List of Figures

Chapter 1

Introduction.

Logic is the science of reasoning; it seeks to catalogue and discover "ways of thinking that always work". To date, the most notable success of logic is in the realm of mathematics. Indeed many would say that logic is a part of mathematics. This thesis investigates ways of implementing mathematical logic on a computer.

To start, we give a loose characterization of logic. Suppose someone believes a mathematical statement. She may believe it for any number of reasons, but if she is to convince someone else of the verity of that statement, she must supply a chain of reasoning, a *proof*, that another person can follow to also attain a state of belief in the statement. A proof has lasting value, while belief may come and go. Mathematicians regard the notion of proof as being of fundamental importance and have expended a great deal of thought to pin it down.

If the mathematical statement is groundbreaking or contentious, its proof will need to stand up under close scrutiny. Therefore, the statement and proof must be expressible in a simple, unambiguous language so that no confusion is possible. Each step in the chain of reasoning must be so simple that it is impossible to disagree with, or decomposable into a sequence of such simple steps. It is essential that the most complicated of arguments be expressible by such a chain of reasoning.

Thus a system of logic has a "formal" language, a notion of proof as a chain of simple reasoning steps in that language, a requirement on proof of expressiveness, and a requirement that no step can be disagreed with. (To a logician, these last two requirements amount to completeness and soundness.) There may also be fundamental assumptions that have been agreed upon beforehand.

In principle then, a proof of a mathematical statement can be translated into a system of logic (such a proof is called *formal*), so logic provides an adequate framework for mathematical argumentation. In practice, however, formal proofs are unmanageably detailed. This is one reason why most proofs are "semi-formal": a typical proof found in a mathematics journal uses a combination of notation, reference to previous results, and natural language phrases, for example "it follows that", "hence", "by induction", and "trivial". A consequence of this is that a surprising number of journal proofs have been found to be incorrect, for the reasoning is carried out in an ill-defined medium which permits subtle inconsistencies to be overlooked.

The advent of the digital computer made it possible to write a program that implements a logic. The computer offers remedies for the problems of size and detail of formal proof: the speed of the computer allows the checking of proofs that would be overwhelmingly large for a human, while the accuracy of the computer allows him to shunt the burden of remembering the state of the proof to the machine. For a mathematician, therefore, a computer implementation of logic could prove to be a valuable tool, ensuring the correctness of his argument while allowing him to do proofs in his usual style.

A more ambitious goal is to use computers to search for proofs, rather than to merely check proofs. There are theoretical reasons why this not possible in general, but some techniques have been discovered that work well in important areas.

1.1 Outline of thesis

This thesis explores some issues surrounding the mechanization of a logic, in this case a variant of Church's Simple Type Theory, called *Higher Order Logic*, or *HOL*. The implementation is called **hol90**.

In the rest of Chapter One, we give a literature review of logic implementations. The review is preceded by a very brief history of modern logic.

In Chapter Two, we intertwine a definition of the logic with a description of our implementation of it. This includes expositions of:

- Types and terms;
- The embedding of the *HOL* syntax in the programming metalanguage Standard ML;
- The type inference algorithm for *HOL* terms; and
- The encapsulation of the various means of obtaining theorems in the logic.

In Chapter Three, we investigate the underlying support, or prelogic, for the theorem proving machinery of *HOL*. We identify the three fundamental operations in the prelogic for *HOL* (α-conversion, or identity of terms modulo renaming of bound variables; term substitution; and type substitution), give formal specifications for them, and go through their implementations in detail.

Chapters Two and Three form the first in-depth presentation of the implementation of a higher-order logic theorem prover. **hol90** is intended to replace the current system (**hol88**), the creation of Mike Gordon at Cambridge University [Gor85, Gor88]. **hol88** is widely used as a tool for Hardware Verification and mathematics. It has performed some of the largest mechanical proofs completed to date [Coh88, Coh89, GB89]. Theorem provers of this calibre are not simple—the implementation of **hol90** takes up approximately 18,000 lines of Standard ML and requires a knowledge of topics ranging through compiler technology (parsing, type inference, and macro expansion), the lambda calculus, and logic, principally what might be called applied proof theory, *e.g.*, the efficient mechanization of primitive logical operations such as substitution.

In Chapter Four, we give a detailed proof of correctness for the algorithm for α-conversion. This is a first step in verifying the implementation of hol90.

In Chapters Five and Six, we take some first steps in integrating the fields of Term Rewriting Systems and *LCF*-style proof, in the hope that each have something to offer the other. We implement the fundamental algorithm in automated equational deduction, Knuth-Bendix completion, as a derived rule of inference in *HOL*. In Chapter Six, we describe, by example, how the (difficult) associative-commutative unification algorithm works. We use our implementation of the algorithm to improve the power of the current term replacement rule of inference by incorporating a notion of associative-commutative equivalence. A cornerstone of these chapters is the identification of the first order terms of a given type.

HOL is widely used in hardware verification proofs, but to be more useful in hardware design it must be integrated into the design environment. In Chapter Seven, we present a method for translating *HOL* descriptions of hardware into a commonly used netlist description language. This provides a much-needed link between the hardware verifier and the hardware designer.

In Chapter Eight, we present conclusions and topics for further research.

1.2 History

The computer logic systems of today are founded upon the pioneering work of Gottlob Frege in the late nineteenth century. Frege marks the beginning of modern logic. He introduced the seminal notion of *formal system* and defined theorem and proof in terms of formal systems. He also proposed a particular formal system as a foundation for mathematics [Fre67].

Bertrand Russell found that Frege's system allowed the expression of a set-theoretic paradox and was therefore inconsistent. His analysis of why Frege's system

allowed the paradox resulted in the theory of types [Rus67], in which he and Whitehead formalized a portion of mathematics.

Three important properties of a logical system are soundness, completeness, and decidability. Soundness and completeness were intuitively sketched above. A decidable property is one that has a ***decision procedure***, which is a program that will take an arbitrary statement expressible in the language of the logic and correctly classify it as having the specified property, or not. In his thesis (1930), Godel proved the completeness of first order logic [God67a]. Shortly thereafter (1931), he proved the incompleteness of any system that can express arithmetic [God67b]. In 1936, Church proved the undecidability of first order logic [Chu36]. For computer implementations, the consequences of these results are:

- The soundness and completeness of first order logic imply that there is a semi-decision procedure for validity. That is, *if* a formula is a theorem, then a program can find a proof for it in finite time. The first procedures for doing this were for theoretical purposes and unsuitable for computer implementation.
- The undecidability of first order logic implies that one cannot write a program that will take an arbitrary formula of first order logic and correctly classify it as a theorem or non-theorem in finite time.
- The incompleteness of arithmetic implies that there exist properties that hold of mathematical objects that we cannot show to hold by use of a computer system.

A *refutation* proof of a statement shows that the negation of the statement leads to a contradiction. In some logics, this allows one to infer that the original statement is a theorem. Herbrand's dissertation (1930) provides the foundations of refutation theorem proving [And86, p.132-138]. Gentzen (1935) introduced the sequent calculus and the *Cut* rule of inference [Gen69]. Various formulations of the sequent calculus

underlie many of today's so-called proof-checkers, while the Cut rule and Herbrand's theorem are essential in most so-called automatic theorem provers.

Church joined his lambda calculus with Russell's theory of types to get Simple Type Theory (1940), an elegant foundation for Mathematics [Chu40]. This thesis describes aspects of an implementation of a version of Simple Type Theory.

This brief summary shows that the theoretical ideas underlying most of today's computer logic implementations were published before the end of the Second World War. As we shall see, there have been recent advances in logic that have spurred new implementations.

1.3 Literature survey

J.A. Robinson [Rob65] introduced the resolution method, which uses substitution and the Cut rule to do refutational proof. Unification is used to find the best substitutions to make, thus constraining the search space. If a formula of first order logic is a theorem, then the resolution method will result in a contradiction. A good introductory text is [CL73]. In the name of efficiency, there have been a huge number of modifications proposed to the resolution method; see [Wos88] for some of the more important ones.

The work of Milner and others at Stanford then Edinburgh in the early 1970's reveals a second approach, the so-called "proof checking" of LCF [GMW79]. The important features of this are:

- The logic is formalized in a programming metalanguage.
- Theorems are only derivable by using the axioms and primitive rules of the logic. This is enforced by encapsulating the axioms and primitive rules in an abstract data type.

- The user constructs proof procedures in the metalanguage. This activity is eased by tactics and tacticals, which provide a convenient language for performing goal-directed proofs. Since the metalanguage is an interactive programming language, proofs may be developed interactively.

The abstract data type ensures soundness, while the programming metalanguage allows the implementation of any algorithm, *e.g.*, proof procedure or decision procedure, one wishes. The implementation described in this thesis belongs to the LCF family of theorem provers.

The Boyer-Moore theorem prover [BM79] is probably the most successful theorem prover to date: it has (semi-)automatically proved many difficult theorems, including Godel's incompleteness theorem, and a Church-Rosser theorem for the lambda calculus. The expressive poverty of the language (unquantified first order statements), is overcome by the heuristic use of induction over recursively defined structures. The m-EVES theorem prover[PS89] uses many of the heuristics developed by Boyer and Moore and goes beyond, in the expressiveness of the logic and in the variety of proof procedures used.

A fourth approach uses the notion of decision procedure as fundamental. The work of Nelson and Oppen in the late 1970's allows the cooperation of decision procedures in the search for a proof [NO79, NO80]. This method is constrained by the computational complexity of the slowest decision procedure used. Shostak [Sho84, Sho79] extended the work of Nelson and Oppen and also presented a useful simplification of Presburger arithmetic.

The seminal paper of Knuth and Bendix [KB70] initiated the study of automatic deduction in equational logic. Central in this is the Knuth-Bendix completion procedure and rewrite rules. The early survey paper by Huet and Oppen [HO80] establishes the (often heavy) terminology in this field and is still very useful as an introduction, although [KM88] is easier for novices. Methods

for doing "induction" proofs were initiated by Musser [Mus80] and extended by Huet and Hullot and Bachmair[HH82, Bac88]. Automatic equational deduction depends a great deal on well-founded term orderings [Der87] and advanced unification algorithms[Sti81, For85, PS81].

A refutation system for second order logic was presented by Pietrzykowski[Pie73]. Refutation systems for full higher order logic [Hue73, And81] had to wait until Huet discovered a complete unification algorithm for typed lambda calculus terms [Hue75].

Since the original ideas have been confirmed by implementation, attention has shifted to expanding their power. There is now interest in seeing if combinations of these ideas will interact beneficially. One promising approach is that of Hsiang who combines term rewriting with refutation proof [HD83]. He discovered a confluent set of rewrite rules for propositional logic, allowing, via Skolemization, completion-based refutation proofs to be conducted for first order logic. Analogous work has been carried out by Kapur and Narendran in the RRL system[KN84], using the Grobner basis computation [Buc85].

A different combination of ideas shows the value of treating proofs as data. The work of Felty [Fel86], based on that of Miller[Mil83, Mil87], uses expansion trees (a compact representation of proofs) as a link between refutation proofs and the tactical natural deduction proofs of LCF. This allows us to contemplate the notion of proof portability between systems, along with sound cooperation among proof systems.

The "proofs as data" approach is in marked contrast to most implementations wherein a proof does not even exist! For example, consider a typical resolution theorem prover. A "proof" in such a system is really a dynamically changing data structure in computer memory. If a contradiction is found, that fact is reported, the proof procedure terminates, and the data structure is wiped out. The only permanent object is the code that implements the proof procedure. The idea of proof in such a system is not well defined: the proof is not retained and cannot be

studied or transformed. (Choosing not to keep proofs is understandable in view of the large size of formal proofs.)

The Constructivist approach to mathematics, as originally espoused by Brouwer has lately given impetus to research in intuitionistic higher order logics that are claimed to be more suited to Computer Science issues than classical first order logic [ML85, CAB+86, CH88]. These logics allow the treatment of formulas of the logic as types and proofs as programs [GTL89]. A proof of a formula can be converted into a program that, when run, obeys the formula, which can be therefore regarded as a specification for the program. This "proofs as programs" approach has the benefit that it provides a single system for dealing with programs, specifications, and proofs; a program is formally derived from a specification, thus opening up a new slant on specification-based programming.

Implementing a theorem prover is a difficult task; in the past few years some "generic" theorem provers have been suggested [HHP87, AHM89]: these provide a framework for quickly implementing logics from their specifications. Larry Paulson's Isabelle [Pau90] seems to be the first implementation of a logical framework. The *LEGO* system [LPT89] under development at Edinburgh is another noteworthy implementation.

The field of theorem proving in modal and intuitionistic logic is explored in [Wal90].

Chapter 2

hol90

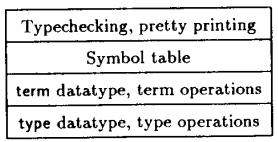
The original implementation of *HOL* is due to Mike Gordon at Cambridge University [Gor85, Gor88]. This implementation (**hol88**) has been used for most of the largest hardware verifications to date [Coh88, GB89]. It is one of the few sufficiently well developed theorem provers able to handle such challenging verifications. We have re-implemented **hol88** in the Standard ML programming language. The result is called **hol90**.

This chapter can be viewed as a followup to the paper *Representing a Logic in the LCF Metalanguage*, by Mike Gordon [Gor82]. We explore the representations of types, terms, and theorems, how to embed the syntax of the logic in the programming metalanguage, and how to provide security in the production of theorems. We apply to *HOL* the methodology espoused by Gordon in his paper. The implementation of **hol90** does, in the main, follow that of **hol88**. In the course of it, we have discovered several bugs, remedied some infelicities, and obtained a complete overview of the implementation, something that very few people have. To our knowledge, this chapter and the next provide the first in-depth description of an LCF-style logic implementation. The book by Paulson[Pau87] and the *HOL* Manual [Gor89] also cover LCF-style logic implementations, but as user guides. The work presented here takes us on a tour of the underworld of the implementation.

Why re-implement **hol88**? The simple answer is that **hol88** is built on software that is no longer very well understood, except by a very few individuals. Hence it is not easily maintained or extended. More detailed reasons are:

- **hol88** was written in a mixture of Lisp and ML. There are always pragmatic problems when dealing with two languages, plus we wanted to code the entire implementation in a strongly typed language, for security.
- **hol88** is *LCF* with its logic (*PPLAMBDA*) replaced by the *HOL* logic. This led to some "bending" of *HOL* objects to meet the *PPLAMBDA* requirements.
- The primitive routines for term manipulation in **hol88** are intricately coded and deserve to be held to the light.
- Theory files have an obscure, Lisp-dependent syntax.
- It is by no means clear how the system is built. *HOL* suffices to formalize a great deal of mathematics, so it is interesting to see exactly how the development proceeds. For example, the definitional mechanism is used to introduce abbreviations into the logic, but the mechanism itself depends on other abbreviations: how are they introduced?
- With the definition of Standard ML (SML), a successor to ML, and the availability of high quality native code compilers for SML, we anticipated that the implementation would become faster merely by switching to such a compiler.

The implementation of **hol90** can be viewed as being layered, with higher layers depending on lower layers. The lowest layers define the polymorphically typed lambda calculus, the symbol table, type inference, and some support routines, notably the functions for α–conversion, substitution, and type instantiation.



The next layer provides an embedding of the syntax of the logic in the programming language via a form of macro expansion. This is the interface most users see. Recall that the system is interactive: the user formulates the proposition to be proved and then interactively finds the proof.

Parsing of HOL (quotation, antiquotation)

The next layer defines theorems and how to get them. There are several means of getting a theorem in the logic: ensuring that they are all used properly is the concern of this level.

Connective definitions and axioms
Basic definitional mechanisms
Theory interface
thm datatype, primitive rules of inference

The preceding comprises the basis of the logic. For it to be usable requires a great deal more effort, most of which has been documented in [Gor89, Pau87].

Libraries (Sets, groups, integers, process algebras, ...)
More powerful definitional mechanisms (Recursive types)
Basic theories: Natural numbers, unit, sums, ...
Backwards proof interface (goal stack)
Conversions, tacticals, tactics
Derived rules of inference

There are six sections in this chapter: the first is concerned with types; the second with terms and the role of the symbol table in the implementation; the third describes how *HOL* terms are embedded into Standard ML; the fourth enumerates the primitive methods of theorem production and how they are encapsulated; the fifth section briefly describes some of the extensions that make the logic usable for

large proofs; and the concluding section points out the value of the reimplementation and suggests further improvements.

A basic knowledge of the syntax of functional programming languages is required for this chapter. Also, the idea of ***abstraction*** is heavily used in the implementation. Abstraction is an idea that ties in with notions of access to information and interface. For our purposes, an abstraction is something with an "opaque" interface: "clients" interact with "servers" solely through means of the interface. The important thing is that clients are not allowed to know *how* their requests are serviced, and they are not allowed to affect the operations of the servers except by following the actions allowed by the interface. In the sequel, we will often say "X is of abstract type"; this means that an item of type X can only be obtained by use of the interface. *Encapsulation* is used as a synonym for abstraction.

2.1 Types

Church allowed *simple types*, those defined by closing the function space over the primitive types ι (individuals) and o (boolean truth values). *HOL* allows type variables. Thus types in *HOL* are first order terms over a type signature. The set of types τ is formed from the following disjoint sets of primitive symbols:

- $\mathcal{V} = \{v_1, v_2, \ldots\}$, an infinite set of variables;
- $\mathcal{C} = \{(bool, 0), (ind, 0), (s_1, n_1), \ldots (s_k, n_k)\}$, a set of *type constants*. A type constant is a (name,arity) pair, where the arity must be a natural number. If two type constants have the same name, they must have the same arity, i.e., no two distinct type constants may have the same name. We distinguish $\mathcal{A} = \{(c, a) \in \mathcal{C} : a = 0\} \cup \mathcal{V}$, the ***atomic types***.

The set τ is the smallest set such that:

1. All elements of $\mathcal{A}$ are members of τ.
2. $t_1 \rightarrow t_2 \in \tau$ if t_1 and t_2 are members of τ
3. $(f,n)(t_1,\ldots,t_n) \in \tau$ if $t_1,\ldots,t_n$ are members of τ and $(f,n) \in \mathcal{C}$.

The atomic types *bool* and *ind* represent the (built in) type of boolean values and an infinite set of individuals, respectively. In HOL, the set $\mathcal{C}$ is initially restricted to these two, but can be augmented by means of the *new_type* function, which takes a name and an arity and adds a new type constant to $\mathcal{C}$. Type variables provide polymorphism - a theorem with type variables in it represents a family of theorems derivable by instantiating those type variables. An *instance* δ' of a type δ is obtained by replacing all occurrences of a type variable in δ by a type.

In hol90, the datatype *type* is defined

```
  BOOL
| IND
| type_var   of string
| type_const of (string * int)
| FUN        of (type * type)
| type_app    of (type * (type list))
```

Types must be abstract, since the construction of well-formed types depends on information held in the symbol table. Functional types are written as in normal mathematical usage, *e.g.*, : *ind*→*bool*, rather than $o\iota$, as Church would have written it. Type variables are usually denoted by stars: the type of I, the polymorphic identity function, is : $*\rightarrow*$.

2.2 Terms

A *signature over* $\mathcal{C}$, $\Sigma_{\mathcal{C}}$, is a set of (name,type) pairs, where the type is a member of τ. The set of HOL terms over a signature, $Terms_{\Sigma_{\mathcal{C}}}$, is defined to be the smallest set such that:

- variables - $v : \gamma$ is a term if v is a name and $\gamma \in \tau$
- constants - $c : \gamma'$ is a term if $(c,\gamma) \in \Sigma_{\mathcal{C}}$ and γ' is an instance of γ
- combinations - if $tm_1 : (\gamma_1 \rightarrow \gamma_2)$ is a term and $tm_2 : \gamma_1$ is a term, then $(tm_1\ tm_2)$ is a term of type γ_2
- abstractions - if $v : \gamma_1$ is a term and *body* : γ_2 is a term, then $\lambda v.body$ is a term of type $\gamma_1 \rightarrow \gamma_2$.

Conceptually, each term is a lambda calculus term decorated with a type. But this would imply that each subterm would also have a type, which is redundant and requires a lot of memory, so we include just enough information to reconstruct the type for any subterm. If we view a term as a tree, type information at the leaves suffices to infer a type for the term, and any subterm of it. In hol90, the datatype *term* is defined as

```
  Var   of (type * string)
| Const of (type * string)
| Comb  of (term * term)
| Abs   of (term * term)
```

Then the type extraction function is defined by pattern matching:

```
type_of (Var (ty,_))  = ty
      | (Const(ty,_)) = ty
      | (Comb(t1,_))  = let val FUN(_, beta) = type_of t1 in beta end
      | (Abs(v,body)) = FUN (type_of v, type_of body)
```

Terms are of abstract type, also depending on the symbol table to ensure well-formedness. There are two ways of constructing a *HOL* term: by explicit construction, and by quotation. Explicit construction entails making each subterm and type "by hand". The following functions are used to explicitly construct terms:

```
fun mk_var (s,ty)  = Var(ty,s);

fun mk_const (s,ty) =
   let val ty1 = st_type_of_term s  (* find type of s from symbol table *)
   in
   if (can (match_type ty1) ty) (* if ty is an instance of ty1, then OK *)
   then Const(ty,s)
   else raise DOES_NOT_TYPE ("mk_const.not_type_instance: "^s)   end;

fun mk_comb(t1,t2) =
   let val (FUN(ty2',_)) = type_of t1
   in
   if (type_of t2 = ty2')
   then Comb(t1,t2)
   else raise DOES_NOT_TYPE "mk_comb"   end;

fun mk_abs (v as Var _, body) = Abs(v,body);
    mk_abs _ = raise DOES_NOT_TYPE "mk_abs";
```

There is an alternative to explicit construction: *quotation* provides an embedding of HOL terms in SML, such that the structure of the term can be obtained by parsing and the type of the term can be obtained by type inference. Quotation is discussed in Section 2.3.

When a new constant is defined, what happens intuitively is that the term datatype is extended with a new constructor. However, this notion does not work very well in practice, since it would involve automatically re-defining the datatype and all of the functions encapsulated with term. Therefore, we resort to using a symbol table, the Computer Science version of a signature. We use the HOL symbol table, in which term and type constants are stored after being defined, to ensure the well-formedness of types and terms. When asked to construct a term (or a type) containing a constant, the symbol table is consulted to see if that constant has been defined. If not, an error is signalled. Thus the symbol table is the principal means of maintaining consistency in the logic: when we attempt to define a constant, the definition fails if another constant with the same name is already in the symbol table.

The symbol table also helps in other areas:

- It plays a role in the parsing of *HOL*: when an identifier is encountered, if it is found in the symbol table, then it is a constant; otherwise, it is a variable.
- The parsing status of a constant is also held in the symbol table, *i.e.*, whether it should parse as an infix, a binder, or a prefix, and with what precedence.
- The symbol table is also essential in type inference, since the type of a constant is stored with it in the symbol table.

The symbol table is of abstract type; conceptually, it is included in the encapsulation of types and terms.

2.2.1 Type inference

In hol90, the *quotation* mechanism (described in Section 2.3) plays an important role in getting input. Quotation depends on automatic inference of types for terms. The type inference algorithm used in hol90 is similar to that used in SML, (indeed, both are due to Robin Milner [Mil78]), but is weaker in that it will not infer most general types. Also, *HOL* terms do not have *let* expressions. The algorithm for type inference presented here was communicated to me by Mike Gordon.

Typechecking is bottom up: the type of a term is determined from the types of its subterms. The type inference algorithm proceeds on the structure of the term:

Var The type of a variable is initially a new type variable.

Const The type of a constant is the type given for it in the symbol table, with all type variables consistently replaced by new ones.

Comb To typecheck Comb(t_1,t_2), we do the following:

1. Ensure that t_1 has a function type, say $\alpha \rightarrow \beta$, and find the type of t_2, say γ.

2. Ensure that all free variables with the same name in both terms have the same type. This is done with unification and type instantiation. The result is $t_1' : \alpha' \rightarrow \beta'$ and $t_2' : \gamma'$.
3. Unify $\alpha' \rightarrow \beta'$, with a constructed type $\gamma' \rightarrow \delta$, where δ is a new type variable. This gives a substitution θ.
4. Instantiate the types of t_1' and t_2' with θ.

One optimization of this follows from noting that the two instantiations can be collapsed to one by composing the two substitutions. The routine for typing Comb terms is called **q_mk_comb**.

Abs To typecheck **Abs**(v, *body*), we do the following:

1. Ensure that v is a variable.
2. Unify the type of v with the type of any free occurrence of v in *body*. This gives the substitution θ. Note that due to the bottom-up computation in this algorithm, all free occurrences of v in *body* will already have identical types.
3. Instantiate the types of v and *body* with θ.

The routine for typing abstractions is called **q_mk_abs**.

The function **mk_atom** takes an *HOL* identifier and looks it up in the symbol table: if it is there, it performs the constant case described above; if not, it performs the variable case.

A *type constraint* γ can be applied to any term. Suppose the term has type α. If there is a match θ such that $\theta(\alpha) = \gamma$, then we enforce the constraint by instantiating the term with θ. For example, $\lambda x.(x : bool)$ has type $bool \rightarrow bool$.

It was a design decision in *LCF* that the type inference algorithm should not introduce any new type variables into a term. For instance, if we present "$\lambda x.x$"

to the system, it will not type the term, for the variable x has a new type variable. However, if we constrain the type so that all type variables are user-given, as in "$\lambda x.(x : \star)$", then the term will type. The reason for this is that we wish type inference to be a "compile time" activity. If we infer new type variables, then the application of inference rules requires unification of types at run time, which is too expensive.

The type inference algorithm is not available to the user—it is "built-in" by the mechanism of the next section.

2.3 Embedding HOL in SML

Interactive computer languages, such as SML, often use the **read–eval–print** loop [WH81] as the way of handling user input. It is an infinite loop that repeatedly:

1. prompts the user for input;
2. reads a complete program expression from the input (this includes lexing and parsing);
3. evaluates the output from step 2; and
4. prints the result of the evaluation.

A typical top level looks like this:

```
fun top_level() = (prompt(); print (eval (read())); top_level());
```

For an embedded language, *HOL* for example, things become slightly more complicated. We want a separate reader for *HOL* terms and we also want output that is a *HOL* term to be printed in the *HOL* syntax. The solution we adopt is that the reader for *HOL* will translate *HOL*-strings to SML-strings (*i.e.*, acceptable to the SML parser), and the printer for *HOL* is integrated into the SML printer, so

that *HOL* objects are specially formatted. Therefore, we have provided a form of macro-expansion. We have the new top level:

```
fun top_level() = ( prompt();
                    print+ (eval (read (read_HOL())));
                    top_level());
```

In an SML program, HOL is distinguished from SML by the backquote `. The following is an example of a *quotation* [1] and the expansion it undergoes before being submitted to SML. Notice that the call-by-value semantics of SML nicely enforces the bottom-up computation of type inference.

```
`x ∧ y`
 ==> ``q_mk_comb(q_mk_comb(mk_atom "∧", mk_atom "x"), mk_atom "y")''
 ==> Comb(Comb(Const(BOOL → BOOL → BOOL, "∧"),
               Var(BOOL, "x")),
          Var(BOOL,"y"))
 ==> x ∧ y : term
```

2.3.1 Antiquotation: embedding SML in HOL

It is convenient to mix HOL and SML so that, in a quotation (a bit of HOL), a bit of SML can occur that denotes an HOL object, *i.e.*, that when *evaluated*, evaluates to a term or type. This injection of SML into HOL is called *antiquotation* and is set off by a prefix delimiter, a caret (^). Antiquotation suppresses macro expansion. To implement antiquotation requires that **read_HOL** be augmented (to read_HOL+ say) so that, when translating HOL, if some SML is found, it does not get expanded, but is just inserted unaltered into the expansion. Compare the string expansion of the conjunction in the following example with the previous example.

[1] Unfortunately, hol90 delimits HOL terms with backquotes, since " is used to delimit strings—just the opposite of hol88. It is doubtful if the term "backquotation" will catch on.


```
let val x = `x:bool`
in
``x ∧ y`
end;

==> ``let val x = mk_constraint(mk_atom "x", mk_type "bool")
      in
      q_mk_comb(q_mk_comb(mk_atom "∧", x), mk_atom "y")
      end;''

==> Comb(Comb(Const(BOOL → BOOL → BOOL, "∧"),
              Var(BOOL, "x")),
         Var(BOOL,"y"))

==> x ∧ y : term
```

For the sake of generality, we allow arbitrary levels of quotation and antiquotation. How is this accomplished? To promote **read_HOL** to read_HOL+ requires that when read_HOL+ encounters an antiquote, it gather up the entire body of the antiquote into an SML-string, and concatenate it *untranslated* to S, the translation accumulated so far. We then continue with translation. This "folding in" of antiquotes is handled by the function **get_anti** which has to deal with the following two possibilities in the syntax of antiquotation:

^<**SML identifier**> Then the identifier is merely concatenated with S.

^(<**SML expression**>) The SML expression is concatenated with S. However, there is an interesting possibility here in that the SML expression may deal with an HOL term, which may be presented as a quotation. In this case, read_HOL+ must be called recursively in the construction of the SML expression.

In summary, nested antiquotation is handled by making read_HOL+ mutually recursive with **get_anti**. For example, consider the expansion in the following example.

```
let fun I x = x
    fun K x y = x
in
``(I `x ∧ `(K `y:bool` `x:bool`))`
end;

==> ``let fun I x = x
          fun K x y = x
      in
      I (q_mk_comb(q_mk_comb(mk_atom "∧", mk_atom "x"),
                   K (mk_constraint (mk_atom "y", mk_type "bool") )
                     (mk_constraint (mk_atom "x", mk_type "bool"))))
      end;''

==> Comb(Comb(Const(BOOL → BOOL → BOOL, "∧"),
              Var(BOOL, "x")),
         Var(BOOL,"y"))

==> x ∧ y : term
```

In Huet's class notes from CMU [Hue86], he remarks that *antiquotation* is a form of macro expansion, whereas we have said that quotation is a form of macro expansion. Both statements are true. The difference is that quotation is accomplished by a macro for the metalanguage, while antiquotation may be viewed as a macro for the object language. It would be interesting to iterate this notion, perhaps by formulating a programming language inside the logic, and defining macro expansion for that. Maybe there is a general viewpoint for macros in hierarchies of language embeddings.

2.4 Theorems

An important decision in an LCF-style logic implementation is how and what to encapsulate with the type of theorems. Gordon's paper dealt with a variant of propositional logic in which theorems could only be axioms, or be derived from them by rules of inference. Therefore, the axioms and primitive rules of inference had to be encapsulated with the type of theorems. In contrast, there are several basic ways

to get an object of type *thm* in **hol90**: by use of primitive inference rules, by loading from a theory, by declaring it to be an axiom, by a constant definition, by a constant specification, or by a type definition. All these methods must be encapsulated, *i.e.*, each method must be able to explicitly construct a theorem from a term. Outside of the encapsulation, however, it must be that the only way to produce a theorem is by using the programming metalanguage to (eventually) call one of the primitive theorem construction routines.

In Gordon's paper, abstract data types were used to provide the encapsulation. The strong typing of ML means that the *only* way to get a theorem is via the rules of inference in the abstract data type. For **hol90**, it is possible in theory to incorporate the theorem construction routines in an abstract data type, but that would be a very large and unwieldy entity, so we use Standard ML's "programming-in-the-large" constructs (*signature*, *structure*, and *functor*) [MTH90] to provide encapsulation.

2.4.1 Primitive rules of inference

HOL is a natural deduction style logic. There are three primitive logical constants: $=$ (equality), $\supset$ (implication), and ϵ (Hilbert's choice operator). The code for the primitive rules of inference requires the use of some term constructor and destructor functions, which will query the symbol table, since a fundamental principle in the implementation is that **mk_const** should succeed just in case the constant has been defined, *i.e.*, has been set into the symbol table. But, at the time of definition of the primitive rules of inference, the symbol table is empty. This is only apparently a problem, since the term handling functions access the symbol table at runtime, not compile time, as do the primitive rules of inference. We give definitions for only a few of the constructor functions.

```
local val BBB = mk_type("fun",[bool_type,
                               mk_type("fun",[bool_type, bool_type])])
in
fun mk_conj (t1,t2)= mk_comb(mk_comb(mk_const(∧ ,BBB),t1),t2)
fun mk_disj (t1,t2) = mk_comb(mk_comb(mk_const(∨,BBB),t1),t2)
fun mk_iff  (t1,t2) = mk_comb(mk_comb(mk_const(⇔,BBB),t1),t2)
end;
```

Theorems are members of the datatype *thm*:

```
datatype thm = seq of (term list) * term
```

The following are the primitive rules of inference. Notice that when assumption lists are merged, α-equivalent terms are held to be equal. This helps to provide implementation-wide enforcement of identity-modulo-α-convertibility, *i.e.*, in every situation, α-convertible terms should be considered identical. Also notice that we do not use the quotation parser in the code for the primitive rules of inference or in the term construction routines; to do so would mean that the correctness of the rules would depend on the correctness of the quotation parser, which we don't want (this is discussed further in Chapter Three).

Assumption Introduction (ASSUME)

$$\frac{}{t \vdash t}$$

```
fun ASSUME tm = if (type_of tm = BOOL)
                then seq([tm],tm)
                else raise NOT_A_PROP;
```

Reflexivity (REFL)

$$\frac{}{\vdash t = t}$$


```
fun REFL tm = seq([],mk_eq(tm,tm));
```

Beta-Conversion (BETA_CONV)

$$\frac{}{\vdash (\lambda x.t_1)t_2 = t_1[t_2/x]}$$

$t_1[t_2/x]$ is taken to be the replacement of all free occurrences of x by t_2 in t_1. Bound variables of t_1 may be renamed to avoid binding free variables of t_2 after replacement.

```
fun BETA_CONV (t1 as (Comb(Abs(x,b),tm))) =
              seq([], mk_eq(t1, subst [(tm,x)] b))
  | BETA_CONV _ = raise NOT_A_REDEX;
```

Substitution (SUBST)

$$\frac{(A_1 \vdash l_1 = r_1)/v_1; \ldots; (A_n \vdash l_n = r_n)/v_n \qquad t[v_1, \ldots, v_n] \qquad B \vdash t[l_1, \ldots, l_n]}{A_1 \cup \ldots \cup A_n \cup B \vdash t[r_1, \ldots, r_n]}$$

$t[l_1, \ldots, l_n]$ denotes a term with some free occurrences of subterms $l_1, \ldots l_n$ singled out and $t[v_1, \ldots, v_n]$ is a template for substituting those l_i in $B \vdash t[l_1, \ldots, l_n]$ by r_i. Bound variables of $t[l_1, \ldots, l_n]$ may be renamed in order to prevent free variables in the r_i becoming bound after substitution.

```
fun SUBST replacements template (th as (seq(asl,c))) =
    let val (ths,vars) = unzip replacements
        val (ls,rs) = unzip (map (dest_eq o concl) ths)
    in
    if (aconv (subst (zip ls vars) template) c)
    then seq(aconv_U (asl::(map hyp ths)), subst (zip rs vars) template)
    else th
    end;
```

Abstraction (ABS)

$$\frac{A \vdash t_1 = t_2}{A \vdash (\lambda x.t_1) = (\lambda x.t_2)}$$

x is not allowed to occur free in A.

```
fun ABS (v as Var _) (seq(asl,c)) =
     let val (t1,t2) = dest_eq c
     in
     if (mem v (free_varsl asl))
     then raise FREE_VAR_IN_ASSUMPTIONS
     else seq(asl, mk_eq(mk_abs(v,t1), mk_abs(v,t2)))
     end;
```

Type instantiation (INST_TYPE)

$$\frac{A \vdash t \qquad \theta_{type}}{A \vdash \theta_{type}(t)}$$

θ_{type} is a list of ty_i/v_i pairs, in which v_i must be a type variable and ty_i is a type. $\theta_{type}(t)$ denotes the parallel replacement, in the type variables of the term t, of occurrences of v_i by ty_i. None of the v_i in θ_{type} may occur in the assumption list, and distinct variables of t are not allowed to be the same after the replacement.

```
fun INST_TYPE [] th = th
  | INST_TYPE type_subst (seq(asl,c)) =
       let val asl_vars = all_varsl asl
           val problem_tyvars = intersect (type_vars_in_term c)
                                          (type_vars_in_term_list asl)
       in
       if (null_intersection problem_tyvars (map snd type_subst))
       then seq(asl, inst asl_vars type_subst c)
       else raise INST_TYPE_ERR
       end;
```

Discharging an assumption (DISCH)

$$\frac{\Gamma \vdash t_2}{\Gamma - \{t_1\} \vdash t_1 \supset t_2}$$


```
fun DISCH w (seq(asl,c)) = seq(gather (fn x => not(aconv x w)) asl,
                               mk_imp(w,c));
```

Modus Ponens (MP)

$$\frac{\Gamma \vdash t_1 \supset t_2 \qquad \Delta \vdash t_1}{\Gamma \cup \Delta \vdash t_2}$$

```
fun MP (seq(asl1,c1)) (seq(asl2,c2)) =
   let val (ant1,conc1) = dest_imp c1
   in
   if (aconv ant1 c2)
   then seq(aconv_union asl1 asl2, conc1)
   else raise MP_FAIL
   end;
```

2.4.2 Theories

A *theory* in logic is a possibly infinite set of consequences of some axioms. In *HOL* a theory is much like one in logic, consisting of axioms and theorems derivable from them, but it cannot be infinite, since we have to implement it on a computer. In a *HOL* theory, we can declare new constants, types, definitions, and axioms, thus providing a new context for doing proofs. Then we can prove theorems in this context. A saving in labour occurs because we store the theorems and the context on disk, so that we don't have to replicate the work when we wish to use the results of the theory later on.

HOL theories are structured in an acyclic directed graph, so that work done in ancestor theories can be re-used. When using the system, the current theory is in one of two possible states: draft mode or proof mode. In *draft mode* one builds up the context: constants, types, definitions, and axioms can be installed. Also, theorems can be proved. In *proof mode*, the context can no longer be built up and the only thing that can affect the current theory is the saving of theorems to disk.

Operations on theories include:

new_theory Opens a new context in draft mode.

load_theory Opens an existing context in proof mode.

extend_theory Opens an existing context in draft mode.

close_theory Changes the current context to proof mode.

Both load_theory and extend_theory act as "goto" statements in the theory graph; they replace the current context with a new one. There is one constraint on these two operations: one can only go to a theory that has the current theory as an ancestor.

There is also a graph forming operation: **new_parent** makes previous contexts available to the current one. The context of a theory is provided by constants, types, definitions, axioms, and theorems of the current theory as well as those of all parent theories. There must, of course, be a test for acyclicity in **new_parent**. Further details about the theory interface can be found in [Pau87].

2.4.2.1 Theory file syntax

Here we present the theory file interface for hol90. Each line in a theory file is of one of the following forms.

Constants.

```
constant <name> PREFIX <precedence> <type>
constant <name> INFIX <precedence> <type>
constant <name> BINDER <precedence> <type>
type_constant <name> <arity>
```

Theorems and their kin.

```
theorem <name> <term>
axiom <name> <term>
definition <name> <term>
```

Parents.


```
parent <name>
```

The SML Definition does not describe a rich set of input/output functions: only string input and output is specified. Therefore, when we write a term out to the file, we first transform it to a string version of the syntax tree. When we load the string back in, we simply feed it to the compiler and the term reappears in the environment.

2.4.3 Definitional mechanisms

There are three ways to introduce a theorem with the definition mechanisms of *HOL*: by defining a constant, by specifying some constants loosely, and by defining a type. In this section, we describe how the basic constants are defined in the logic. Before defining anything, we must install the primitive constants in the logic.

```
new_theory "min";
new_infix(=, ':*->(*-> bool)');
new_infix(⊃, ':bool->bool->bool');
new_binder(ε, ':(*->bool)->*');
close_theory();
```

2.4.3.1 Constant definition

The most basic principle of definition in *HOL* is that for defining constants. If $c = tm : ty$ is a well typed term such that

1. c is not in the symbol table, *i.e.*, $c \notin \Sigma_C$
2. tm is a closed term, *i.e.*, it has no free variables
3. all the type variables in tm are in the type of c

then $c : ty$ gets added to the symbol table. Also, $c = tm$ is added to the current theory file as a definition, and $\vdash c = tm$ is returned. There are various flavours of constant definition, differing only in the parsing status of the introduced constant.

2.4.3.2 Constant specification

Recently, another principle of definition has been introduced. In a *constant specification*, one supplies a name c and a theorem $\vdash \exists x.p[x]$ stating that there exists an object satisfying the property of interest. The result is an axiom $\vdash p[c]$. This gives a facility for so-called "loose specifications", those that do not fully specify the introduced constant. Constant specification is actually a bit more general, in that a set of constants satisfying a property can be specified. Formally a constant definition is a pair

$$< (c_1, \ldots, c_n), \lambda x_1 \ldots x_n.tm[x_1, \ldots, x_n] >$$

such that

1. The $x_1, \ldots, x_n$ are distinct.
2. $tm[x_1, \ldots, x_n]$ has type *bool*.
3. The $c_1, \ldots, c_n$ are distinct and not already in the symbol table.
4. If we let the type variables in $\lambda x_1 \ldots x_n.tm[x_1, \ldots, x_n]$ be Θ, and the type variables in each of $x_1, \ldots, x_n$ be $\theta_1, \ldots \theta_n$, then $\Theta = \theta_1 \wedge \ldots \wedge \Theta = \theta_n$
5. $\vdash \exists x_1 \ldots x_n.tm[x_1, \ldots, x_n]$

The result of a constant definition is that

1. $c_1, \ldots, c_n$ get added to the symbol table, each c_i having the type of the corresponding x_i.
2. $tm[c_1, \ldots, c_n]$ is added to the current theory file as a definition
3. $\vdash tm[c_1, \ldots, c_n]$ is returned

After installing the primitive constants in the symbol table, as above, we wish to define the logical constants. But there is a problem. Constant definition is taken as a special case of constant specification in **hol88**. Recall that a constant specification

requires an existence proof for the term being introduced. The problem is that to prove an existence theorem requires (at the very least) that $\exists$ already be installed in the symbol table. But $\exists$ is a *defined* symbol and hence would need an existence proof, which is not possible. We apparently have a hole in our logic. Fortunately, it is not a serious problem, for the first definition in the logic is that of the existential quantifier:

```
val _ = new_theory "bool";
val EXISTS_DEF = simple_binder_definition("EXISTS_DEF",
    '∃ = λ(P:* → bool). P ($ε P)');
val _ = close_theory();
```

To define it, we use a simple principle of definition, which does not require proof, but merely checks a syntactic criterion, then installs the definition. It is not particularly elegant to have two principles of definition, but they are both sound means of extending the logic. Another method (the one used in **hol88**) is to just accept the "definition" of $\exists$ since it obviously meets the criteria of the principle of constant definition.

Having installed $\exists$, we can use the normal principle of definition for the rest of the logic, starting with the logical connectives.

```
val _ = extend_theory "bool";

    val T_DEF = new_definition("TRUTH",
       'T = ((λ (x:bool). x) = (λ(x:bool). x))');
    val FORALL_DEF = new_binder_definition ("FORALL_DEF",
        '∀ = (λ(P:* → bool). P = (λ x. T))');
    val AND_DEF = new_infix_definition("AND_DEF",
                   '∧ = (λ t1. λ t2.∀ t. (t1 ⊃ (t2 ⊃ t)) ⊃ t)');
    val OR_DEF = new_infix_definition("OR_DEF",
                  '∨ = (λ t1 t2. ∀ t. (t1 ⊃ t) ⊃ ((t2 ⊃ t) ⊃ t))');
    val IFF_DEF = new_infix_definition("IFF_DEF",
                   '<=> = (λ t1 t2. (t1 ⊃ t2) ∧ (t2 ⊃ t1))');
    val F_DEF = new_definition("F_DEF", 'F = (∀ t.t)');
    val NOT_DEF = new_definition("NOT_DEF", '¬ = (λ t. t  ⊃  F)');

val _ = close_theory();
```

2.4.3.3 Type definition

The last definitional mechanism deals with types. One of the most powerful means of "raising the level of discourse" in the system is the definition of new types from existing ones. With this facility, the user can single out a well-defined set of objects to reason about, and do that reasoning without the clutter engendered by always dragging the identifying predicate about.

A new type constant is added to the system by a *type definition*, which requires the following information:

- An existing type σ.
- A characteristic predicate $P : \sigma \rightarrow bool$ that is true of an object just when that object is in the subset of σ that is to "become" the new type.
- A theorem stating that the set defined by P is not empty: $\vdash \exists(x : \sigma).Px$.
- a name *ty* for the new type.

The result of this is that

- A new type constant *ty* of arity equal to the number of distinct type variables in σ is added to the type signature ($\mathcal{C}$).
- An axiom is asserted that says *ty* is isomorphic to the subset of σ defined by P.

2.4.4 Axioms

There are five axioms, introduced into the theory *bool* with new_axiom:

BOOL_CASES_AX	$\vdash \forall(b : bool).\ (b = T) \vee (b = F)$
IMP_ANTISYM_AX	$\vdash \forall b_1 b_2.\ (b_1 \supset b_2) \supset (b_2 \supset b_1) \supset (b_1 = b_2)$
ETA_AX	$\vdash \forall f : (* \rightarrow *).\ (\lambda x.(fx)) = f$
SELECT_AX	$\vdash \forall(P : * \rightarrow bool)\ (x : *).\ (Px) \supset P(\varepsilon P)$
INFINITY_AX	$\vdash \exists(f : ind \rightarrow ind).\ One\_One\ f \wedge \neg(Onto\ f)$

2.5 Extensions

From a certain point of view, the *LCF* approach to theorem proving is Socialist and hence deserves its own *Manifesto*:

> The user controls the means of (theorem) production.

Once we have implemented terms and types and encapsulated the means of theorem production, we have implemented the logic. To make the logic usable requires a lot more work. There are several unifying ideas that make this easier. First, the metalanguage is a programming language, so a derived rule of inference is merely a program that returns a value of type *thm*. Very powerful derived rules of inference can be programmed in this way; Chapters Five and Six give detailed examples of this. Although *all* proofs are executed by forward inference, it is well known that the proof is often easier to find when working *backwards* from the proposition. This is known as goal-directed proof. The *LCF* family of theorem provers provide the fundamental notion of *tactic*[Mil85] to do goal-directed proof. Other very helpful features provided by hol88 and ported to hol90 are facilities for rewriting [Pau83] and recur-

sive type definition [Mel89]. We refer to the HOL Manual for further information and examples [Gor89].

2.6 Conclusions

The re-implementation comprises approximately 18,000 lines of code and took approximately 7 months of work. I have been helped at various points by Mike Gordon, who illuminated some of the mysteries of type inference and gave me a start on quotation; Dave Matthews, who showed me how to install a new top level loop in PolyML; and Rajagopal Nagarajan, who has ported many theories.

In the course of the reimplementation, some bugs in hol88 have been found and fixed and some infelicities have been overcome:

- There was a bug in α-conversion.
- Some primitive rules of inference did not consistently treat α-convertible terms as identical.
- There were some problems with the implementation of the principle of definition.
- The theory file syntax is now documented.
- Dummy theorems on theory files that describe the parsing status of constants are no longer necessary.
- It is no longer possible to drop down into Lisp and commit logical atrocities.

The main value of the reimplementation is that this chapter shines a light into the dark corners of the previous implementation. The information about what the system *is* is now available; this gives us a basis for maintaining the code in the system and for improving the system.

There is still much interesting work to be done to improve the usefulness of hol90:

- *HOL* is expressive enough to directly embed various calculi such as modal logic, process algebra, *etc.* For readability, it is desirable that each of these have its own syntax. It will happen that mixtures of these calculi will take place, *e.g.*, one may want to use modal logic to prove a proposition in process algebra. Therefore, it is of interest to see how the parsers and pretty printers for subcalculi can be gracefully merged.
- It is clear that interactive logic implementations can benefit by a window system interface, but, as yet, there is no unifying idea.
- Something that is not interesting, but important in the short run is the provision of backwards compatibility. A useful program for this would be an ML-to-SML translator. Also, a program to translate theory files from the old format to the new format would be helpful.

Chapter 3

The Prelogic

The concept of the *prelogic* is due to Haskell Curry [CF58]; he points out that some of the operations necessary for defining a logic system are complex. His example was to compare the simplicity of an operation like *modus ponens* with the *process* of substitution. He argued that complex primitive operations like substitution should not just be taken as given. Of course, Curry's solution was to invent the calculus of *combinators*, which has very simple substitution properties. *HOL* has a prelogic too, in fact, the previous chapter can be seen as an excursion through the prelogic of *HOL*. However, we have postponed discussion of the three basic algorithms in the prelogic until this chapter.

The three fundamental algorithms of *HOL* provide the ability to compare terms for α-convertibility (**aconv**), to do term replacement (**subst**), and to instantiate the type variables in a term (**inst**). We examine the specifications and implementations of them in this chapter.

Identifying the prelogic of *HOL* provides a way to armour the Achilles' heel of **hol90**: how does one know that the logic implementation is itself correct? After all, the algorithms used in theorem proving are some of the most complex ever devised. Why should we place our trust in an implementation of them? The answer is that we shouldn't. Like any large piece of software, the reliability of a theorem prover depends on how good its specifiers and programmers were, how long it has been used, how vociferous in tracking down bugs its maintainers and user community have been, *etc.*

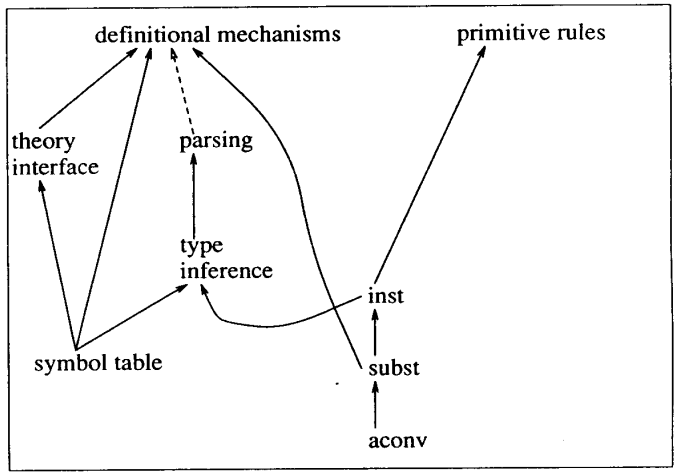


Figure 3.1: Prelogic Dependencies

One alternative to this unsatisfactory situation is to verify the implementation of the theorem prover. This is not feasible for a typical theorem prover, which is a giant program, where theoremhood can be determined in any number of places. In contrast to these less well structured theorem provers, one that uses Milner's idea of encapsulating the production of theorems has at least a chance of being verified with current technology. For then, all we must verify is the prelogic.

The prelogic has its own internal dependencies, as can be seen by figure 3.1. There are two ways to get a theorem: through proof, and through definition. These

both depend on the primitive algorithms; any partial verification of the logic requires us to verify **aconv**, **subst**, and **inst**. The parser and type inference mechanism are not essential to the correctness of the logic, hence the dashed line.

A group working for ICL in Great Britain has also re-implemented hol88, paying close attention to the correctness of the implementation. Towards that end, Rob Arthan of ICL has formally specified *HOL* in *HOL* itself [Art90]. His specifications encompass well formed types and terms, theories, rules of inference, theorems, and axioms. His specifications of the primitive rules of inference necessitate specifications of **aconv**, **subst**, and **inst**. These specifications differ from the ones presented here in that they are more executable in flavour: for instance, his definition of **aconv** is formulated as a function that recurses on subterms, whereas the specification of aconv presented here has been taken from the lambda calculus literature, and is phrased as a miniature proof system. It would be reassuring to prove the equivalence of the two sets of specifications.

3.1 α-conversion

Terms that differ only in the names given to bound variables are called α-convertible. For example, $\lambda x.\lambda y.x$ is α-convertible to $\lambda y.\lambda z.y$, but not to $\lambda y.\lambda y.y$. In all models of the lambda calculus, α-convertible terms are identical. We wish to support this view of terms in *HOL*.

In the following, we use the notion of *occurrence*, which provides an unambiguous notation for accessing subterms of a term. Huet[1980] has a detailed development for first order terms, which is easily modified for the lambda calculus. We do not provide a definition, only saying that the notation P/occ, where P is a term, denotes the subterm of P addressed by *occ*; also, $P/occ := M$ denotes a term identical with P except that the subterm addressed by *occ* is M. A *free* occurrence of a variable v

has no binder λv on the path from the occurrence to the root of the term. FV is a function that returns the free variables in a term.

3.1.1 Specification

Definition. *Substitution (for variables)* [HS86]. Define $[N/x]M$ to be the result of substituting N for every free occurrence of x in M, changing bound variables to avoid clashes. The precise definition is by induction on M:

- $[N/x]x = N$;
- $[N/x]a = a$, for all variables a not equal to x, and all constants;
- $[N/x](P\ Q) = ([N/x]P\ [N/x]Q)$;
- $[N/x]\lambda x.M = \lambda x.M$;
- $[N/x]\lambda y.P = \lambda y.[N/x]P$, if $y \neq x$ and $(y \notin FV(N)\ \vee\ x \notin FV(P))$.
- $[N/x]\lambda y.P = \lambda z.[N/x]([z/y]P)$ if $y \neq x$ and $y \in FV(N)$ and $x \in FV(P)$. z is chosen to be the first variable not in $FV(NP)$.

Typed substitution is defined similarly, with the added proviso that the term substituted for a variable must have the same type as the variable. For simplicity, we will ignore types in the verifications; the addition of type information would add another level of (usually superfluous) detail to the proofs.

Definition. *One Step Renaming (OSR).* Let a term P contain an occurrence of $\lambda x.M$ and let $y \notin FV(M)$. The act of replacing this $\lambda x.M$ by $\lambda y.[y/x]M$ is called a *One-Step Renaming* (OSR) in P.

```
OSR(P,Q) = ∃ occ M x y. P/occ = λ x.M ∧
                        Q/occ = λ y.[y/x]M ∧
                        y ∉  FV(M) ∧
                        y ≠  x ∧
                        Q/occ := λ x.M  = P
```

Definition. *α-equivalence.*[HS86]. P and Q are α-equivalent, written $P \equiv_\alpha Q$ *iff* Q has been obtained from P by a finite series of one-step renamings. This is formalized by the definition of an *OSR_proof*:

```
OSR_proof P Q 0  =  (P = Q)   ∧
OSR_proof P Q (n+1) = ∃ P'. OSR(P,P')  ∧  OSR_proof P' Q n
```

(Notice that this is not the only choice possible for α-conversion. A more general formulation would take into consideration renaming between type variables. For example, it could be allowed that

$$\lambda x.(x : *) \equiv_\alpha \lambda y.(y : **)$$

since type variables are implicitly universally quantified at the outermost level.)

3.1.2 Implementation

The algorithm for α-conversion, aconv, traverses its two term arguments in parallel, maintaining a scope in each term by means of the S argument to acnv. S may be thought of as an encoding of two lists S_1 and S_2; S_1 is the scope for the first term and S_2 is the scope for the second term. When λ-abstractions are encountered, the bound variables are bundled into a pair and put on the front of S, after their types are confirmed to be identical. When variables v_1 and v_2 are encountered, we look along S_1 to find how many "hops" it takes to get to v_1. We do the same for v_2 on S_2. If the number of hops to find v_1 is the same as the number of hops to find v_2, the variables are the same, and the algorithm continues. If neither variable is on S, then they are free variables and hence must be identical for the algorithm to continue. If there is a difference in the number of hops, the two terms are not α-convertible. If aconv returns true, it has traversed both terms completely and not found any clashes in looking up variables or in the structures of the two terms.

The algorithm for aconv is a corrected and modified version of the aconv for hol88.

The code for aconv depends on routines for looking an item up in a list:

```
datatype 'a Option = NONE | SOME of 'a;

local
fun lindex item [] _ = NONE
  | lindex item ((a,b)::L) n = if (item = a)
                               then (SOME n)
                               else lindex item L (n+1)
in
fun left_index item L = lindex item L 0
end;
```

NONE is returned when we haven't found the item being searched for, while SOME n indicates the number of hops required to find the item. The code for right_index is exactly the same, except that it looks in the right side of the pair.

```
local
fun acnv (v1 as Var _) (v2 as Var _) S =
        (case (left_index v1 S, right_index v2 S)
           of (NONE,NONE) => v1 = v2 |
              (SOME n, SOME n') => n = n' |
              (_,_) => false) |
    acnv (c1 as Const _) (c2 as Const _) _ = (c1=c2) |
    acnv (Comb(M1,N1)) (Comb(M2,N2)) S = (acnv M1 M2 S) andalso
                                         (acnv N1 N2 S) |
    acnv (Abs(v1 as Var(ty1,_),M1)) (Abs(v2 as Var(ty2,_),M2)) S =
              (ty1 = ty2) andalso (acnv M1 M2 ((v1,v2)::S)) |
    acnv _ _ _ = false
in
fun aconv tm1 tm2 = acnv tm1 tm2 []
end
```

aconv implements a "dynamic" version of deBruijn terms [dB72]: instead of storing the distance to the binder at each variable occurrence, as is done in deBruijn terms, we compute the distance each time we encounter a variable in the parallel traversal of the two terms. Therefore, our implementation is slower for checking α-convertibility between terms.

It is readily seen that acnv terminates by noticing that terms in recursive calls are strictly smaller than the argument terms. Hence aconv terminates.

The correctness of aconv is established in Chapter Four.

3.2 Term substitution

A fundamental operation in theorem proving is term replacement, or substitution. Usually, substitution is defined, as earlier, exclusively in terms of replacing variables by terms. A generalization of this, used in *HOL*, is replacement of arbitrary terms by terms. The function that performs general term substitutions in hol90 is subst. It is common to define substitution in the λ-calculus so that it might fail in cases where it would succeed on a term that was α-equivalent [And86]. For example,

```
[y/x] λy.x
```

would fail, whereas

```
[y/x] λz.x
```

would succeed. For a theorem prover, this situation arises often enough that failure is frustrating. Therefore, we rename bound variables so that the substitution succeeds.

Here is an example, taken from the source code for *LCF*. Although it does not illustrate the replacement of arbitrary terms, it shows that renaming problems can be subtle.

```
theta = { x'/y; x/z }
tm = λ x. (f y z)
subst theta tm = λ x''.f x' x : term
```

Substitution is the parallel replacement of terms by terms. A naive substitution algorithm will perform the replacements in `theta` to get

```
λx.  (f x' x)
```


Now we have captured the incoming `x` and must rename the bound variable `x`. The renaming algorithm must have a fixed strategy for renaming `x`; if it naively just primes `x` to be different than the free variables of the body ($\{y, z\}$), `x` becomes `x'` and we have

```
λx'.  (f x' x)
```

Now we have captured a different incoming variable. What we must do is rename the bound variable so that it is different from the free variables in the body and the incoming free variables from the substitution.

3.2.1 Specification

We first define when a term is free in another term.

Definition. *free_in.* A term tm is free in M when an occurrence of tm in M has none of its free variables bound by the scope of the occurrence. The definition is by induction on M:

- `free_in` $tm\ M = true$, if $tm \equiv_\alpha M$
- If $tm \not\equiv_\alpha M$, then
  - `free_in` $tm\ (M_1\ M_2) =$ (`free_in` $tm\ M_1$) $\vee$ (`free_in` $tm\ M_2$)
  - `free_in` $tm\ \lambda v.body = false$, if $v \in FV(tm)$
  - `free_in` $tm\ \lambda v.body =$ `free_in` $tm\ body$, if $v \notin FV(tm)$
  - `free_in` $tm\ M = false$, in all other cases.

Definition. *restriction.* The restriction of a substitution θ to the free terms of a term M is denoted $\theta \restriction M$ and is defined as

$$\{N/tm \in \theta : free\_in\ tm\ M\}$$

The following definition generalizes the notion of substituting for one item in a term to substituting for many items in a term.

Definition. *Substitution (for arbitrary terms).* Define $[N_1/tm_1, \ldots, N_n/tm_n]M$ to be the result of the parallel replacement of every free occurrence of tm_i by N_i in M, changing bound variables to avoid clashes. The formal definition uses induction on M:

- $[N_1/tm_1, \ldots, N_n/tm_n]M = N_i$, if for some i, $tm_i \equiv_\alpha M$.
- if there is no i such that $tm_i \equiv_\alpha M$ then (letting $\theta = [N_1/tm_1, \ldots, N_n/tm_n]$)

  $\theta(a) = a$, if a is a variable or constant

  $\theta(P\ Q) = (\theta(P)\ \theta(Q))$

  $\theta(\lambda v.body) = \lambda v.body$, if θ is empty, or there does not exist a free occurrence of any tm_i in $\lambda v.body$.

  $\theta(\lambda v.body) = \lambda v.\ (\theta \uparrow \lambda v.body)(body)$, if there is a N_i/tm_i in θ such that tm_i is free in $\lambda v.body$, and for every N_j/tm_j in θ, tm_j is free in $\lambda v.body$ implies N_j is free in $\lambda v.body$.

  $\theta(\lambda v.body) = \lambda z.(\theta \uparrow \lambda v.body)((z/v)body)$, if there exists an N_i/tm_i in θ such that tm_i is free in $\lambda v.body$, $v \in FV(N_i)$, and z is not free in $body$ and z is not free in any N_j with a corresponding tm_j free in $\lambda v.body$.

It is difficult to come up with a specification of replacement that is not algorithmic. The above specification of substitution nearly defines a recursive function for performing substitution. The implementation actually follows the specification quite closely, differing only in that it is more efficient in checking the conditions for substituting under a λ.

3.2.2 Implementation

The implementation of **subst** depends essentially on *exceptions*, a feature of Standard ML. Before an expression is evaluated, an *exception handler* may be set up. The syntax [1] for this is

```
<exp1> handle <exception> => <exp2>
```

In the course of evaluation of `exp1`, an exception can be *raised*. This exception "propagates" until it reaches its handler. When the exception meets its handler, execution of `exp2` begins, in the state in effect at the time of evaluating `exp1`. (This does not cover the results of evaluating intervening imperative expressions, which do have a permanent effect on the state.) Conceptually, the handler gets placed on the call stack, and exception propagation is much like returning from function calls.

For example, consider the SML expression

```
(2 div 0) handle Div => 1
```

Before the divide is executed, the handler is set up. When the machine raises the exception `Div`, it propagates to the (only) handler and execution resumes, returning 1.

The algorithm for **subst** presented here uses exceptions in their full generality: an exception may carry a value along with it when it propagates. Also, when an exception meets its handler, the handler can raise another exception, in effect "kicking it upstairs".

A substitution is a list of *bindings* (pairs of terms). The second member of such a pair is called the *redex*, and the first member is called the *contractum*. The algorithm traverses the term being substituted into, checking, at each subterm, whether the subterm is α-equivalent to a redex. This is accomplished by the use of aconv_assoc:

[1] This simple syntax does not cover the full generality of exceptions in SML, but it serves to introduce the concept.

```
fun aconv_assoc item [] = NONE |
    aconv_assoc item ((entry as (_,key))::rst) =
          if (aconv item key)
          then SOME entry
          else aconv_assoc item rst;
```

Given substitution θ and term `tm`, the algorithm works in the following manner (the boxed code fragments in the following are optimized: they may not make sense until the optimizations presented in Section 3.2.2.2 are read):

1. Search θ for a binding `(tm1,tm')` such that `tm` $\equiv_\alpha$`tm'`. If no such binding exists, go to 2.

   ```
   fun sub1 theta scope tm =
      case (aconv_assoc tm theta)   (* test 1 *)
        of NONE => sub2 theta scope tm |
           SOME (tm1,_) =>
             let val clashes = intersect scope (free_vars tm1)
             in
             if (null clashes)  (* test 3 *)
             then tm1  (* perform replacement *)
             else raise (INCOMING_VARS_CLASH clashes)
             end
   ```

   If a free variable of `tm'`, or, equivalently, `tm`, occurs in the scope, it is not valid to perform the substitution, so `tm` is returned. Otherwise, if no free variable of `tm1` occurs in the scope, we have found a replacement, and return `tm1`. Otherwise, we raise an exception. The exception must propagate back to the right binding, so we return the offending variables (the free variables of `tm1` bound by the scope) with the exception. The exception propagates up the stack, while enclosing handlers attempt to pluck off their bound variables. The propagation ends when there is a single offending variable left and it meets its enclosing abstraction. At that point, we rename the variable, rename all free occurrences of it in the body, then resume applying the substitution to the body.

2. There are four cases, based on the term structure:

   `tm = (Var _)`. There is no structure to recurse into, and there is no binding in θ for this variable, so we just return `tm`.

   `tm = (Const _)`. There is no structure to recurse into, and there is no binding in θ for this constant, so we just return `tm`.

   `tm = Comb(tm1,tm2)`. Apply the substitution to both `tm1` and `tm2`, getting `tm1'` and `tm2'`. Return `Comb(tm1',tm2')`.

   ```
   sub2 theta scope (Comb(t1,t2)) = Comb(sub1 theta scope t1,
                                         sub1 theta scope t2)
   ```

   `tm = Abs(v,body)`. Add `v` to the scope, set up a handler, and apply substitution θ to `body`. If an exception occurs because of this abstraction, rename `v` to a `v'` that is different from any variable in `tm_vars` and replace all free occurrences of `v` in the body by `v'` to get `body'`. Then apply θ to `body'` to get `body''`. Return `Abs(v',body'')`. If the exception occurs because of an enclosing abstraction, we propagate the exception. The final thing that may happen is for the error to be caused by a number of enclosing abstractions. In this case, we have two choices:

   - the abstraction can rename and then propagate the error
   - the abstraction can propagate and eventually rename.

   Currently, we just propagate the error and eventually rename, because it seems to be a very rare case, and it keeps the code size down.

```
sub2 theta scope (Abs(v,body)) =
    let val theta' = restrict_wrt theta v  (* optimization 1 *)
    in
    Abs(v,sub1 theta' (v::scope) body)
    handle INCOMING_VARS_CLASH [clash_var] =>
              if (clash_var = v)
              then let val v' = variant (!tm_vars) v
                       val _ = tm_vars := v'::(!tm_vars)
                   in   (* optimization 2 *)
                   Abs(v', sub1 ((v,v')::theta') scope body)
                   end
              else raise (INCOMING_VARS_CLASH [clash_var])
         | INCOMING_VARS_CLASH [] => raise SUBST_IMP_ERR
         | INCOMING_VARS_CLASH clashes =>
                  raise (INCOMING_VARS_CLASH (set_diff clashes [v]))
    end
```

3.2.2.1 Renaming

When we rename bound variables, we must choose a name that

- does not occur in the scope, for otherwise we may capture a variable bound by the scope;
- is not a free variable in the body;
- is not a bound variable of an inner scope, for if that inner scope must be renamed as a consequence of the current renaming, unnecessary work has to be done;
- does not occur free in the contractum terms of the substitution (this is necessary because of optimization 2 below).

Therefore, it seems simplest to take the union of the set of free variables in the contractum terms of the substitution and the set of variables occurring in the term and, when we have to, rename a bound variable to be different from every variable in that set. (Of course, we must then add that new variable to the set.)

3.2.2.2 Optimizations

1. When we enter an abstraction, we can immediately remove any binding in θ that has the bound variable free in the redex, for we will never be able to substitute for that redex.

2. When we have to do a variable renaming, say `v` to `v'`, we restrict θ as with optimization 1 to get θ', but then we add the binding `(v',v)` to θ'. This saves us from doing two substitutions when one would do: instead of applying the variable renaming substitution `[(v',v)]` to `body` to get `body'` and then applying θ' to `body'` to get `body''`, we do the equivalent by applying `(v',v)::`θ' to `body`. There is a subtle reason why this works: we don't add `v'` to the scope for the recursive call. We don't have to, because `v'` is different from every other variable in the term and every other variable that can enter the term from this substitution. Since `v'` is added to the **tm_vars** list, it will be different from any other variable generated in the future also. Therefore, `v'` can never clash with anything and can be excluded from the clash check.

3.3 Type instantiation

One feature that sets *HOL* apart from other logics is the use of type variables. A theorem with type variables in it is actually a family of theorems: one for every type. The way one gets a member of that family is by type instantiation, provided by the primitive rule of inference INST_TYPE, which is primarily implemented by a call to the routine **inst**.

A term is instantiated by a type by applying a *type substitution* to the type variables in its types. The function **type_subst** is almost identical to its specification, so we merely provide the definition of **type_subst**.

```
fun type_subst [] ty = ty (* If no substitution, don't do anything *)
  | type_subst theta BOOL = BOOL
  | type_subst theta IND = IND
  | type_subst theta (c as type_const _) = c
  | type_subst theta (v as type_var _) =
       (case (assoc2 v theta)  (* Is there a binding? *)
          of NONE => v         (* No, do nothing *)
           | SOME(ty,_) => ty) (* Yes, replace *)
  | type_subst theta (FUN(ty1,ty2)) = FUN(type_subst theta ty1,
                                          type_subst theta ty2)
  | type_subst theta (type_app(c,type_list)) =
       type_app(c, map (type_subst theta) type_list);
```

3.3.1 A naive solution

Given the type substitution code, it would seem to be trivial to write inst:

```
fun inst theta (Var(ty,s)) = Var(type_subst theta ty, s)
  | inst theta (Const(ty,s)) = Const(type_subst theta ty, s)
  | inst theta (Comb(tm1,tm2)) = Comb(inst theta tm1,inst theta tm2)
  | inst theta (Abs(v,body)) = Abs(inst theta v,inst theta body)
```

But there is a problem, which affects the soundness of the system, discovered by the group at ICL: naive inst can change the term structure; consider

```
inst [(':bool',':*')] 'λ(x:*).λ(x:bool).(x:*)'
```

The result is

```
'λ(x:bool).λ(x:bool).(x:bool)'
```

The variable has been captured by the inner λ, so the resulting term is structurally different than the initial term. This is not sound. The example given by ICL was this:

```
val x = 'x:*';

val thm1 = TAC_PROOF(([],'!(y:*). y = (λ ^x. λ x.^x) y T'),
                     BETA_TAC THEN REWRITE_TAC[]);
val thm2 = TAC_PROOF(([],'F'), ONCE_REWRITE_TAC[thm1] THEN BETA_TAC);
```


The value *thm1* is valid. If we show its types, we see that we have our previous example embedded in the theorem.

```
val thm1 = ⊢ !y:*. y:* = ((λ x:*. λ x:bool. x:*) (y:*) (T:bool)) : thm
```

We attempt to prove falsity by rewriting *'F'* with *thm1*. Instantiation is necessary in rewriting. To be specific, the left hand side of *thm1* must be made identical to *'F'*, before it can be replaced with the (instantiated) right hand side. This is done by applying both a term and a type substitution to *thm1*. The type substitution `[(':bool','*')]` must be applied first. This is done by inst. An incorrect implementation of inst would give

```
⊢  y:bool = ((λ x:bool.λ x:bool.x:bool) (y:bool) (T:bool)) : thm
```

The damage done, the term substitution would give

```
⊢ F = ((λx:bool.λx:bool.x:bool) F T) : thm
```

and β-conversion yields

```
⊢ F = T : thm.
```

3.3.2 Specification

Define the action of a type substitution on a list of variables as

$$\theta[v_1, \ldots, v_n] = [\theta v_1, \ldots, \theta v_n]$$

Definition. *inst.* The definition of type instantiation is by induction on the structure of the term M:

Var. $inst\ \theta\ (Var(ty, s)) = Var(type\_subst\ \theta\ ty, s)$,

Const. $inst\ \theta\ (Const(ty, s)) = Const(type\_subst\ \theta\ ty, s)$

Comb. $inst\ \theta\ (M1\ M2) = (inst\ \theta\ M1, inst\ \theta\ M2)$,

Abs. $inst\ \theta\ (Abs(v, body)) =$

- $Abs(inst\ \theta\ v,\ \ inst\ \theta\ body)$, provided
  $\theta(FV(\lambda v.body)) = FV(\lambda(inst\ \theta\ v).(inst\ \theta\ body))$.
- $Abs(inst\ \theta\ z, inst\ \theta\ ([z/v]body))$,
  if $FV(\lambda(inst\ \ \theta\ \ v).(inst\ \ \theta\ \ body))$ is a subset of $\theta(FV(\lambda v.body))$ and $z \notin FV(body)$ and z and v have the same type.

Although we do not specify it here, **inst** must also take a list of variables that all renaming variables must be disjoint from.

3.3.3 Implementation

The solution to the problem combines features of **aconv** and **subst**. At a variable, we first perform the type substitution. Then we must find out if it is still bound by the same abstraction as it was before the type substitution. We perform a similar trick as in α-conversion: we look back in both new and old scopes; if the number of hops are equal, then things are as they should be; if the number of hops in the new scope is less than in the old scope, the new variable has been captured by an intervening binder and we raise an exception that will be handled by the code for the abstraction case; finally, if the number of hops in the new scope is more than in the old scope, then a programming error has been made.


```
(* Variable case *)
inst theta S (v as Var(ty,vstr)) =
   let val v' = Var(type_subst theta ty,vstr)
   in
   case (left_index v S, right_index v' S)
     of (NONE, NONE) => v' (* free var goes to free var *)
      | (NONE,_) => raise VAR_CLASH v'  (* free var gets bound *)
      | (_,NONE) => raise INST_IMP_ERR "bound var becomes free."
      | (SOME i1,SOME i2) =>
           if (i1=i2)  (* same binder *)
           then v'
           else if (i1>i2)  (* bound in inner scope, rename inner scope *)
                then raise VAR_CLASH v'
                else raise INST_IMP_ERR "no longer bound by original binder!"
   end
```

The constant case is the easiest: the type of the constant is instantiated. Since constants cannot be bound, we have no more to do in this situation.

```
(* Constant case *)
inst theta _ (Const(ty,c)) = Const(type_subst theta ty,c) |
```

Combinations are also easy: the substitution is performed recursively.

```
(* Comb case *)
inst theta S (Comb(t1,t2)) = Comb(inst1 theta S t1,inst1 theta S t2) |
```

In the abstraction case, the variable being bound has its type instantiated, if possible, then we proceed to instantiate the types in the body of the abstraction. Also, a handler is set up. If a variable clash occurs, an exception is raised at the variable and propagates up the call stack to the handler that was set up for that variable. Then the name of the variable is changed, the new variable is substituted for the old variable throughout the body, and we proceed to instantiate types in the new body. Because we rename the variable to be completely different than any other variable, we are guaranteed that we will not have to rename.

```
(* Abstraction case *)
inst theta S (Abs(v as Var(vty,vstr),body)) =
    let val v'_ty = type_subst theta vty
        val v' = Var(v'_ty,vstr)  (* new type, old name *)
    in
    Abs(v', inst theta ((v,v')::S) body)
    handle
    VAR_CLASH (clash_var as Var _) =>
       if (clash_var = v')
       then    (* old type, new name *)
       let val (v'' as Var(_,vstr'')) = variant (!tm_vars) v
           val _ = tm_vars := v''::(!tm_vars)
           val v''' = Var(v'_ty,vstr'')   (* new type, new name *)
       in
       Abs(v''', inst theta ((v'',v''')::S) (subst [(v'',v)] body))
       end
       else raise VAR_CLASH clash_var |
    VAR_CLASH _ => raise INST_IMP_ERR "var clash"
    end
```

3.4 Conclusion

In this chapter we examined the operations of the prelogic in detail. What is new?

- We have argued for the importance of the prelogic.
- We have presented the specifications and implementations of the operations.
- We have argued that an *LCF*-style implementation of a logic has good properties as far as the coexistence of extensibility and correctness.

In view of the complexity of the "primitive" algorithms presented here, a simpler alternative should be sought. deBruijn terms [dB72] are touted as providing a superior alternative, since issues of bound variable capture never arise. To bring us full circle with Curry, P.L. Curien [Cur86] has shown that deBruijn terms form a combinatory calculus. We plan to investigate them.

Chapter 4

Grundlagen der α-conversion

In this chapter, we prove the correctness of the algorithm for α-conversion in **hol90**. In all the books about lambda calculus, the authors dispense very quickly with the theory of α-conversion, preferring to get on with more weighty matters. Models of the lambda calculus identify α-convertible terms, so identity is usually taken to be equality modulo α-convertibiblity. Such is also the case for an implementation of the lambda calculus.

We found that the theory of α-conversion is not so easily dispensed with as in the literature. Although the algorithm is simple, its proof takes a lot of space. In this chapter, we go into what seems an excessive amount of detail in the proofs since we want the proofs to be easily translated into **hol90** proofs in the future. The proof divides into three sections, of asymptotically diminishing interest. The first section presents the high level proof of correctness for **aconv**. The second section presents the proof of soundness, linchpin of the high level proof. The third section consists of a large number of support lemmas. The final section of the chapter contains some observations on the proof.

Proofs will be done by structural induction. The proofs are informal, but are intended to be in the operational semantics of Standard ML, as presented in [MTH90] and explained in [Pit90]. Therefore, we proceed by a combination of logical reasoning and "symbolic execution" of the program: we distinguish between a step in the program execution, which will be denoted by a $\rightsquigarrow$, and ordinary equality reasoning, which is denoted by $=$, as usual. We think of $\rightsquigarrow$ as the *single-step computation relation*. The reflexive-transitive closure of $\rightsquigarrow$ is denoted $\overset{*}{\rightsquigarrow}$.

An informal proof is valuable, in that it shows the right way to do the formal verification. The major effort in proving something, in hol90 or with paper and pencil, is finding the right statements to prove. (This is a point in favour of an *LCF*-style theorem prover—the formal proof is very easily derived from a sufficiently detailed informal proof. The same cannot be said for automatic theorem provers, which may need goals with certain computational properties.) As a matter of fact, the proofs showed ways of improving **aconv**—adjusting the algorithm so that the proofs were easier made the algorithm both faster and easier to understand.

In this chapter, we will not distinguish between terms of the typed lambda calculus and Standard ML objects of type **term**. For example, in the statement of the correctness of α-conversion, notice that P and Q to the left of the *iff* are not the same as those on the right. On the left, we are talking about typed lambda calculus terms; on the right we are talking about **terms**. Although we haven't proved it, we assume that there is an isomorphism between the two.

4.1 The high level proof

Our goal is to prove the following proposition:

```
P ≡α Q  iff  acnv P Q [] ⇝* true
```

Proof.

$\boxed{\supset}$

We prove $P \equiv_\alpha Q \supset$ `acnv P Q [] ⇝* true` by induction on the length of the *OSR_proof*.

Base case. This is satisfied by reflexivity.

Induction step. The inductive hypothesis is

$\forall P\ Q.\ OSR\_proof\ P\ Q\ n \supset$ `acnv P Q [] ⇝* true`.

We must show $OSR\_proof\ P\ Q\ (n+1) \supset$ `acnv P Q [] ⇝* true`. Using the inductive hypothesis and the definition of *OSR_proof*, we now need to show `acnv P Q [] ⇝* true` under assumptions $OSR(P, P')$ and `acnv P' Q [] ⇝* true`. This follows from the soundness and transitivity of **aconv**.

$\boxed{\subset}$

Assume `aconv P Q`. Construct the OSR_proof in the following manner:

1. Traverse the two terms in parallel, in left-to-right order. Stop at the first abstraction where `P` differs from `Q`.
2. If there are no differences between terms, stop.
3. Otherwise, we are at an occurrence `occ` with `P/occ` = λ`x.M`. and `Q/occ` = λ`y.N`.
4. Choose a new variable v.
5. Perform `P/occ` := λ`v.[v/x]M` and `Q/occ` := λ`v.[v/y]N`.
6. Go to 1.

(In this proof, the symmetry and transitivity properties are for $\equiv_\alpha$, as in [HS86, lemma 1.17, p. 9].) Understanding the proof may be aided by contemplation of figure 4.1.

Assume we have traversed the loop n times. By construction and transitivity we have OSR_proof (P,P_n) and OSR_proof(Q,Q_n). If $P_n = Q_n$, then we are done, by symmetry. If not, we perform the replacement and continue.

This algorithm terminates, since it always removes one difference between the two terms and the two terms are finite. Because of the traversal strategy, the only differences between terms that are detectable are at abstractions.

∎

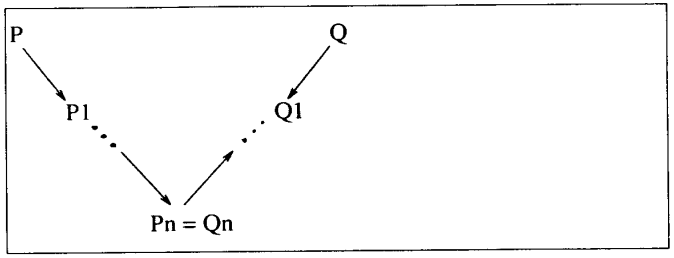


Figure 4.1: Creation of OSR_proof

4.2 Soundness

The proof that if P and Q are in the one-step renaming relation, then aconv P Q $\dot{\rightsquigarrow}$ true is mostly straightforward. However, there is a difficulty having to do with the renaming case for substitution under a λ. This occurs in the proof of the *substitution congruence lemma.*

Lemma. *Soundness.* $\forall P\ Q.\ OSR(P,Q) \supset$ acnv P Q [] $\dot{\rightsquigarrow}$ true.

Proof.

The proof is by induction on the structure of terms P and Q. We note two necessary conditions for there to be a one-step renaming between two terms: they must have the same structure; and they can not be variables or constants (since an OSR requires an occurrence of an abstraction). Therefore, of the sixteen cases, fourteen are vacuously satisfied. We focus on the remaining two:

(Comb, Comb) The inductive hypothesis is

OSR(M1, M2) $\supset$ acnv M1 M2 [] $\dot{\rightsquigarrow}$ true $\wedge$
OSR(N1, N2) $\supset$ acnv N1 N2 [] $\dot{\rightsquigarrow}$ true.

Assume OSR((M1 N1), (M2 N2)). To show:

acnv (M1 N1) (M2 N2) [] $\dot{\rightsquigarrow}$ true.

There are two cases:

1. The occurrence is in the left subtree. Then N1 = N2. This follows from the inductive hypothesis and the *mk_comb identity lemma.*
2. The occurrence is in the right subtree. Then M1 = M2. This follows from the inductive hypothesis and the *mk_comb identity lemma.*

(Abs, Abs) The inductive hypothesis is

OSR(P,Q) $\supset$ acnv P Q [] $\dot{\rightsquigarrow}$ true.

We wish to show

OSR(λv.P, λu.Q) $\supset$ acnv λv.P λu.[u/v]P [] $\dot{\rightsquigarrow}$ true.

There are two cases:

1. The occurrence is in P. This case is solved by the inductive hypothesis and the *congruence lemma.*
2. The occurrence is the root. Expand with the definition of OSR to get

   (λv.P = λw.N $\wedge$ λu.Q = λz.[z/w]N $\wedge$ z $\notin$ FV(N)) $\supset$

   acnv λv.P λu.Q [].

   Using the equalities in the antecedent (and then ignoring them), we have

   z $\notin$ FV(N) $\supset$ acnv λw.N λz.[z/w]N [] .

This is satisfied by the *substitution congruence lemma.*

∎

Consider the abstraction case of an inductive proof, where one must show (for the proposition P to be proved)

$$P(M) \supset P(\lambda v.M)$$

In one case in the next lemma, $P(\lambda v.M)$ involves renaming M to M', which is α-convertible to M, but not identical. Hence the inductive hypothesis can no longer be brought to bear.

We skirt this by first proving the result when no renaming occurs, then by showing that, if renaming occurs, we can translate to an equivalent situation where no renaming occurs. This is summarized in figure 4.2. In the figure, if no renaming occurs, the proof stays on the left hand side of the figure. If renaming does occur, then we conceptually start the proof over again with M', which is α-equivalent to M, and furthermore, will not need renaming.

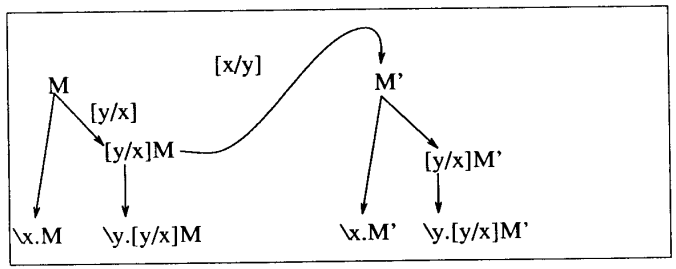


Figure 4.2: Soundness

Lemma. *Substitution Congruence.*
`y ∉ FV(M) ⊃ acnv λx.M λy.[y/x]M [] ⇝̇ true.`
Proof.
There are two cases:

1. No renaming takes place in `[y/x] M`. Then we are done, courtesy of the *no renaming lemma.*
2. Renaming takes place. In that case, we transform `M` into an α-equivalent term `M'` in which renaming will not take place. Let `M' = [x/y][y/x]M`. By the *α-identity lemma* and congruence, we have `acnv λx.M λx.M' [] ⇝̇ true`. By the *substitution lemma* and identity, we have `λy.[y/x]M = λy.[y/x]M'`. By the *fixed point lemma* and the *no renaming lemma*, we have `acnv λx.M' λy.[y/x]M' [] ⇝̇ true`. By transitivity, we have `acnv λx.M λy.[y/x]M`.

∎

In the next lemma, it is necessary to note that

`[x/y][y/x]M = M ⊃ y ∉FV(M).`

Lemma. *No renaming.*
`([x/y][y/x]M = M) ⊃ acnv λx.M λy.[y/x]M [] ⇝̇ true`
The proof proceeds by induction on `M`.

Var. `[x/y][y/x]v = v ⊃ acnv λx.v λy.[y/x]v []`. There are two cases:

`x = v.` $\rightsquigarrow$ `acnv x y [(x,y)]`

$\dot{\rightsquigarrow}$ `true`

`x ≠ v.` $\rightsquigarrow$ `acnv v v [(x,y)]`

$\dot{\rightsquigarrow}$ `true`, since `y ≠ v`, by the assumption.

Const. acnv λx.c λy.[y/x]c []

$\rightsquigarrow$ acnv c c [(x,y)]

$\rightsquigarrow$ c = c

$\rightsquigarrow$ true

Comb. We wish to show

y ∉ FV(M N) ⊃ acnv λx.(M N) λy.[y/x](M N) $\dot{\rightsquigarrow}$ true

The inductive hypothesis is

y ∉ FV(M) ⊃ acnv λx.M λy.[y/x]M $\dot{\rightsquigarrow}$ true ∧
y ∉ FV(N) ⊃ acnv λx.N λy.[y/x]N $\dot{\rightsquigarrow}$ true

y ∉ FV(M N) implies y ∉ FV(M) ∧ y ∉ FV(M), so the antecedents of the induction hypotheses are satisfied, so we have (by $\rightsquigarrow$)

acnv M [y/x]M [(x,y)] $\dot{\rightsquigarrow}$ true ∧
acnv N [y/x]N [(x,y)] $\dot{\rightsquigarrow}$ true

Now,

acnv λx.(M N) λy.[y/x](M N) []

$\rightsquigarrow$ acnv (M N) [y/x](M N) [(x,y)]

$\rightsquigarrow$ acnv M [y/x]M [(x,y)] andalso

acnv N [y/x]N [(x,y)]

$\dot{\rightsquigarrow}$ true andalso true, by the inductive hypotheses

$\rightsquigarrow$ true

Abs. The inductive hypothesis is:

y ∉ FV(M) ⊃ acnv λx.M λy.[y/x]M $\dot{\rightsquigarrow}$ true

Assume y ∉ FV(λs.M). To show:

aconv λx λs.M λy [y/x]λs.M

There are three cases, based on the definition of substitution into an abstraction:

x = s. aconv (λx λx.M) (λy [y/x]λx.M)

= acnv (λx λx.M) (λy λx.M) []

$\dot{\rightsquigarrow}$ true, by reflexivity and the *no free vars lemma*

x ≠ s ∧ (s ≠ y ∨ x ∉ FV(M)). There are two choices here, based on the disjunction.

x ∉ FV(M). This is exactly the same as the previous case.

s ≠ y. Then aconv λx λs.M λy [y/x]λs.M

= acnv λx λs.M λy λs.[y/x]M []

$\dot{\rightsquigarrow}$ true, by IH and the *swap lemma.*

x ≠ s ∧ s = y ∧ x ∈FV(M). Our assumption rules this clause out.

∎

Lemma. *α-identity.*
y ∉ FV(M) ⊃ acnv M [x/y][y/x]M [] $\dot{\rightsquigarrow}$ true.
The proof proceeds by induction on the structure of M.

Var. `acnv v [x/y][y/x]v []`

There are two cases:

`v` = `x`. Then, `[x/y][y/x]x = x`, and `acnv x x []` $\stackrel{*}{\rightsquigarrow}$ `true`.

`v` $\neq$ `x`. Then `[x/y][y/x]v = v`, since `y` $\neq$ `v`. Hence `acnv v v []` $\stackrel{*}{\rightsquigarrow}$ `true`.

Const. As usual.

Comb. As usual.

Abs. There are three cases:

No replacement. `acnv` λ`x.M [x/y][y/x]`λ`x.M []`

`[x/y][y/x]` λ`x.M = [x/y]`λ`x.M`. Since `y` $\notin$`FV(`λ`x.M)`,

`[x/y]`λ`x.M =` λ`x.M`, so this case is proven by reflexivity.

Ordinary replacement. `acnv` λ`v.M [x/y][y/x]`λ`v.M []`

This is true, by the inductive hypothesis and the congruence lemma.

Renaming, then ordinary replacement.

`acnv` λ`y.M [x/y][y/x]`λ`y.M []`

`=` `acnv` λ`y.M` λ`z.[x/y][y/x][z/y]M []`

`=` `acnv` λ`y.M` λ`z.[z/y]M []`, since `y` $\notin$ `FV([z/y]M)`, and by [HS86, lemma 1.15b, p. 8]. The *no renaming lemma* finishes this case.

∎

Lemma. *Substitution.* Let `M' = [x/y][y/x]M`.

Then `y` $\notin$`FV(M)` $\supset$ `[y/x]M = [y/x]M'`

The proof is by induction on the structure of `M`.

Var. `[y/x]v = [y/x][x/y][y/x]v`. There are two cases:

`v` = `x`. Then `[y/x]x = y = [y/x][x/y]y`.

`v` $\neq$`x`. There are two cases:

`v` = `y`. Then `[y/x][x/y][y/x]y = [y/x][x/y]y = y = [y/x]y`.

`v` $\neq$`y`. Then `[y/x][x/y][y/x]v = [y/x][x/y]v = v = [y/x]v`.

Const. As usual

Comb. As usual.

Abs. There are three cases:

No replacement.

$$\begin{aligned}
\texttt{[y/x]}\ \lambda\texttt{x.M} &= \texttt{[y/x][x/y][y/x]}\lambda\texttt{x.M} \\
\lambda\texttt{x.M} &= \texttt{[y/x][x/y]}\lambda\texttt{x.M} \\
&= \texttt{[y/x]}\lambda\texttt{x.M}, \text{ since } y \notin FV(M) \\
&= \lambda\texttt{x.M}
\end{aligned}$$

Ordinary replacement.

$$\begin{aligned}
\texttt{[y/x]}\ \lambda\texttt{v.M} &= \texttt{[y/x][x/y][y/x]}\lambda\texttt{v.M} \\
\lambda\texttt{v.[y/x]M} &= \texttt{[y/x][x/y]}\lambda\texttt{v[y/x].M} \\
&= \lambda\texttt{v.[y/x][x/y][y/x]M} \\
&= \lambda\texttt{v.[y/x]M}, \text{ by inductive hypothesis and congruence}
\end{aligned}$$

Renaming, then replacement.

$$\begin{aligned}
\texttt{[y/x]}\ \lambda\texttt{y.M} &= \texttt{[y/x][x/y][y/x]}\lambda\texttt{y.M} \\
\lambda\texttt{z.[y/x][z/y]M} &= \texttt{[y/x][x/y]}\lambda\texttt{z.[y/x][z/y]M} \\
&= \lambda\texttt{z.[y/x][z/y]M}, \text{ by [HS86, lemma 1.15b, p. 8].}
\end{aligned}$$

∎

We require a fixed point result for substitution: an identity substitution applied to a term may result in some internal renaming, but a re-application of the identity substitution will not result in any renaming.

Lemma. There is no free occurrence of `x` in the scope of a λ`y` in `[x/y][y/x]M`.

Proof. If there is such an `x` in `M`, the λ`y` would be renamed to a λ`z`, by the definition of substitution.

∎

Corollary. *Fixed point lemma.* Let `M' = [x/y][y/x]M`.

Then `[x/y][y/x]M' = M'`.

Proof. This is a consequence of [lemma 1.15b, p.8,Hindley and Seldin], which has an overly strong assumption. The previous lemma is more precise in avoiding renaming.

∎

4.3 Properties of aconv

We will prove that `aconv` is an equivalence relation. Also, we will prove some extra properties needed to make the top level proof go through.

To begin, we require the following simple facts about the index routines. In the following, the `X` in `X_index` may be replaced by either `left` or `right`. `Sxx` will be used as an abbreviation for `S` with the list `[(x,x)]` appended to it.

1. `(X_index v S` $\stackrel{\bullet}{\rightsquigarrow}$ `SOME n)` $\supset$ `(X_index v Sxx` $\stackrel{\bullet}{\rightsquigarrow}$ `SOME n)`.
2. `(X_index v S` $\stackrel{\bullet}{\rightsquigarrow}$ `NONE)` $\supset$ `(X_index v Svv` $\stackrel{\bullet}{\rightsquigarrow}$ `SOME |S|)`.
3. ($v \neq x \wedge$ `X_index v S` $\stackrel{\bullet}{\rightsquigarrow}$ `NONE)` $\supset$ `(X_index v Sxx` $\stackrel{\bullet}{\rightsquigarrow}$ `NONE)`

Lemma. `acnv P Q S = acnv P Q Sxx`.

Proof. By induction on terms P and Q This gives 16 cases, of which only 4 are of interest.

(Var,Var). `acnv v1 v2 S = acnv v1 v2 Sxx` .

$\boxed{\supset}$. Assume `acnv v1 v2 S` $\stackrel{\bullet}{\rightsquigarrow}$ `true`.

There are two cases of `(left_index v1 S, right_index v2 S)`:

1. $\stackrel{\bullet}{\rightsquigarrow}$ `(SOME n, SOME n)`. So

   `(left_index v1 Sxx,right_index v2 Sxx)`

   $\stackrel{\bullet}{\rightsquigarrow}$ `(SOME n, SOME n)`, by simple fact 1

   $\rightsquigarrow$ `true`.

2. $\stackrel{\bullet}{\rightsquigarrow}$ `(NONE,NONE)`. So $v1 = v2$. There are two cases:

   $\underline{v1 = x}$. `acnv x x Sxx` $\stackrel{\bullet}{\rightsquigarrow}$ `true` follows from simple fact 2.

   $\underline{v1 \neq x}$. `acnv v1 v1 Sxx` $\stackrel{\bullet}{\rightsquigarrow}$ `true` follows from simple fact 3.

$\boxed{\subset}$. Assume `acnv v1 v2 Sxx` $\stackrel{\bullet}{\rightsquigarrow}$ `true`. Then there are two cases for `(left_index v1 Sxx, right_index v2 Sxx)`:

1. $\stackrel{\bullet}{\rightsquigarrow}$ `(SOME n,SOME n)`. There are two cases for n:

   $\underline{n < |S|}$. Then `(left_index v1 S,right_index v2 S)` $\stackrel{\bullet}{\rightsquigarrow}$ `(SOME n,SOME n)`.

   $\underline{n = S}$. Then $v1 = v2 = x$. So `(left_index x S,right_index x S)`

   $\stackrel{\bullet}{\rightsquigarrow}$ `(NONE,NONE)`

   $\stackrel{\bullet}{\rightsquigarrow}$ `true`.

2. $\stackrel{\cdot}{\rightsquigarrow}$ `(NONE,NONE)`. So

   `(left_index v1 S, right_index v2 S)`

   $\stackrel{\cdot}{\rightsquigarrow}$ `(NONE,NONE)`

   $\stackrel{\cdot}{\rightsquigarrow}$ `true`, since $v1 = v2$.

(Const, Const). By reduction.

(Comb,Comb)

`acnv (M1 N1) (M2 N2) S`	`acnv (M1 N1) (M2 N2) Sxx`
$\stackrel{\cdot}{\rightsquigarrow}$	$\stackrel{\cdot}{\rightsquigarrow}$
`acnv M1 M2 S andalso`	`acnv M1 M2 Sxx andalso`
`acnv N1 N2 S`	`acnv N1 N2 Sxx`

Then the IH finishes this case off.

(Abs, Abs) Follows immediately from $\stackrel{\cdot}{\rightsquigarrow}$ and the IH.

∎

Corollary. *Congruence.* `aconv M N` $\supset$ `aconv` λ`v.M` λ`v.N`.

∎

Lemma. *Reflexivity.* `acnv M M []` $\stackrel{\cdot}{\rightsquigarrow}$ `true`.

The proof proceeds by induction on the structure of M.

Var. `acnv v v []`

$\rightsquigarrow$ `(left_index v [], right_index v [])`

$\rightsquigarrow$ `(NONE,NONE)`

$\rightsquigarrow$ `v = v`

$\rightsquigarrow$ `true`

Const. `acnv c c []`

$\rightsquigarrow$ `c = c`

$\rightsquigarrow$ `true`

Comb. `acnv (M N) (M N) []`

$\rightsquigarrow$ `acnv M M [] andalso acnv N N []`

$\stackrel{\cdot}{\rightsquigarrow}$ `true`, by IH

Abs. `acnv` λ`v.M` λ`v.M []`

$\stackrel{\cdot}{\rightsquigarrow}$ `true`, by IH and congruence

∎

To prove the symmetric property for **aconv**, we need to define an operation that "flips" all the bindings in a scope:

$$[]^{-1} = [] \quad \wedge$$

$$((a, b) :: S)^{-1} = (b, a) :: S^{-1}$$

Lemma. `acnv M N S` $\stackrel{\cdot}{\rightsquigarrow}$ `true` $\supset$ `acnv M N` S^{-1} $\stackrel{\cdot}{\rightsquigarrow}$ `true`.

The proof proceeds by induction on the structure of M and N.

(Var, Var). Induct on S:

base case. This is a tautology.

inductive case. `acnv v1 v2 (u1,u2)::S` $\dot{\rightsquigarrow}$ `true` $\supset$ `acnv v2 v1 (u2,u1)::S`$^{-1}$ $\dot{\rightsquigarrow}$ `true`. Assume `acnv v1 v2 (u1,u2)::S` $\dot{\rightsquigarrow}$ `true`. There are two cases:

1. `v1 = u1` and `v2 = u2`. Obvious.
2. `acnv v1 v2 S` $\dot{\rightsquigarrow}$ `true`. By the inductive hypothesis.

(Const, Const). `acnv c1 c2 S` $\dot{\rightsquigarrow}$ `true` $\supset$ `acnv c1 c2 S`$^{-1}$ $\dot{\rightsquigarrow}$ `true`.

Assume `acnv c1 c2 S` $\dot{\rightsquigarrow}$ `true`. Hence `c1 = c2`.

`acnv c1 c1 S`$^{-1}$

$\rightsquigarrow$ `c1 = c1`

$\rightsquigarrow$ `true`

(Comb, Comb). `acnv (M1 M2) (N1 N2) S` $\dot{\rightsquigarrow}$ `true` $\supset$ `acnv (N1 N2) (M1 M2) S`$^{-1}$ $\dot{\rightsquigarrow}$ `true`.

`acnv (N1 N2) (M1 M2) S`$^{-1}$

$\rightsquigarrow$ `acnv N1 M1 S`$^{-1}$ `andalso acnv N2 M2 S`$^{-1}$

$\dot{\rightsquigarrow}$ `true`, by the inductive hypothesis.

(Abs, Abs). `acnv` λ`v.M` λ`u.N S` $\dot{\rightsquigarrow}$ `true` $\supset$ `acnv` λ`u.N` λ`v.M S`$^{-1}$ $\dot{\rightsquigarrow}$ `true`.

`acnv` λ`u.N` λ`v.M S`$^{-1}$

$\rightsquigarrow$ `acnv N M (u,v)::S`$^{-1}$

$\dot{\rightsquigarrow}$ `true`, by the inductive hypothesis and the assumption, after reduction.

∎

Corollary. *Symmetry.* `aconv M N` $\supset$ `aconv N M`.

∎

Lemma. *Transitivity.* (`acnv M N []` $\dot{\rightsquigarrow}$ `true` $\wedge$ `acnv N P []` $\dot{\rightsquigarrow}$ `true`) $\supset$ `acnv M P []` $\dot{\rightsquigarrow}$ `true`.

Proved by a straightforward but tedious triple induction.

∎

Lemma. *mk_comb identity.* (`acnv M1 M2 []` $\dot{\rightsquigarrow}$ `true` $\supset$ `acnv (M1 N) (M2 N) []` $\dot{\rightsquigarrow}$ `true` $\wedge$ `acnv (N M1) (N M2) []` $\dot{\rightsquigarrow}$ `true`

Proof. A straightforward induction on `N`.

∎

In the next proof, `Suvxy` denotes a list `S` with the list `[(u,v), (x,y)]` appended. Likewise, `Sxyuv` denotes `S` with the list `[(x,y),(u,v)]` tacked on.

Lemma. $\forall$`M N S`.($x = u \Leftrightarrow y = v$) $\supset$ `acnv M N Suvxy = acnv M N Sxyuv`.

Proof. By induction on the structure of M and N.

(Var, Var). This case is proved by induction on S:

base case. `acnv w1 w2 [(u,v),(x,y)] = acnv w1 w2 [(x,y),(u,v)]`.

Nine cases ensue, based on the values that `w1` and `w2` may take:

	`w2 = v`	`w2 = y`	`w2` $\neq$ `v` $\wedge$ `w2` $\neq$ `y`
`w1 = u`	`(true,true)`	`(false,false)`	`(false,false)`
`w1 = x`	`(false,false)`	`(true,true)`	`(false,false)`
`w1` $\neq$ `u` $\wedge$ `w2` $\neq$ `x`	`(false,false)`	`(false,false)`	`(w1 = w2, w1 = w2)`

inductive case. `acnv w1 w2 (z1,z2)::Suvxy = acnv w1 w2 (z1,z2)::Sxyuv`

There are four possibilities:

1. `w1 = z1 ∧ w2 = z2`. This produces `true,true`.
2. `w1 = z1 ∧ w2 ≠ z2`. This produces `false,false`.
3. `w1 ≠ z1 ∧ w2 = z2`. This produces `false,false`.
4. `w1 ≠ z1 ∧ w2 ≠ z2`. Solved by the inductive hypothesis.

(Const, Const). `acnv c1 c2 Suvxy = acnv c1 c2 Sxyuv`.

As usual, constants are only α-convertible to themselves.

(Comb, Comb). `acnv (M1 M2) (N1 N2) Suvxy = acnv (M1 M2) (N1 N2) Sxyuv`.

This case is solved by reduction on both sides of the equality and appeal to the inductive hypothesis.

(Abs, Abs). `acnv λw1.M λw2.N Suvxy = acnv λw1.M λw2.N Sxyuv`.

By reduction we have

`acnv M N (w1,w2)::Suvxy = acnv M N (w1,w2)::Sxyuv`,

which the inductive hypothesis solves.

∎

Corollary *Swap lemma.*

$(x = u \Leftrightarrow y = v) \supset$ `aconv λxu.M λyv.N = aconv λux.M λvy.N` .

∎

In the following proof, `Sxy` denotes a scope `S` with the pair `(x,y)` appended.

Lemma. `∀M N S. (x ∉ FV(M) ∧ y ∉ FV(N) ∧ acnv M N S ~>* true) ⊃ acnv λx.M λy.N Sxy ~>* true`.

Proof. By induction on the structure of `M` and `N`.

(Var,Var). Induction on `S`.

(Const, Const). Trivial

(Comb, Comb). By reduction and the inductive hypothesis.

(Abs, Abs). By reduction and the inductive hypothesis.

Corollary *No Free Vars lemma.* `(x ∉ FV(M) ∧ y ∉ FV(N) ∧ aconv M N) ⊃ aconv λx.M λy.N Exy`.

∎

By virtue of the reflexive, symmetric, and transitive properties, `aconv` divides *HOL* terms into equivalence classes.

4.4 Conclusion

The above verification is not entirely satisfactory. Although most of the goals aren't especially difficult, the size of the goals is nearly unmanageable, and if we did a fully formal proof using the operational semantics of SML, they *would* be unmanageable. A theorem proving environment would certainly ease the "bookkeeping" of the proof. Also, an impartial and inexhaustible observer, such as a proof checker, would ensure that I hadn't fooled myself or made a mistake at some step in a proof.

Chapter 5

Rules of Inference for Equational Deduction

Equalities are ubiquitous in proof; however, undirected equalities are notoriously difficult to deal with, since a combinatorial explosion of possibilities results. For this reason, *rewriting* (the most common use of equality in *HOL* proofs), is usually performed under user control. The investigation of rewriting forms a separate research field, that of *Term Rewriting Systems (TRSs)*[HO80], which lies at the intersection between proof and computation.

Loosely, a TRS is a set of pairs of patterns (p_i, q_i) used to reduce a supplied term tm. Reduction is done by replacing an instance of p_i in tm by a corresponding instance of q_i. Reduction continues until no instance of a p_i can be found. As an example, the following Term Rewriting System [Der87] plays the "Dutch National Flag" game: given any arrangement of white, red, and blue tokens, the rules when applied in any order will reduce the sequence of tokens to one where all the white tokens lie between all the red tokens on the left and the blue tokens on the right.

```
white.red → red.white
blue.red → red.blue
blue.white → white.blue
```

Term Rewriting Systems are a very abstract model of computation, with universal power. For example, Curry's Combinatory Logic is a TRS. Term Rewriting Systems are useful in the study of many subjects in Computer Science, for example, the operational semantics of functional programming languages [Tur85], the optimization of machine code via so-called "peephole" optimization, program transformation, and verification.

In this chapter, we investigate situations in which equational inference can be replaced by directed rewriting. The central work in the field is by Knuth and Bendix [KB70], in which they studied ways of "completing" equational axiom systems. They present a procedure that takes a set of equations E and returns an oriented set of equations R that yields the same equality as the one generated by E. Rewriting with R can then be used to decide if two terms are deducibly equal in equational logic, using E as axioms. Thus, when the procedure succeeds, it provides a solution to the *validity problem* for E.

We have implemented Knuth-Bendix completion (henceforth completion) as a derived rule of inference in **hol90**. Since research into completion has, except for some very recent work [Nip90], focused on first order terms, we have identified a set of *HOL* terms isomorphic to the first order terms, and defined completion only for them. Thus we have identified an equational logic fragment of *HOL*.

We emphasize that completion and AC rewriting (described in the next chapter) are derived rules of inference. Most other logic implementations incorporate them as *ad hoc* procedures. Going to the trouble to do inference rather than computation does slow down our implementations of these procedures somewhat, but we feel that the resultant loss of speed is not significant, especially in light of the confidence that we are allowed to retain in our system. For theorem provers that do not (explicitly or implicitly) follow the encapsulation route, every new proof procedure added to the system can only decrease ones confidence in the system, while those who follow the encapsulation path can maintain their confidence at a constant level. Since the amount encapsulated can be small, that level can be high.

It is noteworthy that Term Rewriting Theory has not been utilized in the implementation of *HOL* rewriting [Pau83]; this is due to the focus on unquantified first order terms in TRS, which does not work in the quantified and higher order setting provided by *HOL*.

In the first section of this chapter, we give a description of equational logic. This is largely taken from an excellent introductory paper by Klop and Middeldorp [KM88]. In the second section of this chapter, we define Term Rewriting Systems and some of their important properties, chiefly those that have to hold if we are to use a TRS as a decision procedure for validity. In the third section, we present Knuth-Bendix completion as a procedure. In the fourth section, we show how completion may be implemented as a derived rule of inference. The last section of the chapter is about possible applications in hol90 of these procedures and of some others in TRS theory.

Completion depends essentially on the *unification* of terms. The theory of first order substitution and unification is described in detail in Appendix A.

5.1 Equational logic

The set of first order terms $\mathcal{F}(\Sigma \cup \mathcal{V})$ is inductively formed from Σ, a set of aritied constants, and $\mathcal{V}$ a countably infinite set of variables:

- if x is in $\mathcal{V}$, then x is in $\mathcal{F}(\Sigma \cup \mathcal{V})$;
- if f of arity $n \geq 0$ is in Σ, and $t_1, \ldots, t_n$ are in $\mathcal{F}(\Sigma \cup \mathcal{V})$, then $f(t_1, \ldots, t_n)$ is in $\mathcal{F}(\Sigma \cup \mathcal{V})$.

In the rest of this section we abbreviate $\mathcal{F}(\Sigma \cup \mathcal{V})$ by $\mathcal{F}$. We may define $\mathcal{F}$ in SML as follows:

```
datatype F = Fvar of string |
             Fconst of (string*int) |
             Fapp of (F*(F list))
```

F must, of course, be abstract so that its objects are well-formed according to Σ.

If l and r are in $\mathcal{F}$ then $l = r$ is an *equation* in $\mathcal{F}$. Equations are the only formulas, and are implicitly universally quantified.

A substitution θ is a function from $\mathcal{F}$ to $\mathcal{F}$ satisfying

$$\theta(f(t_1, \ldots, t_n)) = f(\theta(t_1), \ldots, \theta(t_n))$$

for every function symbol f. Therefore, the behaviour of θ depends on how it maps variables.

The deductive system for equational logic consists of a set of equations E and the following inference rules:

$$\frac{t \in \mathcal{F}}{E \vdash t = t} \qquad \frac{l = r \in E}{E \vdash l = r}$$

$$\frac{E \vdash t_1 = t_2, \quad E \vdash t_2 = t_3}{E \vdash t_1 = t_3} \qquad \frac{E \vdash l = r}{E \vdash r = l}$$

$$\frac{E \vdash l = r}{E \vdash \theta(l) = \theta(r)} \qquad \frac{E \vdash l_1 = r_1, \ldots, E \vdash l_n = r_n}{E \vdash f(l_1, \ldots, l_n) = f(r_1, \ldots, r_n)}$$

A Σ-algebra $\mathcal{A}$ is a set A plus an n-ary function $f_n : A^n \rightarrow A$ for every constant f of arity n in Σ. An equation $t_1 = t_2$, where t_1 and t_2 are in $\mathcal{F}$ is assigned a "meaning" in $\mathcal{A}$ by interpreting the constants in t_1 and t_2 as functions in $\mathcal{A}$. If the interpretations of t_1 and t_2 coincide for every variable assignment, we write $\mathcal{A} \models t_1 = t_2$ and say that $\mathcal{A}$ *models* $t_1 = t_2$, or that $t_1 = t_2$ is *valid* in $\mathcal{A}$. We extend this terminology to sets of equations: $\mathcal{A} \models E$ if every equation in E is valid in $\mathcal{A}$. If every Σ-algebra that models E_1 is also a model of E_2, we write $E_1 \models E_2$.

The *validity problem* for a set of equations E over $\mathcal{F}$ is: given t_1 and t_2, both from $\mathcal{F}$, decide whether or not $E \models t_1 = t_2$. The completeness theorem of Birkhoff [Bir35] allows us to focus on provability:

Theorem. Let E be a set of equations over $\mathcal{F}$. For all t_1, t_2 in $\mathcal{F}$,

$$E \models t_1 = t_2 \Leftrightarrow E \vdash t_1 = t_2. \blacksquare$$

The results of the next section give sufficient conditions for efficient equational deduction that decides if $E \vdash t_1 = t_2$.

5.2 Term Rewriting Systems

A TRS comprises a set of rewrite rules. If $s = t$ is an equation in $\mathcal{F}$, s is not a variable, and the variables of t are a subset of the variables of s, then $s = t$ is allowed to be a rewrite rule, and is usually written as $s \rightarrow t$. A *context* is a term that contains a single occurrence of $\Box$, a special symbol not found in $\Sigma \cup \mathcal{V}$. A context is denoted $C[\,]$. If $C[\,]$ is a context and tm is a term, then $C[tm]$ is the term that results from replacing $\Box$ by tm. A rule has a set of *instances*, determined by the set of substitutions. In an instance $\theta s \rightarrow \theta t$, θs is known as the *redex* and θt is known as the *contractum* [1]. Rewriting a term involves replacing an occurrence of a redex by its contractum. The resulting term is called a *reduct*. The *(one-step) reduction relation* of a rewrite rule r, $\rightarrow_r$ is $\{(tm, tm') : tm' \text{ is a reduct of } tm \text{ using } r\}$. The reflexive transitive closure of $\rightarrow_r$ is denoted by $\stackrel{*}{\rightarrow}_r$. We use the same terminology when speaking about a set of rewrite rules, and often we will drop the subscript from $\stackrel{*}{\rightarrow}_R$ when it may be understood from context. The symmetric closure of $\rightarrow$ is denoted $\leftrightarrow$. The reflexive and transitive closure of $\leftrightarrow$ is denoted $\stackrel{*}{\leftrightarrow}$.

An *irreducible* term, also called a *normal form*, has no occurrences of $\rightarrow_R$ redexes. A term tm has a normal form tm' if $tm \stackrel{*}{\rightarrow} tm'$ and tm' is irreducible. There are many interesting properties that a TRS may have. Among the most studied of these are *termination* and *confluence*. (Confluence is also known as the Church-Rosser property.) A set of rules is terminating if all reductions for all terms eventually stop, *i.e.*, every reduction sequence is finite. If R terminates, then one will not have to wait forever when reducing a term by R.

A relation $\rightarrow$ is confluent if and only if it has the following property:

$$\forall\ x\ y\ w.\ (w \stackrel{*}{\rightarrow} x \wedge w \stackrel{*}{\rightarrow} y) \supset (\exists\ z.\ x \stackrel{*}{\rightarrow} z \wedge y \stackrel{*}{\rightarrow} z)$$

[1] The choice of *redex* and *contractum* is perhaps unfortunate, as it provoked ribald comments from several of my proof readers.

A relation is *locally confluent* if and only if it has the following property:

$$\forall\ x\ y\ w.\ (w \rightarrow x \wedge w \rightarrow y) \supset (\exists\ z.\ x \stackrel{*}{\rightarrow} z \wedge y \stackrel{*}{\rightarrow} z)$$

These definitions are rendered in Figure 5.1.

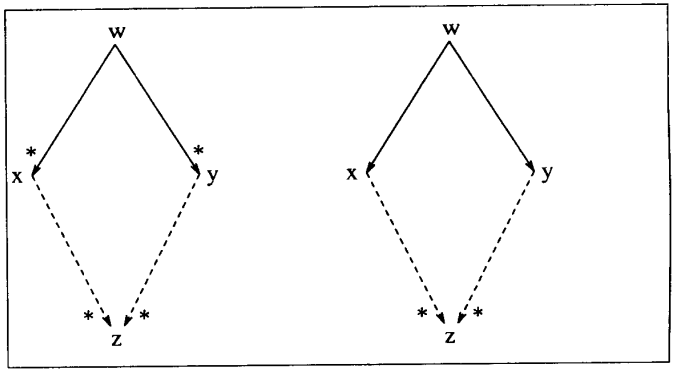


Figure 5.1: Confluence and Local Confluence

Theorem [Bir35]. $E \vdash t_1 = t_2 \Leftrightarrow t_1 \stackrel{*}{\leftrightarrow} t_2$. ∎

Theorem [Hue80]. If R is confluent, then it has unique normal forms. ∎

Theorem [KM88]. If R is confluent, then $t_1 \stackrel{*}{\leftrightarrow} t_2$ if and only if $\exists z. t_1 \stackrel{*}{\rightarrow} z \wedge t_2 \stackrel{*}{\rightarrow} z$. Hence, if R is terminating, $t_1 \stackrel{*}{\leftrightarrow} t_2$ if and only if the normal forms of t_1 and t_2 coincide. ∎

Theorem. Suppose $s \stackrel{*}{\underset{R}{\leftrightarrow}} t$ if and only if $s \stackrel{*}{\underset{E}{\leftrightarrow}} t$. If R is confluent and terminating, we have a decision procedure for deciding validity: reduce s and t to their irreducible

other way of reducing the ambiguity is with $rule_2$, and is a bit more complicated: the second half of the critical pair is the replacement of the occurrence of θl_2 in the ambiguity by θr_2. Notice that this is easily mapped to the top half of the diagram for local confluence.

Example. Let r_1 be $f(x, g(x,a)) \rightarrow h(x)$ and r_2 be $g(b,y) \rightarrow k(y)$. Then the ambiguous term is $f(b, g(b,a))$ and the critical pair is $< h(b),\ f(b, k(a)) >$.

The completion algorithm alternates the production of critical pairs with the reduction of critical pairs by rewrite rules. If a critical pair does not get reduced to identity by the current rule set, then the term ordering is used to make the reduced critical pair into a rewrite rule. If ordering is impossible, the completion fails.

```
Input : - a set of first order equations E
        - ≻, a reduction ordering for terms
        - R is initially empty
Output : R, a confluent, terminating TRS equivalent to E

KB □ R = R  |
KB ((s = t)::E) R =
  let (s', t') = the full reduction of s and t by R
  in
  if (s' = t')
  then KB E R
  else let (l → r) = order ≻ (s', t')
       let critical_pairs = {(P,Q) : (P,Q) is critical pair between R & l → r}
       in
       KB (E ∪ critical_pairs - {s = t})
          ({l → r}::R)
```

The procedure can have one of three outcomes: successful termination, inability to order a critical pair into a rule, and nontermination. In the first case, we have our decision procedure. In the second case, we must start from the beginning with an ordering that will order the two recalcitrant terms (as well as all the others, of course). In the third case, the procedure computes an infinite TRS.

There are more efficient procedures for completion. We have implemented one which keeps the rules in R inter-reduced for efficiency [Hue81].

Lately it has become fashionable to present completion as a deductive system, thus separating the inference from control. This allows easy proofs of correctness for a wide variety of optimized completion procedures [BDH86]. The essence of a proof of correctness for the completion procedure is to show that, every step i through the loop,

$$(\overset{*}{\underset{E}{\leftrightarrow}})_i \cup (\overset{*}{\underset{R}{\leftrightarrow}})_i \ = \ (\overset{*}{\underset{E}{\leftrightarrow}})_0.$$

Completion provides a semi-decision procedure for equality [Hue81]. This is the basis of Term Rewriting theorem proving, initiated by Hsiang[HD83], who discovered a way to use using the completion procedure to do refutation theorem proving in first order logic.

5.4 Completion as a derived rule of inference

The *LCF* approach to proof enforces a rigorous and clarifying distinction between theorems and meta-theorems: meta-theorems are not directly usable. For example, in an *LCF*-style implementation of natural deduction propositional logic, we would not be able to directly use the truth table method—the type system of ML enforces adherence to natural deduction. We might use the truth table method to give us the *knowledge* that a formula is *deducible* by natural deduction, but we wouldn't be allowed to simply assert that knowledge as a theorem: we would have to use the devices of the formal system to prove the theorem.

To implement a theorem proving procedure in HOL, therefore, requires that one distinguish between what tasks must be done by inference and what may be done by computation. Hence we draw a distinction between being "inside" the logic, *i.e.*, using inference, and being "outside" the logic, *i.e.*, using other computation.

HOL implements rewriting inside the logic, as a derived rule of inference [Pau83]. This implies that rewrite rules, the end product of a completion, must be theorems of

inside — inference	*outside — computation*
rewriting by a set of rewrite rules	unification
computing critical pairs	term ordering
renaming variables apart	matching
orienting an equation to a rule	deciding if a term is first order

the logic, which in turn implies that every critical pair must be a theorem. Therefore, critical pairs must be obtained by inference. This reasoning gives us the breakdown illustrated in the figure.

5.4.1 The first order restriction on terms

We define the "essentially first order terms" of *HOL*. Our aim is to describe single-sorted first order terms. The main idea in this is to extend the notion of arity to types.

Definition. A *curried type of sort* γ is a function type $\gamma \to \gamma'$ where γ' is either γ or a curried type of sort γ. The *width* of $\gamma_1 \to \gamma_2 \to \ldots \to \gamma_n$, a curried type of sort γ, is n.

Definition. The set of first order terms of well-formed type $\gamma, \mathcal{F}_\gamma$, is the smallest subset of **term** ($Terms_{\Sigma_c}$, the *HOL* terms defined in Chapter Two) formed by the following rules:

variables: if $v : \gamma \in$ **term**, then $v : \gamma$ is in $\mathcal{F}_\gamma$. Note that there is a countably infinite set of variables at each well-formed type.

constants: if $c : \gamma \in$ **term**, then $c : \gamma$ is in $\mathcal{F}_\gamma$

combinations: if combination $t : \gamma \in$ **term**, then, since we wish to exclude partial applications, we strip $t = (\ldots(f : \delta\ t_1)\ \ldots\ t_m)$ to f and an argument list

$[t_1, \ldots, t_m]$. If each member of the argument list is in $\mathcal{F}_\gamma$, f is a constant, and δ is the curried type of sort γ of width $m+1$, then t is in $\mathcal{F}_\gamma$.

abstractions: are not allowed.

Definition. A *first order equality theorem of type* γ is a (possibly universally quantified) theorem $\Gamma \vdash t_1 = t_2$ where t_1 and t_2 are both members of $\mathcal{F}_\gamma$. A *homogeneous list of first order equality theorems* $[\Gamma_1 \vdash (l_1 : \gamma) = r_1; \ldots; \Gamma_n \vdash l_n = r_n]$ is a list of first order equality theorems, all of type γ.

Proposition. For any $\gamma \in \tau$ the first order matching, unification, and term ordering algorithms retain their properties in $\mathcal{F}_\gamma$, since there is an isomorphism between $\mathcal{F}$ and $\mathcal{F}_\gamma$.

Proof.

Σ for $\mathcal{F}$ is determined by $\mathcal{F}_\gamma$:

- If $c : \gamma \in \mathcal{F}_\gamma$, then $(c, 0) \in \Sigma$,
- If $(\ldots(f : \delta\ t_1)\ \ldots\ t_n) \in \mathcal{F}_\gamma$, and the width of δ is $n+1$, then $(f, n) \in \Sigma$.

To map from $\mathcal{F}$ to $\mathcal{F}_\gamma$ we use $\oplus$

- $\oplus\ \gamma\ (Fvar\ v) = Var(\gamma, v)$
- $\oplus\ \gamma\ (Fconst\ c) = Const(\gamma, v)$
- $\oplus\ \gamma\ (Fapp(Fconst(f, n), [t_1, \ldots, t_n])) = (\ldots(f : \delta\ (\oplus\ \gamma)(t_1))\ \ldots\ (\oplus\ \gamma)(t_n))$, where δ is the curried type of sort γ of width $n+1$.

To map from $\mathcal{F}_\gamma$ we use $\ominus$:

- $\ominus\ Var(\gamma, v) = (Fvar\ v)$
- $\ominus\ Const(\gamma, v) = (Fconst\ c)$
- $\ominus\ (\ldots(f : \delta\ t_1)\ \ldots\ t_n) = Fapp(Fconst(f, n), [\ominus(t_1), \ldots, \ominus(t_n)])$

It should be obvious that $(\oplus\ \gamma) \circ \ominus = I$ and hence there is an isomorphism. ∎

We have thus established a fragment of *HOL* corresponding to single sorted first order equational logic. An interesting aspect of this is that the definition of terms is parameterized on an arbitrary type, which could be polymorphic.

5.4.2 Subsidiary inference rules

We consider the application of a substitution to a theorem and the computation of critical pairs as inference rules, since they are important components of completion that need to be inside the logic. The rewriting of theorems is done by using the primitive rule of inference SUBST. This is only possible because equational logic theorems are universally quantified at the outermost. Sound rewriting of arbitrarily quantified theorems is the subject of [Pau83].

5.4.2.1 Applying a substitution

The derived rule of inference INST applies a substitution to a theorem and is thereby the basis of term replacement in the HOL system. It is unfortunate that INST is easily confused with INST_TYPE; remember that INST_TYPE instantiates the *type variables* of a theorem, while INST instantiates the *free term variables* of a theorem.

$$\texttt{INST} \quad \frac{A1, \ldots, An \vdash tm}{A1, \ldots, An \vdash \theta(tm)}$$

Since this is not a primitive facility in HOL, it is effected by

1. $\theta' \ = \ \{t/v \in \theta : v \text{ is not free in the assumptions }\}$
2. Converting $\theta' \ = \ \{t_1/v_1, \ldots, t_m/v_m\}$ into two sequences:
   - $(v_1, \ldots, v_m)$ — a generalization sequence
   - $(t_m, \ldots, t_1)$ — a specialization sequence

3. Generalizing (in left-to-right order) on the variables in $(v_1, \ldots, v_m)$ to get $\{A_1, \ldots, A_n\} \vdash \forall v_m \ldots v_1.\ tm$
4. Specializing (in left-to-right order) with all the terms in $(t_m, \ldots, t_1)$ to get $\{A_1, \ldots, A_n\} \vdash \theta(tm)$

To get the effect of applying a substitution, the last-generalized variable must correspond to the first term specialized. Further, all generalizations must be done first, so that a specialization doesn't get done and a subsequent generalization bind variables introduced by the specialization.

5.4.2.2 Critical pair formation

The heart of the completion algorithm is the production of critical pairs. As already mentioned, critical pairs must be theorems, hence they must follow from an inference rule. The split between inference and computation comes with the computation of occurrences of critical pairs, or overlaps. An *overlap* between rules $r_1 = t_1 \rightarrow u_1$ and $r_2 = t_2 \rightarrow u_2$ is a pair (θ, occ) of a substitution (produced by the first order unification algorithm) and an occurrence. The occurrence defines the path to the non-variable subterm of t_1 that unified with t_2. The following derived rule of inference, given two rules, and an overlap between the first rule and the second rule, returns the critical pair corresponding to that overlap.

$$\texttt{CRITICAL\_PAIR} \quad \frac{A \vdash t_1 = u_1, \quad B \vdash t_2 = u_2 \quad (\theta, occ)}{A \cup B \vdash (\theta(t_1))/occ := \theta(u_2)) = \theta(u_1)}$$

(We re-use the notation from Chapter 3 that $tm_1/occ := tm_2$ denotes the term identical with tm_1 except that the subterm pointed at by occ has been replaced by tm_2.)

We will step through the execution of CRITICAL_PAIR on the example given in section 4.3. Assume that we have already derived the theorems $r1 = \vdash f\ x\ (g\ x\ a) = h\ x$ and $r2 = \vdash g\ b\ y = k\ y$. We first compute the overlap:

```
#val [(theta,occ)] = overlap r1 r2 [];
theta = [('b', 'x'); ('a', 'y')] :  (term * term) list
occ = [2] :  int list
```

1. r_1' = INST θ r_1.

```
#val r1' = INST theta r1;
r1' = ⊢ f b (g b a) = h b :  thm
```

2. r_2' = RENAME r_2. RENAME is a derived rule of inference that merely changes all the free variables of a theorem to be new to the system. This prevents spurious "occurs" check errors in unification. In this example we do not have to do any renaming, because the variables of $r1$ and $r2$ are disjoint.

3. r_2'' = INST θ r_2'.

```
#val r2'' = INST theta r2;
r2'' = ⊢ g b a = k a :  thm
```

4. Generate v, a brand new variable of the right type, and form a template by replacing the subterm of θt_1 at *occ* by v:
   template = $(\theta t_1)[occ := v] = (\theta u_1)$.

```
#val (v,template) = mk_template r1' occ;
v = 'v' :  term
template = 'f b v = h b' :  term
```

5. *critical_pair* = SUBST $[r_2''/v]$ *template* r_1'

```
#val critical_pair = SUBST [(r2'',v)] template r1';
critical_pair = ⊢ f b (k a) = h b :  thm
```


The SML function *critical_pairs* that incorporates CRITICAL_PAIR has the type $rule \rightarrow rule \rightarrow thm\ \ list$, and is thus a derived rule of inference. The SML function *kb* implements Huet's version of Knuth-Bendix completion [Hue81] and calls *critical_pairs*. It has type $(term \rightarrow term \rightarrow bool) \rightarrow thm\ \ list \rightarrow thm\ \ list$, hence is also a derived rule of inference. Its first argument should be a reduction ordering, and it checks that its second argument is a homogeneous list of first order equality theorems.

5.4.3 Example

We complete an equational presentation of group theory, the ubiquitous example in Term Rewriting Systems. The term order is the recursive path ordering with status [KL80]. Notice that the declared constants are polymorphic, as are all the returned theorems: the resulting set of theorems can be instantiated to any type by use of INST_TYPE.

```
#new_theory "group";
#new_infix (*,':*->*->*'); new_constant (~,':*->*'); new_constant (1,':*');

#let e1 = new_axiom ("e1", '1 * x = x');
##and e2 = new_axiom ("e2", '(~ x) * x = 1')
##and e3 = new_axiom ("e3", '(x * y) * z = x * (y * z)');
e1 = ⊢ !x. 1 * x = x
e2 = ⊢ !x. (~x) * x = 1
e3 = ⊢ !x y z. (x * y) * z = x * (y * z)
() : void

#kb (rpos status ~_*_1) [e1,e2,e3];
[⊢ 1 * x = x,   ⊢ (~x) * x = 1,
 ⊢ (x * y) * z = x * (y * z),
 ⊢ (~x) * (x * y) = y,
 ⊢ x * 1 = x,   ⊢ ~1 = 1,
 ⊢ ~~x = x,   ⊢ x * (~x) = 1,
 ⊢ x * ((~x) * y) = y,
 ⊢ ~(x * y) = (~y) * (~x)] : thm list
Time: 16.2s
Intermediate theorems generated: 17436
```

The completion procedure given here has many applications, amongst them the standard one of providing a decision procedure for equality for (some) equational theories. This would be an aid to those developing such theories [Gun89], although there is typically much more than just rewrite rules to provide for a theory.

The advantage of having completion in the logic is that the user can rely on its output to be theorems; use of it will not require justification by metatheorems.

5.5 Conclusions

In this chapter, we have incorporated into *HOL* the fundamental algorithm from the field of Term Rewriting Systems. We feel that the two fields have a natural intersection. Original contributions rising out of this are the identification of $\mathcal{F}_\gamma$, the parametric single sorted first order terms of *HOL*, and the implementation of Knuth-Bendix completion as a derived rule of inference in the equational logic defined by $\mathcal{F}_\gamma$.

The foundational work done in this chapter opens up many possibilities:

- This work should be extended to equational completion[PS81].
- We should investigate completion over less restricted sets of terms.
- If we recall that HOL *types* are essentially first order terms, then it becomes a possibility to do completion on type equations. Completion of type equations is, of course, extra-logical and currently, this is only of technical interest. This idea is due to a remark by Tobias Nipkow.
- One of the most promising uses of completion seems to be as a semi-decision procedure. Before manually proving an equality, we could just submit it to the completion procedure and see if it can automatically derive it.

- Proof by consistency [Mus80, HH82, Bac88]("inductionless" induction) should be investigated. There may be a connection with recursively defined datatypes.
- Tobias Nipkow has proved a critical pair lemma for a subset of the typed lambda calculus [Nip90]. We would like to investigate a completion implementation for that.
- In the unification field, investigation into strong subsets of the lambda calculus with the most general unifier property is of interest.
- How fast we can make rewriting go? If we represent the syntax of a programming language in **hol90**, it is possible to execute programs by rewriting (logical inference) with the operational semantics. If this mode of execution can be made fast enough to be usable, that would allow a new slant on logic programming, since the programming language is embedded in the logic and meta-theoretic results about programs can be proved in the logic. Also, compilation and optimization could perhaps be expressed as rewrite rules that get incorporated into the rewriting process.

Chapter 6

AC Unification

Unification is the study of how to turn equations into identities by applying substitutions. Applications of unification are found in programming languages, computer algebra systems, artifical intelligence, theorem proving, and systems for polymorphic type inference such as that given in Chapter Two. An introduction to first order unification is given in Appendix A.

A problem with automatic proof systems is that they have a hard time doing inference with permutative equations, such as those expressing commutativity and associativity. One remedy for this, initiated by Plotkin [Plo72], is to incorporate the permutative axioms into unification and rewriting. This began the study of equational unification, which relaxes the constraint that the unified terms be identical; instead, the unified terms are to be provably equal in equational logic. Equational unification is a very general field, encompassing for instance Hilbert's Tenth problem [JK90], but the fundamental algorithm in this area is undoubtedly associative-commutative (AC) unification, in which the equational axioms express the associative and commutative properties of members of Σ. Some common examples are $\{\cap, \cup\}$ in set theory; $\{\wedge, \vee\}$ in logic; and $\{\times, +\}$ in arithmetic.

The notion of a single most general unifier no longer works in AC unification. Consider $x \underset{AC}{=} a + b$; it has unifiers $\{a + b/x\}$ and $\{b + a/x\}$, neither of which are instantiations of the other. The best we can do is to find the smallest set of unifiers from which we can produce all unifiers. The *minimal, complete* set of unifiers μCSU has the properties

completeness $unifier(\sigma) \supset \exists \theta \in \mu CSU.\ \exists \gamma.\ \gamma \circ \theta = \sigma$

minimality $\forall \theta_1 \theta_2 \in \mu CSU.\ \exists \gamma.\ \gamma \circ \theta_1 = \theta_2 \supset \theta_1 = \theta_2$

The original AC unification algorithm was discovered by Stickel [Sti81]. The termination of the algorithm was solved by Fages [Fag84]. Fortenbacher [For85] gives an elegant algebraic presentation and some optimizations of Stickel's algorithm. The most recent advances in the study of AC unification are described in [AK90, BCD90]. A test suite and the performance of some of the leading implementations is given by Buerckert *et al.* [BHK+88].

We have implemented Fortenbacher's AC unification algorithm. In this chapter, we describe in detail the workings of the algorithm and show how to use the algorithm to implement AC-rewriting, which represents a significant increase in power over the current rewriting tools for **hol90**. An interesting aspect of AC rewriting is the expression of an algorithm to decide AC equivalence as a tactic.

6.1 The algorithm

Consider a term made from applications of an associative and commutative function symbol, e.g.,

$$(x + 1) + (f(x) + (y + x))$$

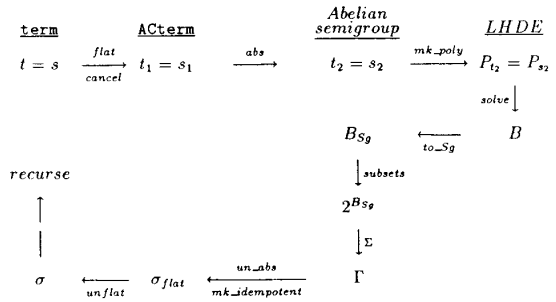
The properties of + allow us to flatten the term and regard it as a variadic + applied to the the multiset $\{x; 1; f(x); y; x\}$. Stickel's insight was to map the multiset to a simpler domain, do unification there, and transform those substitutions back to unifiers in the domain of terms. He introduced the concept of *variable abstraction* whereby all constants and function applications in the flattened term are mapped to new variables.

The result of variable abstraction is considered as a term in an Abelian semigroup. Linear Homogeneous Diophantine Equations (LHDE's) over the positive integers are isomorphic to Abelian semigroups, so a solution of the LHDE corresponding to two

abstracted terms induces a solution to the unification problem in the Abelian semigroup. To return to the domain of terms requires reconciling the Abelian semigroup unifier with the variable abstraction substitution.

A feature of Stickel's algorithm is that it places no restrictions on the first order terms: more than one AC constant is allowed and non-AC constants are also allowed in the terms to be unified. As a consequence, Stickel's algorithm is actually embedded into the normal first order unification algorithm [MM82] (see Appendix A). When we wish to unify two applications, say $f(t_1, \ldots, t_n)$ and $f(s_1, \ldots s_m)$, if f is an AC constant, then we must perform the algorithm to be described shortly, as opposed to the usual decomposition operation. Extra difficulties then arise, since AC unification returns a set of substitutions, instead of the usual, single, substitution.

The following diagram summarizes Stickel's approach. Read the right-pointing arrows as problem mappings, and the down- and left-going arrows as the finding of solutions to those problems. In cases where functions are above and below an arrow, the top function is executed first. We shall explain the algorithm by explaining the transitions in the diagram.



The algorithm uses four domains: first order terms, AC terms, Abelian semigroup, and LHDE. We develop an example along the way, the one given in Stickel's original paper:

$$(x + (y + a)) + b + x + c = b + b + c + z + b.$$

flat

Flattening of a term only proceeds down to leaves, where a leaf is a variable, constant, or an application headed by a function not equal to the top AC function symbol. Therefore, we obtain

$$+(x, x, y, a, b, c) = +(z, b, b, b, c).$$

cancel

Stickel proved that cancelling common subterms preserves unifiers. Therefore, we obtain

$$+(x, x, y, a) = +(z, b, b).$$

abs

All non variable subterms are replaced by new variables. We remember this operation by forming a substitution that, when applied to the variables, results in the original terms. We are now in an Abelian semigroup (an algebra with a single associative and commutative operator). To mirror this, we also map the AC operator to $\star$, the semigroup operator. The new variables are w and u; the substitution θ is $\{a/w, b/u\}$. The unification problem is now

$$x \star x \star y \star w = u \star u \star z.$$

mk_poly

Sum the multiplicities of each variable in each term. This gives us a diophantine equation with no exponents and constant coefficients. Abelian semigroups are isomorphic to LHDE's [Fag84]. Therefore, the problem is now

$$2x + y + w = 2u + z.$$

solve

We can compute a (finite) basis set of solutions to the equation. A basis B has the property that every solution to the equation can be expressed as a linear combination of basis solutions. The basis set is computed by a backtracking algorithm that simply enumerates the solutions up to a bound. The bounds on solutions are derived in [Hue78]. The basis solutions for our example are:

			(a)	(b)	
	$2x$	y	w	$2u$	z
1	0	0	1	0	1
2	0	1	0	0	1
3	0	0	2	1	0
4	0	1	1	1	0
5	0	2	0	1	0
6	1	0	0	0	2
7	1	0	0	1	0

to_Sg

Now we must transfer LHDE unifiers back to semigroup unifiers. Generate new variables: one for each basis solution. Consider a basis solution b_i and v_i its newly generated variable: for each box in b_i regard the number as a multiplicity for v_i. If

the box has a 1, put v_i; if the box has a 2, put a $v_i \star v_i$, etc. If the box has 0, don't put anything. So we have

				(a)	(b)	
	new var.	$2x$	y	w	$2u$	z
1	v_1	-	-	v_1	-	v_1
2	v_2	-	v_2	-	-	v_2
3	v_3	-	-	$v_3 \star v_3$	v_3	-
4	v_4	-	v_4	v_4	v_4	-
5	v_5	-	$v_5 \star v_5$	-	v_5	-
6	v_6	v_6	-	-	-	$v_6 \star v_6$
7	v_7	v_7	-	-	v_7	-

2^B

It is our aim to transform the basis solutions into unifiers in the semigroup, but there is one catch—a 0 in a basis solution does not correspond to any element of a semigroup; semigroups do not have an identity element. The next best thing is to consider all the subsets of B that have non-zero column sums. There are many of these, so we impose some constraints based on optimizations noticed by Fortenbacher:

Condition 4 [For85, page 386]. For a subset with all column sums non-zero, it is not allowed that a column is headed by a constant or a function application and the sum of that column is > 1. This implies that a row with an entry ≥ 2 in a non-variable column can be eliminated from the process of subset construction. (This allows us to delete row 3.)

Condition 5 [For85, page 386]. Within a row of the basis, all boxes containing a 1 must have their corresponding terms AC-unifiable. (This allows us to delete row 4.)

Using these criteria, we are left with 4 subsets, out of a possible $2^7 = 128$:

$$\{\ \{1,5,6\},\ \{1,2,5,6\},\ \{1,2,7\},\ \{1,2,6,7\}\ \}$$

Σ

For each subset, make a substitution by binding the column variable with the sum of the columns. The result Γ is the set of semigroup unifiers.

$\Gamma =$
$\{\ \{v_6/x;\ v_5 \star v_5/y;\ v_1/w;\ v_5/u;\ v_1 \star v_6 \star v_6/z\},$
$\{v_6/x;\ v_2 \star v_5 \star v_5/y;\ v_1/w;\ v_5/u;\ v_1 \star v_2 \star v_6 \star v_6/z\},$
$\{v_7/x;\ v_2/y;\ v_1/w;\ v_7/u;\ v_1 \star v_2/z\},$
$\{v_6 \star v_7/x;\ v_2/y;\ v_1/w;\ v_7/u;\ v_1 \star v_2 \star v_6 \star v_6/z\}\ \}$

un_abs

We recall the substitution θ from *abs*: $\{a/w;\ b/u\}$. We must reconcile the bindings of θ with each member of Γ. Every binding in θ is of the form tm/v, where tm is a constant or an application and v is a variable created to be new. If there is a binding s/v in a $\gamma \in \Gamma$, then s is a variable, since tm can not unify with a $\star$-term. Hence we replace the binding s/v in γ with tm/s.

mk_idempotent

Idempotence is essential for minimality. Recall that an idempotent unifier σ is one that has the property $\sigma \circ \sigma = \sigma$. Another characterization of idempotence is that the variables in the domain and the range of the substitution do not intersect. In our example, we now have Γ as

$\{v_6/x;\ b \star b/y;\ a/v_1;\ b/v_5;\ a \star v_6 \star v_6/z\}$
$\{v_6/x;\ v_2 \star b \star b/y;\ a/v_1;\ b/v_5;\ a \star v_2 \star v_6 \star v_6/z\}$
$\{b/x;\ v_2/y;\ a/v_1;\ b/v_7;\ a \star v_2/z\}$
$\{v_6 \star b/x;\ v_2/y;\ a/v_1;\ b/v_7;\ a \star v_2 \star v_6 \star v_6/z\}$

unflat

This is the trivial mapping of terms in the flattened unifier to structured terms. Replace $\star$ with $+$. Add brackets in any old way. We now have a set of "unifiers" Θ.

$\{v_6/x;\ b + b/y;\ a/v_1;\ b/v_5;\ a + v_6 + v_6/z\}$
$\{v_6/x;\ v_2 + b + b/y;\ a/v_1;\ b/v_5;\ a + v_2 + v_6 + v_6/z\}$
$\{b/x;\ v_2/y;\ a/v_1;\ b/v_7;\ a + v_2/z\}$
$\{v_6 + b/x;\ v_2/y;\ a/v_1;\ b/v_7;\ a + v_2 + v_6 + v_6/z\}$

recurse

Let us take stock of the situation. We have a set of "unifiers" Θ that are not quite unifiers since they have not taken account of the results of any subproblems. What are the subproblems? They arise from unflattened application terms, *e.g.*, in this example, those not equal to $+$. Fortunately for the reader, no such subterms exist in this example: a and b are the only non-variable subterms. A unification problem like $x + x + a + a = g(y) + g(a) + z + z$ does have subproblems: one is $g(y) = g(a)$.

The recursive calls comprise one of the most expensive parts of the algorithm since, in general, a subproblem may have a μCSU Θ'. Each σ in Θ' must get composed with each member of Θ to get the final set. The combinatorial aspects of this are exacerbated by the fact that a σ_1 in Θ' may have a binding tm_1/v and σ_2 in Θ' may have a binding tm_2/v. This requires solving the new subproblem $tm_1 = tm_2$ with its own μCSU, and folding those solutions in with the others.

6.2 Using AC unification in hol90

If we have two terms, t_1 and t_2, in $\mathcal{F}_\gamma$ (for some γ), that we unify with the AC unification algorithm, we obtain a minimal, complete set of unifiers Θ. If the implementation of the algorithm is correct, then each instance is a theorem: $\vdash \theta t_1 \underset{AC}{=} \theta t_2$. Of course, it is not valid to claim $\vdash t_1 = t_2$. How then can we use the results of the algorithm?

Recall that rewriting takes a list L of equality theorems and a theorem thm and wherever an instantation θl of a left hand side of a theorem of L is found in thm, that subterm of thm is replaced by θr. The way that θ is found is by means of a matching algorithm. Matching is "one-way" unification. By "one-way" we mean that only one term is allowed to be substituted into. In the case of rewriting, the subterm of thm and l are unified to get θ, the difference being that any variables in the subterm of thm are treated as constants, *i.e.*, they are not allowed to become instantiated in the unification process. Therefore, one application of AC unification is *AC rewriting*, performed in the following manner:

- Let θ be an AC match between $\vdash l = r$ of a theorem in the rewrite list and a subterm t, *i.e.*, $\theta l \underset{AC}{=} t$. At this point, it is not valid to replace t by θr.
- Prove that $\theta l = t$, by using the AC theorems.
- It is now valid to replace t by θr.

The only interesting part of this is the automatic proof that $\theta l = t$ using the AC theorems. We do this via an algorithm expressed in the tactic language of *HOL*. Let $\theta l = t_1$ and $t_2 = t$. For $t_1 = t_2$, we first associate everything to the right, by rewriting with the associativity theorem. Then we have $X + t_1' = Y + t_2'$. There are three cases:

lhs = *rhs*. Then we are done.

$X \underset{AC}{=} Y$. If $X \equiv Y$, we cancel X and Y off their respective sides, and continue. Otherwise, we make a recursive call to the algorithm to prove $X \underset{AC}{=} Y$, use that theorem to rewrite X to Y, cancel Y from both sides, and continue.

$X \not\underset{AC}{=} Y$. In this case, we know by the correctness of the unification algorithm that an AC-equivalent to X does exist somewhere in t_2'. What we must do is rearrange the *rhs* so that cancellation is possible. We do this by using the commutativity property on the *rhs* to get $X + t_1' = t_2' + Y$. Then we reassociate the *rhs*. This has the effect of "rotating" the *rhs* by one element. We rotate by one element at a time, until a term AC-equivalent to X appears in the left-most position of the *rhs*, whereupon we can cancel and continue.

Example. Consider

$$(x + 0) \times ((y + 0) \times (2 \times 2)) = (z + (z + z)) \times w$$

with $\{\times, +\}$ as the AC operators. This has μCSU:

$$\{2 \times (2 \times (y + 0))/w;\ 0/z;\ 0 + 0/x\}$$
$$\{2 \times (2 \times (y + 0))/w;\ v_1 + 0/z;\ v_1 + (v_1 + (v_1 + (0 + 0)))/x\}$$
$$\{2 \times (2 \times (x + 0))/w;\ 0/z;\ 0 + 0/y\}$$
$$\{2 \times (2 \times (x + 0))/w;\ v_2 + 0/z;\ v_2 + (v_2 + (v_2 + (0 + 0)))/y\}$$

The last unifier exercises the algorithm fully. Instantiate the equation with the last unifier to get

$$\begin{aligned} & (x + 0) \times (((v_2 + (v_2 + (v_2 + (0 + 0)))) + 0) \times (2 \times 2)) \\ = \ & ((v_2 + 0) + ((v_2 + 0) + (v_2 + 0))) \times (2 \times (2 \times (x + 0))) \end{aligned}$$

We wish to prove this is a theorem. Use the association laws to associate everything to the right:

$$\begin{aligned} & (x + 0) \times (v_2 + (v_2 + (v_2 + (0 + (0 + 0)))) \times (2 \times 2)) \\ = \; & (v_2 + (0 + (v_2 + (0 + (v_2 + 0)))) \times (2 \times (2 \times (x + 0)))) \end{aligned}$$

The multiplicands are not AC-equivalent, so the *rhs* must be rotated until they are. When the rotation is done, we find that they are identical:

$$\begin{aligned} & (x + 0) \times ((v_2 + (v_2 + (v_2 + (0 + (0 + 0))))) \times (2 \times 2)) \\ = \; & (x + 0) \times (v_2 + (0 + (v_2 + (0 + (v_2 + 0)))) \times (2 \times 2)) \end{aligned}$$

Now we cancel:

$$\begin{aligned} & (v_2 + (v_2 + (v_2 + (0 + (0 + 0))))) \times (2 \times 2) \\ = \; & (v_2 + (0 + (v_2 + (0 + (v_2 + 0))))) \times (2 \times 2)) \end{aligned}$$

The multiplicands are AC-equivalent, but not identical; we recursively prove them equivalent and replace the multiplicand on the *rhs* by that of the left.

$$\begin{aligned} & (v_2 + (v_2 + (v_2 + (0 + (0 + 0))))) \times (2 \times 2) \\ = \; & (v_2 + (v_2 + (v_2 + (0 + (0 + 0))))) \times (2 \times 2)) \end{aligned}$$

Now we have an identity and have proved the theorem.

6.3 Conclusion

The contributions of this chapter include the implementation of a decision procedure for AC-equivalence, and the provision of an AC rewriting facility.

Independently, Nipkow has done something similar in the *Isabelle* theorem prover [Nip89a, Nip89b]. He presents matching, unification, and rewriting for various equational theories by means of presenting the algorithms as proof rules. The machinery of *Isabelle* does the rest. The advantage of this is that it is very simple to get an algorithm going. As he notes, the disadvantage of this is that it is very slow and his

AC unification algorithm need not terminate. In contrast, the implementation presented here should be as fast as any other, terminates, and is therefore more suitable for production use.

Subjects for future research include:

- exploring the utility of AC rewriting in proofs. After all, it is NP complete [BKN85]. (This paper also show that deciding AC equivalence in in P.)
- using the AC unification algorithm in the implementation of *HOL* datatypes defined over a set of equations.

Chapter 7

From Logic to Layout

The previous chapters are concerned with the development of a theorem prover and the provision of proof tools. In this chapter, we focus on an application of the theorem prover to a problem in hardware verification. The problem is that verification is often carried out in isolation from the production of the circuit.

We have implemented a system that transforms *HOL* specifications of circuits to net lists suitable for gate array implementation. By virtue of an accompanying proof, and since the transformation occurs completely inside the logic, the final net list is guaranteed correct over all inputs and outputs. Standard vendor hardware then floorplans, places, routes, and computes clock speeds. This style of design neatly separates concerns, allowing the specifier to concentrate upon accurate and complete specifications and the functional correctness of a chip design.

7.1 Overview

Simulation and formal verification [Gor85] are two means of showing that a circuit meets a specification. Both use models: a simulation has the model embodied in its code, while a description in logic of a circuit has the model embodied as specifications at the leaves of the design hierarchy. For a given circuit and model, both approaches use essentially the same specification.

In spite of their abstract similarity, verification and simulation have their differences, most arising out of the brute-force-computation — user-directed-deduction tradeoff.

- For a large class of circuits, those that can be proved correct by an inductive proof, simulation is more expensive than verification because a simulation must consider cases exponential on the size of the input, whereas a verification need only consider two cases. Many subsystems fall into this category, including adders, stacks, and others.

- Verification, as we practice it in **hol90**, is cumulative - new proof procedures can be built on top of existing procedures, and there is a burgeoning user community that actively contributes to the proof tools at hand. Thus, although someone may spend six months proving theorems about a certain style of design or the theory of bit strings, the work never has to be done again. The principal challenge in the hardware verification field, as we see it, is the building of useful libraries of specifications, theorems, and proof methods.

- A simulation is easier to change than a proof. For a simulation, the description of the circuit is changed and the program simply re-executed. For a verification, a great deal of user effort may be required to re-prove a changed circuit.

We are interested in functional correctness, that is, we do not concern ourselves with issues such as clock speed and layout. This has both pros and cons - functional correctness is an important issue in itself, but a proof in isolation is not a useful object. This chapter presents work that allows us to import logic definitions of circuits into the design environment. This has several benefits: a large number of relatively high level building blocks are available to us for design since we use the LSI library [lsi86]; our proofs give us peace of mind about the correctness of the circuit; circuits defined by primitive recursion can be instantiated to any size; and the LSI chip building methodology handles issues we are less interested in, such as layout, timing, and routing.

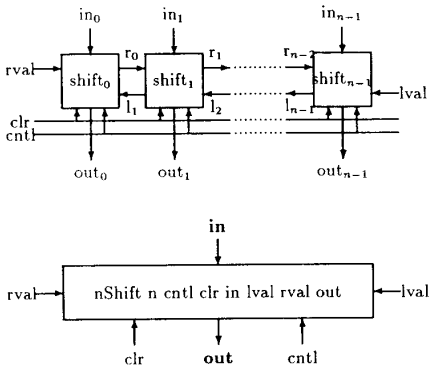


Figure 7.1: The nShifter circuit

7.2 The specification of the shifter

We take as our example the shifter shown in figure 7.1. For this proof, we choose a Boolean model in which a signal on a wire can only be either high or low (more complicated models can be easily introduced). All signals are assumed to settle between clock pulses. The current state of the register is held in the vector **out**. From clock tick to clock tick, the shifter may either broadside load the vector **in**, shift *rval* in from the left, shift *lval* in from the right, or remain as is. Both **out** and **in** are n-bits wide. The shifter control lines are named *clr* and *cntl*.

The value to be seen on **out** at time *t+1* is determined by the values on the control lines at time *t*. If *clr* is high at time *t*, then the next value on **out** will be all F (low). Otherwise, if *clr* is low at time *t*, we examine the value on *cntl*. *cntl* has

two wires (c_1 and c_0) to accommodate the 4 commands it can accept. We assign the following codes to (c_1,c_0):

```
LOAD = (T,T)
SHL  = (T,F)
SHR  = (F,T)
NOP  = (F,F)
```

In the following *HOL* specification, we use the notation **out** *t* to represent the value of the vector **out** at time *t*. The same holds for the vector **in**. The specification of the shifter tells what to expect on **out** at time *t+1* given the values of the control signals and the other inputs at time *t*. *n* gives the size of the sub-system.

```
nShift_spec n cntl clr in lval rval out =
(∀ t:num. (out (t+1) = clr t) => zeros
        | (cntl t = LOAD) => in t
        | (cntl t = SHL)  => (tl (out t)) :: [(lval t)]
        | (cntl t = SHR)  => (rval t) :: front (out t)
        |                    out t)
```

7.2.1 The HOL description of the shifter

In this work, we allow two types of definitions of circuits: basic definitions and primitive recursive definitions. Employing these, the designer builds a hierarchical description of his circuit. A HOL definition of a hardware device can be read as a structural description of the circuit [Gor85].

Basic definitions. For the description of non-replicated devices, we adopt a standard form: internal nodes at the same level in the design hierarchy of a circuit are conjoined, and lines running between those devices are existentially quantified. Leaf nodes in the design express the behaviour of the building blocks in a certain model.

Devices can be named and parameterized, so that the following defines a four-to-one multiplexer, which is formed by wiring together four three-input and one four-input nand gates.

```
4to1_mux (s0, s0bar, s1, s1bar, i0, i1, i2, i3) d =
  ∃ p q r s .
  (ND3 (s1bar,s0bar,i0) p) ∧
  (ND3 (s1bar,s0,i1) q)    ∧
  (ND3 (s1,s0bar,i2) r)    ∧
  (ND3 (s1,s0,i3) s)       ∧
  (ND4 (p,q,r,s) d)
```

The following are the HOL definitions of the building blocks of the shifter: an inverter, two types of nand gate, and a flip flop. A definition amenable to the translator must have two arguments; the first argument is held to be the input ports and the second to be the output ports. (This requirement is loosened for parameterized definitions, in which the first argument is taken to be the size of the device and the following two are inputs and outputs.) Notice that all these definitions except for *flip_flop* do not say anything about structure, only about behaviour. The definition of *flip_flop* is merely to hide the unused wire *qbar*.

```
⊢ IV i out = (∀ t. out t = ¬ i t)

⊢ ND3 (a,b,c) (out) =
   ∀ (t:num). (out t = ¬(a t ∧ b t ∧ c t))

⊢ ND4 (a,b,c,d) (out) =
   ∀ (t:num). (out t = ¬(a t ∧ b t ∧ c t ∧ d t))

⊢ FD2 (d,cp,cd) (q,qbar)) =
   ∀(t:num). q(t+1) = (¬ cd t) => F |
                      (cp t) => (d t) |
                      (q t)

⊢ flip_flop (d,cp,cd) (q) = ∃ qbar. FD2 (d,cp,cd)(q,qbar)
```

An *instance* of a definition is analogous to a function call on the definition: it is the head of the definition with the ports instantiated. For example, the definition of *4to1_mux* contains five instances of two definitions:


```
   (ND3 (s1bar,s0bar,i0) p), (ND3 (s1bar,s0,i1) q),
   (ND3 (s1,s0bar,i2) r),    (ND3 (s1,s0,i3) s),
   (ND4 (p,q,r,s) d)
```

Primitive recursive definitions. In HOL, a replicated device is described by primitive recursion, in effect defining an infinite family of devices. Primitive recursive definitions are typically used in inductive proofs about properties shared by the entire family; however, they have another use: a fully expanded instantiation of a primitive recursive definition reveals the structure of that member of the family. When we translate replicated devices, we first expand them fully.

The following define the selector and the delay vectors by primitive recursion, and the shifter by a basic definition. Notice that there is no connection between adjacent members in either vector. That wiring, which is complex, is done in the definition of the shifter. (The unused wires *s0bar* and *s1bar* are hidden by the existential quantification.)

```
⊢ (nDel 0 (clr,clk,d) q = flip_flop ((d 0),clk,clr) (q 0)) ∧
(nDel (SUC n) (clr,clk,d) q =
    (nDel n (clr,clk,d) q) ∧
    (flip_flop (d(SUC n),clk,clr) q(SUC n)))


⊢ (nSelect 0 (s0,s0bar,s1,s1bar,par,dsl,dsr,hld) d =
     4to1_mux (s0,s0bar,s1,s1bar,(par 0),
               (dsl 0),(dsr 0),(hld 0)) (d 0)) ∧
(nSelect (SUC n) (s0,s0bar,s1,s1bar,par,dsl,dsr,hld) d =
  (nSelect n (s0,s0bar,s1,s1bar,par,(hld(SUC n)),
              dsr,(hld n)) d)  ∧
  (4to1_mux (s0, s0bar, s1, s1bar, (par(SUC n)),
             dsl,(hld n),(hld(SUC n)))
            (d (SUC n))))
```

```
⊢ nShift n (s1,s0,clr,clk,in,lval,rval) out =
   ∃ v s0bar s1bar.
   (IV s0 s0bar) ∧
   (IV s1 s1bar) ∧
   (nSelect n (s0,s0bar,s1,s1bar,in,lval,rval,out) v) ∧
   (nDel n (clr,clk,v) out)
```

7.2.2 Verification of the shifter

The actual proofs are unenlightening to those not versed in *HOL*, so we content ourselves with listing the time taken for the steps in the complete machine verification along with the number of primitive inferences steps (a measure of the complexity of the proof). The proofs of *nDel* and *nSelect* are by induction.

```
4to1_mux_correct =
⊢ 4to1_mux (s0,s0bar,s1,s1bar,i0,i1,i2,i3) o =
   4to1_mux_spec (s0,s0bar,s1,s1bar,i0,i1,i2,i3) o
Run time: 18.8s      Garbage collection time: 6.1s
Intermediate theorems generated: 1360

nSelect_correct =
⊢ nSelect n (s0,s0bar,s1,s1bar,par,dsl,dsr,hld) out =
  nSelect_spec n (s0,s0bar,s1,s1bar,par,dsl,dsr,hld) out
Run time: 126.5s     Garbage collection time: 61.3s
Intermediate theorems generated: 8956

nDel_correct =
⊢ nDel n (clr,clk,d) q = nDel_spec n (clr,clk,d) q
Run time: 77.4s      Garbage collection time: 32.0s
Intermediate theorems generated: 7013

nShift_correct =
⊢ nShift n (s1,s0,clr,clk,in,lval,rval) out =
   nShift_spec n (s1,s0,clr,clk,in,lval,rval) out
Run time: 124.5s     Garbage collection time: 61.8s
Intermediate theorems generated: 7072
```

These proofs were done by Graham Birtwistle.

7.3 Netlist descriptions

In NDL, a circuit is described by a *module*. Larger modules are hierarchically formed from occurrences of other modules. The building block modules out of which all others are eventually built are found in the LSI library. A module is described by its name, its input and output pins, the sub-components it is made of, and internal lines, known as *signals*, that join the sub-components to each other or to the input or output pins. The discussion that follows depends heavily on [ndl85].

An NDL module has the following format (text in angle brackets is commentary):

```
MODULE  <the type name of the module>;
DESCRIPTION <documentation of some sort>;
INPUTS  <names of input pins to the module>;
OUTPUTS <names of output pins to the module>;
{ BIDIRECTS }  < an optional statement >
DEFINE
   < component descriptions >
   < connection descriptions >
END MODULE;
```

When a module is defined, it is given a *type name*. When a module is used as a sub-component of another module, it is given an *occurrence name*. The DEFINE section of a module describes how the input pins are connected to sub-components, how the sub-components connect to each other, and how the sub-components connect to the output pins. This is achieved by two types of statements, *component descriptions* and *connection descriptions*.

A component description has the following form:

```
<occ. name> {output ports} = <type name> <input ports>;
```

It establishes the existence of a sub-component in the module being defined. The sub-component is an occurrence of a module defined previously; the occurrence

name, which must be unique, is bound to the module identified by the type name on the right hand side of the equality. NDL provides several ways to define the input and output ports lists. We use *implicit* format here - see [ndl85] for information on other formats.

The input ports list enumerates the connections of signals (which can be the input pins of the module, or internal lines) to the input pins of the sub-component. The output ports list is optional and enumerates the connections of the output pins of the instance to either an internal line or an output pin of the module being defined. Internal lines are defined merely by their existence in input and output ports lists. To connect two or more sub-components with an internal line, the line must appear in the appropriate input and output ports lists of the connected sub-components.

A connection description establishes and names a connection between the listed signals, *i.e.*, that they come together and therefore must have the same value. It has the following form: `<target> = <signal list>` where a signal has the form `<occurrence name> (<module pin name>)`.

In the case of *HOL* descriptions of hardware, the target name is always the name of an output pin of the module being defined. Furthermore, each output pin of the module being defined has a connection description. That description associates the module output pin with the output pins of the sub-components that send signals out of the module through it. The following example is the NDL description of the *4to1_mux*.


```
MODULE  4to1_mux;
DESCRIPTION  Automatically generated from HOL;
INPUTS  s0, s0bar, s1, s1bar, i0, i1, i2, i3;
OUTPUTS  d;
DEFINE
U1 (d) = ND4(p, q, r, s);
U2 (s) = ND3(s1, s0, i3);
U3 (r) = ND3(s1, s0bar, i2);
U4 (q) = ND3(s1bar, s0, i1);
U5 (p) = ND3(s1bar, s0bar, i0);
d = (U1(out));
END MODULE;
```

7.4 Translation from HOL to NDL

The algorithm operates by expanding the circuit down to its building blocks (by rewriting) as follows:

1. If the circuit definition is composed of only building blocks, stop.
2. Otherwise, find all definitions corresponding to top level instances.
3. For each definition, translate to NDL, as follows
   - the definition maps to an NDL module, instances map to sub-components, and existentially quantified variables in the body map to internal signals connecting sub-components.
4. Rewrite the top level instances with their definitions.
5. Go to 1.

At each step in its expansion, the circuit definition is a theorem of the logic. Therefore, if the definition has been proven, the net list is not only guaranteed to be that of the original circuit, it is also guaranteed to be that of a correct circuit.

7.4.1 Translation to NDL

The extraction of the input and output ports is trivial, owing to the convention we imposed. The first tuple corresponds to the input ports list and the second to the output ports list. The extraction of the component statements is also trivial. All instances in the body are collected and, for each instance we:

1. generate a unique identifier (the occurrence name);
2. print the output ports list for the instance;
3. print an '=';
4. print the definition's name (the type name); and
5. print the input ports list.

The extraction of the connection statements is a bit more involved. First, we note that there is going to be a connection statement for each output port in the definition (module). We also note that each output port in the definition must appear in the output ports list of at least one component statement. Therefore, for each output port, we collect all the occurrences of it in the output ports lists of the component statements and:

1. print the output port name, then print '= (' ;
2. for each such occurrence we
   - print the occurrence name;
   - print '(';
   - print the output pin name of the occurrence; and
   - print '), '.
3. Once all the occurrences have been handled, we print ');'.

In this way, the output pins of the occurrences are connected to the output pins of the module. Note that this solution takes care of the case where the outputs of some occurrences come together before connecting to an output pin.

We include the translation of an instantiation of an n-bit shifter, in this case instantiated to size 4. We omit the translation of the *4to1_mux*.

```
MODULE  4Shift;
DESCRIPTION  Automatically generated from HOL;
INPUTS  s1,s0,clr,clk,in0,in1,in2,in3,lval,rval;
OUTPUTS  out0, out1, out2, out3;
DEFINE
U1(v0,v1,v2,v3) = 4Selects(s0,s0bar,s1,s1bar,
                           in0,in1,in2,in3,lval,rval,
                           out0,out1,out2,out3);
U2(out0,out1,out2,out3) = 4Del(clk,clr,v0,v1,v2,v3)
U3(s0bar) = IV(s0);
U4(s1bar) = IV(s1);
out0 = (U2(q0));
out1 = (U2(q1));
out2 = (U2(q2));
out3 = (U2(q3));      END MODULE;


MODULE  4Selects;
DESCRIPTION  Automatically generated from HOL;
INPUTS  s0,s0bar,s1,s1bar,
        par0,par1,par2,par3,
        dsl, dsr,
        hld0,hld1,hld2,hld3,
OUTPUTS  d0, d1, d2, d3;
DEFINE
U1(d3) = 4to1_mux(s0,s0bar,s1,s1bar,par3,dsl,hld2,hld3);
U2(d2) = 4to1_mux(s0,s0bar,s1,s1bar,par2,hld3,hld1,hld2);
U3(d1) = 4to1_mux(s0,s0bar,s1,s1bar,par1,hld2,hld0,hld1);
U4(d0) = 4to1_mux(s0,s0bar,s1,s1bar,par0,hld1,dsr,hld0);
d0 = (U4(d));
d1 = (U3(d));
d2 = (U2(d));
d3 = (U1(d));         END MODULE;
```

```
MODULE  4Del;
DESCRIPTION  Automatically generated from HOL;
INPUTS  clk, clr, d0, d1, d2, d3;
OUTPUTS  q0, q1, q2, q3;
DEFINE
U1 (q3) = flip_flop(d3, clk, clr);
U2 (q2) = flip_flop(d2, clk, clr);
U3 (q1) = flip_flop(d1, clk, clr);
U4 (q0) = flip_flop(d0, clk, clr);
q0 = (U4(q));
q1 = (U3(q));
q2 = (U2(q));
q3 = (U1(q));          END MODULE;

MODULE  flip_flop;
DESCRIPTION  Automatically generated from HOL;
INPUTS  d, cp, cd;
OUTPUTS  q;
DEFINE
U1 (q, qbar) = FD2(d, cp, cd);
q = (U1(q));           END MODULE;
```

To date, the work on integrating formal verification into a design environment has moved from the design environment to the logic, through a form of schematic capture [NS88]. Some interesting new work at Cambridge [BGHvT90] attacks the problem by defining a semantics for the circuit description language and embedding the syntax and semantics of the design language into *HOL*.

In this chapter, we describe an implementation that translates *HOL* definitions into LSI netlists. This represents a first step towards integrating formal verification with the set of tools available to a working hardware designer. As an illustrative example, we considered the specification, proof, and translation to netlist of a shift register.

Chapter 8

Conclusions

The work reported in Chapters Two and Three of this thesis represents the consolidation of a line of research on theorem proving that has been in existence since the mid-1970's. The work in the rest of the thesis presents some results in new avenues of investigation: proof system verification; the intersection between the fields of Term Rewriting Systems and LCF-style proof; and the use of a proof system in CAD.

8.1 Implementation

hol90 is an improvement over hol88; it is more portable, more secure, more rational, and faster. It is now written in only one programming language: Standard ML, which is superior to ML in many respects:

- SML has a formal semantics (essential for total system correctness proofs).
- There are at least two high quality SML implementations available.
- In hol88, the encapsulation of *thm* was hidden in the implementation of ML and could not, therefore, be easily verified. Now the encapsulation is made explicit by the use of structures.
- SML has superior "programming-in-the-large" features.

Some implementation aspects of the system have been brought into focus, *e.g.*, exactly how the system is built; the details of the type inference algorithm; and how quotation and antiquotation are implemented.

The main, apparently mundane, advantage in the reimplementation is that the whole system has been re-examined and the results of that archaeology have been made public in this work. We have fixed up some problems and the *HOL* community can now apply its ingenuity in order to make the system even better.

8.2 The Prelogic

The identification of the prelogic and its dependencies is important, since it may become necessary to verify hol90. For example, if hol90 is used to validate a "critical" system where malfunction could lead to catastrophe, the correctness of hol90 itself becomes quite important. We do not have to go all the way to verification of the logic: the work done in specifying and describing the algorithms for α-conversion, substitution, and instantiation goes a long way towards clearing up the mystery of what they are about.

The verification of the logic depends on the correctness of the prelogic, as we have shown. For hol90, this correctness depends on the correctness of aconv, subst, and inst. We have made a start by verifying the algorithm for α-conversion by hand. We would like to see proofs of correctness for subst and inst.

A more satisfactory method of verification would be the following:

1. Implement the logic in a programming metalanguage (SML) as we have done.
2. Develop a theory of SML semantics in the logic.
3. Define the SML routines implementing *thm* in the logic.
4. Prove these *logical* routines correct in the logic against a specification of them in terms of (2).

In this way, the logic would be used to validate its implementation. At the moment, this is not possible, for item 2 above has not been done in *HOL*, and it would be a very large effort.

8.3 Term Rewriting Systems and Equational Unification

It was my goal to bring the fields of Term Rewriting Systems and LCF-style proof together to see if their confluence would be fruitful. The groundwork has been done, and some standard TRS algorithms have been expressed in the *HOL* framework. It is clear that the powerful idea of Knuth-Bendix completion has a role as a rule of inference. I expect also that AC-rewriting can be useful in easing the difficulty of doing some proofs, but this must be borne out by experimentation. Further development is needed to make these tools more widely usable.

8.4 Embedding into CAD Systems

The method presented in Chapter Five for using *HOL* in a CAD environment provides a bridge between the body of existing verifications in *HOL* and a CAD environment. One can move directly from a formal proof of a circuit to a foundry description of a large class of designs (gate arrays). This provides an interesting alternative to simulation, especially in view of the fact that most errors in commercial chips still occur at the functional level.

8.5 The Future of *HOL*

HOL suffers from some drawbacks:

- it is somewhat more complex than first order logic to implement correctly,

- [illegible]
- [illegible]
- [illegible]
- [illegible]

[illegible]

- [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible]

Looking at the pros and cons, it seems that the pros are fundamental, whereas the cons are mainly engineering problems that can be overcome with sufficient effort to [illegible] making the system more usable. I would like to see the following advances:

- [illegible] with a symbolic mathematics system like Maple or Mathematica.
- [illegible] interactive [illegible]. This [illegible] steps towards a intuitive solution to the user interface problem. Many of the [illegible] which is expressed in good notation. The system [illegible]

[illegible] mathematical data [illegible] computer simulations [illegible] the [illegible]

- The system also must support the proof needs of mathematicians. The current [illegible] detail is too small. It is worth noting that the use of tactics has [illegible] symbolic from its original competitor, which was to provide a language [illegible] [illegible] procedures for classes of formulas. Currently, tactics are used [illegible] a [illegible] manner to build proofs. More general [illegible] may be necessary.
- The embedding of a programming language in *HOL*. This raises interesting research issues of [illegible] optimization and reflection.
- Some performance in *HOL*. There are several reasons for this:
  - There [illegible] which I have implemented with a minimum of [illegible] of the system, which makes more challenging proof efforts feasible.
  - It may be possible to characterize the terms that the system generates or [illegible]. Much of the work is done in [illegible] first order [illegible]. If this is true, that should stimulate implementation of first order proof methods.
- Research into reflection principles for *HOL*. Many proof procedures [illegible] resolution, depend on meta-theorems. We need a way to utilize these theorems in the logic.
- More powerful polymorphic principles are very important, especially if the [illegible] can be automatically massaged to get some free theorems. The current [illegible] for [illegible] records types [Mel85] is a good example of this.

- Proofs are not formal entities in *HOL*. Elsa Gunter has suggested that retaining and studying proofs in the system is a interesting avenue of research in theorem proving.

Our ultimate goal should be to provide all the security and support needed by a computer scientist or mathematician in solving the formal aspects of a problem.

Appendix A

Syntactic unification

Here we review the fundamentals of basic unification theory. None of this material is new; however, the proofs are, in the main, my own.

A.1 Substitutions

A *substitution* is a partial mapping from variables to terms. In this paper, substitutions are represented by θ, σ, and γ. A substitution θ can be extended to be an endomorphism on terms by decreeing

- $\theta(v) = v$, for all v not in the domain of θ.
- $\theta(c) = c$, for all c in $\mathcal{C}$.
- $\theta(f(t_1, \ldots, t_n)) = f(\theta(t_1), \ldots, \theta(t_n))$.

A substitution can also be described by a finite set of bindings. A *binding* t/v (with *redex* v and *contractum* t) associates a member of $\mathcal{V}$ with a member of $\mathcal{F}$. A substitution θ is written

$$\theta = \{t_1/v_1, \ldots, t_n/v_n\}.$$

There are two restrictions on substitutions as finite sets of bindings: *identity* bindings (those of the form v_i/v_i) are not allowed, and no two bindings t_i/v_i, t_j/v_j may occur with $v_i = v_j$. The *length* of a substitution is the size of its set of bindings. As a finite binding, the null substitution ϵ, which has no effect on any term, is just $\emptyset$. The function *vars* returns the set of variables in a term. Two terms t_1 and t_2 are *identical*, written $t_1 \equiv t_2$ just in case they are the same term. The function *domain* returns the variables in the redexes of a substitution, i.e., $\{v_i : 1 \leq i \leq n\}$.

The function *range* returns the variables in the contracta of a substitution, i.e., $\bigcup\{\text{vars}(\theta(v_i)) : 1 \leq i \leq n\}$.

The *application* $\theta(t)$ of a substitution θ to a term t is defined to be the parallel replacement of occurrences of v_i in t by t_i. If there is an occurrence of a variable v in t such that there is no corresponding t/v binding in θ, then $\theta(t)$ effects no replacements on occurrences of v. For example,

$$\{y/x, 3/z, 2/y\}(f(x,y)) \equiv f(y,2).$$

A *solution* to an equation $t_1 = t_2$ is a substitution θ such that $\theta(t_1) \equiv \theta(t_2)$. This notion extends to equation sets: θ is a solution to an equation set $\{t_1 = t_2, \ldots, t_n = t_{n+1}\}$ if $\theta(t_i) \equiv \theta(t_{i+1})$, for i in $1 \ldots n$.

A.1.1 Composition of substitutions

If we think of substitutions as endomorphisms, substitution composition is just function composition, but often we deal with substitutions as finite sets of bindings. Given $\theta = \{t_1/v_1, \ldots, t_n/v_n\}$ and $\sigma = \{s_1/u_1, \ldots, s_m/u_m\}$, the *composition* of σ with θ, written $\sigma \circ \theta$ is the substitution obtained in the following manner:

$$\begin{aligned}
\text{let } \gamma \;=\; & \{(\sigma(t_1))/v_1, \ldots, (\sigma(t_n))/v_n\} \\
& \cup \\
& (\sigma \setminus \{s_j/u_j : \exists t_i/v_i \in \theta.\ u_j = v_i\}) \\
& \text{in} \\
& \gamma \setminus \{v_i/v_i : 1 \leq i \leq n\}
\end{aligned}$$

Example.

$$\begin{aligned}
\sigma &= \{a/x, b/y, y/z\} \\
\theta &= \{f(y)/x, z/y\} \\
\gamma &= \{f(b)/x, y/y\} \cup (\sigma \setminus \{a/x, b/y\})
\end{aligned}$$

$$\begin{aligned}
&= \{f(b)/x, y/y, y/z\} \\
\sigma \circ \theta &= \{f(b)/x, y/z\}
\end{aligned}$$

A.1.2 Properties of substitution composition

Substitutions are composable in isolation from the term being applied to. The proof for the composability property will be gone through in some detail because of its fundamental importance. Other useful properties of composition are associativity and idempotence.

Proposition 1. *Composability.*

$(\theta_2 \circ \theta_1)t \;\equiv\; \theta_2(\theta_1(t))$

Proof.

Since variables are the only part of a term affected by substitution application, and since application is done to all variables in parallel, we restrict our attention to an arbitrary variable v. There are four cases to consider:

1. $v \in \text{domain}(\theta_1)$ and $v \in \text{domain}(\theta_2)$.

$$\begin{aligned}
\theta_1 &= \{t_1/v_1, \ldots, t/v, \ldots, t_n/v_n\} \\
\theta_2 &= \{s_1/u_1, \ldots, s/v, \ldots, s_m/u_m\} \\
\theta_2 \circ \theta_1 &= (\{(\theta_2(t_1))/v_1, \ldots, (\theta_2(t))/v, \ldots, \theta_2(t_n)/v_n\} \\
&\quad \cup (\theta_2 \setminus \{s/v \text{ and others of no interest}\})) \setminus \{v_i/v_i\}
\end{aligned}$$

   Now there are two cases:

   (a) $\theta_2(t) \not\equiv v$. Notice that $t \equiv \theta_1(v)$. Then $(\theta_2 \circ \theta_1)(v) \equiv \theta_2(\theta_1(v))$.

   (b) $\theta_2(t) \equiv v$. In this case, $t/v \in \theta_1$ and $v/t \in \theta_2$, i.e., t is a variable. $\theta_2(t)/v = v/v$, which gets deleted from $\theta_2 \circ \theta_1$. Therefore, $(\theta_2 \circ \theta_1)(v) \equiv v$, by the definition of application. On the right hand side, $\theta_2(\theta_1(v)) \equiv \theta_2(t) \equiv v$.

2. $v \in \text{domain}(\theta_1)$ and $v \notin \text{domain}(\theta_2)$

$$\begin{aligned} \theta_1 &= \{t_1/v_1, \ldots, t/v, \ldots, t_n/v_n\} \\ \theta_2 &= \{s_1/u_1, \ldots, s_m/u_m\} \\ \theta_2 \circ \theta_1 &= (\{\theta_2(t_1)/v_1, \ldots, (\theta_2(t))/v, \ldots, \theta_2(t_n)/v_n\} \\ &\quad \cup (\theta_2 \setminus \{\text{some bindings of no interest}\})) \setminus \{v_i/v_i\} \end{aligned}$$

Exactly the same subcases arise as above, and are resolved in exactly the same manner.

3. $v \notin \text{domain}(\theta_1)$ and $v \in \text{domain}(\theta_2)$

$$\begin{aligned} \theta_1 &= \{t_1/v_1, \ldots, t_n/v_n\} \\ \theta_2 &= \{s_1/u_1, \ldots, s/v, \ldots, s_m/u_m\} \\ \theta_2 \circ \theta_1 &= (\{\theta_2(t_1)/v_1, \ldots, \theta_2(t_n)/v_n\} \\ &\quad \cup (\theta_2 \setminus \{\text{some bindings of no interest}\})) \setminus \{v_i/v_i\} \end{aligned}$$

Therefore $(\theta_2 \circ \theta_1)(v) \equiv s \equiv \theta_2(v) \equiv \theta_2(\theta_1(v))$.

4. $v \notin \text{domain}(\theta_1)$ and $v \notin \text{domain}(\theta_2)$

$$\theta_2 \circ \theta_1 = \text{some substitution } \gamma \text{ s.t. } v \notin \text{domain}(\gamma)$$

Therefore, $(\theta_2 \circ \theta_1)(v) \equiv v \equiv \theta_2(\theta_1(v))$. ∎

Proposition 2. *Associativity.*
$(\theta_3 \circ (\theta_2 \circ \theta_1))t \equiv ((\theta_3 \circ \theta_2) \circ \theta_1)t$
Proof.

The proof proceeds completely by composability.

$$\begin{aligned} \text{lhs} &= (\theta_3 \circ (\theta_2 \circ \theta_1))(t) \\ &= \theta_3((\theta_2 \circ \theta_1)(t)) \\ &= \theta_3(\theta_2(\theta_1(t))) \\ \text{rhs} &= ((\theta_3 \circ \theta_2) \circ \theta_1)(t) \\ &= (\theta_3 \circ \theta_2)(\theta_1(t)) \\ &= \theta_3(\theta_2(\theta_1(t))). \end{aligned}$$ ∎

Proposition 3. *Idempotency.*
$\theta \circ \theta = \theta$ iff $\text{domain}(\theta) \cap \text{range}(\theta) = \emptyset$
Proof.
($\Rightarrow$) Assume $\theta \circ \theta = \theta$. For contradiction, further assume $\text{domain}(\theta) \cap \text{range}(\theta) \neq \emptyset$, i.e., that there exist bindings t/v and v/x in θ. Then $\theta(\theta(x)) \equiv t \not\equiv \theta(x)$, which contradicts the initial assumption. Hence $\text{domain}(\theta) \cap \text{range}(\theta) = \emptyset$.
($\Leftarrow$) Assume $\text{domain}(\theta) \cap \text{range}(\theta) = \emptyset$. Then

$$\begin{aligned} \theta \circ \theta &= (\{\theta(t_1)/v_1, \ldots, \theta(t_n)/v_n\} \cup (\theta \setminus \theta)) \setminus \{v_i/v_i\} \\ &= \theta. \end{aligned}$$ ∎

A.2 Unifiers

A *unifier* is just a solution to an equation:

$$\text{unifier}(\theta, t_1 = t_2) \text{ iff } \theta(t_1) \equiv \theta(t_2).$$

Example.
Given $t_1 = f(a, g(y), x)$ and $t_2 = f(x, g(g(x)), z)$, $\theta = \{a/x, g(a)/y, a/z\}$ is a unifier of t_1 and t_2:

$$\theta(t_1) \equiv f(a, g(g(a)), a) \equiv \theta(t_2).$$

A *most general unifier* (mgu) is a unifier from which all other unifiers can be obtained by composition:

$$\forall t_1 t_2.\ \mathrm{mgu}(\theta, t_1 = t_2) \text{ iff } \forall \sigma.\ \mathrm{unifier}(\sigma, t_1 = t_2) \supset \exists \gamma.\ \sigma = \gamma \circ \theta.$$

An mgu is an interesting object because it is complete: *every* solution is an instance of the mgu.

Proposition 4. *Completeness.*
$\forall \sigma\theta.\ \mathrm{mgu}(\theta, t_1 = t_2) \wedge \mathrm{unifier}(\sigma, t_1 = t_2) \supset \exists \gamma.\ \sigma = \gamma \circ \theta.$

In general, there is not only one mgu for a pair of unifiable terms. The mgus *equivalent* to a given mgu are those unifiers derivable from it by composition with a *cyclic variable-pure* substitution. A cyclic variable pure substitution is a set of bindings $\theta = \{t_1/v_1, \ldots, t_n/v_n\}$ where all the t_i are variables, and there is a subset of θ, $\{t_i/v_i, t_{i+1}/t_i, \ldots, v_i/t_m\}$.

Example.
Given $f(x) = f(y)$, consider $\theta_1 = \{y/x\}$ and $\theta_2 = \{z/y, z/x, w/z, y/w\}$. Both θ_1 and θ_2 are mgu's of the equation, which means that both are derivable from each other by composition: $\{z/y, w/z, y/w\} \circ \theta_1 = \theta_2$ and $\{w/y, z/w, y/z\} \circ \theta_2 = \theta_1$.

Notice that θ_1 is idempotent; a most general unifier with this property has a length that is smaller than that of any non-idempotent mgu. Further, all idempotent mgus of an equation have the same length - they are all derivable from each other by a non-cyclic variable-pure substitution, or a *variable renaming*.

The notion of idempotent mgu is valuable since it provides a canonical way to represent all the solutions to an equality. Is it easy to compute? Fortunately, yes. There are many algorithms for computing an idempotent mgu of a set of equalities of members of $\mathcal{F}$. The fastest asymptotically is that of Paterson and Wegman[PW78], who propose a linear algorithm, but the fastest in practice seems to be that of

Martelli and Montanari [MM82]. Next we examine a simple unification algorithm from [MM82], from which they derive their fast algorithm.

A.3 A unification algorithm

The non-deterministic algorithm, `unify`, presented in Figure A.1, is phrased as a term rewriting system. It operates over a multiset of equalities E_0, terminating successfully with E_f when no rule applies to any element of the multiset, or unsuccessfully when the application of a rule returns Failure.

Terminology.
The context notation is used, so that we can talk about a particular equality in the multiset. For example, $E[a = b]$ "picks out" one $a = b$ equation in E. If we have $E[a = b] \longrightarrow E[b = a]$, then one occurrence of $a = b$ in E has been replaced by $b = a$. If we have $E[a = b] \longrightarrow E$, then one occurrence of $a = b$ has been deleted from E. Finally, if we have $E[a = b] \longrightarrow (\sigma(E))[a = b]$, it means apply σ to E without that particular occurrence of $a = b$, then add $a = b$ to the result. For example, if we have $E = \{x = 3, y = 2, x = 3\}$, then the rule $E[x = 3] \longrightarrow (\{x/3\}(E))[x = 3]$ applied to E would result in $\{y = 2, 3 = 3, x = 3\}$.

$E[f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)]$	$\longrightarrow$	$E[t_1 = s_1, \ldots, t_n = s_n]$
$E[f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)]$	$\longrightarrow$	Failure, if $f \neq g$
$E[x = x]$	$\longrightarrow$	E
$E[t = x]$	$\longrightarrow$	$E[x = t]$, provided $t \notin \mathcal{V}$.
$E[x = t]$	$\longrightarrow$	$(\{t/x\}(E))[x = t]$, if $x \in \mathrm{vars}(E)$, and $x \notin \mathrm{vars}(t)$ (if so, Fail), and $x \neq t$

Figure A.1: `unify` algorithm

Example.
(The entries in the second column are the rules chosen. An underlined equality is the one chosen for the next step.)

0. $\{\underline{g(x_2) = x_1}, f(x_1, h(x_1), x_2) = f(g(x_3), x_4, x_3)\}$
1. (4) $\{x_1 = g(x_2), \underline{f(x_1, h(x_1), x_2) = f(g(x_3), x_4, x_3)}\}$
2. (1) $\{x_1 = g(x_2), \underline{x_1 = g(x_3)}, h(x_1) = x_4, x_2 = x_3)\}$
3. (5) $\{\underline{g(x_3) = g(x_2)}, x_1 = g(x_3), h(g(x_3)) = x_4, x_2 = x_3)\}$
4. (1) $\{\underline{x_3 = x_2}, x_1 = g(x_3), h(g(x_3)) = x_4, x_2 = x_3)\}$
5. (5) $\{x_3 = x_2, x_1 = g(x_2), h(g(x_2)) = x_4, \underline{x_2 = x_2}\}$
6. (3) $\{x_3 = x_2, x_1 = g(x_2), \underline{h(g(x_2)) = x_4}\}$
7. (4) $\{x_3 = x_2, x_1 = g(x_2), x_4 = h(g(x_2))\}$

When the algorithm terminates successfully, as it does in the example, it leaves a *solved form* equation set, which has the following easily checked properties.

Proposition 5. *Solved form properties.*

1. all the left hand sides of the equations are variables;
2. all the left hand sides are distinct;
3. no left hand side occurs anywhere else in the equation set.

Hence, by Proposition Three and the finite binding notion of substitution, a solved form equation set is essentially an idempotent substitution. The function *mk_subst* does the trivial conversion. ∎

For example:

$$\text{mk\_subst}(unify(E_0)) = \{x_2/x_3, g(x_2)/x_1, h(g(x_2))/x_4\}.$$

The occurs check

The requirement in rule five that the variable x not be a subterm of t is known as the *occurs check* and is required to ensure correctness and termination. Suppose the unification algorithm without the occurs check is run with the following input: $\{x = f(x)\}$. This "algorithm" will terminate immediately with "unifier" $\{f(x)/x\}$, but applying this substitution to the input gives $\{f(x) = f(f(x))\}$, which is not an identity. Also, consider the input $\{x = f(x), y = x\}$. In this case the "algorithm" won't terminate. By this reasoning, no substitution with a binding of the form $\{f(\ldots\ x\ \ldots)/x\}$ can be a unifier.

(If we allow terms to be infinite, then $\{f(x)/x\}$ would make $\{f(x) = f(f(x))\}$ an equality provided x was an infinite term of the form $f(f(f(f\ldots)))$.)

A.4 Correctness of the unification algorithm

Proposition 6. *Termination.*
The unification algorithm terminates under all inputs.
Proof.

- Rule one decreases the number of symbols in the set.
- Rule two results in termination.
- Rule three reduces the number of equations, and therefore the number of symbols, in the set.
- Rule four does not decrease the number of equations, or the number of symbols, but it can only be applied a finite number of times before another rule must be chosen.

- Rules one to four are only applicable a finite number of times before termination occurs, failure occurs, or rule five must be tried. Rule five decreases the number of variables in the equations or fails. Since there are only a finite number of variables in the terms, the algorithm terminates. ∎

Proposition 7. `unify` *preserves solutions.*

unifier(θ, E_0) iff unifier$(\theta, \texttt{unify}(E_0))$

Proof.

The proof proceeds by induction on the number of steps in the computation. At any time, only one rule is applicable, if at all, to an equation in the multiset, so for each rule we show that it preserves solutions.

Base case.

1. $E_0 = E[f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)] \longrightarrow E[t_1 = s_1, \ldots, t_n = s_n]$

   ($\Rightarrow$) Assume that θ unifies E_0, so $\theta(f(t_1, \ldots, t_n)) \equiv \theta(f(s_1, \ldots, s_n))$. Since θ is an endomorphism, we have $f(\theta(t_1), \ldots, \theta(t_n)) \equiv f(\theta(s_1), \ldots, \theta(s_n))$. By the requirements of syntactic identity, $\theta(t_i) \equiv \theta(s_i)$, for i in $1 \ldots n$. Therefore, θ unifies $E[t_1 = s_1, \ldots, t_n = s_n]$.

   ($\Leftarrow$) Assume that θ unifies $E[t_1 = s_1, \ldots, t_n = s_n]$. By the requirements of syntactic identity and the endomorphism of θ, θ unifies E_0.

2. $E_0 = E[f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)] \longrightarrow$ Failure if $f \neq g$.

   ($\Rightarrow$) Since $f \neq g$, there is no unifier for E_0, and the algorithm fails, as required.

   ($\Leftarrow$) Since this is the first step, the algorithm has not introduced the selected equation, hence E_0 has no unifiers.

3. $E_0 = E[x = x] \longrightarrow E$

   Every substitution unifies $x = x$, so the unifiers of E are exactly those of E_0.

4. $E_0 = E[t = x] \longrightarrow E[x = t]$ (provided t $\notin \mathcal{V}$).

   This follows from the identity relation being reflexive.

5. $E_0 = E[x = t] \longrightarrow (\{t/x\}(E))[x = t]$

   We first consider the case where $x \in \text{vars}(E)$ This amounts to showing that, with respect to a unifier θ of $x = t$, $\theta(E) \equiv \theta(t/x(E))$. By Proposition One, this amounts to showing that $\theta = \theta \circ \{t/x\}$.

$$\begin{aligned} & \theta \circ \{t/x\} \\ = \; & (\{\theta(t)/x\} \cup (\theta \setminus \{\theta(x)/x\})) \setminus \{x/x\} \\ = \; & (\{\theta(x)/x\} \cup (\theta \setminus \{\theta(x)/x\})) \setminus \{x/x\} \\ = \; & \theta \end{aligned}$$

   The other case is the occurs check case, where $x \in \text{vars}(t)$. It is clear that this case produces Failure if and only if we detect an occurs check.

Inductive case. The inductive hypothesis is that unifiers have been preserved in the previous steps of the computation, i.e., unifier(θ, E_0) iff unifier(θ, E_k), where E_k is the result of k rewrites of E_0 by `unify`. Arguments 1,3,4,and 5 follow from the induction hypothesis and the corresponding argument in the base case.

1. $E_k = E[f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)] \longrightarrow E[t_1 = s_1, \ldots, t_n = s_n]$

2. $E_k = E[f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)] \longrightarrow$ Failure if $f \neq g$.

   ($\Rightarrow$) The condition on applying this rule ensures that E_k is not unifiable. The inductive hypothesis ensures that E_0 is not unifiable, and the algorithm fails, as required.

   ($\Leftarrow$) Suppose the algorithm fails. Since the inductive hypothesis assures us that all unifiers have been preserved, then E_0 was not unifiable.

3. $E_k = E[x = x] \longrightarrow E$

4. $E_k = E[t = x] \longrightarrow E[x = t]$ (provided t $\notin \mathcal{V}$).

5. $E_k = E[x = t] \longrightarrow (\{t/x\}(E))[x = t]$

∎

Henceforth, mk_subst(**unify**(E_0)) will be represented by Θ.

Proposition 8. *Solved form unifies itself.*

unifier(Θ, E_f)

Proof.

If E_f is $\{v_1 = t_1, \ldots, v_n = t_n\}$, then Θ is $\{t_1/v_1, \ldots, t_n/v_n\}$. By Proposition Six, Θ is idempotent, so $\Theta(E_f)$ is $\{t_1 = t_1, \ldots, t_n = t_n\}$. ∎

Corollary 9. *Solved form unifies the initial equation multiset.*

unifier(Θ, E_0)

Proof.

This follows from Propositions Seven and Eight. ∎

Proposition 10. *Solved form is an idempotent mgu.*

mgu(Θ, E_0)

Proof.

First, Corollary Nine says that the result of **unify** is a unifier. We can show that this unifier is most general if we can show $\forall\sigma.$ unifier$(\sigma, E_0) \supset \exists\gamma.\ \sigma = \gamma \circ \Theta$. Assume that σ is a unifier of E_0. It remains to show $\exists\gamma.\ \sigma = \gamma \circ \Theta$. We do this by showing $\forall v.\ \sigma(v) = (\sigma \circ \Theta)v$. There are two cases to consider:

$v \in$ domain(Θ). Then there is a binding v/t in Θ. By Proposition Seven, σ unifies E_f, so $\sigma(v) \equiv \sigma(t)$. By the occurs check argument, v cannot be a subterm of t. Now, by Proposition Eight, $t \equiv \Theta(v)$, and $v \notin$ vars(t) (by idempotence), so $\sigma(v) \equiv \sigma(\Theta(v)) \equiv (\sigma \circ \Theta)v$.

$v \notin$ domain(Θ). Then $\Theta(v) = v$, so $\sigma(\Theta(v)) = \sigma(v)$, so $\sigma(v) = (\sigma \circ \Theta)v$. ∎

Proposition 11. *Correctness.*

If two terms are unifiable, the unification algorithm computes an idempotent mgu; otherwise, it fails.

Proof.

For the unifiable case, the unification algorithm computes solved form, by definition. Solved form is an idempotent mgu, by Proposition Ten.

For the non-unifiable case, Proposition Seven shows that, if two terms are not unifiable, then Failure will be returned. ∎

A.5 Other facets of unification

As has been shown, unification is not really straightforward; close examination of the details is necessary in order to make the proofs go through. This is not surprising, since a substitution is nothing more than a parallel assignment statement, and a non-idempotent substitution is a side-effecting parallel assignment statement.

In spite of that, basic unification is fairly well understood and current research looks to extend the notion of unification. There are several avenues of study:

- Unification of terms modulo an equational theory.
- Combination of existing unification algorithms.
- Unification of systems of equalities and inequalities.
- The complexity of unification.

- Unification of higher order terms.
- Models for unification.

References

[AHM89] Arnon Avron, Furio Honsell, and Ian Mason. An overview of the Edinbugh Logical Framework. In Graham Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer-Verlag, 1989.

[AK90] Mohamed Adi and Claude Kirchner. AC-unification race: The system solving approach and its implementation. In A. Miola, editor, *Proceedings Design and Implementation of Symbolic Computation Systems: International Symposium DISCO '90 (LNCS 429)*, pages 174–183, Capri, Italy, April 1990.

[And81] Peter Andrews. Theorem proving via general matings. *Journal of the Association for Computing Machinery*, 28(2):193–214, April 1981.

[And86] Peter Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press, 1986.

[Art90] R.D. Arthan. A formal specification of HOL. Technical Report DS/FMU/IED/SPC001, ICL Defence Systems, April 1990.

[Bac88] Leo Bachmair. Proof by consistency in equational theories. In *Proceedings of the Third Annual Symposium on Logic in Computer Science*, pages 228–233, Edinburgh, Scotland, July 1988. Computer Society Press.

[BCD90] A. Boudet, E. Contejean, and H. Devie. A new AC unification algorithm with an algorithm for solving systems of Diophantine equations. In *Proceedings of the Fifth Annual Symposium on Logic in Computer*

Science, pages 289–299, Philadelphia, Pennsylvania, June 1990. Computer Society Press.

[BDH86] Leo Bachmair, Nachum Dershowitz, and Jeh Hsiang. Orderings for equational proofs. In *Proceedings of the Third Annual Symposium on Logic in Computer Science*, pages 346–357, 1986.

[BGHvT90] Richard Boulton, Mike Gordon, John Herbert, and John van Tassel. The HOL verification of ELLA designs. To appear at the 1991 Workshop on Formal Methods in VLSI Design, 1990.

[BHK+88] Hans-Juergen Buerckert, Alexander Herold, Deepak Kapur, Jorg Siekmann, Mark Stickel, Michael Tepp, and Hantao Zhang. Opening the AC-unification race. Technical Report SR-88-11, SEKI, July, 1988.

[Bir35] Garrett Birkhoff. On the structure of abstract algebras. *Proceedings of the Cambridge Philosophical Society*, 31:433–454, 1935.

[BKN85] Dan Benavav, Deepak Kapur, and Paliath Narendran. Complexity of matching problems. In Jean-Pierre Jouannaud, editor, *Proceeding of the First International Conference on Rewriting Techniques and Applications (LNCS 202)*, pages 415–429, Dijon, France, May 1985. Springer-Verlag.

[BM79] Robert S. Boyer and J. Strother Moore. *A Computational Logic*. Academic Press, 1979.

[Buc85] Bruno Buchberger. Grobner: an algorithmic method in polynomial ideal theory. In N.K. Bose, editor, *Multidimensional Systems Theory*. Dordrecht Reidel, 1985.

[CAB+86] Robert Constable, S. Allen, H. Bromly, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden,

J. Sasaki, and S. Smith. *Implementing Mathematics With the Nuprl Proof Development System*. Prentice-Hall, New Jersey, 1986.

[CF58] Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume 1. North-Holland, 1958. Two sections by William Craig.

[CH88] Thierry Coquand and Gerard Huet. The Calculus of Constructions. *Information and Computation*, 76(2-3), February-March 1988.

[Chu36] Alonzo Church. A note on the *entscheidungsproblem*. *Journal of Symbolic Logic*, 1:40–41, 1936.

[Chu40] Alonzo Church. A formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[CL73] Chin-Liang Chang and Richard Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York, 1973.

[Coh88] Avra Cohn. A proof of correctness of the Viper microprocessor: The first level. In Graham Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, pages 27–71. Kluwer Academic Publishers, 1988.

[Coh89] Avra Cohn. Correctness of the Viper block model: The second level. In Graham Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 1–91. Springer-Verlag, 1989.

[Cur86] Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms, and Functional Programming*. Research Notes in Theoretical Computer Science. Pitman, 1986.

[dB72] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indag. Math.*, (34):381–392, 1972.

[Der87] Nachum Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.

[Fag84] Francois Fages. Associative-commutative unification. In Robert Shostak, editor, *Proceedings of the Seventh International Conference on Automated Deduction (LNCS 170)*, pages 194–208, Napa, California, USA, May 1984. Springer-Verlag.

[Fel86] Amy Felty. Using extended tactics to do proof transformations. Master's thesis, University of Pennsylvania, Department of Computer and Information Science, 1986.

[For85] Albrecht Fortenbacher. An algebraic approach to unification under associativity and commutativity. In Jean-Pierre Jouannaud, editor, *Proceeding of the First International Conference on Rewriting Techniques and Applications (LNCS 202)*, pages 381–397, Dijon, France, May 1985. Springer-Verlag.

[Fre67] Gottlob Frege. *Begriffsschrift*, a formula language, modeled upon that of arithmetic, for pure thought. In Jean van Heijenoort, editor, *From Frege to Godel: A Source Book in Mathematical Logic 1879-1931*. Harvard University Press, 1967.

[GB89] B. Graham and G. Birtwistle. Formalising the design of an SECD chip. In M. Leeser and G. Brown, editors, *Proceedings of a Workshop on Hardware Specification, Verification, and Synthesis: Mathematical Aspects (LNCS 408)*, Cornell, 1989.

[Gen69] Gerhard Gentzen. Investigations into logical deduction. In M. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–128. North-Holland, 1969.

[GMW79] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[God67a] Kurt Godel. The completeness of the axioms of the functional calculus of logic. In Jean van Heijenoort, editor, *From Frege to Godel: A Source Book in Mathematical Logic 1879-1931*, pages 582–592. Harvard University Press, 1967.

[God67b] Kurt Godel. On formally undecidable propositions of Principia Mathematica and related systems I. In Jean van Heijenoort, editor, *From Frege to Godel: A Source Book in Mathematical Logic 1879-1931*, pages 592–618. Harvard University Press, 1967.

[Gor82] Michael Gordon. Representing a logic in the LCF metalanguage. In D. N'eel, editor, *Tools and Notions for Program Construction*, pages 163–185. Cambridge University Press, 1982.

[Gor85] Mike Gordon. Why Higher-Order Logic is a good formalism for specifying and verifying hardware. Technical Report 77, Computer Laboratory, The University of Cambridge, 1985.

[Gor88] Mike Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis, Workshop Proceedings*. Kluwer, Calgary, January 1988.

[Gor89] Mike Gordon. *The HOL System: Description.* Cambridge Research Center, SRI International, 1989.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types.* Cambridge University Press, 1989.

[Gun89] Elsa Gunter. Doing algebra in simple type theory. Technical Report MS-CIS-89-38, Logic & Computation 09, Department of Computer and Information Science, University of Pennsylvania, 1989.

[HD83] Jeh Hsiang and Nachum Dershowitz. Rewrite methods for clausal and non-clausal theorem proving. In *Proceeding of the 10th ICALP (LNCS 154)*, pages 331–346. Springer-Verlag, July 1983.

[HH82] Gerard Huet and Jean-Marie Hullot. Proofs by induction in equational theories with constructors. *Journal of Computer and System and Sciences*, 25:239–266, 1982.

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Proceedings of the Second Annual Conference on Logic in Computer Science*, Ithaca, New York, USA, 1987. Computer Society Press.

[HO80] Gerard Huet and Derek Oppen. Equations and rewrite rules: A survey. In Ronald Book, editor, *Formal Language Theory: Perspectives and Open Problems.* Academic Press, 1980.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1 of *London Mathematical Society Student Texts.* Cambridge University Press, 1986.

[Hue73] Gerard Huet. A mechanization of type theory. In *Proceedings of the Third International Joint Conference on Artificial Intelligence*, pages 139–146, 1973.

[Hue75] Gerard Huet. A unification algorithm for typed lambda calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Hue78] Gerard Huet. An algorithm to generate the basis of solutions to homogeneous linear Diophantine equations. *Information Processing Letters*, 7(3):144–147, April 1978.

[Hue80] Gerard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the Association for Computing Machinery*, 27(4):797–821, October 1980.

[Hue81] Gerard Huet. A complete proof of correctness of the Knuth-Bendix completion algorithm. *Journal of Computer and System Sciences*, 21:11–21, 1981.

[Hue86] Gerard Huet. Formal structures for computation and deduction. Class notes for a course at CMU, 1986.

[JK90] Jean-Pierre Jouannaud and Claude Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. Technical Report 561, Unite' Associee' au CNRS UA 410, March 1990.

[KB70] Donald Knuth and Peter Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra.* Pergamon Press, Oxford, 1970.

[KL80] S. Kamin and J.J. Levy. Two generalizations of the recursive path ordering. Technical report, University of Illinois, Urbana, Illinois, 1980. Unpublished.

[KM88] Jan Willem Klop and Aart Middeldorp. An introduction to Knuth-Bendix completion. Quarterly 3, CWI, September 1988.

[KN84] Deepak Kapur and Paliath Narendran. An equational approach to theorem proving in first order predicate calculus. Technical report, Computer Science Branch, General Electric Company, Schenectady, New York, 1984.

[LPT89] Z. Luo, R. Pollack, and P. Taylor. How to use LEGO (a preliminary user's manual). Technical Report LFCS-TN-27, Department of Computer Science, Edinburgh University, 1989.

[lsi86] *Databook and Design Manual: HCMOS Macrocells and Macrofunctions.* LSI Logic Corporation, 1986.

[Mel89] Tom Melham. Automating recursive type definitions in higher order logic. In Graham Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer-Verlag, 1989.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mil83] Dale Miller. *Proofs in Higher Order Logic.* PhD thesis, Carnegie-Mellon University, 1983.

[Mil85] Robin Milner. The use of machines to assist in rigorous proof. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, International Series in Computer Science, pages 77–87. Prentice-Hall, 1985.

[Mil87] Dale Miller. A compact representation of proofs. *Studia Logica*, 46(4):347–370, 1987.

[ML85] Per Martin-Lof. Constructive mathematics and computer programming. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, Prentice-Hall International Series in Computer Science, pages 167–184. Prentice-Hall, 1985.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[Mus80] David Musser. On proving inductive properties of abstract data types. In *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, Las Vegas, Nevada, USA, January 1980.

[ndl85] *Network Description Language Reference.* LSI Logic Corporation, 1985.

[New42] M.H.A. Newman. On theories with a combinatorial definition of equivalence. *Annals of Mathematics*, pages 223–243, 1942.

[Nip89a] Tobias Nipkow. Equational reasoning in Isabelle. *Science of Computer Programming*, 12:123–149, 1989.

[Nip89b] Tobias Nipkow. Term rewriting and beyond—theorem proving in Isabelle. *Formal Aspects of Computing*, 1:320–338, 1989.

[Nip90] Tobias Nipkow. A critical pair lemma for higher-order rewrite systems and its application to $\lambda*$. In Gerard Huet and Gordon Plotkin, editors, *Proceedings of the First Workshop on Logical Frameworks*, pages 361–376, Antibes, France, May 1990.

[NO79] Greg Nelson and Derek Oppen. Simplification by cooperating decision procedures. *Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[NO80] Greg Nelson and Derek Oppen. Fast decision procedures based on congruence closure. *Journal of the Association for Computing Machinery*, 27(2):356–364, April 1980.

[NS88] P. Narendran and J. Stillman. Hardware verification in the interactive VHDL workstation. In G. Birtwistle and P. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis, Workshop Proceedings*. Kluwer, Calgary, Alberta, 1988.

[Pau83] Lawrence Paulson. A higher order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[Pau87] Lawrence Paulson. *Logic and Computation: Interactive Proof With Cambridge LCF*. Cambridge University Press, 1987.

[Pau90] Lawrence Paulson. Isabelle: The next 700 theorem provers. In P.G. Oddifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[Pie73] T. Pietrzykowski. A complete mechanization of second order type theory. *Journal of the Association for Computing Machinery*, 20(2):333–365, April 1973.

[Pit90] Andy Pitts. Semantics of programming languages. Course notes for Computer Science Tripos, Part 2, University of Cambridge, 1990.

[Plo72] Gordon Plotkin. Building-in equational theories. *Machine Intelligence*, (7):73–90, 1972.

[PS81] Gary Peterson and Mark Stickel. Complete sets of reductions for some equational theories. *Journal of the Association for Computing Machinery*, 28:233–264, 1981.

[PS89] William Pase and Mark Saaltink. Formal verification in m-EVES. In Graham Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer-Verlag, 1989.

[PW78] M.S. Paterson and M.N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12:23–41, 1965.

[Rus67] Bertrand Russell. Mathematical logic as based on the theory of types. In Jean van Heijenoort, editor, *From Frege to Godel: A Source Book in Mathematical Logic 1879-1931*. Harvard University Press, 1967.

[Sho79] Robert Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the Association for Computing Machinery*, 26(2):351–360, April 1979.

[Sho84] Robert Shostak. Deciding combinations of theories. *Journal of the Association for Computing Machinery*, 31(1):1–12, January 1984.

[Sti81] Mark Stickel. A unification algorithm for associative-commutative functions. *Journal of the Association for Computing Machinery*, 28(3):423–434, July 1981.

[Tur85] D.A. Turner. Programs as executable specifications. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming*

Languages, International Series in Computer Science, pages 29–50. Prentice-Hall, 1985.

[Wal90] Lincoln Wallen. ***Automated Proof Search in Non-Clasical Logics:*** *Efficient Matrix Proof Methods for Modal and Intuitionistic Logics.* MIT Press, Cambridge, Massachusetts, 1990.

[WH81] P.H. Winston and B.K.P Horn. *LISP.* Addison Wesley, 1981.

[Wos88] Larry Wos. ***Automated Reasoning:*** *33 Basic Research Problems.* Prentice-Hall, New Jersey, 1988.