW81]. Had this not been the case we should have certainly not mentioned this in the specification, and would have proceeded through the proof to this stage and realised (perhaps) that an extra condition was needed. There is an important lesson to be learned here about the relationship between the specification and the proof. It should also be noted that we

tried hard, without success, when attempting this proof to drop the idempotence condition.

We believe that it is worth discussing whether we derived an object which met a specification or showed that some object that we had prepared earlier, so to speak, met the specification. As the proof is presented in this thesis it would appear that the latter is the case. We should point out that the function presented in Chapter 8 was constructed after contemplation of this proof *and* of the unification algorithms that we discussed in Chapter 8. We should not like to claim that it was immediately apparent to us that the algorithm that we presented met its specification, and that the only task left was to check this. We should, however, point out that there is a point in any such proof where an object of a required type must simply be produced. In this case this step would be at the level of observing that the empty substitution is a most general idempotent unifier of *Pair(None,None)*, that the substitution *x --> Var(y)* is a most general idempotent unifier of *Pair(Var(x),Var(y))*, and so on.

Chapter 11

Comments

In this chapter we present some comments on the work presented in the thesis.

11.1 Representing terms

The representation of terms, as presented in Chapter 5, was more problematical than might at first have been expected. To represent terms as we wished we were forced to use mutually recursive types. Other approaches tried included using terms like those of [Pau85b] and using *TERM* lists rather than the type *TERMS*. We have pointed out elsewhere that we found Paulson's terms, although convenient, rather unsatisfactory. The use of *TERM* lists is, initially, attractive because lists are a well-known type. However the problems of saying what the computation rules for the associated non-canonical constant were, were just as tricky as those associated with stating the computation rules for the non-canonical constant that we used. Furthermore we felt that there was more technical interest in the use of mutually recursive types, and that any techniques that we used would have more chance of generalisability.

11.2 Mutual recursion

We suggest that Backhouse *et al* ([BCMS88]) are rather too dismissive of the interest of mutual recursion. Although the introduction rules are important it is with the computation and elimination rules that the hard work is to be done by the computer scientist.

Contemplation of the mutually recursive types illuminates the relationship between the introduction rules, the computation rules and the elimination rules for all types. We agree that there are very deep reasons (see, for example, [Dum91]) why the introduction rules are to be considered primary. We contend that from an understanding of the form of the introduction rules the computation rules for the non-canonical constant for the type (or group of types) can be inferred. We notice that we have to have some informal understanding of the notion of computation. Then, *and only then*, can the elimination rule for the type be constructed. The usual emphasis is that the elimination rules can be inferred from the introduction rules, and then the computation rules for the non-canonical constants inferred from the elimination rules. (See, for example, [BCMS88], [NPS90] and [Tho91]. [Mar84] makes little or no comment about the computation rules.) The usual

presentations of intuitionistic logic (e.g. [Gen36], [Pra65], [TD88]) emphasise that the elimination rules can be derived from the introduction rules by appealing to some more-or-less formalised principle of 'harmony'. [Dum91] discusses this issue further. Very informally we may say that we must not get any more or less out of a proposition from elimination rule than we put in with the introduction rules. The use of proof objects, and the computation rules for the non-canonical constants allows us to capture exactly the notion of harmony. If we had different informal notions of computation then we should expect to have different elimination rules for the same group of introduction rules.

11.3 Well-founded recursion

Another feature of the proof that we presented was the use of well-founded recursion. This technique proved to be very powerful, and although the justification of the induction was without doubt the hardest part of the thesis it would have been ever harder to write out a structurally recursive function to unify two terms.

11.4 Automation

One way that might be suggested to improve the proof would be to use some automated proof-assistant, such as GTTS [Pet82], Nuprl [Con86], ELF [HHP87], Oyster [Hor88], Isabelle [Pau90] or PICT [Ham90]. The reasons that this was not done were mainly practical. Firstly, none of these systems was available on suitable hardware. A port to such hardware may have been possible. However there was no certainty that any port would be successful and there was a certainty that any attempt at such a port would be time consuming. It was also not clear that any of the systems that were available would be able conveniently to represent mutually recursive types and well-founded induction. The task of adapting an existing proof-assistant to cope specifically with the rules of the 1984 theory with mutually recursive types and well-founded induction was thought to be outwith the scope of this thesis.

It is certainly the case that the use of an automated proof assistant would have eased the tedium of some of the finer parts of the proof and a fully formal proof could have been presented.

It is also worth noting that most of the automated proof-assistants available depend for their correctness on the correctness of some unification algorithm.

11.5 On specifications

We return to the problem of the idempotence of the most general unifier that we construct. As we saw idempotence was crucial, yet we contend that it was impossible for this to be foreseen when we first wrote the specification. For instance, we have never seen a Prolog textbook which mentioned that Prolog's unification algorithm constructs

idempotent unifiers. Our original specification was too weak and, for the induction proof to work, we needed to prove a stronger proposition. Of course, our proof of the stronger proposition contains a proof of the weaker proposition. We suggest that this sort of problem will arise more often as larger problems are tried using M-LTT. We note that the problem of idempotence would not have been noticed (had we not had [MW81] available) until the proof was very nearly complete. This problem would, of course, have arisen whatever language had been used to write the specification.

11.6 Disjunctive specifications

The specification that we used contained a disjunction in which one of the disjuncts was essentially negative in that it asserted that there was no unifier of the terms that we were interested in. When evaluating the function we usually have little or no interest in *why* there is no unifier of the pair of terms. We used a rather *ad hoc* technique to suppress this information.

The specification was almost an instance of the 'law of the excluded middle'. There is, of course, a classical proof of this proposition which has no algorithmic content. As we have pointed out we have little interest in the algorithmic content of the proof of the negative part of the specification. We could see no way to exploit this observation as a means to structure the proof or suppress the uninteresting parts.

Chapter 12

Conclusion

We have used Martin-Löf's type theory to specify and express a unification algorithm. We have utilised an extension of the 1984 extensional theory with mutually recursive types, justified both directly and *via* an encoding using the type of general trees. We used well-founded induction over a group of mutually recursive types to prove the correctness of the algorithm that we presented.

References

[Apt90] K. R. Apt, Logic Programming, *in*: J. van Leeuwen (ed), *Handbook of Theoretical Computer Science vol. B Formal Methods and Semantics* (Elsevier Science Publishers and MIT Press, 1990).

[Bac86] R. Backhouse, On the Meaning and Construction of Rules in Martin-Löf's Theory of Types, Technical Report CS 8606, Wiskunde en Informatica, Rijksuniversiteit Groningen (1986).

[BCMS88] R. Backhouse, P. Chisholm, G. Malcolm and E. Saaman, Do-it-yourself Type Theory, Technical Report CS 8811, Wiskunde en Informatica, Rijksuniversiteit Groningen (1988), also *Formal Aspects of Computer Science* **1** (1989), pp 19-84.

[Bar84] H. P. Barendregt, *The Lambda Calculus: Its Syntax and Semantics*, Studies in Logic and the Foundations of Mathematics 103 (North-Holland, 1984).

[BB85] E. Bishop and D. Bridges, *Constructive Analysis*, Grundlehren der mathematischen Wissenschaften 279 (Springer-Verlag, 1985).

[Bro81] L. E. J. Brouwer, *Brouwer's Cambridge Lectures on Intuitionism* (Cambridge University Press, 1981).

[CL73] C.-L. Chang and R. C.-T. Lee, *Symbolic Logic and Mechanical Theorem Proving* (Academic Press, 1973).

[Chi88] P. Chisholm, *Investigations into Martin-Löf Type Theory as a Programming Logic*, Ph.D. thesis, Heriot-Watt University (1988).

[Con86] R. L. Constable *et al. Implementing Mathematics with the NuPRL Proof Development System.* (Prentice-Hall, 1986).

[DP60] M. Davis and H. Putnam, A Computing Procedure for Quantification Theory, *Journal of the Association for Computing Machinery* **7** (1960), pp 201-216. Also in: H. Siekmann and G. Wrightson (eds) *Automation of Reasoning. Classical Papers on Computational Logic (1957-1966)* Two Vols. (Springer, 1983).

[Dum75] M. Dummett, The Philosophical Basis of Intuitionistic Logic, *in*: [RS75], pp. 5-40.

[Dum77] M. Dummett, *Elements of Intuitionism*, Oxford Logic Guides (Clarendon Press, 1977).

[Dum90] M. Dummett, The Source of the Concept of Truth, *in*: G. Boolos (ed), *Meaning and Method* (Cambridge University Press, 1990) pp. 1-15.

[Dum91] M. Dummett, *The Logical Basis of Metaphysics*, (Duckworth, 1991).

[Dyb89] P. Dybjer, An Inversion Principle for Martin-Löf's Type Theory, *in*: P. Dybjer, L. Hallnas, B. Nördstrom, K. Petersson and J. M. Smith (eds), Proceedings of Workshop on Programming Logic, Båstad 1989 Report 54, Programming Methodology Group, Chalmers University of Technology/Göteborg University, Göteborg (1989), pp. 177-190.

[Eri84] L.-H. Eriksson, Synthesis of a Unification Algorithm in a Logic Programming Calculus, *Journal of Logic Programming* **1**(1984), pp. 3-18.

[GM89] G. Gazdar and C. S. Mellish, *Natural Language Processing in PROLOG. An Introduction to Computational Linguistics* (Addison-Wesley, 1989).

[Gen36] G. Gentzen, Investigations into Logical Deduction, *in*: Szabo (ed), *The Collected Papers of Gerhard Gentzen* (North-Holland, 1969), pp. 68-131.

[Ham90] A. G. Hamilton, The Pict System, Department of Computing Science Technical Report 57, Stirling University (1990).

[HHP87] R. Harper, F. Honsell and G. Plotkin, A Framework for Defining Logics, *in: Proceedings of the Symposium on Logic in Computer Science* (IEEE, 1987), pp 194-204.

[Her71] J. Herbrand, Investigations in Proof Theory, *in*: W. D. Goldfarb (ed), *Logical Writings: Jacques Herbrand* (D. Reidel, 1971).

[Hey71] A. Heyting, *An Introduction to Intuitionism*, 3rd edition (North-Holland, 1971).

[Hor88] C. Horn, The Nuprl Proof Development System, Working Paper 214, Department of Artificial Intelligence, University of Edinburgh (1988). The Edinburgh version of Nuprl has been renamed Oyster.

[How80] W. A. Howard, The Formulae-as-types Notion of Construction, *in*: J. P. Seldin and J. R. Hindley (eds), *To H. B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism* (Academic Press, 1980).

[Kha86] M. A.-A. Khamiss, *Program Construction in Martin-Löf's Theory of Types*, Ph.D. thesis, Essex University (1986).

[MW81] Z. Manna and R. Waldinger, Deductive Synthesis of the Unification Algorithm, *Science of Computer Programming* **1** (1981), pp. 5-48.

[MW85] Z. Manna and R. Waldinger, *The Logical Basis For Computer Programming, Vol. 1 Deductive Reasoning.* (Addison-Wesley, 1985).

[MW90] Z. Manna and R. Waldinger, *The Logical Basis For Computer Programming, Vol. 2 Deductive Systems.* (Addison-Wesley, 1990).

[Mar75] P. Martin-Löf, An Intuitionistic Theory of Types: Predicative Part, *in*: [RS75], pp. 73-118.

[Mar82] P. Martin-Löf, Constructive Mathematics and Computer Programming, *in*: L. J. Cohen, *Logic, Methodology and Philosophy of Science VI* Studies in Logic and the Foundations of Mathematics 104 (North-Holland, 1979).

[Mar83] P. Martin-Löf, On the Meanings of the Logical Constants and the Justifications of the Logical Laws, Notes taken by Giovanni Sambin and Aldo Ursini of a short course given at the meeting Teoria dell Dimonstrazione e Filosofia della Logica, Siena, (1983).

[Mar84] P. Martin-Löf, *Intuitionistic Type Theory*, Studies in Proof Theory: Lecture Notes (Bibliopolis, 1984).

[Mar87] P. Martin-Löf, Truth of a Proposition, Evidence of a Judgement, Validity of a Proof, *Synthese* **73** (1987), pp. 407-420.

[Mil78] R. Milner, A Theory of Type Polymorphism in Programming, *Journal of Computer and Systems Sciences* **17** (1978), pp. 348-375.

[Nar89] D. Nardi, Formal Synthesis of a Unification Algorithm by the Deductive Tableaux Method, *Journal of Logic Programming* **7** (1989), pp. 1-43.

[Nor88] B. Nordström, Terminating General Recursion, *BIT* **28** (1988), pp. 605-619.

[NPS90] B. Nordström, K. Petersson and J. M. Smith, *Programming in Martin-Löf's Type Theory*, International Series of Monographs on Computer Science 7 (Oxford University Press, 1990).

[NS84] B. Nordström and J. M. Smith, Propositions and Specifications of Programs in Martin-Löf's Type Theory, Programming Methodology Group Report 13, Chalmers University of Technology/Göteborg University, Göteborg (1986), also *BIT* **24** (1984), pp. 288-301.

[Nor91] P. Norvig, Correcting a Widespread Error in Unification Algorithms, *Software Practice and Experience* **21** (2) (1991), pp. 231-233.

[Pau84] L. C. Paulson, Constructing Recursion Operators in Intuitionistic Type Theory, Technical Report 57, Universtiy of Cambridge Computer Laboratory, (1984), also *Journal of Symbolic Computation* **2** (1986).

[Pau85a] L. C. Paulson, Lessons learned from LCF: A Survey of Natural Deduction Proofs, *Computer Journal* **28** No. 5 (1985), pp. 474-479.

[Pau85b] L. C. Paulson, Verifying the Unification Algorithm in LCF, *Science of Computer Programming* **5** (1985), pp. 143-169.

[Pau87] L. C. Paulson, *Logic and Computation: Interactive Proof with Cambridge LCF*, Cambridge Tracts in Theoretical Computer Science 2 (Cambridge University Press, 1987).

[Pau90] L. C. Paulson, Isabelle: The Next 700 Theorem Provers, *in:* P. Odifreddi (ed), *Logic and Computer Science.* (Academic Press, 1990).

[Pau91] L. C. Paulson, *ML for the Working Programmer* (Cambridge University Press, 1991).

[Pet82] K. Petersson, A Programming System for Type Theory, Programming Methodology Group Report 9, Chalmers University of Technology, Göteborg (1982, 1984).

[Pet86] K. Petersson, Terms as Proofs, Programming Methodology Group Memo 50 Chalmers University of Technology/Göteborg University, Göteborg (1986).

[Pey87] S. L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice Hall International Series in Computer Science (Prentice Hall, 1987).

[Pra60] D. Prawitz, An Improved Proof Procedure, *Theoria* **26** (1960), pp. 102-139.

[Pra65] D. Prawitz, *Natural Deduction: A Proof Theoretical Study*, Stockholm Studies in Philosophy 3 (Almqvist and Wiksel, 1965).

[Pra77] D. Prawitz, Meaning and Proofs: On the Conflict Between Classical and Intuitionistic Logic, *Theoria* **43** (1977) pp. 2-40.

[Rob65] J. A. Robinson, A Machine-Oriented Logic Based on the Resolution Principle, *Journal of the Association for Computing Machinery* **12** (1965) pp. 23-41.

[Rob92] J. A. Robinson, Logic and Logic Programming, *Communications of the Association for Computing Machinery* **35** (1992), pp. 40-65.

[RS75] H. Rose and J. Shepherdson (eds), *Logic Colloquim 73*, Studies in Logic and Foundations of Mathematics 80 (North Holland, 1975).

[SM87] E. Saaman and G. Malcolm, Well-founded Recursion in Type Theory, Technical Report, Wiskunde en Informatica, Rijksuniversiteit Groningen (1987).

[Sc-H84] P. Schroeder-Heister, Generalised Rules for Quantifiers and the Completeness of the Intuitionistic Operators &, ∨, →, Λ, ∀, ∃, *in*: M. M. Richter, E. Borger, W. Oberschelp, B. Schinzel and W. Thomas (eds), *Computation and Proof Theory, Proceedings of the Aachen Colloquium*, Lecture Notes in Mathematics 1104 (Springer-Verlag, 1984).

[Shi86] S. M. Shieber, *An Introduction to Unification-based Approaches to Grammar*, Center for the Study of Language and Information Lecture Notes 4 (CSLI, 1986).

[Sie89] J. H. Siekmann, Unification Theory, *Journal of Symbolic Computation* **7** (1989), pp. 207-274.

[Smi83] J. M. Smith, The Identification of Propositions and Types in Martin-Löf's Type Theory: A Programming Example, Programming Methodology Group Report 14 Chalmers University of Technology/Göteborg University, Göteborg (1983).

[SS86] L. Sterling and E. Shapiro, *The Art of Prolog* (MIT Press, 1986).

[Sun86a] G. Sundholm, Systems of Deduction, *in*: D. Gabbay and F. Guenthner (eds), *Handbook of Philosophical Logic Vol. 1, Elements of Classical Logic*, Synthese Library 164 (D. Reidel, 1986).

[Sun86b] G. Sundholm, Proof Theory and Meaning, *in*: D. Gabbay and F. Guenthner (eds), *Handbook of Philosophical Logic Vol. 3, Alternatives to Classical Logic*, Synthese Library 167 (D. Reidel, 1986).

[Tho91] S. Thompson, *Type Theory and Functional Programming* (Addison-Wesley, 1991).

[Tro69] A. S. Troelstra, *Principles of Intuitionism*, Lecture Notes in Mathematics 95 (Springer-Verlag, 1969).

[TD88] A. S. Troelstra and D. van Dalen, *Constructivism in Mathematics, An Introduction Vol. 1*, Studies in Logic and the Foundations of Mathematics 121 (North-Holland, 1988).

[Wik87] A. Wikström, *Functional Programming Using Standard ML*, Prentice Hall International Series in Computer Science (Prentice Hall, 1987).