Self-Checking Prover Study
Final Report

M. J. C. Gordon, J. M. J. Herbert, R. W. S. Hale
SRI International
Cambridge Computer Science Research Centre

J. Harrison, W. Wong, J. von Wright
University of Cambridge
Computer Laboratory

Contents

Part I

Project Overview

The Project

The Self-Checking Prover Study is Task 2 of the Foundations of Secure Formal Methods project. The main goal of this task was the design and development of a prover/checker pair for the HOL theorem proving system; that is, a version of the HOL prover modified to record and output the proofs that are made in the system, and a companion checker to determine whether such proofs are logically sound.

The prover/checker pair responds to the need for a simple independent test of the trustworthiness of claimed proofs. Such a test becomes more important as mechanically-produced proofs about computer systems become more common, and as the proof procedures used become less easily inspected. It would, for example, enable verification work to be subcontracted with greater confidence as the resultant proofs could easily be checked by the principal.

The research at SRI Cambridge has developed an independent proof checker based on the HOL system, but using techniques that are applicable to similar systems (e.g. Ehdm/PVS). The HOL system can now provide a full trace of the inferences used in a proof, and an independent proof checking program can then be used to check each inference. The core algorithm used in the proof checking has been verified with respect to a formalization of the theory of HOL proofs.

The work at SRI Cambridge has been carried out under the technical direction of Dr. Mike Gordon and Dr. John Herbert. Dr. Gordon also had a similar, but more foundational, project supported by the UK government at the University of Cambridge Computer Laboratory and the resulting co-operation and cross fertilization of ideas between the two projects has been highly productive.

Both projects were based on initial work at SRI, with much of the supporting theoretical work being done at the Computer Laboratory. Subsequent work at SRI concentrated on development of the theoretical work and integration of the theory of proofs with the proof checker. As a result of this co-operation, more has been accomplished than would otherwise have been possible. In particular, verification of the proof checker was much more thorough than had been envisaged and the checker itself is much more capable.

The remainder of this report includes an overview of the research and the final deliverables funded on the SRI project. Some of these build on results of the Computer Laboratory project and the contributions of Mr. John Harrison, Dr. Wai Wong and Dr. Joakim von Wright of the University of Cambridge Computer Laboratory are gratefully acknowledged.

Research tasks

The research was divided into the five tasks below. All have been completed.

- *A format for transferring* HOL *proofs to a separate checker.*

A proof format has been designed and documented that is both simple and portable. The format follows the standard notion of a proof as a linear sequences of lines, where each line is either an axiom or is deduced from earlier lines by a rule of inference (*a la* Gödel and Hilbert). Portability is achieved by using a simple ASCII representation.

- *Modification of the* HOL *system to generate proof texts.*

The HOL system has been modified to generate proof texts as a side effect of its normal operation. This feature is switchable and only needs to be used for the final version of a proof. The modifications are included as a library package in v.2.02 of the standard HOL88 system.

The proof texts can be very large. For example, a medium-sized proof, such as the HOL multiplier benchmark which generates 14,500 intermediate theorems produces a proof file of 43.6Mb. However, this is reduced to 0.7Mb by compression.

- *Specification, implementation and verification of a prototype proof checker.*

The theory of proofs in HOL has been fully formalized and is available in the HOL88 distribution. The specification of the checker follows from this.

The proof checking algorithm, specified in a generic algorithm language, has been formally verified using the HOL theory of proofs. It has also been translated automatically into an implementation in the Standard ML programming language (SML).

The simple proof checker has also been "implemented" in a simple imperative programming language defined in HOL and this implementation has been formally verified in HOL.

A more comprehensive and efficient, but unverified, version of the checker in SML is also available (due to Wai Wong). The main problem in checking real proofs is the sheer size of the generated proof texts and this checker is carefully designed, working in two passes and using compression, to alleviate the effects of size. We have incorporated the SML translation of the verified algorithm as the core checking function in a less efficient version of this checker.

- *Experimental testing of the proposed proof format and prototype checker.*

A number of small to medium sized proofs have been successfully checked using the unverified checker, including the multiplier benchmark that is distributed with HOL. However, the verified checker has only been tested on very small examples to demonstrate that it works.

• *Investigation of the use of decision procedures in the* HOL *framework.*[1]

There are two principal ways to incorporate decision procedures securely in a general-purpose prover such as HOL. The most straightforward is to encode the appropriate algorithms as performance-optimized derived inference rules. Another approach is 'reflection', where the theorem prover is used to prove that bits of code, which might implement a decision procedure, can be added to itself soundly.

We had thought that reflection might offer a way to incorporate highly efficient decision procedures soundly into HOL. However, a careful study of the issues has led us to conclude that, although inuitively very appealing, it is very hard to use reflection securely in practice (there are few examples of its proper use in other provers). Moreover, it appears to be possible to implement quite useable decision procedures as derived rules by careful programming; frequently it is proof search that is the really time consuming part of the overall proof, and it's not critical that this is implemented securely.

Timescales and Deliverables

The proposal detailed the following deliverables:

1. *Half way through the project:*
   - (a) *A report describing the proof text format chosen for* HOL.
   - (b) *A report documenting the amendments required to the* HOL *system to support the generation of proof texts.*

These two reports have already been delivered, but are included in part II of this report for completeness.

2. *At the end of the project:*
   - (a) *A report documenting the proof checker.*
   - (b) *A report describing the correctness condition for the proof checker and progress made towards the verification of the condition in* HOL.
   - (c) *A report on the feasibility of incorporating reflection into the* HOL *system.*

[1] This work-package was to be carried out by Dr. Pitts of the University of Cambridge, who is an eminent theoretician. When it became clear that Dr. Pitts would not be able to complete this task in time, a more practical study on the same lines was undertaken.

(d) *A Sun compatible cartridge tape containing the prover/checker pair based on a modified* HOL *system.*

Items (a), (b) and (c) are assembled in this report.

Part III, "A Verified Proof Checking Algorithm and Its Implementation", describes the proof checker. Chapter 3 introduces the algorithm description language, AL, which is used to specify the proof checking algorithm. This gives an independent description relating Joakim von Wright's theory of proofs in HOL [vW94b] to an implementable proof checking function. The specification and verification of the checker in AL are described in Chapter 4, which also describes the translation to an SML implementation.

Part IV, "A Verified Imperative Proof Checker for HOL", reports on the verification of a version of the proof checker written in a small imperative language that has been formalised in HOL. Chapter 5 describes a methodology for formal verification of modular imperative programs in HOL (due to Joakim von Wright [vW94c]). Chapter 6 describes an interface to the program verification theory, implemented by Mike Gordon, which permits the user to enter procedures in a form close to ordinary Pascal-style syntax. Chapter 7 describes the imperative proof checker and its verification and Chapter 8 describes a simple implementation using an imperative fragment of ML.

Part V, "Metatheory and Reflection in Theorem Proving", reviews the uses of reflection and the discusses the merits and feasibility of incorporating reflection into systems such as HOL. Metatheory and reflection are often suggested as ways to improve the efficiency of computer theorem proving procedures. Part V, due to John Harrison, looks at exactly what is meant by "reflection" in this context, contrasts it with other approaches, and summarizes existing practice. The final chapter contains an assessment of whether incorporating reflection into the HOL system is likely to be of practical value.

Summary/Conclusions

We have successfully designed, implemented and formally verified a prover/-checker pair for the HOL system. The system is rudimentary but useable and has been tested on small-scale proofs.

A number of directions directions for futher research are suggested by this work:

- The proof format is at a sufficiently low level that it might form the basis of a format for porting proofs between different theorem provers. This would permit proofs done in one theorem prover to be imported into another. Different underlying formalisms would present problems, but it should, at least, be possible to develop a generic proof format for the interchange of proofs between higher-order logic provers.

- Although the HOL proof checker is reasonably efficient, it has not been tested on large proofs.

  There are various ways in which the scope and efficiency of the checker could be improved, e.g. by working with higher level proof steps, but this would need to be done securely.

- The study of decision procedures and reflection has led to the hypothesis that secure decision procedures can be implemented efficiently as derived inference rules, but there is a need for more empirical evidence to support this. Few decision procedures have been implemented in HOL and there is insufficient knowledge of how to do it well.

Part II

Recording HOL Proofs

Chapter 1

A HOL Proof File Format

1.1 Introduction

This chapter describes a proof file format for HOL. The purpose of this format is to facilitate the interchange of information between HOL and other automatic reasoning systems, such as proof checkers and other theorem provers. The capability of generating proof files in this format is incorporated into a new version of the HOL system; the implementation details are documented in the next chapter.

This proof format allows a proof generated by the HOL system to be checked independently. Intended application areas for this technology include safety-critical and security applications, where the correctness of a design should be clearly demonstrable.

The proof files written in the format described here are intended for automatic checking. Therefore, the following two considerations have influenced the design of the format:

- the format should contain only ASCII text characters to make the transfer of proof files between machines easier;
- the format should be simple so that it can be parsed easily.

To make the format simpler, a linear model, i.e. Hilbert style rather than tree model, has been chosen. A proof in such a style consists of a number of lines. Each line is a step in a chain of inferences, and it consists of a line number, a list of theorems and a justification. The theorems in a line are the results of the inference step while the justification tells which inference rule was applied

1.	$\vdash m+n=n+m$	`Hypothesis`
2.	$\vdash \mathsf{SUC}(m+n)=\mathsf{SUC}(m+n)$	`REFL`
3.	$\vdash \mathsf{SUC}(m+n)=\mathsf{SUC}(n+m)$	`SUBST 1,2`

Figure 1.1: A simple proof

to obtain the results. The justification may refer to any previous lines by their numbers.

Figure 1.1 shows a simple proof. There are three lines in this proof. They are numbered 1 to 3. The theorem part of each line consists of only a single theorem. The first line is the hypothesis of the proof, its theorem is the initial theorem of the chain of inferences. The justification of the second line is `REFL`, a primitive inference rule which returns an equation when called with a term. Both sides of the equation are the term given to the rule. The justification of the last line is `SUBST`, a substitution in which some subterm in the theorem of Line 2 matching the left hand side of the theorem in Line 1 is substituted by the right hand side of this theorem. It refers to two previous lines by their line numbers.

Simple proofs can be carried out in HOL by calling inference rules directly in the way explained above. This is known as *forward* proof, but very often proofs are more complicated. A more powerful method of proof, known as *goal-directed* proof, or *tactical* proof will be used. In this proof style, a term in the same form as the required theorem is set up as a goal, tactics are used to reduce the goal to simpler subgoals recursively until all the subgoal are resolved. In such a proof, the user does not call the inference rules directly. However, a correct sequence of inferences is calculated and performed by the system behind the scenes to derive the theorem. Therefore, a proof can be described as a chain of inferences in either style.

1.2 The syntax

The syntax of the proof file format for HOL is based on LISP S-expressions. Objects, such as lines, theorems, terms and so on, are enclosed in a pair of matching parentheses. The first atom in an object is a tag specifying what kind of object it is. A file in this format may consist of one or more proofs. This section specifies the concrete syntax of the format, i.e. the sequence of characters which appears in the file. The meanings of the expressions will be explained in Section 1.3.

The syntax of the proof format is specified in an augmented BNF. The conventions used in this augmented BNF are:

TERMINAL	STRING	TERMINAL	STRING
PROOF	PROOF	LINE	LINE
THM	THM		
VAR	V	CONST	C
APP	A	ABS	L
TYVAR	TV	TYOP	TO
LP	(	RP	)

Figure 1.2: Input strings for terminals

- The terminal symbols are in upper case. The actual strings of the terminals which appear in the file are given in Figure 1.2.
- The token `NUMBER` is a string of one or more digits (`0`...`9`) and optionally prefixed by a minus sign(`-`).
- The token `STRING` is a string of characters which denotes names of variables, constants, and so on. The characters allowed in different expressions will be detailed in Section 1.3 when the expressions are explained.
- The non-terminal symbols are in lower case.
- A non-terminal symbol whose name ends with the suffix '`_list`' stands for a list of similar elements. A list is enclosed by a pair of square brackets, e.g., '`[...]`'. A list may be empty, in such case, there will be nothing in between the square brackets. The production rule for `xxx` then specifies the syntax of each element of the list `xxx_list`. Since each element is enclosed in a pair of parentheses, no element separator is needed.
- In the rule for `Justification`, the first terminal of each justification is its name. The input string for this is the same as the name but the case of the characters is ignored.
- the start symbol is `proof`.

The syntax of the HOL proof file format is as below:

```
proof ::= LP PROOF STRING line_list RP ;;

line ::= LP LINE NUMBER LP justification RP thm_list RP ;;

thm ::= LP THM term_list term RP ;;

term ::= LP VAR STRING type RP
      |  LP CONST STRING type RP
      |  LP APP term term RP
```

```
      |  LP ABS term term RP ;;

type ::= LP TYVAR STRING RP
      |  LP TYOP STRING type_list RP ;;

justification ::= HYPOTHESIS
     | ASSUME term
     | REFL term
     | SUBST number_term_list term NUMBER
     | BETACONV term
     | ABS term NUMBER
     | INSTTYPE type_type_list NUMBER
     | DISCH term NUMBER
     | MP NUMBER NUMBER
     | MKCOMB NUMBER NUMBER
     | MKABS NUMBER
     | ALPHA term term
     | ADDASSUM term NUMBER
     | SYM NUMBER
     | TRANS NUMBER NUMBER
     | IMPTRANS NUMBER NUMBER
     | APTERM term NUMBER
     | APTHM NUMBER term
     | EQMP NUMBER NUMBER
     | EQIMPRULE NUMBER
     | SPEC term NUMBER
     | EQTINTRO NUMBER
     | GEN term NUMBER
     | ETACONV term
     | EXT NUMBER
     | EXISTS LP term term RP NUMBER
     | CHOOSE LP term NUMBER RP NUMBER
     | IMPANTISYMRULE NUMBER NUMBER
     | MKEXISTS NUMBER
     | SUBS NUMBER list NUMBER
     | SUBSOCCS number_list_number_list NUMBER
     | SUBSTCONV number_term_list term term
     | CONJ NUMBER NUMBER
     | CONJUNCT1 NUMBER
     | CONJUNCT2 NUMBER
     | DISJ1 NUMBER term
     | DISJ2 term NUMBER
     | DISJCASES NUMBER NUMBER NUMBER
     | NOTINTRO NUMBER
```

```
      | NOTELIM NUMBER
      | CONTR term NUMBER
      | CCONTR term NUMBER
      | INST term_term_list NUMBER
      | STOREDEFINITION STRING term
      | DEFINITION STRING STRING
      | DEFEXISTSRULE term
      | NEWAXIOM STRING term
      | AXIOM STRING STRING
      | THEOREM STRING STRING
      | NEWCONSTANT STRING type
      | NEWTYPE NUMBER STRING
      | NUMCONV term;;

number_term ::= LP NUMBER term RP ;;

type_type ::= LP type type RP ;;

number_list_number  ::= LP NUMBER_list NUMBER RP ;;

term_term ::= LP term term RP ;;
```

Space between tokens is optional except where its absence will result in ambiguity. For example, a space is required between the tag `C` and the string `T` and between the tag `TO` and the string `bool` in the constant expression below:

```
  (C T(TO bool[]))
```

The space character (ASCII code 32), tab (9), carriage return (13) and line feed (10) are considered to be space. Consecutive spaces are treated as a single space.

1.3 The semantics

The semantics of the proof file is described in this section in a bottom-up fashion starting with the rule for types.

1.3.1 Types

HOL types are represented in the proof file by type expressions. There are two kinds of types: type variables and type operators. A type variable is represented by the `TYVAR` expression. The string in this expression is the name of the variable. In HOL88, all type variable names begin with the character '*'. This should be checked when the proof is parsed. For example, the expression `(tv **)` represents the HOL type $: **$.

The `TYOP` expression represents both type constants and type operators in HOL. The string in this expression is the name of the type operator and the `type_list` is the list of arguments of the type operator. A type constant is considered as a type operator taking no argument. For example, the HOL built-in type : *bool* is represented as `(TO bool [])`. The names of the HOL built-in infix type operators $\rightarrow$, $\#$ and $+$ are `fun`, `prod` and `sum`, respectively. The examples below show some HOL types in the proof format:

`(TO fun [(TV *)(TV *)])`	$: * \rightarrow *$
`(TO prod [(TO num[])(TO bool[])])`	$: num\#bool$
`(TO sum [(TO num[])(TO bool[])])`	$: num + bool$

The string denoting the name of type operators must be alphanumeric as described in next subsection.

1.3.2 Terms

There are four kinds of terms: variables, constants, applications and abstractions. They are represented respectively by the four kinds of expressions: `VAR`, `CONST`, `APP` and `ABS`.

The string in a `VAR` expression is the name of the variable. In HOL88, the variable name can be either alphanumeric or symbolic. An alphanumeric name is a sequence of one or more characters consisting of letters, digits, underscore(_), prime(') and percent sign(%) which begins with a letter. A symbolic name is a sequence of one or more characters which is an initial subsequence of one of the following character sequences:

```
**   ++   <--   <->   -->   ---   ><   >>
>=   <==   <=>   ===   ==>   \/   //   /\
!?   !!   !\   ?!   ?? ?\   :=   <>   <-   <<   ->   =>
```

The `type` field in a `VAR` expression specifies the type of the variable. Some examples of `VAR` expressions are:

`(V foo (TV *))`	$foo : *$
`(V GEN%VAR%1234 (TO bool[]))`	$GEN\%VAR\%1234 : bool$
`(V -- (TV *))`	$-- : *$

The string in a `CONST` expression is the name of the constant. It has the same form as the name of the variables. In addition, a sequence of one or more digits is also allowed, and it denotes a numeric constant. The `type` field of a constant expression gives the type of the specific instance of the constant. For instance, the HOL constant CONS has the polymorphic type $: * \rightarrow (*)list \rightarrow (*)list$, but the type of the occurrence of CONS in the term $\mathsf{CONS}\,\mathsf{T}\,l$ is $: bool \rightarrow (bool)list \rightarrow (bool)list$ which is more specific. This is the type given in the `CONST` expression representing this occurrence of CONS. Below are some examples of constant expressions:

`(C T (TO bool[]))`	$\mathsf{T} : bool$
`(C 2 (TO num[]))`	$\mathsf{2} : num$

The expression representing function application has two subterms: the first is the function and the second is its argument. No explicit type information is included since this can be deduced from the type of the subterms. For example, the expression

```
(A(C SUC(TO fun[(TO num[])(TO num[])]))(C 2(TO num[])))
```

represents $\mathsf{SUC}\,\mathsf{2}$.

The abstraction expression represents λ-abstraction in HOL. The two subterms are the bound variable and the abstraction body. Like function application expression, no explicit type information is included in an abstraction. For example, the expression

```
(L (V x(TO num[]))
   (A (A (C = (TO fun[(TO num[])(TO fun[(TO num[])(TO bool[])])]))
         (V x(TO num[])))(C 0(TO num[]))))
```

represents the abstraction $(\lambda x.x = 0)$.

1.3.3 Theorems

A HOL theorem consists of a list of hypotheses and a conclusion. This is represented by the `THM` expression. The `term_list` field is the list of hypotheses and the `term` field is the conclusion.

1.3.4 Lines

Each line in a proof represents a single inference. The `NUMBER` in a line acts as a label of the line. It can be either a positive or a negative integer, but not zero. The lines in a proof occur in ascending order of the line numbers. No two lines have the same line number. The `justification` field tells which inference rule is used in this step. Many justifications refer to the resulting theorems of other lines using their line numbers. It is illegal to refer to a line whose number is larger than the current line. The `thm_list` field contains the result of applying the inference rule.

1.3.5 Proofs

A proof has a name and consists of a sequence of lines. The name of the proof is a string. The `line_list` field contains the list of lines. If a proof is checked and no error is found, the theorems in all lines are valid theorems.

1.3.6 Justifications

The justification expression in a proof line represents the inference rule to be applied. There are eight primitive inference rules in the HOL logic. All other inference rules can be derived from these primitives. Theorems in the HOL system are represented by values of the abstract type `thm`. Applying inference rules is the primary means of creating values of type `thm`. This implies that to derive a theorem through proof in HOL requires the complete evaluation of all primitive inference steps. In order to improve the system performance, a number of simple and frequently used derived inference rules have been programmed directly in ML instead of calling the primitive rules. In recording HOL proofs, all these 'non-fully expanded' inferences are also recorded. Therefore, there are in total fifty-two kinds of justifications allowed in a proof. The HOL Manual [SRI91] describes the primitive and derived inference rules in details.

In a `justification` expression, the first field is the name of the inference rule. The name is a sequence of letters, and their case is ignored, e.g., `Subst` and `SUBST` indicate the same inference rule, substitution. The remaining fields are the arguments to the inference rule. If an argument is a theorem, it is not explicitly written, instead a number is given. This refers to the theorem in a line having the given number. Arguments of other types are written explicitly. For example, the inference rule `REFL` takes a term, it is explicitly written in the justification expression. The expression below represets the justification in Line 2 of Figure 1.1.

```
(Refl (A (C SUC(TO fun[(TO num[])(TO num[])]))
(A (A (C + (TO fun[(TO num[])(TO fun[(TO num[])(TO num[])])]))
(V m(TO num[])))(V n(TO num[])))))
```

1.4 A complete proof file

This section shows a complete proof file. The simple proof shown in Figure 1.1 is reproduced below for convenience.

1.	$\vdash m + n = n + m$	`Hypothesis`
2.	$\vdash \mathsf{SUC}(m + n) = \mathsf{SUC}(m + n)$	`REFL`
3.	$\vdash \mathsf{SUC}(m + n) = \mathsf{SUC}(n + m)$	`SUBST 1,2`

This proof can be carried out in HOL as follows:

```
#let th = SPEC_ALL ADD_SYM;;
Theorem ADD_SYM autoloading from theory `arithmetic` ...
ADD_SYM = |- !m n. m + n = n + m

th = |- m + n = n + m

#let v = genvar ":num";;
"GEN%VAR%540" : term

#record_proof true;;
() : void

#let th1 = (REFL "SUC(m + n)");;
th1 = |- SUC(m + n) = SUC(m + n)

#let th2 = SUBST [th,v] "SUC(m + n) = SUC ^v" th1;;
th2 = |- SUC(m + n) = SUC(n + m)

#write_last_proof`ap_term.prf`;;
() : void
```

An instance of the general theorem `ADD_SYM` is used as the hypothesis; `th1` is the result of applying the primitive inference rule `REFL`. This corresponds to Line 2 in the proof above; `th2` is the result of the substitution which corresponds to Line 3. The proof is output to a file named `ap_term.prf` whose contents are:

```
(PROOF ap_term.prf [
(LINE -1(Hypothesis)
[(THM [](A (A (C = (TO fun[(TO num[])(TO fun[(TO num[])(TO bool[])])]))
(A (A (C + (TO fun[(TO num[])(TO fun[(TO num[])(TO num[])])]))
(V m(TO num[])))(V n(TO num[]))))
(A (A (C + (TO fun[(TO num[])(TO fun[(TO num[])(TO num[])])]))
(V n(TO num[])))(V m(TO num[])))))
])

(LINE 1(Refl (A (C SUC(TO fun[(TO num[])(TO num[])]))
(A (A (C + (TO fun[(TO num[])(TO fun[(TO num[])(TO num[])])]))
(V m(TO num[])))(V n(TO num[])))))
[(THM [](A (A (C = (TO fun[(TO num[])(TO fun[(TO num[])(TO bool[])])]))
(A (C SUC(TO fun[(TO num[])(TO num[])]))
(A (A (C + (TO fun[(TO num[])(TO fun[(TO num[])(TO num[])])]))
(V m(TO num[])))(V n(TO num[])))))(A (C SUC(TO fun[(TO num[])(TO num[])]))
(A (A (C + (TO fun[(TO num[])(TO fun[(TO num[])(TO num[])])]))
(V m(TO num[])))(V n(TO num[]))))))
```

```
])

(LINE 2(Subst [(-1(V GEN%VAR%540(TO num[])))]
(A (A (C = (TO fun[(TO num[])(TO fun[(TO num[])(TO bool[])])]))
(A (C SUC(TO fun[(TO num[])(TO num[])]))
(A (A (C + (TO fun[(TO num[])(TO fun[(TO num[])(TO num[])])]))
(V m(TO num[])))(V n(TO num[])))))(A (C SUC(TO fun[(TO num[])(TO num[])]))
(V GEN%VAR%540(TO num[]))))1)
[(THM [](A (A (C = (TO fun[(TO num[])(TO fun[(TO num[])(TO bool[])])]))
(A (C SUC(TO fun[(TO num[])(TO num[])]))
(A (A (C + (TO fun[(TO num[])(TO fun[(TO num[])(TO num[])])]))
(V m(TO num[])))(V n(TO num[])))))(A (C SUC(TO fun[(TO num[])(TO num[])]))
(A (A (C + (TO fun[(TO num[])(TO fun[(TO num[])(TO num[])])]))
(V n(TO num[])))(V m(TO num[]))))))
])

])
```

Notice that the line numbers are generated by the HOL system automatically. Hypotheses are given negative line numbers.

Chapter 2

Modifying HOL for recording proofs

2.1 Introduction

A proof in the HOL system is a sequence of inferences. The user interacts with the system by writing and evaluating ML programs which instruct the system to perform the inferences [SRI91]. However, the proof itself is transient in the sense that no proof object persists once a theorem has been derived. In many applications, it is desirable for the correctness of the sequence of inferences to be independently checked. The ML programs a user develops while doing the proof are not adequate, and very often are too complicated for this purpose. To allow the independent checking of HOL proofs, the system has to be enhanced to output the proofs in terms of inference steps. An implementation of this feature has been developed in HOL88. This chapter describes this implementation; the previous chapter describes the format in which the proof is written to file.

2.2 User interface

To the user, proof recording is a feature which can be enabled, disabled and suspended temporarily at any time by calling ML functions. The typical use of this feature is as follows:

1. the user carries out a proof in the usual manner;

2. when he/she is satisfied with the proof, the proof recording feature is enabled and the proof is re-done once more;
3. an ML function is called to generate the proof file when the proof is complete.

The ML functions available to the users for managing proof recording are:

- `record_proof:bool -> void` When called with `true`, this function clears the internal buffer for storing proof steps and enables the recording of proof. When called with `false`, it just disables the feature without clearing the internal buffer.
- `is_recording_proof:void -> bool` This function returns the current state of the proof recording feature. If it is enabled, `true` is returned.
- `last_proof:void -> step list` This function returns the proof as a list of inference steps since the last time the proof recording feature was enabled.
- `write_last_proof:string -> void` This function outputs the contents of the internal proof buffer. The string argument is the name of the output file. If a file of the given name exists, it will be overwritten.
- `write_last_proof_add_to:string -> string -> void` This function appends the proof since the last time the proof recording feature is enabled into a file in the proof file format. The first string argument is the name of the output file. If no file of the given name exists, it will be created. The second string argument is the name of the proof.

Below is a simple interactive session with the HOL system to generate the proof file shown in the proof file format document.

```
#let th = SPEC_ALL ADD_SYM;;
Theorem ADD_SYM autoloading from theory `arithmetic` ...
ADD_SYM = |- !m n. m + n = n + m

th = |- m + n = n + m

#record_proof true;;
() : void

#let th' = AP_TERM "SUC" th;;
th' = |- SUC(m + n) = SUC(n + m)

#write_last_proof`ap_term.prf`;;
() : void
```

1.	$\vdash m + n = n + m$	`Hypothesis`
2.	$\vdash \mathsf{SUC}(m + n) = \mathsf{SUC}(m + n)$	`REFL`
3.	$\vdash \mathsf{SUC}(m + n) = \mathsf{SUC}(n + m)$	`SUBST 1,2`

Figure 2.1: A simple proof

The file `ap_term.prf` will then contain a proof, in proof file format, representing the proof shown in Figure 2.1. Although this example is extremely simple, any proof functions, such as conversions and tactics can be used. As these functions eventually invoke some basic inference rules, all the inference steps will be recorded in the proof file.

2.3 The implementation

The process of recording proof and generating proof file can be divided into three stages:

1. recording inference steps;
2. generating a proof;
3. outputing to a text file.

Once proof recording is enabled by calling the function `record_proof`, every application of basic inference rules[1] is recorded in an internal data structure. In the second stage, the raw records of the inference steps are processed to reduce the duplicated information and to generate a complete proof. The last stage writes the proof to a file.

2.3.1 Data structures

The internal buffer `steplist` for storing the raw records of inference steps is a list whose elements are of type `step`. Each step represents an application of a basic inference rule. This buffer can only be accessed by the user indirectly by calling the function `last_proof`.

Information contained in each step consists of the actual arguments passed to the inference rule. For example, the primitive inference rule `ASSUME` requires a term `t`, so the step corresponding to the application of this rule is `(AssumeStep t)`. The complete definition of the type `step` can be found in Appendix 2.4.

[1] Basic inference rules are the trusted ML functions implementing the HOL logic in which the function `mk_thm` is called to create a theorem.

In the second stage, the step list in the internal buffer is processed and a list of lines, i.e., a list whose elements are of type `line`, is created. Each step in the step list will generate a line in the line list. Each line represents a single inference which corresponds to a line written in Hilbert style of proof, such as the one shown in Figure 2.1. The definition of the type `line` is:

```
type line = Line of (int # thm list # justification);;
```

The `int` field is the line number. The second field is a list of theorems which are the results of applying the inference rule. The last field is the justification of the inference. It contains the name of the inference rule and the arguments to the rule. If an argument is a theorem, it is replaced by a line number. This refers to the line in which the theorem appears as the result of the inference. The definition of the type `justification` can be found in Appendix 2.4.

2.3.2 Internal states

Two internal variables govern the recording of proof: `record_proof_flag` and `suspended`. They can only be accessed indirectly by the user using the following functions:

- `record_proof` This function sets/clears the `record_proof_flag` to enable/disable the recording.
- `is_recording_proof` This function returns the current setting of the internal variable `record_proof_flag`.
- `suspend_recording` This function suspends the proof recording temporarily. It clears `record_proof_flag` and sets `suspended`.
- `resume_recording` This function resumes the proof recording. It sets `record_proof_flag` and clears `suspended`.

The temporary suspension of recording allows an auxiliary proof to be performed in the midst of a main proof without it being recorded. The internal states of the proof recording mechanism are illustrated in Figure 2.2. The states are labelled by the triple `(record_proof_flag, suspended, steplist)`. Three dots (...) in the square brackets indicate that `steplist` is not null. The transaction between states is caused by calling one of the functions above.

2.3.3 Recording inference steps

The actual recording of basic inference steps is done by the function `RecordStep`. This function takes a single argument of type `step`. It checks the value of the

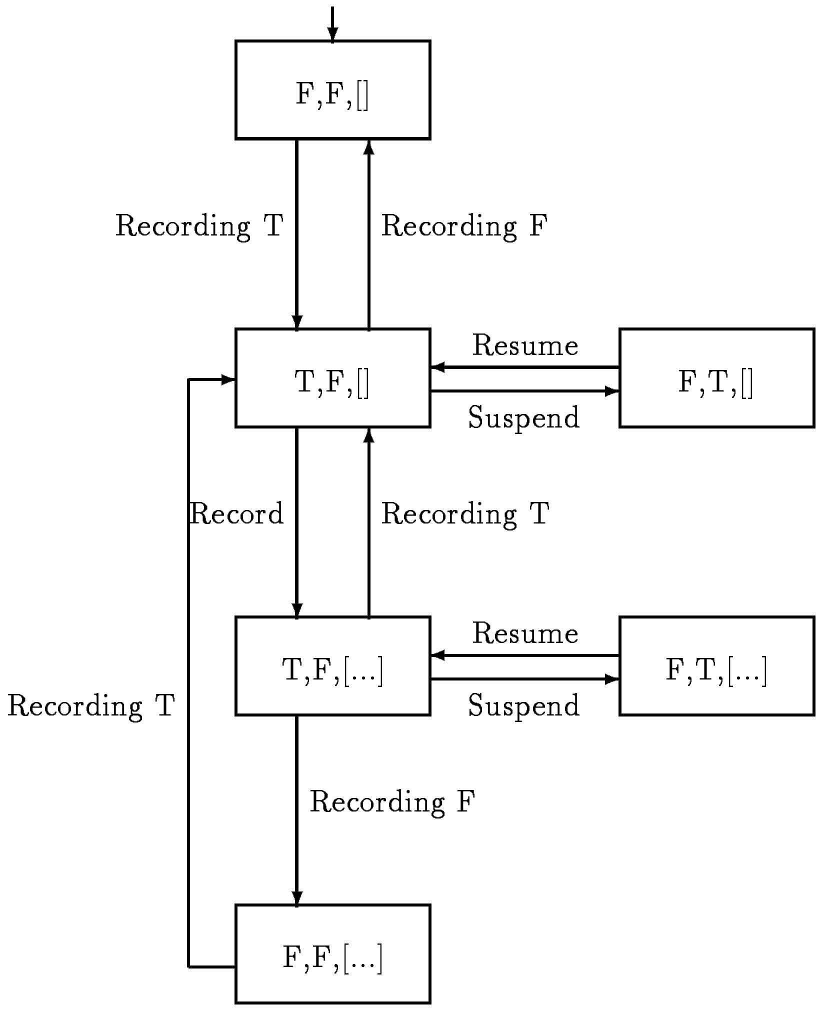


Figure 2.2: State diagram of proof recorder

internal variable record_proof_flag. If it is true, the step is added to the internal buffer steplist.

As mentioned above, every application of a basic inference rule is recorded. A call to the function RecordStep with the appropriate argument is added in each function that implements a basic inference rule. Note that basic inference rules include all the primitive rules and some derived rules. There are in total fifty-one such functions in the HOL system. These are exactly the places where the function mk_thm is called, which is the only function which creates a theorem. In this way, all the information about the application of inference rules is recorded dynamically. A list of all functions implementing basic inference rules can be found in Appendix 2.5. Below is the step list of the proof shown in Figure 2.1. It is in reverse order, i.e., the head of the list is the last step of the proof. There are two steps in this proof: the first is a ReflStep which is an application of the primitive rule REFL, and the second is the primitive rule SUBST.

```
[SubstStep([(|- m + n = n + m, "GEN%VAR%540")],
           "SUC(m + n) = SUC GEN%VAR%540",
           |- SUC(m + n) = SUC(m + n));
 ReflStep"SUC(m + n)"]
```

The primitive rule REFL takes a term as its argument and returns a equational theorem. The primitive rule SUBST performs a substitution on the theorem -SUC(m + n) = SUC(m + n)—.

2.3.4 Generating a list of lines

When a proof is complete, the list of steps is processed and a list of lines is generated. This is done by the function MakeProof. It converts each step into a line. The justification field of the lines contains the same information as in the steps. However, the theorems appeared in the steps are replaced by their line numbers.

Some of the theorems are not the result of any previous lines. These are the initial theorems of the current proof. They should have been proved in other proofs. These are added to the line list as hypotheses. The hypothesis lines are given negative line numbers.

Below is the list of lines for the proof shown in Figure 2.1.

```
[Line(-1, [|- m + n = n + m], Hypothesis);
 Line(1, [|- SUC(m + n) = SUC(m + n)], Refl"SUC(m + n)");
 Line(2, [|- SUC(m + n) = SUC(n + m)],
      Subst([(-1, "GEN%VAR%540")],
            "SUC(m + n) = SUC GEN%VAR%540", 1))]
```

Notice that the initial theorem is added as an hypothesis, and in Line 2, the last field of the justification is the line number 1 which refers to the theorem in Line 1. The corresponding step shown in last subsection contains the theorem explicitly.

2.3.5 Outputting to proof file

The user functions `write_last_proof` and `write_last_proof_add_to` are implemented by the top level output function `write_line` which takes two arguments: the first is an output port which connects to a file opened for writing, and the second is the line to be output. This function is in turn implemented by lower level output functions which write the line number, the theorem and the justification, and so on. These output functions are organized like a recursive descend parser which converts the internal structure of lines into the textual proof file format. They are local to the top level function `write_line`.

2.4 Types defined for recording proofs

```
type step = AssumeStep of term
 | ReflStep of term
 | SubstStep of (thm#term)list # term # thm
 | BetaConvStep of term
 | AbsStep of term # thm
 | InstTypeStep of (type#type)list # thm
 | DischStep of term # thm
 | MpStep of thm # thm
 | MkCombStep of thm # thm
 | MkAbsStep of thm
 | AlphaStep of term # term
 | AddAssumStep of term # thm
 | SymStep of thm
 | TransStep of thm # thm
 | ImpTransStep of thm # thm
 | ApTermStep of term # thm
 | ApThmStep of thm # term
 | EqMpStep of thm # thm
 | EqImpRuleStep of thm
 | SpecStep of term # thm
 | EqtIntroStep of thm
 | GenStep of term # thm
 | EtaConvStep of term
```

```
 | ExtStep of thm
 | ExistsStep of (term # term) # thm
 | ChooseStep of (term # thm) # thm
 | ImpAntisymRuleStep of thm # thm
 | MkExistsStep of thm
 | SubsStep of thm list # thm
 | SubsOccsStep of (int list # thm) list # thm
 | SubstConvStep of (thm # term) list # term # term
 | ConjStep of thm # thm
 | Conjunct1Step of thm
 | Conjunct2Step of thm
 | Disj1Step of thm # term
 | Disj2Step of term # thm
 | DisjCasesStep of thm # thm # thm
 | NotIntroStep of thm
 | NotElimStep of thm
 | ContrStep of term # thm
 | CcontrStep of term # thm
 | InstStep of (term # term) list # thm
 | StoreDefinitionStep of string # term
 | DefinitionStep of string # string
 | DefExistsRuleStep of term
 | NewAxiomStep of string # term
 | AxiomStep of string # string
 | TheoremStep of string # string
 | NewConstantStep of string # type
 | NewTypeStep of int # string
 | NumConvStep of term;;

type justification =
   Hypothesis
 | Assume of term
 | Refl of term
 | Subst of (int#term)list # term # int
 | BetaConv of term
 | Abs of term # int
 | InstType of (type#type)list # int
 | Disch of term # int
 | Mp of int # int
 | MkComb of int # int
 | MkAbs of int
 | Alpha of term # term
 | AddAssum of term # int
 | Sym of int
```

```
 | Trans of int # int
 | ImpTrans of int # int
 | ApTerm of term # int
 | ApThm of int # term
 | EqMp of int # int
 | EqImpRule of int
 | Spec of term # int
 | EqtIntro of int
 | Gen of term # int
 | EtaConv of term
 | Ext of int
 | Exists of (term # term) # int
 | Choose of (term # int) # int
 | ImpAntisymRule of int # int
 | MkExists of int
 | Subs of int list # int
 | SubsOccs of (int list # int) list # int
 | SubstConv of (int # term) list # term # term
 | Conj of int # int
 | Conjunct1 of int
 | Conjunct2 of int
 | Disj1 of int # term
 | Disj2 of term # int
 | DisjCases of int # int # int
 | NotIntro of int
 | NotElim of int
 | Contr of term # int
 | Ccontr of term # int
 | Inst of (term # term) list # int
 | StoreDefinition of string # term
 | Definition of string # string
 | DefExistsRule of term
 | NewAxiom of string # term
 | Axiom of string # string
 | Theorem of string # string
 | NewConstant of string # type
 | NewType of int # string
 | NumConv of term;;

type line = Line of (int # thm list # justification);;
```

2.5 Basic inference rules

This section lists all basic inference rules in the HOL system. The functions implementing these rules are modified to incorporate the feature of recording proof. They are grouped by the file where they are defined.

`hol-rule.ml`

ASSUME	REFL	SUBST	BETA_CONV
ABS	INST_TYPE	DISCH	MP

`drul.ml`

MK_COMB	MK_ABS	ALPHA

`hol-drule.ml`

ADD_ASSUM	SYM	TRANS	IMP_TRANS
AP_TERM	AP_THM	EQ_MP	EQ_IMP_RULE
SPEC	EQT_INTRO	GEN	ETA_CONV
EXT	EXISTS	CHOOSE	IMP_ANTISYM_RULE
MK_EXISTS	SUBS	SUBS_OCCS	SUBST_CONV
CONJ	CONJUNCT1	CONJUNCT2	DISJ1
DISJ2	DISJ_CASES	NOT_INTRO	NOT_ELIM
CONTR	CCONTR	INST	

`hol-syn.ml`

new_axiom	axiom	definition	store_definition
new_type	theorem	new_constant	DEF_EXISTS_RULE

`numconv.ml`

num_CONV

Part III

A Verified Proof Checking Algorithm and Its Implementation

Chapter 3

The Algorithm Specification Language AL

3.1 Introduction

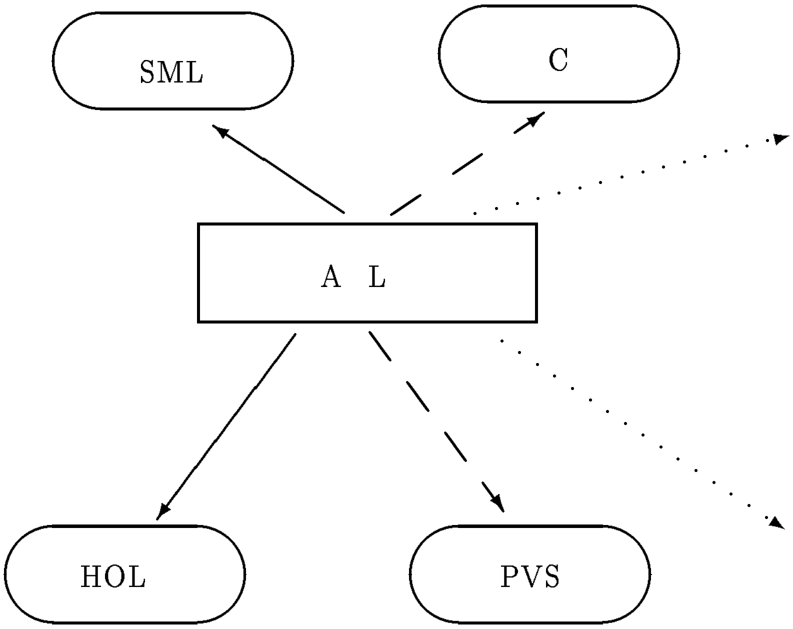
While algorithms may be described in both standard mathematical notations and standard programming languages, both of these have disadvantages. Mathematical notations may be more complex than is necessary for algorithm description and may not be executable or easily related to programming languages. Programming languages, while providing an executable and probably straightforward description of the algorithm, may not provide an abstract description, and may may not provide a good basis for proof.

The idea of AL is to provide a simple description of an algorithm that is very close to a subset of standard mathematical logic and standard functional programming languages. The goal is to have in the notation just the essentials needed — therefore the parts of a mathematical notation dealing with quantifiers are omitted and the parts of a functional language dealing with evaluation and exceptions are omitted. It is hoped that AL should address an approriate level for algorithm description lying between an abstract specification and an implementation, thus making the task of relating implementation to specification more tractable. In addition, it is intended that AL provide a portable, neutral description of the algorithm. By 'neutral' we mean that the description should be compatible with a number of mathematical notations so that proofs can be done, and a number of programming languages so that the algorithm can be implemented.

3.2 Overview

The language, AL, that has been developed and used is very simple: it consists of just type and function declarations. So it doesn't actually do anything but maps out how the computation can be achieved. Datatypes describe the problem space and solution space and the functions describe the algorithm for computing the required solutions.

The language looks simpler than mathematical notations since it does not have any quantifiers or other mathematical symbols. It is simpler than most programming languages due to the small number of constructs. AL corresponds to a subset of a functional language like ML. Although simple the use of rich datatypes and higher order functions makes the language expressive. Such datatypes and higher order functions do not exist in programming languages such as C but it is important to note that one just needs the programming language to be able to implement these features rather than actually have them in the language. Thus, for example, although C may not have as rich a datatype declaration facility, one could produce records and functions in C which would implement an AL datatype.



In the first implementation of AL, presented here, the main programming language target has been Standard ML (SML) and the main mathematical notation for representing the semantics has been HOL. SML is an appropriate choice because it is close to AL, a semantics of the language has been defined, and research on the embedding of the semantics of SML in HOL has been done [Sym93]. The relationships between AL and C and PVS have also been looked at. The diagram above illustrates the intended role of AL as a common intermediate language. The solid lines denote automatated interfaces that have been constructed, the dashed lines the other relationships which have just been looked at using hand translation.

3.3 AL constructs

In providing a description of the constructs of AL a simple grammar is given, where items in capitals are terminal tokens in the language, the operator | denotes alternation, { and } group items, and * denotes repetition (possibly zero).

An AL text consists of a series of declarations, each declaration is either a type declaration or a function declaration.

```
al_text     := {tydec | fundec}*
```

A type declaration declares a new type; this type may be a type operator if type variables are arguments of its constructors. The type declaration states the name of the new type, the constructors, the types of the arguments of the constructors, the accessors for the arguments of the constructors, and recognisors for the different constructors.

```
tydec     := TYPE type_dec END_TYPE
type_dec  := ty_name = dec {| dec}*
dec       := con_name con_args :: disc_name
con_args  := ( acc_name: atype ) {( acc_name: atype )}*
ty_name   := tyname
disc_name := name
con_name  := name
acc_name  := name
atype     := type {tyop} *
```

A type argument must either be be a valid type expression in the language or a recursive reference to the type being defined. The recursive reference may be either a simple reference to the type or a reference to a list of that type.

At least one constructor must have type arguments with no recursive references. The new type being defined is either a constant type, if there are no type variables, or an n-ary type operator if there are n distinct type variables among the type arguments.

So, the general template for type declarations looks like the following:

```
TYPE
ntype = C1 (acc1: ty1) ... (accN: tyN) :: is_C1 |
        C2 .... |
        Cn (accX: tyX) ... (accZ: tyZ) :: is_Cn
END_TYPE
```

`ntype` is the new type being declared;
`C1 ... Cn` are the type constructors;
`ty1 .. tyN` are the types of the arguments of type constructor `C1`;
`acc1 ... accX` are the corresponding accessors for the arguments of `C1`;
`is_C1 ... is_CN` are the recognisors for the different constructors.

For example, declaration of the type operator list in AL might be:

```
TYPE
list = null :: is_null |
       cons (car: *) (cdr: list) :: is_cons
END_TYPE
```

where the constructors are null and cons, car and cdr return the head and tail of the list respectively, and `is_null` and `is_cons` act as recognisors for the empty and non-empty lists.

A two place constructor may be used as an infix.

3.3.1 Function declaration

A function declaration declares a new function. The declaration states the name of the new function, its arguments, the type of the result, and the expression corresponding to the body of the function. There may be a recursive call to the function being declared in the body. There are no restrictions in the language on the form of recursion, but if the recursion is not well founded then the semantics of the function will not yield anything useful.

```
fundec    := FUN fun_dec END_FUN
fun_dec   := new_fun_name fun_args : atype = expr
fun_args  := (arg_name: atype) {(arg_name: atype)}*
expr      := const_name |
             var_name  |
             fun_name expr
new_fun_name := name
const_name := name
var_name  := name
fun_name  := name
```

```
arg_name  := name
atype     := type {tyop} *
```

The general template for function declarations is as follows:

```
FUN fn_name (arg1: type1) ... (argn: typen) : type =
 body
END_FUN
```

An example of a function declaration is the following function which computes the length of a list:

```
FUN length (lst: * list) : num =
 IF (null lst) THEN 0
 ELSE (1 + (length(cdr lst)))
END_FUN
```

The above definition is recursive. A recursive function declaration delivers a well-defined function only if the declaration can be shown to describe a unique function. The declaration mechanism is kept simple so that recursive, non-recursive and invalid recursive declarations all look similar, however the semantic model ensures that no inconsistencies are introduced.

A two place function may be used as an infix.

3.3.2 Built-in

As well as the facilities for declaring types and functions AL has a small number of built-in types and functions. The type bool, type operator ->, polymorphic equality = and conditional are built in. (We also experimented with a pattern matching version of AL where the conditional and boolean are not built-in.) Also built-in is a polymorphic constant ? which denotes a fixed, unknown constant value of the appropriate type. A convention in the use of names beginning with * to denote type variables is also assumed.

In the use of AL to describe and verify the proof checking algorithm a number of other types and functions have been assumed. These consist of the types num, list, set and string and their associated functions. These are all built-in to proof systems such as HOL and to redefine them would have been confusing and inefficient. A full list of these types and functions is given in the Appendix.

3.4 Semantics and Implementation

As well as providing an appropriate notation for describing algorithms, AL is intended to provide an intermediate level description which can be related to both a specification in a formal logic and an implementation in a programming language. AL should provide a common intermediate description which can be interfaced to a number of proof systems and programming languages. While a number of proof systems and programming languages were considered for interfaces, the prime candidates were the HOL proof system and the Standard ML programming language since the design of the notation was very much influenced by these.

3.4.1 Semantics in HOL

An AL text simply consists of a series of type and function declarations so a straightforward way of expressing the semantics of the text is through corresponding type and function declarations in a proof system such as HOL. The mapping from AL text to semantics may be done via a shallow embedding, where the parse tree is in the meta-language and a meta-language function defines the mapping from text to logical terms, or a deep embedding where the parse tree is in the object language and a function in the object language defines the mapping.

In doing a semantics in HOL, we have opted for a shallow embedding. The main reason for this is that HOL type and function declarations are done by functions in the meta-language and so doing such declarations in the object language would require an embedding of an appropriate infra-structure. It is more straightforward to implement the embedding by a direct use of HOL type and function definitions.

Type declarations

The type declarations of AL fit the following pattern:

```
TYPE
ntype = C1 (acc1: ty1) ... (accN: tyN) :: is_C1 |
        C2 ... |
        Cn (accX: tyX) ... (accZ: tyZ) :: is_Cn
END_TYPE
```

This declaration introduces a new type, its type constructors and a number of acccessor and recognisor functions. For a valid declaration which will produce a corresponding definition in the HOL logic, there must be at least one constructor with non-recursive arguments and recursive references must be either to the type or a list of the type. The admissable types therefore correspond to those accepted by the extended HOL type definition package [Gra93]. The new types are constructed in HOL using the technique devised for the standard type definition package [Mel89], namely that all types can be represented by an appropriate labelled tree. This means that all the type definitions are conservative extensions of the HOL logic since they can represented by pre-existing type operators and types.

In the core HOL system there are the primitive types `bool` and `ind`, representing the booleans and individuals respectively, and a primitive type operator, `->`, denoting the function type. All other types are defined and have representations using the primitive types. Types such as `num`, `string`, `list` and `set` are defined in the basic HOL system and so have not been redefined in AL. However the AL text which would define these types is given in Appendix A. One could envisage a changed HOL system containing just the primitive types and the type definition package with labelled trees. This would require a bootstrapping of the type definition package with type definitions which would not be included in the basic HOL system. One could then use the AL definitions to introduce basic types such as `num` and `list`, and then build the associated infrastructure on top.

As well as defining a new type, a type declaration in AL introduces several functions for accessing the arguments of constructors and recognising each constructor of a type. The semantics of the constructor and recognisor functions are represented in a straightforward way in HOL by HOL functions. The result of the application of an accessor to an object with the appropriate constructor returns the required argument, the application of an accessor to an object with an incorrect constructor returns a value in the logic which represents the unknown constant value ? in AL. These functions are not recursive so the general model of functions described in the next section is not used. If the more general model were used exactly the same functions would be derived.

Function declarations

A function declaration in AL looks like this:

```
FUN fn_name (arg1: type1) ... (argn: typen) : type =
 body
END_FUN
```

This declaration introduces a new function

The semantics of the function declaration is represented in HOL by a new function `fn_name` of type `(type1 -> type2 ... -> typen) -> type` which is defined to be the function

```
@fn_name.  fn_name arg1 ... argn =  body
```

i.e. any function that satisfies the equation `fn_name arg1 ... argn = body`

Since the body may contain recursive calls to `fn_name` one must prove that a function exists which satisfies the equation, and that this function is unique. Then one can eliminate the choice operator and derive an equation for the function in the logic which corresponds directly to the AL text:

```
fn_name (arg1: type1) ... (argn: typen) =  body
```

The use the choice operator in the semantics of functions has the effect of providing a unified model for recursive, non-recursive, and malformed recursive functions. By doing secure transformation of the logical terms in HOL one can simplify the non-recursive and recursive definitions to obtain the appropriate logical representation of functions. Malformed recursive functions do not introduce any inconsistencies and it is not possible to eliminate the choice operator in such cases to obtain functions in the logic corresponding to the AL text. The use of this model also means that there is just a single method of defining functions, non-recursive or recursive, in AL and the different proof obligations needed are taken care of in the semantic model.

In the core HOL system the only primitive functions are equality, implicaton and choice. In AL the primitive functions are equality and the conditional, `IF ... THEN ... ELSE ... END_IF`.

3.4.2 Translation to SML

The main design goal of AL is to provide a simple description of an algorithm in a languange that maps to different formal notations and programming languages. In particular AL is declarative and so has no execution model. It is of course useful to have an executable implementation corresponding to the functions of AL and since AL matches closely a subset of Standard ML, SML is an obvious candidate for implementations.

Type declarations

Type declarations in AL are translated into datatype and associated function declarations in SML. The datatype declaration closely matches AL except that in AL higher order constructors are allowed, whereas in SML each constructor has just a single argument. In SML one can use tuples or records to gather together multiple arguments. In translating higher-order constructors of AL into SML, a representation using tuples and associated constructors is used. The higher-order constructors are defined in terms of this intermediate SML respresentation.

An AL type declaration has the general pattern:

```
TYPE
ntype = C1 (acc1: ty1) ... (accN: tyN) :: is_C1 |
        C2 .... |
        Cn (accX: tyX) ... (accZ: tyZ) :: is_Cn
END_TYPE
```

The corresponding SML datatype declaration is:

```
datatype ntype = C1_int of (ty1 * ty2 .... * tyn) |
                 C2_int of ..... |
                 Cn_int of (tyX * tyY .... * tyZ) ;
```

along with the mapping back to the higher-order constructors:

```
  fun C1 arg1 arg2 ... argn = C1_int (arg1 * arg2 .... * argn);
```

Accessor and recognisor functions are defined along with these declarations. The following definitions for the first constructor of the type show the pattern of these:

```
fun acc1 (C1 arg1 arg2 ... argn) = arg1 |
    acc1 (C2 ....)               = raise access_ex |
    ...
    acc1 (Cn ....)               = raise access_ex;

fun is_C1 (C1 arg1 arg2 ... argn) = true  |
    is_C1 (C2 ....)               = false |
    ...
    is_C1 (Cn ....)               = false ;
```

It should be noted that datatypes in SML can be recursive but that no constraints are imposed on the recursion, thus non-wellfounded recursive types can be declared. In AL a recursive type must be one of the class of recursive types defined earlier. An invalid type will not produce a corresponding definition in the logic representing the semantics.

The core SML system includes the types `bool`, `int`, `string` and `list` along with the constants and functions associated with these types. The built-in SML type `int` is used to represent `num` and the other types are all used for the corresponding AL types. An efficient package for implementing sets is available as part of the distributed SML of New Jersey but to keep the mapping to SML as simple as possible an implementation of sets based on lists is defined. This is included in the SML prologue used for translation, which is included in Appendix D.

Function declarations

Function declarations in AL are translated into function declarations in SML. A function declaration in AL has the general pattern:

```
FUN fn_name (arg1: type1) ... (argn: typen) : type =
 body
END_FUN
```

and results in a corresponding SML function declaration:

```
fun fn_name (arg1: type1) ... (argn: typen) : type =
    body
```

SML, like AL, does not enforce any restrictions on resursive function definitions, so it is possible to declare and translate into SML a non-wellfounded function. While this might result in the user defining and trying to use functions which may be impossible to evaluate, the semantic model for functions in AL means that this will not lead to any false correctness results. In a future version of AL a warning might be issued to the user when possible non-wellfounded definitions are encountered.

3.4.3 Implementation of AL

The present implementation of AL is a prototype system which has allowed the investigation of AL as a convenient notation for describing algorithms and providing the basis for implementation and proof. A future, more robust, implementation will take advantage of generic tools to support embeddings.[1]

Being a prototype system means that there are a number of dangers that one must be aware of. For example, the parsing and interfacing of AL has been done using lisp and ML code and the parser of HOL. Issues like precedence of operators have not been made explicit or defined. Care has been taken to avoid any errors, for example by liberal use of parentheses to avoid precedence issues, but the basic issue still exists. A project to engineer a robust tool-set for AL would try to address these issues.

[1] A current project at the University of Cambridge Computer Laboratory is investigating such tools.

Generation of HOL

The programs that have been written to generate the HOL theory are mostly coded in classic ML, the ML of the present HOL88 system. The user uses the tools by either running an interactive HOL session and executing certain functions, or by running a batch job using a file with the appropriate function invocations.

The user calls an ML function `HOL_eval` which has as its argument a string corresponding to the stem of the name of the file containing the AL description. `HOL_eval` reads the named file with extension `.al` in the sub-directory `AL` and then produces a file with extension `.ml` in a sub-directory called `HOL`. The function does two things: it firstly calls the program that translates the AL into HOL, and secondly it loads that file and uses predefined functions which implement the actual type and function declarations which are defined by the HOL translation. For example, for the AL datatype declaration a string corresponding to the body of the declaration is parsed and the corresponding HOL type and associated function definitions are generated from this.

As mentioned elsewhere, the translation is a simple one and the use of the choice function in the model means that incorrect function definitions will not introduce inconsistencies. There are a number of tools which may be used to simplify these initial translations. The main task is the proof for each function that it does implement a valid definition and that the choice operator can be eliminated. This involves a number of tasks including finding the argument (if any) on which a definition recurses; verification that there exists a function satisfying the recursion equation and that this function is unique; and transforming the definition into one which is more like a cases enumeration for the function. For example the following is the AL description of how one checks that a certain type is well-formed:

```
FUN Type_OK (tyl:(string # num)list) (ty:Type) :bool =
CASE
(Is_Tyvar ty, true);
(Is_Tyop ty, (mem1 (Tyop_nam ty) tyl) AND
             (LENGTH (Tyop_tyl ty) = corr1 (Tyop_nam ty) tyl) AND
             EVERY (Type_OK tyl) (Tyop_tyl ty) )
END_CASE
END_FUN
```

When this is initially transformed into HOL and loaded into the system, the following definition is made:

```
Type_OK =
|- Type_OK =
   (@Type_OK'.
```

```
!tyl ty.
 Type_OK' tyl ty =
 CASE_FN
 [Is_Tyvar ty,T;
  Is_Tyop ty,
  (mem1(Tyop_nam ty)tyl) AND
  ((LENGTH(Tyop_tyl ty) = corr1(Tyop_nam ty)tyl) AND
   (EVERY(Type_OK' tyl)(Tyop_tyl ty)))])
```

The above definition is not very useful, so the tools are used to automatically prove that the equation contained within the choice expression uniquely defines a function, and then to generate a more useful defining equation for `Type_OK`. The following is the resultant theorem in HOL which has been proved from the initial definition:

```
|- (!(tyl:(string # num)list) (s:string).
     Type_OK tyl(Tyvar s) = T) /\
   (!(tyl:(string # num)list) (s:string) (Tlst:(Type)list).
     Type_OK tyl(Tyop s Tlst) =
     mem1 s tyl /\
     (LENGTH Tlst = corr1 s tyl) /\
     EVERY(Type_OK tyl)Tlst)
```

Generation of SML

The generation of SML is done in a similar way to the generation of HOL. Again, the user interacts with the tools by calling functions from the ML of the HOL system. The user calls an ML function `SML_eval` which has as its argument a string corresponding to the stem of the name of the file containing the AL description in a sub-directory `AL`. This function reads the AL file and generates an SML file with the extension `.sml` in the sub-directory `SML`. This file can then be loaded directly into an SML session using the `use` command.

The generation of SML does not involve any proofs but some extra work is needed because of the use of curried arguments in AL and the non-curried form in SML. The system generates some extra internal functions for constructing and accessing curried datatype declarations and then defines the other accessors and constructors in terms of these.

For example, the following is the AL text describing the datatype representing HOL types:

```
TYPE
Type = Tyvar (Tyvar_nam: string) :: Is_Tyvar |
       Tyop (Tyop_nam: string) (Tyop_tyl: Type list) :: Is_Tyop
```

```
END_TYPE
```

The automatic translation of this into SML generates the following code:

```
datatype Type = Tyvar of (( string)) |
                Tyop_tup of (( string))*(( Type list)) ;
fun Is_Tyvar(Tyvar _) = true |
    Is_Tyvar(Tyop_tup _) = false ;
fun Is_Tyop(Tyvar _) = false |
    Is_Tyop(Tyop_tup _) = true ;
exception de_Tyvar_ex;
exception de_Tyop_tup_ex;
fun de_Tyvar (Tyvar x) = x  |
    de_Tyvar (Tyop_tup _) = raise de_Tyvar_ex ;
fun de_Tyop_tup (Tyvar _) = raise de_Tyop_tup_ex |
    de_Tyop_tup (Tyop_tup x) = x  ;
val Tyvar_nam = (tup1_1 o de_Tyvar) ;
val Tyop_nam = (tup2_1 o de_Tyop_tup) ;
val Tyop_tyl = (tup2_2 o de_Tyop_tup) ;
fun Tyop a b = Tyop_tup (a,b) ;
```

The above datatype declaration and associated function declarations include some extra generated constructors, such as `Tyop_tup` and `de_Tyvar`. The actual costructors such as `Tyop` and `Tyvar_nam` are then defined in terms of these.

3.4.4 Translation to PVS

One of the main goals of AL is to provide a simple neutral description of an algorithm that can be translated into different programming languages and can be modelled in different formal systems. Translation of AL to PVS was done for all the datatypes presenting the hierarchy of HOL types, terms, sequents and proofs and many of the functions over these types. A full embedding of the theory of HOL proofs was not done because PVS is still gaining maturity as a tool and a large effort would be needed to automate the translation and proofs. However, the issues of translation to logics such as PVS was one of the main design considerations behind the simple AL notation, and there are no fundamental difficulties in doing this translation.

Examples

The following examples show some of the datatype and function declarations in AL and the corresponding ones in PVS. The PVS declaration corresponding to the AL one for `Type` is:

```
HType [sty:TYPE] : DATATYPE
  BEGIN
  Tyvar(Tyvar_nam:sty): Is_Tyvar
  Tyop (Tyop_nam:sty, Tyop_tyl:list[HType]): Is_Tyop
  END HType
```

The declaration looks quite like AL since the AL notation is modelled on that of PVS and nqthm. Differences to note are that the constructors are not curried and that the type operator is prefix rather than postfix. In the version of PVS being used the type `string` was not built in so the datatype is parameterised by a type variable which can be instantiated to a particular type implementing strings. This style of type parameterisation is an important feature of PVS and there are subtle differences between the use of such polymorphic variables in PVS and HOL. In HOL a polymorphic variable has essentially global scope and can be instantiated at will. In PVS theories are parameterised by their type variables and these may need to be explicitly instantiated when a theory is imported, or one may need to import two different instantiations of a theory.

The following is the declaration in PVS of the datatype representing HOL terms:

```
Pterm [sty:TYPE]  : DATATYPE
   BEGIN
   IMPORTING HType
      Const(Const_nam: sty, Const_ty:HType[sty]) : Is_Const
      PVar (Var_var_st: sty, Var_var_t:HType[sty]) : Is_Var
      App  (App_fun: Pterm, App_arg: Pterm) : Is_App
      PLam (Lam_var_st:sty,
            Lam_var_t:HType[sty], Lam_bod: Pterm) : Is_Lam
   END Pterm
```

The `IMPORTING` clause is used since `HType` is needed in this definition. This theory is again parameterised on the string type and this is also used in the instantiation `HType[sty]`.

3.4.5 Translation to Imperative Languages

The translation of AL to C was also investigated. The main difficulties that were identified were the use of higher-order constructors and functions and the use of polymorphism. While rewriting the algorithm in AL in a style suitable for translation to C would have been possible, it would have meant several changes from the style used as the basis for the formal model of HOL theorems and proofs.

One would have to consider carefully the semantics of even a subset of C corresponding to AL. In contrast to SML there was no effort to provide a semantics when the language was being defined and no full semantics of the language is yet available.

What has been done on this project is to define a small imperative language in HOL and then verify that an implementation of the proof checking algorithm in this language does satisfy the requirements of proof checking as defined by the AL algorithm and translated into a HOL theory of proofs. This small imperative language has in turn been translated by hand to the classic ML of HOL, and in this way some small examples can be executed.

The interface between AL and this imperative language is right now at the semantic level in HOL. A better solution would be to formally derive the refinements from AL to an imperative language so that this refinement can be done at the source text level.

3.5 Future Work

The development and implementation of AL has been very useful in the present project as a common intermediate notation for algorithm description. However, its development has been driven by need and improvements could be made. The main one would be to engineer a better set of tools and possibly use generic tool development kits. A better embedding in HOL would be possible with generic tools to support notation embedding in HOL.

The AL notation may be improved in a number of ways. The use of curried constructors for datatypes was driven by the desire for compatibility with the previous formalisation of HOL proofs. There is no particular advantage in having this feature and a future version will probably drop it. The use of pattern matching in SML is one of its attractions and the introduction of this would make AL descriptions easier to read. Pattern matching was avoided in the first version of AL because it was thought to be an obstacle to interfacing to more conventional non-pattern matching programming languages. In fact it seems possible to use pattern matching in the AL description and then have a translation to a non pattern matching version where necessary.

AL has been interfaced to HOL and SML and some hand translation to other notations has been done. It should be possible to construct interfaces to other proof systems and programming languages. The interface to the formal system is important for formal proofs and one of the advantages of interfacing to HOL is the ability to define a very safe, simple translation and then use the programmability of HOL to write the tools to do the simplification necessary. The embedding of AL in other proof systems will need some sort of similar support.

AL has not been interfaced to an imperative programming language but a simple programming language has been embedded in HOL and this related directly to an algorithm description via the embedded semantics in HOL. Interfacing at this low level is not very efficient and a better solution would be to have the sort of transformation from a functional to an imperative style described in Part IV defined directly at the language level.

Since AL is non executable and describes datastructures and the functional mappings over them one could also consider having a visual representation of algorithms. This would be especially appropriate if the proposal to adopt a pattern matching notation were implemented. A graphical representation might be useful for visualisation or when debugging an actual implementation of an algorithm developed independently.

AL has been very useful as a portable intermediate language for describing algorithms. It might also be used as an intermediate notation for interfacing between tools such as model checkers and proof tools. If all the types declared are simple enumerated types, and the functions are over these types then correctness proofs may be reduced to a binary decision diagram problems. If one had a problem which partitioned into decidable and non-decidable subsets then one might want to do proofs both in a proof tool and a BDD tool. Using AL as the description language might provide a means of transferring subproblems to different tools and integrating the result. One would have to base the interfacing on a precise semantics of the notations being used.

Associated with the changes to the notation suggested above it would be useful to explore the possibility of algorithm development and refinement. For example, early versions of an algorithm might leave certain cases unspecified and the user would like to be able to incrementally add to the algorithm. An appropriate notation and semantics to support this sort of program development would be very useful.

Chapter 4

A Verified Proof Checker in AL

4.1 Specification

We now describe the specification of the proof checker — or more precisely what is required for proof checking — in AL. The specification is based directly on the theory of HOL proofs by von Wright [vW94b] and an attempt has been made to keep the names of the various types, constructors and functions identical so that the the descriptions can be directly related. The theory developed here has a different starting point, AL, and has more automation to transform the semantics of AL to palatable HOL. In addition, the theory has been kept flat as no modularisation facility was included in the AL notation. The following description closely follows von Wright's original formulation.

The specification of the proof checker corresponds to specifying what a proof is in HOL and therefore one needs to build up a description of what HOL types, terms, sequents and inferences are. In addition, one needs to formalise the various concepts needed for describing properties relevant to proof, for example type instantiation and free and bound variables.

The aim of the formalisation is to be able to check proofs, not to generate them; therefore one does not need to model inference rules as functions which return theorems. Instead, it is enough that we are able to recognise a correct inference, once the result is given. This means that one does not have to deal with HOL's intricate procedures for variable renaming used in the two primitive inference rules `INST_TYPE` and `SUBST`. The formalisation permits arbitrary renaming schemes, and the one used by HOL as a special instance.

A *theory* in HOL is characterised by a type structure, a set of defined constants (a signature) and a set of axioms. In our formalisation, a type structure is represented by a list of pairs `(op,n)` of type `string#num`, where `n` is the arity of the type operator `op`.

The constants of a theory are represented by a list of pairs `(const,ty)` where `ty` is the generic type (which can be polymorphic) of the constant `const`.

Finally, the axioms of a theory are represented by a list of sequents. Sequents, in turn, are constructions `Pseq as tm`, where `as` is a set of terms (the assumptions) and `tm` is a term (the conclusion). The equations that define constants are also considered to be axioms.

4.1.1 Types

The HOL logic has four different kinds of types: type variables, type constants, function types and n-ary type operators. Type constants and the function type are taken to be special cases of type operators.

```
TYPE
Type = Tyvar (Tyvar_nam: string) :: Is_Tyvar |
       Tyop (Tyop_nam: string) (Tyop_tyl: Type list) :: Is_Tyop
END_TYPE
```

The type structure of the current theory is represented by a list of pairs of type `:string#num`. For example, the simplest possible theory (referring only to booleans) has the following type structure list:

```
[(`bool`,0);(`fun`,2)]
```

The HOL type `:bool` is then represented by `Tyop `bool`[]` while the function type `:bool→bool` is represented by `Tyop `fun` [Tyop `bool` [];Tyop `bool` []]`.

The following function states that a Type is OK if it is a type variable or it is composed from OK types by a permitted type operator.

```
FUN Type_OK (tyl:(string # num)list) (ty:Type) :bool =
CASE
(Is_Tyvar ty, true);
(Is_Tyop ty, (mem1 (Tyop_nam ty) tyl) AND
             (LENGTH (Tyop_tyl ty) = corr1 (Tyop_nam ty) tyl) AND
             EVERY (Type_OK tyl) (Tyop_tyl ty) )
END_CASE
END_FUN
```

Here `mem1 s l` holds if `s` is the first component of some pair in the list `l` and `corr1 s l` is the corresponding second component (these are defined in a separate theory containing useful definitions and theorems, mainly about lists).

Other functions over Type are also declared. For example, `Type_occurs a ty` is defined to hold if the type variable a occurs anywhere in the type `ty`, and the function `Type_compat` is defined so that `Type_compat ty ty'` holds when `ty` is compatible with `ty'`, in the sense that the structure of `ty` is can be mapped onto the structure of `ty'`.

4.1.2 Terms

The AL datatype `Pterm` represents the fact that a HOL term can be a constant, a variable, an application or a lambda abstraction.

```
TYPE
Pterm = Const (Const_nam: string) (Const_ty: Type) :: Is_Const |
        Var (Var_var: (string#Type)) :: Is_Var |
        App (App_fun: Pterm) (App_arg: Pterm) :: Is_App |
        Lam (Lam_var: (string#Type)) (Lam_bod: Pterm) :: Is_Lam
END_TYPE
```

The constants of the current theory are represented by a list. A constant always has a generic type which is given in this list. When the constant occurs in a term, it has an actual type which must be an instance of the generic type. For example, equality on booleans is represented by the Pterm

```
Const `=` (Tyop `fun`
            [Tyop `bool`[];
             Tyop `fun`[Tyop `bool` [];Tyop `bool` []]])
```

This term is well-typed if the list of constants contains the pair

```
(`=`,Tyop `fun`[Tyvar `*`;Tyop `fun`[Tyvar `*`;Tyop `bool`[]]])
```

A simple logic might have the following the list of constants:

```
[`T`,Tyop `bool`[] ;
 `F`,Tyop `bool`[] ;
 `=`,Tyop `fun` [Tyvar `*`;Tyop `fun`[Tyvar `*`;Tyop `bool`[]]]
 `==>`,Tyop `fun`
       [Tyop `bool`[];
        Tyop `fun`[Tyop `bool`[];Tyop `bool`[]]]]
```

i.e., truth, falsity, equality and implication.

Well-typedness Every Pterm has a unique Type, computed by the function `Ptype_of`. Our syntax permits terms which are ill-typed, in the sense that they do not correspond to the any 'real' HOL terms. A term is well-typed if it satisfies two requirements: the types of the subterms of applications must match, and the constants occurring in the term must have types which are correct instantiations of their generic types. The function `Pwell_typed` checks these conditions.

```
FUN Pwell_typed
(Typl:(string#num)list)(Conl:(string#Type)list)(ptm:Pterm):bool =
CASE
(Is_Const ptm,
         (mem1 (Const_nam ptm) Conl) AND
         (Type_OK Typl (Const_ty ptm)) AND
         (Type_compat(Const_ty ptm)(corr1(Const_nam ptm)Conl))AND
         (nocontr
          (Type_instl(Const_ty ptm)(corr1(Const_nam ptm)Conl))));
(Is_Var ptm, Type_OK Typl (snd (Var_var ptm)));
(Is_App ptm,
         (Pwell_typed Typl Conl (App_fun ptm)) AND
         (Pwell_typed Typl Conl (App_arg ptm)) AND
         (Is_Tyop(Ptype_of (App_fun ptm))) AND
         (Tyop_nam(Ptype_of (App_fun ptm)) = `fun`) AND
         (LENGTH(Tyop_tyl(Ptype_of (App_fun ptm))) = 2) AND
         (car(Tyop_tyl(Ptype_of (App_fun ptm))) =
                                        Ptype_of(App_arg ptm)));
(Is_Lam ptm,
         (Pwell_typed Typl Conl (Lam_bod ptm)) AND
         (Type_OK Typl(snd (Lam_var ptm))))
END_CASE
END_FUN
```

Alpha-renaming The notions of free and bound variables are defined in the obvious way. For example, we define `Pfree` so that `Pfree x t` holds if the variable `x` occurs free in the Pterm `t`.

```
FUN Pfree (x:string#Type) (ptm: Pterm) :bool =
CASE
(Is_Const ptm, false);
(Is_Var ptm, (Var_var ptm = x) );
(Is_App ptm, (Pfree x (App_fun ptm)) OR (Pfree x (App_arg ptm)));
(Is_Lam ptm, (NOT(Lam_var ptm = x)) AND (Pfree x (Lam_bod ptm)))
END_CASE
END_FUN
```

The functions `Pbound` and `Poccurs` are defined similarly.

Alpha-renaming and substitution of a term for a variable are closely bound together. We have defined `Palreplace` so that `Palreplace t' tvl t` holds if `t'` is the result of substituting according to the list `tvl` and alpha-renaming. The list `tvl` consists of pairs `(t,a)` of type `Pterm#(string#Type)`, indicating what terms should be substituted for what variables.

We can then define alpha-equivalence using an empty substitution:

```
FUN Palpha (t':Pterm)(t:Pterm):bool = Palreplace t' [] t  END_FUN
```

Multiple substitutions We can now also formalise HOL's notion of a substitution, as it occurs in the inference rule `SUBST`. Assume that `ttvl` is a list of triples of type `Pterm#Pterm#(string#Type)`. For each triple `(tm',tm,d)` in this list, `tm'` is a Pterm that is to replace `tm` and `d` is a dummy variable used to indicate the positions where this substitution is to be made. Then `Psubst t' ttvl td t` holds if `t` is the result of substituting some `tm`-terms for `d`-dummies in the term `t` and if `t'` is the result of substituting `tm'`-terms for `d`-dummies. Both substitutions are done according to `ttvl`, and they may involve alpha-renaming.

Type instantiation Type instantiation is quite tricky to check. First, it is necessary to check that the instantiation has not identified two variables that were previously distinct. Second, the type instantiation rule of HOL permits free variables to be renamed (in fact, the specification of the HOL logic in [GM93] does not permit this, but in this case we follow the implementation rather than the specification).

Checking a renaming of a free variable is more complicated that checking a renaming of a bound variable, because bound variables are always "announced" (in the left subtree of the abstraction), but a free variable can occur in two widely separated subtrees, without being announced in the same way.

A number of auxiliary functions have been defined before defining the `Ptyinst` function. Given that `tyl` is a list of pairs of type `Type#string`, indicating what types are to be substituted for what type variables, and given that `ass` is a Pterm set that represents the assumptions of the theorem that is to be type-instantiated, then `Ptyinst ass t' tyl t` holds if `t'` is the result (after renaming) of replacing type variables in `t` according to `tyl` and if no variables that are type instantiated occur free in `ass`.

4.1.3 Inferences

Sequents are represented by the following datatype:

```
TYPE
Psequent = Pseq(Pseq_assum:(Pterm)set)(Pseq_concl:Pterm)::Is_Pseq
END_TYPE
```

The first argument to `Pseq` is the set of assumptions and the second argument is the conclusion. The corresponding destructor functions are `Pseq_assum` and `Pseq_concl`.

Basic inferences An inference step consists of a *conclusion* (result sequent) that is 'below the line' and a list of *hypotheses* (argument sequents) that are 'above the line'.

HOL inference rules are functions which in addition to the hypotheses may require some information (e.g., a term) in order to compute the conclusion. For example, the rule of abstraction (`ABS`) in the logic is

$$\frac{\Gamma \;\vdash\; \mathtt{t} \;=\; t'}{\Gamma \;\vdash\; (\lambda x.\, t) \;=\; (\lambda x.\, t)}$$

with the side condition that x must not occur free in Γ. As an inference rule in the HOL system, `ABS` is a function which takes a term representing the variable x and a theorem (the hypothesis) as arguments and returns a new theorem (the conclusion).

Following von Wright's formulation, inferences need to be checked rather than produced. In this way the formalisation of an inference rule always has an additional (first) argument, which is the conclusion of the inference.

For each inference rule, we define a function which returns a boolean value: `T` for a correct inference and `F` for an incorrect one. The `ASSUME` rule is modelled by the function `PASSUME`:

```
FUN PASSUME Typl Conl pseq tm =
        Pwell_typed Typl Conl tm AND
        Pboolean tm AND
        (Pseq_concl pseq = tm) AND
        (Pseq_assum pseq = (SetOf[tm]))
END_FUN
```

`Pboolean tm` is a function checking that the Pterm `tm` has boolean Type, The well-typedness condition is also checked here and this ensures that all result sequents in a proof are well-typed. To make this check, we must have the type structure `Typl` and the constant list `Conl` as explicit arguments to `ASSUME`.

The seven other primitive inferences are modelled in a similar way.

Inferences as a new type The arguments of the basic inference rules are not uniform. For example, the `MP` rule takes two theorems as arguments, while the `REFL` rule takes a single term. To be able to reason about inferences in a uniform way, we define a new type `Inference` with the following syntax:

```
TYPE
Inference =
AX_inf (AX_con: Psequent) :: is_AX |
AS_inf (AS_con: Psequent) (AS_tm: Pterm) :: is_AS |
RE_inf (RE_con: Psequent) (RE_tm: Pterm) :: is_RE |
BE_inf (BE_con: Psequent) (BE_tm: Pterm) :: is_BE |
SU_inf (SU_con: Psequent)
       (SU_eqthms: (Psequent#(string#Type)) list)
       (SU_tmdum: Pterm) (SU_thm: Psequent) :: is_SU |
AB_inf (AB_con: Psequent)
       (AB_tm: Pterm)
       (AB_eq: Psequent) :: is_AB |
IN_inf (IN_con: Psequent)
       (IN_tyl:(Type#string)list)
       (IN_thm: Psequent) :: is_IN |
DI_inf (DI_con: Psequent)
       (DI_tm: Pterm)
       (DI_thm: Psequent) :: is_DI |
MP_inf (MP_con: Psequent)
       (MP_ithm: Psequent)
       (MP_thm: Psequent) :: is_MP
END_TYPE
```

The first case corresponds to an inference by axiom, each of the other rules corresponds to a basic inference rule. The first argument of an inference constructor is always the conclusion of the inference. The remaining arguments represent the hypotheses and other arguments. In some cases (e.g., `AX_inf`) the list of assumptions is empty, but for `SU_inf` it can be of arbitrary length.

An inference is correct if it satisfies the defining properties of the inference rule. For example, an object built with `AS_inf` is a correct inference if its arguments satisfy the predicate `PASSUME`, i.e. if it represents an application of the HOL inference rule `ASSUME`.

The function `OK_Inf` is defined to represent this notion of correct inference. Thus `OK_Inf i` holds if and only if `i` represents a correct inference, according to the primitive inference rules of the HOL logic. The definition of `OK_Inf` is shown in Appendix A.

```
FUN OK_Inf Typl Conl Axil inf =
CASE
(is_AX inf, mem (AX_con inf) Axil);
(is_AS inf, PASSUME Typl Conl(AS_con inf)(AS_tm inf));
(is_RE inf, PREFL Typl Conl(RE_con inf)(RE_tm inf));
(is_BE inf, PBETA_CONV Typl Conl(BE_con inf)(BE_tm inf));
(is_SU inf, PSUBST Typl Conl
            (SU_con inf)(SU_eqthms inf)(SU_tmdum inf)(SU_thm inf));
(is_AB inf, PABS Typl Conl(AB_con inf)(AB_tm inf)(AB_eq inf));
(is_IN inf, PINST_TYPE Typl(IN_con inf)(IN_tyl inf)(IN_thm inf));
(is_DI inf, PDISCH Typl Conl(DI_con inf)(DI_tm inf)(DI_thm inf));
(is_MP inf, PMP(MP_con inf)(MP_ithm inf)(MP_thm inf))
END_CASE
END_FUN
```

4.2 Proofs

By a proof we mean a sequence of correct inferences where each inference has the property that all its hypotheses appear as conclusions of some inference earlier in the proof.

This is captured in the recursively defined predicate `Is_proof`:

```
FUN Is_proof Typl Conl Axil l =
CASE
(is_null l, true);
(is_cons l, OK_Inf Typl Conl Axil (car l) AND
            lmem(Inf_hyps (car l))(MAP Inf_concl (cdr l)) AND
            Is_proof Typl Conl Axil (cdr l) )
END_CASE
END_FUN
```

The definition of `Is_proof` provides the specification of the proof checker, or more accyrately the proof checking algorithm. An actual program which checks proofs must implement this algorithm.

There is no way to prove that this specification of a proof actually captures the HOL notion of a proof. However, one can reason about proofs and check that they satisfy some minimal requirements. As an example of this, we have shown that proofs can only yield sequents where the hypotheses and the conclusions are well-typed and boolean:

```
|- !P. Is_proof Typl Conl Axil P /\
       Is_standard(Typl,Conl,Axil) ==>
```

```
EVERY Pseq_boolean(MAP Inf_concl P) /\
EVERY (Pseq_well_typed Typl Conl) (MAP Inf_concl P)
```

where `Is_standard` holds for the triple `(Typl,Conl,Axil)` if the type structure contains booleans and function types, the constant list contains polymorphic equality and implication and the axiom list contains only well-typed boolean sequents.

In addition, there are about 300 tests devised by von Wright to test the formulation of the HOL theory of proofs. The translation into HOL of the AL description has also been subjected to these tests. For example, the following is the result of checking whether `Is_proof` holds of a tiny proof, where the parsing and pretty-printing developed by von Wright is used to show the result in a compact form:

```
|- Is_Xproof
   [MP_Xinf
    (Xseq{y = y}(x = x))
    (Xseq{}((y = y) ==> (x = x)))
    (Xseq{y = y}(y = y));AS_Xinf(Xseq{y = y}(y = y))(y = y);
    DI_Xinf(Xseq{}((y = y) ==> (x = x)))(y = y)(Xseq{}(x = x));
    RE_Xinf(Xseq{}(x = x))x] =
   T
```

The theory in HOL built by translating the AL specification is the basis against which the implementations of the proof checking algorithm are verified. The SML version of the checker is generated directly and automatically from the AL description. The checker in a small imperative language was verifed with respect to the original formulation which is distributed as a contribution in the HOL system. This theory is compatibile with the theory developed from AL but the small differences, due to for example variable names and quantifier ordering, require a reworking of proofs.

4.3 Implementation in SML

In the previous section, a specification of the proof checker in AL is outlined. The specification of the proof checker is different from certain other specifications in that it specifies what is required to be checked i.e. an algorithm. The AL language description as well as providing the basis for proof also provides the basis for implementation.

The main target language for translation of AL is SML. While a semantics of SML has been defined, there has been no attempt to relate the semantics of AL to the semantics of SML since the embedding of the semantics of SML in proof tools is still an ongoing research area. It might be possible in the future to do this. Therefore the translation to SML must be seen as a possible source of errors, although the closeness of AL datatype and function declaration mechanisms to those of SML means that this possibility is minimised.

An automatic translation from AL to SML has been implemented. This is used by calling the ML function `SML_eval` from the ML of the HOL system. The argument of `SML_eval` is a string corresponding to the stem of the name of the file containing the AL description. This function reads in the AL file and generates an SML file which can then be loaded directly into an SML session using the `use` command. As part of translation, as outlined earlier, the system generates some extra internal functions for constructing and accessing curried datatype declarations and then defines the other accessors and constructors in terms of these.

4.3.1 SML code

The SML code generated from AL is given in Appendix D. The following typical datatype and function declarations show how closely the descriptions correspond as well as some differences.

The declaration in AL of `Type`, its accessors and recognisors is:

```
TYPE
Type = Tyvar (Tyvar_nam: string) :: Is_Tyvar |
       Tyop (Tyop_nam: string) (Tyop_tyl: Type list) :: Is_Tyop
END_TYPE


FUN Type_OK (tyl:(string # num)list) (ty:Type) :bool =
CASE
(Is_Tyvar ty, true);
(Is_Tyop ty, (mem1 (Tyop_nam ty) tyl) AND
             (LENGTH (Tyop_tyl ty) = corr1 (Tyop_nam ty) tyl) AND
             EVERY (Type_OK tyl) (Tyop_tyl ty) )
END_CASE
END_FUN
```

The automatic translation into SML produces the following code.

```
datatype Type = Tyvar of (( string)) |
                  Tyop_tup of (( string))*(( Type list)) ;
fun Is_Tyvar(Tyvar _) = true |
    Is_Tyvar(Tyop_tup _) = false ;
fun Is_Tyop(Tyvar _) = false |
```

```
    Is_Tyop(Tyop_tup _) = true ;
exception de_Tyvar_ex;
exception de_Tyop_tup_ex;
fun de_Tyvar (Tyvar x) = x  |
    de_Tyvar (Tyop_tup _) = raise de_Tyvar_ex ;
fun de_Tyop_tup (Tyvar _) = raise de_Tyop_tup_ex |
    de_Tyop_tup (Tyop_tup x) = x  ;
val Tyvar_nam = (tup1_1 o de_Tyvar) ;
val Tyop_nam = (tup2_1 o de_Tyop_tup) ;
val Tyop_tyl = (tup2_2 o de_Tyop_tup) ;
fun Tyop a b = Tyop_tup (a,b) ;


fun  Type_OK (tyl:(string  *  num)list) (ty:Type) :bool =
CASE_fun [
(Is_Tyvar ty, fn zz => true),
(Is_Tyop ty, fn zz => (mem1 (Tyop_nam ty) tyl) AND
             (LENGTH (Tyop_tyl ty) = corr1 (Tyop_nam ty) tyl) AND
             EVERY (Type_OK tyl) (Tyop_tyl ty) )
]
; (* end fun dec *)
```

Firstly one may note the similarity between the basic datatype and function declarations. However, one should also note some differences. The introduction of dummy functions to allow the mapping between curried and uncurried forms of functions has been described earlier. In addition one should see that some functions raise exceptions. The concept of exceptions is not in AL but when an executable translation in SML is being run and analysed then such exceptions may be useful. Another point to note is the slightly obscure use of the abstraction notation in the translated case statement. For example:

```
 (Is_Tyvar ty, fn zz => true)
```

This is used to avoid exceptions caused by the strict evaluation mechanism of SML. Without such a mechanism, the evaluation of the above function on a type variable will fail, as the type operator accessors `Tyop_nam` and `Tyop_tyl` will be applied to a type variable even though such an object would satisfy the first clause and fail the second.

By using the SML pattern matching mechanism in function definitions, the above problem is avoided. If a new AL notation does include a similar pattern matching construct then the above problem may not arise in the translated code.

Both of the above definitions, datatype and function, are recursive. As was stated earlier, non-wellfounded recursive datatypes and functions can be defined in SML. Although one could translate a non-wellfounded AL definition into a non-wellfounded SML definition, the semantics of AL ensures that this does not introduce an inconsistency into the formal model.

4.3.2 Using the checker

The SML checker can be run directly to check proofs by constructing the appropriate sequents involving terms and types and applying the function `Is_proof`. This would be tedious and it would be very difficult to check more than tiny proofs. Instead the SML checker has been interfaced to the HOL proof record format so that the checker can be run independently to check HOL proofs which create a proof record log file.

The mechanism for recording proofs and the format used is described in Part II. One can select an option in HOL whereby all inferences are logged and a proof log is saved on disk in a low level format suitable for transfer between systems or suitable for independent checking.

To use the SML core checker generated from the AL specification, one must also load the various parsing, printing and other utility functions needed. This is done by running the command:

```
use "join1.sml";
```

where `join1.sml` in turn loads all the files needed including the the core proof checking functions. The non-critical parts of the checker, which manage the proof file input, and parsing, are adopted from an earlier, unverified, checker due to Wai Wong which uses a different data representation and checking functions.

After loading all the checking functions, the system can be run by invoking the command `parse_file` where the argument is the name of the file containing the recorded proof. For example, a simple proof is described as follows in the style used for proof recording:

```
1) |- T Hypothesis
2) F |- F ASSUME
3) |- F ==> T Disch 1
4) F |- T MP 3,2
```

The corresponding recorded proof log is in the file `add_assum.prf`. This proof can then be checked by invoking `parse_file` and the resultant output is as follows:

```
val parse_file = fn : string -> unit
val it = () : unit
- parse_file "add_assum.prf" handle e => Raise e;
```

```
socket name:RTMP9269
SERVER:accept socket No.4
socket name:WTMP9269
SERVER:accept socket No.5

Current environment: HOL
Current environment: HOL

Proof: add_assum is valid
val it = () : unit
-
```

The final output states that the proof recorded in add_assum is valid.

4.3.3 Limitations

The SML proof checker which has been constructed based on the AL specification of proof checking has two main drawbacks: it is not very efficient, and it only deals with primitive inferences.

These are not fundamental issues. The inefficiency could be improved by a verified refinement of AL into a more efficient algorithm, and for the sort of proof checking envisaged speed should not be a critical factor. Dealing only with primitive inferences requires that the user does not use non-primitive inference rules. These non-primitive inference rules are listed in a file unsafe_rules in the HOL system. Where these rules appear in the HOL system files, an alternative version using only primitive inferences is used. The system is built using the non-primitive rules for better efficiency. It would be possible for a user to build a version of HOL where only the primitive rules are used and then generate proof record logs which could then be checked by SML proof checker.

A more efficient SML proof checker is available which checks both the primitive and non-primitive rules. However, this checker has not been related to any formal specification. It should be noted of course that it is the core checking function, Is_proof, that has been generated from the AL specification and one should be aware of the possibility of errors in the utility functions that surround it. There is always an issue of where one should draw the line when ensuring correctness — can the hardware and operating system be trusted to do what they should? It is certainly of benefit to have proofs of correctness where possible, and be aware of the correctness issues in the areas which have not been verified.

Part IV

A Verified Imperative Proof Checker for HOL

Chapter 5

A methodology for program verification

This chapter describes von Wright's methodology for developing functional procedures that have been verified correct using HOL. The method is based on the RefCalc contribution which is part of Version 2.02 of the HOL system [vW94a]. For a full documentation we refer to this reference and to [vW94c]. For a list of definitions and theorems, see Appendix F. Here we give a quick overview and discuss some implications for the verification of a proof checker.

5.1 Overview of the verification methodology

5.1.1 The programming notation

The programming notation that we use has a simple syntax. To make life simple, we use the syntax of the HOL-embedding directly. In a sense, this means that we blur the distinction between syntax and semantics.

Predicates A *predicate* is an arbitrary boolean-valued function `p:*->bool`. In instances, the argument type `*` will be a product type and elements of this product type are called *states.*

A predicate always makes explicit what the state space is. For example if we want to model the predicate

$$x < y + 1$$

we have to know what the underlying state space is. Assuming that it is a triple (x, y, z) of variables ranging over the natural numbers, we would represent the predicate as

```
λ(x,y,z). x < y + 1
```

This means that the state components (which model the program variables) are anonymous; they are identified by their position in the state tuple rather than by name.

By "lifting", we define functions corresponding to the ordinary logical operations on predicates. For example,

```
⊢def false    = λu. F
⊢def not p    = λu. ¬p u
⊢def p andd q = λu. p u ∧ q u
```

We also introduce generalized conjunctions (greatest lower bounds, `glb`) and disjunctions (least upper bounds, `lun`) of sets of predicates, where sets are modelled by their characteristic functions.

Furthermore, we define the implication order on predicates:

```
⊢def p implies q =  ∀u. p u ⇒ q u
```

Thus `p implies q` means that `p` is stronger than `q`.

Commands A *command* is a function `c:(*2->bool)->(*1->bool)` from predicates to predicates. The idea behind this is to identify a command `c` with its weakest precondition predicate transformer. Predicate transformers map predicates over the final state space to predicates over the initial state space (note that these need not be the same state space). Intuitively, the predicate `c q` holds in an initial state `s` if execution of `c` from initial state `s` is guaranteed to terminate in a final state where the predicate `q` holds. What we call `c q` would in traditional weakest precondition methodology be called `wp(c,q)`.

In principle, we accept *any* function of type `(*2->bool)->(*1->bool)` as a command. However, we mainly want to reason about commands that have a specific syntax. For this, we introduce a command syntax, which essentially just consists of a number of abbreviations for commands. We start with a simple set of commands; later on we will extend them with a specification construct and recursion. The commands we want to reason about at this point have the following syntax:

```
c ::= skip
    | assert g
    | assign e
    | c1 seq c2
    | cond g c1 c2
    | do g c
```

where g is a predicate (for "guard") and e is a state-state function (of type *1->*2).

Intuitively, seq models sequential composition, cond models conditional composition (if-then-else) and do models iteration (a while-loop). Finally, assign e is a command which transforms a state s into a state e s. Ordinary assignment statements can always be written using this command.

The predicate transformer semantics of these commands is at the same time their definitions:

```
⊢def skip q          = q
⊢def assert g q      = g and q
⊢def assign e q      = λu. q (e u)
⊢def (c1 seq c2) q   = c1(c2 q)
⊢def cond g c1 c2 q  = (g andd (c1 q)) or ((not g) andd (c2 q))
⊢def do g c q        = fix(λp. (g andd (c p)) or ((not g) andd q)))
```

Here fix is a least fixpoint operator for monotonic functions on predicates. The above semantics is a straightforward encoding of traditional weakest precondition semantics. For example, skip q = q encodes what is traditionally written $\mathrm{wp}(\mathit{skip}, q) = q$. However, we identify commands with predicate transformers, so we do not need any separate semantic function.

As an example, consider the assignment $x := x + y$ in a three-component state space where x is the first and y the second component. This is represented by the command

```
assign λ(x,y,z). (x+y,y,z)
```

Note that we explicitly state that the values of the second and third component are unchanged.

Our command notation is in a sense arbitrary. We may add more constructs (as we do below). We also note that the skip command is redundant; it can be written as an assignment:

```
⊢ skip = assign λu.u
```

Healthiness conditions We identify four important criteria (often called "healthiness conditions") that commands with a weakest precondition semantics are usually assumed to satisfy. These criteria are monotonicity, strictness, conjunctivity and termination. Their definitions are as follows:

```
⊢def monotonic c   =  (∀p q. p implies q ⇒ (c p) implies (c q))
⊢def strict c      =  (c false = false)
⊢def conjunctive c =
       ∀P. (∃p. P p) ⇒ (c(glb P) = glb(. ∃p. P p ∧ (q = c p)))
⊢def terminating c =  (c true = true)
```

In fact, all the commands in our notation are monotonic, strict and conjunctive. The `assert` and `do` commands may be nonterminating.

Conjunctivity of `c` means that `c` distributes over arbitrary nonempty conjunctions of predicates. Intuitively, the conjunctivity assumption means that nondeterminism is assumed to be demonic. Note that since `true` is the empty conjunction, a command which is terminating and conjunctive distributes over all conjunctions.

5.1.2 Total correctness

A command `c` is said to be totally correct with respect to precondition `p` and postcondition `q` if `c` is guaranteed to establish `q` whenever it is executed from an initial state where `p` holds. Our definition is simple:

```
⊢ correct p c q  =  p implies (c q)
```

We have derived (i.e., proved in HOL) a number of rules for proving total correctness of commands. For a complete listing of these, see Appendix D.

For assignments, the rule is as follows:

```
⊢ (∀v. p v ⇒ q(e v)) ⇒ correct p (assign e) q
```

This corresponds closely to the ordinary Hoare rule (of total or partial correctness) for assignment:

$$\frac{P \;\Rightarrow\; Q[E/x]}{\{P\}\; x := E\; \{Q\}}$$

For sequential composition, we have a simple sequencing rule, familiar from Hoare logic:

```
⊢ (∃q. monotonic c1 ∧ correct p c1 q ∧ correct q c2 r)
    ⇒ correct p (c1 seq c2) r
```

When applying this rule, the user must supply a suitable intermediate predicate q. The monotonicity condition can be proved automatically for commands that are built using the basic command notation.

For the conditional command, we also have a simple Hoare-style rule:

```
⊢ correct (g and p) c1 q ∧ correct ((not g) and p) c2 q
   ⇒ correct p (cond g c1 c2) q
```

The rule for iteration is the well-known invariant-bound rule:

```
⊢ (∃inv t. monotonic c ∧
          p implies inv ∧
          ((not g) and inv) implies q ∧
          (∀x. correct (inv and (g and (λu. t u = x)))
                       c
                       (inv and (λu. (t u) < x))))
   ⇒ correct p (do g c) q
```

When this rule is used, the user has to supply the invariant `inv` and the bound function `t`.

We do not give a separate correctness rule for the `assert` command. This is because this command is not interesting as such. Instead it is used to make reasoning about conditional correctness easy (for details, see [vW94c]).

5.1.3 Functional abstraction

We assume that a program `c` is supposed to compute a function `f`. We take the view that functions are uncurried and they always have at least one argument. This means that we can without loss of generality assume that `f` has type `*1->*2`. In an instantiation, the initial state `*1` contains one component for each argument of the function, while the final state `*s2` has only one component (the result returned by the function). Since this component may contain subcomponents, a multi-valued function can be treated as a single-valued function whose value is a tuple.

Now assume that we have written a program fragment `c` of which we can prove the following:

```
⊢ c = assign f
```

Then it is fair to say that `c` is an implementation of the function `f`, and that any occurrence of `f` in a program can be replaced by a call to `c`.

This idea has been formalised in the following way. We define an operator `fcall`, which extracts the function `f` from a command `c` in the case when `c = assign f` holds. It satisfies the following property:

```
⊢ fcall (assign f) = f
```

In fact, we can prove a theorem which is easily used as a proof rule:

```
⊢ conjunctive c ∧ strict c ∧ (∀u0. correct (λu.u = u0) c (λv.v = f u0))
    ⇒ (fcall c = f)
```

Using this theorem, we can prove that a given program `c` implements a function `f`. Once this is done, we can replace any occurrence of `f` by `fcall c`, which models the call to the functional procedure `c`.

When the above theorem is used as a proof rule, the conjunctivity and strictness conditions can be proved automatically (we have implemented HOL tactics for this), so we are left with a total correctness condition. This can be proved using the rules in Section 5.1.2.

Example: list membership As a small example, we implement a list membership function. List membership can be defined in HOL as follows:

```
⊢def (∀x. MEMBER(x,[]) = F) ∧
     (∀x h t. MEMBER(x,(CONS h t)) = (h = x) ∨ MEMBER(x,t))
```

(note that this is an uncurried version of the membership function).

The program that we propose to implement this function with is defined as follows:

```
⊢def fMEMBER =
     (assign λ(x,l).(F,x,l)) seq
     (do (λ(r,x,l).¬(l=[]) ∧ ¬r)
         (assign λ(r,x,l).((x = HD l),x,TL l)) ) seq
     (assign λ(r,x,l).r)
```

Note how this program is built up. The global state components (i.e., the reference parameters) are `x` and `l`; the element and the list. The first `assign` command initializes the local boolean variable `r` to the boolean value `F`. The main body of the program is the iteration, which checks the elements of the list, one by one, until either the list is empty or until a match is found, in which case `r` is set to the boolean value `T`. The final `assign` command returns `r` as the result. The correctness of the `fMEMBER` procedure is proved in the theorem

```
⊢ fcall fMEMBER = MEMBER
```

which shows that any occurrence of the `MEMBER` function can be correctly replaced by a call to `fMEMBER`.

Note that in the program `fMEMBER` we use other function that HOL can recognize: `HD` and `TL`. These can either be assumed to be built-in, or else we can write separate programs that implement them.

For an example of how the correctness proof is done, we refer to Section 7.2.

Conditional correctness In some cases we only want to be sure that a functional procedure correctly implements a function in the case when a given precondition holds. We call this *conditional correctness* (not to be confused with partial correctness, which does not require termination).

We have derived the following rule for proving conditional correctness assertions:

```
⊢ conjunctive c ∧ strict c ∧
   (∀u0. p u0 ⇒ correct(λu. u = u0)c(λv. v = f u0))
   ⇒ (∀u. p u ⇒ (fcall c u = f u))
```

Here p is the precondition that the implementation is proved under.

For example, we may want to implement a function which retrieves a value from an association list, provided that the given argument occurs in the list.

Such a function ASSOC can be specified as follows:

```
⊢ ASSOC(x0,(CONS(x,y)l)) = ((x = x0) →  y | ASSOC(x0,l))
```

(note that this is a partial specification; we do not specify what happens in the case when the value is not found in the list).

If x0 occurs as a first component of some pair in a list l then ASSOC(x0,l) returns the corresponding second component. Otherwise, its behaviour is unspecified.

The procedure fASSOC implements this function, under the precondition that we can find the given element in the association list.

```
⊢ fASSOC =
    (assign(λ(x,l).((εr.T),x,l))) seq
    (do (λ(r,x,l). ¬(l = []))
        (cond (λ(r,x,l).x = FST(HD l))
              (assign λ(r,x,l).(SND(HD l),x,[]))
              (assign λ(r,x,l).(r,x,TL l)) ) ) seq
    (assign(λ(r,x,l). r))
```

The first assign command initializes the local component r to an arbitrary value (since the type of r is polymorphic, the only way to choose an arbitrary value is by using the Hilbert operator ϵ). Then the list is searched until it is empty. If a match is found then the corresponding value is stored in r and the list is nulled.

```
⊢ MEMBER(x,(MAP FST l)) ⇒ (fcall fASSOC(x,l) = ASSOC(x,l))
```

This means that an occurrence of ASSOC can be replaced by a call to fASSOC provided that we have a guarantee that the membership condition holds.

5.1.4 A nondeterministic assignment

To model a nondeterministic assignment of a value to a variable, we extend our notation with the construct `nondass`. Let `P:*s1->*s2->bool` be a relation between the initial and the final state space. Intuitively, `nondass P` is a command which for the initial state `v` chooses an arbitrary final state `v'` such that `P v v'` holds. If there is no such state, then the command is *miraculous*, i.e., its behaviour is not specified. For example, the following command increases the first component of a two-component state space arbitrarily:

```
nondass λ(x,y)(x',y'). (x'>x) ∧ (y'=y)
```

Formally, the definition of the nondeterministic assignment is the following:

```
⊢def nondass P q = (λv. ∀v'. P v v' ⇒ q v')
```

The nondeterministic assignment is useful in a situation where an ordinary assignment is over-specific. For example, we often do not care how a local variable is initialized. If we use the `assign` construct to introduce the variable, we are forced to supply an initial value. On the other hand, the `nondass` construct permits us to introduce the variable without restricting its initial value.

As an example, we consider a functional procedure for the `ASSOC` function, defined as follows:

```
⊢ fASSOC =
    (nondass λ(x,l)(r,x',l'). (x'=x) ∧ (l'=l)) seq
    (do (λ(r,x,l). ¬(l = [])
        (cond (λ(r,x,l).x = FST(HD l))
              (assign λ(r,x,l).(SND(HD l),x,[]))
              (assign λ(r,x,l).(r,x,TL l)) ) ) seq
    (assign(λ(r,x,l). r))
```

Here the local variable `r` is not initialised, i.e., its initial value is unspecified. This procedure avoids the use of an arbitrary initialization. even though it contains a nondeterministic assignment, the procedure as a whole is deterministic.

5.2 Choosing level of detail for verification

It is not always clear to what level of detail we should go in the verification. For example, we might not want to implement the function `LENGTH`, assuming that any programming language with list types also contains a predefined function for computing the length of a list.

If we do top-down verification, then we can at some point decide that we have reached a sufficient level of detail. At this point we stop verification and regard the verification task as completed. For example, we may decide to implement the LENGTH function as

```
(assign λl.(0,l)) seq
(do (λ(r,l).¬(l=[]))
    (assign λ(r,l).(SUC r,TL l))) seq
(assign λ(r,l).r)
```

However, we may well assume that the functions SUC and TL do not need to be implemented.

5.3 Implementing higher-order functions

Our programming notation also permits higher-order functions to be implemented. A higher-order functions is implemented as a procedure which takes a function as one of its parameters. As an example, consider the HOL function MAP. Its first argument is a function which is mapped over the second argument. a possible implementation of MAP is the following:

```
(assign λ(f,l).([],f,l)) seq
(do (λ(r,f,l).¬(l=[]))
    (assign λ(r,f,l).(SNOC(f(HD l))r,TL l))) seq
(assign λ(r,f,l).r)
```

The correctness theorem is

```
⊢ fcall fmap = MAP
```

and it is straightforward to prove using the standard technique with an invariant and a termination function, as described above.

When rewriting an expression containing MAP, we use the correctness theorem, as usual. Thus, e.g., the expression

$$\texttt{MAP LENGTH } l$$

can be rewritten as

$$\texttt{fcall fmap}(\texttt{LENGTH}, l)$$

Now we can also use the correctness theorem for flength to rewrite further, getting

$$\texttt{fcall fmap}(\texttt{fcall flength}, l)$$

Modern imperative languages permit this kind of procedures, where one of the argument is itself a procedure. Thus we have a way of modelling procedural parameters in our verification methodology.

5.4 Refining procedures

The procedure declaration for which we prove a correctness theorem need not be the final version. Often it is best first to write a clear and readable, though perhaps not very efficient procedure declaration, the correctness proof of which is not too difficult. Once this procedure is proved correct, we can do refinements to it. The HOL theory underlying the program verification methodology also supports a number of refinement techniques.

We will not describe these in detail in this report; we only show how refinement methods can be used to improve procedure implementation.

5.4.1 Improving efficiency

Efficiency can be improved in a number of ways. For example, if the same expressions is computed many times, we can add an assignment which performs the computation once, and stores the value in a new local variable.

It may also be possible to decrease the number of actual iterations of a loop, if the final result of the loop is known before all the iterations have been performed.

These kinds of efficiency improving refinements can be done within the HOL system, using the notion of refinement defined in the complete RefCalc contribution of the HOL system [vW94a]. There a relation of refinement `ref` is defined and the following theorem is proved:

$$\vdash \texttt{strict c}' \wedge \texttt{conjunctive c}' \wedge \texttt{c ref c}' \wedge (\texttt{fcall c = f}) \Rightarrow \ (\texttt{fcall c}' = \texttt{f})$$

This shows that if `c'` refines `c` and `c` implements the function f, then `c'` also implements the function f. As before, the strictness and conjunctivity conditions can be proved automatically.

5.4.2 Changing data representation

Besides to improve efficiency, refinement transformation can also be used to change the data representation used. For example, we may use sets in a procedure definition, but may then decide to implement the procedure in a language which does not permit set types. We can then represent sets as lists. This means that we have to implement all set operations in terms of lists. This can often be done in many different ways. For example, we may implement set union simply as list `APPEND`, but we may also want to eliminate duplicates from the result. The first implementation is more time-efficient while the second one is more space-efficient.

The correctness of implementations with changed representation of data can be verified using *data refinement* methods. The RefCalc contribution supports data refinement. For more detail, we refer to the documentation of the RefCalc contribution.

Chapter 6

An interface for program verification

The programming notation described in Chapter 5 is not very convenient to use in practice. The repeated occurrences of lambda-abstractions makes the notation difficult to read and also to write programs in. To remedy this, parser and pretty-printer support has been developed by Mike Gordon. This lets the user enter and see the procedures in a notation which is close to that of ordinary imperative programming languages, such as Pascal, C or Modula.

This chapter shows how the parser and pretty-printer works. Its documentation is listed in Appendix G.

6.1 Syntax

The parser/pretty-printer uses traditional imperative syntax. The syntax is always relative to a *state vector s*, which is a tuple containing the names of the state variables. For example, if the state vector is $(x : num, y : num, z : bool)$, then the program variables are x, y and z.

Expression syntax is HOL syntax, with the exception of the pattern matching `CASE`-expressions (see below). This means that the parser/pretty-printer can handle arbitrary data types and arbitrary constant names; they are simply printed exactly as they are normally printer by the HOL system.

6.1.1 Basic statement syntax

Relative to state vector s we have the following translations to the underlying programming notation.

$$
\begin{array}{rcl}
\{\texttt{SKIP}\}\texttt{s} & \rightarrow & \texttt{skip} \\
\{\texttt{ASSERTe}\}\texttt{s} & \rightarrow & \texttt{assert } \lambda\texttt{s.e} \\
\{\texttt{s} := \texttt{e}\}\texttt{s} & \rightarrow & \texttt{assign } \lambda\texttt{s.e} \\
\{\texttt{c1}; \texttt{c2}\}\texttt{s} & \rightarrow & \{\texttt{c1}\}\texttt{s seq } \{\texttt{c2}\}\texttt{s} \\
\{\texttt{IF e THEN c1 ELSE c2}\}\texttt{s} & \rightarrow & \texttt{cond } (\lambda\texttt{s.e})\{\texttt{c1}\}\texttt{s}\{\texttt{c2}\}\texttt{s} \\
\{\texttt{WHILE e DO c}\}\texttt{s} & \rightarrow & \texttt{do } (\lambda\texttt{s.e}) \ \{\texttt{c}\}\texttt{s}
\end{array}
$$

Furthermore, in assignments we can leave out those state component (variables) that do not change.

For example, assume that we want our program to work relative to the state vector (b:bool,x:num,y:num). Then

```
SKIP;
IF x=0 THEN b:=false
ELSE y:=x
```

is parsed into

```
skip seq
(cond (λ(b,x,y). x=0)
      (assign λ(b,x,y). (false,x,y))
      (assign λ(b,x,y). (b,x,x)))
```

6.1.2 Procedure declaration

A procedure declaration has the form

$$< head > < init > ; < body > ; < return >$$

The procedure head $< head >$ is of the form

$$\texttt{PROCEDURE } < name > (< global >)$$

where $< name >$ is the name of the procedure and $< global >$ is the parameter tuple (i.e., the global variables).

The local variable initialisation $< init >$ is of the form

$$\texttt{VAR } < varinit >$$

where $< \mathit{varinit} >$ is a sequence (separated by commas) of variable initialisations $< \mathit{variable} >=< \mathit{value} >$.

The body $< \mathit{body} >$ is a statement according to the statement syntax given above. Finally, the return statement $< \mathit{return} >$ is of the form

$$\texttt{RETURN} \; < \mathit{expr} >$$

where $< \mathit{expr} >$ is an expression, the value of which is returned when the procedure call terminates.

A procedure declaration must contain $< \mathit{head} >$ and $< \mathit{return} >$, but $< \mathit{init} >$ and $< \mathit{body} >$ are optional.

Translation of a procedure declaration A procedure declaration is translated into the underlying programming notation as follows. The procedure head gives the name of the procedure. Furthermore, the variable names in the head and the initialisation together give the names of all the variables, i.e., the state tuple s. When these are known, the body is translated into the programming notation as described above. Furthermore, the initialisation is translated into an assignment which adds the local variables and leaves the global variables unchanged. The return statement is translated into an assignment which has a final state containing only one component, corresponding to the returned expression.

As an example, we consider the procedure declaration

```
"PROCEDURE fmember(x,l)
  VAR r=F;
  WHILE ¬(l=[]) ∧ ¬r DO
   r,l := (x = HD l),TL l;
  RETURN r"
```

This is translated into the following term in the underlying notation:

```
"(assign λ (x,l). (T,x,l)) seq
 (do (λ(r,x,l). ¬(l=[]) ∧ ¬r)
     (assign λ(r,x,l). (SUC r,x,TL l))) seq
 (assign λ(r,x,l). r)"
```

Note that the name of the procedure does not appear in this translation. It is used when defining the procedure as a new constant (see below).

Uninitialised variables Our programming notation permits local variables to be uninitialised. Such variables are only given names (but no initial values) when they are introduced in a procedure declaration. Such a declaration is translated into a nondeterministic assignment. As an example, consider the following procedure which finds the element associated with a given value in an association list.

```
"PROCEDURE fassoc(x,l)
  VAR y;
  WHILE ¬(l=NIL) DO
   IF x = FST(HD l) THEN
    y,l := SND(HD l),NIL
   ELSE
    l := TL l;
  RETURN y"
```

Here the local variable y is not given any particular initial value. This is translated into the following term in the underlying notation:

```
"(nondass λ (x,l)(y',x',l'). (x'=x) ∧ (y'=y)) seq
 (do (λ(y,x,l). ¬(l=NIL))
     (cond (λ(y,x,l). x = FST(HD l))
           (assign λ(y,x,l). (SND(HD l),x,NIL))
           (assign λ(y,x,l). (y,x,TL l))) seq
 (assign λ(y,x,l). y)"
```

6.2 Pattern matching

We often want a choice of action to depend on the type constructor used in the current value of some variable. For example, we may want different actions to be taken, depending on whether the value of natural number variable x is zero or a successor. A typical example of this is when we implement recursive definitions over a data type.

6.2.1 Case operators

The pattern-matching depends on the definition of a case operator for each recursive type. Such a case operator is easily defined in HOL from the theorem which characterises the type in question.

As an example, consider the natural numbers. The case operator num_CASE is defined by the following:

```
⊢ ∀f0 f1.
    (num_CASE f0 f1 0 = f0) ∧
    (num_CASE f0 f1 (SUC n) = f1 n)
```

This operator lets us express many recursive functions in a simple way. For example, the predecessor function PRE (where PRE $0 = 0$) is easily characterised as follows:

```
⊢ PRE = num_CASE 0 (λn. n)
```

because PRE $0 = 0$ and PRE(SUC n) $= n$.

6.2.2 Case declaration

The case operators can be used to implement a simple pattern matching facility via the parser/pretty-printer support.

In order for such definitions to be acceptable, we first have to declare to the parser/pretty-printer the case theorem for the type in question. This is done by means of the ML function `declare_cases`.

For example, we enter

```
declare_cases num_CASE
```

In the same way, we can declare the CASE-theorems of other recursive types, e.g.,

```
declare_cases list_CASE
```

where `list_CASE` is the name of the theorem

```
⊢ ∀f0 f1 f2.
    (list_CASE f0 f1 NIL = f0) ∧
    (list_CASE f0 f1 (CONS h t) = f1 h t)
```

6.3 Defining a procedure

The parser/pretty-printer has a facility for defining procedures. The ML function `define_procedure` lets us enter the definition of a procedure using the parser/pretty-printer syntax. We enter

```
#define_procedure
# "PROCEDURE fpre(x)
#    RETURN
#      CASE x OF 0:     0
#              | SUC n: n
#      END";;
```

The system then stores the definition of `fpre` under the name `fpre_DEF`, and binds it to the same name in the current ML-environment as well. Without pretty-printing, the definitional theorem is

```
⊢ fpre = assign λx. (num_CASE 0 (λn. n)) x
```

but the pretty-printer prints it as

```
⊢ PROCEDURE fpre(x)
    RETURN
      CASE x OF 0: 0
              | SUC n: n
      END
```

Here the turnstile shows that this procedure is actually defined (or proved) to be equal to the constant `fpre`.

Chapter 7

The proof checker and its verification

This chapter describes the formally verified proof checker. It has been specified as a number of procedures, according to the theory described in Chapter 5. Joakim von Wright's theory of higher order logic proofs [vW94b] provides the data types that the program handles. We use the parser/pretty-printer syntax described in Chapter 6 to make the procedures readable.

7.1 Structure of the proof checker

The general principle is that we have defined one procedure for each function defined in the subtheories that make up the HOLproof theory. The complete list of definitions can be found in Appendix I.

The methodology permits us to develop these in arbitrary order. The presentation below starts from the lower level procedures that only call predefined HOL functions. We then move to higher-level procedures which call the lower-level ones. We have also implemented some predefined HOL functions, in particular some higher-order list handling functions.

For each procedure we prove a correctness theorem. Most of them are unconditional, but there are a few conditional correctness theorems as well.

7.2 Sample procedures and correctness proofs

We shall now show a few procedures and briefly describe some correctness proofs. The complete list of procedures is given in Appendix I and all the correctness theorems can found in Appendix J. The complete proof scripts are in the following files in the PV directory: `mk_PVbasic`, `mk_PVtype`, `mk_PVpterm`, `mk_PVinf` and `mk_PVproof`.

7.2.1 Procedures for lists and sets

First of all, we have a number of procedures that implement functions that handle lists and sets. For example, the procedure `fmem1` checks whether the first argument occurs as the first component of an element in the second argument (the second argument is a list of pairs):

```
fmem1_DEF
  ⊢ PROCEDURE fmem1 (x,l)
      VAR r = F;
      WHILE ¬(l = []) ∧ ¬r DO r,l := (x = FST(HD l)),TL l;
      RETURN r
```

Structure of a correctness proof The correctness proof for this procedure is

```
⊢ ∀x l. fcall fmem1 (x,l) = mem1 x l
```

It is proved using standard verification methods, using the proof rule (see Chapter 5)

```
⊢ conjunctive c ∧ strict c ∧ (∀u0. correct (λu.u = u0) c (λv.v = f u0))
    ⇒ (fcall c = f)
```

We set up the goal

```
"∀x l. fcall fmem1 (x,l) = fmem1 x l"
```

and match it against the theorem. The resulting conjunctivity and strictness goals are proved automatically by ML functions that are part of the `RefCalc` theory. Left to prove is the goal

```
"∀u0. correct (λu.u = u0)
              fmem1
              (λv.v = mem1 (FST u0) (SND u0))"
```

Since `fmem1` is a sequence of three commands (the initialisation, the loop and the `RETURN` command), we supply two intermediate assertions. Both are obvious and the only goal which is not proved automatically is the one for the loop:

```
"correct (λ(r,x,l).(r=T) ∧ (x=FST u0) ∧ (l=SND u0))
         (WHILE ¬(l=[]) ∧ ¬r DO r,l := (x=FST(HD l)),TL l)
         (λ(r,x,l).r = mem1 (FST u0) (SND u0))"
```

This goal is matched against the loop correctness rule, with invariant

```
"λ(r,x,l). (x = FST u0) ∧ (mem1 (FST u0) (SND u0) = r ∨ mem1 x l)"
```

and with termination function `LENGTH l`. This produces a few trivial subgoals and a simple subgoal for the body of the loop (an assignment).

Most correctness proofs are similar to this one: intermediate assertions, invariants and termination functions are supplied, and trivial subgoals are proved automatically. In some cases, the correctness proofs for loop bodies are non-trivial and in some cases the invariants are quite elaborate.

A higher-order example An example of a higher-order list-handling procedure is the procedure which implements the function `MAP`:

```
fmap_DEF
  ⊢ PROCEDURE fmap (f,l)
      VAR r = [];
      WHILE ¬(l = []) DO r,l := SNOC(f(HD l))r,TL l;
      RETURN r
```

Higher-order functions are treated exactly like ordinary functions. The correctness theorem for `fmap` is

```
⊢ ∀x l. fcall fmap (f,l) = MAP f l
```

and it is also proved using standard verification methods. In this case, the loop subgoal is

```
"correct (λ(r,x,l). (r=[]) ∧ (f=FST u0) ∧ (l=SND u0))
         (WHILE ¬(l=[]) DO r,l := SNOC(f(HD l))r,TL l)
         (λ(r,x,l). r = MAP (FST u0) (SND u0))"
```

and it is proved using invariant

```
"λ(r,x,l). (f = FST u0) ∧ (MAP (FST u0) (SND u0) = APPEND r (MAP x l))"
```

and termination function `LENGTH l`.

7.2.2 Low-level proof checker procedures

By low-level procedures we mean procedures that do not call other proof checker procedures. These are procedures that implement simple functions on Types, Pterms and `inferences`.

A trivial example Many of these procedures are trivial, like the one checking whether a Pterm is a constant:

```
fIs_Const_DEF
  ⊢ PROCEDURE fIs_Const (t)
      RETURN CASE Pterm t
               OF Const s ty . T
                | Var x . F
                | App t1 t2 . F
                | Lam x t1 . F
             END
```

A nontrivial example A nontrivial example is the function which implements `Pfree`:

```
⊢ PROCEDURE fPfree (x,t)
    VAR r = F, l = [t];
    t := Var(``,Tyvar ``);
    WHILE ¬(l = []) ∧ ¬r DO
     (l,t := TL l,HD l;
      r,l,x,t :=
       CASE Pterm t
         OF Const s ty . F,l,x,t
          | Var y . ((y=x) →  (T,[],x,t) | (F,l,x,Var y))
          | App t1 t2 . F,CONS t1(CONS t2 l),x,t
          | Lam y t1 . ((y=x) →  (F,l,x,t) | (F,CONS t1 l,x,t))
       END);
    RETURN r
```

This procedure is a good example of our iterative implementations of functions that have been defined by recursion. The head of the local list variable `l` always contains the next term to be inspected. Whenever that term contains subterms, these are "pushed" onto the list. This is reflected in the invariant used in the correctness proof; the crucial conjunct of the invariant is

```
"λ(r,l,x,t). Pfree (FST u0) (SND u0) = (r ∨ SOME (Pfree x) l)"
```

where `SOME p l` holds if the predicate `p` holds for some element in the list `l` (so `SOME` is the dual of `EVERY`).

Termination of `fPfree` requires a measure of the size of a Pterm. We use a measure that we call `Pterm_size`, defined recursively as

```
⊢ (∀s ty. Pterm_size(Const s ty) = SUC 0) ∧
  (∀x. Pterm_size(Var x) = SUC 0) ∧
  (∀t1 t2. Pterm_size(App t1 t2) = SUC((Pterm_size t1)+(Pterm_size t2))) ∧
  (∀x t1. Pterm_size(Lam x t1) = SUC(Pterm_size t1))
```

We can then use `LISTSUM (MAP Pterm_size l)` as the termination function, where `LISTSUM` is defined to sum the elements of a list.

Some functions over Types require a similar measure, called Type_size. Definitions and basic properties of these and other auxiliary functions used in the correctness proofs are listed in Appendix H.

7.2.3 Higher-level proof checker procedures

A higher-level procedure is one which calls other procedure but which may also be called by other still higher level procedures.

A middle-level procedure Consider, for example, the procedure which implements the function Plnotfree (given a list of variables l and a term t it returns truth if no variable in l is free in t):

```
⊢ PROCEDURE fPlnotfree (l,t)
    VAR r = T;
    WHILE ¬(l = []) ∧ r DO r,l := (¬Pfree(HD l)t),TL l;
    RETURN r
```

The procedure fPlnotfree calls the HOL function Pfree. Since this function is implemented by a procedure, the correctness theorem for that procedure can be used to replace the function call by a procedure call. In fact, a simple rewrite steps give us

```
⊢ PROCEDURE fPlnotfree (l,t)
    VAR r = T;
    WHILE ¬(l = []) ∧ r DO
     r,l := (¬(fcall fPfree(HD l,t)),TL l;
    RETURN r
```

Other higher-level procedures work in a similar way. Nested function calls can be replaced by procedure calls in exactly the same way. In the cases when the correctness theorem is conditional, some theorem proving is required before the function calls can be replaced by procedure calls.

The top-level procedure The top-level procedure implements the function Is_proof:

```
⊢ ∀Typl Conl Axil.
    PROCEDURE fIs_proof (l)
     VAR r = T;
     WHILE ¬(l = []) DO
      IF OK_Inf Typl Conl Axil(HD l) ∧
         lmem(Inf_hyps(HD l))(MAP Inf_concl(TL l))
        THEN l := TL l
        ELSE r,l := F,[];
     RETURN r
```

Note the universally quantified variables Typl, Conl and Axil. They are the theory determinants which do not change within a proof, so they are treated as global constants. Thus they are not part of the state space.

After rewriting with correctness theorems, the procedure fIs_proof is

```
⊢ ∀Typl Conl Axil.
    PROCEDURE fIs_proof (l)
     VAR r = T;
     WHILE ¬(l = []) DO
      IF (fcall (fOK_Inf Typl Conl Axil)(HD l) ∧
         fcall flmem(fcall fInf_hyps(HD l),
                    (fcall fmap(fcall fInf_concl,TL l))
        THEN l := TL l
        ELSE r,l := F,[];
     RETURN r
```

Now fIs_proof calls other procedures, which in turn call other procedures and so on, until at the bottom we find procedures without further calls, or with calls to functions which we assume are basic enough so that we do not have to implement them ourselves (such as HD and TL in fIs_proof).

Thus we can say that once we have used the correctness theorems to rewrite all the procedure declarations, we have the complete verified proof checker. We do not list the rewritten set of procedure declarations separately, as they are simply the declarations listed in Appendix I, with replacements according to the theorems in Appendix J.

7.3 Possible refinements

The proof checker as described above is not very efficient. However, our first and main goal was to produce a formally verified proof checker, regardless of efficiency. The HOL theory of program verification that we have used (the RefCalc contribution) also supports refinement of programs, i.e., transformation that preserve all total correctness properties. The theory of refinements is based on Back's refinement calculus [Bac88]. Generally, it permits a number off different kinds of refinements, both to improve efficiency and change control structure and data structures. We have not performed refinements on the proof checker code, so the examples below can be seen as suggestions for further work.

First of all, we can do a number of small refinement to improve efficiency locally. In particular, whenever the same expression is computed many times in the same procedure, we can compute it once and store the result in a new local variable. This will improve efficiency, provided that the added storage requirement is not too costly.

The present formalisation of proofs is very verbose, since the hypotheses of each inference are written out in full. We could make storage requirement much smaller if we used line number references instead. Thus an inference would refer to the numbers of the lines where the hypotheses were proved, rather than copying the hypotheses. Switching to this representation of proofs would be a *data refinement* step. Data refinement can be verified within the existing HOL theory of refinements, so we could actually perform this transformation and still have a guarantee that the proof checker works correctly.

The change in data representation to using line numbers in proofs would also facilitate a dramatic improvement in time complexity. We could make the checker work in two passes, never having to store the whole proof in a single data structure. On the first pass, the checker would record the last line on which each conclusion is used as a hypothesis. On the second pass, when the actual proof checking takes place, the space used to store a line could be freed as soon as the line is not used as a hypothesis in further inferences. This is a refinement which changes the control structure of the program while preserving its input-output behaviour, i.e., while preserving total correctness.

Chapter 8

An implementation of the proof checker

The proof checker that was verified in the preceding chapter can not be run on a computer as such. It is specified in a language which is very close to many imperative programming languages, but it still requires translation into a particular language. Preferrably, this language should have facilities for defining recursive data types, in addition to assignments and a standard set of imperative control structures.

One way would be to implement the data types (lists, types, terms, sequents and inferences) in some traditional imperative language (such as C), and from that point on translate the procedure definitions given in the Chapter 7. The implementation of the data types would then need to be verified separately in some way.

Another possibility is to use a functional language which has recursive data types. We have chosen this approach, using ML (the version in which the HOL system is implemented). This is feasible, since ML has all the imperative constructs that we need: assignments (to reference variables) and control structures.

8.1 Implementation principles

Each procedure is easily translated into a corresponding ML function definition. The input parameters and the local variables are represented by local reference variables and assignments by ML assignments.

Control structures are mapped to their correspondents: `WHILE` to `while` and `IF` to `if`. The `if` statement will sometimes contain an extra evaluation of `()` (of type `void`) at the end of either branch, to guarantee that both branches have the same type (this is required by th ML type-checker). For example,

```
"IF x = FST(HD l) THEN
   y,l := SND(HD l),NIL
 ELSE
  l := TL l"
```

is translated into

```
if x = fst(hd l) then
  (y,l := SND(HD l),NIL; ())
else
  (l := tl l ; ())
```

The `RETURN` statement at the end of each procedure is represented by evaluation of the expression that is to be returned.

Finally, expressions are generally translated into their ML equivalents. The `CASE` construct is represented by the pattern matching facility of ML, so that, for example,

```
"CASE n OF 0: 0 | SUC m: m END"
```

is translated into

```
(fun 0 . 0 | SUC m . m) n
```

Example As an example of the translation of a procedure, we consider the list membership procedure

```
"PROCEDURE fmember(x,l)
  VAR r=F;
  WHILE ¬(l=[]) ∧ ¬r DO
   r,l := (x = HD l),TL l;
  RETURN r"
```

becomes the following ML function definition:

```
let fmember(X,L)
  letref x,l,r = X,L,F in
  while (not (l=[])) & (not r) do
   r,l := (x = hd l),tl l;
  r;
```

where we use the built-in head and tail functions of ML: `hd` and `tl`.

Uninitialised local variables One feature of our language that does not map directly to ML is the use of uninitialised local variables. In order to avoid this problem, we have implemented procedures with uninitialised local variables so that an initialisation is given explicitly, as an extra parameter of the function. For example, the association list procedure

```
"PROCEDURE fassoc(x,l)
  VAR y;
  WHILE ¬(l=NIL) DO
   IF x = FST(HD l) THEN
    y,l := SND(HD l),NIL
   ELSE
    l := TL l;
  RETURN y"
```

is translated into

```
let fassoc(X,L,A)
  letref x,l,y = X,L,A in
  while not(l=[]) do
   if x = fst(hd l) then
     (y,l := SND(HD l),NIL; ())
   else
     (l := tl l ; ());
  y;
```

Even though we have not proved that this yields the same result, the following meta-level argument is valid: Since we have proved a correctness theorem for the procedure with an unspecified initial value for the variable in question, we know that the procedure works correctly for arbitrary initialisation.

In fact, we could prove (inside HOL) a theorem to this effect for each procedure with an uninitialised local variable (the checker actually contains only two such procedures).

Theory determinants The theory determinants (the type structure, the signature and the list of axioms) were treated in a special way in the proof checker procedures. Since they remain unchanged during a proof, they are not part of the state space. In the ML implementation, we implement them with reference variables: `Typl`, `Conl` and `Axil`. This allows us to define all the procedures once and for all, regardless of what determinants are going to be used in a particular proof.

Theory determinants can be entered by assigning to the reference variables. This can be seen in the example in Appendix L.

8.2 Sample functions

Below we show some sample function definitions. For a full listing, we refer to Appendix K.

As a simple example we show the implementation of the procedure `fPfree`:

```
let Pfree (X,T) =
  letref r,l,x,t = false,[T],X,T in
  t := Var(``,Tyvar``);
  while (not l=[]) & (not r) do
    (l,t := tl l,hd l;
     r,l := (fun Const(s,ty) . false,l
              | Var y       . if y=x then (true,[]) else (false,l)
              | App(t1,t2)  . false,t1.t2.l
              | Lam(y,t1)   . if y=x then (false,l) else (false,t1.l)
            )t
    );
  r;;
```

This is a straightforward translation of the verified procedure:

```
⊢ PROCEDURE fPfree (x,t)
     VAR r = F, l = [t];
     t := Var(``,Tyvar ``);
     WHILE ¬(l = []) ∧ ¬r DO
      (l,t := TL l,HD l;
       r,l,x,t :=
        CASE Pterm t
          OF Const s ty . F,l,x,t
           | Var y . ((y=x) →  (T,[],x,t) | (F,l,x,Var y))
           | App t1 t2 . F,CONS t1(CONS t2 l),x,t
           | Lam y t1 . ((y=x) →  (F,l,x,t) | (F,CONS t1 l,x,t))
        END);
     RETURN r
```

8.3 Efficiency considerations

The proof checker we have implemented is not very efficient. It would be tempting to make some optimisations in the ML code to make it more efficient. However, our goal was to implement a direct translation into ML of the procedures that we have verified using HOL. Thus any change should be done in these procedures first. This can be done using the refinement methodology described in Section 7.3.

Chapter 9

Conclusion

The proof checker verification described in this report can be seen as an example of verification of full-size software. The methodology that we have used permits complete verification, using HOL, of large programs built from functional procedures.

A proof checker for higher order logic is an unusual kind of software, because it works with data that are not generally handled by programs: types, terms and inferences. This means that we had to build up a large theory in HOL, before we could start declaring procedures and proving facts about them.

On the other hand, the HOL theory of proofs was built up with the proof checker in mind. For example, function definitions were intentionally made as "algorithmic" as possible. This often (but not always) makes it easy to implement them as procedures and to come up with the necessary invariants.

Connection to proof recording Version 2.02 of HOL88 permits proofs to be recorded using the library `record_proof`. Inside HOL, proofs are represented by lists of inferences which are similar to the notion of proof that we have used. Proofs can also be stored in files. This output uses a compact, LISP-style syntax.

Our proof checker could be adjusted to check proofs that are in the format used by the `record_proof` library. However, that would require some inferences to be expanded into primitive inferences. This is because the HOL system has more than 30 basic kinds of inferences (rules that are not expanded to more basic inferences), even though the HOL logic has only eight primitive inference rules. Our theory of proofs is based on the eight primitive inferences, while the `record_proof` library outputs proofs containing nonprimitive inferences. Thus we would first need a program which translates the proofs that `record_proof` has stored into a form that contains only primitive inferences. Such a program would let us try out our proof checker on larger proofs.

Part V

Metatheory and Reflection in Theorem Proving: A Survey and Critique

Chapter 10

Extending theorem provers

Computer theorem provers typically implement a certain repertoire of inference mechanisms. For example, they may solve tautologies, do first order reasoning by resolution, perform induction, make simplifying rewrites and allow the user to invoke other more delicate logical manipulations.

Certain systems (e.g. fast tautology checkers) are only useful for applications in a quite restricted field. Others, by virtue of the generality of the logic they implement and the theorem proving procedures they provide, are of wider applicability, whether or not such wider applicability was an original design goal. HOL was originally intended mainly for the verification of hardware, but has subsequently been applied to software and protocol verification, work in other embedded formalisms like temporal logic and CCS, and even in pure mathematics.

When a theorem prover is of such general applicability, it is difficult for the supplier to provide a basic repertoire of theorem proving facilities which is adequate for all purposes. The most desirable policy is to make the theorem prover extensible, i.e. provide a facility for augmenting this basic repertoire. At its simplest, this might consist of a macro language to automate certain common repeated patterns of inference. However this does not address the question of implementing radically different proof procedures from those already included. In particular a macro language is likely to have limited facilities for direct construction of terms and formulas, since one has to demarcate the valid inferences somehow – one can't allow arbitrary formulas to be nominated as 'theorems'. So what are the options?

1. If some new inference rule proves useful, simply augment the theorem prover's basic primitives to include it.

2. Allow a full programming language to manipulate the basic rules, so that users may write arbitrarily complex inference rules which ultimately decompose to these primitives.

3. Incorporate a principle of reflection, so that the user can verify within the existing theorem proving infrastructure that the code implementing a new rule is correct, and add that code to the system.

We should say that some theorem provers adopt a mixture of these policies. For example Nuprl has 'sacred' and 'profane' forms. In its sacred incarnation, all inference is done by decomposing to the primitives rules of Martin-Löf type theory. However in practical applications, various additional facilities such as binary decision diagrams and arithmetic decision procedures have sometimes been grafted onto it. Furthermore, the third option has been investigated by Nuprl researchers over the course of many years; we shall discuss this further below.

If the first of the three options is taken, then considerable care should be exercised to ensure that the proposed extension is sound. If users are allowed to tailor the system themselves, what are we to make of claims that a theorem has been proved using that system? If a user's bespoke variant is unsound, they are no longer using the original theorem prover at all, but some incorrect mutation of it. The supplier will be unwilling to concede that the proof has been performed to a satisfactory standard. It can hardly be held up as meeting an objective standard of correctness. In a small user community (e.g. within a single research establishment) consultation with peers may help to achieve a considered view of how sound an enhancement is. However in a wider community with people at work on personal projects, this is not realistic.

In practice then, for sociological reasons, major changes must be implemented by the supplier. This may be enforced by making the system 'closed' (no source code available, for example). However the suppliers still need to exercise careful judgement over which new facilities to add. It's very tempting to add just one more 'obviously true' fact, and reason by induction that the addition of n such 'obviously true' facts will not compromise the system. Experience shows that such 'obviously true' facts are frequently not true at all. Perhaps the suppliers will attempt to verify, informally or formally, that the code with which they are augmenting the system is correct. If this is done formally, then the approach looks similar to the third (reflection), but reflection has the distinguishing feature that the proof is conducted *in the existing theorem prover*.

Even with careful thought, this does not seem a principled way of making a reliable system. Nevertheless we do not mean to suggest that it is an intellectually disreputable policy. On the contrary, if the purpose of theorem proving is to highlight flaws in informal reasoning, it may be most efficient to add lots of facilities in an ad-hoc way in order to get quickly to the interesting and errorprone parts of the reasoning. Certainly, a final assertion of correctness means less than it otherwise might, but the bugs that are found may emerge more quickly. For example, interesting work on a 'hybrid' system including HOL and a fast symbolic model checker is reported by [JS93a]. This allows one to tackle leading-edge problems in hardware verification while still taking advantage of the higher level of abstraction that HOL permits.

Such a policy has been argued for by [Rus91], and one can point out similar ideas elsewhere. For example, some work on floating point verification at ORA reported by [HM92] aims to prove correctness asymptotically as the precision of the arithmetic tends to infinity. Though this is useless at giving true error bounds, experience shows that it does quickly highlight many bugs in software. Another example: computer algebra systems are widely used (much more so that theorem provers) despite sometimes giving wrong answers.

Nevertheless, the remainder of this report will assume that we are trying to construct a highly secure theorem prover, and accordingly will focus on the second and third alternatives. First, we will examine in fairly general terms the issues of metatheory and reflection. Then we will look at the LCF approach to theorem proving, and at various forms of reflection.

Chapter 11

Metatheory and reflection

Various antinomies have been discovered since antiquity, which were classified by [Ram26] into 'logical' and 'epistemological' contradictions. The classic example of a logical paradox is Russell's[1] which arises by considering "the set of all sets which are not members of themselves", or the equivalent using predication instead of set membership. These paradoxes all seem to involve some kind of self-reference inside the logical system, and have been avoided by adding a notion of type, either explicitly as in Type Theory or implicitly as in cumulative set theories or those based on stratified formulas.

Typical examples of epistemological paradoxes are the Liar ("this statement is false") and Santa Claus[2] ("if this statement is true then Santa Claus exists"). These also involve a kind of self reference, but involving questions which go beyond mathematics – for example a *sentence* referring to itself. In order to avoid them it is usual in formal logic, following [Tar56] and [Car37], to enforce a strict separation between the object logic and the metalogic (which is used to reason about the object logic).

In a formal logical system, it may sometimes happen that the most natural form of reasoning is metareasoning. For example, inferences in Hilbert-style proof systems are greatly eased by using the Deduction Theorem, which states that $A \vdash p \Rightarrow q$ if and only if $A \cup \{p\} \vdash q$.

[1] Zermelo arrived at the same paradox at least as early as Russell but chose to make little of it.

[2] Gödel's theorem can be viewed as a formalized version of the Liar, and Löb's of Santa Claus.

11.1 Metatheorem or higher order theorem?

When one discusses a particular formal system, there is a straightforward distinction between an object-level theorem and a metatheorem. Sometimes a metatheorem may say essentially the same as an object level theorem. For example, the metalevel statement $\vdash_{object} \phi$ corresponds to the object level statement ϕ. In general it may be impossible to state a object level theorem with the same import as a metatheorem. Nevertheless it is crucial to note that *what can be stated at the object level depends critically on the nature of the object level formal system.* Consider the following breathtakingly categorical statements from [AW80]:

> ...if the reasoning system you are dealing with has no capability of explicitly representing metatheoretic knowledge, many of the statements in elementary math books cannot even be expressed. This might not be interesting if such meta-statements never appeared in practice. On the contrary they arise *very* often in mathematics books ...

and [ACS86]:

> ...in reading a book on algebra one realizes that most of the stated lemmas and theorems are in fact metatheorems.

No instances of these putative metatheorems are cited. However [Mat94] is more explicit:

> Mathematics is not done with a proof development system in quite the same way as it is done in a textbook, even when the two look like one another. For instance in a book on algebra one might read:
>
> > 'If A is an abelian group, then, for all a, b in A, the equivalence
> >
> > $$\overbrace{(a \circ b) \circ \ldots \circ (a \circ b)}^{n\ times} = \overbrace{(a \circ \ldots \circ a)}^{n\ times} \circ \overbrace{(b \circ \ldots \circ b)}^{n\ times}$$
> >
> > holds'
>
> ...On the other hand, instead of a book, imagine a proof development system for algebra; there the theorem cannot be stated, since it is not a theorem of abelian group theory, it is, rather, a metatheorem, a theorem *about* abelian group theory.

Underlying this is the implicit assumption that we are talking about an axiomatization in first order logic. In a higher order logic, or in an embedding in set theory (conventional foundational systems for mathematics) iterated operations and algebraic structures are definable, and the argument has no force. To take this argument to its extreme, if I take simply arithmetic statements involving ground terms as my 'logic' then anything with a variable in it is a metatheorem. This is not a frivolous point. According to [Edw89] a major part of Kronecker's objections to 'Cantorian' mathematics was that for him it made no sense to talk about 'all functions' or even 'an arbitrary function', even though the corresponding statement for a particular instance was something he found perfectly acceptable. A comment of similar import made by the referees of the paper by [AW80] is paried thus:

> ...in elementary algebra we are taught how to manipulate equations ... This manipulation of equations (i.e. syntactic expressions) is straightforwardly metamathematical. It is the ability to do this *directly* that makes our formalization attractive.

This seems to be making the weaker claim that proof strategies and manipulative techniques, which connect mathematical results and are used to derive them, can be said to be outside the formal system. This is true – however it is not clear that a purely algorithmic metalogic (i.e. the ML programming language) is inadequate for this, and LCF-style systems have it. Is it evident that there is any need for a more elaborate metatheory? Well, one sometimes sees in mathematics books assertions like 'the other cases are similar', 'we may assume without loss of generality that $x < y$', or 'the case of an upper semilattice is treated analogously'. These have the flavour of metatheorems in that they relate similar *proofs* rather than *theorems*. But bear in mind that when presenting proofs informally, this is just one of many abbreviatory devices used. One way of formalizing statements like this is as metatheorems connecting proofs. But another is simply to do (or program the computer to do!) all the cases explicitly. There seems to reason to suppose this is inadequate in practice. Performing similar proofs in different contexts may anyway be a hint that one should be searching for a suitable mathematical generalization to include all the cases. For example, the similarity of the proofs of arithmetical theorems for different kinds of limits (pointwise limits of real functions, limits of real functions at infinity, limits of real sequences) leads to more general limit notions like nets and filters – see for example [Dud88].

Of course, we are not denying that there are many fascinating things one can do with proofs. For example, some very interesting work has been done on finding computational content of classical proofs via a generalized double-negation translation in Nuprl; see for example the paper by [CM90]. There are lots of other interesting avenues of investigation, for example translating proofs of analytic number theory into very weak systems of arithmetic, as described by [Tak78]. However we claim:

1. Such interests are largely the preserve of logicians rather than 'mainstream' mathematicians, and are of no obvious relevance in verification work.
2. It is unusual to want to feed the results back into the system; their interest arises as an independent piece of mathematics. Metalogic is just another branch of mathematics.
3. Many of the transformations are algorithmic, and can be investigated using only minor modifications of the LCF method (for example, storing proof trees as concrete objects, as already done by several systems including Nuprl).

11.2 Metalogical frameworks

In response to the proliferation of theorem provers for different logics, a number of 'generic theorem provers' or 'logical frameworks' have been developed. Well-known examples include the Isabelle system described by [Pau90], Lambda Prolog described by [FM88] and the LF system described by [HHP87]. These provide a simple metalogic in which different object logics can be represented. Proof procedures are provided which are intended to be applicable to a wide variety of logics. The objective is to avoid needing to write a new theorem prover from scratch for each new logic one is interested in. Users, though they may in fact be proving facts at the metalevel, often need not be aware of the nature of the metalogic.

Usually the metalogic chosen is rather weak; Isabelle for example uses intuitionistic second order logic based on lambda calculus, without any numbers or recursive data types. This is ideal for formalizing object logics; in particular, binding constructs like universal quantifiers can be implemented using lambda-calculus binding in the metalogic. The framework is designed to support object-level reasoning in a uniform manner. However it is not sufficient to perform much metatheoretic reasoning beyond the fact that a particular object-level statement is a theorem. In fact the metalogic is usually so weak that even proving the Deduction Theorem is impossible.

It is of course possible to extend the metalogic. As was pointed out by Randy Pollock, the LF logic can be embedded as a natual subset of the much more powerful Calculus of Constructions, as implemented in the Coq and LEGO provers. Pollock, in unpublished work, proved the Deduction Theorem for a Hilbert-style propositional logic in the LEGO prover, and [Tay88] verified a tactic for LF encodings in LEGO, which proves equations in an associative semigroup using list equality of the fringes of the term trees.

Alternatively one can break away from traditional framework logics. [Mat94] explores the use of Feferman's FS_0 system as a metalogic. This has independent interest as FS_0 was proposed theoretically for precisely the purpose of representing formal systems – perhaps even as an explication of what constitutes a formal system. Matthews gives a detailed sketch of how to prove cut-elimination for propositional logic in FS_0 (the algorithm is simple enough, but the usual textbook formulation uses induction over higher ordinals, not directly available in FS_0, for the termination proof). For a general plea for more extensive use of metatheoretic reasoning, see the paper by [BC91].

Metatheoretic proofs share something with verification proofs: they are mostly a detailed and messy technical execution of a fundamenatally simple, though often ingenious, idea. Nevertheless, as the reader may have guessed from the examples cited, the existing work on metatheoretic proof in computer theorem provers is surprisingly limited. Probably the most substantial example is the proof of Gödel's first Incompleteness Theorem by [Sha94], but apparently this was done without any interest in exploiting it metatheoretically in the theorem prover.

11.3 The basic idea of reflection

The term 'reflection' is applied to a number of distinct but loosely connected ideas in logic and computation. The key idea of reflection is that a logical system or computer program is used to reason or compute *about itself*, i.e. to make a connection between its own object and meta levels.

This may be in order to deduce facts or perform computations which cannot be done in the object system, but are nonetheless seen by reflecting upon the system from outside, to be in some sense true or admissible, or simply useful. On the other hand the objective in logical systems may be simply to deduce things which could (in principle) have been obtained, albeit in a lengthier fashion (intractably lengthy perhaps) by direct inference in the object logic.

A reflection principle, then, is an expansion of the methods of inference of the logic. It allows transfer between formalized statements of provability in the metatheory and object level assertions. Our main interest will be in the use of metareasoning to make inference more efficient; a detailed exploration is given in a later section. We should note that we may in fact strengthen the logic by adding reflection principles. That is, the set of theorems which can be derived by explicitly writing down sequences of inferences (briefly: the closure of the primitive inferences under programming constructs) may be properly smaller that the set of formulas which can be proved (nonconstructively for example) in a formal metatheory. Sometimes this is incidental, but it may be deliberate. For example, [Mat92] proposes the use of logic-strengthening reflection principles, in the context of computerized theorem proving.

11.4 Seamless use of metatheory

The power of metatheoretic reasoning arises from the ability to step back from the constraints of a formal system and exploit syntactic properties, and their connection with semantics. For example the Löwenheim-Skolem theorems prove that there exist groups, rings, fields, Boolean algebras and any first order axiomatizable structures of arbitrary infinite cardinality. By means of a *syntactically based* classification of structures we are able to prove a theorem of attractive generality. More nice examples are surveyed by [Kre56], and some quite elaborate applications of model theory to algebra are given by [Rob63] and [KK71].

Higher order logic provides sufficient resources to carry out what is essentially metatheoretic reasoning (in the sense that it operates on syntactically demarcated subsets of the logic) without in any way tampering with or extending the simple inference mechanisms. Datatypes representing syntactic objects like terms can be set up, and interpretation functions into the logic defined to connect the internal representation to the corresponding constructs in logic. For example, the usual set-theoretic semantics of first order logic with respect to some interpretation and environment (valuation) can be defined in an obvious manner (bound variables are the only nontrivial consideration, and they are easily dealt with). Indeed, the simple logic of the Boyer-Moore prover, of which more later, allows an evaluation function (`MEANING`) to be defined on terms under a given variable assignment. As [BM81] emphasize:

> There is nothing magic or "meta" about this function.

For reasons connected with the undefinability of truth demonstrated by [Tar56] it is not possible to define a semantics for the *whole* logic inside itself. Nevertheless useful subsets can be dealt with, and in rich type theories such as Nuprl, stratification by universe level allows one to come very close to this ideal. [How88] has actually implemented such a scheme in Nuprl and verified an embedded term rewriting system, including matching algorithms. This appears to go beyond any of the work on reflection proper in Nuprl which we will discuss below. The stress in Howe's work was on verifying theorem proving procedures. More recently, the present author has been experimenting with proving some more 'mathematical' results using similar techniques in HOL.

Let us reiterate: in many cases everything can be done in the simple formal system as it stands, if that system is higher order or supports set theory. Full internalization of semantics is impossible by Tarski's theorem, which may show itself in practice by the type system providing an obstacle. Nevertheless a lot of interesting things, including more or less all the usual metatheorems about first order logic, can be done in this way.

Chapter 12

The LCF approach to theorem proving

The Edinburgh LCF system, desribed by [GMW79], was developed in the mid 70s in response to dissatisfaction with highly automatic provers and with low-level proof checkers. A prover of the latter kind was developed as Stanford LCF by [Mil72] and provided much of the motivation. The LCF system was subsequently reengineered and improved, resulting in Cambridge LCF, described by [Pau87]. The original system implemented a version of Scott's Logic of Computable Functions (hence the name LCF), but as emphasized by [Gor82], the LCF approach is applicable to any logic.

In LCF-like systems, the ML programming language is used to define data types representing logical entities such as types, terms and theorems. The name ML is derived from 'Meta Language'; effectively LCF systems *do* have a kind of meta-logic, but an explicitly algorithmic one, where the only way to prove something is a theorem is to actually prove it. Since its LCF-related origin, the ML language has achieved a life of its own. It is a higher-order functional programming language with polymorphic types together with a typesafe exception mechanism and some imperative features. A reasonably formal semantics of the Standard ML core language has been published – see [MTH90] and [MT91].

In LCF-like systems, a collection of ML functions are provided which produce theorems; these implement primitive inference rules of the logic. For HOL there are 8 such rules; Nuprl, which implements Martin-Löf's constructive type theory, requires well over a hundred. For HOL, these primitive inference rules have been proved sound via a set-theoretic semantics devised by Andy Pitts and published in [GM93]. [Pot92] has also proved that they are complete with respect to Henkin's general models. Thus, provided only the primitive ML inference rules are used, any theorem resulting will be 'correct'. Such adherence to the primitive rules is enforced by encoding theorems as an ML abstract datatype whose only constructors are the primitive inference rules of the logic. Anything of type `thm` must have arisen by applying the primitive rules.

This security is attractive, but proofs of nontrivial facts in terms of primitive inferences are often extremely tedious. Using the primitive rules directly means HOL is functioning as a 'proof checker' for proofs at an extremely low level. Such an approach is not to be scorned, since some systems exist which only function at this level, yet have been used to check substantial parts of mathematics. A pioneering example was the work by [vBJ77], formalizing the famous book by [Lan66] in the AUTOMATH system of [dB80]. More recently, a large and disparate body of mathematics has been checked in the Mizar system described by [Rud92], and there is even a journal 'Formalized Mathematics' devoted to Mizar formalizations[1].

Nevertheless, it is attractive to be able to direct proofs at a higher level, rather than perform proofs of trivial tautologies or facts of arithmetic explicitly. This is even more important in verification applications than in pure mathematics, since the proofs, though shallower, tend to be much more involved and intricate, since there are few abstraction mechanisms to hide the layers of complexity; though see [Mel93].

This can be done in LCF provers: using ML programming, one can program complicated patterns of inference, provided they ultimately decompose into primitives. Hence one can write derived inference rules which work at a higher level. In order to exploit them, a user may just call the functions, without understanding how they decompose into primitives. Only the original implementor of the derived rule need understand that. Because of the abstract datatype, a user can have equal confidence in the correctness of the resulting theorems, since ultimately they must arise by primitive inference.

The user or derived rule is still free to decompose and rebuild terms and formulas in any way desired in order to decide on proof strategy etc. – and the extremely simple term structure of HOL makes this very convenient. Nevertheless all *theorems* must be built by primitive inference. In this way, LCF can offer the security and controllability of a low level proof checker together with the power and flexibility of a more sophisticated prover, provided someone is prepared to put in the work required to provide useful derived inference rules.

For the sake of completeness, we should add that 'morally', a true LCF implementation has rather simple primitive rules. Primitives in HOL range in complexity from reflexivity of equality to simultaneous parallel substitution.

[1]For more information about this journal, which is surprisingly inexpensive, contact: Fondation Philippe le Hodey, MIZAR Users Group, Av. F. Roosevelt 134 (Bte7), 1050 Brussels, Belgium, fax +32(2)6408968.

They do not include rewriting or arithmetic decision procedures. An alternative policy, exemplified by the PVS system described by [ORS92], is to make the primitives powerful. This means that less programming is required to build up a suitable set of high-level operations, and that these high-level operations may be more efficient. However it has the defect that it's much harder to be confident about the correctness of such complex primitives, and in any case which primitives are useful can depend on context (for example, users of an embedded formalism may require unusual proof procedures).

12.1 The advantages of LCF

One obvious advantage of the LCF approach is that it offers a good degree of security that a purported theorem really is a theorem. The critical code is confined to that implementing the primitive rules (and any support functions they use). However it should be admitted that there is also a dependence on the correctness of the ML implementation, in particular the correctness of its type system.

If for this or other reasons one finds the assertion of security unconvincing, it is quite easy to change the system to record proofs. This has actually been done in HOL (as documented previously in this report), so that each primitive inference is logged. Such a proof may then be checked by a simpler external program, written in any chosen programming language. As described previously, such a proof checker has been implemented and verified in HOL. An important part of this project is to analyze carefully what constitutes a HOL proof, and to this end the notion of HOL proof has itself been formalized in HOL by [vW94b]. In the process a few errors and obscurities in the logical core were uncovered. For an overview of various research in HOL connected with these ideas, see [GHH$^+$94]

Whether or not one personally considers it worthwhile, such proof checking is recommended by certain procurement standards for safety critical software, e.g. [MOD91]:

> 32.3.1 In practice, it is very unlikely that Formal Proofs of any size will be created by hand. Instead, they will be developed using theorem proving assistants, which are interactive programs that carry out symbol manipulation under the guidance of a human operator. But theorem proving assistants are large programs whose correctness cannot readily be demonstrated by Formal Proof. It is, however, possible to remove the reliance on the correctness of the theorem proving assistant from the case for correctness of an application by arranging that a version of the final proof (omitting all history of its construction) is passed from the theorem proving assistant to a

> proof checker. For reasonable languages, such a proof checker could be a very simple program (perhaps ten pages in a functional programming language) that could be developed to the highest level of assurance.

Naturally, it is possible for non-LCF systems to provide a low-level proof script. See [KP94] for a discussion of adding proof logging to the NEVER system. However in making a theorem prover capable of producing primitive inferences, one is effectively writing a second, LCF-style mode for an existing theorem prover. Furthermore, if the primitive inferences are complex, their correctness can still be questioned, so perhaps even the decomposition to simple primitives will need to be emulated. It seems much more elegant to adopt the LCF approach from the start.

Security aside, the LCF approach offers great flexibility to ordinary users, who may extend the system with customized derived rules[2]. These rules do not need to be verified; if there is an error, then the proof procedure may fail, but because of the ML type discipline, will never produce an invalid 'theorem'. Users do not need to provide theoretical justification for adding a new derived rule, still less prove the correctness of the code implementing it. They simply have to design it to decompose to primitive inferences. In simple cases this is straightforward; in more complex cases it is not, but arguably it is normally easier than performing correctness proofs of the code.

12.2 The inherent inefficiency of LCF?

These advantages are undeniable; however it appears that there is a heavy price to pay: every derived rule and proof procedure must be forced into the strait-jacket of decomposing to primitive rules. Apart from seeming rather unnatural in some instances, it is hardly likely to be as efficient as a hand-coded proof procedure. This of course is not necessarily a problem; the key question is whether it is a serious constraint in practice.

Indeed, the view firmly embedded in theorem proving folklore is that it is a serious constraint, making the LCF approach ultimately untenable for practical examples. LCF has certainly been influential, but it is subsidiary features of it, such as the use of tactics to implement backward proof, or the general stress on interactive rather than fully automatic proving, which have been most widely imitated. Systems which adhere to the pure LCF philosophy are rare. In fact, of those widely used in verification examples, HOL is arguably the only system which does. Characteristic criticisms of LCF are these from [DS79]:

[2] One wit has remarked that in HOL 'the workers control the means of production (of theorems)'.

> ...the Edinburgh LCF system ...employs the device of "tacticals" to obtain a modest degree of extensibility. However an LCF tactical is limited to a fixed combination of existing rules of inference. This has the virtue that no correctness proof in our sense is needed but also have [sic] the obvious limitation that no really new inference rule can be adjoined.

from [ACV93]:

> The major drawback of this approach is that each proof procedure (even the most sophisticated ones) must ultimately invoke the basic inference rules. In many cases this turns out to be both unnatural and ineffective. Indeed, most of the decision procedures for decidable theories (e.g. the truth table method for propositional calculus) can be hardly rephrased as proof strategies based on the inference rules of the corresponding calculus. Moreover, the performance of the translated proof strategies is much less effective than a direct implementation of the original procedure.

from [RS94]:

> In our opinion the currently available validation mechanisms are not powerful enough for large applications, for example in software verification. ...For example, in HOL (and most of the other systems above) extensions are correct per construction but the application of large tactics is inefficient. For practical applications this is a bad compromise.

and from [Bas94]:

> ..., I believe that tactics alone are insufficient to provide a proper level of reasoning for interactive theorem proving and additional mechanisms for metatheoretic extensibility are required. For example, we should be able to extend a prover with new kinds of simplifiers and decision procedures that are not hampered by the need to produce a proof with primitive rules.

When encountering such trenchant criticism one naturally expects to find theoretical justification or the results of case studies to lend support to the assertions. However here, as usual, one searches in vain for any such thing. Indeed, in the same conference [Art94] writes:

> I would feel a lot more confident in predicting the way theorem prover architecture should go, if more was known about the complexity issues both theoretical and practical that we face. ...Are there natural theorem proving problems with a reasonably tractable

decision procedure but with no tractable means of finding proofs from primitive inferences?

It appears that nobody has ever answered this last question convincingly. But it is of central importance. Such is the power and intellectual simplicity of the LCF approach that it seems foolish to enter more nebulous and complicated areas like reflection unless there are some convincing reasons for supposing the LCF approach practically insufficient[3]. With basic questions such as $P = NP$ unresolved, it may be too much to expect a rigorous theoretical answer to Arthan's question. But we should be able to say something useful.

12.3 The sufficiency of LCF

We will maintain that the inefficiency of the LCF approach has often been overplayed. Our remarks will apply to HOL, and to some extent the validity of our remarks will depend on the logic's being higher order, as well as other details perhaps. Firstly there are two obvious practical counterarguments:

- HOL is currently a widely used theorem proving system. Experience shows that in the overwhelming majority of cases, the difficulty in proving a theorem in HOL is not due to inefficiency, and few users are constrained by such inefficiency. Rather, the difficulty is in providing a proof at all; the key time factor is user thinking time! Of course it may be that to alleviate this problem, more sophisticated derived rules need to be provided, and then efficiency *would* become a serious constraint. This is possible, but practical experience leads us to doubt it.
- Computers are getting faster all the time, and it may be that what today is an efficiency problem ceases to be one tomorrow. In very early versions of LCF, the author has heard anecdotal evidence of rewrites taking half an hour. Today, for a rewrite in HOL to take even one second is unusual. Of course, if the size of typical proofs expanded in time with the development of machines, this argument would no longer hold. But such a correlation seems improbable. In its early days, Unix ran a little more slowly because it was written in C rather than assembler. Even at the time the slowdown was modest considering the organizational advantages of a higher level language. Nowadays nobody considers it (and almost nobody writes assembler).

There are several more specific arguments. These are contentions which seem hard to justify by theoretical musings; rather they need case studies. However

[3] Apart of course from sheer intellectual curiosity. This is a laudable motivation but it should not be confused it with the dictates of practice. To parody Dijkstra, it is important to distinguish the research fields of 'theorem provers' and 'theorem proving'.

we believe current experience provides some strong support for them. For a detailed examination of sources of inefficiency in HOL, see [Bou93].

1. Most other proof procedures may be implemented in a fairly natural way using primitive inferences. Instead of plugging terms together in an ad-hoc manner, such transformations may be justified at each stage by matching against a suitable theorem, requiring only a few primitive inferences each time to instantiate the theorem and perform, say, a Modus Ponens step.

2. In most cases of sophisticated inference rules, inference is (or may be construed as) only a small part. Search often dominates, which can be done without requiring any primitive inferences; this need not even be written in ML or performed by HOL itself.

3. Much of the inefficiency of derived rules in HOL arises from poor (or excessively simpleminded) programming. Using more careful or sophisticated programming, many of the efficiency problems disappear.

4. There are many implementation improvements that can be applied to ML systems. Questions of primitive inference aside, ML is often an order of magnitude slower than C in comparable applications.

The HOL implementation of rewriting illustrates some of these assertions. HOL's rewriting rules and tactics take a set of equational theorems and repeatedly replace instances of the left hand sides by corresponding instances of the right hand sides in another term. The subterms to be rewritten and the number of repeated rewrites may be precisely controlled, but by default the term is searched for matching subterms in top-down order, repeatedly until no more are possible.

The initial search for matching subterms can be implemented quote independently of the primitive inferences required to perform the rewrite. In HOL, term nets are used to achieve fast lookup, and only the apparent matches thrown out by this matcher are actually tried. This part appears to dominate rewriting's efficiency, especially when there are a large number of rewriting theorems. Perhaps more sophisticated algorithms could be used; in any case it is unaffected by any need to perform inference. The rewriting of a subterm is performed using a few instantiations, and the buildup of the whole term done by iterating congruence rules for equality – not much slower than simply plugging the terms together in an ad-hoc way, especially given some optimizations implemented by [Bou93].

Now let us consider the assertions in more detail.

12.3.1 Using theorems to justify inference

The contention above was that most patterns of inference carried out ad-hoc by may be implemented in terms of primitive inference in a fairly straightforward fashion and with only a moderate slowdown. To some extent, this is particularly true of HOL, since higher order logic provides the power to encode quite sophisticated structures in theorems. We have already remarked on the phenomenon that higher order logic often provides an adequate substitute for a formal metalogic. Patterns which in first order logic require a metalanguage to express can be embedded in the HOL object logic without difficulty. For example, [Har93] defines a generic representation of univariate polynomials using lists:

```
(poly [] x = &0) /\
(poly (CONS h t) x = h + (x * poly t x))
```

Now it is possible to prove (by list induction), for example that every polynomial is differentiable. Then for any specific instance it is necessary only to instantiate this 'proforma' theorem and perform some rewrites to unwind the recursive definition. It is not necessary to repeat the whole proof every time. Using similar proformas quite sophisticated inferences are encoded, for example:

```
|- !l a b. a < b /\
           (!x. a < x /\ x < b ==> FORALL (POS x) l)
         = a < b /\
           FORALL (POS (inv(&2) * (a + b))) l /\
           (~?x. a < x /\ x < b /\ EXISTS (ZERO x) l)
```

meaning that all of a finite set (list) of polynomials are strictly positive throughout an interval if and only if they are all positive at the middle of the interval and are nonzero everywhere in it.

In fact, examples like this awaken one to the fact that HOL's logic incorporates a simple functional programming language, and many operations which could be implemented outside can alternatively be internalized in the logic and executed by primitive inference. Of course this is quite a lot slower, but it seems unlikely to mark the difference between tractability and intractability except right at the boundary of what is possible using special proof procedures. One would expect the slowdown to be linear.

Consequently it is close to being plausible that anything implementable in ML using purely functional programming can also be implemented inside the logic with only a moderate constant factor slowdown. Cases where this is not true are likely to be where the type system constrains the generality of a proforma theorem; for example one might want to state a theorem for arbitrary iterations of the function space constructor. (In more general type systems like Nuprl's, even this may be unproblematical.)

For example, it is useful to be able to justify inductive definitions using the Tarski fixpoint theorem, but to interpret 'monotonicity' not just for unary relations (sets) but relations of arbitrary arities; i.e.

$$\begin{array}{l}(\forall x_1 \ldots x_n.\ R\ x_1 \ldots x_n \Rightarrow S\ x_1 \ldots x_n) \\ \Rightarrow (\forall x_1 \ldots x_n.\ E[R]\ x_1 \ldots x_n \Rightarrow E[S]\ x_1 \ldots x_n)\end{array}$$

This particular case can be dealt with in various ways while maintaining an instantiation linear in the number of variables (indeed repeating the proof every time satisfies the constraint and does not use so many primitive inferences each time).

12.3.2 Avoiding inference

It is a fact that a lot of theorem proving procedures involve substantial amounts of search, but once the search is complete, it may be simple to produce a proof in terms of primitive inferences. For example, a resolution proof may involve searching through thousands, even millions, of clauses for a refutation. Once found, however, the path to a refutation is usually quite short, and provided the clauses and unifiers are recorded, can easily be transformed into a HOL proof using a little instantiation and propositional reasoning. Other tableau-based decision procedures for first order logic have actually been developed in HOL in this way; see work by [KKS91] and [SKK92]. Similar remarks apply to many arithmetic decision procedures, and some of these have actually been implemented in HOL as very useful practical tools by [Bou93].

Some other intersting procedures are those like the factorization of polynomials (or numbers!), the finding of antiderivatives and the solution of equations. These are difficult, computationally intensive or require sophisticated heuristics. Nevertheless, once a putative answer is found, it is a relatively straightforward matter to check its correctness – all that is required for a formal proof. This idea has been explored in [HT93] using a link between HOL and the Maple computer algebra system. More general than using external oracles to produce checkable *answers* is the idea of delegating all the proof-finding to an external 'proof planner'. This idea has been explored by [BvHHS91].

12.3.3 Other optimizations

There are a number of other techniques which may make the LCF approach more efficient. For example, in many situations (Nuprl typechecking obligations are a good example) the same trivial theorem is proved over and over again. It may be that by cacheing (aka 'memoizing') previous theorems, a substantial gain in efficiency could be obtained. An example of this situation for arithmetic theorems is described by [Bou93].

A more general technique for optimizing LCF implementations has been proposed by [Bou93]: lazy theorems. The idea is to delay the inference phase of rules until later, accumulating a function closure during 'inference' and only executing it when and if the final theorem is needed (as already happens in HOL *backward* proof – it is exactly how tactics work). This offers some potential for making interactive proof more efficient, while postponing costly inferences until later (even overnight perhaps). As it stands, the idea could more simply be achieved by 'slow' and 'fast' modes, where no inference is done in the latter. However lazy theorems may allow one to program in a more free and easy style using inference rules directly, rather than forcing programmers to separate exploratory search from inference explicitly, without incurring much efficiency penalty. Furthermore there may be special situations where inferences may be more efficiently decided on after examining the whole 'proof'; for an example, see the work on BDDs in HOL by [Har94].

12.3.4 Partial evaluation

There is a substantial research area of 'partial evaluation' which aims to optimize functional programs by precomputing parts of them. The idea is similar to the well-known idea of constant folding in compilers, but much more sophisticated. A very nice summary is the following, from [BEJ88]:

> In the large, the goal of PE is to construct, when given a program and some form of restriction on its usage (e. g. knowledge of some but not all of its input parameter values), a more efficient new or "residual" program that is equivalent to the original program when used according to the restriction.

Some HOL derived rules have been manually optimized to perform computation before they see all their arguments. For example, the rewriting rules canonicalize the rewrites and set up the term net before they are applied to the term to be rewritten. There may be considerable potential for similar optimizations to be made automatically. Ideally, whole sequences of primitive inferences might be folded down to something much more efficient. Work on applying partial evaluation to HOL is being undertaken by Morten Welinder, and preliminary results are promising. We should point out that the use of imperative features, in particular proof recording, at the level of primitive inferences, is likely to all but destroy the potential optimizations.

Incidentally, papers by [Dan88] and [TW88] explore some possible connections between partial evaluation and reflection.

12.3.5 Difficult cases

It seems, then, that the cases where a tractable proof by primitive inference is hard to find are likely to be algorithms making essential use of imperative features (arrays, shared data structures) and not allowing a cheap checking process. Binary Decision Diagrams as described by [Bry92] constitute just such an algorithm. Nevertheless it has been shown in (still somewhat preliminary) work by [Har94] that even here an acceptable implementation in terms of primitive inferences may be possible. Furthermore the second of the above palliatives does apply to some extent, since very often a well chosen variable ordering makes a tremendous difference to the efficiency of a BDD-based tautology check. Deciding on a good variable ordering can be done without requiring primitive inference.

There is a somewhat different example where a proof by primitive inference is likely to be dramatically less efficient: arithmetic. (And more generally, any situation where we end up duplicating a hardware facility inside the logic, but this seems the only real example.) Rather than using the machine arithmetic we must perform inference inside the logic even to add a couple of integers. We can use a binary or decimal representation inside the logic, and consequently the speed of arithmetic operations on numbers of size n is likely to take $O(log_2(n))$ or $O(log_2(n)^2)$. From a purely theoretical point of view of course, machine arithmetic is no different when one allows arbitrary precision. However from a practical perspective, the constant factor difference in speed may be enormous. Fortunately, manipulation of large numbers does not in any case seem to occur very frequently in typical applications.

Finally, it is worth noting that in cases where the terms involved become very large, there may arise hidden overheads; for example when instantiating a theorem with a very large assumption, a complete traversal of the assumption term is required to check that the instantiated variable is not free. It may even be that a decomposition to primitive inferences forces a retraversal many times, whereas metaknowledge might convince us that one check suffices. Sometimes this can be achieved, paradoxically, by making certain aspects of the theorem explicit in the logic, rather than relying on the way the primitive rules work. As an example, the implicit union of assumption lists which happens during certain inferences becomes very inefficient when the lists are large. It is better to form large conjunctions of the assumptions and do the 'unioning' explicitly in the logic. For some examples of this phenomenon, see [Har94]. It is not clear whether this represents a serious difficulty in any practical cases.

In conclusion, it is not clear that there are any useful practical proof procedures which are impossible to accommodate in the pure LCF approach. Indeed from a more theoretical (and unrealistic) perspective it has not been demonstrated that any proof procedure suffers more than a constant factor slowdown when implemented in the LCF style. Certainly it does not seem that many areas of verification naturally demand such procedures. The most likely exceptions are fast model-checking algorithms like BDDs which are much used in hardware verification. It is not clear that software verification makes similar demands, and almost certain that pure mathematics does not. In any case, if reflection is to represent a substantial practical advance, justifying its greater difficulty, verification of essentially imperative code is likely to be required.

Chapter 13

Other kinds of reflection

In their interesting paper [GS89] propose a distinction between a 'reflection principle' (strengthening the logic) and the process of 'reflection' (merely making the deductive process more efficient). An attempt at clarifying terminology is laudable, but we will set ourselves the more modest goal of describing four uses of 'reflection', the last of which will be the one which is most relevant to us. First the others. We should add that there are a few instances in the literature which do not fit easily into the categorization we have chosen. An example is reflection in logic programming, as proposed by [BK82] which is perhaps a blend of logical and procedural reflection.

13.1 Logical reflection

[Göd67] showed how the syntactic entities of a formal system could be coded up as natural numbers[1]. We will write the encoded form of the formula ϕ using corner quotes: $\ulcorner \phi \urcorner$. Various syntactical transformations on the encoded formulas, such as substitution, can also be defined, and finite sequences of numbers can be encoded as a single number. Now if the axiom set and the inference rules are recursively enumerable, their encoded counterparts are representable, and we can put all the pieces together in a quantifier-free representable predicate $Prov$ where:

$$Prov(p, \ulcorner \phi \urcorner)$$

[1] Similar techniques can be applied to formal systems including hereditarily finite sets or free recursive datatypes, with the possible advantage that the Gödelized form of a formula may be quite readable, instead of just a huge number.

means that p is the Gödel number of an encoded proof of the encoded counterpart to ϕ. Using a slight elaboration of this technique, with a parameter in the formula ϕ, Gödel was able to diagonalize and write down a statement which expressed its own unprovability in the system:

$$\phi \equiv \forall p.\ \neg Prov(p, \ulcorner \phi \urcorner)$$

This sentence cannot be provable, or a contradiction results, so it must be unprovable. But that means the statement is true! What's more, if the original system is 1-consistent, the formula's negation is not provable either – the logic is incomplete. This is Gödel's first Incompleteness Theorem. [Ros36] subsequently refined the proof to show that a simply consistent system must be incomplete. Gödel also deduced his Second Incompleteness Theorem, that such a formal system S is unable to prove the following statement (usually abbreviated $Con(S)$)

$$\forall p.\ \neg Prov_S(p, \ulcorner \perp \urcorner)$$

asserting its own consistency (unless the system *is* actually inconsistent, in which case it proves everything).

While Gödel's theorems show the limitations of formal systems, they also point to a systematic way of making a given system stronger. Given some axiom system S_0, a natural way of strengthening it is the addition of a new axiom amounting to a statement of S_0's consistency. This gives a new system:

$$S_1 = S_0 \cup \{Con(S_0)\}$$

What's more, this procedure can be iterated, giving S_2, S_3 and so on. The iteration can even be continued transfinitely:

$$S_\lambda = \bigcup_{\alpha < \lambda} S_\alpha$$

This was first investigated by [Tur39], who showed that the limiting system (unioning over all ordinal numbers α) arising from Peano arithmetic by repeatedly adding statements of consistency was capable of proving all true Π^0_1 (universal) sentences of number theory. For example, it could prove Fermat's Last Theorem, if true. Turing's explorations were carried much further by [Fef62], who also pointed out that the intensional description of the axioms may be significant as far as the resulting hierarchy is concerned. Feferman coined the term 'reflection principle' for an assertion, like a statement of consistency, which amounts to an expression of trust in a system of axioms (presumably so called because it arises by 'reflecting upon' those axioms from outside). To quote his own definition:

> In contrast to an arbitrary procedure for moving from A_κ to $A_{\kappa+1}$, a reflection principle provides that the axioms of $A_{\kappa+1}$ shall express a certain trust in the system of axioms A_κ.

A stronger reflection principle would be an assertion of soundness with respect to some standard model, e.g. the natural numbers for arithmetic theories.

$$\forall p.\ Prov(p, \ulcorner \phi \urcorner) \Rightarrow True(\ulcorner \phi \urcorner)$$

As it stands this so-called *global reflection schema* cannot be expressed in the logic, since it was shown by [Tar56] that there is no definable predicate $True$ corresponding to arithmetic truth. However we can express something intuitively similar by the following schema (in which ϕ is any sentence):

$$\forall p.\ Prov(p, \ulcorner \phi \urcorner) \Rightarrow \phi$$

This schema, now known as the *local reflection schema*, was also considered by Turing. Since the special case where ϕ is $\perp$ is a statement of consistency, this statement is at least as strong. What about other instances – might they be provable in the logic as it stands? Because of the property given above for the formula arising in the proof of Gödel's first theorem, that furnishes us with another unprovable instance. In fact[2] it was proved by [Löb55] that an instance of the above schema is provable precisely when the corresponding ϕ is itself already provable. By the above remark, Gödel's second theorem is a special case of Löb's theorem.

In contrast to a statement of consistency, it is possible for a consistent theory to become inconsistent by adding the local reflection schema. For example, Gödel's second theorem shows that the theory resulting from adjoining to Peano arithmetic the statement 'Peano arithmetic is inconsistent', is itself consistent. But the further addition of the local reflection schema then immediately yields a contradiction. Nevertheless it follows from Feferman's work that a 1-*consistent* theory remains so even under transfinite addition of the local reflection schema. Turing conjectured that the limiting system from repeatedly adding the local reflection schema would be properly stronger than that resulting from repeatedly adding a statement of consistency. This conjecture was refuted by Feferman: in fact the limiting systems have equal power.

[2] Assuming a few extra conditions on the provability relation beyond those required for Gödel's theorems.

Since Feferman's original coining, the meaning of 'reflection principle' has become slightly specialized towards statements like the local reflection schema which seem to make a connection between a theory and its metatheory of the form 'if ϕ is provable then it is true'. To quote [KL68]:

> By a "reflection principle" for a formal system S, we mean, roughly, the formal assertion stating the soundness of S: If a statement ϕ (in the formalism S) is provable in S then ϕ is valid.

It may be that a slight shift in the perceived metaphor is behind this changed usage: the Gödelization is a representation which mirrors the actual proof system. We hope that it accurately reflects the real picture. A still stronger schema is the so-called *uniform reflection principle*:

$$\forall p.\ Prov(p, subst[\mathbf{n}, x]\phi)) \Rightarrow \phi[n/x]$$

Here $subst[\mathbf{n}, x]\phi$ denotes the result of substituting the Gödelized form of the numeral n for the Gödelized form of the variable x. Modulo this pedantry, the assertion is just the local reflection principle extended to formulas with one free variable. It was shown by [KL68] that, with respect to first order number theory among others, the above schema is equivalent to transfinite induction up to ε_0, which was precisely the additional property used by Gentzen in his consistency proof for number theory[3]. Furthermore, Feferman showed that a transfinite iteration based on the uniform reflection schema is stronger, and in fact proves all true sentences of number theory.

Rather than stressing its role in making a logical system stronger, Kreisel and Lévy exploit the fact that the reflection schema for S is unprovable in S to yield a way of comparing the strengths of logical systems. If a system T can prove the reflection principle for S, then T is properly stronger than S. It is claimed that the intuitive significance of the reflection schema tends to make such proofs easier to find than those for a simple statement of consistency which is, as noted above, a special case and thus in principle weaker.

Finally, we should observe that although the full reflection schema is unprovable, it may happen that by suitably restricting the kinds of provability allowed in $Prov$, the analogous schema becomes provable. Such facts are called 'partial reflection schemas', following [Tro73] where the logical complexity of formulas in derivations is restricted. That was in Heyting arithmetic; a related example in ZF set theory is given below.

[3] Weil referred to Gentzen as the lunatic who justified induction on the natural numbers, ω, using induction on a higher ordinal.

13.2 Set theoretic reflection

Zermelo-Fraenkel set theory, and most other modern variants, present the set theoretic universe as a cumulative hierarchy of sets. This is built up from the empty set, and possibly a given collection of 'urelements', by iterating the powerset construction[4]. The successive levels are usually written V_α where α runs through the ordinal numbers. The recursive definition splits into two cases, for successor and limit ordinals:

$$V_{\alpha+1} = \wp(V_\alpha)$$

$$V_\lambda = \bigcup_{\alpha<\lambda} V_\alpha$$

The complete 'universe' is usually written V; we can write:

$$V = \bigcup_{\alpha} V_\alpha$$

but we should be aware that V is not a set, and the above is really a figure of speech for the formal assertion:

$$\forall x.\ \exists \alpha.\ Ordinal(\alpha) \wedge x \in V_\alpha$$

All sets contained in V_ω, and hence in V_n for some finite ordinal n, are hereditarily finite (i.e. they are finite and all their members are in turn hereditarily finite). V_ω itself is the first infinite set. All Zermelo's axioms are satisfied if the 'universe' is the set $V_{\omega+\omega}$, i.e. all sets arise from applying the powerset operation finitely often to the empty set or the first infinite set.

Using the Axiom of Replacement (available in ZF but not in Zermelo's original system), we can show that there is actually a set $V_{\omega+\omega}$, as follows: by recursion and the Axiom of Replacement, we can construct a function whose range is $V_{\omega+\omega}$, and the Axiom of Union then allows us to collect the range in a new set. The iteration may similarly be continued to any transfinite ordinal. Consequently the hierarchy extends much further than it need do without Replacement.

A set-theoretic reflection principle asserts, roughly speaking, that some restricted initial portion of the ZF hierarchy (which will of course be a set occurring in the next level up) 'reflects' the structure of the whole universe. Crudely

[4] This now ubiquitous picture was only arrived at by Zermelo 20 years after his original axiomatization.

speaking, *any property true of V is also true of some V_α*. This is a vague statement and cannot be taken too literally. For example, 'every set is a member of V' is clearly false if V is replaced by any V_α. However by suitably formalizing and restricting the notion of 'property' in the above assertion, meaningful statements are obtained.

In particular, if we restrict ourselves to 'properties' expressible in an orthodox first order axiomatization of ZF set theory, the natural formalization of the reflection principle turns out to be provable. This was first shown by [Mon66] and [Lév60]; the latter coined the term 'reflection principle' and initiated a thorough study of such principles. Let us write ϕ^R for the 'relativization' of some formula ϕ to the set (or, with obvious change, class) R. This means restricting all the quantifiers in ϕ as follows: $\forall x.\ \psi[x]$ becomes $\forall x.\ x \in R \Rightarrow \psi[x]$ and $\exists x.\ \psi[x]$ becomes $\exists x.\ x \in R \wedge \psi[x]$. Now the reflection schema states that for any formula ϕ with free variables $x_1 \ldots x_n$:

$$\forall \alpha.\ \exists \beta.\ \beta > \alpha \wedge \forall x_1 \ldots x_n \in V_\beta.\ \phi \equiv \phi^{V_\beta}$$

For a proof, see for example [Kun80] or [Kri71]. In the special case where ϕ is a sentence then we see that there exist arbitrarily large ordinals β such that ϕ is interdeducible with its relativization to V_β. Among the most simple consequences we see that ZF cannot be finitely axiomatizable in first order logic. Indeed if a finite set of axioms S sufficed to axiomatize ZF, then we could form their conjunction ϕ. But now the reflection schema yields a model (more precisely, proves in ZF the existence of a model) for ϕ, and hence S, inside ZF. This amounts to proving ZF's consistency inside ZF, which we know to be impossible by Gödel's second theorem. (A more direct proof not relying on Gödel's theorem is also possible from the reflection principle.)

Sharpening these observations demonstrates the interesting connection with logical reflection principles[5]. If $Prov_N(p, \ulcorner \phi \urcorner)$ means 'p codes a proof of ϕ from the first N axioms of ZF', then it is not hard to see that the following (which one might regard as another kind of partial reflection schema) is provable:

$$Prov_N(p, \ulcorner \phi \urcorner) \Rightarrow \phi$$

We know that $\vdash_{ZF} \phi \vee \neg\phi$. We can now reason inside ZF as follows. If ϕ, then the consequent in the implication is trivially true. Otherwise the formula which is the conjunction of the first N axioms of ZF together with $\neg\phi$ is true, so by the reflection theorem, is true in some initial portion of the hierarchy. This means it has a model V_α, which provides a countermodel to the antecedent of the implication.

[5] This connection goes right back to Montague's work; nevertheless the term 'reflection' seems to be based on different idioms in the two cases.

Apart from such applications, the deductive strength of this and related principles has been studied by Lévy and others. In the presence of the other ZF axioms, the first order reflection schema given above turns out to be equivalent to the axioms of Infinity and Replacement taken together. This raises the possibility that one might axiomatize set theory using reflection principles.

Extended reflection principles, for example the higher order versions introduced by [Ber76], turn out to be equivalent to rather recherché additions to the ZF axioms, such as inaccessible cardinals, regular fixed points for normal functions, and Mahlo cardinals. Hence they provide, for some, a persuasive way of motivating such additions. One can even eschew any kind of formalization and simply regard reflection principles as quasi-philosophical assertions about the bottomless, arbitrarily extensible nature of the ZF hierarchy.

The proposal of [Kre67] that the notions of semantic validity for second order logic when interpreted in a formal, cumulative set theory like ZF and in 'informal set theory' (permitting for example proper classes), might be coextensive, has similar consequences. Some formalizations of Kreisel's principle turn out to be equivalent to higher order reflection principles. See for example [Sha91].

13.3 Procedural reflection

Computer programs are ultimately run as machine code which exists ephemerally in the memory of the machine. However for many purposes it is useful to be able to step back from this simple picture, contemplating (and perhaps changing) the relationship between the running program and the original source. Some obvious examples, starting with the routine and ending in the exotic, are:

- Debugging – here it is desirable to relate the execution of the program to its original (source code) syntax, for the benefit of the user, and allow the user to step through the program, inspecting and altering the state of the machine at various points.
- Profiling – here we want to associate runtimes with function bodies in the original source (which may have no simple relationship with portions of the eventual machine code), in order to identify 'hot spots' in the code.
- Self-modification – it is commonplace in Artificial Intelligence to have programs modify themselves in the light of the interaction with their environment. For example, a chess program may alter its play on the basis of past experience.

Implementing such facilities in an ad-hoc way is sometimes quite involved, and it is not easy for ordinary users to add their own related facilities. It was argued by [Smi84] that a general reflective programming language offers a uniform and flexible way of doing such things. He compares reflection with recursion. At first it seems a complicated, arcane and inefficient way of programming, in danger of infinite regress. But with experience it may come to be seen as natural, and hence tend to be implemented more efficiently. It promises to provide a highly flexible facility which may then be used to implement otherwise inexpressible programming constructs (such as adding exception-handling to the language). A more developed treatment can be found in [dRS84].

Smith's approach is to start with a dialect of LISP, called 2-LISP, which is LISP shorn of the use-mention confusions which Smith detects in the mainstream version. Smith argues that LISP 'crosses semantic levels', confusing the notionally separate processes of passing from syntax to denotation and evaluating the denotation. For example, `(+ 1 '2)` is acceptable in all LISP and SCHEME implementations.

Programs are run via an interpreter whose code and data is made concrete in explicit datastructures. This results in 3-LISP, which has the ability to run code at different levels of interpretation. Instead of running a program 'at level n' the concrete representation can be run by an interpreter 'at level $n+1$'. This interpreter may itself be run by another interpreter 'at level $n+2$' and so on ad infinitum. The crucial points are that firstly, all properties of the program and its interpreter are made concrete, and secondly, that those concrete versions may be modified, affecting the program's behaviour.

In LISP implementations, this already happens to a limited extent when procedure calls and `EVAL` are supported uniformly for compiled code and interpreted S-expressions. Even running a BASIC interpreter on a microcoded CPU exhibits a multiplicity of levels. The reflective approach is distinguished by its unlimited scope and the homogeneity of the successive levels. It actually gives rise to a potential infinity of levels, the so-called 'reflective tower'. Smith's idea was that a program should find the lowest level available (since levels of interpretation are inefficient), only rising to higher levels when necessary. As far as practical implementations go, experience has been limited, but it may be that a less comprehensive form of reflection than that proposed by Smith, where the concrete representations are separate from the code executed and perhaps only partial (this is sometimes called 'declarative reflection'), is acceptably efficient.

Smith did not give a rigorous discussion of the intended semantics of the reflective tower. This was later undertaken by [WF86], who gave a denotational description, using an additional 'metacontinuation' parameter to store the state of the interpreters above the one currently being considered. They also use a slightly refined terminology: 'reification' is the process by which an interpreter makes its state available, and 'reflection' is when the program changes that state and hence installs new data.

Since it was originally proposed, procedural reflection has attracted considerable attention in the object oriented programming community, initiated explicitly by [Mae87]. The approach fits nicely with the object philosophy, where the programmer's whole 'world' is supposed to be open to redefinition, yet some facilities, e.g. communication between processes in sophisticated ways, may be hard to implement in existing systems. A comprehensive survey is given by [MN88][6]. Nevertheless it is not clear that reflection's practical utility has been convincingly demonstrated. For work on procedural reflection in a high-level functional programming language, see the paper by [Zhu94].

[6] The author has not read this book, so this assertion is speculative, based on second-hand information.

Chapter 14

Computational reflection: theory and practice

Focusing now on computational reflection, we should look in more detail at precisely what it demands, and at existing practice and proposals. The idea of computational reflection is not to make a formal system stronger, but rather to make its deductive process more efficient by utilizing information which avoids having to construct formal proofs in full detail. The first question to ask is whether we should expect to achieve worthwhile gains in efficiency. [DS79], among the earliest advocates, do not marshal any convincing arguments:

> On the basis of ordinary mathematical experience we have every reason to expect that the difficulty (in the precise sense we have defined) of various important theorems will be greatly decreased this way. Although we have been unable to formulate and prove any metatheorems that would serve as a formal demonstration of this conjecture, we can point to some suggestive evidence. It is well known in proof theoretic research that the addition of new rules of inference to so-called cut-free systems can drastically decrease the length of proofs. ...we have seen that the introduction of an appropriate "algebra" rule of inference shortens to 1 the difficulty of a sentence which asserts an algebraic identity.

The appeal to 'ordinary mathematical experience' is vague. Experience with Mizar and HOL in pure mathematics has shown no hint that expansion into primitive inferences is not feasible. Admittedly, only a tiny part of mathematics has been formalized, but there seems no obvious reason to expect any other branches to be different in this regard. Since cut-free sequent proofs are a theoretical device and nobody would dream of basing a general computer theorem prover on them, the second point is irrelevant – it's hardly surprising that by artificially hobbling the logic one can make the sizes of proofs explode. Finally, judging efficiency on the basis of the number of inferences, without regard to the computational complexity of those inferences, is not acceptable if the argument is supposed to be one of practical utility.

If theoretical speculation is unconvincing, what of examples? We have already mentioned the Deduction Theorem, which indeed demonstrates convincingly how metareasoning can make proofs shorter, obviating the need for a number of intermediate steps. However again this example is artificial, as conceded by [MSB91], since a proof system is not likely to be based on a Hilbert-style axiomatization.

For another example, consider the following system of classical biconditional logic. The only logical connective is bi-implication ($\equiv$). The axioms are all substitution instances of:

$$p \equiv p$$

$$(p \equiv q) \equiv (q \equiv p)$$

$$(p \equiv (q \equiv r)) \equiv ((p \equiv q) \equiv r)$$

and the sole rule of inference is the following variant of Modus Ponens:

$$\frac{p \equiv q \quad p}{q}$$

It is not hard to see that a formula is provable if and only if every propositional variable in it occurs an even number of times. This fact appears to have first been explicitly written down by [Łuk70]. Since provability here is also equivalent to being truth-table valid, the same observation obtains for the equivalential fragment of any conventional axiomatization of propositional logic. Anyway, using this metatheorem we can justify saying a formula is provable simply by pairing off the propositional variables. A less striking but more realistic example is algebraic simplification using associative and commutative laws and/or cancellation. Given an assertion of the form:

$$a_1 + \ldots + a_n = b_1 + \ldots + b_n$$

we can justify its truth just by showing that the sets $\{a_1, \ldots, a_n\}$ and $\{b_1, \ldots, b_n\}$ are equal. To produce a proof by primitive inferences, though, we need to delicately rewrite with the associative and commutative laws to make the two sides identical.

But let us look more critically at this example. How are we to compare the sets $\{a_1, \ldots, a_n\}$ and $\{b_1, \ldots, b_n\}$? Of course it depends on how terms are represented inside the theorem prover. But in any case the problem of testing n-element sets for equality is known – see [Knu73] for example – to be $O(n^2)$ in the worst case (assuming a constant-time pairwise equality test is available). Even if the subterms are pairwise orderable somehow, we can't do better than $O(n\ log(n))$. It's not hard to devise algorithms using primitive inferences which have the same complexity. Of course there may be a significant constant factor difference, but nevertheless, the example is hardly especially persuasive from the perspective of efficiency. Perhaps there is an argument that the LCF coding is much less *natural*. To an extent this is true, but then any encoding of an algorithm in a strict formalism like a programming language is 'unnatural' – it's just a question of degree.

14.1 Computational reflection and code verification

Computational reflection requires us to verify, inside the existing theorem prover, the correctness of some algorithm implementing a theorem proving procedure, following which we may add it to the prover's repertoire. The algorithm may be expressed inside the logic; the higher order type theory of systems like HOL and Nuprl corresponds nicely to an idealized functional programming language – one where all functions are total. Similarly NQTHM's logic corresponds closely to pure LISP. Verification of nontrivial algorithms may already be within reach in this way. For example [AL94] have verified a Boolean simplifier, consisting of approximately 1000 lines of Standard ML code, in Nuprl.

After being verified, the algorithm may then be executed in the logic by inference. However this is itself likely to lead to a fairly large constant factor slowdown, probably worse than the expansion to primitive inferences for the rule itself. Therefore it is much more appealing to write out the algorithm in the implementation language of the theorem prover (LISP, ML or whatever). This raises three serious problems.

1. How exactly do we regiment the process of iteratively adding code to a running implementation? We are trying to repair the hull of a ship while it is sailing, without bringing it into dry dock.

2. How do we justify the correspondence between an idealized mathematical description inside the logic and an implementation in a real programming language? We know that the latter has awkward features like finite limits for arithmetic, complicated evaluation orders and subtle semantics for nontermination and exceptional conditions.

3. How do we represent inside the logic imperative language features like arrays and pointers? As we have argued above, it is important to reason about these features if we want to implement certain important proof procedures effectively.

A satisfactory answer to the first question depends on the implementation language. In languages such as LISP the seamless use of compiled and interpreted code makes it much easier. In an LCF-style system, there are some quite formidable problems; we must somehow rip open an abstract type, tinker with it to add a new constructor, and then close it up again. An alternative is to perform the final addition of code 'informally' after performing the proof, then restart the enhanced system. The user must take responsibility for correctly sequencing the additions and editing the source code.

A completely satisfactory answer to the second and third points is simple: don't just verify an abstract version of an algorithm, verify actual code using the formal semantics of the implementation language[1]. This means embedding the syntax and formal semantics of the implementation language in the logic. Here we make several presuppositions, most notably that the implementation language has a suitable formal semantics which is stable and likely to be adequate and tractable for serious proofs. These are properties satisfied by very few languages. The difficulty of code verification of this kind is serious, and as far as we are aware no substantial examples exist.

All existing instances of reflection make the leap from an abstract to a concrete implementation via a naive syntactical transformation without complete formal justification. The correctness proof in abstracto is much easier, and perhaps for most practical purposes the distinction between the abstract description and the concrete implementation is not likely to trip one up. Nevertheless the third point is still problematical; it is our belief that for many efficient proof procedures, imperative code is required, and it is not so easy to associate such constructs with parts of the logic in a convincing way. Arrays can be identified with functions, as in some versions of Floyd-Hoare logic, but care needs to be taken over indexing exceptions, and the problem of aliasing makes any proofs much harder.

Another difficult question is: what do we mean by 'correct' in this context?

[1] Of course one can still doubt that the compiler correctly implements the language, or that various issues abstracted away, like running out of memory, are really irrelevant; but these are not new problems arising with reflection.

The minimal requirement is that if a procedure terminates, it always produces something valid. This is a partial correctness condition – we do not prove termination – and may well be the best we can do in many cases. Resolution methods performing unbounded search, or algorithms for higher order unification (used with great success in practice in the TPS, Isabelle and LAMBDA systems) may fail to terminate. In the case of complicated heuristic procedures, it may be quite impossible to demarcate formally those instances where termination is to be expected.

Even if termination can be proved, the proof may require much more complex mathematics than that required to prove partial correctness. We are not aware of any particularly convincing practical instances, but justifying bounds for the solution of Diophantine equations requires deep analytic number theory – see the work of [Bak75] in this regard. It may even happen that termination is only provable in a stronger logic – following the work of [Kre52] it has become a fashionable research topic to classify logics on the basis of which functions they can prove total.

The main reason the question of termination should interest us is that most of the abstract versions of proof procedures are modelled in the logic as total functions. Reasoning about partial functions is widely believed to be much more difficult. Nevertheless, a serious attempt to address partiality in reflected proof procedures is surveyed by [GACT94].

A more extreme point is that many of the most efficient algorithms *aren't* correct in a strict sense. They may assume that machine arithmetic will never overflow for example. Now it may be an entirely justifiable assumption on the basis that machine resources would become exhausted before it could happen. But to carry through details like this in a formal proof is much more complicated. To take an analogy with physics: if I want to analyze the dynamics of a person riding a bicycle, I intuitively *know* that I can neglect relativistic effects, and use classical mechanics. But if I want to prove this formally, I have to apply relativistic mechanics (or at least derive a more general theorem to that effect first).

14.2 Computational reflection in practice

Reflection is a popular topic for investigation in theorem proving, and one might expect large numbers of real systems to have experimented with it. On the whole, experiments have been limited to small projects. We shall examine three significant implementations, and one relevant proposal.

14.2.1 FOL

Perhaps the first actual use of reflection in a formal reasoning system was by [Wey80]. His FOL system is an implementation of first order logic which allows reasoning in multiple theories. It provides a notion of 'simulation structure', which may be described as a computable, partial model. A full model is unattainable since even facts about ground terms may be uncomputable. However one can associate with a logical system L some simulation structure S giving a restricted part of the information that a model would provide, in the shape of some kind of evaluator. For example, one might associate with the addition symbol an evaluator which rewrites expressions involving addition and other ground terms. This process of association is called 'semantic attachment', because it stands in place of a full model (though as it is entirely computable, the use of 'semantic' has little technical meaning). The resulting couple is called an 'L/S pair' or 'context' (in some related work a context includes the currently proved set of theorems). The standard set-theoretic account of first order logic semantics has been tweaked to use simulation structures rather than full models by [WT94].

One of the contexts in FOL is called META, and formalizes the syntax of FOL's own logic, including the structure of formulas and logical derivability. Then a reflection principle is asserted which justifies a transition between $\vdash_{object} \phi$ and $\vdash_{meta} Prov(p, \ulcorner\phi\urcorner)$. The principle allows these statements to be interderived both ways; these processes are usually referred to as 'reflection up' (left to right) and 'reflection down' (right to left).

At its simplest, this connects inference in the object theory with computation in the metatheory. The syntax operations and inference rules of the object logic are just function symbols in the metalogic, and may, by semantic attachment, be associated with the natural operations on formulas. This identification of 'theorem proving in the theory' with 'evaluation in the metatheory' (to coin a FOL slogan) is reminiscent of LCF. However since a full logic is available to formalize the metatheory, it is more general. Metatheorems can be proven which justify certain kinds of inference without needing to expand down to the original primitives. A simple illustrative example is given in [Wey82] based on a Hilbert-style axiomatization of propositional logic.

The obvious defect of the FOL approach is that there is no check on the user attaching arbitrary actions to function symbols. Logical consistency is not enforced. For example, one might make the evaluator transform $1+1$ into 1. This doesn't affect FOL's appeal as an AI project, but for the approach is to make inroads into the formal verification community something better is needed.

This is being worked on by a number of researchers in Italy. FOL has been reimplemented and reengineered as GETFOL, and attempts are being made to achieve formal demarcation of acceptable proof strategies rather than permitting arbitrary attachments. An interesting summary is given by [ACV93]. Proof strategies in the object theory are simply terms in the metatheory. It is possible to demarcate, purely syntactically, terms which implement safe proof strategies; so-called 'logic tactics'. These are built from existing primitives using a few simple connectives like the conditional, much as in LCF. However it is possible to use more general metatheoretic reasoning should it prove necessary. The next step being considered is to compile proven proof procedures down to the implementation language (apparently using a naive transliteration) to make them more efficient than interpreting them in the metatheory. This so-called 'flattening' process has already been investigated in distinct but related work by [BGT91].

14.2.2 NQTHM

The NQTHM prover, described in detail by [BM79], is a fully automatic theorem prover for a quantifier free first order logic. The logic has its roots in Primitive Recursive Arithmetic (PRA), as developed by Skolem and [Goo57], but allows arbitrary recursive types, not just the natural numbers. Natural numbers are used to encode boolean values, and the propositional functions like conjunction are just operations on these natural numbers. There is no separate class of formulas. An induction rule is available, but there are no explicit quantifiers. The logic is represented using LISP syntax. The prover has no real interactive features, but the user may direct the prover by choosing a suitable chain of lemmas, each of which can be proved automatically.

One of the earliest practical applications of reflection in a major theorem prover, indeed `the` earliest where the stress was placed on soundness, was the work of [BM81] in adding a reflection principle to NQTHM. (Note that they do not use the word 'reflection' to describe the process, but rather talk about adding 'metafunctions'.) This was then used to implement a simplifier which performs cancellation in arithmetic equations. Other simplifiers, including a tautology checker, have also been verified. Apparently metatheoretic extension is a facility not widely used in practice, but it is not just a theoretical flight of fancy.

The implementation of reflection in the Boyer-Moore prover does not entail the full internalization of the logic's rule of inference. Rather, a denotation function `MEANING` is defined (just an ordinary recursive function, as has been mentioned already). This gives the value of an encoded term (the coding is rather simple, since the logic is in LISP syntax to start with) under a given assignment to variables.

If the user wishes to introduce a term-transforming function `fn` as a new logical primitive, then first, as usual with definitions in NQTHM, the prover must show that the recursion equations given define a unique total function. (So it seems that it would be quite a bit harder to extend the approach to the verification of rules which may fail to terminate.) Then the prover must prove a metatheorem, which states that for any formula `F` and any assignment `A`, then firstly `fn(F)` is also a formula (the system is untyped, so this is not automatic), and second that `MEANING(A,F) = MEANING(A,fn(F))`. That is, the simplified term is always equal to the original term under any variable assignment. Note that the transformation function may be parametrized by free variables, provided it obeys the above strictures under all assignments.

If these obligations are proved, the system installs compiled INTERLISP code corresponding to `fn`. This jump deserves detailed consideration. INTERLISP does not behave quite like the encoded abstract pure LISP environment. In particular there is a finite limit for integers, with silent wrapping on overflow. Boyer and Moore pay careful attention to these difficulties, using for example a custom addition function `badd1` which fails on overflow. One or two other tweaks are applied, mostly for reasons of efficiency rather than correctness. The approach is careful, but rather informal and ad-hoc.

Another point to note is that what in some systems would be automatic type correctness conditions (for example, that `fn` maps terms to terms), must, since we are in the untyped world of LISP, be proved explicitly. Indeed, if the system has already been extended unsoundly (the user may posit inconsistent axioms, precisely in order to derive an inconsistency!) then the proof may be nonsense and the newly installed metafunction might not just derive falsity, but fail in arbitrary and damaging ways.

It has to be said that the cancellation function is trivial, and easily implemented in the LCF style with adequate efficiency. But Boyer and Moore report that the correctness proof was actually rather easy (taking one of the authors just a day), so we should not take this example as truly indicative of the potential state of the art. Nevertheless, the issue of verifying imperative code is not addressed.

14.2.3 Nuprl

Nuprl, described by [Con86], is an LCF-descended theorem prover which supports an extension, including for example inductive types, of a type theory as described by [ML85]. It is intended as an environment for constructive mathematics and computer programming, and for exploring their connections. The richness of the Nuprl type theory generates a profusion of additional primitive inferences compared with HOL[2]. Furthermore the system stores proof trees as concrete objects, so invocation of primitive inferences is particularly expensive in space and hence garbage collection time. Finally, in applied work, decision

[2] It is not clear whether a superior implementation would alleviate these problems.

procedures have often been tacked on to Nuprl, and the user community is more accustomed to them; it is however desirable to place them on a firmer theoretical footing. All these facts make computational reflection especially appealing.

We have already discussed the work of Howe, where the connection between an internalization of derivability and the logic itself is made via a function in the logic. However Nuprl researchers have also experimented with adding explicit reflection rules allowing the deduction of $H \vdash G$ from $\vdash Prov(p, \ulcorner H \vdash G \urcorner)$. There is a danger of infinite regress here, since we are using the *same* object theory and metatheory. We have seen that adding the reflection schema makes the theory stronger, so for the sake of consistency, we need to modify the $Prov$ relation, and so on ad infinitum. In summary, without restriction the above rule is unsound. Two possibilities have been pursued.

The first is explained in [KC86]. The idea is that repeatedly adding the reflection rules gives rise to a sequence of logics PRL^0, PRL^1, PRL^2, ... The reflection rule connects formalized provability at one level to the logic in the level below it. This is rather complex because there are an infinite number of levels. Nevertheless for many purposes 'provability in some PRL^k' can be taken as the standard notion of provability. As far as we are aware, no practical work has been done using this scheme.

The second alternative, described in [ACHA90] is to have just one logic and restrict the reflection rule. The reflection rule is parametrized by a natural number called the 'reflection level', and in any instance of the reflection rule, embedded instances must have a lower reflection level. With this change the reflection rule becomes a conservative extension to the logic, although its eliminability from proofs cannot be proven *inside* the logic.

A further step in this direction is described by [How92]. He proposes a slight modification of the Nuprl type theory, including for example a 'denotation' (better: 'evaluation') function, which allows a particularly clean internalization of the logic's semantics. (The presence of dependent types blurs the distinction between syntax and semantics, so one shouldn't read the terminology used here too critically.) The semantics is stratified by universe level; at level i the evaluation semantics of lower universe levels can be completely formalized. Using this internal semantics, Howe was able to derive the reflection rule without extending the logic.

Most of the work in reflection proper appears to be theoretical, and we are not aware of any practical applications. The position of the Nuprl community on some of the issues raised above regarding real programming languages is not entirely clear.

14.2.4 HOL

No work on reflection has actually been done in HOL, but [Sli92] has made some interesting proposals. His approach is distinguished from those considered previously in two important respects.

Firstly, he focuses on proving properties of programs written in Standard ML using the formal semantics to be found in [MTH90]. This contrasts with the other approaches we have examined, where the final jump from an abstract function inside the logic to a concrete implementation in a serious programming language which *appears to correspond to it* is a glaring leap of faith.

Secondly, he points out that absolute program verification is not necessary. It suffices to show that some new piece of code behaves in the same way as another piece of code which is implemented in the normal LCF way as a composition of primitive inference rules. More precisely, if an ML function `f` is a 'safe' derived rule returning a theorem, and we can prove that for some function `g` which returns a term list-term pair, `dest_thm o f = g` then we may safely incorporate `mk_thm o g` as a new rule.

How realistic is this proposal? The more modest requirement for proofs of program equivalence has considerable promise. It seems more suited to simple low-level proof procedures which have a straightforward but inefficient proof in terms of primitive inferences. The use of program equivalence is less appealing in instances where a reasonable proof by primitive inferences must deviate substantially from the 'standard' presentation. Nevertheless, as far as it goes, it may allow reasonably tractable correctness proofs without forcing the practitioner to worry about the low-level codification of the logic.

It is conceivable that a useful set of equational transformations for reasoning about Standard ML programs could be derived from the operational semantics of Standard ML. On the other hand, the semantics of Standard ML is not trivial, and has certain defects from the point of view of correctness proofs. For example, it says nothing about the behaviour of arithmetic operations; in practice some implementations use machine arithmetic while others use infinite precision arithmetic. If one takes program verification seriously, these issues have to be addressed.

There has been some work in HOL on the formal semantics of ML by [Sym93] and [VG93]. But this is in its early stages, and more than just a formalized semantics is needed to make program verification tractable. One approach might be to isolate a sufficient core of ML for the implementation of HOL, and attempt to produce a formal semantics for it. The difficulty is that, if our arguments are correct, the appeal of reflection is greatest where the verification is of imperative code. This is known to be difficult to reason about formally; see [MT92] for some preliminary work in this direction.

Finally, it is interesting to observe how Slind dealt with the problem of inserting compiled code into an LCF-style implementation. His technique makes essential use of first-class environments, a relatively new feature of the New Jersey compiler. It may be that it holds interesting lessons for the incremental extension of abstract data types in general, not just in connection with LCF-style theorem proving.

Chapter 15

Conclusion

Extensibility of theorem provers is an important issue, and if demands of rigour are to be taken seriously, the relative merits of reflection and the pure LCF approach should be analyzed carefully.

For many purposes, ostensibly metatheoretic reasoning can be implemented without any logical extensions. In any case, there are various subtly different notions described as reflection, and it is important to distinguish them. The most interesting from the point of theorem proving technology is computational reflection used to make inferences more efficient without unprincipled addition of new rules.

Programming a derived rule in LCF requires a certain discipline, and in complex cases, good programming skills are needed. There is certainly an argument that verifying a more direct implementation is more natural. It's rather like coding an algorithm which appears naturally imperative in a pure functional language. Nevertheless there seems no convincing evidence that it is fundamentally inadequate from the efficiency point of view. Airy claims about the hopeless inefficiency of LCF-style provers on real examples have limited support in theory and are contradicted by practice (HOL is used!) We have looked at techniques which often render LCF proof procedures quite efficient. It has not clearly been established that there are any efficient proof procedures which cannot be implemented as a HOL derived rule with more than a moderately large constant factor slowdown. This constant factor may well be practically important, but it seldom separates tractability from intractability.

Reflection is an intellectually attractive idea, in that it offers a way of adding efficient proof procedures while maintaining a guarantee of security. However the fact that despite all the research reports and proposals it has only once been used in a major practical prover, and even there not much in practice, speaks against it. Furthermore we have argued that the most interesting proof procedures such as Binary Decision Diagrams depend for their efficiency on imperative features such as arrays or shared data structures. Most real implementations are written in C, a difficult language to reason about formally. If reflection principles are to accommodate such programs, then the state of the art in program verification needs to advance, or the correctness proofs will be unbearably difficult. Then the difficulties of dynamically adding code to a running implementation need to be taken seriously.

There is considerable intellectual and practical benefit in sticking to the pure LCF approach in HOL, and the case against it is questionable. Reflection is appealing, but there are several significant technical obstacles. It may also interfere with projects related to proof checking. Whether HOL exhibits an unusual synergy in this regard is an interesting question, and our conclusions, even if correct for HOL, do not necessarily extend to other systems. Reflection offers many interesting ideas and challenges, but it isn't yet ready to push back the boundaries of what is feasible in theorem proving. Attempts to present it as a practical necessity and panacea for theorem proving in real world applications are simply naive.

Bibliography

[ACHA90] S. Allen, R. Constable, D. Howe, and W. Aitken. The semantics of reflected proof. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pages 95–107, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[ACS86] Luigi Aiello, Carlo Cecchi, and Dario Sartini. Representation and use of metaknowledge. *Proceedings of the IEEE*, 74:1304–1321, 1986.

[ACV93] Alessandro Armando, Alessandro Cimatti, and Luca Viganò. Building and executing proof strategies in a formal metatheory. In Pietro Torasso, editor, *Advances in Artifical Intelligence: Proceedings of the Third Congress of the Italian Association for Artificial Intelligence, IA*AI'93*, volume 728 of *Lecture Notes in Computer Science*, pages 11–22, Torino, Italy, 1993. Springer-Verlag.

[AJLW91] Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors. *Proceedings of the 1991 International Workshop on the HOL theorem proving system and its Applications*, University of California at Davis, Davis CA, USA, 1991. IEEE Computer Society Press.

[AL94] Mark Aagaard and Miriam Leeser. Verifying a logic synthesis tool in nuprl: A case study in software verification. In G. v. Bochmann and D. K. Probst, editors, *Computer Aided Verification: Proceedings of the Fourth International Workshop, CAV'92*, number 663 in Lecture Notes in Computer Science, pages 69–81, Montreal, Canada, 1994. Springer Verlag.

[Art94] R. D. Arthan. Issues in implementing a high integrity proof tool. In Basin et al. [BGK94], pages 34–36.

[AW80] Luigia Aiello and Richard W. Weyhrauch. Using meta-theoretic reasoning to do algebra. In Wolfgang Bibel and Robert Kowalski, editors, *5th Conference on Automated Deduction*, volume 87 of *Lecture Notes in Computer Science*, pages 1–13, Les Arcs, France, 1980. Springer-Verlag.

[Bac88] R.J.R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[Bak75] Alan Baker. *Transcendental Number Theory.* Cambridge University Press, 1975.

[Bas94] David A. Basin. Constructive metatheoretic extensibility. In Basin et al. [BGK94], pages 23–24.

[BC91] David A. Basin and Robert Constable. Metalogical frameworks. In Huet et al. [HPJ91], pages 47–72. Available by FTP from 129.215.160.150 as export/bra/proc91.dvi.Z.

[BEJ88] Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors. *Partial Evaluation and Mixed Computation: Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*, Gammel Avernæs, Denmark, 1988. North-Holland.

[Ber76] P. Bernays. On the problem of schemata of infinity in axiomatic set theory. In Gert H. Müller, editor, *Sets and Classes: on the work by Paul Bernays*, number 84 in Studies in Logic and the Foundations of Mathematics, pages 121–172. North-Holland, 1976.

[BGK94] David Basin, Fausto Giunchiglia, and Matt Kaufmann, editors. *12th International Conference on Automated Deduction, Workshop 1A: Correctness and metatheoretic extensibility of automated reasoning systems*, INRIA Lorraine, 1994.

[BGT91] David Basin, Fausto Giunchiglia, and Paolo Traverso. Automating meta-theory creation and system extension. In *Trends in Artificial Intelligence: Proceedings of the 2nd Congress of the Italian Association for Artificial Intelligence, IA*AI*, number 549 in Lecture Notes in Computer Science, pages 48–57. Springer-Verlag, 1991.

[BK82] Kenneth A. Bowen and Robert A. Kowalski. Amalgamating language and metalanguage in logic programming. In K. L. Clark and S. A. Tärnlund, editors, *Logic Programming*, number 16 in APIC Studies in Data Processing, pages 153–172. Academic Press, 1982.

[BM79] Robert S. Boyer and J Strother Moore. *A Computational Logic.* ACM Monograph Series. Academic Press, 1979.

[BM81] Robert S. Boyer and J Strother Moore. Metafunctions: proving them correct and using them efficiently as new proof procedures.

In Robert S. Boyer and J Strother Moore, editors, *The Correctness Problem in Computer Science*, pages 103–184. Academic Press, 1981.

[Bou93] Richard John Boulton. *Efficiency in a fully-expansive theorem prover*. University of cambridge computer laboratory technical report, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, 1993.

[Bry92] Randall E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24:293–318, 1992.

[BvHHS91] A. Bundy, F. van Harmeleu, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–323, 1991.

[Car37] Rudolf Carnap. *The Logical Syntax of Language*. International library of psychology, philosophy and scientific method. Routledge & Kegan Paul, 1937. Translated from 'Logische Syntax der Sprache' by Amethe Smeaton (Countess von Zeppelin), with some new sections not in the German original.

[CG92] Luc J. M. Claesen and Michael J. C. Gordon, editors. *Higher order logic theorem proving and its applications: proceedings of the IFIP TC10/WG10.2 International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume A-20 of *IFIP Transactions A: Computer Science and Technology*, IMEC, Leuven, Belgium, 1992. North-Holland.

[CM90] Robert L. Constable and Chet Murthy. Extracting computational content from classical proofs. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 141–156, 1990. Unpublished.

[Con86] Robert Constable. *Implementing Mathematics with The Nuprl Proof Development System*. Prentice-Hall, 1986.

[Dan88] Olivier Danvy. Across the bridge between reflection and partial evaluation. In Bjørner et al. [BEJ88], pages 83–116.

[dB80] N. G. de Bruijn. A survey of the project automath. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.

[dRS84] Jim des Rivières and Brian Cantwell Smith. The implementation of procedurally reflective languages. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 331–347. ACM, 1984.

[DS79] Martin Davis and Jacob T. Schwartz. Metatheoretic extensibility for theorem verifiers and proof-checkers. *Computers and Mathematics with Applications*, 5:217–230, 1979.

[Dud88] Richard M. Dudley. *Real analysis and probability.* The Wadsworth & Brooks/Cole mathematics series. Wadsworth & Brooks/Cole, 1988.

[Edw89] Harold M. Edwards. Kronecker's views on the foundations of mathematics. In David E. Rowe and John McCleary, editors, *The History of Modern Mathematics; Volume 1: Ideas and Their Reception*, pages 67–77. Academic Press, 1989.

[Fef62] S. Feferman. Transfinite recursive progressions of axiomatic theories. *Journal of Symbolic Logic*, 27:259–316, 1962.

[FM88] Amy Felty and D. Miller. Specifying theorem provers in a higher-order logic programming language. In Lusk and Overbeek [LO88], pages 61–80.

[GACT94] Fausto Giunchiglia, Alessandro Armando, Alessandro Cimatti, and Paolo Traverso. First steps towards provably correct system synthesis of system code. In Basin et al. [BGK94], pages 28–30.

[GHH+94] Michael J. C. Gordon, Roger Hale, John Herbert, Joakim von Wright, and Wai Wong. Proof checking for the hol system. In Basin et al. [BGK94], pages 49–50.

[GM93] Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[GMW79] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[Göd67] Kurt Gödel. On formally undecidable propositions of principia mathematica and related systems, i. In Jean van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic 1879–1931*, pages 592–618. Harvard University Press, 1967.

[Goo57] R. L. Goodstein. *Recursive Number Theory.* Studies in Logic and the Foundations of Mathematics. North-Holland, 1957.

[Gor82] Michael J. C. Gordon. Representing a logic in the lcf metalanguage. In D. Neel, editor, *Tools and notions for program construction: an advanced course*, pages 163–185. Cambridge University Press, 1982.

[Gra93] B.T. Graham. rec_tys_listop — a contribution to the HOL system. Distributed with HOL system, 1993.

[GS89] Fausto Giunchiglia and Alan Smaill. Reflection in constructive and non-constructive automated reasoning. In Harvey Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 123–140. MIT Press, 1989.

[Har93] John Harrison. A hol decision procedure for elementary real algebra. In Joyce and Seger [JS93b], pages 426–436.

[Har94] John Harrison. Binary decision diagrams as a hol derived rule. In Melham and Camilleri [MC94], pages 254–268.

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Proceedings of the Second Annual Symposium on Logic in Computer Science*, pages 194–204, Ithaca, NY, 1987. IEEE Computer Society Press.

[HM92] D. N. Hoover and D. McCullough. Verifying launch interceptor routines with the asymptotic method. ORA internal report, 1992.

[How88] Douglas J. Howe. Computational metatheory in nuprl. In Lusk and Overbeek [LO88], pages 238–257.

[How92] Douglas J. Howe. Reflecting the semantics of reflected proof. In Peter Aczel, Harold Simmons, and Stanley Wainer, editors, *Proof Theory*, pages 229–250. Cambridge University Press, 1992.

[HPJ91] Gérard Huet, Gordon Plotkin, and Claire Jones, editors. *Proceedings of the First Workshop on Logical Frameworks*, 1991. Available by FTP from 129.215.160.150 as export/bra/proc91.dvi.Z.

[HT93] John Harrison and Laurent Théry. Extending the hol theorem prover with a computer algebra system to reason about the reals. In Joyce and Seger [JS93b], pages 174–184.

[JS93a] Jeffrey Joyce and Carl Seger. The hol-voss system: Model-checking inside a general-purpose theorem-prover. In Joyce and Seger [JS93b], pages 185–198.

[JS93b] Jeffrey J. Joyce and Carl Seger, editors. *Proceedings of the 1993 International Workshop on the HOL theorem proving system and its applications*, volume 780 of *Lecture Notes in Computer Science*, UBC, Vancouver, Canada, 1993. Springer-Verlag.

[KC86] T. Knoblock and R. Constable. Formalized metareasoning in type theory. In *Proceedings of the First Annual Symposium on Logic in Computer Science*, pages 237–248, Cambridge, MA, USA, 1986. IEEE Computer Society Press.

[KK71] Georg Kreisel and Jean-Louis Krivine. *Elements of mathematical logic: model theory*. Studies in Logic and the Foundations of Mathematics. North-Holland, revised second edition, 1971. First edition 1967. Translation of the French 'Eléments de logique mathématique, théorie des modeles'.

[KKS91] Ramaya Kumar, Thomas Kropf, and Klaus Schneider. Integrating a first-order automatic prover in the hol environment. In Archer et al. [AJLW91], pages 170–176.

[KL68] Georg Kreisel and A. Lévy. Reflection principles and their use for establishing the complexity of axiomatic systems. *Zeitschrift fur Mathematical Logic*, 14:97–142, 1968.

[Knu73] Donald E. Knuth. *The Art of Computer Programming; Volume 3: Sorting and Searching*. Addison-Wesley Series in Computer Science and Information processing. Addison-Wesley, 1973.

[KP94] Senton Kromodimoeljo and William Pase. Proof logging and proof checking in never. In Basin et al. [BGK94], page 41.

[Kre52] Georg Kreisel. On the concepts of completeness and interpretation of formal systems. *Fundamenta Mathematicae*, 39:103–127, 1952.

[Kre56] Georg Kreisel. Some uses of metamathematics. *British Journal for the Philosophy of Science*, 7:161–173, 1956.

[Kre67] Georg Kreisel. Informal rigour and completeness proofs. In Imre Lakatos, editor, *Problems in the Philosophy of Mathematics: Proceedings of the International Colloquium in the Philosophy of Science*, pages 138–171, Bedford College, Regent's Park, London, 1967. North-Holland.

[Kri71] Jean-Louis Krivine. *Introduction to Axiomatic Set Theory*. Synthese Library. D. Reidel Publishing Company, 1971. Translation of the French 'Théorie Axiomatique des Ensembles', first published by Presses Universitaires de France, Paris. Translated by David Miller.

[Kun80] Kenneth Kunen. *Set Theory: An Introduction to Independence Proofs*, volume 102 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1980.

[Lan66] Edmund Landau. *Foundations of analysis: the arithmetic of whole, rational, irrational, and complex numbers. A supplement to textbooks on the differential and integral calculus*. Chelsea Publishing Company, third edition, 1966. Translated from German 'Grundlagen der Analysis' by F. Steinhardt.

[Lév60] A. Lévy. Principles of reflection in axiomatic set theory. *Fundamenta Mathematicae*, 49:1–10, 1960.

[LO88] E. Lusk and R. Overbeek, editors. *9th International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, Argonne, Illinois, USA, 1988. Springer-Verlag.

[Löb55] M. H. Löb. Solution of a problem of leon henkin. *Journal of Symbolic Logic*, 20:115–118, 1955.

[Luk70] Jan Łukasiewicz. The equivalential calculus. In L. Borkowski, editor, *Jan Łukasiewicz: Selected Works*, Studies in Logic and the Foundations of Mathematics, pages 250–277. North-Holland, 1970.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of OOPSLA 87*, pages 147–155. ACM, 1987.

[Mat92] Seán Matthews. Reflection in logical systems. In Yonezawa and Smith, editors, *Proceedings of IMSA'92: Reflection and Meta-level Architecture*, 1992.

[Mat94] Seán Matthews. A theory and its metatheory in FS_0. Publication unknown, 1994.

[MC94] Thomas F. Melham and Juanito Camilleri, editors. *Higher Order Logic Theorem Proving and Its Applications: Proceedings of the 7th International Workshop*, volume 859 of *Lecture Notes in Computer Science*, Valletta, Malta, 1994. Springer-Verlag.

[Mel89] Thomas F. Melham. Automating recursive type definitions in higher order logic. In *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer-Verlag, 1989.

[Mel93] Thomas F. Melham. *Higher Order Logic and Hardware Verification*, volume 31 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1993. Substantially a revision of Melham's thesis.

[Mil72] Robin Milner. Implementation and applications of scott's logic for computable functions. *ACM SIGPLAN Notices*, 7(1):1–6, January 1972.

[ML85] Per Martin-Löf. 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.

[MN88] Pattie Maes and D. Nardi, editors. *Meta-Level Architectures and Reflection.* North-Holland, 1988.

[MOD91] U. K. MOD. The procurement of safety critical software in defence equipment. Interim Defence Standard 00-55, UK Ministry of Defence, Directorate of Standardization, Kentigern House, 65 Brown Street, GLASGOW G2 8EX, UK, 1991.

[Mon66] Richard Montague. Fraenkel's addition to the axioms of zermelo. In Y. Bar-Hillel, E. I. J. Poznanski, M. O. Rabin, and A. Robinson, editors, *Essays on the Foundations of Mathematics: dedicated to A. A. Fraenkel on his seventieth anniversary*, pages 91–114. The Magnes Press, the Hebrew University, Jerusalem, 1966.

[MSB91] Seán Matthews, Alan Smaill, and David Basin. Experience with FS_0 as a framework theory. In Huet et al. [HPJ91], pages 231–252. Available by FTP from 129.215.160.150 as export/bra/proc91.dvi.Z.

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML*. The MIT Press, 1991.

[MT92] Ian A. Mason and Carolyn L. Talcott. References, local variables and operational reasoning. In *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 186–197, Santa Cruz, CA, USA, 1992. IEEE Computer Society Press.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[ORS92] S. Owre, J. M. Rushby, and N. Shanker. Pvs: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752, Saratoga, NY, 1992. Springer-Verlag.

[Pau87] Lawrence C. Paulson. *Logic and computation: interactive proof with Cambridge LCF*. Number 2 in Cambridge tracts in theoretical computer science. Cambridge University Press, 1987.

[Pau90] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. G. Odifreddi, editor, *Logic and Computer Science*, number 31 in APIC Studies in Data Processing, pages 361–386. Academic Press, 1990.

[Pot92] Garrell Pottinger. Completeness for the hol logic: Preliminary report. Posted to `info-hol` mailing list on 28th Jan 1992. Available in the `info-hol` archive by anonymous FTP from `ftp.cl.cam.ac.uk` in directory `hvg/info-hol-archive.`, 1992.

[Ram26] F. P. Ramsey. The foundations of mathematics. *Proceedings of the London Mathematical Society*, 25:338–384, 1926.

[Rob63] Abraham Robinson. *Introduction to model theory and to the metamathematics of algebra.* Studies in logic and the foundations of mathematics. North-Holland, 1963.

[Ros36] J. B. Rosser. Extensions of some theorems of gödel and church. *Journal of Symbolic Logic*, 1:87–91, 1936.

[RS94] Wolfgang Reif and Arno Schönegge. A reflection mechanism in kiv using structured specifications. In Basin et al. [BGK94], pages 19–21.

[Rud92] Piotr Rudnicki. An overview of the mizar project. Unpublished; available by anonymous FTP from `menaik.cs.ualberta.ca` as `pub/Mizar/Mizar_Over.tar.Z`, 1992.

[Rus91] John Rushby. Design choices in specification languages and verification systems. In Archer et al. [AJLW91], pages 194–204.

[Sha91] Stewart Shapiro. *Foundations without Foundationalism: a case for second-order logic.* Number 17 in Oxford Logic Guides. Clarendon Press, 1991.

[Sha94] N. Shankar. *Metamathematics, Machines and Gödel's Proof.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1994.

[SKK92] Klaus Schneider, Ramaya Kumar, and Thomas Kropf. Efficient representation and computation of tableaux proofs. In Claesen and Gordon [CG92], pages 39–57.

[Sli92] Konrad Slind. Adding new rules to an lcf-style logic implementation. In Claesen and Gordon [CG92], pages 549–559.

[Smi84] Brian Cantwell Smith. Reflection and semantics in lisp. In *Conference Record of the 14th ACM Symposium on Principles of Programming Languages*, pages 23–35. ACM, 1984.

[SRI91] SRI International and DSTO Australia. *The HOL System.* Cambridge Computer Science Research Center, SRI International, revised version, 1991. Four volumes.

[Sym93] Donald Syme. Reasoning with the formal definition of standard ml in hol. In Joyce and Seger [JS93b], pages 43–60.

[Tak78] Gaisi Takeuti. *Two applications of logic to mathematics.* Number 13 in Publications of the Mathematical Society of Japan. Iwanami Shoten, Tokyo, 1978.

[Tar56] Alfred Tarski. The concept of truth in formalized languages. In *Logic, Semantics and Metamathematics*, pages 152–278. Clarendon Press, 1956.

[Tay88] Paul Taylor. Using constructions as a metalanguage. LFCS Report Series ECS-LFCS-88-70, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, 1988.

[Tro73] A. S. Troelstra. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis.* Number 344 in Lecture Notes in Mathematics. Springer-Verlag, 1973.

[Tur39] A. M. Turing. Systems of logic based on ordinals. *Proceedings of the London Mathematical Society (2)*, 45:161–228, 1939.

[TW88] Carolyn Talcott and Richard Weyhrauch. Partial evaluation, higher-order abstractions, and reflection principles as system building tools. In Bjørner et al. [BEJ88], pages 507–529.

[vBJ77] L. S. van Bentham Jutting. *Checking Landau's "Grundlagen" in the AUTOMATH System.* PhD thesis, Eindhoven University of Technology, 1977.

[VG93] Myra VanInwegen and Elsa Gunter. Hol-ml. In Joyce and Seger [JS93b], pages 61–74.

[vW94a] Joakim von Wright. RefCalc — a contribution to the HOL system. Distributed with HOL system, 1994.

[vW94b] Joakim von Wright. Representing higher-order logic proofs in hol. In Melham and Camilleri [MC94], pages 456–470.

[vW94c] Joakim von Wright. Verifying modular programs in HOL. University of Cambridge Computer Laboratory Technical Report 324, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, 1994.

[Wey80] Richard W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, 13:133–170, 1980.

[Wey82] Richard W. Weyhrauch. An example of fol using metatheory. In Donald W. Loveland, editor, *Proceedings of the 6th Conference on Automated Deduction*, number 138 in Lecture Notes in Computer Science, pages 151–158, New York, 1982. Springer Verlag.

[WF86] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In *Conference Record of the 1986 ACM Symposium on LISP and Functional Programming*, pages 298–307. ACM, 1986.

[WT94] Richard W. Weyhrauch and Carolyn Talcott. The logic of fol systems: Formulated in set theory. In Neil D. Jones, Masami Hagiya, and Masahiko Sato, editors, *Logic, Language and Computation: Festschrift in Honor of Satoru Takasu*, number 792 in Lecture Notes in Computer Science, pages 119–132. Springer Verlag, 1994.

[Zhu94] Ming-Yuan Zhu. Computational reflection in powerepsilon. *ACM SIGPLAN Notices*, 29(1):13–19, January 1994.

Part VII

Appendices for Part III

Appendix A

AL specification of proof checker

A certain set of standard definitions are assumed as a prelude to the AL specification.

```
% ****************************************************************** %
% boolean functions %
% ****************************************************************** %

FUN NOT b =
 IF is_true b THEN false ELSE true END_IF
END_FUN

FUN AND a b =
  IF is_true a THEN b ELSE false END_IF
END_FUN

FUN OR a b =
  IF is_true a THEN true ELSE  b END_IF
END_FUN

% ****************************************************************** %
% natural numbers %
% ****************************************************************** %

TYPE
num = 0 :: is_zero | SUC (PRE: num) :: is_suc
END_TYPE

% ****************************************************************** %
% pairs %
% ****************************************************************** %
```

```
TYPE
prod = mk_prod (fst: *) (snd: **) :: is_prod
END_TYPE

SYNTAX
(a,b) = mk_prod a b
END_SYNTAX

% ****************************************************************** %
% strings %
% ****************************************************************** %

TYPE
ascii = ASCII (a0: bool) (a1: bool) (a2: bool) (a3: bool)
              (a4: bool) (a5: bool) (a6: bool) (a7: bool) :: is_ascii
END_TYPE

TYPE
string = null_str :: is_null_str |
         STR (fst_char: ascii) (rst_char: string) :: is_STR
END_TYPE

SYNTAX
`` = null_str
`a`  = STR a null_str
`aX` = STR a `X`
END_SYNTAX

% ****************************************************************** %
% list %
% ****************************************************************** %

TYPE
list = null :: is_null | cons (car: *) (cdr: list) :: is_cons
END_TYPE

% ****************************************************************** %
% list functions %
% ****************************************************************** %

FUN EVERY Pred l1 =
CASE_fun [
(is_null l1, true);
(is_cons l1, (Pred (car l1)) AND (EVERY Pred (cdr l1)) )
]
END_FUN

FUN LENGTH l1 =
CASE_fun [
(is_null l1, 0);
(is_cons l1, SUC (LENGTH (cdr l1)) )
]
END_FUN

FUN MAP fn l1 =
```

```
CASE_fun [
(is_null l1, []);
(is_cons l1, cons (fn(car l1)) (cdr l1) )
]
END_FUN

SYNTAX
[] = null ;
[id_a] = cons id_a null ;
[id_0 ; X] = cons id_0 [X] ;
END_SYNTAX


% ****************************************************************** %
% case statement %
% ****************************************************************** %

FUN CASE_fun pairlist =
IF (is_null pairlist)
THEN ?
ELSE IF fst(car pairlist)
     THEN snd(car pairlist)
     ELSE CASE_fun (cdr pairList)
     END_IF
END_IF
END_FUN


% ****************************************************************** %
% set %
% ****************************************************************** %

TYPE
set =  SetOf (ListOf: * list) :: is_set
END_TYPE

FUN empty = SetOf [] END_FUN

FUN is_empty s = (s = empty) END_FUN

FUN is_insert s = NOT (s = empty) END_FUN

FUN IS_IN (elem: *) (s: * set) =
     mem elem (ListOf s)
END_FUN

FUN INSERT (elem: *) (s: * set) =
    IF mem elem (ListOf s)
    THEN s
    ELSE SetOf (cons elem (ListOf s))
    END_IF
END_FUN

FUN Setify l1 =
 IF is_null l1
```

```
 THEN []
 ELSE IF mem (hd l1) (tl l1)
      THEN Setify (tl l1)
      ELSE cons (hd l1) (Setify (tl l1))
      END_IF
 END_IF
END_FUN

FUN UNION s1 s2 =
 SetOf(Setify (Append (ListOf s1)(ListOf s1)))
END_FUN


FUN Remove el lst =
 IF is_null l1 THEN []
 ELSE IF (el = (hd l1))
      THEN Remove el (tl l1)
      ELSE cons (hd l1) (Remove el (tl l1))
      END_IF
 END_IF
END_FUN

FUN DELETE el s =
 IF mem el (ListOf s)
 THEN SetOf (Remove el (ListOf s))
 ELSE ?
END_FUN

FUN SEVERY pr s =
  EVERY pr (ListOf s)
END_FUN

SYNTAX  {} = empty_set,
        {id_0} = id_0 INSERT empty_set
        {id_1 ; _b} = id_1 INSERT {_b},
END_SYNTAX
```

The rest of the AL is presented in sections corresponding to break-up of the AL into files and corresponding to the structuring of the original theory of HOL proofs on which it is based.

```
% ****************************************************************** %
% auxiliary functions %
% ****************************************************************** %

FUN EVERY2 (Pred:*a->*b->bool) (l1:*a list) (l2:*b list) :bool =
CASE
(is_null l1, true);
(is_cons l1,
 (Pred (car l1) (car l2)) AND (EVERY2 Pred (cdr l1) (cdr l2)) )
END_CASE
END_FUN

FUN LAPPEND (l1:(*a list)list) :*a list =
```

```
CASE
(is_null l1, []);
(is_cons l1, APPEND (car l1) (LAPPEND (cdr l1)) )
END_CASE
END_FUN

FUN LUNION (l1:((*a)set) list) :(*a)set =
CASE
(is_null l1, empty);
(is_cons l1, (car l1) UNION (LUNION (cdr l1)) )
END_CASE
END_FUN

FUN lor (l1:bool list) :bool =
CASE
(is_null l1, false);
(is_cons l1, (car l1) OR (lor (cdr l1)) )
END_CASE
END_FUN

FUN land (l1:bool list) :bool =
CASE
(is_null l1, true);
(is_cons l1, (car l1) AND (land (cdr l1)) )
END_CASE
END_FUN

FUN corr1 (x:*a) (l1:(*a#*b) list) :*b =
CASE
(is_null l1, ?);
(is_cons l1,
 IF (x = fst(car l1))
 THEN snd(car l1)
 ELSE corr1 x (cdr l1)
 END_IF)
END_CASE
END_FUN

FUN corr2 (x:*b) (l1:(*a#*b) list) :*a =
CASE
(is_null l1, ?);
(is_cons l1,
 IF (x = snd(car l1))
 THEN fst(car l1)
 ELSE corr2 x (cdr l1)
 END_IF)
END_CASE
END_FUN

FUN mem (x:*a) (l1:*a list) :bool =
CASE
(is_null l1, false);
(is_cons l1, (x = (car l1)) OR (mem x (cdr l1)) )
END_CASE
END_FUN
```

```
FUN lmem (l1:*a list) (l2:*a list) :bool =
CASE
(is_null l1, true);
(is_cons l1, (mem (car l1) l2) AND (lmem (cdr l1) l2) )
END_CASE
END_FUN

FUN mem1 (x:*a) (l1:(*a#*b) list) :bool =
CASE
(is_null l1, false);
(is_cons l1, (x = fst(car l1)) OR (mem1 x (cdr l1)) )
END_CASE
END_FUN

FUN mem2 (x:*b) (l1:(*a#*b) list) : bool =
CASE
(is_null l1, false);
(is_cons l1, (x = snd(car l1)) OR (mem2 x (cdr l1)) )
END_CASE
END_FUN

FUN nocontr (l1:(*a#*b) list) :bool =
CASE
(is_null l1, true);
(is_cons l1,
   ((NOT(mem2 (snd(car l1)) (cdr l1))) OR
       (corr2 (snd(car l1)) (cdr l1) = fst(car l1))) AND
    (nocontr (cdr l1))  )
END_CASE
END_FUN
```

The specification of type and its associated functions is as follows

```
% ***************************************************************** %
% Representation of HOL types %
% ***************************************************************** %

TYPE
Type = Tyvar (Tyvar_nam: string) :: Is_Tyvar |
        Tyop (Tyop_nam: string) (Tyop_tyl: Type list) :: Is_Tyop
END_TYPE


FUN XMAP2 (afn:*a->*b->*c) (l1:*a list) (l2:*b list) :(*c)list =
CASE
(is_null l1, []);
(is_cons l1, cons (afn (car l1) (car l2))
                  (XMAP2 afn (cdr l1) (cdr l2)) )
END_CASE
END_FUN


FUN Type_OK (tyl:(string # num)list) (ty:Type) :bool =
```

```
CASE
(Is_Tyvar ty, true);
(Is_Tyop ty, (mem1 (Tyop_nam ty) tyl) AND
             (LENGTH (Tyop_tyl ty) = corr1 (Tyop_nam ty) tyl) AND
             EVERY (Type_OK tyl) (Tyop_tyl ty) )
END_CASE
END_FUN

FUN Type_compat (ty1:Type) (ty2:Type) :bool =
CASE
(Is_Tyvar ty2, true);
(Is_Tyop ty2, (Is_Tyop ty1) AND
              ((Tyop_nam ty1) = (Tyop_nam ty2)) AND
              (LENGTH (Tyop_tyl ty1) = LENGTH (Tyop_tyl ty2)) AND
              (EVERY2 Type_compat (Tyop_tyl ty1) (Tyop_tyl ty2)) )
END_CASE
END_FUN


FUN Type_occurs (str:string) (ty:Type) :bool =
CASE
(Is_Tyvar ty, (str = Tyvar_nam ty));
(Is_Tyop ty,  lor (MAP (Type_occurs str) (Tyop_tyl ty)) )
END_CASE
END_FUN


FUN Type_replace (l:(Type#string)list) (ty:Type) :Type =
CASE
(Is_Tyvar ty, IF mem2 (Tyvar_nam ty) l
              THEN corr2 (Tyvar_nam ty) l
              ELSE ty
              END_IF );
(Is_Tyop ty, Tyop (Tyop_nam ty) (MAP(Type_replace l)(Tyop_tyl ty)) )
END_CASE
END_FUN


FUN Type_instlx (ty1:Type) (ty2:Type) :(Type#string)list =
CASE
(Is_Tyvar ty1, [(ty2,Tyvar_nam ty1)]);
(Is_Tyop ty1,
 LAPPEND(XMAP2 Type_instlx (Tyop_tyl ty1) (Tyop_tyl ty2)) )
END_CASE
END_FUN


FUN Type_instl (ty1:Type) (ty2:Type) :(Type#string)list =
    Type_instlx ty2 ty1
END_FUN
```

The specification of term and associated functions are described by the following AL:

```
% ****************************************************************** %
```

```
% Representation of HOL terms %
% *************************************************************** %

TYPE
Pterm = Const (Const_nam: string) (Const_ty: Type) :: Is_Const |
        Var (Var_var: (string#Type)) :: Is_Var |
        App (App_fun: Pterm) (App_arg: Pterm) :: Is_App |
        Lam (Lam_var: (string#Type)) (Lam_bod: Pterm) :: Is_Lam
END_TYPE

FUN Ptype_of  (ptm: Pterm) :Type =
CASE
(Is_Const ptm, Const_ty ptm);
(Is_Var ptm, snd(Var_var ptm) );
(Is_App ptm, car(cdr(Tyop_tyl(Ptype_of (App_fun ptm)))) );
(Is_Lam ptm, Tyop `fun` [snd (Lam_var ptm); Ptype_of (Lam_bod ptm)] )
END_CASE
END_FUN

FUN Pboolean (t: Pterm) :bool = (Ptype_of t = Tyop `bool` []) END_FUN

FUN Pwell_typed
(Typl:(string#num)list) (Conl:(string#Type)list) (ptm:Pterm) :bool =
CASE
(Is_Const ptm,
         (mem1 (Const_nam ptm) Conl) AND
         (Type_OK Typl (Const_ty ptm)) AND
         (Type_compat (Const_ty ptm) (corr1 (Const_nam ptm) Conl)) AND
         (nocontr(Type_instl (Const_ty ptm) (corr1 (Const_nam ptm)Conl))) );
(Is_Var ptm, Type_OK Typl (snd (Var_var ptm)) );
(Is_App ptm,
         (Pwell_typed Typl Conl (App_fun ptm)) AND
         (Pwell_typed Typl Conl (App_arg ptm)) AND
         (Is_Tyop(Ptype_of (App_fun ptm))) AND
         (Tyop_nam(Ptype_of (App_fun ptm)) = `fun`) AND
         (LENGTH(Tyop_tyl(Ptype_of (App_fun ptm))) = 2) AND
         (car(Tyop_tyl(Ptype_of (App_fun ptm))) = Ptype_of (App_arg ptm)) );
(Is_Lam ptm,
         (Pwell_typed Typl Conl (Lam_bod ptm)) AND
         (Type_OK Typl(snd (Lam_var ptm))))
END_CASE
END_FUN

FUN Pfree (x:string#Type) (ptm: Pterm) :bool =
CASE
(Is_Const ptm, false);
(Is_Var ptm, (Var_var ptm = x) );
(Is_App ptm, (Pfree x (App_fun ptm)) OR (Pfree x (App_arg ptm)) );
(Is_Lam ptm, (NOT(Lam_var ptm = x)) AND (Pfree x (Lam_bod ptm)) )
END_CASE
END_FUN


FUN Pbound (x:string#Type) (ptm: Pterm) :bool =
CASE
```

```
(Is_Const ptm, false);
(Is_Var ptm, false );
(Is_App ptm, (Pbound x (App_fun ptm)) OR (Pbound x (App_arg ptm)) );
(Is_Lam ptm, (Lam_var ptm = x) OR (Pbound x (Lam_bod ptm)) )
END_CASE
END_FUN


FUN Poccurs (x:string#Type) (t:Pterm) :bool =
   (Pfree x t) OR (Pbound x t) END_FUN


FUN Plnotfree (l1:(string#Type) list) (t:Pterm) :bool =
CASE
(is_null l1, true);
(is_cons l1, (NOT(Pfree (car l1) t)) AND (Plnotfree (cdr l1) t) )
END_CASE
END_FUN


FUN Plnotbound (l1:(string#Type) list) (t:Pterm) :bool =
CASE
(is_null l1, true);
(is_cons l1, (NOT(Pbound (car l1) t)) AND (Plnotbound (cdr l1) t) )
END_CASE
END_FUN

FUN Plnotoccurs (xl:(string#Type) list) (t:Pterm) :bool =
    (Plnotfree xl t) AND (Plnotbound xl t) END_FUN

FUN NotPfree (x:string#Type) (t:Pterm) :bool = NOT(Pfree x t) END_FUN

FUN Pallnotfree (x:string#Type) (tms:(Pterm)set) :bool =
    SEVERY (NotPfree x) tms END_FUN

FUN Plallnotfree (xl:(string#Type) list) (tms:(Pterm)set) =
    SEVERY (Plnotfree xl) tms END_FUN

FUN Palreplace1 (t':Pterm) (vvl:((string#Type)#(string#Type))list)
           (tvl:(Pterm # (string # Type))list) (ptm: Pterm) :bool =
CASE
(Is_Const ptm, (t' = ptm) );

(Is_Var ptm, IF (Is_Var t') AND (mem1 (Var_var t') vvl)
             THEN ((Var_var ptm) = corr1(Var_var t')vvl)
             ELSE (NOT(mem1 (Var_var ptm) vvl)) AND
                  (IF (mem2 (Var_var ptm) tvl)
                   THEN (t' = corr2 (Var_var ptm) tvl)
                   ELSE (t' = ptm)
                   END_IF)
             END_IF );

(Is_App ptm, (Is_App t') AND
         Palreplace1(App_fun t')vvl tvl (App_fun ptm) AND
         Palreplace1(App_arg t')vvl tvl (App_arg ptm) );
```

```
(Is_Lam ptm, (Is_Lam t') AND
         (snd(Lam_var t') = snd (Lam_var ptm)) AND
         Palreplace1
         (Lam_bod t')
         (cons(Lam_var t',Lam_var ptm)vvl)
         tvl
        (Lam_bod ptm) )

END_CASE
END_FUN


FUN Palreplace (t':Pterm) (tvl:(Pterm #(string#Type))list)
               (t:Pterm) :bool =
    Palreplace1 t' [] tvl t END_FUN


FUN Palpha (t':Pterm) (t:Pterm) :bool = Palreplace t' [] t  END_FUN


FUN Psubst_triples (l1:(Pterm#(Pterm#(string#Type)))list):bool =
CASE
(is_null l1, true);
(is_cons l1,
         (Ptype_of(fst(car l1)) = Ptype_of(fst(snd(car l1)))) AND
         (Ptype_of(fst(car l1)) = snd(snd(snd(car l1)))) AND
         Psubst_triples (cdr l1))
END_CASE
END_FUN

FUN list13 (l1:(*a#(*b #*c))list) :(*a#*c)list =
CASE
(is_null l1, []);
(is_cons l1, cons(fst (car l1),snd(snd (car l1)))(list13 (cdr l1)) )
END_CASE
END_FUN


FUN Psubst (Typl:(string#num)list) (Conl:(string#Type)list) (t':Pterm)
           (ttvl:(Pterm # (Pterm # (string # Type)))list)
           (td:Pterm) (t:Pterm) :bool =
        (Psubst_triples ttvl) AND
        (Pwell_typed Typl Conl td) AND
        (Plnotoccurs(MAP snd(MAP snd ttvl))t) AND
        (Palreplace t (MAP snd ttvl)td) AND
        (Palreplace t'(list13 ttvl) td)
END_FUN


FUN Pbeta (t':Pterm) (x:string#Type) (t1:Pterm) (t2:Pterm) :bool =
    Palreplace t' [(t2,x)] t1 END_FUN


% ****************************************************************** %
```

```
% Type occurance, instantiation %
% ************************************************************** %

FUN Pty_occurs (a:string) (ptm:Pterm) :bool =
CASE
(Is_Const ptm, Type_occurs a (Const_ty ptm) );
(Is_Var ptm, Type_occurs a (snd(Var_var ptm)) );
(Is_App ptm, (Pty_occurs a (App_fun ptm)) OR (Pty_occurs a(App_arg ptm)) );
(Is_Lam ptm, (Type_occurs a(snd(Lam_var ptm))) OR (Pty_occurs a(Lam_bod ptm)))
END_CASE
END_FUN

FUN NotPty_occurs (a:string) (t:Pterm) :bool = NOT(Pty_occurs a t) END_FUN

FUN Pty_snotoccurs (a:string) tms = SEVERY (NotPty_occurs a) tms END_FUN

FUN Plty_snotoccurs l1 tml =
CASE
(is_null l1, true);
(is_cons l1, (Pty_snotoccurs (car l1) tml) AND
             (Plty_snotoccurs (cdr l1) tml) )
END_CASE
END_FUN


FUN Pnewfree1 t bl ptm =
CASE
(Is_Const ptm, [] );
(Is_Var ptm, IF (mem (Var_var ptm) bl) OR (fst(Var_var t) = fst (Var_var ptm))
             THEN []
             ELSE  [(Var_var t,Var_var ptm)]
             END_IF );

(Is_App ptm, APPEND(Pnewfree1(App_fun t)bl (App_fun ptm))
                   (Pnewfree1(App_arg t)bl (App_arg ptm)) );
(Is_Lam ptm, Pnewfree1(Lam_bod t)(cons(Lam_var ptm) bl) (Lam_bod ptm) )
END_CASE
END_FUN

FUN Pnewfree t' t = Pnewfree1 t' [] t END_FUN

FUN Ptyinst1
    (t:Pterm) (bvl:((string#Type)#(string#Type))list)
    (fvl:((string#Type)#(string#Type))list)
    (tyl:(Type # string)list) (ptm:Pterm)   =

CASE
(Is_Const ptm, (t = Const (Const_nam ptm) (Type_replace tyl (Const_ty ptm))) );

(Is_Var ptm,
     IF ((Is_Var t) AND (mem1(Var_var t)bvl))
     THEN ((Var_var ptm) = corr1(Var_var t)bvl) AND
          (snd(Var_var t) = Type_replace tyl(snd (Var_var ptm)))
     ELSE
         (IF (mem2 (Var_var ptm) fvl)
```

```
          THEN
           (t = Var(fst(corr2 (Var_var ptm) fvl),
                    Type_replace tyl(snd(Var_var ptm))))
          ELSE
           (t = Var(fst (Var_var ptm),
                    Type_replace tyl(snd (Var_var ptm))))
          END_IF)
     END_IF );

(Is_App ptm, Is_App t AND
             Ptyinst1(App_fun t)bvl fvl tyl (App_fun ptm) AND
             Ptyinst1(App_arg t)bvl fvl tyl (App_arg ptm) );

(Is_Lam ptm, Is_Lam t AND
            (NOT(mem1(Lam_var t)fvl)) AND
            (snd(Lam_var t) = Type_replace tyl(snd (Lam_var ptm))) AND
             Ptyinst1
            (Lam_bod t)
            (cons(Lam_var t,(Lam_var ptm))bvl)
            fvl
            tyl
            (Lam_bod ptm)  )
END_CASE
END_FUN

FUN Ptyinst ass t' tyl t =
        Ptyinst1 t' [] (Pnewfree t' t) tyl t AND
        Plallnotfree (MAP snd(Pnewfree t' t)) ass
END_FUN
```

The final set of AL definitions specifies sequents, inferences and relevant functions. The specification of `Is_proof` provides the overall specification of what a proof checking function should do.

```
% *************************************************************** %
% Representation of HOL inference %
% *************************************************************** %

TYPE
Psequent = Pseq (Pseq_assum:(Pterm)set) (Pseq_concl: Pterm) :: Is_Pseq
END_TYPE

FUN Pseq_boolean pseq =
 SEVERY Pboolean (Pseq_assum pseq) AND
 Pboolean (Pseq_concl pseq)
END_FUN

FUN Pseq_well_typed Typl Conl pseq =
        SEVERY(Pwell_typed Typl Conl)(Pseq_assum pseq) AND
        Pwell_typed Typl Conl (Pseq_concl pseq)
END_FUN

FUN PEQ t1 t2 =
        App
```

```
        (App
         (Const
          `=`
          (Tyop
           `fun`
           [Ptype_of t1;Tyop `fun`[Ptype_of t1;Tyop `bool`[]]]))
         t1)
        t2
END_FUN

FUN PIMP t1 t2 =
        App
        (App
         (Const
          `==>`
          (Tyop
           `fun`
           [Tyop `bool`[];Tyop `fun`[Tyop `bool`[];Tyop `bool`[]]]))
         t1)
        t2
END_FUN

FUN Is_EQtm t =
        Is_App t AND
        Is_App(App_fun t) AND
        (App_fun(App_fun t) =
         Const
         `=`
         (Tyop
          `fun`
          [Ptype_of(App_arg t);
           Tyop `fun`[Ptype_of(App_arg t);Tyop `bool`[]]]))
END_FUN

% ****************************************************************** %
% requirements for the eight primitive inferences %
% ****************************************************************** %

FUN PASSUME Typl Conl pseq tm =
        Pwell_typed Typl Conl tm AND
        Pboolean tm AND
        (Pseq_concl pseq = tm) AND
        (Pseq_assum pseq = (SetOf[tm]))
END_FUN

FUN PREFL Typl Conl pseq tm =
        Pwell_typed Typl Conl tm AND
        ((Pseq_assum pseq) = empty) AND
        ((Pseq_concl pseq) = PEQ tm tm)
END_FUN

FUN PBETA_CONV Typl Conl pseq tm =
        (Pwell_typed Typl Conl tm) AND
        (Pseq_assum pseq = empty) AND
        (Is_App tm) AND
```

```
        Is_Lam(App_fun tm) AND
        ((Pseq_concl pseq) = PEQ tm(App_arg (Pseq_concl pseq))) AND
        (Pbeta
        (App_arg (Pseq_concl pseq))
        (Lam_var(App_fun tm))
        (Lam_bod(App_fun tm))
        (App_arg tm))
END_FUN

FUN Psubst_newlist (l1:(Psequent # (string # Type))list)
                   :(Pterm # (Pterm # (string # Type))) list =
CASE
(is_null l1, []);
(is_cons l1, cons
         (App_arg(Pseq_concl(fst (car l1))),
          (App_arg(App_fun(Pseq_concl(fst (car l1)))),snd (car l1)))
         (Psubst_newlist (cdr l1)) )
END_CASE
END_FUN


FUN PSUBST Typl Conl pseq thdl td th =
        Pwell_typed Typl Conl td AND
        EVERY Is_EQtm (MAP Pseq_concl(MAP fst thdl)) AND
        Psubst Typl Conl
        (Pseq_concl pseq) (Psubst_newlist thdl)td(Pseq_concl th) AND
        ((Pseq_assum pseq) =
         ((Pseq_assum th) UNION (LUNION(MAP Pseq_assum(MAP fst thdl)))))
END_FUN


FUN PABS Typl Conl pseq tm th =
        Pwell_typed Typl Conl tm AND
        Is_EQtm(Pseq_concl th) AND
        Is_Var tm AND
        Type_OK Typl(snd(Var_var tm)) AND
        ((Pseq_concl pseq) =
         PEQ
         (Lam(Var_var tm)(App_arg(App_fun(Pseq_concl th))))
         (Lam(Var_var tm)(App_arg(Pseq_concl th)))) AND
        (Pseq_assum pseq = Pseq_assum th) AND
        Pallnotfree(Var_var tm)(Pseq_assum pseq)
END_FUN

FUN PINST_TYPE Typl pseq tyl th =
        EVERY(Type_OK Typl)(MAP fst tyl) AND
        Ptyinst (Pseq_assum pseq) (Pseq_concl pseq) tyl (Pseq_concl th) AND
        Plty_snotoccurs(MAP snd tyl)(Pseq_assum pseq) AND
        (Pseq_assum pseq = Pseq_assum th)
END_FUN

FUN PDISCH Typl Conl pseq tm th =
        Pwell_typed Typl Conl tm AND
        Pboolean tm AND
        (Pseq_concl pseq = PIMP tm(Pseq_concl th)) AND
```

```
        (Pseq_assum pseq = ((Pseq_assum th) DELETE tm))
END_FUN

FUN PMP pseq th1 th2 =
        (Pseq_concl th1 = PIMP(Pseq_concl th2)(Pseq_concl pseq)) AND
        ((Pseq_assum pseq) = ((Pseq_assum th1) UNION (Pseq_assum th2)))
END_FUN


% *************************************************************** %
% Inference %
% *************************************************************** %

TYPE
Inference =
AX_inf (AX_con: Psequent)  :: is_AX |
AS_inf (AS_con: Psequent) (AS_tm: Pterm)  :: is_AS |
RE_inf (RE_con: Psequent) (RE_tm: Pterm)  :: is_RE |
BE_inf (BE_con: Psequent) (BE_tm: Pterm)  :: is_BE |
SU_inf (SU_con: Psequent)
       (SU_eqthms: (Psequent#(string#Type)) list)
       (SU_tmdum: Pterm) (SU_thm: Psequent)  :: is_SU |
AB_inf (AB_con: Psequent) (AB_tm: Pterm) (AB_eq: Psequent)  :: is_AB |
IN_inf (IN_con: Psequent) (IN_tyl:(Type#string)list) (IN_thm: Psequent) :: is_IN |
DI_inf (DI_con: Psequent) (DI_tm: Pterm) (DI_thm: Psequent) :: is_DI |
MP_inf (MP_con: Psequent) (MP_ithm: Psequent) (MP_thm: Psequent)  :: is_MP
END_TYPE


FUN Inf_concl inf =
CASE
(is_AX inf, AX_con inf);
(is_AS inf, AS_con inf);
(is_RE inf, RE_con inf);
(is_BE inf, BE_con inf);
(is_SU inf, SU_con inf);
(is_AB inf, AB_con inf);
(is_IN inf, IN_con inf);
(is_DI inf, DI_con inf);
(is_MP inf, MP_con inf)
END_CASE
END_FUN


FUN Inf_hyps inf =
CASE
(is_AX inf, []);
(is_AS inf, []);
(is_RE inf, []);
(is_BE inf, []);
(is_SU inf, cons (SU_thm inf) (MAP fst (SU_eqthms inf)) );
(is_AB inf, [AB_eq inf] );
(is_IN inf, [IN_thm inf] );
(is_DI inf, [DI_thm inf] );
(is_MP inf, [(MP_ithm inf); (MP_thm inf)] )
```

```
END_CASE
END_FUN


FUN OK_Inf Typl Conl Axil inf =
CASE
(is_AX inf, mem (AX_con inf) Axil);
(is_AS inf, PASSUME Typl Conl (AS_con inf) (AS_tm inf) );
(is_RE inf, PREFL Typl Conl (RE_con inf) (RE_tm inf) );
(is_BE inf, PBETA_CONV Typl Conl (BE_con inf) (BE_tm inf)  );
(is_SU inf, PSUBST Typl Conl
            (SU_con inf)(SU_eqthms inf)(SU_tmdum inf)(SU_thm inf) );
(is_AB inf, PABS Typl Conl (AB_con inf) (AB_tm inf) (AB_eq inf) );
(is_IN inf, PINST_TYPE Typl (IN_con inf) (IN_tyl inf) (IN_thm inf) );
(is_DI inf, PDISCH Typl Conl (DI_con inf) (DI_tm inf) (DI_thm inf) );
(is_MP inf, PMP (MP_con inf) (MP_ithm inf) (MP_thm inf) )
END_CASE
END_FUN


FUN Is_proof Typl Conl Axil l =
CASE
(is_null l, true);
(is_cons l, OK_Inf Typl Conl Axil (car l) AND
            lmem(Inf_hyps (car l))(MAP Inf_concl (cdr l)) AND
            Is_proof Typl Conl Axil (cdr l) )
END_CASE
END_FUN
```

Appendix B

HOL generated from AL specification

Before the AL is translated into HOL a number of definitions for compatability are made:

```
CASE_FN =
|- (CASE_FN[] = ARB) /\
   (!h tl. CASE_FN(CONS h tl) = (FST h => SND h | CASE_FN tl))

IS_CONS = |- (IS_CONS[] = F) /\ (!h tl. IS_CONS(CONS h tl) = T)

AND = |- $AND = $/\

OR = |- $OR = $\/

IF_FN = |- !a b c. IF_FN(a,b,c) = (a => b | c)

IS_INSERT = |- !st. IS_INSERT st = ~(st = {})

IS_EMPTY = |- !st. IS_EMPTY st = (st = {})

SEVERY_SPEC = |- !P s. SEVERY P s = (!x. x IN s ==> P x)

SEVERY_DEF =
|- (!P. SEVERY P{} = T) /\
   (!P x s. SEVERY P(x INSERT s) = P x /\ SEVERY P s)

SetOf =
|- (SetOf[] = {}) /\ (!a l. SetOf(CONS a l) = a INSERT (SetOf l))

SetOf_lem = |- SetOf[tm] = {tm}
```

The rest of the HOL defintions are generated automatically from the AL specification. They are presented in groups corresponding to the break-up of the AL into different files which in turn reflects the structuring of the original theory of HOL proofs on which it is based.

```
EVERY2 =
|- EVERY2 =
   (@EVERY2'.
     !Pred l1 l2.
      EVERY2' Pred l1 l2 =
      CASE_FN
      [NULL l1,T;
       IS_CONS l1,(Pred(HD l1)(HD l2)) AND (EVERY2' Pred(TL l1)(TL l2))])

LAPPEND =
|- LAPPEND =
   (@LAPPEND'.
     !l1.
      LAPPEND' l1 =
      CASE_FN[NULL l1,[];IS_CONS l1,APPEND(HD l1)(LAPPEND'(TL l1))])

LUNION =
|- LUNION =
   (@LUNION'.
     !l1.
      LUNION' l1 =
      CASE_FN[NULL l1,{};IS_CONS l1,(HD l1) UNION (LUNION'(TL l1))])

lor =
|- lor =
   (@lor'.
     !l1.
      lor' l1 = CASE_FN[NULL l1,F;IS_CONS l1,(HD l1) OR (lor'(TL l1))])

land =
|- land =
   (@land'.
     !l1.
      land' l1 =
      CASE_FN[NULL l1,T;IS_CONS l1,(HD l1) AND (land'(TL l1))])

corr1 =
|- corr1 =
   (@corr1'.
     !x l1.
      corr1' x l1 =
      CASE_FN
      [NULL l1,ARB;
       IS_CONS l1,IF_FN((x = FST(HD l1)),SND(HD l1),corr1' x(TL l1))])

corr2 =
|- corr2 =
   (@corr2'.
```

```
    !x l1.
     corr2' x l1 =
     CASE_FN
     [NULL l1,ARB;
      IS_CONS l1,IF_FN((x = SND(HD l1)),FST(HD l1),corr2' x(TL l1))])

mem =
|- mem =
   (@mem'.
     !x l1.
      mem' x l1 =
      CASE_FN[NULL l1,F;IS_CONS l1,(x = HD l1) OR (mem' x(TL l1))])

lmem =
|- lmem =
   (@lmem'.
     !l1 l2.
      lmem' l1 l2 =
      CASE_FN[NULL l1,T;IS_CONS l1,(mem(HD l1)l2) AND (lmem'(TL l1)l2)])

mem1 =
|- mem1 =
   (@mem1'.
     !x l1.
      mem1' x l1 =
      CASE_FN[NULL l1,F;IS_CONS l1,(x = FST(HD l1)) OR (mem1' x(TL l1))])

mem2 =
|- mem2 =
   (@mem2'.
     !x l1.
      mem2' x l1 =
      CASE_FN[NULL l1,F;IS_CONS l1,(x = SND(HD l1)) OR (mem2' x(TL l1))])

nocontr =
|- nocontr =
   (@nocontr'.
     !l1.
      nocontr' l1 =
      CASE_FN
      [NULL l1,T;
       IS_CONS l1,
       ((~mem2(SND(HD l1))(TL l1)) OR
        (corr2(SND(HD l1))(TL l1) = FST(HD l1))) AND
       (nocontr'(TL l1))])
```

The following definitions are introduced by the translation of `type.al`. It begins with the axiom for the type `Type`.

```
|- !f0 f1.
    ?! fn.
     (!s. fn(Tyvar s) = f0 s) /\
     (!s Tlst. fn(Tyop s Tlst) = f1(MAP fn Tlst)s Tlst)
```

```
XMAP2 =
|- XMAP2 =
   (@XMAP2'.
     !afn l1 l2.
      XMAP2' afn l1 l2 =
      CASE_FN
      [NULL l1,[];
       IS_CONS l1,CONS(afn(HD l1)(HD l2))(XMAP2' afn(TL l1)(TL l2))])

Type_OK =
|- Type_OK =
   (@Type_OK'.
     !tyl ty.
      Type_OK' tyl ty =
      CASE_FN
      [Is_Tyvar ty,T;
       Is_Tyop ty,
       (mem1(Tyop_nam ty)tyl) AND
       ((LENGTH(Tyop_tyl ty) = corr1(Tyop_nam ty)tyl) AND
        (EVERY(Type_OK' tyl)(Tyop_tyl ty)))])

Type_compat =
|- Type_compat =
   (@Type_compat'.
     !ty1 ty2.
      Type_compat' ty1 ty2 =
      CASE_FN
      [Is_Tyvar ty2,T;
       Is_Tyop ty2,
       (Is_Tyop ty1) AND
       ((Tyop_nam ty1 = Tyop_nam ty2) AND
        ((LENGTH(Tyop_tyl ty1) = LENGTH(Tyop_tyl ty2)) AND
         (EVERY2 Type_compat'(Tyop_tyl ty1)(Tyop_tyl ty2))))])

Type_occurs =
|- Type_occurs =
   (@Type_occurs'.
     !str ty.
      Type_occurs' str ty =
      CASE_FN
      [Is_Tyvar ty,(str = Tyvar_nam ty);
       Is_Tyop ty,lor(MAP(Type_occurs' str)(Tyop_tyl ty))])

Type_replace =
|- Type_replace =
   (@Type_replace'.
     !l ty.
      Type_replace' l ty =
      CASE_FN
      [Is_Tyvar ty,IF_FN(mem2(Tyvar_nam ty)l,corr2(Tyvar_nam ty)l,ty);
       Is_Tyop ty,Tyop(Tyop_nam ty)(MAP(Type_replace' l)(Tyop_tyl ty))])

Type_instlx =
|- Type_instlx =
```

```
   (@Type_instlx'.
     !ty1 ty2.
      Type_instlx' ty1 ty2 =
      CASE_FN
      [Is_Tyvar ty1,[ty2,Tyvar_nam ty1];
       Is_Tyop ty1,
       LAPPEND(XMAP2 Type_instlx'(Tyop_tyl ty1)(Tyop_tyl ty2))])

Type_instl =
|- Type_instl =
   (@Type_instl'. !ty1 ty2. Type_instl' ty1 ty2 = Type_instlx ty2 ty1)
```

Next are the definitions generated by `term.ml`.

```
|- !f0 f1 f2 f3.
    ?! fn.
     (!s T'. fn(Const s T') = f0 s T') /\
     (!p. fn(Var p) = f1 p) /\
     (!P1 P2. fn(App P1 P2) = f2(fn P1)(fn P2)P1 P2) /\
     (!p P. fn(Lam p P) = f3(fn P)p P)


Ptype_of =
|- Ptype_of =
   (@Ptype_of'.
     !ptm.
      Ptype_of' ptm =
      CASE_FN
      [Is_Const ptm,Const_ty ptm;Is_Var ptm,SND(Var_var ptm);
       Is_App ptm,HD(TL(Tyop_tyl(Ptype_of'(App_fun ptm))));
       Is_Lam ptm,Tyop `fun`[SND(Lam_var ptm);Ptype_of'(Lam_bod ptm)]])

Pboolean =
|- Pboolean =
   (@Pboolean'. !t. Pboolean' t = (Ptype_of t = Tyop `bool`[]))

Pwell_typed =
|- Pwell_typed =
   (@Pwell_typed'.
     !Typl Conl ptm.
      Pwell_typed' Typl Conl ptm =
      CASE_FN
      [Is_Const ptm,
       (mem1(Const_nam ptm)Conl) AND
       ((Type_OK Typl(Const_ty ptm)) AND
        ((Type_compat(Const_ty ptm)(corr1(Const_nam ptm)Conl)) AND
         (nocontr(Type_instl(Const_ty ptm)(corr1(Const_nam ptm)Conl)))));
       Is_Var ptm,Type_OK Typl(SND(Var_var ptm));
       Is_App ptm,
       (Pwell_typed' Typl Conl(App_fun ptm)) AND
       ((Pwell_typed' Typl Conl(App_arg ptm)) AND
        ((Is_Tyop(Ptype_of(App_fun ptm))) AND
         ((Tyop_nam(Ptype_of(App_fun ptm)) = `fun`) AND
```

```
          ((LENGTH(Tyop_tyl(Ptype_of(App_fun ptm))) = 2) AND
           (HD(Tyop_tyl(Ptype_of(App_fun ptm))) = Ptype_of(App_arg ptm))))));
       Is_Lam ptm,
       (Pwell_typed' Typl Conl(Lam_bod ptm)) AND
       (Type_OK Typl(SND(Lam_var ptm)))])

Pfree =
|- Pfree =
   (@Pfree'.
     !x ptm.
      Pfree' x ptm =
      CASE_FN
      [Is_Const ptm,F;Is_Var ptm,(Var_var ptm = x);
       Is_App ptm,(Pfree' x(App_fun ptm)) OR (Pfree' x(App_arg ptm));
       Is_Lam ptm,(~(Lam_var ptm = x)) AND (Pfree' x(Lam_bod ptm))])

Pbound =
|- Pbound =
   (@Pbound'.
     !x ptm.
      Pbound' x ptm =
      CASE_FN
      [Is_Const ptm,F;Is_Var ptm,F;
       Is_App ptm,(Pbound' x(App_fun ptm)) OR (Pbound' x(App_arg ptm));
       Is_Lam ptm,(Lam_var ptm = x) OR (Pbound' x(Lam_bod ptm))])

Poccurs =
|- Poccurs =
   (@Poccurs'. !x t. Poccurs' x t = (Pfree x t) OR (Pbound x t))

Plnotfree =
|- Plnotfree =
   (@Plnotfree'.
     !l1 t.
      Plnotfree' l1 t =
      CASE_FN
      [NULL l1,T;IS_CONS l1,(~Pfree(HD l1)t) AND (Plnotfree'(TL l1)t)])

Plnotbound =
|- Plnotbound =
   (@Plnotbound'.
     !l1 t.
      Plnotbound' l1 t =
      CASE_FN
      [NULL l1,T;IS_CONS l1,(~Pbound(HD l1)t) AND (Plnotbound'(TL l1)t)])

Plnotoccurs =
|- Plnotoccurs =
   (@Plnotoccurs'.
     !xl t. Plnotoccurs' xl t = (Plnotfree xl t) AND (Plnotbound xl t))

NotPfree = |- NotPfree = (@NotPfree'. !x t. NotPfree' x t = ~Pfree x t)

Pallnotfree =
|- Pallnotfree =
```

```
   (@Pallnotfree'. !x tms. Pallnotfree' x tms = SEVERY(NotPfree x)tms)

Plallnotfree =
|- Plallnotfree =
   (@Plallnotfree'.
     !xl tms. Plallnotfree' xl tms = SEVERY(Plnotfree xl)tms)

Palreplace1 =
|- Palreplace1 =
   (@Palreplace1'.
     !t' vvl tvl ptm.
      Palreplace1' t' vvl tvl ptm =
      CASE_FN
      [Is_Const ptm,(t' = ptm);
       Is_Var ptm,
       IF_FN
       ((Is_Var t') AND (mem1(Var_var t')vvl),
        (Var_var ptm = corr1(Var_var t')vvl),
        (~mem1(Var_var ptm)vvl) AND
        (IF_FN
         (mem2(Var_var ptm)tvl,(t' = corr2(Var_var ptm)tvl),(t' = ptm))));
       Is_App ptm,
       (Is_App t') AND
       ((Palreplace1'(App_fun t')vvl tvl(App_fun ptm)) AND
        (Palreplace1'(App_arg t')vvl tvl(App_arg ptm)));
       Is_Lam ptm,
       (Is_Lam t') AND
       ((SND(Lam_var t') = SND(Lam_var ptm)) AND
        (Palreplace1'
         (Lam_bod t')
         (CONS(Lam_var t',Lam_var ptm)vvl)
         tvl
         (Lam_bod ptm)))])

Palreplace =
|- Palreplace =
   (@Palreplace'.
     !t' tvl t. Palreplace' t' tvl t = Palreplace1 t'[]tvl t)

Palpha = |- Palpha = (@Palpha'. !t' t. Palpha' t' t = Palreplace t'[]t)

Psubst_triples =
|- Psubst_triples =
   (@Psubst_triples'.
     !l1.
      Psubst_triples' l1 =
      CASE_FN
      [NULL l1,T;
       IS_CONS l1,
       (Ptype_of(FST(HD l1)) = Ptype_of(FST(SND(HD l1)))) AND
       ((Ptype_of(FST(HD l1)) = SND(SND(SND(HD l1)))) AND
        (Psubst_triples'(TL l1)))])

list13 =
|- list13 =
```

```
   (@list13'.
     !l1.
      list13' l1 =
      CASE_FN
      [NULL l1,[];
       IS_CONS l1,CONS(FST(HD l1),SND(SND(HD l1)))(list13'(TL l1))])

Psubst =
|- Psubst =
   (@Psubst'.
     !Typl Conl t' ttvl td t.
      Psubst' Typl Conl t' ttvl td t =
      (Psubst_triples ttvl) AND
      ((Pwell_typed Typl Conl td) AND
       ((Plnotoccurs(MAP SND(MAP SND ttvl))t) AND
        ((Palreplace t(MAP SND ttvl)td) AND
         (Palreplace t'(list13 ttvl)td)))))

Pbeta =
|- Pbeta =
   (@Pbeta'. !t' x t1 t2. Pbeta' t' x t1 t2 = Palreplace t'[t2,x]t1)

Pty_occurs =
|- Pty_occurs =
   (@Pty_occurs'.
     !a ptm.
      Pty_occurs' a ptm =
      CASE_FN
      [Is_Const ptm,Type_occurs a(Const_ty ptm);
       Is_Var ptm,Type_occurs a(SND(Var_var ptm));
       Is_App ptm,
       (Pty_occurs' a(App_fun ptm)) OR (Pty_occurs' a(App_arg ptm));
       Is_Lam ptm,
       (Type_occurs a(SND(Lam_var ptm))) OR (Pty_occurs' a(Lam_bod ptm))])

NotPty_occurs =
|- NotPty_occurs =
   (@NotPty_occurs'. !a t. NotPty_occurs' a t = ~Pty_occurs a t)

Pty_snotoccurs =
|- Pty_snotoccurs =
   (@Pty_snotoccurs'.
     !a tms. Pty_snotoccurs' a tms = SEVERY(NotPty_occurs a)tms)

Plty_snotoccurs =
|- Plty_snotoccurs =
   (@Plty_snotoccurs'.
     !l1 tml.
      Plty_snotoccurs' l1 tml =
      CASE_FN
      [NULL l1,T;
       IS_CONS l1,
       (Pty_snotoccurs(HD l1)tml) AND (Plty_snotoccurs'(TL l1)tml)])

Pnewfree1 =
```

```
|- Pnewfree1 =
   (@Pnewfree1'.
     !t bl ptm.
      Pnewfree1' t bl ptm =
      CASE_FN
      [Is_Const ptm,[];
       Is_Var ptm,
       IF_FN
       ((mem(Var_var ptm)bl) OR (FST(Var_var t) = FST(Var_var ptm)),[],
        [Var_var t,Var_var ptm]);
       Is_App ptm,
       APPEND
       (Pnewfree1'(App_fun t)bl(App_fun ptm))
       (Pnewfree1'(App_arg t)bl(App_arg ptm));
       Is_Lam ptm,
       Pnewfree1'(Lam_bod t)(CONS(Lam_var ptm)bl)(Lam_bod ptm)])

Pnewfree =
|- Pnewfree = (@Pnewfree'. !t' t. Pnewfree' t' t = Pnewfree1 t'[]t)

Ptyinst1 =
|- Ptyinst1 =
   (@Ptyinst1'.
     !t bvl fvl tyl ptm.
      Ptyinst1' t bvl fvl tyl ptm =
      CASE_FN
      [Is_Const ptm,
       (t = Const(Const_nam ptm)(Type_replace tyl(Const_ty ptm)));
       Is_Var ptm,
       IF_FN
       ((Is_Var t) AND (mem1(Var_var t)bvl),
        (Var_var ptm = corr1(Var_var t)bvl) AND
        (SND(Var_var t) = Type_replace tyl(SND(Var_var ptm))),
        IF_FN
        (mem2(Var_var ptm)fvl,
         (t =
          Var
          (FST(corr2(Var_var ptm)fvl),Type_replace tyl(SND(Var_var ptm)))),
         (t = Var(FST(Var_var ptm),Type_replace tyl(SND(Var_var ptm))))));
       Is_App ptm,
       (Is_App t) AND
       ((Ptyinst1'(App_fun t)bvl fvl tyl(App_fun ptm)) AND
        (Ptyinst1'(App_arg t)bvl fvl tyl(App_arg ptm)));
       Is_Lam ptm,
       (Is_Lam t) AND
       ((~mem1(Lam_var t)fvl) AND
        ((SND(Lam_var t) = Type_replace tyl(SND(Lam_var ptm))) AND
         (Ptyinst1'
          (Lam_bod t)
          (CONS(Lam_var t,Lam_var ptm)bvl)
          fvl
          tyl
          (Lam_bod ptm))))])

Ptyinst =
```

```
|- Ptyinst =
   (@Ptyinst'.
     !ass t' tyl t.
      Ptyinst' ass t' tyl t =
      (Ptyinst1 t'[](Pnewfree t' t)tyl t) AND
      (Plallnotfree(MAP SND(Pnewfree t' t))ass))
```

The final set of definitions are generated by the file `inf.al` and include the type definition axioms for the types representing sequents and inferences.

```
|- !f. ?! fn. !s P. fn(Pseq s P) = f s P

Pseq_boolean =
|- Pseq_boolean =
   (@Pseq_boolean'.
     !pseq.
      Pseq_boolean' pseq =
      (SEVERY Pboolean(Pseq_assum pseq)) AND (Pboolean(Pseq_concl pseq)))

Pseq_well_typed =
|- Pseq_well_typed =
   (@Pseq_well_typed'.
     !Typl Conl pseq.
      Pseq_well_typed' Typl Conl pseq =
      (SEVERY(Pwell_typed Typl Conl)(Pseq_assum pseq)) AND
      (Pwell_typed Typl Conl(Pseq_concl pseq)))

PEQ =
|- PEQ =
   (@PEQ'.
     !t1 t2.
      PEQ' t1 t2 =
      App
      (App
       (Const
        `=`
        (Tyop `fun`[Ptype_of t1;Tyop `fun`[Ptype_of t1;Tyop `bool`[]]]))
       t1)
      t2)

PIMP =
|- PIMP =
   (@PIMP'.
     !t1 t2.
      PIMP' t1 t2 =
      App
      (App
       (Const
        `==>`
        (Tyop
         `fun`
         [Tyop `bool`[];Tyop `fun`[Tyop `bool`[];Tyop `bool`[]]]))
       t1)
```

```
      t2)

Is_EQtm =
|- Is_EQtm =
   (@Is_EQtm'.
     !t.
      Is_EQtm' t =
      (Is_App t) AND
      ((Is_App(App_fun t)) AND
       (App_fun(App_fun t) =
        Const
        `=`
        (Tyop
         `fun`
         [Ptype_of(App_arg t);
          Tyop `fun`[Ptype_of(App_arg t);Tyop `bool`[]]]))))

PASSUME =
|- PASSUME =
   (@PASSUME'.
     !Typl Conl pseq tm.
      PASSUME' Typl Conl pseq tm =
      (Pwell_typed Typl Conl tm) AND
      ((Pboolean tm) AND
       ((Pseq_concl pseq = tm) AND (Pseq_assum pseq = SetOf[tm]))))

PREFL =
|- PREFL =
   (@PREFL'.
     !Typl Conl pseq tm.
      PREFL' Typl Conl pseq tm =
      (Pwell_typed Typl Conl tm) AND
      ((Pseq_assum pseq = {}) AND (Pseq_concl pseq = PEQ tm tm)))

PBETA_CONV =
|- PBETA_CONV =
   (@PBETA_CONV'.
     !Typl Conl pseq tm.
      PBETA_CONV' Typl Conl pseq tm =
      (Pwell_typed Typl Conl tm) AND
      ((Pseq_assum pseq = {}) AND
       ((Is_App tm) AND
        ((Is_Lam(App_fun tm)) AND
         ((Pseq_concl pseq = PEQ tm(App_arg(Pseq_concl pseq))) AND
          (Pbeta
           (App_arg(Pseq_concl pseq))
           (Lam_var(App_fun tm))
           (Lam_bod(App_fun tm))
           (App_arg tm))))))

Psubst_newlist =
|- Psubst_newlist =
   (@Psubst_newlist'.
     !l1.
      Psubst_newlist' l1 =
```

```
      CASE_FN
      [NULL l1,[];
       IS_CONS l1,
       CONS
       (App_arg(Pseq_concl(FST(HD l1))),
        App_arg(App_fun(Pseq_concl(FST(HD l1)))),SND(HD l1))
       (Psubst_newlist'(TL l1))])

PSUBST =
|- PSUBST =
   (@PSUBST'.
     !Typl Conl pseq thdl td th.
      PSUBST' Typl Conl pseq thdl td th =
      (Pwell_typed Typl Conl td) AND
      ((EVERY Is_EQtm(MAP Pseq_concl(MAP FST thdl))) AND
       ((Psubst
         Typl
         Conl
         (Pseq_concl pseq)
         (Psubst_newlist thdl)
         td
         (Pseq_concl th)) AND
        (Pseq_assum pseq =
         (Pseq_assum th) UNION (LUNION(MAP Pseq_assum(MAP FST thdl)))))))

PABS =
|- PABS =
   (@PABS'.
     !Typl Conl pseq tm th.
      PABS' Typl Conl pseq tm th =
      (Pwell_typed Typl Conl tm) AND
      ((Is_EQtm(Pseq_concl th)) AND
       ((Is_Var tm) AND
        ((Type_OK Typl(SND(Var_var tm))) AND
         ((Pseq_concl pseq =
           PEQ
           (Lam(Var_var tm)(App_arg(App_fun(Pseq_concl th))))
           (Lam(Var_var tm)(App_arg(Pseq_concl th)))) AND
          ((Pseq_assum pseq = Pseq_assum th) AND
           (Pallnotfree(Var_var tm)(Pseq_assum pseq)))))))

PINST_TYPE =
|- PINST_TYPE =
   (@PINST_TYPE'.
     !Typl pseq tyl th.
      PINST_TYPE' Typl pseq tyl th =
      (EVERY(Type_OK Typl)(MAP FST tyl)) AND
      ((Ptyinst(Pseq_assum pseq)(Pseq_concl pseq)tyl(Pseq_concl th)) AND
       ((Plty_snotoccurs(MAP SND tyl)(Pseq_assum pseq)) AND
        (Pseq_assum pseq = Pseq_assum th))))

PDISCH =
|- PDISCH =
   (@PDISCH'.
     !Typl Conl pseq tm th.
```

```
      PDISCH' Typl Conl pseq tm th =
      (Pwell_typed Typl Conl tm) AND
      ((Pboolean tm) AND
       ((Pseq_concl pseq = PIMP tm(Pseq_concl th)) AND
        (Pseq_assum pseq = (Pseq_assum th) DELETE tm))))

PMP =
|- PMP =
   (@PMP'.
     !pseq th1 th2.
      PMP' pseq th1 th2 =
      (Pseq_concl th1 = PIMP(Pseq_concl th2)(Pseq_concl pseq)) AND
      (Pseq_assum pseq = (Pseq_assum th1) UNION (Pseq_assum th2)))

|- !f0 f1 f2 f3 f4 f5 f6 f7 f8.
    ?! fn.
     (!P. fn(AX_inf P) = f0 P) /\
     (!P0 P1. fn(AS_inf P0 P1) = f1 P0 P1) /\
     (!P0 P1. fn(RE_inf P0 P1) = f2 P0 P1) /\
     (!P0 P1. fn(BE_inf P0 P1) = f3 P0 P1) /\
     (!P0 l P1 P2. fn(SU_inf P0 l P1 P2) = f4 P0 l P1 P2) /\
     (!P0 P1 P2. fn(AB_inf P0 P1 P2) = f5 P0 P1 P2) /\
     (!P0 l P1. fn(IN_inf P0 l P1) = f6 P0 l P1) /\
     (!P0 P1 P2. fn(DI_inf P0 P1 P2) = f7 P0 P1 P2) /\
     (!P0 P1 P2. fn(MP_inf P0 P1 P2) = f8 P0 P1 P2)

Inf_concl =
|- Inf_concl =
   (@Inf_concl'.
     !inf.
      Inf_concl' inf =
      CASE_FN
      [is_AX inf,AX_con inf;is_AS inf,AS_con inf;is_RE inf,RE_con inf;
       is_BE inf,BE_con inf;is_SU inf,SU_con inf;is_AB inf,AB_con inf;
       is_IN inf,IN_con inf;is_DI inf,DI_con inf;is_MP inf,MP_con inf])

Inf_hyps =
|- Inf_hyps =
   (@Inf_hyps'.
     !inf.
      Inf_hyps' inf =
      CASE_FN
      [is_AX inf,[];is_AS inf,[];is_RE inf,[];is_BE inf,[];
       is_SU inf,CONS(SU_thm inf)(MAP FST(SU_eqthms inf));
       is_AB inf,[AB_eq inf];is_IN inf,[IN_thm inf];
       is_DI inf,[DI_thm inf];is_MP inf,[MP_ithm inf;MP_thm inf]])

OK_Inf =
|- OK_Inf =
   (@OK_Inf'.
     !Typl Conl Axil inf.
      OK_Inf' Typl Conl Axil inf =
      CASE_FN
      [is_AX inf,mem(AX_con inf)Axil;
       is_AS inf,PASSUME Typl Conl(AS_con inf)(AS_tm inf);
```

```
       is_RE inf,PREFL Typl Conl(RE_con inf)(RE_tm inf);
       is_BE inf,PBETA_CONV Typl Conl(BE_con inf)(BE_tm inf);
       is_SU inf,
       PSUBST
       Typl
       Conl
       (SU_con inf)
       (SU_eqthms inf)
       (SU_tmdum inf)
       (SU_thm inf);
       is_AB inf,PABS Typl Conl(AB_con inf)(AB_tm inf)(AB_eq inf);
       is_IN inf,PINST_TYPE Typl(IN_con inf)(IN_tyl inf)(IN_thm inf);
       is_DI inf,PDISCH Typl Conl(DI_con inf)(DI_tm inf)(DI_thm inf);
       is_MP inf,PMP(MP_con inf)(MP_ithm inf)(MP_thm inf)])

Is_proof =
|- Is_proof =
   (@Is_proof'.
     !Typl Conl Axil l.
      Is_proof' Typl Conl Axil l =
      CASE_FN
      [NULL l,T;
       IS_CONS l,
       (OK_Inf Typl Conl Axil(HD l)) AND
       ((lmem(Inf_hyps(HD l))(MAP Inf_concl(TL l))) AND
        (Is_proof' Typl Conl Axil(TL l)))])
```

Appendix C

HOL semantics of the AL specification after automatic simplification

The results of simplification are given below grouped into the same sets as the earlier specifications. For clarity, the functions defined for the accessors and recognisors are not included.

```
EVERY2_DEF
  |- (!Pred l2. EVERY2 Pred[]l2 = T) /\
     (!Pred h t l2.
       EVERY2 Pred(CONS h t)l2 = Pred h(HD l2) /\ EVERY2 Pred t(TL l2))
LAPPEND_DEF
  |- (LAPPEND[] = []) /\
     (!h t. LAPPEND(CONS h t) = APPEND h(LAPPEND t))
LUNION_DEF
  |- (LUNION[] = {}) /\ (!h t. LUNION(CONS h t) = h UNION (LUNION t))
lor_DEF  |- (lor[] = F) /\ (!h t. lor(CONS h t) = h \/ lor t)
land_DEF  |- (land[] = T) /\ (!h t. land(CONS h t) = h /\ land t)
corr1_DEF
  |- (!x. corr1 x[] = ARB) /\
     (!x h t. corr1 x(CONS h t) = ((x = FST h) => SND h | corr1 x t))
corr2_DEF
  |- (!x. corr2 x[] = ARB) /\
     (!x h t. corr2 x(CONS h t) = ((x = SND h) => FST h | corr2 x t))
mem_DEF
  |- (!x. mem x[] = F) /\
     (!x h t. mem x(CONS h t) = (x = h) \/ mem x t)
lmem_DEF
  |- (!l2. lmem[]l2 = T) /\
     (!h t l2. lmem(CONS h t)l2 = mem h l2 /\ lmem t l2)
```

```
mem1_DEF
  |- (!x. mem1 x[] = F) /\
     (!x h t. mem1 x(CONS h t) = (x = FST h) \/ mem1 x t)
mem2_DEF
  |- (!x. mem2 x[] = F) /\
     (!x h t. mem2 x(CONS h t) = (x = SND h) \/ mem2 x t)
nocontr_DEF
  |- (nocontr[] = T) /\
     (!h t.
       nocontr(CONS h t) =
       (~mem2(SND h)t \/ (corr2(SND h)t = FST h)) /\ nocontr t)
```

The second group corresponding to the original `type.al` is:

```
XMAP2_DEF
  |- (!afn l2. XMAP2 afn[]l2 = []) /\
     (!afn h t l2.
       XMAP2 afn(CONS h t)l2 = CONS(afn h(HD l2))(XMAP2 afn t(TL l2)))
Type_OK_DEF
  |- (!tyl s. Type_OK tyl(Tyvar s) = T) /\
     (!tyl s Tlst.
       Type_OK tyl(Tyop s Tlst) =
       mem1 s tyl /\
       (LENGTH Tlst = corr1 s tyl) /\
       EVERY(Type_OK tyl)Tlst)
Type_compat_DEF
  |- (!ty1 s. Type_compat ty1(Tyvar s) = T) /\
     (!ty1 s Tlst.
       Type_compat ty1(Tyop s Tlst) =
       Is_Tyop ty1 /\
       (Tyop_nam ty1 = s) /\
       (LENGTH(Tyop_tyl ty1) = LENGTH Tlst) /\
       EVERY2 Type_compat(Tyop_tyl ty1)Tlst)
Type_occurs_DEF
  |- (!str s. Type_occurs str(Tyvar s) = (str = s)) /\
     (!str s Tlst.
       Type_occurs str(Tyop s Tlst) = lor(MAP(Type_occurs str)Tlst))
Type_replace_DEF
  |- (!l s.
       Type_replace l(Tyvar s) = (mem2 s l => corr2 s l | Tyvar s)) /\
     (!l s Tlst.
       Type_replace l(Tyop s Tlst) = Tyop s(MAP(Type_replace l)Tlst))
Type_instlx_DEF  |- !ty1 ty2. Type_instlx ty2 ty1 = Type_instl ty1 ty2
Type_instl_DEF
  |- (!s ty1. Type_instl ty1(Tyvar s) = [ty1,s]) /\
     (!s Tlst ty1.
       Type_instl ty1(Tyop s Tlst) =
       LAPPEND(XMAP2 Type_instlx Tlst(Tyop_tyl ty1)))
```

The following are the simplified HOL definitions generated from `term.ml`.

```
Ptype_of_DEF
  |- (!s T'. Ptype_of(Const s T') = T') /\
     (!p. Ptype_of(Var p) = SND p) /\
     (!P1 P2. Ptype_of(App P1 P2) = HD(TL(Tyop_tyl(Ptype_of P1)))) /\
```

```
     (!p P. Ptype_of(Lam p P) = Tyop `fun`[SND p;Ptype_of P])
Pboolean_DEF  |- !t. Pboolean t = (Ptype_of t = Tyop `bool`[])
Pwell_typed_DEF
  |- (!Typl Conl s T'.
       Pwell_typed Typl Conl(Const s T') =
       mem1 s Conl /\
       Type_OK Typl T' /\
       Type_compat T'(corr1 s Conl) /\
       nocontr(Type_instl T'(corr1 s Conl))) /\
     (!Typl Conl p.
       Pwell_typed Typl Conl(Var p) = Type_OK Typl(SND p)) /\
     (!Typl Conl P1 P2.
       Pwell_typed Typl Conl(App P1 P2) =
       Pwell_typed Typl Conl P1 /\
       Pwell_typed Typl Conl P2 /\
       Is_Tyop(Ptype_of P1) /\
       (Tyop_nam(Ptype_of P1) = `fun`) /\
       (LENGTH(Tyop_tyl(Ptype_of P1)) = 2) /\
       (HD(Tyop_tyl(Ptype_of P1)) = Ptype_of P2)) /\
     (!Typl Conl p P.
       Pwell_typed Typl Conl(Lam p P) =
       Pwell_typed Typl Conl P /\ Type_OK Typl(SND p))
Pfree_DEF
  |- (!x s T'. Pfree x(Const s T') = F) /\
     (!x p. Pfree x(Var p) = (p = x)) /\
     (!x P1 P2. Pfree x(App P1 P2) = Pfree x P1 \/ Pfree x P2) /\
     (!x p P. Pfree x(Lam p P) = ~(p = x) /\ Pfree x P)
Pbound_DEF
  |- (!x s T'. Pbound x(Const s T') = F) /\
     (!x p. Pbound x(Var p) = F) /\
     (!x P1 P2. Pbound x(App P1 P2) = Pbound x P1 \/ Pbound x P2) /\
     (!x p P. Pbound x(Lam p P) = (p = x) \/ Pbound x P)
Poccurs_DEF  |- !x t. Poccurs x t = Pfree x t \/ Pbound x t
Plnotfree_DEF
  |- (!t. Plnotfree[]t = T) /\
     (!h t' t. Plnotfree(CONS h t')t = ~Pfree h t /\ Plnotfree t' t)
Plnotbound_DEF
  |- (!t. Plnotbound[]t = T) /\
     (!h t' t.
       Plnotbound(CONS h t')t = ~Pbound h t /\ Plnotbound t' t)
Plnotoccurs_DEF
  |- !xl t. Plnotoccurs xl t = Plnotfree xl t /\ Plnotbound xl t
NotPfree_DEF  |- !x t. NotPfree x t = ~Pfree x t
Pallnotfree_DEF  |- !x. Pallnotfree x = SEVERY(NotPfree x)
Plallnotfree_DEF  |- !xl. Plallnotfree xl = SEVERY(Plnotfree xl)
Palreplace1_DEF
  |- (!t' vvl tvl s T'.
       Palreplace1 t' vvl tvl(Const s T') = (t' = Const s T')) /\
     (!t' vvl tvl p.
       Palreplace1 t' vvl tvl(Var p) =
       ((Is_Var t' /\ mem1(Var_var t')vvl) =>
        (p = corr1(Var_var t')vvl) |
        (~mem1 p vvl /\
         (mem2 p tvl => (t' = corr2 p tvl) | (t' = Var p))))) /\
     (!t' vvl tvl P1 P2.
```

```
       Palreplace1 t' vvl tvl(App P1 P2) =
       Is_App t' /\
       Palreplace1(App_fun t')vvl tvl P1 /\
       Palreplace1(App_arg t')vvl tvl P2) /\
      (!t' vvl tvl p P.
       Palreplace1 t' vvl tvl(Lam p P) =
       Is_Lam t' /\
       (SND(Lam_var t') = SND p) /\
       Palreplace1(Lam_bod t')(CONS(Lam_var t',p)vvl)tvl P)
Palreplace_DEF  |- !t'. Palreplace t' = Palreplace1 t'[]
Palpha_DEF  |- !t'. Palpha t' = Palreplace t'[]
Psubst_triples_DEF
  |- (Psubst_triples[] = T) /\
      (!h t.
       Psubst_triples(CONS h t) =
       (Ptype_of(FST h) = Ptype_of(FST(SND h))) /\
       (Ptype_of(FST h) = SND(SND(SND h))) /\
       Psubst_triples t)
list13_DEF
  |- (list13[] = []) /\
      (!h t. list13(CONS h t) = CONS(FST h,SND(SND h))(list13 t))
Psubst_DEF
  |- !Typl Conl t' ttvl td t.
      Psubst Typl Conl t' ttvl td t =
      Psubst_triples ttvl /\
      Pwell_typed Typl Conl td /\
      Plnotoccurs(MAP SND(MAP SND ttvl))t /\
      Palreplace t(MAP SND ttvl)td /\
      Palreplace t'(list13 ttvl)td
Pbeta_DEF  |- !t' x t1 t2. Pbeta t' x t1 t2 = Palreplace t'[t2,x]t1


Pty_occurs_DEF
  |- (!a s T'. Pty_occurs a(Const s T') = Type_occurs a T') /\
      (!a p. Pty_occurs a(Var p) = Type_occurs a(SND p)) /\
      (!a P1 P2.
       Pty_occurs a(App P1 P2) = Pty_occurs a P1 \/ Pty_occurs a P2) /\
      (!a p P.
       Pty_occurs a(Lam p P) = Type_occurs a(SND p) \/ Pty_occurs a P)
NotPty_occurs_DEF  |- !a t. NotPty_occurs a t = ~Pty_occurs a t
Pty_snotoccurs_DEF  |- !a. Pty_snotoccurs a = SEVERY(NotPty_occurs a)
Plty_snotoccurs_DEF
  |- (!tml. Plty_snotoccurs[]tml = T) /\
      (!h t tml.
       Plty_snotoccurs(CONS h t)tml =
       Pty_snotoccurs h tml /\ Plty_snotoccurs t tml)
Pnewfree1_DEF
  |- (!t bl s T'. Pnewfree1 t bl(Const s T') = []) /\
      (!t bl p.
       Pnewfree1 t bl(Var p) =
       ((mem p bl \/ (FST(Var_var t) = FST p)) => [] | [Var_var t,p])) /\
      (!t bl P1 P2.
       Pnewfree1 t bl(App P1 P2) =
       APPEND(Pnewfree1(App_fun t)bl P1)(Pnewfree1(App_arg t)bl P2)) /\
      (!t bl p P.
```

```
       Pnewfree1 t bl(Lam p P) = Pnewfree1(Lam_bod t)(CONS p bl)P)
Pnewfree_DEF  |- !t'. Pnewfree t' = Pnewfree1 t'[]
Ptyinst1_DEF
  |- (!t bvl fvl tyl s T'.
       Ptyinst1 t bvl fvl tyl(Const s T') =
       (t = Const s(Type_replace tyl T'))) /\
     (!t bvl fvl tyl p.
       Ptyinst1 t bvl fvl tyl(Var p) =
       ((Is_Var t /\ mem1(Var_var t)bvl) =>
        ((p = corr1(Var_var t)bvl) /\
         (SND(Var_var t) = Type_replace tyl(SND p))) |
        (mem2 p fvl =>
         (t = Var(FST(corr2 p fvl),Type_replace tyl(SND p))) |
         (t = Var(FST p,Type_replace tyl(SND p)))))) /\
     (!t bvl fvl tyl P1 P2.
       Ptyinst1 t bvl fvl tyl(App P1 P2) =
       Is_App t /\
       Ptyinst1(App_fun t)bvl fvl tyl P1 /\
       Ptyinst1(App_arg t)bvl fvl tyl P2) /\
     (!t bvl fvl tyl p P.
       Ptyinst1 t bvl fvl tyl(Lam p P) =
       Is_Lam t /\
       ~mem1(Lam_var t)fvl /\
       (SND(Lam_var t) = Type_replace tyl(SND p)) /\
       Ptyinst1(Lam_bod t)(CONS(Lam_var t,p)bvl)fvl tyl P)
Ptyinst_DEF
  |- !ass t' tyl t.
      Ptyinst ass t' tyl t =
      Ptyinst1 t'[](Pnewfree t' t)tyl t /\
      Plallnotfree(MAP SND(Pnewfree t' t))ass
```

The simplified definitions corrresponding to the file `inf.al` are as follows:

```
Pseq_boolean_DEF
  |- !as t. Pseq_boolean(Pseq as t) = SEVERY Pboolean as /\ Pboolean t

Pseq_well_typed_DEF
  |- !Typl Conl as t.
      Pseq_well_typed Typl Conl(Pseq as t) =
      SEVERY(Pwell_typed Typl Conl)as /\ Pwell_typed Typl Conl t

PEQ_DEF
  |- !t1.
      PEQ t1 =
      App
      (App
       (Const
        `=`
        (Tyop
         `fun`
         [Ptype_of t1;Tyop `fun`[Ptype_of t1;Tyop `bool`[]]]))
       t1)
PIMP_DEF
  |- !t1.
      PIMP t1 =
```

```
      App
      (App
       (Const
        `==>`
        (Tyop
         `fun`
         [Tyop `bool`[];Tyop `fun`[Tyop `bool`[];Tyop `bool`[]]]))
       t1)
Is_EQtm_DEF
  |- !t.
      Is_EQtm t =
      Is_App t /\
      Is_App(App_fun t) /\
      (App_fun(App_fun t) =
       Const
       `=`
       (Tyop
        `fun`
        [Ptype_of(App_arg t);
         Tyop `fun`[Ptype_of(App_arg t);Tyop `bool`[]]]))
PASSUME_DEF
  |- !Typl Conl as t tm.
      PASSUME Typl Conl(Pseq as t)tm =
      Pwell_typed Typl Conl tm /\
      Pboolean tm /\
      (t = tm) /\
      (as = {tm})
PREFL_DEF
  |- !Typl Conl as t tm.
      PREFL Typl Conl(Pseq as t)tm =
      Pwell_typed Typl Conl tm /\ (as = {}) /\ (t = PEQ tm tm)
PBETA_CONV_DEF
  |- !Typl Conl as t tm.
      PBETA_CONV Typl Conl(Pseq as t)tm =
      Pwell_typed Typl Conl tm /\
      (as = {}) /\
      Is_App tm /\
      Is_Lam(App_fun tm) /\
      (t = PEQ tm(App_arg t)) /\
      Pbeta
      (App_arg t)
      (Lam_var(App_fun tm))
      (Lam_bod(App_fun tm))
      (App_arg tm)
Psubst_newlist_DEF
  |- (Psubst_newlist[] = []) /\
     (!h t.
       Psubst_newlist(CONS h t) =
       CONS
       (App_arg(Pseq_concl(FST h)),
        App_arg(App_fun(Pseq_concl(FST h))),SND h)
       (Psubst_newlist t))
PSUBST_DEF
  |- !Typl Conl as t thdl td th.
      PSUBST Typl Conl(Pseq as t)thdl td th =
```

```
      Pwell_typed Typl Conl td /\
      EVERY Is_EQtm(MAP Pseq_concl(MAP FST thdl)) /\
      Psubst Typl Conl t(Psubst_newlist thdl)td(Pseq_concl th) /\
      (as =
       (Pseq_assum th) UNION (LUNION(MAP Pseq_assum(MAP FST thdl))))
PABS_DEF
  |- !Typl Conl as t tm th.
      PABS Typl Conl(Pseq as t)tm th =
      Pwell_typed Typl Conl tm /\
      Is_EQtm(Pseq_concl th) /\
      Is_Var tm /\
      Type_OK Typl(SND(Var_var tm)) /\
      (t =
       PEQ
       (Lam(Var_var tm)(App_arg(App_fun(Pseq_concl th))))
       (Lam(Var_var tm)(App_arg(Pseq_concl th)))) /\
      (as = Pseq_assum th) /\
      Pallnotfree(Var_var tm)as
PINST_TYPE_DEF
  |- !Typl as t tyl th.
      PINST_TYPE Typl(Pseq as t)tyl th =
      EVERY(Type_OK Typl)(MAP FST tyl) /\
      Ptyinst as t tyl(Pseq_concl th) /\
      Plty_snotoccurs(MAP SND tyl)as /\
      (as = Pseq_assum th)
PDISCH_DEF
  |- !Typl Conl as t tm th.
      PDISCH Typl Conl(Pseq as t)tm th =
      Pwell_typed Typl Conl tm /\
      Pboolean tm /\
      (t = PIMP tm(Pseq_concl th)) /\
      (as = (Pseq_assum th) DELETE tm)
PMP_DEF
  |- !as t th1 th2.
      PMP(Pseq as t)th1 th2 =
      (Pseq_concl th1 = PIMP(Pseq_concl th2)t) /\
      (as = (Pseq_assum th1) UNION (Pseq_assum th2))


Inf_concl_DEF
  |- (!s. Inf_concl(AX_inf s) = s) /\
     (!s t. Inf_concl(AS_inf s t) = s) /\
     (!s t. Inf_concl(RE_inf s t) = s) /\
     (!s t. Inf_concl(BE_inf s t) = s) /\
     (!s tdl t s1. Inf_concl(SU_inf s tdl t s1) = s) /\
     (!s t s1. Inf_concl(AB_inf s t s1) = s) /\
     (!s tyl s1. Inf_concl(IN_inf s tyl s1) = s) /\
     (!s t s1. Inf_concl(DI_inf s t s1) = s) /\
     (!s s1 s2. Inf_concl(MP_inf s s1 s2) = s)
Inf_hyps_DEF
  |- (!s. Inf_hyps(AX_inf s) = []) /\
     (!s t. Inf_hyps(AS_inf s t) = []) /\
     (!s t. Inf_hyps(RE_inf s t) = []) /\
     (!s t. Inf_hyps(BE_inf s t) = []) /\
     (!s sdl t s1. Inf_hyps(SU_inf s sdl t s1) = CONS s1(MAP FST sdl)) /\
```

```
     (!s t s1. Inf_hyps(AB_inf s t s1) = [s1]) /\
     (!s tyl s1. Inf_hyps(IN_inf s tyl s1) = [s1]) /\
     (!s t s1. Inf_hyps(DI_inf s t s1) = [s1]) /\
     (!s s1 s2. Inf_hyps(MP_inf s s1 s2) = [s1;s2])
OK_Inf_DEF
  |- (!Typl Conl Axil s. OK_Inf Typl Conl Axil(AX_inf s) = mem s Axil) /\
     (!Typl Conl Axil s t.
       OK_Inf Typl Conl Axil(AS_inf s t) = PASSUME Typl Conl s t) /\
     (!Typl Conl Axil s t.
       OK_Inf Typl Conl Axil(RE_inf s t) = PREFL Typl Conl s t) /\
     (!Typl Conl Axil s t.
       OK_Inf Typl Conl Axil(BE_inf s t) = PBETA_CONV Typl Conl s t) /\
     (!Typl Conl Axil s tdl t s1.
       OK_Inf Typl Conl Axil(SU_inf s tdl t s1) =
       PSUBST Typl Conl s tdl t s1) /\
     (!Typl Conl Axil s t s1.
       OK_Inf Typl Conl Axil(AB_inf s t s1) = PABS Typl Conl s t s1) /\
     (!Typl Conl Axil s tyl s1.
       OK_Inf Typl Conl Axil(IN_inf s tyl s1) =
       PINST_TYPE Typl s tyl s1) /\
     (!Typl Conl Axil s t s1.
       OK_Inf Typl Conl Axil(DI_inf s t s1) = PDISCH Typl Conl s t s1) /\
     (!Typl Conl Axil s s1 s2.
       OK_Inf Typl Conl Axil(MP_inf s s1 s2) = PMP s s1 s2)
Is_proof_DEF
  |- (!Typl Conl Axil. Is_proof Typl Conl Axil[] = T) /\
     (!Typl Conl Axil h t.
       Is_proof Typl Conl Axil(CONS h t) =
       OK_Inf Typl Conl Axil h /\
       lmem(Inf_hyps h)(MAP Inf_concl t) /\
       Is_proof Typl Conl Axil t)
```

Appendix D

SML generated from AL specification

Before the AL is translated into HOL a number of definitions for compatability are made. Some of these might seem to be unnecessary, such as definitions of sets for example since packages implementing sets are available. However, the approach being followed is to make everything as simple and explicit as possible.

```
(* standard prelude before loading AL generated SML *)

fun fst (a,b) = a;
fun snd (a,b) = b;

(* boolean *)

infix AND;
fun a AND b = a andalso b;

infix OR;
fun a OR b = a orelse b;

val NOT = not;

(* strings *)

fun concat a b = a ^ b;

(* lists *)

fun mem a []  = false  |
    mem a (b::l) =  (a=b) orelse (mem a l);

fun EVERY pr [] = true |
    EVERY pr (a::lst) =
```

```
     (pr a) andalso EVERY pr lst;;

val LENGTH = length;
val MAP = map;

fun cons a b = a::b;

fun APPEND a b = a @ b;

val is_cons = not o null;
val is_null = null;

val car = hd;
val cdr = tl;

fun filter pr [] = [] |
    filter pr (a::lst) =
     if (pr a) then (a::(filter pr lst)) else filter pr lst;

fun remove elem [] = [] |
    remove elem (a::lst) =
     if (a=elem) then (remove elem lst) else (a::(remove elem lst));

(* sets *)

fun Setify [] = []  |
    Setify (a::b) = if (mem a b) then Setify b else a::(Setify b);

datatype 'a set = SetOf of ('a list);

val empty = SetOf [];

fun is_empty s = (s = SetOf []);;
fun is_insert s = not(s = SetOf []);;

fun insert elem (SetOf lst) =
    if (mem elem lst) then (SetOf lst) else SetOf (elem::lst);

infix UNION;
fun (SetOf l1) UNION (SetOf l2) =  SetOf(Setify (l1@l2));

exception NotInSet;

infix DELETE;
fun (SetOf lst) DELETE elem =
    if (mem elem lst) then SetOf (remove elem lst) else SetOf lst;

fun IS_IN elem (SetOf lst) = mem elem lst;

infix SubSet;
fun (SetOf [])     SubSet s2 =  true  |
    (SetOf (a::l)) SubSet s2 =  (IS_IN a s2) andalso (SetOf l) SubSet s2;

infix SetEq;
fun s1 SetEq s2 =  (s1 SubSet s2) andalso (s2 SubSet s1);
```

```
fun SEVERY pr (SetOf lst) = EVERY pr lst;

(* EXTRA *)

fun CASE_fun ((a,b)::rst) = if a then b () else CASE_fun rst;

fun tup1_1(a) = a;
fun tup2_1(a,_) = a;
fun tup2_2(_,a) = a;
fun tup3_1(a,_,_) = a;
fun tup3_2(_,a,_) = a;
fun tup3_3(_,_,a) = a;

fun tup4_1 (a,_,_,_) = a;
fun tup4_2 (_,a,_,_) = a;
fun tup4_3 (_,_,a,_) = a;
fun tup4_4 (_,_,_,a) = a;

exception ARB;
exception ACCESS_ex;

type num = int;
```

The rest of the SML text is generated automatically from the AL specification. It is presented in groups corresponding to the break-up of the AL into different files which in turn reflects the structuring of the original theory of HOL proofs on which it is based. First, a set of auxiliary functions are defined.

```
fun  EVERY2 (Pred:''a->''b->bool) (l1:''a list) (l2:''b list) :bool =
CASE_fun [
(is_null l1, fn zz => true),
(is_cons l1, fn zz =>
 (Pred (car l1) (car l2)) AND (EVERY2 Pred (cdr l1) (cdr l2)) )
]
; (* end fun dec *)

fun  LAPPEND (l1:(''a list)list) :''a list =
CASE_fun [
(is_null l1, fn zz => []),
(is_cons l1, fn zz => APPEND (car l1) (LAPPEND (cdr l1)) )
]
; (* end fun dec *)

fun  LUNION (l1:((''a)set) list) :(''a)set =
CASE_fun [
(is_null l1, fn zz => empty),
(is_cons l1, fn zz => (car l1) UNION (LUNION (cdr l1)) )
]
; (* end fun dec *)

fun  lor (l1:bool list) :bool =
```

```
CASE_fun [
(is_null l1, fn zz => false),
(is_cons l1, fn zz => (car l1) OR (lor (cdr l1)) )
]
; (* end fun dec *)


fun  land (l1:bool list) :bool =
CASE_fun [
(is_null l1, fn zz => true),
(is_cons l1, fn zz => (car l1) AND (land (cdr l1)) )
]
; (* end fun dec *)

fun  corr1 (x:''a) (l1:(''a * ''b) list) :''b =
CASE_fun [
(is_null l1, fn zz => raise ARB),
(is_cons l1, fn zz =>
 (if (x = fst(car l1))
 then snd(car l1)
 else corr1 x (cdr l1)
 ))
]
; (* end fun dec *)

fun  corr2 (x:''b) (l1:(''a * ''b) list) :''a =
CASE_fun [
(is_null l1, fn zz => raise ARB),
(is_cons l1, fn zz =>
 (if (x = snd(car l1))
 then fst(car l1)
 else corr2 x (cdr l1)
 ))
]
; (* end fun dec *)

fun  mem (x:''a) (l1:''a list) :bool =
CASE_fun [
(is_null l1, fn zz => false),
(is_cons l1, fn zz => (x = (car l1)) OR (mem x (cdr l1)) )
]
; (* end fun dec *)

fun  lmem (l1:''a list) (l2:''a list) :bool =
CASE_fun [
(is_null l1, fn zz => true),
(is_cons l1, fn zz => (mem (car l1) l2) AND (lmem (cdr l1) l2) )
]
; (* end fun dec *)

fun  mem1 (x:''a) (l1:(''a * ''b) list) :bool =
CASE_fun [
(is_null l1, fn zz => false),
(is_cons l1, fn zz => (x = fst(car l1)) OR (mem1 x (cdr l1)) )
]
```

```
; (* end fun dec *)

fun  mem2 (x:''b) (l1:(''a * ''b) list) : bool =
CASE_fun [
(is_null l1, fn zz => false),
(is_cons l1, fn zz => (x = snd(car l1)) OR (mem2 x (cdr l1)) )
]
; (* end fun dec *)

fun  nocontr (l1:(''a * ''b) list) :bool =
CASE_fun [
(is_null l1, fn zz => true),
(is_cons l1, fn zz =>
   ((NOT(mem2 (snd(car l1)) (cdr l1))) OR
       (corr2 (snd(car l1)) (cdr l1) = fst(car l1))) AND
    (nocontr (cdr l1))  )
]
; (* end fun dec *)
```

The following SML definitions are introduced by the translation of `type.al`.

```
datatype Type = Tyvar of (( string)) |
                Tyop_tup of (( string))*(( Type list)) ;
fun Is_Tyvar(Tyvar _) = true |
    Is_Tyvar(Tyop_tup _) = false ;
fun Is_Tyop(Tyvar _) = false |
    Is_Tyop(Tyop_tup _) = true ;
exception de_Tyvar_ex;
exception de_Tyop_tup_ex;
fun de_Tyvar (Tyvar x) = x  |
    de_Tyvar (Tyop_tup _) = raise de_Tyvar_ex ;
fun de_Tyop_tup (Tyvar _) = raise de_Tyop_tup_ex |
    de_Tyop_tup (Tyop_tup x) = x  ;
val Tyvar_nam = (tup1_1 o de_Tyvar) ;
val Tyop_nam = (tup2_1 o de_Tyop_tup) ;
val Tyop_tyl = (tup2_2 o de_Tyop_tup) ;
fun Tyop a b = Tyop_tup (a,b) ;


fun  XMAP2 (afn:''a->''b->''c) (l1:''a list) (l2:''b list) :(''c)list =
CASE_fun [
(is_null l1, fn zz => []),
(is_cons l1, fn zz => cons (afn (car l1) (car l2))
                 (XMAP2 afn (cdr l1) (cdr l2)) )
]
; (* end fun dec *)


fun  Type_OK (tyl:(string  *  num)list) (ty:Type) :bool =
CASE_fun [
(Is_Tyvar ty, fn zz => true),
(Is_Tyop ty, fn zz => (mem1 (Tyop_nam ty) tyl) AND
             (LENGTH (Tyop_tyl ty) = corr1 (Tyop_nam ty) tyl) AND
             EVERY (Type_OK tyl) (Tyop_tyl ty) )
```

```
]
; (* end fun dec *)

fun  Type_compat (ty1:Type) (ty2:Type) :bool =
CASE_fun [
(Is_Tyvar ty2, fn zz => true),
(Is_Tyop ty2, fn zz => (Is_Tyop ty1) AND
              ((Tyop_nam ty1) = (Tyop_nam ty2)) AND
              (LENGTH (Tyop_tyl ty1) = LENGTH (Tyop_tyl ty2)) AND
              (EVERY2 Type_compat (Tyop_tyl ty1) (Tyop_tyl ty2)) )
]
; (* end fun dec *)

fun  Type_occurs (str:string) (ty:Type) :bool =
CASE_fun [
(Is_Tyvar ty, fn zz => (str = Tyvar_nam ty)),
(Is_Tyop ty, fn zz =>  lor (MAP (Type_occurs str) (Tyop_tyl ty)) )
]
; (* end fun dec *)

fun  Type_replace (l:(Type * string)list) (ty:Type) :Type =
CASE_fun [
(Is_Tyvar ty, fn zz => (if mem2 (Tyvar_nam ty) l
              then corr2 (Tyvar_nam ty) l
              else ty
              ) ),
(Is_Tyop ty, fn zz => Tyop (Tyop_nam ty) (MAP(Type_replace l)(Tyop_tyl ty)) )
]
; (* end fun dec *)

fun  Type_instlx (ty1:Type) (ty2:Type) :(Type * string)list =
CASE_fun [
(Is_Tyvar ty1, fn zz => [(ty2,Tyvar_nam ty1)]),
(Is_Tyop ty1, fn zz =>
 LAPPEND(XMAP2 Type_instlx (Tyop_tyl ty1) (Tyop_tyl ty2)) )
]
; (* end fun dec *)

fun  Type_instl (ty1:Type) (ty2:Type) :(Type * string)list =
    Type_instlx ty2 ty1
; (* end fun dec *)
```

The SML code generated by `term.al` is as follows:

```
datatype Pterm = Const_tup of (( string))*(( Type)) |
                 Var of (( (string * Type))) |
                 App_tup of (( Pterm))*(( Pterm)) |
                 Lam_tup of (( (string * Type)))*(( Pterm)) ;
fun Is_Const(Const_tup _) = true |
    Is_Const(Var _) = false |
    Is_Const(App_tup _) = false |
    Is_Const(Lam_tup _) = false ;
fun Is_Var(Const_tup _) = false |
    Is_Var(Var _) = true |
```

```
    Is_Var(App_tup _) = false |
    Is_Var(Lam_tup _) = false ;
fun Is_App(Const_tup _) = false |
    Is_App(Var _) = false |
    Is_App(App_tup _) = true |
    Is_App(Lam_tup _) = false ;
fun Is_Lam(Const_tup _) = false |
    Is_Lam(Var _) = false |
    Is_Lam(App_tup _) = false |
    Is_Lam(Lam_tup _) = true ;
exception de_Const_tup_ex;
exception de_Var_ex;
exception de_App_tup_ex;
exception de_Lam_tup_ex;
fun de_Const_tup (Const_tup x) = x  |
    de_Const_tup (Var _) = raise de_Const_tup_ex |
    de_Const_tup (App_tup _) = raise de_Const_tup_ex |
    de_Const_tup (Lam_tup _) = raise de_Const_tup_ex ;
fun de_Var (Const_tup _) = raise de_Var_ex |
    de_Var (Var x) = x  |
    de_Var (App_tup _) = raise de_Var_ex |
    de_Var (Lam_tup _) = raise de_Var_ex ;
fun de_App_tup (Const_tup _) = raise de_App_tup_ex |
    de_App_tup (Var _) = raise de_App_tup_ex |
    de_App_tup (App_tup x) = x  |
    de_App_tup (Lam_tup _) = raise de_App_tup_ex ;
fun de_Lam_tup (Const_tup _) = raise de_Lam_tup_ex |
    de_Lam_tup (Var _) = raise de_Lam_tup_ex |
    de_Lam_tup (App_tup _) = raise de_Lam_tup_ex |
    de_Lam_tup (Lam_tup x) = x  ;
val Const_nam = (tup2_1 o de_Const_tup) ;
val Const_ty = (tup2_2 o de_Const_tup) ;
val Var_var = (tup1_1 o de_Var) ;
val App_fun = (tup2_1 o de_App_tup) ;
val App_arg = (tup2_2 o de_App_tup) ;
val Lam_var = (tup2_1 o de_Lam_tup) ;
val Lam_bod = (tup2_2 o de_Lam_tup) ;
fun Const a b = Const_tup (a,b) ;
fun App a b = App_tup (a,b) ;
fun Lam a b = Lam_tup (a,b) ;

fun  Ptype_of  (ptm: Pterm) :Type =
CASE_fun [
(Is_Const ptm, fn zz => Const_ty ptm),
(Is_Var ptm, fn zz => snd(Var_var ptm) ),
(Is_App ptm, fn zz => car(cdr(Tyop_tyl(Ptype_of (App_fun ptm)))) ),
(Is_Lam ptm, fn zz => Tyop "fun" [snd (Lam_var ptm), Ptype_of (Lam_bod ptm)] )
]
; (* end fun dec *)

fun  Pboolean (t: Pterm) :bool = (Ptype_of t = Tyop "bool" []) ; (* end fun dec *)

fun  Pwell_typed
(Typl:(string * num)list) (Conl:(string * Type)list) (ptm:Pterm) :bool =
CASE_fun [
```

```
(Is_Const ptm, fn zz =>
         (mem1 (Const_nam ptm) Conl) AND
         (Type_OK Typl (Const_ty ptm)) AND
         (Type_compat (Const_ty ptm) (corr1 (Const_nam ptm) Conl)) AND
         (nocontr(Type_instl (Const_ty ptm) (corr1 (Const_nam ptm)Conl))) ),
(Is_Var ptm, fn zz => Type_OK Typl (snd (Var_var ptm)) ),
(Is_App ptm, fn zz =>
         (Pwell_typed Typl Conl (App_fun ptm)) AND
         (Pwell_typed Typl Conl (App_arg ptm)) AND
         (Is_Tyop(Ptype_of (App_fun ptm))) AND
         (Tyop_nam(Ptype_of (App_fun ptm)) = "fun") AND
         (LENGTH(Tyop_tyl(Ptype_of (App_fun ptm))) = 2) AND
         (car(Tyop_tyl(Ptype_of (App_fun ptm))) = Ptype_of (App_arg ptm)) ),
(Is_Lam ptm, fn zz =>
         (Pwell_typed Typl Conl (Lam_bod ptm)) AND
         (Type_OK Typl(snd (Lam_var ptm))))
]
; (* end fun dec *)

fun  Pfree (x:string * Type) (ptm: Pterm) :bool =
CASE_fun [
(Is_Const ptm, fn zz => false),
(Is_Var ptm, fn zz => (Var_var ptm = x) ),
(Is_App ptm, fn zz => (Pfree x (App_fun ptm)) OR (Pfree x (App_arg ptm)) ),
(Is_Lam ptm, fn zz => (NOT(Lam_var ptm = x)) AND (Pfree x (Lam_bod ptm)) )
]
; (* end fun dec *)

fun  Pbound (x:string * Type) (ptm: Pterm) :bool =
CASE_fun [
(Is_Const ptm, fn zz => false),
(Is_Var ptm, fn zz => false ),
(Is_App ptm, fn zz => (Pbound x (App_fun ptm)) OR (Pbound x (App_arg ptm)) ),
(Is_Lam ptm, fn zz => (Lam_var ptm = x) OR (Pbound x (Lam_bod ptm)) )
]
; (* end fun dec *)

fun  Poccurs (x:string * Type) (t:Pterm) :bool =
   (Pfree x t) OR (Pbound x t) ; (* end fun dec *)

fun  Plnotfree (l1:(string * Type) list) (t:Pterm) :bool =
CASE_fun [
(is_null l1, fn zz => true),
(is_cons l1, fn zz => (NOT(Pfree (car l1) t)) AND (Plnotfree (cdr l1) t) )
]
; (* end fun dec *)

fun  Plnotbound (l1:(string * Type) list) (t:Pterm) :bool =
CASE_fun [
(is_null l1, fn zz => true),
(is_cons l1, fn zz => (NOT(Pbound (car l1) t)) AND (Plnotbound (cdr l1) t) )
]
; (* end fun dec *)

fun  Plnotoccurs (xl:(string * Type) list) (t:Pterm) :bool =
```

```
    (Plnotfree xl t) AND (Plnotbound xl t) ; (* end fun dec *)


fun  NotPfree (x:string * Type) (t:Pterm) :bool = NOT(Pfree x t) ; (* end fun dec *)

fun  Pallnotfree (x:string * Type) (tms:(Pterm)set) :bool =
    SEVERY (NotPfree x) tms ; (* end fun dec *)

fun  Plallnotfree (xl:(string * Type) list) (tms:(Pterm)set) =
    SEVERY (Plnotfree xl) tms ; (* end fun dec *)

fun  Palreplace1 (t':Pterm) (vvl:((string * Type) * (string * Type))list)
          (tvl:(Pterm  *  (string  *  Type))list) (ptm: Pterm) :bool =
CASE_fun [
(Is_Const ptm, fn zz => (t' = ptm) ),

(Is_Var ptm, fn zz => (if (Is_Var t') AND (mem1 (Var_var t') vvl)
             then ((Var_var ptm) = corr1(Var_var t')vvl)
             else (NOT(mem1 (Var_var ptm) vvl)) AND
                  ((if (mem2 (Var_var ptm) tvl)
                   then (t' = corr2 (Var_var ptm) tvl)
                   else (t' = ptm)
                   ))
             ) ),

(Is_App ptm, fn zz => (Is_App t') AND
         Palreplace1(App_fun t')vvl tvl (App_fun ptm) AND
         Palreplace1(App_arg t')vvl tvl (App_arg ptm) ),

(Is_Lam ptm, fn zz => (Is_Lam t') AND
         (snd(Lam_var t') = snd (Lam_var ptm)) AND
         Palreplace1
         (Lam_bod t')
         (cons(Lam_var t',Lam_var ptm)vvl)
         tvl
        (Lam_bod ptm) )

]
; (* end fun dec *)

fun  Palreplace (t':Pterm) (tvl:(Pterm  * (string * Type))list)
               (t:Pterm) :bool =
    Palreplace1 t' [] tvl t ; (* end fun dec *)

fun  Palpha (t':Pterm) (t:Pterm) :bool = Palreplace t' [] t  ; (* end fun dec *)

fun  Psubst_triples (l1:(Pterm * (Pterm * (string * Type)))list):bool =
CASE_fun [
(is_null l1, fn zz => true),
(is_cons l1, fn zz =>
         (Ptype_of(fst(car l1)) = Ptype_of(fst(snd(car l1)))) AND
         (Ptype_of(fst(car l1)) = snd(snd(snd(car l1)))) AND
         Psubst_triples (cdr l1))
]
; (* end fun dec *)
```

```
fun  list13 (l1:(''a * (''b  * ''c))list) :(''a * ''c)list =
CASE_fun [
(is_null l1, fn zz => []),
(is_cons l1, fn zz => cons(fst (car l1),snd(snd (car l1)))(list13 (cdr l1)) )
]
; (* end fun dec *)


fun  Psubst (Typl:(string * num)list) (Conl:(string * Type)list) (t':Pterm)
           (ttvl:(Pterm  *  (Pterm  *  (string  *  Type)))list)
           (td:Pterm) (t:Pterm) :bool =
        (Psubst_triples ttvl) AND
        (Pwell_typed Typl Conl td) AND
        (Plnotoccurs(MAP snd(MAP snd ttvl))t) AND
        (Palreplace t (MAP snd ttvl)td) AND
        (Palreplace t'(list13 ttvl) td)
; (* end fun dec *)


fun  Pbeta (t':Pterm) (x:string * Type) (t1:Pterm) (t2:Pterm) :bool =
    Palreplace t' [(t2,x)] t1 ; (* end fun dec *)

fun  Pty_occurs (a:string) (ptm:Pterm) :bool =
CASE_fun [
(Is_Const ptm, fn zz => Type_occurs a (Const_ty ptm) ),
(Is_Var ptm, fn zz => Type_occurs a (snd(Var_var ptm)) ),
(Is_App ptm, fn zz => (Pty_occurs a (App_fun ptm)) OR (Pty_occurs a(App_arg ptm)) ),
(Is_Lam ptm, fn zz => (Type_occurs a(snd(Lam_var ptm))) OR (Pty_occurs a(Lam_bod ptm)))
]
; (* end fun dec *)

fun  NotPty_occurs (a:string) (t:Pterm) :bool = NOT(Pty_occurs a t) ; (* end fun dec *)

fun  Pty_snotoccurs (a:string) tms = SEVERY (NotPty_occurs a) tms ; (* end fun dec *)

fun  Plty_snotoccurs l1 tml =
CASE_fun [
(is_null l1, fn zz => true),
(is_cons l1, fn zz => (Pty_snotoccurs (car l1) tml) AND
             (Plty_snotoccurs (cdr l1) tml) )
]
; (* end fun dec *)

fun  Pnewfree1 t bl ptm =
CASE_fun [
(Is_Const ptm, fn zz => [] ),
(Is_Var ptm, fn zz => (if (mem (Var_var ptm) bl) OR (fst(Var_var t) = fst (Var_var ptm))
             then []
             else  [(Var_var t,Var_var ptm)]
             ) ),

(Is_App ptm, fn zz => APPEND(Pnewfree1(App_fun t)bl (App_fun ptm))
                    (Pnewfree1(App_arg t)bl (App_arg ptm)) ),
(Is_Lam ptm, fn zz => Pnewfree1(Lam_bod t)(cons(Lam_var ptm) bl) (Lam_bod ptm) )
```

```
]
; (* end fun dec *)

fun  Pnewfree t' t = Pnewfree1 t' [] t ; (* end fun dec *)

fun  Ptyinst1
    (t:Pterm) (bvl:((string * Type) * (string * Type))list)
    (fvl:((string * Type) * (string * Type))list)
    (tyl:(Type  *  string)list) (ptm:Pterm)   =

CASE_fun [
(Is_Const ptm, fn zz => (t = Const (Const_nam ptm) (Type_replace tyl (Const_ty ptm))) ),

(Is_Var ptm, fn zz =>
     (if ((Is_Var t) AND (mem1(Var_var t)bvl))
     then ((Var_var ptm) = corr1(Var_var t)bvl) AND
          (snd(Var_var t) = Type_replace tyl(snd (Var_var ptm)))
     else
         ((if (mem2 (Var_var ptm) fvl)
          then
           (t = Var(fst(corr2 (Var_var ptm) fvl),
                   Type_replace tyl(snd(Var_var ptm))))
          else
           (t = Var(fst (Var_var ptm),
                   Type_replace tyl(snd (Var_var ptm))))
          ))
     ) ),

(Is_App ptm, fn zz => Is_App t AND
            Ptyinst1(App_fun t)bvl fvl tyl (App_fun ptm) AND
            Ptyinst1(App_arg t)bvl fvl tyl (App_arg ptm) ),

(Is_Lam ptm, fn zz => Is_Lam t AND
           (NOT(mem1(Lam_var t)fvl)) AND
           (snd(Lam_var t) = Type_replace tyl(snd (Lam_var ptm))) AND
            Ptyinst1
           (Lam_bod t)
           (cons(Lam_var t,(Lam_var ptm))bvl)
           fvl
           tyl
           (Lam_bod ptm)  )
]
; (* end fun dec *)

fun  Ptyinst ass t' tyl t =
       Ptyinst1 t' [] (Pnewfree t' t) tyl t AND
       Plallnotfree (MAP snd(Pnewfree t' t)) ass
; (* end fun dec *)
```

The final set of SML definitions are generated by the file `inf.al`. The final definition of `Is_proof` defines a function which checks a HOL proof.

```
datatype Psequent = Pseq_tup of (((Pterm)set))*(( Pterm)) ;
fun Is_Pseq(Pseq_tup _) = true ;
```

```
exception de_Pseq_tup_ex;
fun de_Pseq_tup (Pseq_tup x) = x  ;
val Pseq_assum = (tup2_1 o de_Pseq_tup) ;
val Pseq_concl = (tup2_2 o de_Pseq_tup) ;
fun Pseq a b = Pseq_tup (a,b) ;

fun  Pseq_boolean pseq =
 SEVERY Pboolean (Pseq_assum pseq) AND
 Pboolean (Pseq_concl pseq)
; (* end fun dec *)

fun  Pseq_well_typed Typl Conl pseq =
        SEVERY(Pwell_typed Typl Conl)(Pseq_assum pseq) AND
        Pwell_typed Typl Conl (Pseq_concl pseq)
; (* end fun dec *)

fun  PEQ t1 t2 =
        App
        (App
         (Const
          "="
          (Tyop
           "fun"
           [Ptype_of t1,Tyop "fun"[Ptype_of t1,Tyop "bool"[]]]))
         t1)
        t2
; (* end fun dec *)

fun  PIMP t1 t2 =
        App
        (App
         (Const
          "==>"
          (Tyop
           "fun"
           [Tyop "bool"[],Tyop "fun"[Tyop "bool"[],Tyop "bool"[]]]))
         t1)
        t2
; (* end fun dec *)

fun  Is_EQtm t =
        Is_App t AND
        Is_App(App_fun t) AND
        (App_fun(App_fun t) =
         Const
         "="
         (Tyop
          "fun"
          [Ptype_of(App_arg t),
           Tyop "fun"[Ptype_of(App_arg t),Tyop "bool"[]]]))
; (* end fun dec *)

fun  PASSUME Typl Conl pseq tm =
        Pwell_typed Typl Conl tm AND
        Pboolean tm AND
```

```
        (Pseq_concl pseq = tm) AND
        (Pseq_assum pseq = (SetOf[tm]))
; (* end fun dec *)

fun  PREFL Typl Conl pseq tm =
        Pwell_typed Typl Conl tm AND
        ((Pseq_assum pseq) = empty) AND
        ((Pseq_concl pseq) = PEQ tm tm)
; (* end fun dec *)

fun  PBETA_CONV Typl Conl pseq tm =
        (Pwell_typed Typl Conl tm) AND
        (Pseq_assum pseq = empty) AND
        (Is_App tm) AND
        Is_Lam(App_fun tm) AND
        ((Pseq_concl pseq) = PEQ tm(App_arg (Pseq_concl pseq))) AND
        (Pbeta
        (App_arg (Pseq_concl pseq))
        (Lam_var(App_fun tm))
        (Lam_bod(App_fun tm))
        (App_arg tm))
; (* end fun dec *)

fun  Psubst_newlist (l1:(Psequent  *  (string  *  Type))list)
                   :(Pterm  *  (Pterm  *  (string  *  Type))) list =
CASE_fun [
(is_null l1, fn zz => []),
(is_cons l1, fn zz => cons
         (App_arg(Pseq_concl(fst (car l1))),
          (App_arg(App_fun(Pseq_concl(fst (car l1)))),snd (car l1)))
         (Psubst_newlist (cdr l1)) )
]
; (* end fun dec *)

fun  PSUBST Typl Conl pseq thdl td th =
        Pwell_typed Typl Conl td AND
        EVERY Is_EQtm (MAP Pseq_concl(MAP fst thdl)) AND
        Psubst Typl Conl
        (Pseq_concl pseq) (Psubst_newlist thdl)td(Pseq_concl th) AND
        ((Pseq_assum pseq) =
         ((Pseq_assum th) UNION (LUNION(MAP Pseq_assum(MAP fst thdl)))))
; (* end fun dec *)

fun  PABS Typl Conl pseq tm th =
        Pwell_typed Typl Conl tm AND
        Is_EQtm(Pseq_concl th) AND
        Is_Var tm AND
        Type_OK Typl(snd(Var_var tm)) AND
        ((Pseq_concl pseq) =
         PEQ
         (Lam(Var_var tm)(App_arg(App_fun(Pseq_concl th))))
         (Lam(Var_var tm)(App_arg(Pseq_concl th)))) AND
        (Pseq_assum pseq = Pseq_assum th) AND
        Pallnotfree(Var_var tm)(Pseq_assum pseq)
; (* end fun dec *)
```

```
fun  PINST_TYPE Typl pseq tyl th =
        EVERY(Type_OK Typl)(MAP fst tyl) AND
        Ptyinst (Pseq_assum pseq) (Pseq_concl pseq) tyl (Pseq_concl th) AND
        Plty_snotoccurs(MAP snd tyl)(Pseq_assum pseq) AND
        (Pseq_assum pseq = Pseq_assum th)
; (* end fun dec *)

fun  PDISCH Typl Conl pseq tm th =
        Pwell_typed Typl Conl tm AND
        Pboolean tm AND
        (Pseq_concl pseq = PIMP tm(Pseq_concl th)) AND
        (Pseq_assum pseq = ((Pseq_assum th) DELETE tm))
; (* end fun dec *)


fun  PMP pseq th1 th2 =
        (Pseq_concl th1 = PIMP(Pseq_concl th2)(Pseq_concl pseq)) AND
        ((Pseq_assum pseq) = ((Pseq_assum th1) UNION (Pseq_assum th2)))
; (* end fun dec *)


datatype Inference = AX_inf of (( Psequent)) |
                 AS_inf_tup of (( Psequent))*(( Pterm)) |
                 RE_inf_tup of (( Psequent))*(( Pterm)) |
                 BE_inf_tup of (( Psequent))*(( Pterm)) |
                 SU_inf_tup of
                 (( Psequent))*
                 (( (Psequent * (string * Type)) list))*
                 (( Pterm))*(( Psequent)) |
                 AB_inf_tup of (( Psequent))*(( Pterm))*(( Psequent)) |
                 IN_inf_tup of (( Psequent))*(((Type * string)list))*(( Psequent)) |
                 DI_inf_tup of (( Psequent))*(( Pterm))*(( Psequent)) |
                 MP_inf_tup of (( Psequent))*(( Psequent))*(( Psequent)) ;
fun is_AX(AX_inf _) = true |
    is_AX(AS_inf_tup _) = false |
    is_AX(RE_inf_tup _) = false |
    is_AX(BE_inf_tup _) = false |
    is_AX(SU_inf_tup _) = false |
    is_AX(AB_inf_tup _) = false |
    is_AX(IN_inf_tup _) = false |
    is_AX(DI_inf_tup _) = false |
    is_AX(MP_inf_tup _) = false ;
fun is_AS(AX_inf _) = false |
    is_AS(AS_inf_tup _) = true |
    is_AS(RE_inf_tup _) = false |
    is_AS(BE_inf_tup _) = false |
    is_AS(SU_inf_tup _) = false |
    is_AS(AB_inf_tup _) = false |
    is_AS(IN_inf_tup _) = false |
    is_AS(DI_inf_tup _) = false |
    is_AS(MP_inf_tup _) = false ;
fun is_RE(AX_inf _) = false |
    is_RE(AS_inf_tup _) = false |
    is_RE(RE_inf_tup _) = true |
```

```
    is_RE(BE_inf_tup _) = false |
    is_RE(SU_inf_tup _) = false |
    is_RE(AB_inf_tup _) = false |
    is_RE(IN_inf_tup _) = false |
    is_RE(DI_inf_tup _) = false |
    is_RE(MP_inf_tup _) = false ;
fun is_BE(AX_inf _) = false |
    is_BE(AS_inf_tup _) = false |
    is_BE(RE_inf_tup _) = false |
    is_BE(BE_inf_tup _) = true |
    is_BE(SU_inf_tup _) = false |
    is_BE(AB_inf_tup _) = false |
    is_BE(IN_inf_tup _) = false |
    is_BE(DI_inf_tup _) = false |
    is_BE(MP_inf_tup _) = false ;
fun is_SU(AX_inf _) = false |
    is_SU(AS_inf_tup _) = false |
    is_SU(RE_inf_tup _) = false |
    is_SU(BE_inf_tup _) = false |
    is_SU(SU_inf_tup _) = true |
    is_SU(AB_inf_tup _) = false |
    is_SU(IN_inf_tup _) = false |
    is_SU(DI_inf_tup _) = false |
    is_SU(MP_inf_tup _) = false ;
fun is_AB(AX_inf _) = false |
    is_AB(AS_inf_tup _) = false |
    is_AB(RE_inf_tup _) = false |
    is_AB(BE_inf_tup _) = false |
    is_AB(SU_inf_tup _) = false |
    is_AB(AB_inf_tup _) = true |
    is_AB(IN_inf_tup _) = false |
    is_AB(DI_inf_tup _) = false |
    is_AB(MP_inf_tup _) = false ;
fun is_IN(AX_inf _) = false |
    is_IN(AS_inf_tup _) = false |
    is_IN(RE_inf_tup _) = false |
    is_IN(BE_inf_tup _) = false |
    is_IN(SU_inf_tup _) = false |
    is_IN(AB_inf_tup _) = false |
    is_IN(IN_inf_tup _) = true |
    is_IN(DI_inf_tup _) = false |
    is_IN(MP_inf_tup _) = false ;
fun is_DI(AX_inf _) = false |
    is_DI(AS_inf_tup _) = false |
    is_DI(RE_inf_tup _) = false |
    is_DI(BE_inf_tup _) = false |
    is_DI(SU_inf_tup _) = false |
    is_DI(AB_inf_tup _) = false |
    is_DI(IN_inf_tup _) = false |
    is_DI(DI_inf_tup _) = true |
    is_DI(MP_inf_tup _) = false ;
fun is_MP(AX_inf _) = false |
    is_MP(AS_inf_tup _) = false |
    is_MP(RE_inf_tup _) = false |
    is_MP(BE_inf_tup _) = false |
```

```
    is_MP(SU_inf_tup _) = false |
    is_MP(AB_inf_tup _) = false |
    is_MP(IN_inf_tup _) = false |
    is_MP(DI_inf_tup _) = false |
    is_MP(MP_inf_tup _) = true ;
exception de_AX_inf_ex;
exception de_AS_inf_tup_ex;
exception de_RE_inf_tup_ex;
exception de_BE_inf_tup_ex;
exception de_SU_inf_tup_ex;
exception de_AB_inf_tup_ex;
exception de_IN_inf_tup_ex;
exception de_DI_inf_tup_ex;
exception de_MP_inf_tup_ex;
fun de_AX_inf (AX_inf x) = x  |
    de_AX_inf (AS_inf_tup _) = raise de_AX_inf_ex |
    de_AX_inf (RE_inf_tup _) = raise de_AX_inf_ex |
    de_AX_inf (BE_inf_tup _) = raise de_AX_inf_ex |
    de_AX_inf (SU_inf_tup _) = raise de_AX_inf_ex |
    de_AX_inf (AB_inf_tup _) = raise de_AX_inf_ex |
    de_AX_inf (IN_inf_tup _) = raise de_AX_inf_ex |
    de_AX_inf (DI_inf_tup _) = raise de_AX_inf_ex |
    de_AX_inf (MP_inf_tup _) = raise de_AX_inf_ex ;
fun de_AS_inf_tup (AX_inf _) = raise de_AS_inf_tup_ex |
    de_AS_inf_tup (AS_inf_tup x) = x  |
    de_AS_inf_tup (RE_inf_tup _) = raise de_AS_inf_tup_ex |
    de_AS_inf_tup (BE_inf_tup _) = raise de_AS_inf_tup_ex |
    de_AS_inf_tup (SU_inf_tup _) = raise de_AS_inf_tup_ex |
    de_AS_inf_tup (AB_inf_tup _) = raise de_AS_inf_tup_ex |
    de_AS_inf_tup (IN_inf_tup _) = raise de_AS_inf_tup_ex |
    de_AS_inf_tup (DI_inf_tup _) = raise de_AS_inf_tup_ex |
    de_AS_inf_tup (MP_inf_tup _) = raise de_AS_inf_tup_ex ;
fun de_RE_inf_tup (AX_inf _) = raise de_RE_inf_tup_ex |
    de_RE_inf_tup (AS_inf_tup _) = raise de_RE_inf_tup_ex |
    de_RE_inf_tup (RE_inf_tup x) = x  |
    de_RE_inf_tup (BE_inf_tup _) = raise de_RE_inf_tup_ex |
    de_RE_inf_tup (SU_inf_tup _) = raise de_RE_inf_tup_ex |
    de_RE_inf_tup (AB_inf_tup _) = raise de_RE_inf_tup_ex |
    de_RE_inf_tup (IN_inf_tup _) = raise de_RE_inf_tup_ex |
    de_RE_inf_tup (DI_inf_tup _) = raise de_RE_inf_tup_ex |
    de_RE_inf_tup (MP_inf_tup _) = raise de_RE_inf_tup_ex ;
fun de_BE_inf_tup (AX_inf _) = raise de_BE_inf_tup_ex |
    de_BE_inf_tup (AS_inf_tup _) = raise de_BE_inf_tup_ex |
    de_BE_inf_tup (RE_inf_tup _) = raise de_BE_inf_tup_ex |
    de_BE_inf_tup (BE_inf_tup x) = x  |
    de_BE_inf_tup (SU_inf_tup _) = raise de_BE_inf_tup_ex |
    de_BE_inf_tup (AB_inf_tup _) = raise de_BE_inf_tup_ex |
    de_BE_inf_tup (IN_inf_tup _) = raise de_BE_inf_tup_ex |
    de_BE_inf_tup (DI_inf_tup _) = raise de_BE_inf_tup_ex |
    de_BE_inf_tup (MP_inf_tup _) = raise de_BE_inf_tup_ex ;
fun de_SU_inf_tup (AX_inf _) = raise de_SU_inf_tup_ex |
    de_SU_inf_tup (AS_inf_tup _) = raise de_SU_inf_tup_ex |
    de_SU_inf_tup (RE_inf_tup _) = raise de_SU_inf_tup_ex |
    de_SU_inf_tup (BE_inf_tup _) = raise de_SU_inf_tup_ex |
    de_SU_inf_tup (SU_inf_tup x) = x  |
```

```
    de_SU_inf_tup (AB_inf_tup _) = raise de_SU_inf_tup_ex |
    de_SU_inf_tup (IN_inf_tup _) = raise de_SU_inf_tup_ex |
    de_SU_inf_tup (DI_inf_tup _) = raise de_SU_inf_tup_ex |
    de_SU_inf_tup (MP_inf_tup _) = raise de_SU_inf_tup_ex ;
fun de_AB_inf_tup (AX_inf _) = raise de_AB_inf_tup_ex |
    de_AB_inf_tup (AS_inf_tup _) = raise de_AB_inf_tup_ex |
    de_AB_inf_tup (RE_inf_tup _) = raise de_AB_inf_tup_ex |
    de_AB_inf_tup (BE_inf_tup _) = raise de_AB_inf_tup_ex |
    de_AB_inf_tup (SU_inf_tup _) = raise de_AB_inf_tup_ex |
    de_AB_inf_tup (AB_inf_tup x) = x  |
    de_AB_inf_tup (IN_inf_tup _) = raise de_AB_inf_tup_ex |
    de_AB_inf_tup (DI_inf_tup _) = raise de_AB_inf_tup_ex |
    de_AB_inf_tup (MP_inf_tup _) = raise de_AB_inf_tup_ex ;
fun de_IN_inf_tup (AX_inf _) = raise de_IN_inf_tup_ex |
    de_IN_inf_tup (AS_inf_tup _) = raise de_IN_inf_tup_ex |
    de_IN_inf_tup (RE_inf_tup _) = raise de_IN_inf_tup_ex |
    de_IN_inf_tup (BE_inf_tup _) = raise de_IN_inf_tup_ex |
    de_IN_inf_tup (SU_inf_tup _) = raise de_IN_inf_tup_ex |
    de_IN_inf_tup (AB_inf_tup _) = raise de_IN_inf_tup_ex |
    de_IN_inf_tup (IN_inf_tup x) = x  |
    de_IN_inf_tup (DI_inf_tup _) = raise de_IN_inf_tup_ex |
    de_IN_inf_tup (MP_inf_tup _) = raise de_IN_inf_tup_ex ;
fun de_DI_inf_tup (AX_inf _) = raise de_DI_inf_tup_ex |
    de_DI_inf_tup (AS_inf_tup _) = raise de_DI_inf_tup_ex |
    de_DI_inf_tup (RE_inf_tup _) = raise de_DI_inf_tup_ex |
    de_DI_inf_tup (BE_inf_tup _) = raise de_DI_inf_tup_ex |
    de_DI_inf_tup (SU_inf_tup _) = raise de_DI_inf_tup_ex |
    de_DI_inf_tup (AB_inf_tup _) = raise de_DI_inf_tup_ex |
    de_DI_inf_tup (IN_inf_tup _) = raise de_DI_inf_tup_ex |
    de_DI_inf_tup (DI_inf_tup x) = x  |
    de_DI_inf_tup (MP_inf_tup _) = raise de_DI_inf_tup_ex ;
fun de_MP_inf_tup (AX_inf _) = raise de_MP_inf_tup_ex |
    de_MP_inf_tup (AS_inf_tup _) = raise de_MP_inf_tup_ex |
    de_MP_inf_tup (RE_inf_tup _) = raise de_MP_inf_tup_ex |
    de_MP_inf_tup (BE_inf_tup _) = raise de_MP_inf_tup_ex |
    de_MP_inf_tup (SU_inf_tup _) = raise de_MP_inf_tup_ex |
    de_MP_inf_tup (AB_inf_tup _) = raise de_MP_inf_tup_ex |
    de_MP_inf_tup (IN_inf_tup _) = raise de_MP_inf_tup_ex |
    de_MP_inf_tup (DI_inf_tup _) = raise de_MP_inf_tup_ex |
    de_MP_inf_tup (MP_inf_tup x) = x  ;
val AX_con = (tup1_1 o de_AX_inf) ;
val AS_con = (tup2_1 o de_AS_inf_tup) ;
val AS_tm = (tup2_2 o de_AS_inf_tup) ;
val RE_con = (tup2_1 o de_RE_inf_tup) ;
val RE_tm = (tup2_2 o de_RE_inf_tup) ;
val BE_con = (tup2_1 o de_BE_inf_tup) ;
val BE_tm = (tup2_2 o de_BE_inf_tup) ;
val SU_con = (tup4_1 o de_SU_inf_tup) ;
val SU_eqthms = (tup4_2 o de_SU_inf_tup) ;
val SU_tmdum = (tup4_3 o de_SU_inf_tup) ;
val SU_thm = (tup4_4 o de_SU_inf_tup) ;
val AB_con = (tup3_1 o de_AB_inf_tup) ;
val AB_tm = (tup3_2 o de_AB_inf_tup) ;
val AB_eq = (tup3_3 o de_AB_inf_tup) ;
val IN_con = (tup3_1 o de_IN_inf_tup) ;
```

```
val IN_tyl = (tup3_2 o de_IN_inf_tup) ;
val IN_thm = (tup3_3 o de_IN_inf_tup) ;
val DI_con = (tup3_1 o de_DI_inf_tup) ;
val DI_tm = (tup3_2 o de_DI_inf_tup) ;
val DI_thm = (tup3_3 o de_DI_inf_tup) ;
val MP_con = (tup3_1 o de_MP_inf_tup) ;
val MP_ithm = (tup3_2 o de_MP_inf_tup) ;
val MP_thm = (tup3_3 o de_MP_inf_tup) ;
fun AS_inf a b = AS_inf_tup (a,b) ;
fun RE_inf a b = RE_inf_tup (a,b) ;
fun BE_inf a b = BE_inf_tup (a,b) ;
fun SU_inf a b c d = SU_inf_tup (a,b,c,d) ;
fun AB_inf a b c = AB_inf_tup (a,b,c) ;
fun IN_inf a b c = IN_inf_tup (a,b,c) ;
fun DI_inf a b c = DI_inf_tup (a,b,c) ;
fun MP_inf a b c = MP_inf_tup (a,b,c) ;

fun  Inf_concl inf =
CASE_fun [
(is_AX inf, fn zz => AX_con inf),
(is_AS inf, fn zz => AS_con inf),
(is_RE inf, fn zz => RE_con inf),
(is_BE inf, fn zz => BE_con inf),
(is_SU inf, fn zz => SU_con inf),
(is_AB inf, fn zz => AB_con inf),
(is_IN inf, fn zz => IN_con inf),
(is_DI inf, fn zz => DI_con inf),
(is_MP inf, fn zz => MP_con inf)
]
; (* end fun dec *)

fun  Inf_hyps inf =
CASE_fun [
(is_AX inf, fn zz => []),
(is_AS inf, fn zz => []),
(is_RE inf, fn zz => []),
(is_BE inf, fn zz => []),
(is_SU inf, fn zz => cons (SU_thm inf) (MAP fst (SU_eqthms inf)) ),
(is_AB inf, fn zz => [AB_eq inf] ),
(is_IN inf, fn zz => [IN_thm inf] ),
(is_DI inf, fn zz => [DI_thm inf] ),
(is_MP inf, fn zz => [(MP_ithm inf), (MP_thm inf)] )
]
; (* end fun dec *)

fun  OK_Inf Typl Conl Axil inf =
CASE_fun [
(is_AX inf, fn zz => mem (AX_con inf) Axil),
(is_AS inf, fn zz => PASSUME Typl Conl (AS_con inf) (AS_tm inf) ),
(is_RE inf, fn zz => PREFL Typl Conl (RE_con inf) (RE_tm inf) ),
(is_BE inf, fn zz => PBETA_CONV Typl Conl (BE_con inf) (BE_tm inf)  ),
(is_SU inf, fn zz => PSUBST Typl Conl
            (SU_con inf)(SU_eqthms inf)(SU_tmdum inf)(SU_thm inf) ),
(is_AB inf, fn zz => PABS Typl Conl (AB_con inf) (AB_tm inf) (AB_eq inf) ),
(is_IN inf, fn zz => PINST_TYPE Typl (IN_con inf) (IN_tyl inf) (IN_thm inf) ),
```

```
(is_DI inf, fn zz => PDISCH Typl Conl (DI_con inf) (DI_tm inf) (DI_thm inf) ),
(is_MP inf, fn zz => PMP (MP_con inf) (MP_ithm inf) (MP_thm inf) )
]
; (* end fun dec *)

fun  Is_proof Typl Conl Axil l =
CASE_fun [
(is_null l, fn zz => true),
(is_cons l, fn zz => OK_Inf Typl Conl Axil (car l) AND
            lmem(Inf_hyps (car l))(MAP Inf_concl (cdr l)) AND
            Is_proof Typl Conl Axil (cdr l) )
]
; (* end fun dec *)
```

Part VIII

Appendices for Part IV

Appendix E

Theory of proofs

This appendix shows the constant definitions of the subtheories that make up the theory of higher order logic proofs.

E.1 The `proofaux` theory

This theory defines a number of general functions for handling lists and sets, needed in the handling of proofs. They are listed here for completeness.

```
EVERY_DEF
  |- (!P. EVERY P[] = T) /\
     (!P x l. EVERY P(CONS x l) = P x /\ EVERY P l)
EVERY2
  |- (!P. EVERY2 P[][] = T) /\
     (!P x xl y yl.
       EVERY2 P(CONS x xl)(CONS y yl) = P x y /\ EVERY2 P xl yl)
EVERY3
  |- (!P. EVERY3 P[][][] = T) /\
     (!P x xl y yl.
       EVERY3 P(CONS x xl)(CONS y yl)(CONS z zl) =
       P x y z /\ EVERY3 P xl yl zl)
XMAP2
  |- (!f. XMAP2 f[][] = []) /\
     (!f x xl y yl.
       XMAP2 f(CONS x xl)(CONS y yl) = CONS(f x y)(XMAP2 f xl yl))
XMAP2_DEF
  |- (!f yl. XMAP2 f[]yl = []) /\
     (!f x xl yl.
       XMAP2 f(CONS x xl)yl = CONS(f x(HD yl))(XMAP2 f xl(TL yl)))
FMAP
  |- (FMAP[][] = []) /\
     (!f fl x xl. FMAP(CONS f fl)(CONS x xl) = CONS(f x)(FMAP fl xl))
```

```
lor_DEF  |- (lor[] = F) /\ (!h t. lor(CONS h t) = h \/ lor t)
land_DEF  |- (land[] = T) /\ (!h t. land(CONS h t) = h /\ land t)
mem_DEF
  |- (!x. mem x[] = F) /\
     (!x h t. mem x(CONS h t) = (x = h) \/ mem x t)
mem1_DEF
  |- (!x. mem1 x[] = F) /\
     (!x h t. mem1 x(CONS h t) = (x = FST h) \/ mem1 x t)
mem2_DEF
  |- (!x. mem2 x[] = F) /\
     (!x h t. mem2 x(CONS h t) = (x = SND h) \/ mem2 x t)
corr1_DEF
  |- (!x. corr1 x[] = (@y. T)) /\
     (!x h t. corr1 x(CONS h t) = ((x = FST h) => SND h | corr1 x t))
corr2_DEF
  |- (!x. corr2 x[] = (@y. T)) /\
     (!x h t. corr2 x(CONS h t) = ((x = SND h) => FST h | corr2 x t))
lmem_DEF
  |- (!l. lmem[]l = T) /\
     (!x xl l. lmem(CONS x xl)l = mem x l /\ lmem xl l)
idlist_DEF
  |- (idlist[] = T) /\
     (!h t. idlist(CONS h t) = (FST h = SND h) /\ idlist t)
LAPPEND_DEF
  |- (LAPPEND[] = []) /\
     (!h t. LAPPEND(CONS h t) = APPEND h(LAPPEND t))
nocontr_DEF
  |- (nocontr[] = T) /\
     (!xy xyl.
       nocontr(CONS xy xyl) =
       (~mem2(SND xy)xyl \/ (corr2(SND xy)xyl = FST xy)) /\
       nocontr xyl)
SEVERY_DEF
  |- (!P. SEVERY P EMPTY = T) /\
     (!P x s. SEVERY P(x INSERT s) = P x /\ SEVERY P s)
EVERYS_DEF
  |- (!P. EVERYS P[] = T) /\
     (!P h t. EVERYS P(CONS h t) = SEVERY P h /\ EVERYS P t)
LUNION_DEF
  |- (LUNION[] = EMPTY) /\
     (!h t. LUNION(CONS h t) = h UNION (LUNION t))
PROX_DEF
  |- (!s. PROX s[] = []) /\
     (!s h t.
       PROX s(CONS h t) =
       ((FST(SND h) = s) => PROX s t | CONS h(PROX s t)))
EVERYX_DEF
  |- (!R Y. EVERYX R[]Y = T) /\
     (!R h t Y. EVERYX R(CONS h t)Y = R h(HD Y) /\ EVERYX R t(TL Y))
```

E.2 The Type theory

Below are the definitions from the Type theory. These theorems characterise a number of functions defined over Types.

```
  Type_Axiom
    |- !f1 f2. ?! fn. (!s. fn(Tyvar s) = f1 s) /\
                      (!s ts. fn(Tyop s ts) = f2 s(MAP fn ts)ts)
  Is_Tyvar_DEF
    |- (!s. Is_Tyvar(Tyvar s) = T) /\ (!s ts. Is_Tyvar(Tyop s ts) = F)
  Is_Tyop_DEF
    |- (!s. Is_Tyop(Tyvar s) = F) /\ (!s ts. Is_Tyop(Tyop s ts) = T)
  Tyvar_nam_DEF
    |- (!s. Tyvar_nam(Tyvar s) = s) /\
       (!s ts. Tyvar_nam(Tyop s ts) = (@y. T))
  Tyop_nam_DEF
    |- (!s. Tyop_nam(Tyvar s) = (@y. T)) /\
       (!s ts. Tyop_nam(Tyop s ts) = s)
  Tyop_tyl_DEF
    |- (!s. Tyop_tyl(Tyvar s) = (@y. T)) /\
       (!s ts. Tyop_tyl(Tyop s ts) = ts)
  Type_OK_DEF
    |- (!tyl s. Type_OK tyl(Tyvar s) = T) /\
       (!tyl s ts. Type_OK tyl(Tyop s ts) =
         mem1 s tyl /\ (LENGTH ts = corr1 s tyl) /\ EVERY(Type_OK tyl)ts)
  Type_compat_DEF
    |- (!s ty. Type_compat ty(Tyvar s) = T) /\
       (!s ts ty. Type_compat ty(Tyop s ts) =
         Is_Tyop ty /\
         (Tyop_nam ty = s) /\
         (LENGTH(Tyop_tyl ty) = LENGTH ts) /\
         EVERY2 Type_compat(Tyop_tyl ty)ts)
  Type_occurs_DEF
    |- (!s' s. Type_occurs s'(Tyvar s) = (s = s')) /\
       (!s' s ts. Type_occurs s'(Tyop s ts) = lor(MAP(Type_occurs s')ts))
  Type_replace_DEF
    |- (!l s. Type_replace l(Tyvar s) = (mem2 s l => corr2 s l | Tyvar s)) /\
       (!l s ts. Type_replace l(Tyop s ts) = Tyop s(MAP(Type_replace l)ts))
  Type_instl_thm
    |- !s ts ty. Type_compat ty(Tyop s ts) ==>
        (Type_instl ty(Tyop s ts) = LAPPEND(MAP2 Type_instl(Tyop_tyl ty)ts))
```

E.3 The Pterm theory

The Pterm theory defines properties of higher-order logic terms. We do not show all term destructor functions, as they are all alike.

```
% destructor functions: a few examples %
  Is_App_DEF
    |- (!s ty. Is_App(Const s ty) = F) /\
       (!x. Is_App(Var x) = F) /\
```

```
      (!t1 t2. Is_App(App t1 t2) = T) /\
      (!x t. Is_App(Lam x t) = F)
  Const_ty_DEF
    |- (!s ty. Const_ty(Const s ty) = ty) /\
      (!x. Const_ty(Var x) = (@y. T)) /\
      (!t1 t2. Const_ty(App t1 t2) = (@y. T)) /\
      (!x t. Const_ty(Lam x t) = (@y. T))
  Var_var_DEF
    |- (!s ty. Var_var(Const s ty) = (@y. T)) /\
      (!x. Var_var(Var x) = x) /\
      (!t1 t2. Var_var(App t1 t2) = (@y. T)) /\
      (!x t. Var_var(Lam x t) = (@y. T))
  Lam_bod_DEF
    |- (!s ty. Lam_bod(Const s ty) = (@y. T)) /\
      (!x. Lam_bod(Var x) = (@y. T)) /\
      (!t1 t2. Lam_bod(App t1 t2) = (@y. T)) /\
      (!x t. Lam_bod(Lam x t) = t)
% other functions %
  Ptype_of_DEF
    |- (!s ty. Ptype_of(Const s ty) = ty) /\
      (!x. Ptype_of(Var x) = SND x) /\
      (!t1 t2. Ptype_of(App t1 t2) = HD(TL(Tyop_tyl(Ptype_of t1)))) /\
      (!x t. Ptype_of(Lam x t) = Tyop `fun` [SND x;Ptype_of t])
  Pboolean_DEF  |- !t. Pboolean t = (Ptype_of t = Tyop `bool`[])
  Pwell_typed_DEF
    |- (!Typl Conl s ty. Pwell_typed Typl Conl(Const s ty) =
        mem1 s Conl /\
        Type_OK Typl ty /\
        Type_compat ty(corr1 s Conl) /\
        nocontr(Type_instl ty(corr1 s Conl))) /\
      (!Typl Conl x. Pwell_typed Typl Conl(Var x) = Type_OK Typl(SND x)) /\
      (!Typl Conl t1 t2. Pwell_typed Typl Conl(App t1 t2) =
        Pwell_typed Typl Conl t1 /\ Pwell_typed Typl Conl t2 /\
        Is_Tyop(Ptype_of t1) /\ (Tyop_nam(Ptype_of t1) = `fun`) /\
        (LENGTH(Tyop_tyl(Ptype_of t1)) = 2) /\
        (HD(Tyop_tyl(Ptype_of t1)) = Ptype_of t2)) /\
      (!Typl Conl x t. Pwell_typed Typl Conl(Lam x t) =
        Pwell_typed Typl Conl t /\ Type_OK Typl(SND x))
  Pfree_DEF
    |- (!x s ty. Pfree x(Const s ty) = F) /\
      (!x y. Pfree x(Var y) = (y = x)) /\
      (!x t1 t2. Pfree x(App t1 t2) = Pfree x t1 \/ Pfree x t2) /\
      (!x y t. Pfree x(Lam y t) = ~(y = x) /\ Pfree x t)
  Pbound_DEF
    |- (!x s ty. Pbound x(Const s ty) = F) /\
      (!x y. Pbound x(Var y) = F) /\
      (!x t1 t2. Pbound x(App t1 t2) = Pbound x t1 \/ Pbound x t2) /\
      (!x y t. Pbound x(Lam y t) = (y = x) \/ Pbound x t)
  Poccurs_DEF  |- !x t. Poccurs x t = Pfree x t \/ Pbound x t
  Plnotfree_DEF
    |- (!t. Plnotfree[]t = T) /\
      (!x xl t. Plnotfree(CONS x xl)t = ~Pfree x t /\ Plnotfree xl t)
  Pallnotfree_DEF
    |- (!x. Pallnotfree x{} = T) /\
      (!x tm tms. Pallnotfree x(tm INSERT tms) = ~Pfree x tm /\ Pallnotfree x tms)
```

```
Plallnotfree_DEF
  |- (!xl. Plallnotfree xl{} = T) /\
     (!xl tm tms. Plallnotfree xl(tm INSERT tms) =
       Plnotfree xl tm /\ Plallnotfree xl tms)
Plnotbound_DEF
  |- (!t. Plnotbound[]t = T) /\
     (!x xl t.
       Plnotbound(CONS x xl)t = ~Pbound x t /\ Plnotbound xl t)
Plnotoccurs_DEF
  |- !xl t. Plnotoccurs xl t = Plnotfree xl t /\ Plnotbound xl t
Pty_snotoccurs_DEF
  |- (!s. Pty_snotoccurs s{} = T) /\
     (!s tm tms. Pty_snotoccurs s(tm INSERT tms) =
       ~Pty_occurs s tm /\ Pty_snotoccurs s tms)
Palreplace1_DEF
  |- (!t' vvl tvl s ty.
       Palreplace1 t' vvl tvl(Const s ty) = (t' = Const s ty)) /\
     (!t' vvl tvl x. Palreplace1 t' vvl tvl(Var x) =
       ((Is_Var t' /\ mem1(Var_var t')vvl) =>
        (x = corr1(Var_var t')vvl) |
        (~mem1 x vvl /\ (mem2 x tvl => (t' = corr2 x tvl) | (t' = Var x))))) /\
     (!t' vvl tvl t1 t2. Palreplace1 t' vvl tvl(App t1 t2) =
       Is_App t' /\ Palreplace1(App_fun t')vvl tvl t1 /\
       Palreplace1(App_arg t')vvl tvl t2) /\
     (!t' vvl tvl x t1. Palreplace1 t' vvl tvl(Lam x t1) =
       Is_Lam t' /\ (SND(Lam_var t') = SND x) /\
       Palreplace1(Lam_bod t')(CONS(Lam_var t',x)vvl)tvl t1)
Palreplace_DEF
  |- !t' tvl t. Palreplace t' tvl t = Palreplace1 t'[]tvl t
Palpha_DEF  |- !t' t. Palpha t' t = Palreplace t'[]t
Psubst_triples_DEF
  |- (Psubst_triples[] = T) /\
     (!ttv ttvl. Psubst_triples(CONS ttv ttvl) =
       (Ptype_of(FST ttv) = Ptype_of(FST(SND ttv))) /\
       (Ptype_of(FST ttv) = SND(SND(SND ttv))) /\
       Psubst_triples ttvl)
list13_DEF
  |- (list13[] = []) /\
     (!xyz xyzl. list13(CONS xyz xyzl) = CONS(FST xyz,SND(SND xyz))(list13 xyzl))
Psubst_DEF
  |- !t' ttvl td t. Psubst Typl Conl t' ttvl td t =
      Psubst_triples ttvl /\ Pwell_typed Typl Conl td /\
      Plnotoccurs(MAP(SND o SND)ttvl)t /\
      Palreplace t(MAP SND ttvl)td /\
      Palreplace t'(list13 ttvl)td
Pbeta_DEF  |- !t' x t1 t2. Pbeta t' x t1 t2 = Palreplace t'[t2,x]t1
Pty_occurs_DEF
  |- (!a s ty. Pty_occurs a(Const s ty) = Type_occurs a ty) /\
     (!a x. Pty_occurs a(Var x) = Type_occurs a(SND x)) /\
     (!a t1 t2.
       Pty_occurs a(App t1 t2) = Pty_occurs a t1 \/ Pty_occurs a t2) /\
     (!a x t1.
       Pty_occurs a(Lam x t1) =
       Type_occurs a(SND x) \/ Pty_occurs a t1)
Plty_snotoccurs_DEF
```

```
  |- (!tms. Plty_snotoccurs[]tms = T) /\
     (!h t tml. Plty_snotoccurs(CONS h t)tml =
       Pty_snotoccurs h tml /\ Plty_snotoccurs t tml)
Pnewfree1_DEF
  |- (!t bl s ty. Pnewfree1 t bl(Const s ty) = []) /\
     (!t bl x. Pnewfree1 t bl(Var x) =
       ((mem x bl \/ (FST(Var_var t) = FST x)) => [] | [Var_var t,x])) /\
     (!t bl t1 t2. Pnewfree1 t bl(App t1 t2) =
       APPEND(Pnewfree1(App_fun t)bl t1)(Pnewfree1(App_arg t)bl t2)) /\
     (!t bl x t1. Pnewfree1 t bl(Lam x t1) = Pnewfree1(Lam_bod t)(CONS x bl)t1)
Pnewfree_DEF  |- !t' t. Pnewfree t' t = Pnewfree1 t'[]t
Ptyinst1_DEF
  |- (!t bvl fvl tyl s ty. Ptyinst1 t bvl fvl tyl(Const s ty) =
       (t = Const s(Type_replace tyl ty))) /\
     (!t bvl fvl tyl x. Ptyinst1 t bvl fvl tyl(Var x) =
       ((Is_Var t /\ mem1(Var_var t)bvl) =>
        ((x = corr1(Var_var t)bvl) /\
         (SND(Var_var t) = Type_replace tyl(SND x))) |
        (mem2 x fvl =>
         (t = Var(FST(corr2 x fvl),Type_replace tyl(SND x))) |
         (t = Var(FST x,Type_replace tyl(SND x)))))) /\
     (!t bvl fvl tyl t1 t2. Ptyinst1 t bvl fvl tyl(App t1 t2) =
       Is_App t /\  Ptyinst1(App_fun t)bvl fvl tyl t1 /\
       Ptyinst1(App_arg t)bvl fvl tyl t2) /\
     (!t bvl fvl tyl x t1. Ptyinst1 t bvl fvl tyl(Lam x t1) =
       Is_Lam t /\ ~mem1(Lam_var t)fvl /\
     (SND(Lam_var t) = Type_replace tyl (SND x)) /\
       Ptyinst1(Lam_bod t)(CONS(Lam_var t,x)bvl)fvl tyl t1)
Ptyinst_DEF
  |- !as t' tyl t. Ptyinst as t' tyl t =
      Ptyinst1 t'[](Pnewfree t' t)tyl t /\
      Plallnotfree(MAP SND(Pnewfree t' t))as
```

E.4 The inference theory

Below are the definitions of the `inference` theory.

```
Pseq_assum_DEF  |- !as t. Pseq_assum(Pseq as t) = as
Pseq_concl_DEF  |- !as t. Pseq_concl(Pseq as t) = t
Pseq_boolean_DEF
  |- !as t. Pseq_boolean(Pseq as t) = SEVERY Pboolean as /\ Pboolean t
Pseq_well_typed_DEF
  |- !Typl Conl as t. Pseq_well_typed Typl Conl(Pseq as t) =
      SEVERY(Pwell_typed Typl Conl)as /\ Pwell_typed Typl Conl t
PEQ_DEF
  |- !t1 t2. PEQ t1 t2 =
       App (App (Const `=` (Tyop `fun` [Ptype_of t1;
                                        Tyop `fun`[Ptype_of t1;Tyop `bool`[]]]))
                t1)
           t2
PIMP_DEF
  |- !t1 t2. PIMP t1 t2 =
```

```
       App (App (Const `==>` (Tyop `fun`[Tyop `bool`[];
                                        Tyop `fun`[Tyop `bool`[];Tyop `bool`[]]]))
                t1)
            t2
 Is_EQtm_DEF
   |- !t. Is_EQtm t =
       Is_App t /\ Is_App(App_fun t) /\
       (App_fun(App_fun t) =
        Const `=` (Tyop `fun`[Ptype_of(App_arg t);
                             Tyop `fun`[Ptype_of(App_arg t);Tyop `bool`[]]]))
 PASSUME_DEF
   |- !Typl Conl as t tm. PASSUME Typl Conl(Pseq as t)tm =
       Pwell_typed Typl Conl tm /\
       Pboolean tm /\ (t = tm) /\ (as = {tm})
 PREFL_DEF
   |- !Typl Conl as t tm. PREFL Typl Conl(Pseq as t)tm =
       Pwell_typed Typl Conl tm /\ (as = {}) /\ (t = PEQ tm tm)
 PBETA_CONV_DEF
   |- !Typl Conl as t tm. PBETA_CONV Typl Conl(Pseq as t)tm =
       Pwell_typed Typl Conl tm /\ (as = {}) /\
       Is_App tm /\ Is_Lam(App_fun tm) /\
       (t = PEQ tm(App_arg t)) /\
       Pbeta (App_arg t) (Lam_var(App_fun tm))
             (Lam_bod(App_fun tm)) (App_arg tm)
Psubst_newlist_DEF
   |- (Psubst_newlist[] = []) /\
      (!h t. Psubst_newlist(CONS h t) =
        CONS (App_arg(Pseq_concl(FST h)),
              App_arg(App_fun(Pseq_concl(FST h))),SND h)
             (Psubst_newlist t))
 PSUBST_DEF
   |- !Typl Conl as t thdl td th. PSUBST Typl Conl (Pseq as t)thdl td th =
       Pwell_typed td /\
       EVERY Is_EQtm(MAP Pseq_concl(MAP FST thdl)) /\
       Psubst Typl Conl t(Psubst_newlist thdl)td(Pseq_concl th) /\
       (as = (Pseq_assum th) UNION (LUNION(MAP Pseq_assum(MAP FST thdl))))
 PABS_DEF
   |- !Typl Conl as t tm th. PABS Typl Conl(Pseq as t)tm th =
       Pwell_typed Typl Conl tm /\ Is_EQtm(Pseq_concl th) /\
       Is_Var tm /\ Type_OK Typl(SND(Var_var tm)) /\
       (t = PEQ (Lam (Var_var tm) (App_arg(App_fun(Pseq_concl th))))
               (Lam (Var_var tm) (App_arg(Pseq_concl th)))) /\
       (as = Pseq_assum th) /\ Pallnotfree(Var_var tm)as
 PINST_TYPE_DEF
   |- !Typl as t tyl th. PINST_TYPE Typl(Pseq as t)tyl th =
       EVERY(Type_OK Typl)(MAP FST tyl) /\
       Ptyinst as t tyl(Pseq_concl th) /\
       Plty_snotoccurs(MAP SND tyl)as /\
       (as = Pseq_assum th)
 PDISCH_DEF
   |- !Typl Conl as t tm th. PDISCH Typl Conl (Pseq as t) tm th =
       Pwell_typed Typl Conl tm /\ Pboolean tm /\
       (t = PIMP tm(Pseq_concl th)) /\ (as = (Pseq_assum th) DELETE tm)
 PMP_DEF
   |- !as t th1 th2. PMP(Pseq as t)th1 th2 =
```

```
       (Pseq_concl th1 = PIMP(Pseq_concl th2)t) /\
       (as = (Pseq_assum th1) UNION (Pseq_assum th2))
Inf_concl_DEF
  |- (!s. Inf_concl(AX_inf s) = s) /\
     (!s t. Inf_concl(AS_inf s t) = s) /\
     (!s t. Inf_concl(RE_inf s t) = s) /\
     (!s t. Inf_concl(BE_inf s t) = s) /\
     (!s tdl t s1. Inf_concl(SU_inf s tdl t s1) = s) /\
     (!s t s1. Inf_concl(AB_inf s t s1) = s) /\
     (!s tyl s1. Inf_concl(IN_inf s tyl s1) = s) /\
     (!s t s1. Inf_concl(DI_inf s t s1) = s) /\
     (!s s1 s2. Inf_concl(MP_inf s s1 s2) = s)
Inf_hyps_DEF
  |- (!s. Inf_hyps(AX_inf s) = []) /\
     (!s t. Inf_hyps(AS_inf s t) = []) /\
     (!s t. Inf_hyps(RE_inf s t) = []) /\
     (!s t. Inf_hyps(BE_inf s t) = []) /\
     (!s sdl t s1. Inf_hyps(SU_inf s sdl t s1) = CONS s1(MAP FST sdl)) /\
     (!s t s1. Inf_hyps(AB_inf s t s1) = [s1]) /\
     (!s tyl s1. Inf_hyps(IN_inf s tyl s1) = [s1]) /\
     (!s t s1. Inf_hyps(DI_inf s t s1) = [s1]) /\
     (!s s1 s2. Inf_hyps(MP_inf s s1 s2) = [s1;s2])
OK_Inf_DEF
  |- (!Typl Conl Axil s. OK_Inf Typl Conl Axil(AX_inf s) = mem s Axil) /\
     (!Typl Conl Axil s t.
       OK_Inf Typl Conl Axil(AS_inf s t) = PASSUME Typl Conl s t) /\
     (!Typl Conl Axil s t.
       OK_Inf Typl Conl Axil(RE_inf s t) = PREFL Typl Conl s t) /\
     (!Typl Conl Axil s t.
       OK_Inf Typl Conl Axil(BE_inf s t) = PBETA_CONV Typl Conl s t) /\
     (!Typl Conl Axil s tdl t s1.
       OK_Inf Typl Conl Axil(SU_inf s tdl t s1) = PSUBST Typl Conl s tdl t s1) /\
     (!Typl Conl Axil s t s1.
       OK_Inf Typl Conl Axil(AB_inf s t s1) = PABS Typl Conl s t s1) /\
     (!Typl Conl Axil s tyl s1.
       OK_Inf Typl Conl Axil(IN_inf s tyl s1) = PINST_TYPE Typl s tyl s1) /\
     (!Typl Conl Axil s t s1.
       OK_Inf Typl Conl Axil(DI_inf s t s1) = PDISCH Typl Conl s t s1) /\
     (!Typl Conl Axil s s1 s2.
       OK_Inf Typl Conl Axil(MP_inf s s1 s2) = PMP s s1 s2)
```

E.5 The `proof` theory

This appendix shows the definitions of the `proof` theory.

```
Is_proof_DEF
  |- (!Typl Conl Axil. Is_proof Typl Conl Axil[] = T) /\
     (!Typl Conl Axil i P.
       Is_proof Typl Conl Axil(CONS i P) =
       OK_Inf Typl Conl Axil i /\ lmem(Inf_hyps i)(MAP Inf_concl P) /\
       Is_proof Typl Conl Axil P)
Is_standard_DEF
```

```
|- !Typl Conl Axil.
    Is_standard(Typl,Conl,Axil) =
    EVERY(Pseq_well_typed Typl Conl)Axil /\ EVERY Pseq_boolean Axil /\
    mem1 `fun` Typl /\ (corr1 `fun` Typl = 2) /\
    mem1 `bool` Typl /\ (corr1 `bool` Typl = 0) /\
    mem1 `==>` Conl /\
    (corr1 `==>` Conl = Tyop`fun`[Tyop`bool`[];Tyop`fun`[Tyop`bool`[];Tyop`bool`[]]]) /\
    mem1 `=` Conl /\
    (corr1 `=` Conl = Tyop `fun`[Tyvar `*`;Tyop `fun`[Tyvar `*`;Tyop `bool`[]]]) /\
    mem1 `@` Conl /\
    (corr1 `@` Conl = Tyop `fun`[Tyop `fun`[Tyvar `*`;Tyop `bool`[]];Tyvar `*`])
```

Appendix F

Embedded programming notation

This appendix shows the formal semantics for the embedded programming notation. we first show the definitions for predicate operators and some properties of commands.

```
false_DEF  |- false = (\u. F)
true_DEF  |- true = (\u. T)
not_DEF  |- !p. not p = (\u. ~p u)
and_DEF  |- !p q. p andd q = (\u. p u /\ q u)
or_DEF  |- !p q. p or q = (\u. p u \/ q u)
imp_DEF  |- !p q. p imp q = (\u. p u ==> q u)
implies_DEF  |- !p q. p implies q = (!u. p u ==> q u)
glb_DEF  |- !P. glb P = (\u. !p. P p ==> p u)
lub_DEF  |- !P. lub P = (\u. ?p. P p /\ p u)
fix_DEF  |- !f. fix f = glb(\p. (f p) implies p)
gfix_DEF  |- !f. gfix f = lub(\p. p implies (f p))
monotonic_DEF
  |- !f. monotonic f = (!p q. p implies q ==> (f p) implies (f q))
strict_DEF  |- !c. strict c = (c false = false)
conjunctive_DEF
  |- !c.
      conjunctive c =
      (!P. (?p. P p) ==> (c(glb P) = glb(\q. ?p. P p /\ (q = c p))))
```

Next come the semantic definitions.

```
assert_DEF  |- !p q. assert p q = p andd q
nondass_DEF  |- !P q. nondass P q = (\u. !u'. P u u' ==> q u')
assign_DEF  |- !e q. assign e q = (\u. q(e u))
skip_DEF  |- !q. skip q = q
seq_DEF  |- !c1 c2 q. (c1 seq c2)q = c1(c2 q)
cond_DEF
  |- !g c1 c2 q.
      cond g c1 c2 q = (g andd (c1 q)) or ((not g) andd (c2 q))
```

```
Dch_DEF  |- !C q. Dch C q = glb(\p. ?c. C c /\ (p = c q))
mu_DEF  |- !f. mu f = Dch(\c. monotonic c /\ (f c) ref c)
do_DEF  |- !g c. do g c = mu(\x. cond g(c seq x)skip)
```

The following theorems characterise the function call operator `fcall`:

```
assign_fcall  |- fcall(assign e) = e
fcall_thm
  |- !c f.
      conjunctive c /\
      strict c /\
      (!u0. correct(\u. u = u0)c(\v. v = f u0)) ==>
      (fcall c = f)
fcall_thm_pre
  |- !g c f.
      conjunctive c /\
      strict c /\
      (!u0. g u0 ==> correct(\u. u = u0)c(\v. v = f u0)) ==>
      (!u. g u ==> (fcall c u = f u))
```

Appendix G

Parser/pretty-printer documentation

This is the documentation of the parser and pretty-printer. The text is found in the file Parse/Documentation in the main directory.

```
======================================================================
Documentation for the parser and pretty printer that shallow embeds in
HOL the imperative programming language described in "Verifying
Modular Programs in HOL" by J. von Wright.
======================================================================


Introduction
------------

The parser and pretty printer support a nice concrete syntax for the
programming constructs in HOL developed by J. von Wright. For example,
the following definition:

   |- fPtype_of =
      (assign(\t. (Tyvar ``,[t],[],[],t))) seq
      ((do
        (\(r,l,fl,tyl,t). ~(l = []))
        (assign
         (\(r,l,fl,tyl,t).
           Pterm_CASE
           (\s ty. (ty,[],fl,tyl,t))
           (\x. (SND x,[],fl,tyl,t))
           (\t1 t2. (r,[t1],CONS F fl,tyl,t))
           (\x t1. (r,[t1],CONS T fl,CONS(SND x)tyl,t))
           (HD l)))) seq
       ((do
         (\(r,l,fl,tyl,t). ~(fl = []))
```

```
         (cond
          (\(r,l,fl,tyl,t). HD fl)
          (assign
           (\(r,l,fl,tyl,t). (Tyop `fun`[HD tyl;r],l,TL fl,TL tyl,t)))
          (assign(\(r,l,fl,tyl,t). (HD(TL(Tyop_tyl r)),l,TL fl,tyl,t))))) seq
        (assign(\(r,l,fl,tyl,t). r))))
```

pretty prints as:

```
   |- PROCEDURE fPtype_of (t)
       VAR r = Tyvar ``, l = [t], fl = [], tyl = [];
       WHILE ~(l = []) DO
        r,l,fl,tyl,t :=
         CASE Pterm HD l
           OF Const s ty . ty,[],fl,tyl,t
            | Var x . SND x,[],fl,tyl,t
            | App t1 t2 . r,[t1],CONS F fl,tyl,t
            | Lam x t1 . r,[t1],CONS T fl,CONS(SND x)tyl,t
         END;
       WHILE ~(fl = []) DO
        IF HD fl
          THEN r,fl,tyl := Tyop `fun`[HD tyl;r],TL fl,TL tyl
          ELSE r,fl := HD(TL(Tyop_tyl r)),TL fl;
       RETURN r
```

This syntax can also be used as input.

Loading the parser and pretty printer

The parser and pretty printer are both implemented in two phases, one in Lisp and the other in ML. The Lisp code is in the subdirectory `lisp` and the ML code is in the subdirectory `ml` of the `Parse` directory. Both are automatically loaded when the main theory is loaded:

The actual commands loading the parser and pretty printer are

```
         loadf `Parse/ml/parse`;;
         loadf `Parse/ml/print`;;
```

[Note 2]

Here is an example (assume the parser and pretty printer have just been loaded by steps 1,2 and 3 above). Note that the global variable "g" has to be declared first (such variables are free variables in procedure bodies that are not formal parameters or local variables; see below).

```
   #declare_global_variables ["g:num"];;

   #let Foo13 =
   # "PROCEDURE Foo13 (x:num,y:num)
   #   VAR p, q=0, r;
   #   SKIP;
   #   ASSERT q=0;
```

```
#   p, r := (x+y), 0;
#   IF p=r THEN r:=r+1 ELSE SKIP;
#   WHILE p<r DO SKIP;
#   RETURN (p * r) + g";;
Foo13 =
"PROCEDURE Foo13 (x,y)
  VAR p, q = 0, r;
  SKIP;
  ASSERT q = 0;
  p,r := x + y,0;
  IF p = r THEN r := r + 1 ELSE SKIP; WHILE p < r DO SKIP;
  RETURN (p * r) + g"
  : term
```

Pretty printing can be switched off by setting the flag 'print_lang' to false (its default is true). The function pp takes a truthvalue and sets 'print_lang' to that value (the previous value of the flag is returned).

```
#pp false;;
true : bool

#Foo13;;
"Foo13 g =
 (nondass(\(x,y) (p',q',r',x',y'). (q'=0) /\ (x'=x) /\ (y'=y))) seq
 ((assign(\(p,q,r,x,y). (p,q,r,x,y))) seq
  ((assert(\(p,q,r,x,y). q = 0)) seq
   ((assign(\(p,q,r,x,y). (x + y,q,0,x,y))) seq
    ((cond
      (\(p,q,r,x,y). p = r)
      (assign(\(p,q,r,x,y). (p,q,r + 1,x,y)))
      (assign(\(p,q,r,x,y). (p,q,r,x,y)))) seq
     ((do(\(p,q,r,x,y). p<r)(assign(\(p,q,r,x,y). (p,q,r,x,y)))) seq
      (assign(\(p,q,r,x,y). (p * r) + g)))))))"
: term
```

Type information can be output by setting the flag 'show_types' to true (its default is false). This can be be done using the function show_types in the usual way.

```
#pp true; show_types true;;
false : bool

#Foo13;;
"PROCEDURE Foo13 (x:num,y:num)
  VAR (p:num), (q:num) = 0, (r:num);
  SKIP;
  ASSERT q = 0;
  p,r := x + y,0;
  IF p = r THEN r := r + 1 ELSE SKIP; WHILE p < r DO SKIP;
  RETURN (p * r) + (g:num)"
  : term
```

Defining procedures

The ML function:

```
define_procedure : term -> void
```

Declares a constant representing the procedure by making a definition whose name is the name of the procedure suffixed with `_DEF`. The resulting theorem is bound to this suffixed name in ML too.

```
#new_theory `Foo`;;
() : void

#define_procedure
# "PROCEDURE Foo13 (x:num,y:num)
#  VAR (p:num), (q:num) = 0, (r:num);
#  SKIP;
#  ASSERT q = 0;
#  p,r := x + y,0;
#  IF p = r THEN r := r + 1 ELSE SKIP; WHILE p < r DO SKIP;
#  RETURN (p * r) + (g:num)";;
() : void

Foo13_DEF =
|- !(g:num).
    PROCEDURE Foo13 (x:num,y:num)
     VAR (p:num), (q:num) = 0, (r:num);
     SKIP;
     ASSERT q = 0;
     p,r := x + y,0;
     IF p = r THEN r := r + 1 ELSE SKIP; WHILE p < r DO SKIP;
     RETURN (p * r) + g

#print_theory`-`;;
The Theory Foo
Parents --  proverb
Constants --
  Foo13 ":num -> ((num -> bool) -> (num # num -> bool))"
Definitions --
  Foo13_DEF
    |- !(g:num).
        PROCEDURE Foo13 (x:num,y:num)
         VAR (p:num), (q:num) = 0, (r:num);
         SKIP;
         ASSERT q = 0;
         p,r := x + y,0;
         IF p = r THEN r := r + 1 ELSE SKIP; WHILE p < r DO SKIP;
         RETURN (p * r) + g

******************** Foo ********************

() : void
```

Switching off the pretty-printer reveals the actual theory:

```
#show_types false; pp false;;
true : bool

#print_theory `-`;;
The Theory Foo
Parents --  proverb
Constants --
  Foo13 ":num -> ((num -> bool) -> (num # num -> bool))"
Definitions --
  Foo13_DEF
    |- !g.
        Foo13 g =
        (nondass
         (\(x,y) (p',q',r',x',y'). (q'=0) /\ (x'=x) /\ (y'=y))) seq
        ((assign(\(p,q,r,x,y). (p,q,r,x,y))) seq
         ((assert(\(p,q,r,x,y). q = 0)) seq
          ((assign(\(p,q,r,x,y). (x + y,q,0,x,y))) seq
           ((cond
             (\(p,q,r,x,y). p = r)
             (assign(\(p,q,r,x,y). (p,q,r + 1,x,y)))
             (assign(\(p,q,r,x,y). (p,q,r,x,y)))) seq
            ((do
              (\(p,q,r,x,y). p < r)
              (assign(\(p,q,r,x,y). (p,q,r,x,y)))) seq
             (assign(\(p,q,r,x,y). (p * r) + g)))))))

******************** Foo ********************

() : void
```

CASE-constructs

Case constructs need to be declared using the ML function declare_cases, for example:

```
declare_cases Psequent_CASE;;
declare_cases Pterm_CASE;;
declare_cases Type_CASE;;
declare_cases Inference_CASE;;
```

The argument to declare_cases must be a theorem of the form:

```
 |- !f0 ... fm.
     (!x11 ... x1n_1.
       <name>_CASE f0 ... fm (C1 x11 ... x1n_1) = f0 x11 ... x1n_1) /\
   ...
     (!xp1 ... xpn_p.
       <name>_CASE f0 ... fm (Cp xp1 ... xpn_p) = fm xp1 ... xpn_p)
```

After declare_cases has been applied to such a theorem, the parser and pretty printer will support the following concrete syntax:

```
CASE <name> e OF
```

```
      C1 x11 ... x1n . e1
    | C2 x21 ... x2n . e2

    | Cm xn1 ... xmn . em
  END
```

which parses to:

```
  <name>_CASE
   (\x11 ... x1n. e1)
   (\x21 ... x2n. e2)
         .
         .
         .
   (\xm1 ... xmn. em)
   e
```

The pretty printer inverts this transformation. For example:

```
  #show_types false; declare_cases Pterm_CASE;;
  () : void

  #let fConst_nam =
  # "PROCEDURE fConst_nam (t:Pterm);
  #   RETURN
  #      CASE Pterm t
  #        OF Const s ty . s
  #         | Var x      . ARB
  #         | App t1 t2  . ARB
  #         | Lam x t1   . ARB
  #      END";;
  fConst_nam =
  "PROCEDURE fConst_nam(t)
    RETURN CASE Pterm t
             OF Const s ty . s
              | Var x . ARB
              | App t1 t2 . ARB
              | Lam x t1 . ARB
           END"
  : term
```

switching off pretty printing reveals the actual term bound to fConst_nam:

```
  #pp false;;
  true : bool

  #fConst_nam;;
  "fConst_nam =
   assign(\t. Pterm_CASE(\s ty. s)(\x. ARB)(\t1 t2. ARB)(\x t1. ARB)t)"
  : term
```

Global variables

Global variables need to be declared using the ML function:

```
declare_global_variables : term list -> void
```

This just assigns its argument to the assignable variable:

```
global_variable : term list
```

All free variables in procedure bodies must have been previously been declared (i.e. must occur in global_variables). The order in which they are listed is the order in which they are made parameters to procedures that have global variables.

Parsing

Th first phase of parsing is written in Lisp (it is in the file lisp/parse.l). This outputs an intermediate representation that is further processed in ML (see ml/parse.ml).

The result of the first phase is a term of the form:

```
"ARB `<keyword>` <constituents> : bool"
```

This representation is somewhat arbitrary; it is chosen so as to not require any new constants, these terms are only transient because the second ML phase converts them to the their standard semantic representation.

In the following "--->" means "parses to".

Commands parse as follows:

```
c ::= SKIP                  --->  ARB `SKIP`   T
   |  ASSERT e              --->  ARB `ASSERT` e
   |  v := e                --->  ARB `ASSIGN` (v=e)
   |  c1 ; c2               --->  ARB `SEQ`    (c1,c2)
   |  IF e THEN c1 ELSE c2  --->  ARB `COND`   (e,c1,c2)
   |  WHILE e DO c          --->  ARB `WHILE`  (e,c)
```

Procedures are parsed as:

```
PROCEDURE name (x1,...,xn){;} VAR v1=e1,...,vm=em; c{;} RETURN e
--->
ARB `PROCEDURE` (name = ((x1,...,xn),((v1=e1),...,(vm=em)),c,e))
```

Missing components are represented by "F".

The second phase, which is written in ML, converts the intermediate representation to the correct semantic form.

This phase is done relative to a state vector s, which is a tuple of variables. The translation of command c with respect to state vector s is denoted by {c}s and is specified by:

```
 {ARB `SKIP` T}s    --->  assign \s. s


 {ARB `ASSERT` e}s  --->  assert \s. e


 {ARB `ASSIGN` ((v1,...,vn)=(e1,...,en))}s
 --->
 assign \s. s[e1,...,en/v1,...,vn]

    -- v1,...,vn must occur in s;
    -- s[e1,...,en/v1,...,vn] denotes the result of substituting

       e1,...,en for v1,...,vn in s, respectively.


 {ARB `ASSIGN` ((v1,v2,...,vn)=e)}s
 --->
 assign \s. s[FST e,FST(SND e)...,SND(...(SND e)...)/v1,...,vn]

    -- e not a manifest tuple of length n

    -- v1,...,vn must occur in s;

 {ARB `SEQ` (c1,c2)}s  --->  {c1}s seq {c2}s

 {ARB `COND` (e,c1,c2)}s  --->  cond (\s. e) {c1}s {c2}s

 {ARB `WHILE` (e,c)}s  --->  do (\s. e) {c}s
```

The translation of a procedure:

```
    ARB
     `PROCEDURE`
     (name = ((x1,...,xn),((v1=e1),...,(vm=em)),(c1 seq ... seq cp),e))
```

is

```
   name (z1,...,zq) =
    (assign \(x1,...,xn).(e1,...,em,x1,...,xn)) seq
    {c1}(v1,...,vm,x1,...,xn)                   seq
                                                 .
                                                 .
                                                 .
    {cp}(v1,...,vm,x1,...,xn)                   seq
    (assign \(v1,...,vm,x1,...,xn). e)
```

where z1,...,zq are the free variables in the right hand side.

Various special cases (combinations of no formals, no locals and no body) are also supported. [Note 3].

Unitialized variables "v" are represented by terms of the form "ARB v" in the list of local variable declarations; they can be interspersed with initialized declarations of the form "c=e" (but for expository purposes are listed first below).

The translation of a procedure:

```
    ARB
     `PROCEDURE`
     (name = ((x1,...,xn),
              ((ARB u1),...,(ARB uq),(v1=e1),...,(vm=em)),
              (c1 seq ... seq cp),e))
```

is

```
   name (z1,...,zq) =
    (nondass \(x1,...,xn) (u1',...,uq',v1',...,vm',x1',...,xn').
              (v1'=e1)/\...(vm'=em)/\(x1'=x1)/\.../\(xn'=xn)) seq
    {c1}(u1,...,uq,v1,...,vm,x1,...,xn)                       seq
    {cp}(u1,...,uq,v1,...,vm,x1,...,xn)                       seq
    (assign \(u1,...,uq,v1,...,vm,x1,...,xn). e)
```

Because a state vector is needed, individual commands are only parsed if they occur inside a procedure body. If they don't an error results; for example. [Note 4]

```
   #"x:=1";;
   evaluation failed     ASSIGN must occur in a procedure
```

Pretty printing

The first phase of pretty printing (in ml/print.ml) transforms procedure declarations into an intermediate representation of the form illustrated by the following example:

```
   "PROCEDURE Foo13 (x:num,y:num)
     VAR p, q=0, r;
     SKIP;
     ASSERT q=0;
     p, r := (x+y), 0;
     IF p=r THEN r:=r+1 ELSE SKIP;
     WHILE p<r DO SKIP;
     RETURN (p * r) + con"
```

has the intermediate representation:

```
   "PROCEDURE
    Foo13
    (x,y)
    [ARB p;q = 0;ARB r]
    (SEQ
     (ASSIGN one one)
     (SEQ
      (ASSERT(q = 0))
      (SEQ
       (ASSIGN(p,r)(x + y,0))
```

```
      (SEQ
       (IF(p = r)(ASSIGN r(r + 1))(ASSIGN one one))
       (WHILE(p < r)(ASSIGN one one))))))
   ((p * r) + con)"
```

where "PROCEDURE", "SEQ", "ASSIGN", IF", WHILE" etc are variables of appropriate type. [Note 5]

The intermediate print format of a procedure declaration has the general form:

```
   PROCEDURE
    <name>
    <formal parameters>
    <local variable declarations>
    <body>
    <return expression>
```

Formal parameters, local declarations and the body can all be empty (which is represented by F). The general scheme for the intermediate print format of:

```
   P(z1,...,zq) =
    assign(\(x1,...,xn).(e1,...,em,x1,...,xn)) seq
    c1 seq ... seq cp                          seq
    assign(\(v1,...,vm,x1,...,xn). e)
```

is:

```
   PROCEDURE
    P
    (x1,...,xn)
    [(v1=e1),...,(vm=em)]
    (c1 seq ... seq cp)
    e
```

The formal parameter tuple (x1,...,xn) is empty if the first element of the sequence is of the form: assign(K(e1,...,em)).

The local declaration list [(v1=e1),...,(vm=em)] is empty if there is only one command, or there are at least two commands, the first of which isn't an assignment of one of the two forms:

```
   assign(\(x1,...,xn).(e1,...,em,x1,...,xn))

   assign(K(e1,...,em))
```

The body is empty if there is only one assignment, or if there are two and the first is of one of the two forms shown above.

This intermediate representation is then pretty printed into the final form using the Lisp code in lisp/parse.l.

------------------------------ Notes ------------------------------

[Note 1]
SRIProoVer is the top level directory of J von Wrights semantic theory of programming constructs.

[Note 2]
The file ml/parse.ml first loads the file 'ld_all' that contains the programming constructs and then loads the lisp code for both the parser and the pretty printer; the file ml/print.ml then loads the ml code for pretty printing.

[Note 3]
See comments in the files lisp/parse.l and ml/parse.ml.

[Note 4]
It would be easy to provide a facility for parsing individual commands with respect to a user-supplied state vector.

[Note 5]
This representation is used rather than the intermediate form employed during parsing (with ARB) because it is more mnemonic. It could not be used during parsing because the types of the arguments are needed in order to make up a head variable of the right type, and typechecking happens after parsing, so these types are not known.

Appendix H

Proof checker theory

This appendix shows the definitions and theorems of the `proverb` theory. these definitions and theorems were used to reason about the correctness of the procedures (see following appendices).

```
Definitions --
  Type_CASE
    |- !f1 f2.
        (!s. Type_CASE f1 f2(Tyvar s) = f1 s) /\
        (!s ts. Type_CASE f1 f2(Tyop s ts) = f2 s ts)
  Pterm_CASE
    |- !f0 f1 f2 f3.
        (!s T'. Pterm_CASE f0 f1 f2 f3(Const s T') = f0 s T') /\
        (!p. Pterm_CASE f0 f1 f2 f3(Var p) = f1 p) /\
        (!P1 P2. Pterm_CASE f0 f1 f2 f3(App P1 P2) = f2 P1 P2) /\
        (!p P. Pterm_CASE f0 f1 f2 f3(Lam p P) = f3 p P)
  Inference_CASE
    |- !f0 f1 f2 f3 f4 f5 f6 f7 f8.
        (!P. Inference_CASE f0 f1 f2 f3 f4 f5 f6 f7 f8(AX_inf P) = f0 P) /\
        (!P0 P1.
          Inference_CASE f0 f1 f2 f3 f4 f5 f6 f7 f8(AS_inf P0 P1) =
          f1 P0 P1) /\
        (!P0 P1.
          Inference_CASE f0 f1 f2 f3 f4 f5 f6 f7 f8(RE_inf P0 P1) =
          f2 P0 P1) /\
        (!P0 P1.
          Inference_CASE f0 f1 f2 f3 f4 f5 f6 f7 f8(BE_inf P0 P1) =
          f3 P0 P1) /\
        (!P0 l P1 P2.
          Inference_CASE f0 f1 f2 f3 f4 f5 f6 f7 f8(SU_inf P0 l P1 P2) =
          f4 P0 l P1 P2) /\
        (!P0 P1 P2.
          Inference_CASE f0 f1 f2 f3 f4 f5 f6 f7 f8(AB_inf P0 P1 P2) =
          f5 P0 P1 P2) /\
        (!P0 l P1.
```

```
        Inference_CASE f0 f1 f2 f3 f4 f5 f6 f7 f8(IN_inf P0 l P1) =
        f6 P0 l P1) /\
      (!P0 P1 P2.
        Inference_CASE f0 f1 f2 f3 f4 f5 f6 f7 f8(DI_inf P0 P1 P2) =
        f7 P0 P1 P2) /\
      (!P0 P1 P2.
        Inference_CASE f0 f1 f2 f3 f4 f5 f6 f7 f8(MP_inf P0 P1 P2) =
        f8 P0 P1 P2)
Psequent_CASE  |- !f s P. Psequent_CASE f(Pseq s P) = f s P
LISTSUM_DEF
  |- (LISTSUM[] = 0) /\ (!h t. LISTSUM(CONS h t) = h + (LISTSUM t))
SOME_DEF
  |- (!P. SOME P[] = F) /\
     (!P h t. SOME P(CONS h t) = P h \/ SOME P t)
MAP3_DEF
  |- (!f l2 l3. MAP3 f[]l2 l3 = []) /\
     (!f h t l2 l3.
       MAP3 f(CONS h t)l2 l3 =
       CONS(f h(HD l2)(HD l3))(MAP3 f t(TL l2)(TL l3)))
Pterm_size_DEF
  |- (!s ty. Pterm_size(Const s ty) = SUC 0) /\
     (!x. Pterm_size(Var x) = SUC 0) /\
     (!t1 t2.
       Pterm_size(App t1 t2) = SUC((Pterm_size t1) + (Pterm_size t2))) /\
     (!x t1. Pterm_size(Lam x t1) = SUC(Pterm_size t1))
NTL  |- (!X. NTL 0 X = X) /\ (!n X. NTL(SUC n)X = NTL n(TL X))
Type_com_DEF
  |- (!r. Type_com r[] = r) /\
     (!r n nl.
       Type_com r(CONS n nl) =
       (FST n =>
        Type_com(CONS(SND(SND n))r)nl |
        Type_com
        (CONS
         (Tyop(FST(FST(SND n)))(FIRSTN(SND(FST(SND n)))r))
         (NTL(SND(FST(SND n)))r))
        nl))
Ptype_of_dec_DEF
  |- (!s ty fl tyl. Ptype_of_dec(Const s ty)fl tyl = ty,fl,tyl) /\
     (!x fl tyl. Ptype_of_dec(Var x)fl tyl = SND x,fl,tyl) /\
     (!t1 t2 fl tyl.
       Ptype_of_dec(App t1 t2)fl tyl = Ptype_of_dec t1(CONS F fl)tyl) /\
     (!x t1 fl tyl.
       Ptype_of_dec(Lam x t1)fl tyl =
       Ptype_of_dec t1(CONS T fl)(CONS(SND x)tyl))
Ptype_of_com_DEF
  |- (!ty tyl. Ptype_of_com ty[]tyl = ty) /\
     (!ty f fl tyl.
       Ptype_of_com ty(CONS f fl)tyl =
       (f =>
        Ptype_of_com(Tyop `fun`[HD tyl;ty])fl(TL tyl) |
        Ptype_of_com(HD(TL(Tyop_tyl ty)))fl tyl))
Pterm_compat_DEF
  |- (!t s ty. Pterm_compat t(Const s ty) = Is_Const t) /\
     (!t x. Pterm_compat t(Var x) = Is_Var t) /\
```

```
      (!t t1 t2. Pterm_compat t(App t1 t2) = Is_App t) /\
      (!t x t1. Pterm_compat t(Lam x t1) = Is_Lam t)

Theorems --
  MAP3
    |- (MAP3 f[][][] = []) /\
       (MAP3 f(CONS h1 t1)(CONS h2 t2)(CONS h3 t3) =
        CONS(f h1 h2 h3)(MAP3 f t1 t2 t3))
  Type_size_DEF
    |- (!s. Type_size(Tyvar s) = SUC 0) /\
       (!s ts. Type_size(Tyop s ts) = SUC(LISTSUM(MAP Type_size ts)))
  Type_dec_DEF
    |- (!f s. Type_dec f(Tyvar s) = [T,(``,0),f(Tyvar s)]) /\
       (!f s ts.
         Type_dec f(Tyop s ts) =
         CONS(F,(s,LENGTH ts),Tyvar ``)(LAPPEND(MAP(Type_dec f)ts)))
  LAPPEND_APPEND
    |- !l1 l2. LAPPEND(APPEND l1 l2) = APPEND(LAPPEND l1)(LAPPEND l2)
  EVERY2_APPEND
    |- !l1a l2a.
        (LENGTH l1a = LENGTH l2a) ==>
        (!l1b l2b.
          EVERY2 P(APPEND l1a l1b)(APPEND l2a l2b) =
          EVERY2 P l1a l2a /\ EVERY2 P l1b l2b)
  MAP2_APPEND
    |- !P l1a l2a l1b l2b.
        (LENGTH l1a = LENGTH l2a) /\ (LENGTH l1b = LENGTH l2b) ==>
        (MAP2 P(APPEND l1a l1b)(APPEND l2a l2b) =
         APPEND(MAP2 P l1a l2a)(MAP2 P l1b l2b))
  LISTSUM_APPEND
    |- !l1 l2. LISTSUM(APPEND l1 l2) = (LISTSUM l1) + (LISTSUM l2)
  Type_compat_thm
    |- (!s s'. Type_compat(Tyvar s)(Tyvar s') = T) /\
       (!s s' ts'. Type_compat(Tyvar s)(Tyop s' ts') = F) /\
       (!s ts s'. Type_compat(Tyop s ts)(Tyvar s') = T) /\
       (!s ts s' ts'.
         Type_compat(Tyop s ts)(Tyop s' ts') =
         (s = s') /\
         (LENGTH ts = LENGTH ts') /\
         EVERY2 Type_compat ts ts')
  Type_com_thm
    |- (!r. Type_com r[] = r) /\
       (!r b l s v.
         Type_com r(CONS(b,(s,l),v)nl) =
         (b =>
          Type_com(CONS v r)nl |
          Type_com(CONS(Tyop s(FIRSTN l r))(NTL l r))nl))
  Type_dec_com
    |- (!s ts. f(Tyop s ts) = Tyop s(MAP f ts)) ==>
       (Type_com[](REVERSE(Type_dec f ty)) = [f ty])
  Ptype_of_dec_com
    |- Ptype_of_com
       (FST(Ptype_of_dec t[][]))
       (FST(SND(Ptype_of_dec t[][])))
       (SND(SND(Ptype_of_dec t[][]))) =
```

```
Ptype_of t
```

Appendix I

Proof checker procedures

This appendix shows the complete set of function definitions for the proof checker. Each procedure implements a function. The name of the function can be found by dropping the prefix `f` from the procedure names. For example, `fType_OK` implements the function `Type_OK`. In some cases, lower case letters must be changed to upper case. For example, `fevery` implements `EVERY`.

```
fevery_DEF
  |- PROCEDURE fevery (f,l)
       VAR r = T;
       WHILE ~(l = []) /\ r DO r,l := f(HD l),TL l;
       RETURN r
fmap_DEF
  |- PROCEDURE fmap (f,l)
       VAR r = [];
       WHILE ~(l = []) DO r,l := SNOC(f(HD l))r,TL l;
       RETURN r
fsevery_DEF
  |- PROCEDURE fsevery (f,l)
       VAR r = T;
       WHILE ~(l = EMPTY) /\ r DO r,l := f(CHOICE l),REST l;
       RETURN r
flunion_DEF
  |- PROCEDURE flunion (l)
       VAR r = EMPTY;
       WHILE ~(l = []) DO r,l := r UNION (HD l),TL l;
       RETURN r
fmem_DEF
  |- PROCEDURE fmem (x,l)
       VAR r = F;
       WHILE ~(l = []) /\ ~r DO r,l := (x = HD l),TL l;
       RETURN r
flmem_DEF
  |- PROCEDURE flmem (l1,l2)
```

```
      VAR r = T;
      WHILE ~(l1 = []) /\ r DO r,l1 := mem(HD l1)l2,TL l1;
      RETURN r
fmem1_DEF
  |- PROCEDURE fmem1 (x,l)
      VAR r = F;
      WHILE ~(l = []) /\ ~r DO r,l := (x = FST(HD l)),TL l;
      RETURN r
fcorr1_DEF
  |- PROCEDURE fcorr1 (x,l)
      VAR r;
      WHILE ~(l = []) DO
       IF x = FST(HD l) THEN r,l := SND(HD l),[] ELSE l := TL l;
      RETURN r
fmem2_DEF
  |- PROCEDURE fmem2 (x,l)
      VAR r = F;
      WHILE ~(l = []) /\ ~r DO r,l := (x = SND(HD l)),TL l;
      RETURN r
fcorr2_DEF
  |- PROCEDURE fcorr2 (x,l)
      VAR r;
      WHILE ~(l = []) DO
       IF x = SND(HD l) THEN r,l := FST(HD l),[] ELSE l := TL l;
      RETURN r
fnocontr_DEF
  |- PROCEDURE fnocontr (l)
      VAR r = T;
      WHILE ~(l = []) /\ r DO
       r,l :=
        (~mem2(SND(HD l))(TL l) \/
         (corr2(SND(HD l))(TL l) = FST(HD l))),TL l;
      RETURN r
fIs_Tyvar_DEF
  |- PROCEDURE fIs_Tyvar (ty)
      RETURN CASE Type ty OF Tyvar s . T | Tyop s ts . F END
fIs_Tyop_DEF
  |- PROCEDURE fIs_Tyop (ty)
      RETURN CASE Type ty OF Tyvar s . F | Tyop s ts . T END
fTyvar_nam_DEF
  |- PROCEDURE fTyvar_nam (ty)
      RETURN CASE Type ty OF Tyvar s . s | Tyop s ts . @tm. T END
fTyop_nam_DEF
  |- PROCEDURE fTyop_nam (ty)
      RETURN CASE Type ty OF Tyvar s . @tm. T | Tyop s ts . s END
fTyop_tyl_DEF
  |- PROCEDURE fTyop_tyl (ty)
      RETURN CASE Type ty OF Tyvar s . @tm. T | Tyop s ts . ts END
fType_OK_DEF
  |- !Typl.
      PROCEDURE fType_OK (ty)
       VAR r = T, l = [ty];
       ty := Tyvar ``;
       WHILE ~(l = []) /\ r DO
        (l,ty := TL l,HD l;
```

```
         r,l,ty :=
          CASE Type ty
            OF Tyvar s . r,l,ty
             | Tyop s ts .
                ((mem1 s Typl /\ (LENGTH ts = corr1 s Typl)) =>
                 (T,APPEND ts l,ty) |
                 (F,[],ty))
          END);
       RETURN r
fType_occurs_DEF
  |- PROCEDURE fType_occurs (s',ty)
      VAR r = F, l = [ty];
      WHILE ~(l = []) /\ ~r DO
       r,l,s',ty :=
        CASE Type HD l
          OF Tyvar s . (s' = s),TL l,s',ty
           | Tyop s ts . r,APPEND ts(TL l),s',ty
        END;
      RETURN r
fType_replace_DEF
  |- PROCEDURE fType_replace (tyl,ty)
      VAR r = [ty], xl = [];
      ty := Tyvar ``;
      WHILE ~(r = []) DO
       (r,ty := TL r,HD r;
        r,xl,tyl,ty :=
         CASE Type ty
           OF Tyvar s .
                (mem2 s tyl =>
                 (r,CONS(T,(``,0),corr2 s tyl)xl,tyl,ty) |
                 (r,CONS(T,(``,0),Tyvar s)xl,tyl,ty))
            | Tyop s ts .
                APPEND ts r,CONS(F,(s,LENGTH ts),Tyvar ``)xl,tyl,ty
         END);
      WHILE ~(xl = []) DO
       IF FST(HD xl)
         THEN r,xl := CONS(SND(SND(HD xl)))r,TL xl
         ELSE r,xl :=
               CONS
               (Tyop
                (FST(FST(SND(HD xl))))
                (FIRSTN(SND(FST(SND(HD xl))))r))
               (NTL(SND(FST(SND(HD xl))))r),TL xl;
      RETURN HD r
fType_compat_DEF
  |- PROCEDURE fType_compat (ty2,ty1)
      VAR r = T, l2 = [ty2], l1 = [ty1];
      ty2,ty1 := Tyvar ``,Tyvar ``;
      WHILE ~(l1 = []) /\ r DO
       (l2,l1,ty2,ty1 := TL l2,TL l1,HD l2,HD l1;
        r,l2,l1,ty2,ty1 :=
         CASE Type ty1
           OF Tyvar s . r,l2,l1,ty2,ty1
            | Tyop s ts .
                ((Is_Tyop ty2 /\
```

```
                (Tyop_nam ty2 = s) /\
                (LENGTH(Tyop_tyl ty2) = LENGTH ts)) =>
               (T,APPEND(Tyop_tyl ty2)l2,APPEND ts l1,ty2,ty1) |
               (F,[],[],ty2,ty1))
         END);
      RETURN r
fType_instl_DEF
  |- PROCEDURE fType_instl (ty2,ty1)
      VAR r = [], l2 = [ty2], l1 = [ty1];
      ty2,ty1 := Tyvar ``,Tyvar ``;
      WHILE ~(l1 = []) DO
       (l2,l1,ty2,ty1 := TL l2,TL l1,HD l2,HD l1;
        r,l2,l1,ty2,ty1 :=
         CASE Type ty1
           OF Tyvar s . SNOC(ty2,s)r,l2,l1,ty2,ty1
            | Tyop s ts .
               r,APPEND(Tyop_tyl ty2)l2,APPEND ts l1,ty2,ty1
         END);
      RETURN r
fIs_Const_DEF
  |- PROCEDURE fIs_Const (t)
      RETURN CASE Pterm t
               OF Const s ty . T
                | Var x . F
                | App t1 t2 . F
                | Lam x t1 . F
             END
fIs_Var_DEF
  |- PROCEDURE fIs_Var (t)
      RETURN CASE Pterm t
               OF Const s ty . F
                | Var x . T
                | App t1 t2 . F
                | Lam x t1 . F
             END
fIs_App_DEF
  |- PROCEDURE fIs_App (t)
      RETURN CASE Pterm t
               OF Const s ty . F
                | Var x . F
                | App t1 t2 . T
                | Lam x t1 . F
             END
fIs_Lam_DEF
  |- PROCEDURE fIs_Lam (t)
      RETURN CASE Pterm t
               OF Const s ty . F
                | Var x . F
                | App t1 t2 . F
                | Lam x t1 . T
             END
fConst_nam_DEF
  |- PROCEDURE fConst_nam (t)
      RETURN CASE Pterm t
               OF Const s ty . s
```

```
                | Var x . @tm. T
                | App t1 t2 . @tm. T
                | Lam x t1 . @tm. T
             END
fConst_ty_DEF
  |- PROCEDURE fConst_ty (t)
      RETURN CASE Pterm t
               OF Const s ty . ty
                | Var x . @tm. T
                | App t1 t2 . @tm. T
                | Lam x t1 . @tm. T
             END
fVar_var_DEF
  |- PROCEDURE fVar_var (t)
      RETURN CASE Pterm t
               OF Const s ty . @tm. T
                | Var x . x
                | App t1 t2 . @tm. T
                | Lam x t1 . @tm. T
             END
fApp_fun_DEF
  |- PROCEDURE fApp_fun (t)
      RETURN CASE Pterm t
               OF Const s ty . @tm. T
                | Var x . @tm. T
                | App t1 t2 . t1
                | Lam x t1 . @tm. T
             END
fApp_arg_DEF
  |- PROCEDURE fApp_arg (t)
      RETURN CASE Pterm t
               OF Const s ty . @tm. T
                | Var x . @tm. T
                | App t1 t2 . t2
                | Lam x t1 . @tm. T
             END
fLam_var_DEF
  |- PROCEDURE fLam_var (t)
      RETURN CASE Pterm t
               OF Const s ty . @tm. T
                | Var x . @tm. T
                | App t1 t2 . @tm. T
                | Lam x t1 . x
             END
fLam_bod_DEF
  |- PROCEDURE fLam_bod (t)
      RETURN CASE Pterm t
               OF Const s ty . @tm. T
                | Var x . @tm. T
                | App t1 t2 . @tm. T
                | Lam x t1 . t1
             END
fPtype_of_DEF
  |- PROCEDURE fPtype_of (t)
      VAR r = Tyvar ``, l = [t], fl = [], tyl = [];
```

```
      WHILE ~(l = []) DO
       r,l,fl,tyl,t :=
        CASE Pterm HD l
          OF Const s ty . ty,[],fl,tyl,t
           | Var x . SND x,[],fl,tyl,t
           | App t1 t2 . r,[t1],CONS F fl,tyl,t
           | Lam x t1 . r,[t1],CONS T fl,CONS(SND x)tyl,t
        END;
      WHILE ~(fl = []) DO
       IF HD fl
         THEN r,fl,tyl := Tyop `fun`[HD tyl;r],TL fl,TL tyl
         ELSE r,fl := HD(TL(Tyop_tyl r)),TL fl;
      RETURN r
fPboolean_DEF
  |- PROCEDURE fPboolean (t) RETURN Ptype_of t = Tyop `bool`[]
fPwell_typed_DEF
  |- !Typl Conl.
      PROCEDURE fPwell_typed (t)
       VAR r = T, l = [t];
       t := Var(``,Tyvar ``);
       WHILE ~(l = []) /\ r DO
        (l,t := TL l,HD l;
         r,l,t :=
          CASE Pterm t
            OF Const s ty .
                ((mem1 s Conl /\
                  Type_OK Typl ty /\
                  Type_compat ty(corr1 s Conl) /\
                  nocontr(Type_instl ty(corr1 s Conl))) =>
                 (T,l,t) |
                 (F,[],t))
             | Var x . (Type_OK Typl(SND x) => (T,l,t) | (F,[],t))
             | App t1 t2 .
                ((Is_Tyop(Ptype_of t1) /\
                  (Tyop_nam(Ptype_of t1) = `fun`) /\
                  (LENGTH(Tyop_tyl(Ptype_of t1)) = 2) /\
                  (HD(Tyop_tyl(Ptype_of t1)) = Ptype_of t2)) =>
                 (T,CONS t1(CONS t2 l),t) |
                 (F,[],t))
             | Lam x t1 .
                (Type_OK Typl(SND x) => (T,CONS t1 l,t) | (F,[],t))
          END);
       RETURN r
fPfree_DEF
  |- PROCEDURE fPfree (x,t)
      VAR r = F, l = [t];
      t := Var(``,Tyvar ``);
      WHILE ~(l = []) /\ ~r DO
       (l,t := TL l,HD l;
        r,l,x,t :=
         CASE Pterm t
           OF Const s ty . F,l,x,t
            | Var y . ((y = x) => (T,[],x,t) | (F,l,x,Var y))
            | App t1 t2 . F,CONS t1(CONS t2 l),x,t
            | Lam y t1 . ((y = x) => (F,l,x,t) | (F,CONS t1 l,x,t))
```

```
        END);
      RETURN r
fPbound_DEF
  |- PROCEDURE fPbound (x,t)
      VAR r = F, l = [t];
      t := Var(``,Tyvar ``);
      WHILE ~(l = []) /\ ~r DO
       (l,t := TL l,HD l;
        r,l,x,t :=
         CASE Pterm t
           OF Const s ty . F,l,x,t
            | Var y . F,l,x,t
            | App t1 t2 . F,CONS t1(CONS t2 l),x,t
            | Lam y t1 . ((y = x) => (T,l,x,t) | (F,CONS t1 l,x,t))
         END);
      RETURN r
fPoccurs_DEF
  |- PROCEDURE fPoccurs (x,t) RETURN Pfree x t \/ Pbound x t
fPlnotfree_DEF
  |- PROCEDURE fPlnotfree (l,t)
      VAR r = T;
      WHILE ~(l = []) /\ r DO r,l := (~Pfree(HD l)t),TL l;
      RETURN r
fPlnotbound_DEF
  |- PROCEDURE fPlnotbound (l,t)
      VAR r = T;
      WHILE ~(l = []) /\ r DO r,l := (~Pbound(HD l)t),TL l;
      RETURN r
fPlnotoccurs_DEF
  |- PROCEDURE fPlnotoccurs (xl,t)
      RETURN Plnotfree xl t /\ Plnotbound xl t
fPallnotfree_DEF
  |- PROCEDURE fPallnotfree (x,l)
      VAR r = T;
      WHILE ~(l = EMPTY) /\ r DO r,l := (~Pfree x(CHOICE l)),REST l;
      RETURN r
fPlallnotfree_DEF
  |- PROCEDURE fPlallnotfree (xl,tl)
      VAR r = T;
      WHILE ~(tl = EMPTY) /\ r DO
       r,tl := Plnotfree xl(CHOICE tl),REST tl;
      RETURN r
fPalreplace1_DEF
  |- PROCEDURE fPalreplace1 (t',vvl,tvl,t)
      VAR r = T, l' = [t'], vl = [], l = [t], vvll = [vvl];
      t',t := Var(``,Tyvar ``),Var(``,Tyvar ``);
      WHILE ~(l = []) /\ r DO
       (l',vl,l,vvll,t',t := TL l',HD vvll,TL l,TL vvll,HD l',HD l;
        r,l',vl,l,vvll,t',vvl,tvl,t :=
         CASE Pterm t
           OF Const s ty . (t' = t),l',vl,l,vvll,t',vvl,tvl,t
            | Var x .
               ((Is_Var t' /\ mem1(Var_var t')vl) =>
                (x = corr1(Var_var t')vl) |
                (~mem1 x vl /\
```

```
                (mem2 x tvl => (t' = corr2 x tvl) | (t' = Var x)))),
               l',vl,l,vvll,t',vvl,tvl,t
            | App t1 t2 .
               (Is_App t' =>
                (T,CONS(App_fun t')(CONS(App_arg t')l'),vl,
                 CONS t1(CONS t2 l),CONS vl(CONS vl vvll),t',vvl,tvl,t) |
                (F,l',vl,l,vvll,t',vvl,tvl,t))
            | Lam x t1 .
               ((Is_Lam t' /\ (SND(Lam_var t') = SND x)) =>
                (T,CONS(Lam_bod t')l',vl,CONS t1 l,
                 CONS(CONS(Lam_var t',x)vl)vvll,t',vvl,tvl,t) |
                (F,l',vl,l,vvll,t',vvl,tvl,t))
         END);
      RETURN r
fPalreplace_DEF
  |- PROCEDURE fPalreplace (t',tvl,t) RETURN Palreplace1 t'[]tvl t
fPalpha_DEF  |- PROCEDURE fPalpha (t',t) RETURN Palreplace t'[]t
fPsubst_triples_DEF
  |- PROCEDURE fPsubst_triples (l)
      VAR r = T;
      WHILE ~(l = []) /\ r DO
       r,l :=
        ((Ptype_of(FST(HD l)) = Ptype_of(FST(SND(HD l)))) /\
         (Ptype_of(FST(HD l)) = SND(SND(SND(HD l))))),TL l;
      RETURN r
flist13_DEF
  |- PROCEDURE flist13 (l)
      VAR r = [];
      WHILE ~(l = []) DO r,l := SNOC(FST(HD l),SND(SND(HD l)))r,TL l;
      RETURN r
fPsubst_DEF
  |- !Typl Conl.
      PROCEDURE fPsubst (t',ttvl,td,t)
       RETURN Psubst_triples ttvl /\
              Pwell_typed Typl Conl td /\
              Plnotoccurs(MAP(SND o SND)ttvl)t /\
              Palreplace t(MAP SND ttvl)td /\
              Palreplace t'(list13 ttvl)td
fPbeta_DEF
  |- PROCEDURE fPbeta (t',x,t1,t2) RETURN Palreplace t'[t2,x]t1
fPty_occurs_DEF
  |- PROCEDURE fPty_occurs (a,t)
      VAR r = F, l = [t];
      WHILE ~(l = []) /\ ~r DO
       r,l,a,t :=
        CASE Pterm HD l
          OF Const s ty . Type_occurs a ty,TL l,a,t
           | Var x . Type_occurs a(SND x),TL l,a,t
           | App t1 t2 . r,CONS t1(CONS t2(TL l)),a,t
           | Lam x t1 . Type_occurs a(SND x),CONS t1(TL l),a,t
        END;
      RETURN r
fPty_snotoccurs_DEF
  |- PROCEDURE fPty_snotoccurs (a,l)
      VAR r = T;
```

```
      WHILE ~(l = EMPTY) /\ r DO
       r,l := (~Pty_occurs a(CHOICE l)),REST l;
      RETURN r
fPlty_snotoccurs_DEF
  |- PROCEDURE fPlty_snotoccurs (l,tms)
      VAR r = T;
      WHILE ~(l = []) /\ r DO r,l := Pty_snotoccurs(HD l)tms,TL l;
      RETURN r
fPnewfree1_DEF
  |- PROCEDURE fPnewfree1 (t,bl,t')
      VAR r = [], l = [t], bll = [bl], l' = [t'];
      WHILE ~(l' = []) DO
       r,l,bll,l',t,bl,t' :=
        CASE Pterm HD l'
          OF Const s ty . r,TL l,TL bll,TL l',t,bl,t'
           | Var x .
              ((mem x(HD bll) \/ (FST(Var_var(HD l)) = FST x)) =>
               r |
               SNOC(Var_var(HD l),x)r),TL l,TL bll,TL l',t,bl,t
           | App t1 t2 .
              r,CONS(App_fun(HD l))(CONS(App_arg(HD l))(TL l)),
              CONS(HD bll)(CONS(HD bll)(TL bll)),
              CONS t1(CONS t2(TL l')),t,bl,t'
           | Lam x t1 .
              r,CONS(Lam_bod(HD l))(TL l),
              CONS(CONS x(HD bll))(TL bll),CONS t1(TL l'),t,bl,t'
        END;
      RETURN r
fPnewfree_DEF  |- PROCEDURE fPnewfree (t',t) RETURN Pnewfree1 t'[]t
fPtyinst1_DEF
  |- PROCEDURE fPtyinst1 (t,bvl,fvl,tyl,t')
      VAR r = T, l = [t], bl = [bvl], l' = [t'];
      WHILE ~(l = []) /\ r DO
       r,l,bl,l',t,bvl,fvl,tyl,t' :=
        CASE Pterm HD l'
          OF Const s ty .
              (HD l = Const s(Type_replace tyl ty)),TL l,TL bl,TL l',
              t,bvl,fvl,tyl,t'
           | Var x .
              (mem1(Var_var(HD l))(HD bl) =>
               (Is_Var(HD l) /\
                (x = corr1(Var_var(HD l))(HD bl)) /\
                (SND(Var_var(HD l)) = Type_replace tyl(SND x))) |
               (mem2 x fvl =>
                (HD l = Var(FST(corr2 x fvl),Type_replace tyl(SND x))) |
                (HD l = Var(FST x,Type_replace tyl(SND x))))),TL l,
              TL bl,TL l',t,bvl,fvl,tyl,t'
           | App t1 t2 .
              Is_App(HD l),
              CONS(App_fun(HD l))(CONS(App_arg(HD l))(TL l)),
              CONS(HD bl)(CONS(HD bl)(TL bl)),CONS t1(CONS t2(TL l')),
              t,bvl,fvl,tyl,t'
           | Lam x t1 .
              (Is_Lam(HD l) /\
               ~mem1(Lam_var(HD l))fvl /\
```

```
                (SND(Lam_var(HD l)) = Type_replace tyl(SND x))),
               CONS(Lam_bod(HD l))(TL l),
               CONS(CONS(Lam_var(HD l),x)(HD bl))(TL bl),
               CONS t1(TL l'),t,bvl,fvl,tyl,t'
         END;
       RETURN r
fPtyinst_DEF
  |- PROCEDURE fPtyinst (as,t',tyl,t)
       RETURN Ptyinst1 t'[](Pnewfree t' t)tyl t /\
              Plallnotfree(MAP SND(Pnewfree t' t))as
fPseq_assum_DEF
  |- PROCEDURE fPseq_assum (s)
       RETURN CASE Psequent s OF Pseq as t . as END
fPseq_concl_DEF
  |- PROCEDURE fPseq_concl (s)
       RETURN CASE Psequent s OF Pseq as t . t END
fPseq_boolean_DEF
  |- PROCEDURE fPseq_boolean (s)
       RETURN CASE Psequent s
                OF Pseq as t . SEVERY Pboolean as /\ Pboolean t
              END
fPseq_well_typed_DEF
  |- !Typl Conl.
       PROCEDURE fPseq_well_typed (s)
        RETURN CASE Psequent s
                 OF Pseq as t .
                     SEVERY(Pwell_typed Typl Conl)as /\
                     Pwell_typed Typl Conl t
               END
fPEQ_DEF
  |- PROCEDURE fPEQ (t1,t2)
       RETURN App
              (App
               (Const
                `=`
                (Tyop
                 `fun`
                 [Ptype_of t1;Tyop `fun`[Ptype_of t1;Tyop `bool`[]]]))
               t1)
              t2
fPIMP_DEF
  |- PROCEDURE fPIMP (t1,t2)
       RETURN App
              (App
               (Const
                `==>`
                (Tyop
                 `fun`
                 [Tyop `bool`[];
                  Tyop `fun`[Tyop `bool`[];Tyop `bool`[]]]))
               t1)
              t2
fIs_EQtm_DEF
  |- PROCEDURE fIs_EQtm (t)
       RETURN Is_App t /\
```

```
              Is_App(App_fun t) /\
              (App_fun(App_fun t) =
               Const
               `=`
               (Tyop
                `fun`
                [Ptype_of(App_arg t);
                 Tyop `fun`[Ptype_of(App_arg t);Tyop `bool`[]]]))
fPASSUME_DEF
  |- !Typl Conl.
      PROCEDURE fPASSUME (s,tm)
       RETURN CASE Psequent s
                OF Pseq as t .
                    Pwell_typed Typl Conl tm /\
                    Pboolean tm /\
                    (t = tm) /\
                    (as = tm INSERT EMPTY)
              END
fPREFL_DEF
  |- !Typl Conl.
      PROCEDURE fPREFL (s,tm)
       RETURN CASE Psequent s
                OF Pseq as t .
                    Pwell_typed Typl Conl tm /\
                    (as = EMPTY) /\
                    (t = PEQ tm tm)
              END
fPBETA_CONV_DEF
  |- !Typl Conl.
      PROCEDURE fPBETA_CONV (s,tm)
       RETURN CASE Psequent s
                OF Pseq as t .
                    Pwell_typed Typl Conl tm /\
                    (as = EMPTY) /\
                    Is_App tm /\
                    Is_Lam(App_fun tm) /\
                    (t = PEQ tm(App_arg t)) /\
                    Pbeta
                    (App_arg t)
                    (Lam_var(App_fun tm))
                    (Lam_bod(App_fun tm))
                    (App_arg tm)
              END
fPsubst_newlist_DEF
  |- PROCEDURE fPsubst_newlist (l)
      VAR r = [];
      WHILE ~(l = []) DO
       r,l :=
        ((Is_App(Pseq_concl(FST(HD l))) /\
          Is_App(App_fun(Pseq_concl(FST(HD l))))) =>
         SNOC
         (App_arg(Pseq_concl(FST(HD l))),
          App_arg(App_fun(Pseq_concl(FST(HD l)))),SND(HD l))
         r |
         r),TL l;
```

```
      RETURN r
fPSUBST_DEF
  |- !Typl Conl.
      PROCEDURE fPSUBST (s,thdl,td,th)
       RETURN CASE Psequent s
                OF Pseq as t .
                    Pwell_typed Typl Conl td /\
                    EVERY Is_EQtm(MAP Pseq_concl(MAP FST thdl)) /\
                    Psubst
                    Typl
                    Conl
                    t
                    (Psubst_newlist thdl)
                    td
                    (Pseq_concl th) /\
                    (as =
                     (Pseq_assum th) UNION
                     (LUNION(MAP Pseq_assum(MAP FST thdl))))
              END
fPABS_DEF
  |- !Typl Conl.
      PROCEDURE fPABS (s,tm,th)
       RETURN CASE Psequent s
                OF Pseq as t .
                    Pwell_typed Typl Conl tm /\
                    Is_EQtm(Pseq_concl th) /\
                    Is_Var tm /\
                    Type_OK Typl(SND(Var_var tm)) /\
                    (t =
                     PEQ
                     (Lam
                      (Var_var tm)
                      (App_arg(App_fun(Pseq_concl th))))
                     (Lam(Var_var tm)(App_arg(Pseq_concl th)))) /\
                    (as = Pseq_assum th) /\
                    Pallnotfree(Var_var tm)as
              END
fPINST_TYPE_DEF
  |- !Typl.
      PROCEDURE fPINST_TYPE (s,tyl,th)
       RETURN CASE Psequent s
                OF Pseq as t .
                    EVERY(Type_OK Typl)(MAP FST tyl) /\
                    Ptyinst as t tyl(Pseq_concl th) /\
                    Plty_snotoccurs(MAP SND tyl)as /\
                    (as = Pseq_assum th)
              END
fPDISCH_DEF
  |- !Typl Conl.
      PROCEDURE fPDISCH (s,tm,th)
       RETURN CASE Psequent s
                OF Pseq as t .
                    Pwell_typed Typl Conl tm /\
                    Pboolean tm /\
                    (t = PIMP tm(Pseq_concl th)) /\
```

```
                    (as = (Pseq_assum th) DELETE tm)
              END
fPMP_DEF
  |- PROCEDURE fPMP (s,th1,th2)
      RETURN CASE Psequent s
               OF Pseq as t .
                   (Pseq_concl th1 = PIMP(Pseq_concl th2)t) /\
                   (as = (Pseq_assum th1) UNION (Pseq_assum th2))
             END
fInf_concl_DEF
  |- PROCEDURE fInf_concl (i)
      RETURN CASE Inference i
               OF AX_inf s . s
                | AS_inf s t . s
                | RE_inf s t . s
                | BE_inf s t . s
                | SU_inf s tdl t s1 . s
                | AB_inf s t s1 . s
                | IN_inf s tyl s1 . s
                | DI_inf s t s1 . s
                | MP_inf s s1 s2 . s
             END
fInf_hyps_DEF
  |- PROCEDURE fInf_hyps (i)
      RETURN CASE Inference i
               OF AX_inf s . []
                | AS_inf s t . []
                | RE_inf s t . []
                | BE_inf s t . []
                | SU_inf s sdl t s1 . CONS s1(MAP FST sdl)
                | AB_inf s t s1 . [s1]
                | IN_inf s tyl s1 . [s1]
                | DI_inf s t s1 . [s1]
                | MP_inf s s1 s2 . [s1;s2]
             END
fOK_Inf_DEF
  |- !Typl Conl Axil.
      PROCEDURE fOK_Inf (i)
       VAR r = F;
       r,i :=
        CASE Inference i
          OF AX_inf s . mem s Axil,i
           | AS_inf s t . PASSUME Typl Conl s t,i
           | RE_inf s t . PREFL Typl Conl s t,i
           | BE_inf s t . PBETA_CONV Typl Conl s t,i
           | SU_inf s tdl t s1 . PSUBST Typl Conl s tdl t s1,i
           | AB_inf s t s1 . PABS Typl Conl s t s1,i
           | IN_inf s tyl s1 . PINST_TYPE Typl s tyl s1,i
           | DI_inf s t s1 . PDISCH Typl Conl s t s1,i
           | MP_inf s s1 s2 . PMP s s1 s2,i
        END;
       RETURN r
fIs_proof_DEF
  |- !Typl Conl Axil.
      PROCEDURE fIs_proof (l)
```

```
       VAR r = T;
       WHILE ~(l = []) DO
        IF OK_Inf Typl Conl Axil(HD l) /\
           lmem(Inf_hyps(HD l))(MAP Inf_concl(TL l))
          THEN l := TL l
          ELSE r,l := F,[];
       RETURN r
Pterm_compat_DEF
  |- (!t s ty. Pterm_compat t(Const s ty) = Is_Const t) /\
     (!t x. Pterm_compat t(Var x) = Is_Var t) /\
     (!t t1 t2. Pterm_compat t(App t1 t2) = Is_App t) /\
     (!t x t1. Pterm_compat t(Lam x t1) = Is_Lam t)
```

Appendix J

Correctness theorems for proof checker procedures

Below are the correctness theorems for all the proof checker procedures. By rewriting the procedure definitions using these theorems, we get the final proof checking program where all call to HOL functions have been replaced by calls to procedures.

```
fevery_thm  |- !f l. fcall fevery(f,l) = EVERY f l
fmap_thm  |- !f l. fcall fmap(f,l) = MAP f l
fsevery_thm  |- !f l. fcall fsevery(f,l) = SEVERY f l
fmem_thm  |- !x l. fcall fmem(x,l) = mem x l
flmem_thm  |- !l1 l2. fcall flmem(l1,l2) = lmem l1 l2
fmem1_thm  |- !x l. fcall fmem1(x,l) = mem1 x l
fmem2_thm  |- !x l. fcall fmem2(x,l) = mem2 x l
fcorr1_thm  |- !x l. mem1 x l ==> (fcall fcorr1(x,l) = corr1 x l)
fcorr2_thm  |- !x l. mem2 x l ==> (fcall fcorr2(x,l) = corr2 x l)
flunion_thm  |- fcall flunion = LUNION
fnocontr_thm  |- fcall fnocontr = nocontr
fIs_Tyvar_thm  |- fcall fIs_Tyvar = Is_Tyvar
fIs_Tyop_thm  |- fcall fIs_Tyop = Is_Tyop
fTyvar_nam_thm
  |- !t. Is_Tyvar t ==> (fcall fTyvar_nam t = Tyvar_nam t)
fTyop_nam_thm  |- !t. Is_Tyop t ==> (fcall fTyop_nam t = Tyop_nam t)
fTyop_tyl_thm  |- !t. Is_Tyop t ==> (fcall fTyop_tyl t = Tyop_tyl t)
fType_OK_thm  |- fcall(fType_OK Typl) = Type_OK Typl
fType_replace_thm
  |- !tyl ty. fcall fType_replace(tyl,ty) = Type_replace tyl ty
fType_compat_thm
  |- !ty2 ty1. fcall fType_compat(ty2,ty1) = Type_compat ty2 ty1
fType_occurs_thm
  |- !s' ty. fcall fType_occurs(s',ty) = Type_occurs s' ty
fType_instl_thm
  |- !ty1 ty2.
```

```
      Type_compat ty2 ty1 ==>
      (fcall fType_instl(ty2,ty1) = Type_instl ty2 ty1)
fIs_Const_thm  |- fcall fIs_Const = Is_Const
fIs_Var_thm  |- fcall fIs_Var = Is_Var
fIs_App_thm  |- fcall fIs_App = Is_App
fIs_Lam_thm  |- fcall fIs_Lam = Is_Lam
fConst_nam_thm
  |- !t. Is_Const t ==> (fcall fConst_nam t = Const_nam t)
fConst_ty_thm  |- !t. Is_Const t ==> (fcall fConst_ty t = Const_ty t)
fVar_var_thm  |- !t. Is_Var t ==> (fcall fVar_var t = Var_var t)
fApp_fun_thm  |- !t. Is_App t ==> (fcall fApp_fun t = App_fun t)
fApp_arg_thm  |- !t. Is_App t ==> (fcall fApp_arg t = App_arg t)
fLam_var_thm  |- !t. Is_Lam t ==> (fcall fLam_var t = Lam_var t)
fLam_bod_thm  |- !t. Is_Lam t ==> (fcall fLam_bod t = Lam_bod t)
fPtype_of_thm  |- fcall fPtype_of = Ptype_of
fPboolean_thm  |- fcall fPboolean = Pboolean
fPwell_typed_thm
  |- fcall(fPwell_typed(Typl,Conl)) = Pwell_typed Typl Conl
fPfree_thm  |- !x t. fcall fPfree(x,t) = Pfree x t
fPbound_thm  |- !x t. fcall fPbound(x,t) = Pbound x t
fPoccurs_thm  |- !x t. fcall fPoccurs(x,t) = Poccurs x t
fPlnotfree_thm  |- !l t. fcall fPlnotfree(l,t) = Plnotfree l t
fPlnotbound_thm  |- !l t. fcall fPlnotbound(l,t) = Plnotbound l t
fPlnotoccurs_thm  |- !x t. fcall fPlnotoccurs(x,t) = Plnotoccurs x t
fPallnotfree_thm  |- !x l. fcall fPallnotfree(x,l) = Pallnotfree x l
fPlallnotfree_thm
  |- !xl tl. fcall fPlallnotfree(xl,tl) = Plallnotfree xl tl
fPalreplace1_thm
  |- !t' vvl tvl t.
      fcall fPalreplace1(t',vvl,tvl,t) = Palreplace1 t' vvl tvl t
fPalreplace_thm
  |- !t' tvl t. fcall fPalreplace(t',tvl,t) = Palreplace t' tvl t
fPalpha_thm  |- !t' t. fcall fPalpha(t',t) = Palpha t' t
fPsubst_triples_thm  |- fcall fPsubst_triples = Psubst_triples
flist13_thm  |- fcall flist13 = list13
fPsubst_thm
  |- !Typl Conl t' ttvl td t.
      fcall(fPsubst(Typl,Conl))(t',ttvl,td,t) =
      Psubst Typl Conl t' ttvl td t
fPbeta_thm
  |- !t' x t1 t2. fcall fPbeta(t',x,t1,t2) = Pbeta t' x t1 t2
fPty_occurs_thm  |- !a l. fcall fPty_occurs(a,l) = Pty_occurs a l
fPty_snotoccurs_thm
  |- !a tms. fcall fPty_snotoccurs(a,tms) = Pty_snotoccurs a tms
fPlty_snotoccurs_thm
  |- !al tms. fcall fPlty_snotoccurs(al,tms) = Plty_snotoccurs al tms
fPnewfree1_thm
  |- !t bl t'. fcall fPnewfree1(t,bl,t') = Pnewfree1 t bl t'
fPnewfree_thm  |- !t' t. fcall fPnewfree(t',t) = Pnewfree t' t
fPtyinst1_thm
  |- !t bvl fvl tyl t'.
      fcall fPtyinst1(t,bvl,fvl,tyl,t') = Ptyinst1 t bvl fvl tyl t'
fPtyinst_thm
  |- !as t' tyl t. fcall fPtyinst(as,t',tyl,t) = Ptyinst as t' tyl t
fPseq_assum_thm  |- fcall fPseq_assum = Pseq_assum
```

```
fPseq_concl_thm  |- fcall fPseq_concl = Pseq_concl
fPseq_boolean_thm  |- fcall fPseq_boolean = Pseq_boolean
fPseq_well_typed_thm
  |- fcall(fPseq_well_typed(Typl,Conl)) = Pseq_well_typed Typl Conl
fPEQ_thm  |- !t1 t2. fcall fPEQ(t1,t2) = PEQ t1 t2
fPIMP_thm  |- !t1 t2. fcall fPIMP(t1,t2) = PIMP t1 t2
fIs_EQtm_thm  |- fcall fIs_EQtm = Is_EQtm
fPASSUME_thm
  |- !Typl Conl s tm.
      fcall(fPASSUME(Typl,Conl))(s,tm) = PASSUME Typl Conl s tm
fPREFL_thm
  |- !Typl Conl s tm.
      fcall(fPREFL(Typl,Conl))(s,tm) = PREFL Typl Conl s tm
fPBETA_CONV_thm
  |- !Typl Conl s tm.
      fcall(fPBETA_CONV(Typl,Conl))(s,tm) = PBETA_CONV Typl Conl s tm
fPsubst_newlist_thm
  |- !l.
      EVERY Is_App(MAP Pseq_concl(MAP FST l)) /\
      EVERY Is_App(MAP App_fun(MAP Pseq_concl(MAP FST l))) ==>
      (fcall fPsubst_newlist l = Psubst_newlist l)
fPSUBST_thm
  |- !Typl Conl s thdl td th.
      fcall(fPSUBST(Typl,Conl))(s,thdl,td,th) =
      PSUBST Typl Conl s thdl td th
fPABS_thm
  |- !Typl s tm th.
      fcall(fPABS(Typl,Conl))(s,tm,th) = PABS Typl Conl s tm th
fPINST_TYPE_thm
  |- !Typl s tyl th.
      fcall(fPINST_TYPE Typl)(s,tyl,th) = PINST_TYPE Typl s tyl th
fPDISCH_thm
  |- !Typl Conl s tm th.
      fcall(fPDISCH(Typl,Conl))(s,tm,th) = PDISCH Typl Conl s tm th
fPMP_thm  |- !s th1 th2. fcall fPMP(s,th1,th2) = PMP s th1 th2
fInf_concl_thm  |- fcall fInf_concl = Inf_concl
fInf_hyps_thm  |- fcall fInf_hyps = Inf_hyps
fOK_Inf_thm
  |- !Typl Conl Axil i.
      fcall(fOK_Inf(Typl,Conl,Axil))i = OK_Inf Typl Conl Axil i
fIs_proof_thm
  |- !Typl Conl Axil t.
      fcall(fIs_proof(Typl,Conl,Axil))t = Is_proof Typl Conl Axil t
```

Appendix K

Proof checker implementation

Below is the code for the proof checker implementation in an imperative fragment of ML.

```
% --- general stuff (proofaux) --- %

letrec firstn n l =
  if n = 0 then []
  else (hd l).(firstn (n-1) (tl l));;
letrec ntl n l =
  if n = 0 then l
  else ntl (n-1) (tl l);;
letrec every p l =
  if l = [] then true
  else
    let h.t = l in
    p h & every p t;;
let mem (X,L) =
   letref r,x,l=false,X,L in
   while (not l=[] & not r) do
       (r,l:=(x=hd l),tl l);
   r;;

let lmem (L1,L2) =
   letref r,l1,l2 = true,L1,L2 in
   while not l1=[] & r do
     (r,l1 := mem(hd l1,l2),tl l1);
   r;;

let mem1 (X,L) =
   letref r,x,l=false,X,L in
```

```
   while (not l=[] & not r) do
       (r,l:=(x=fst(hd l)),tl l);
   r;;

let corr1 (X,L,zz) =
   letref r,x,l=zz,X,L in
   while (not l=[]) do
       (r,l:= if x=fst(hd l) then snd(hd l),[]
              else r,tl l);
   r;;

let mem2 (X,L) =
   letref r,x,l=false,X,L in
   while (not l=[] & not r) do
       (r,l:=(x=snd(hd l)),tl l);
   r;;

let corr2 (X,L,zz) =
   letref r,x,l=zz,X,L in
   while (not l=[]) do
       (r,l:= if x=snd(hd l) then fst(hd l),[]
              else r,tl l);
   r;;

let nocontr (L,zz) =
   letref r,l = true,L in
   while (not l=[] & r) do
     (r,l := (not(mem2(snd(hd l),tl l))) or
               (corr2(snd(hd l),tl l,zz) = fst(hd l)),
              tl l);
   r;;


% --- sets as lists --- %

letrec subset (s1,s2) =
  if s1 = [] then true
  else let x.r = s1 in
       mem(x,s2) & subset(r,s2);;

let sameset(s1,s2) = subset(s1,s2) & subset(s2,s1);;

letrec delete(s,x) =
  if s = [] then []
  else let y.r = s in
       if x=y then delete(r,x)
       else y.(delete(r,x));;

let insert(x,s) = x.s;;

let union(s1,s2) = s1 @ s2;;

letrec lunion sl =
  if sl = [] then []
  else let h.t = sl in h @ (lunion t);;
```

```
let severy(p,l) = every p l;;


% --- Type --- %

rectype Type = Tyvar of string | Tyop of string # Type list;;
letref Typl = [(`fun`,2);(`bool`,0)]:(string # int)list;;
letref Conl = []:(string # Type)list;;

let Is_Tyvar = fun (Tyvar s) . true | (Tyop (s,ts)) . false;;
let Is_Tyop = fun (Tyvar s) . false | (Tyop (s,ts)) . true;;

let Tyvar_nam = fun (Tyvar s) . s ;;
let Tyop_nam = fun Tyop (s,ts) . s;;
let Tyop_tyl = fun Tyop (s,ts) . ts;;

let Type_OK Ty =
   letref r,l,ty = true,[Ty],Tyvar`` in
   while (not l=[] & r) do
     (ty,l:= hd l,tl l;
      r,l := (fun Tyvar s    . (r,l)
               |  Tyop(s,ts) . if (mem1(s,Typl) & (length ts = corr1(s,Typl,0)))
                               then (true,append ts l)
                               else (false,[])
             )ty
     );
   r;;

let Type_occurs (S',Ty) =
   letref r,l,s',ty = false,[Ty],S',Ty in
   while (not (l=[]) & not r) do
     r,l := (fun Tyvar s    . (s'=s),tl l
              |  Tyop(s,ts) . r,append ts (tl l)
            )ty;
   r;;

let Type_compat (Ty2,Ty1) =
   letref r,ty2,l2,ty1,l1 = true,Tyvar``,[Ty2],Tyvar``,[Ty1] in
   while (not l1=[] & r) do
     (ty2,l2,ty1,l1 := hd l2,tl l2,hd l1,tl l1;
      r,l2,l1 := (fun Tyvar s    . (r,l2,l1)
                   |  Tyop(s,ts) .
                        if Is_Tyop ty2 & (Tyop_nam ty2 = s) &
                           (length(Tyop_tyl ty2) = length ts)
                        then (true,append(Tyop_tyl ty2)l2,append ts l1)
                        else (false,[],[])
                 )ty1
     );
   r;;

let Type_instl (Ty2,Ty1) =
   letref r,ty2,l2,ty1,l1 = []:(Type#string)list,Tyvar``,[Ty2],Tyvar``,[Ty1] in
   while not l1=[] do
     (ty2,l2,ty1,l1 := hd l2,tl l2,hd l1,tl l1;
```

```
      r,l2,l1 := (fun Tyvar s    . (r@[ty2,s],l2,l1)
                   |  Tyop(s,ts) . (r,append(Tyop_tyl ty2)l2,append ts l1)
                )ty1
     );
   r;;

let Type_replace (Tyl,Ty) =
   letref r,tyl,ty,xl = [Ty],Tyl,Tyvar``,[] in
   while not r=[] do
     (r,ty:= tl r,hd r;
      r,xl := (fun Tyvar s    . if mem2(s,tyl)
                                 then (r,(true,(``,0),corr2(s,tyl,Tyvar``)).xl)
                                 else (r,(true,(``,0),Tyvar s).xl)
                |  Tyop(s,ts) . (append ts r,(false,(s,length ts),Tyvar``).xl)
              )ty
     );
   while not xl = [] do
     (if fst(hd xl) then
        (r,xl := (snd(snd(hd xl)).r,tl xl) ; ())
      else
        (r,xl := (Tyop(fst(fst(snd(hd xl))),firstn(snd(fst(snd(hd xl))))r)
                 .ntl(snd(fst(snd(hd xl))))r,
                 tl xl) ; ())
     );
   hd r;;


% --- Pterm --- %

rectype Pterm = Const of string # Type
              | Var of string # Type
              | App of Pterm # Pterm
              | Lam of (string # Type) # Pterm;;

let Is_Const =
  fun Const(s,ty) . true
   |  Var y       . false
   |  App(t1,t2)  . false
   |  Lam(y,t1)   . false;;
let Is_Var =
  fun Const(s,ty) . false
   |  Var y       . true
   |  App(t1,t2)  . false
   |  Lam(y,t1)   . false;;
let Is_App =
  fun Const(s,ty) . false
   |  Var y       . false
   |  App(t1,t2)  . true
   |  Lam(y,t1)   . false;;
let Is_Lam =
  fun Const(s,ty) . false
   |  Var y       . false
   |  App(t1,t2)  . false
   |  Lam(y,t1)   . true;;
```

```
let Const_nam = fun Const(s,ty) . s;;
let Const_ty = fun Const(s,ty) . ty;;
let Var_var = fun Var y . y;;
let App_fun = fun App(t1,t2)  . t1;;
let App_arg = fun App(t1,t2)  . t2;;
let Lam_var = fun Lam(y,t1)  . y;;
let Lam_bod = fun Lam(y,t1)  . t1;;

let Ptype_of (T) =
   letref r,l,fl,tyl,t = Tyvar``,[T],[]:(bool)list,[]:(Type)list,T in
   while not l=[] do
     (r,l,fl,tyl := (fun Const(s,ty) . (ty,[],fl,tyl)
                     |  Var x       . (snd x,[],fl,tyl)
                     |  App(t1,t2)  . (r,[t1],false.fl,tyl)
                     |  Lam(x,t1)   . (r,[t1],true.fl,(snd x).tyl)
                    )(hd l)
     );
   while not (fl = []) do
     (if (hd fl) then
        (r,fl,tyl := Tyop(`fun`,[hd tyl;r]),tl fl,tl tyl)
      else
        (r,fl,tyl := hd(tl(Tyop_tyl r)),tl fl,tyl)
     );
   r;;

let Pboolean t = (Ptype_of t = Tyop(`bool`,[]));;

let Pwell_typed T =
  letref r,l,t = true,[T],Var(``,Tyvar``) in
  while (not l=[]) & r do
    (l,t := tl l,hd l;
     r,l := (fun Const(s,ty) .
                   if mem1(s,Conl) & Type_OK ty &
                      Type_compat(ty,corr1(s,Conl,Tyvar``)) &
                      nocontr(Type_instl(ty,corr1(s,Conl,Tyvar``)),Tyvar``)
                   then (true,l)
                   else (false,[])
              |  Var x .
                   if Type_OK (snd x)
                   then (true,l)
                   else (false,[])
              |  App(t1,t2) .
                   if Is_Tyop(Ptype_of t1) &
                      (Tyop_nam(Ptype_of t1) = `fun`) &
                      (length(Tyop_tyl(Ptype_of t1)) = 2) &
                      (hd(Tyop_tyl(Ptype_of t1)) = Ptype_of t2)
                   then (true,t1.t2.l)
                   else (false,[])
              |  Lam(x,t1) .
                   if Type_OK (snd x)
                   then (true,t1.l)
                   else (false,[])
            )t
    );
  r;;
```

```
let Pfree (X,T) =
  letref r,l,x,t = false,[T],X,T in
  t := Var(``,Tyvar``);
  while (not l=[]) & (not r) do
    (l,t := tl l,hd l;
     r,l := (fun Const(s,ty) . false,l
              |  Var y       . if y=x then (true,[]) else (false,l)
              |  App(t1,t2)  . false,t1.t2.l
              |  Lam(y,t1)   . if y=x then (false,l) else (false,t1.l)
            )t
    );
  r;;

let Pbound (X,T) =
  letref r,l,x,t = false,[T],X,T in
  t := Var(``,Tyvar``);
  while (not l=[]) & (not r) do
    (l,t := tl l,hd l;
     r,l := (fun Const(s,ty) . false,l
              |  Var y       . false,l
              |  App(t1,t2)  . false,t1.t2.l
              |  Lam(y,t1)   . if y=x then (true,l) else (false,t1.l)
            )t
    );
  r;;

let Poccurs(X,T) = Pfree(X,T) or Pbound(X,T);;

let Plnotfree (L,T) =
  letref r,l,t = true,L,T in
  while (not l=[]) & r do
    r,l := not Pfree(hd l,t),tl l;
  r;;

let Plnotbound (L,T) =
  letref r,l,t = true,L,T in
  while (not l=[]) & r do
    r,l := not Pbound(hd l,t),tl l;
  r;;

let Plnotoccurs(X,T) = Plnotfree(X,T) & Plnotbound(X,T);;

let Pallnotfree (X,L) =
  letref r,x,l = true,X,L in
  while (not l=[]) & r do
    r,l := not Pfree(x,hd l),tl l;
  r;;

let Plallnotfree (XL,TML) =
  letref r,xl,tml = true,XL,TML in
  while (not tml=[]) & r do
    r,tml := not Plnotfree(xl,hd tml),tl tml;
  r;;
```

```
let Palreplace1 (T',VVL,TVL,T) =
  letref r,l',vl:((string#Type)#(string#Type))list,l,vvll,t',vvl,tvl,t
         = true,[T'],[],[T],[VVL],T',VVL,TVL,T in
  t',t := Var(``,Tyvar``),Var(``,Tyvar``);
  while (not l=[]) & r do
    (l',vl,l,vvll,t',t := tl l',hd vvll,tl l,tl vvll,hd l',hd l;
     r,l',l,vvll :=
       (fun Const(s,ty) . (t'=t),l',l,vvll
         |  Var x       . (if (Is_Var t' & mem1((Var_var t'),vl))
                              then (x = corr1((Var_var t'),vl,(``,Tyvar``)))
                              else (not mem1(x,vl) &
                                     if mem2(x,tvl)
                                     then (t' = corr2(x,tvl,Var(``,Tyvar``)))
                                     else (t' = Var x))
                             ),l',l,vvll
         |  App(t1,t2)  . if Is_App t'
                          then (true,(App_fun t').(App_arg t').l',
                                t1.t2.l,vl.vl.vvll)
                          else (false,l',l,vvll)
         |  Lam(x,t1)   . if Is_Lam t' & (snd(Lam_var t')=snd x)
                          then (true,(Lam_bod t').l',t1.l,((Lam_var t',x).vl).vvll)
                          else (false,l',l,vvll)
       )t
    );
  r;;

let Palreplace (T',TVL,T) = Palreplace1(T',[],TVL,T);;

let Palpha (T',T) = Palreplace(T',[],T);;

let Psubst_triples L =
  letref r,l = true,L in
  while (not l=[]) & r do
    r,l := ((Ptype_of(fst(hd l)) = Ptype_of(fst(snd(hd l)))) &
            (Ptype_of(fst(hd l)) = snd(snd(snd(hd l))))),tl l;
  r;;

let list13 L =
  letref r,l = [],L in
  while (not l=[]) do
    r,l := r @ [(fst(hd l),snd(snd(hd l)))],tl l;
  r;;

let Psubst (T',TTVL,TD,T) =
  Psubst_triples TTVL & Pwell_typed TD &
  Plnotoccurs((map(snd o snd)TTVL),T) &
  Palreplace(T,(map snd TTVL),TD) &
  Palreplace(T',(list13 TTVL),TD);;

let Pbeta (T',X,T1,T2) = Palreplace(T',[T2,X],T1);;

let Pty_occurs (A,T) =
  letref r,l,a,t = false,[],A,T in
  while (not l=[]) & not r do
    r,l := (fun Const(s,ty) . Type_occurs(a,ty),tl l
```

```
             |  Var x       . Type_occurs(a,snd x),tl l
             |  App(t1,t2)  . r,t1.t2.(tl l)
             |  Lam(y,t1)   . Type_occurs(a,snd y),t1.(tl l)
           )(hd l);
  r;;

let Pty_snotoccurs (A,L) =
  letref r,a,l = true,A,L in
  while (not l=[]) & r do
    r,l := not Pty_occurs(a,hd l),tl l;
  r;;

let Plty_snotoccurs (L,TMS) =
  letref r,l,tms = true,L,TMS in
  while (not l=[]) & r do
    r,l := Pty_snotoccurs(hd l,tms),tl l;
  r;;

let Pnewfree1 (T,BL,T') =
  letref r,l,bll,l',t,bl,t'
         = [],[T'],[BL],[T],T,BL,T' in
  t',t := Var(``,Tyvar``),Var(``,Tyvar``);
  while not l=[] do
    r,l,bll,l' :=
       (fun Const(s,ty) . r,tl l,tl bll,tl l'
         |  Var x       . (if mem(x,hd bll) or (fst(Var_var(hd l)) = fst x)
                             then r
                             else r @ [(Var_var(hd l),x)]
                          ),tl l,tl bll,tl l'
         |  App(t1,t2)  . r,App_fun(hd l).App_arg(hd l).(tl l),
                          (hd bll).(hd bll).(tl bll),t1.t2.(tl l')
         |  Lam(x,t1)   . r,Lam_bod(hd l).(tl l),
                          (x.hd bll).(tl bll),t1.(tl l')
       )(hd l');
  r;;

let Pnewfree (T',T) = Pnewfree1(T',[],T);;

let Ptyinst1 (T,BVL,FVL,TYL,T') =
  letref r,l,bl,l',t,bvl,fvl,tyl,t'
         = true,[T],[BVL],[T'],T,BVL,FVL,TYL,T' in
  while not l=[] & r do
    r,l,bl,l' :=
       (fun Const(s,ty) . (hd l = Const(s,(Type_replace(tyl,ty)))),
                          tl l,tl bl,tl l'
         |  Var x       . (if mem1(Var_var(hd l),hd bl)
                           then Is_Var(hd l) & (x=corr1(Var_var(hd l),hd bl,(``,Tyvar``))) &
                                (snd(Var_var(hd l)) = Type_replace(tyl,snd x))
                           else if mem2(x,fvl)
                                then (hd l = Var(fst(corr2(x,fvl,(``,Tyvar``))),
                                                 Type_replace(tyl,snd x)))
                                else (hd l = Var(fst x,Type_replace(tyl,snd x))) ),
                          tl l,tl bl,tl l'
         |  App(t1,t2)  . Is_App(hd l),App_fun(hd l).App_arg(hd l).(tl l),
                          (hd bl).(hd bl).(tl bl),t1.t2.(tl l')
```

```
        |  Lam(x,t1)   . Is_Lam(hd l) & not mem1(Lam_var(hd l),fvl) &
                          (snd(Lam_var(hd l)) = Type_replace(tyl,snd x)),
                         Lam_bod(hd l).(tl l),
                         ((Lam_var(hd l),x).hd bl).(tl bl),t1.(tl l')
       )(hd l');
  r;;

let Ptyinst (AS,T',TYL,T) =
  Ptyinst1(T',[],Pnewfree(T',T),TYL,T) &
  Plallnotfree(map snd(Pnewfree(T',T)),AS);;



% --- Psequent --- %

rectype Psequent = Pseq of Pterm list # Pterm;;
letref Axil = []:Psequent list;;

let Pseq_assum = fun Pseq(as,t) . as;;
let Pseq_concl = fun Pseq(as,t) . t;;

let Pseq_boolean = fun Pseq(as,t) . every Pboolean as & Pboolean t;;

let Pseq_well_typed = fun Pseq(as,t) . every Pwell_typed as & Pwell_typed t;;

let PEQ (T1,T2) =
  App(App(Const(`=`,Tyop(`fun`,[Ptype_of T1;
                                Tyop(`fun`,[Ptype_of T1;Tyop(`bool`,[])])])),
          T1),
      T2);;

let PIMP (T1,T2) =
  App(App(Const(`==>`,
           Tyop(`fun`,[Tyop(`bool`,[]);
                       Tyop(`fun`,[Tyop(`bool`,[]);Tyop(`bool`,[])])])),
          T1),
      T2);;

let Is_EQtm T =
  Is_App T & Is_App(App_fun T) &
  App_fun(App_fun T) =
    Const(`=`,Tyop(`fun`,[Ptype_of(App_arg T);
                         Tyop(`fun`,[Ptype_of(App_arg T);Tyop(`bool`,[])])]));;

let PASSUME = fun (Pseq(as,t),tm) .
    Pwell_typed tm & Pboolean tm & (t = tm) & (as = [tm]);;

let PREFL = fun (Pseq(as,t),tm) .
    Pwell_typed tm & (as=[]) & (t = PEQ(tm,tm));;

let PBETA_CONV = fun (Pseq(as,t),tm) .
    Pwell_typed tm & (as = []) & Is_App tm & Is_Lam(App_fun tm) &
    (t = PEQ(tm,(App_arg t))) &
    Pbeta(App_arg t,Lam_var(App_fun tm),Lam_bod(App_fun tm),App_arg tm);;
```

```
let Psubst_newlist (L) =
  letref r,l = [],L in
  while (not l=[]) do
    r,l := (if Is_App(Pseq_concl(fst(hd l)))
               & Is_App(App_fun(Pseq_concl(fst(hd l))))
            then r @ [App_arg(Pseq_concl(fst(hd l))),
                 App_arg(App_fun(Pseq_concl(fst(hd l)))),
                 snd(hd l)]
            else r),
           tl l;
  r;;

let PSUBST = fun (Pseq(as,t),tdl,tm,s1) .
    Pwell_typed tm &
    every Is_EQtm (map Pseq_concl(map fst tdl)) &
    Psubst(t,Psubst_newlist tdl,tm,Pseq_concl s1) &
    sameset(as,(Pseq_assum s1) @ (flat(map Pseq_assum (map fst tdl))));;

let PABS = fun (Pseq(as,t),tm,s1) .
    Pwell_typed tm & Is_EQtm(Pseq_concl s1) & Is_Var tm &
    Type_OK(snd(Var_var tm)) &
    (t = PEQ(Lam(Var_var tm,App_arg(App_fun(Pseq_concl s1))),
             Lam(Var_var tm,App_arg(Pseq_concl s1)) )) &
    sameset(as,Pseq_assum s1) & Pallnotfree(Var_var tm,as);;

let PINST_TYPE = fun (Pseq(as,t),tyl,s1) .
    every Type_OK (map fst tyl) &
    Ptyinst(as,t,tyl,Pseq_concl s1) &
    Plty_snotoccurs(map snd tyl,as) &
    sameset(as,Pseq_assum s1);;

let PDISCH = fun (Pseq(as,t),tm,s) .
    Pwell_typed tm & Pboolean tm & (t = PIMP(tm,Pseq_concl s)) &
    sameset(as,delete(Pseq_assum s,tm));;

let PMP = fun (Pseq(as,t),s1,s2) .
    (Pseq_concl s1 = PIMP((Pseq_concl s2),t)) &
    sameset(as,union(Pseq_assum s1,Pseq_assum s2));;

% --- Inference --- %

rectype Inference = AX_inf of Psequent
        | AS_inf of Psequent # Pterm
        | RE_inf of Psequent # Pterm
        | BE_inf of Psequent # Pterm
        | SU_inf of Psequent # (Psequent#string#Type)list # Pterm # Psequent
        | AB_inf of Psequent # Pterm # Psequent
        | IN_inf of Psequent # (Type # string)list  # Psequent
        | DI_inf of Psequent # Pterm # Psequent
        | MP_inf of Psequent # Psequent # Psequent;;

let Inf_concl =
  fun AX_inf s          . s
   |  AS_inf(s,t)       . s
   |  RE_inf(s,t)       . s
```

```
   | BE_inf(s,t)        . s
   | SU_inf(s,tdl,t,s1) . s
   | AB_inf(s,t,s1)     . s
   | IN_inf(s,tyl,s1)   . s
   | DI_inf(s,t,s1)     . s
   | MP_inf(s,s1,s2)    . s;;

let Inf_hyps =
  fun AX_inf s          . []
   | AS_inf(s,t)        . []
   | RE_inf(s,t)        . []
   | BE_inf(s,t)        . []
   | SU_inf(s,tdl,t,s1) . s1.(map fst tdl)
   | AB_inf(s,t,s1)     . [s1]
   | IN_inf(s,tyl,s1)   . [s1]
   | DI_inf(s,t,s1)     . [s1]
   | MP_inf(s,s1,s2)    . [s1;s2];;

let OK_Inf =
  fun AX_inf s          . mem(s,Axil)
   | AS_inf(s,t)        . PASSUME(s,t)
   | RE_inf(s,t)        . PREFL(s,t)
   | BE_inf(s,t)        . PBETA_CONV(s,t)
   | SU_inf(s,tdl,t,s1) . PSUBST(s,tdl,t,s1)
   | AB_inf(s,t,s1)     . PABS(s,t,s1)
   | IN_inf(s,tyl,s1)   . PINST_TYPE(s,tyl,s1)
   | DI_inf(s,t,s1)     . PDISCH(s,t,s1)
   | MP_inf(s,s1,s2)    . PMP(s,s1,s2);;


% --- Proof --- %

let Is_proof L =
  letref r,l = true,L in
  while not l=[] do
    (if (OK_Inf (hd l) &
         lmem(Inf_hyps(hd l),map Inf_concl (tl l))) then
       (l :=  tl l ; ())
     else
       (r,l := false,[] ; ())
    );
  r;;
```

Appendix L

Proof checking example

Below is a small example showing how the proof checker works. Further examples can be found in the file `Implement/examples.ml`. After loading the program file `proverb.ml`, we first make a number of definitions, to make the terms more readable. For brevity, we only show what is entered.

```
#let EQ ty =
# Const(`=`,Tyop(`fun`,[ty;Tyop(`fun`,[ty;Tyop(`bool`,[])])]));;

#let PLUS =
# Const(`+`,Tyop(`fun`,
#  [Tyop(`num`,[]);Tyop(`fun`,[Tyop(`num`,[]);Tyop(`num`,[])])]));;

#let SUC =
# Const (`SUC`,Tyop(`fun`,[Tyop(`num`,[]);Tyop(`num`,[])]));;

#let MplusN = App(App(PLUS,Var(`m`,Tyop(`num`,[]))),
#                 Var(`n`,Tyop(`num`,[])));;

#let NplusM = App(App(PLUS,Var(`n`,Tyop(`num`,[]))),
#                 Var(`m`,Tyop(`num`,[])));;

#let ADDSYM = Pseq([],App(App(EQ(Tyop(`num`,[])),MplusN),NplusM));;
```

Thus `EQ ty` returns the equality constant for type `ty`, `PLUS` represents the constant "+", `SUC` the constant "`SUC`", `MplusN` the term "`m+n`" and `NplusM` the term "`n+m`". `ADDSYM` represents the sequent with no assumptions and conclusion "`m+n = n+m`".

Next we enter the current theory basis: the type structure `Typl`, the signature `Conl` and the current theorem list `Axil`. Note that these are entered by assignment to reference variables.

```
Typl := [(`bool`,0);(`fun`,2);(`num`,0)];;
Conl :=
 [(`=`,Tyop(`fun`,[Tyvar``;Tyop(`fun`,[Tyvar``;Tyop(`bool`,[])])]));
  (`⇒ `,Tyop(`fun`,
    [Tyop(`bool`,[]);Tyop(`fun`,[Tyop(`bool`,[]);Tyop(`bool`,[])])]));
  (`0`,Tyop(`num`,[]));
  (`SUC`,Tyop(`fun`,[Tyop(`num`,[]);Tyop(`num`,[])]));
  (`+`,Tyop(`fun`,
    [Tyop(`num`,[]);Tyop(`fun`,[Tyop(`num`,[]);Tyop(`num`,[])])]));
 ];;
Axil := [ADDSYM];;
```

Thus the type structure contains booleans and natural numbers, the signature contains polymorphic equality, implication, zero, and the successor and addition operators. The list of axioms and assumed theorems contains only the symmetry of addition.

Now we can check a simple proof. The following dialogue shows one example.

```
#Is_proof
#   [SU_inf
#     (Pseq([],App(App(EQ(Tyop(`num`,[])),App(SUC,MplusN)),
#                  App(SUC,NplusM) )),
#      [(ADDSYM,`d`,Tyop(`num`,[]))],
#      App(App(EQ(Tyop(`num`,[])),App(SUC,MplusN)),
#           App(SUC,Var(`d`,Tyop(`num`,[]))) ),
#      Pseq([],App(App(EQ(Tyop(`num`,[])),App(SUC,MplusN)),
#                  App(SUC,MplusN) )) )
#   ;RE_inf
#     (Pseq([],App(App(EQ(Tyop(`num`,[])),App(SUC,MplusN)),
#                  App(SUC,MplusN) )),
#      App(SUC,MplusN) )
#   ;AX_inf
#     ADDSYM
#   ];;
true : bool
```

indicating that the list of inferences entered actually are a valid proof. In a traditional notation, the proof is

1.	$m+n=n+m$	hypothesis
2.	$SUC(m+n)=SUC(m+n)$	reflexivity
3.	$SUC(m+n)=SUC(n+m)$	substitution, 1, 2

The proof checker also recognises incorrect proofs. For example, if we change the typing in a single place (`` `bool` `` rather than `` `num` `` in the fifth line), we get the following:

```
#Is_proof
#   [SU_inf
#     (Pseq([],App(App(EQ(Tyop(`num`,[])),App(SUC,MplusN)),
#                   App(SUC,NplusM) )),
#      [(ADDSYM,`d`,Tyop(`bool`,[]))],
#      App(App(EQ(Tyop(`num`,[])),App(SUC,MplusN)),
#           App(SUC,Var(`d`,Tyop(`num`,[]))) ),
#      Pseq([],App(App(EQ(Tyop(`num`,[])),App(SUC,MplusN)),
#                   App(SUC,MplusN) )) )
#   ;RE_inf
#     (Pseq([],App(App(EQ(Tyop(`num`,[])),App(SUC,MplusN)),
#                   App(SUC,MplusN) )),
#      App(SUC,MplusN) )
#   ;AX_inf
#     ADDSYM
#   ];;
false : bool
```