A proof checked for HOL

Wai Wong

March 1996
## Contents

<table>
<thead>
<tr>
<th>I</th>
<th>An Overview</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>1.1</td>
<td>Proofs in HOL</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>Formal HOL proof theory</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>Proof file format</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>The Checker</td>
<td>7</td>
</tr>
<tr>
<td>2.1</td>
<td>Checking HOL Proofs</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Using the Proof Checker</td>
<td>8</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Loading the checker</td>
<td>8</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Invoking the checker</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Operations of the checker</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>Organisation of modules</td>
<td>10</td>
</tr>
<tr>
<td>2.4.1</td>
<td>The Core Modules</td>
<td>11</td>
</tr>
<tr>
<td>2.4.2</td>
<td>The two passes</td>
<td>11</td>
</tr>
<tr>
<td>2.4.3</td>
<td>Auxiliary Modules</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>Benchmarking</td>
<td>14</td>
</tr>
<tr>
<td>II</td>
<td>The Program</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>The first pass</td>
<td>17</td>
</tr>
<tr>
<td>4.1</td>
<td>The specification</td>
<td>17</td>
</tr>
<tr>
<td>4.2</td>
<td>The implementation</td>
<td>17</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Parser</td>
<td>18</td>
</tr>
<tr>
<td>4.2.2</td>
<td>The user function</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>The second pass</td>
<td>23</td>
</tr>
<tr>
<td>5.1</td>
<td>The implementation</td>
<td>23</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Parser</td>
<td>24</td>
</tr>
<tr>
<td>5.1.2</td>
<td>The user function</td>
<td>29</td>
</tr>
<tr>
<td>6</td>
<td>The parsing functions</td>
<td>31</td>
</tr>
<tr>
<td>6.1</td>
<td>The specification</td>
<td>31</td>
</tr>
<tr>
<td>6.2</td>
<td>The implementation</td>
<td>32</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Abstract characters</td>
<td>32</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Mapping functions</td>
<td>33</td>
</tr>
<tr>
<td>6.2.3</td>
<td>character functions</td>
<td>34</td>
</tr>
<tr>
<td>6.2.4</td>
<td>String Functions</td>
<td>36</td>
</tr>
<tr>
<td>6.2.5</td>
<td>Parsing functionals</td>
<td>39</td>
</tr>
</tbody>
</table>
# Keywords

7.1 The specification .................................................. 43
7.2 The implementation .................................................. 44

# The checking rules

8.1 The specification .................................................. 45
8.2 The implementation .................................................. 46
     8.2.1 Functions for outputting to log file ...................... 47
8.3 Checking functions .................................................. 48
     8.3.1 Primitive rules ............................................. 48
     8.3.2 Miscellaneous Functions .................................. 51
     8.3.3 Derived Rules ............................................... 54
     8.3.4 Checking proof lines ...................................... 78

# Proof objects

9.1 The specification .................................................. 82
9.2 The implementation ................................................ 85
     9.2.1 Defining types ............................................. 85
     9.2.2 Exception .................................................. 87
     9.2.3 Internal data structures ................................... 87
     9.2.4 Functions for Pass 1 ....................................... 88
     9.2.5 Functions for Pass 2 ....................................... 90
     9.2.6 Constructors and field selector .......................... 91
     9.2.7 Initialisation ............................................... 91
     9.2.8 Pretty Printer .............................................. 92
     9.2.9 Pretty printer for justification ........................... 93
     9.2.10 Pretty printer for proof line .............................. 96

# Proof environment

10.1 The specification ................................................ 97
    10.1.1 New types ................................................. 97
    10.1.2 Functions and identifiers ................................ 97
    10.1.3 Checking type well-formedness ............................ 98
10.2 The implementation ............................................... 98
    10.2.1 Types ..................................................... 98
    10.2.2 Error handling ............................................ 98
    10.2.3 Constructors and destructors .............................. 98
    10.2.4 Current environment ..................................... 99
    10.2.5 Default environments .................................... 99
    10.2.6 Setting up the environment ................................ 102
    10.2.7 Well-formedness and well-typedness ...................... 103
    10.2.8 Augmenting the environment ............................... 104

# HOL theorems

11.1 The specification ............................................... 106
    11.2 The implementation .......................................... 106
    11.2.1 The type $hthm$ .......................................... 107
    11.2.2 The constructor .......................................... 107
    11.2.3 The destructors .......................................... 107
    11.2.4 The comparator .......................................... 107
    11.2.5 Output function .......................................... 107
    11.2.6 Pretty Printer .......................................... 107
12 HOL terms

12.1 The specification ........................................ 109
12.1.1 Types .................................................. 109
12.1.2 Constructors and destructors ......................... 109
12.1.3 More destructors and testers ......................... 109
12.1.4 Term comparison ........................................ 110
12.1.5 Type of a term ......................................... 110
12.1.6 \(\alpha\)-conversion ..................................... 110
12.1.7 Type instantiation ...................................... 110
12.1.8 Free and bound variables .............................. 111
12.1.9 Substitutions ........................................... 111
12.1.10 Generating unique variables ......................... 112
12.1.11 Special types ......................................... 112
12.1.12 Output function ...................................... 112
12.1.13 Pretty Printer ........................................ 112

12.2 The implementation ...................................... 112
12.2.1 Type for HOL terms ................................... 113
12.2.2 Exception .............................................. 113
12.2.3 Ordering function ...................................... 113
12.2.4 Term constructors ..................................... 114
12.2.5 System generated variables ......................... 115
12.2.6 Term destructors ...................................... 115
12.2.7 Term testers ........................................... 115
12.2.8 More destructors and testers ......................... 115
12.2.9 Free variables ......................................... 118
12.2.10 Variant of variable ................................... 118
12.2.11 Type of a term ....................................... 118
12.2.12 Special types testers ................................. 119
12.2.13 Type variables in term ............................... 119
12.2.14 Type in a term ....................................... 119
12.2.15 Occurrence of type variables ...................... 120
12.2.16 Checking type instantiation ......................... 120
12.2.17 \(\alpha\)-conversion .................................... 122
12.2.18 Pretty Printer ........................................ 123
12.2.19 Substitutions ......................................... 124
12.2.20 Create template for substitution ................... 126
12.2.21 Output function ...................................... 127

12.3 Structure lt-termCap ..................................... 127

13 HOL types .................................................. 128

13.1 The specification ........................................ 128
13.1.1 Types .................................................. 128
13.1.2 Constructors and destructor ......................... 128
13.1.3 Type comparison ...................................... 129
13.1.4 Getting type variables ............................... 129
13.1.5 Test for sub-types ................................... 129
13.1.6 Special types ......................................... 129
13.1.7 Type instantiation .................................... 129
13.1.8 Output function ...................................... 130
13.1.9 Pretty Printer ........................................ 130

13.2 The implementation ...................................... 130
13.2.1 Exception ........................................... 130
13.2.2 cmpType ........................................... 130
13.2.3 type_tyvars and type_tyvars1 ....................... 131
13.2.4 Type constructors, destructors and testers ......... 131
13.2.5 type_in_type ...................................... 132
13.2.6 Special type constants ................................ 132
13.2.7 Type instantiation ................................... 132
13.2.8 Perform type instantiation ......................... 133
13.2.9 Compatibility of types ............................... 134
13.2.10 Output function ................................... 134
13.2.11 Pretty Printer ...................................... 134
13.3 Structure ItypeCpp .................................. 136

14 Error handling ........................................... 137
14.1 Debugging utilities .................................... 137
14.1.1 The implementation ................................ 137
14.1.2 Initialisation ...................................... 138
14.2 Exception handling .................................... 138
14.2.1 The structure Exception ............................ 139

15 Input and output management ............................. 141
15.1 The specification ...................................... 141
15.2 The structure I0 ....................................... 142
15.2.1 Local variables ..................................... 142
15.2.2 Data conversions ................................... 143
15.2.3 Reading input file .................................. 143
15.2.4 Input buffering ..................................... 143
15.2.5 Splitting a string ................................... 144
15.2.6 Input functions .................................... 144
15.2.7 Output functions ................................... 145
15.2.8 Closing sockets .................................... 146
15.2.9 File access checking ................................. 146
15.2.10 Delay function .................................... 147
15.2.11 Opening sockets .................................. 147
15.2.12 Local variables for file name parsing ............ 149
15.2.13 File name parsing ................................ 149

16 Report generation ........................................ 151
16.1 The specification ...................................... 152
16.2 The implementation .................................... 152
16.2.1 Simple items ....................................... 152
16.2.2 A list of items ..................................... 153
16.2.3 Item having a list .................................. 153
16.2.4 Item opening and closing ........................... 153
16.2.5 Time of day ........................................ 153
16.2.6 Writing the preamble ............................... 154

A Static arrays ........................................... 155
A.1 The static array functor ................................ 155

B Linking the modules ...................................... 156
Abstract

Formal proofs generated mechanically by theorem provers are often very large and shallow, and the theorem provers are themselves very complex. Therefore, in certain application areas, such as in safety-critical systems, it is necessary to have an independent means for ensuring the consistency of such formal proofs. This report describes an efficient proof checker for the HOL theorem prover. This proof checker has been tested with practical proofs consisting of thousands of inference steps. It was implemented in Standard ML of New Jersey.

The first part of the report gives an overview of the program. It describes

- the rationale of developing a proof checker,
- how to use the checker, and
- how the checker works.

The second part of the report describes the program in detail. The complete source code is included in the description.
The idea of recording inference steps and generating proof lines has been suggested by many people including Malcolm Newey and Keith Hanna. Mike Gordon implemented a prototype of the recording functions in HOL88. The translation of the formal HOL proof theory into ML functions used in the formal version was carried out by John Herbert. It was a great pleasure to work with Mike, Paul, Brian and others in the Hardware Verification Group in the Computer Laboratory in Cambridge. The work described here would not have been completed without their invaluable advice, help and company. A special thank is given to Paul for his efforts in proof reading a draft of the report. He corrected many mistakes and gave many suggestions.
Part I

An Overview
Formal methods have been used in the development of many safety-critical systems in the form of formal specification and formal proof of correctness. Formal proofs are usually carried out using theorem provers or proof assistants. These systems are based on well-founded formal logic, and provide a programming environment within which the user can discover, construct and perform proofs. The result of this process is usually a set of theorems which can be stored in a disk file and used in subsequent proofs. HOL is one of the most popular theorem proving environments. The users interact with the system by writing and evaluating ML programs. The programs instruct the system how to perform proofs. A proof is a sequence of inferences. It is transient in the HOL system in the sense that there is no object that exists as a proof once a theorem has been derived.

In some safety-critical applications, computer systems are used to implement some of the highest risk category functions. The design of such a system is often formally verified. The verification usually produces a large proof consisting of tens of thousands, even up to several millions, of inferences. [9] describes a proof of correctness of an ALU consisting of a quarter of a million inference steps. In such situations, it is desirable to check the consistency of the sequence of inferences with an independent checker. The reasons for requiring independent checking are:

- the mechanically generated formal proofs are usually very long, often consists of thousands, even millions of inferences;
- the mechanically generated formal proofs are usually very shallow in the sense that they are not mathematically interesting;
- the theorem proving systems are usually very complex so that it is extremely difficult (if not impossible) to verify their correctness;
- the programs that a user develops while doing the proof are very often too complicated and do not have a simple mapping to the sequence of inferences performed by the system;
- the requirement of the certification bodies, for example, the U. K. Defence standard 00 – 55 calls for such an independent proof checker when the ‘highest degree of assurance in the design’ is required [7].

An independent proof checker can be much simpler than the theorem prover so that it is possible to be verified formally. The relation between the checker and the HOL system is shown in Figure 1.1. The top half shows the usual process of using the HOL system to perform proofs. The bottom half shows the process of checking proofs generated by the HOL system. These two process may be carried out at different times by different people in different places since the checker is a totally independent system.

Described in this document is a proof checker for the HOL theorem prover. The dominant requirement of this checker is that it is able to check large proofs generated from real applications. This means that the implementation should be fast and efficient, and should be able to perform reasonably well with limited resources, i.e., limited amount of memory. With its possible verification in mind, the checker fairly closely follows von Wright’s formal theory[8] which is described briefly in Section 1.2.
The proof checker is implemented in Standard ML of New Jersey (SML/NJ). Two versions of the checker have been implemented: the core version and the efficient version. The core version is able to check proofs consisting of only primitive inferences and its critical part is translated directly from the formal theory. The efficient version is able to check larger proofs. The remainder of PART I gives an overview of the checker, and PART II describes the entire program of the efficient version in detail.

1.1 Proofs in HOL

A proof is a sequence of logical inferences. All theorems in a logic can be derived from the axioms by applying the inference rules. The proof theory of the logic specifies what the valid inferences are.

In the HOL logic, there are five axioms and eight primitive inference rules. Derived inference rules are sequences of inferences grouped together. Both primitive and derived inference rules are implemented by ML functions in the HOL system. By calling these ML functions, a user can carry out simple proofs to derive theorems from existing theorems. This style is known as forward proof, and it is used mainly in simple proofs.

More often, users will do goal-directed proofs in HOL. In this style, a conjecture is set up as a goal. Tactics are then applied to reduce the goal to simpler subgoals recursively until the resulting subgoals are all resolved. The HOL system validates the tactics, assembles a derivation, and performs the inferences to derive the theorem.

Both proof styles in HOL can be modelled by Hilbert’s proof style, i.e., a proof is a sequence of inference steps, each step is written on a single line with a line number, the theorem derived in the step and a justification, such as the simple example in Figure 1.2.

According to whether or not proofs are generated entirely in terms of primitive inferences of the logic, theorem provers can be classified as fully-expansive or partially-expansive[2]. Fully-expansive theorem provers perform every single primitive inference when deriving theorems.
1.2. FORMAL HOL PROOF THEORY

1. $\Gamma \vdash t_1 = t_2$  
   [Hypothesis]
2. $\vdash t_1 = t_1$  
   [Reflexivity]
3. $\Gamma \vdash t_2 = t_1$  
   [Substitution of 1 into 2]

Figure 1.2: A simple proof in Hilbert style

Because of the lengthy process, the performance of this type of theorem prover is often limited. Early versions of the HOL system were fully-expansive. To improve the performance, thirty-four frequently used derived inference rules are implemented directly in the current version of HOL. The primitive rules, directly implemented derived rules and several functions for introducing new constants and accessing stored theorems are collectively known as basic inference rules.

The proofs generated by the HOL system can be saved in a proof file which contains a textual representation of the sequences of inference steps constituting the proofs. This file is the interface between the theorem prover and the proof checker.

1.2 Formal HOL proof theory

A formal theory of the HOL logic proof theory has been developed in HOL by von Wright [8]. This provides a theoretical base on which a formal specification of a proof checker may be developed and against which verification of the checker may be performed.

In von Wright's theory of the HOL logic, a type Type is defined to represent HOL types. Similarly, a type Ftype is defined to represent HOL terms. Theorems are represented by the type Fseq, standing for sequent.

HOL inference rules are written as

$$\Gamma_1 \vdash t_1 \quad \ldots \quad \Gamma_n \vdash t_n \quad \text{INF_RULE}$$

where the sequents above the line are known as hypotheses and the sequent below the line is known as the conclusion. The name of the inference rule and the side conditions, if any, are written next to the line. Inference rules are implemented in the HOL system as ML functions which usually take the hypotheses as the arguments and returns the conclusion. In von Wright's theory, inference rules are presented by predicates on the hypotheses and conclusion. A theorem asserting the validity of an inference can be derived if the conclusion can be derived from the set of hypothesis sequents using the inference rule. For example, if Pinf rule is the predicate representing the hypothetical inference INF_RULE above, if the inference is valid, the following theorem can be derived.

$$P_{inf\_rule}(\Gamma_1 \vdash t_1 \quad \ldots \quad \Gamma_n \vdash t_n) \quad \Gamma \vdash t$$

1.3 Proof file format

The proof accepted by the checker is in a proof file format prf. The format is described in detail in the Cambridge University Computer Laboratory technical report No. 306 [10]. The prf format is based on the Hilbert proof style model described in Section 1.1.

The proof file format prf has two levels: the core level, which allows proofs consisting of only primitive inference rules to be written into the file, and the extended level, which allows all basic inference rules.

A proof file may contain one or more proofs. Each proof has a name, a goal list and a list of proof lines. The name is a string used to identify the proof. The goal list may be empty or
have one or more theorems. This is a mechanism for improving the efficiency of the checker. If the goal list is empty, the checker will check the entire proof. If all the theorems in the goal list have been found in the proof lines, the checker can conclude that the goals have been proved and stop processing the remainder of the proof. This is useful because certain automatic proof procedures may perform more inferences than necessary to derive the required theorem. This is based on some observations on the proofs generated by HOL. Its usefulness in practice is still not known. However, the checker can always ignore the goal list and check the entire proof. In fact, this is the default behaviour of the checker. Each proof line has three parts: the \textit{line number}, the \textit{justification}, which is the name of the inference rule and the arguments supplied to the rule, and the \textit{derived theorem}.

The proof file format is primarily an interface between the HOL theorem prover and other independent proof checkers. Files in prf format are intended to be read by machines not humans. The concrete syntax of the prf format is similar to a LISP S-expression. For example, the proof file in prf format of the proof shown in Figure 1.2 is listed in Figure 1.3 below.

(VERSION PRF FORMAT 1.0 EXTENDED)
(ENV HOL[] [] )

(PROOF sym
[]

(LINE -1(Hypothesis)
(THM [][](A (A (C = (o fun[(c bool)(o fun[(c bool)(c bool)])]))
(A (A (C \ (o fun[(c bool)(o fun[(c bool)(c bool)])]))
(V t1(c bool))))(V t2(c bool))))
(A (A (C \ (o fun[(c bool)(o fun[(c bool)(c bool)])]))
(V t2(c bool))))(V t1(c bool))))))

(LINE 1(Ref1 (A (A (C \ (o fun[(c bool)(o fun[(c bool)(c bool)])]))
(V t1(c bool))))(V t2(c bool))))
(THM [][](A (A (C = (o fun[(c bool)(o fun[(c bool)(c bool)])]))
(A (A (C \ (o fun[(c bool)(o fun[(c bool)(c bool)])]))
(V t1(c bool))))(V t2(c bool))))(A (A (C \ (o fun[(c bool)(c bool)])
(o fun[(c bool)(c bool)]))) (V t1(c bool)))(V t2(c bool)))))))

(LINE 2(Subst [[-1(V GENVXVAR509(c bool)])]]
(A (A (C = (o fun[(c bool)(o fun[(c bool)(c bool)])]))
(V GENVXVAR509(c bool))))(A (A (C \ (o fun[(c bool)(c bool)])
(o fun[(c bool)(c bool)]))))(V t1(c bool)))(V t2(c bool))))))
(THM [][](A (A (C = (o fun[(c bool)(o fun[(c bool)(c bool)])]))
(A (A (C \ (o fun[(c bool)(o fun[(c bool)(c bool)])]))
(V t2(c bool))))(V t1(c bool))))(A (A (C \ (o fun[(c bool)(o fun[(c bool)(c bool)])]))
(V t1(c bool)))(V t2(c bool)))))))
)
)

Figure 1.3: Proof file in prf format of a simple proof
2.1 Checking HOL Proofs

To check a HOL proof is to make sure that every inference step in the proof is valid. An inference step is valid if and only if

- all hypotheses are in the correct form as specified by the inference rule;
- all hypotheses are either axioms or theorems derived in previous inference steps;
- the conclusion is in the correct form as specified by the inference rule; and
- all the types and terms appeared in the hypotheses and conclusion are well formed in the current signature, i.e., the types and constants known at that point.

Since the HOL logic has only eight primitive inference rules, a checker for proofs consisting of only primitive inferences will be relatively simple, so it may possibly be verified formally. This corresponds to the core level of the prf proof file format. A version of the checker, the core checker, accepting only core level proof files was developed first. With the ultimate goal of formal verification in mind, the core checker follows von Wright's formal HOL proof theory very closely.

However, the current version (HOL88 version 2.2) of the HOL system is not fully-expansive. The proofs generated by HOL consist of all the basic inferences. The proof files are in an extended level and cannot be accepted by the core checker.

There are basically two different approaches to implementing an extended level checker. The first approach is to write a program to expand the inference steps involving derived rules into a sequence of primitive steps before being sent to the core checker. This approach has the advantage of utilising the core checker which may be formally verified, therefore, achieving higher confidence in the consistency of the proof. However, this approach can increase the number of inference steps considerably so the amount of time required to check the proof will take much longer.\(^1\) The second approach is to check all basic inference rules directly. This approach can result in a more efficient checker since the basic derived rules are relatively simple to check.

Since one of the requirements of this project is to demonstrate the feasibility of proof checking for real practical proofs, which consist of thousands or tens of thousands of inference steps, the checker should be fast and efficient, and should be able to perform reasonably well with limited resources, i.e., limited amount of physical memory and disk space.

Described in Part II of this report is an implementation of the checker using the second approach. This version is known as the efficient checker. It differs from the core level checker mainly in the internal representation of the terms, the handling of the derived theorems. Other parts of the checker, such as the file I/O, the proof file parser and the error and exception handling are identical.

For internal representation of terms, the efficient checker uses de Bruijn's nameless representation. This makes the α-equivalence test and substitution simpler.

\(^1\)By examining the derivations of the derived rules, one can see that each derived rule may be expanded into five to twenty primitive rules. A large proportion of inferences in normal proofs are non-primitive.
No matter which approach is used to implement a checker, its memory requirements are very large for large proofs because all theorems derived in the sequence have to be kept in memory. This is because a theorem derived in an earlier step may be referred to by the very last step. Logically, many modern systems are able to address many gigabytes, even up to terabytes of virtual memory, but physical memory is still limited. When large numbers of theorems are kept in memory, thrashing occurs, thus slowing down the process. This problem has been solved in the efficient checker by processing the proof file in two passes (see Section 2.3).

2.2 Using the Proof Checker

To a user, the checker is a program which reads a proof file, checks the proofs in it and reports back with either a success which means the proofs are correct or a failure which means the opposite (see Figure 1.1). It creates a log file containing information of what hypotheses and stored theorems have been used and the resulting theorems of the proofs. The log file is in a format similar to the proof file. It is mainly for use by other programs.

2.2.1 Loading the checker

Currently, the checker program modules have to be loaded into SML by evaluating the expression

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

This will compile and link the modules to form the checker. A top-level function, namely check_proof, will then be defined as the entry point of the checker. When the program becomes stable, it will be possible to save an executable image of the checker. Then, it can be invoked as a shell command.

2.2.2 Invoking the checker

After loading the checker, it can be invoked by evaluating the function check_proof which takes a string as its sole argument. The string is the proof file name which, by convention, has the suffix .prf but the checker accepts any name. If the filename has a suffix .gz, the checker will assume it is a compressed file. It will run a decompressor automatically, and the log file will also be stored in a compressed form. The default compression/decompression utilities are the GNU gzip/gunzip programs. Below is a sample session of using the checker to check a compressed proof file named NUL_FUN_CURRY in the directory proofs parallel to the current directory. (Some of the output produced by the checker are omitted for brevity.)
2.3. OPERATIONS OF THE CHECKER

- check_proof "../proofs/MULT_FUN_CURRY.prf.gz"

Current environment: MULT_FUN_CURRY

Proof: MULT_FUN_CURRY
Proof MULT_FUN_CURRY has been checked

Proof: MULT_FUN_CURRY_THM
Proof MULT_FUN_CURRY_THM has been checked

Using the following hypotheses:
\(<-8> \quad \text{T : bool}

\{ \ldots \text{some theorems omitted \ldots} \}

Proof: MULT_FUN
Proof MULT_FUN has been checked

Proof: MULT_FUN_DEF
Proof MULT_FUN_DEF has been checked

Using the following hypotheses:

\{ \ldots \text{some theorems omitted \ldots} \}

val it = () : unit

The name of the log file is derived from the input file name. If the input file name has the suffix .prf, it is replaced by the suffix .clg (stands for checker log file). If the input file name has no suffix, the log file suffix is appended.

2.3 Operations of the checker

When the checker is invoked, it creates a decompress process running as a filter in the background. The communication between this process and the checker is via a socket. In the case where the file is not compressed, no decompression process is needed, but the communication is still via a socket. This arrangement simplifies the checker as its input routine always reads from the input socket. Similarly, an output socket is created with a compression process running to compress the output to the log file on the fly. Figure 2.1 illustrates this arrangement.

The checker processes the input file in two passes. In the first pass, it builds up a table of theorem references. Each inference step (also known as proof line) may refer to theorems derived in any previous steps. The table has an entry for each line. The entry contains the largest line number that that proof line is referenced by. As the references only occur in some justifications, most of the text can be quickly skipped over.

In the second pass, the checker analyses and checks every proof line according to the specification of the inference rules. After checking each line, the derived theorem is saved in a dictionary keyed by the line numbers if it is referred to by later lines. When a theorem is fetched from the dictionary for the last time, it is removed. Thus, a minimum amount of memory is required.
2.4 Organisation of modules

The checker is structured into a number of modules as shown in Fig. 2.2. The modules can be divided into two groups: the core group and the auxiliary group. Modules in the core group are shown in the figure with a thick border, whereas other modules are shown with a thin border.
2.4. ORGANISATION OF MODULES

\[ \Gamma \vdash t_1 = t_2 \]

a) The inference rule

\[ |- \text{Typl Con1 as t tm.} \]
\[ \text{PASSUME Typl Con1 (Fseq as t) tm =} \]
\[ \text{PwellTyped Typl Con1 tm} \wedge \]
\[ \text{Pboolean tm} \wedge \]
\[ (t = \text{tm}) \wedge \]
\[ (\text{as} = \{\text{tm}\}) \]

b) The formal definition

c) The checking function

Figure 2.3: Checking function and formal definition of the primitive rule ASSUME

2.4.1 The Core Modules

The core modules implement the internal representation of HOL types, terms, theorems, proofs, and proof environment. Each kind of objects is represented by an ML type. The type and associated operations on the objects of the type are implemented in a module. The module names are generally descriptive so it is very easy to identify the module for a particular kind of object. The Check module contains functions for checking the consistency of all basic inference rules. All checking functions for the primitive rules are the same as the formal version except for very minor changes to take care of the slightly different representation of HOL types and terms. Figure 2.3 shows the primitive rule ASSUME, its formal definition in the HOL proof theory and the checking function. The SML function and the HOL definition are very close.

The functions for checking other basic rules are derived from specification of these rules found in [5]. Fig. 2.4 shows the basic inference rule SYM and its checking function.

2.4.2 The two passes

In the first pass, the checker builds a theorem reference table. This table consists of two dynamic arrays whose elements are integers as shown in Fig. 2.5a. Each element represents a proof line. The indices to the elements are the proof line numbers. Since the proof lines are numbered with both positive and negative numbers, but only non-negative numbers are allowed in indexing the array, two arrays are used. The TabHyp array is for the hypothesis lines whose line numbers are negative, and the TabLine array is for proof lines whose numbers are positive. These arrays

fun chk_Sym(line, n, thm) = 
  let val thml = get_thm(line, n)
  val (left,right) = dest_eq (concl thml)
  \[ \Gamma \vdash t_1 = t_2 \]
  in
  \[ \Gamma \vdash t_2 = t_1 \]
  ((right,left) = dest_eq (concl thm)) andalso
  (HtermSet.equal((hyp thm), (hyp thml)))
  end 

Figure 2.4: Basic rule SYM and its checking function
are created using the DynamicArray module in the SML/NJ library. The use of dynamic arrays instead of static ones removes the upper limit of the number of lines in the proof.

\[
\begin{array}{c|c} 
\text{TabHyp} & \text{TabLine} \\
1 & 3 \\
2 & 0 \\
3 & 0 \\
4 & 0 \\
5 & 0 \\
\end{array}
\begin{array}{c|c} 
\text{key} & \text{theorem} \\
-1 & \ldots \\
2 & \ldots \\
\end{array}
\]

a) Theorem reference table  b) Theorem table

Figure 2.5: Data structures for theorem references

In the first pass, the checker looks at the justification part of the proof lines. When it encounters a reference to a theorem in a previous proof line, it enters the current line number into the element corresponding to the referred line in the table. For example, when the checker is at Line 3, it finds that this line refers to the theorem in hypothesis Line 1. It enters 3 into the first element of TabHyp. To speed up the Pass 1 process, the checker skips over other parts of the proof line quickly. This is done by scanning the input and looking for matching parentheses only. At the end of this pass, each element of the theorem reference table will contain the highest line number which is the latest line referring to the theorem. In the table shown in Fig. 2.5a, Line 5 is the last line referring to the theorem derived in Line 2 and Line 4.

In the second pass, the checker stores theorems referred to by other proof lines in a theorem table. This table is implemented by a dictionary in the Dict module of the SML/NJ library. The key of each entry is the line number. Since the dictionary is represented by a balanced splay tree, searching for a theorem is fast. After checking a proof line, the checker examines the theorem reference table, if the value of the current element is greater than the current line number, i.e., it will be referred to later, the theorem is saved in the theorem table. Fig. 2.5b illustrates the situation in which the checker has just stored the theorem derived in Line 2. When the checker retrieves a theorem, it also examines the theorem reference table. If the current line is the last one to refer to the theorem, i.e., the current line number is equal to the value in the table, the theorem is removed from the dictionary. Continuing the scenario in Fig. 2.5b, the next line is Line 3, which refers to hypotheses Line 1. Since this is the last line referring to the theorem, the checker removes it from the table. This arrangement minimises the number of theorems stored in the table, thus reducing the memory requirement.

2.4.3 Auxiliary Modules

The HOLProofKey module defines the concrete syntax, i.e., the tags, of the proof files. The Parsing module consists of several higher order parsing functions. The parser proper is in the modules Pass1 and Pass2. It is a recursive descent parser.

The Exception and Debug modules are responsible for handling errors. The Debug module maintains a debug flag for each module. The values of these flags are non-negative integers. The
higher the value, the more information will be displayed while checking a proof. The Report module is for formatting the output to the log file.

The Io module handles all file input and output. When the checker is invoked, it creates a decompression process running as a filter in the background. The communication between this process and the checker is via a UNIX domain socket. If the file is uncompressed, no decompression is needed, but a dummy cat process is created, and the communication is still via a socket. This arrangement simplifies the checker as its input routine always reads from the input socket. Similarly, an output socket is created with a compression process to compress the output to the log file on the fly. This arrangement is illustrated in Fig. 2.1.
3 Benchmarking

The proof checker has been tested with many small proof files generated by the HOL system, including all derivations of basic derived rules. The largest proof that has been checked by the checker is a proof of correctness of a simple multiplier described in [4]. This is a medium size proof which generates 14500 intermediate theorems. This proof has been used as a benchmark for many versions of the HOL system.

The multiplier proof consists of four ML files. A proof file is generated for each ML file. It contains all the sub-proofs in the corresponding ML file. [10] describes how to generate these proof files and lists the time required and the files sizes.

Table 3.1: Benchmark for checking the multiplier proof (Time in seconds)

<table>
<thead>
<tr>
<th>Proof File</th>
<th>No. of Thm.</th>
<th>Time Run</th>
<th>System</th>
<th>GC</th>
<th>Real</th>
</tr>
</thead>
<tbody>
<tr>
<td>mk_NEXT</td>
<td>2972</td>
<td>139.3</td>
<td>15.3</td>
<td>3.7</td>
<td>170.0</td>
</tr>
<tr>
<td>MULT_FUN_CURRY</td>
<td>670</td>
<td>77.3</td>
<td>9.6</td>
<td>2.6</td>
<td>100.4</td>
</tr>
<tr>
<td>MULT_FUN</td>
<td>6943</td>
<td>406.3</td>
<td>44.6</td>
<td>15.4</td>
<td>488.8</td>
</tr>
<tr>
<td>HOL_MULT</td>
<td>3946</td>
<td>1472.1</td>
<td>152.0</td>
<td>98.5</td>
<td>1783.3</td>
</tr>
<tr>
<td>Total</td>
<td>14531</td>
<td>2095.0</td>
<td>138.8</td>
<td>120.2</td>
<td>2542.5</td>
</tr>
</tbody>
</table>

The multiplier proof files were successfully checked by the checker. No error was found from the proof files. Table 3.1 lists the time taken to check the proof files. This test ran on a SUN SparcStation 20. The time is in the same order of magnitude as the recording. One important observation is that the process size is relatively small when performing the checking. The process size of the checker when it is just loaded is 14 Mbytes. The maximum size when performing the checking is only 16 Mbytes. This shows that the implementation does keep the memory usage very small.
Part II

The Program

In this part, the checker program is described in detail. Each module of the checker is described in its own chapter. There are two sections within most chapters: the first describes the specification of the module, and the second the implementation. The specification describes the interface between the module and its user. The specifications of the modules are SML signatures. The modules are implemented as SML structures. The order of the chapters follows roughly the reversed sequence in Figure 2.2, i.e., the chapter of the bottom module appears first.

The checker program is written in the MiWeb format, a simple literate programming tool[1]. In the literate programming paradigm[6], the master file contains both the program source and its documentation. This encourages people to write more human readable programs and helps to keep the program and its document consistent. Literate programming tools extract the program source and the documentation from the master file separately. The former will be in a format acceptable to a compiler so that an executable program can be produced. The latter will be processed by a text formatter to produce a printed document. The MiWeb format and tools allow a single master file to contain programs and other machine readable text, such as specifications, in more than one language.
4 The first pass

The primary task of the first pass is to build a table of theorem references. Therefore, it only looks at those justifications which refer to theorems. Any irrelevant items can be skipped over in high speed.

4.1 The specification

The only exported function is `parse_file`. It takes a file name as its sole argument and processes it. If the file name ends with the suffix .gz, it is taken as a compressed file. The default decompressor gunzip is passed to the IO module for preprocessing the file.

```
signature Pass_sig =
  sig
    val parse_file : string -> unit
  end
```

4.2 The implementation

The Pass 1 parser is implemented as the functor `Pass1FUN` with the signature `Pass_sig`.

```
functor Pass1FUN (structure Parsing :Parsing_sig and Henv : Henv_sig
  and Proof : Proof_sig and Hthm : Hthm_sig
  and Hterm : Hterm_sig and Htype : Htype_sig
  sharing Hterm = Hthm.Hterm = Proof.Hterm
  and Htype = Hthm.Htype = Hthm.Hterm.Htype =
    Proof.Htype = Henv.Htype
  and Hthm = Proof.Hthm) : Pass_sig =
  struct
    structure Parsing = Parsing
    structure Io = Parsing.Io
    structure Keyword = Parsing.Keyword
    structure Henv = Henv;
    structure Proof = Proof;
    structure Hthm = Hthm;
    structure Hterm = Hterm;
    structure Htype = Htype;

    open Parsing Htype Hterm Hthm Proof

    fun PASS1_ERR {function, message} =
      Exception.CHK_ERR{origin_structure = "Pass1",
      origin_function = function, message=message};
    val debug = Debug.get_debug("Pass1");
    fun write_out s = (output(std_err, s); flush_out std_err)
```
4.2.1 Parser  The parser is a recursive descent parser. The top expression is a proof file. The exact syntax of the file can be found in [10]. In this pass, we are only interested in theorem references in the justifications, so a lot of input can be skipped.

```plaintext
fun parse_proof_file () =
  (parse_item parse_ver ()
   parse_item parse_env()
   parse_closure parse_proof ()
   ()
   and parse_ver () =
     (get_tag Keyword.VERSION;
      skip_sp ();
      if check_ver_string() then ()
      else raise (PASS1_ERR
                     (function = "parse_ver", 
                      message="incorrect format"));)
   and parse_env () =
     (get_tag Keyword.ENV;
      get_name();
      skip_item ();
      skip_item ()
      ());

The function parse_proof parses a proof. The local variable thms contains a list of goals which are the subexpressions after the tag. Then, it calls new_proof1 to initialise the current proof data structures. Next, parse_list is called to parse the proof lines. At the end of the proof, the pline table is saved for the second pass.

and parse_proof () =
  (get_tag Keyword.PROOF;
   let val name = get_name()
     val thms = parse_list parse_thm ()
   in
     new_proof1 (name,thms);
     parse_list parse_line ();(* handle PARSE_DONE => []
     | e => raise e; *)
     Proof.add_pline_table name;
     if (debug > 0) then (write_out("\nProof: "\nname="\n"");
   Proof.print_prooftab()) else ()
   end;
   ()
   The function parse_line processes each proof line.

and parse_line () =
  (get_tag Keyword.LINE;
   let val n = get_num ()
     val nl = parse_item parse_just ()
     val th = parse_item parse_thm ()
   in
     if (debug > 1) then write_out("\nline "\nmakestring n) else ();
     let
       val proved = Proof.add_pline_tab(n,nl,th)
     in
```
4.2. THE IMPLEMENTATION

```ml
if proved then raise PARSE_DONE else ()
end
end;
()

Now, we are looking at the justification field of a proof line. The function get_name is used
to get the name of the justification which is case insensitive, so the call to toupper folds all
letters to upper case. As we are only interested in the theorem references, we skip over all other
items. The function get_num is used to get the line numbers. A list of line numbers is returned
by parse_just.

end parse_just () =
  let
    val toupper = StringUtil.stringTrans
      ("abcdefghijklmnopqrstuvwxyz", "ABCDEFGHIJKLMNOPQRSTUVWXYZ")
  in
    case toupper(get_name()) of
      | "ABS" => (skip_item(); [get_num ()])
      | "DISCH" => (skip_item(); [get_num ()])
      | "INSTTYPE" => (skip_item(); [get_num ()])
      | "MP" => [get_num(), get_num()]
      | "SUBST" =>
        let
          val l = map (#1) (parse_list_pairs (get_num, skip_item) ())
        in
          (skip_item(); get_num () :: l)
        end
      | "ADDASSUM" => (skip_item(); [get_num()])
      | "APTERM" => (skip_item(); [get_num()])
      | "APTHM" =>
        let val n = [get_num()]
        in
          skip_item(); n
        end
      | "CCONTR" => (skip_item(); [get_num()])
      | "CHOOSE" =>
        let val (_, n1) = parse_pair ((parse_item parse_term), get_num ())
        in
          ([n1, get_num()])
        end
      | "CONJ" => ([get_num(), get_num()])
      | "CONJUNCT1" => ([get_num()])
      | "CONJUNCT2" => ([get_num()])
      | "CONTR" => (skip_item(); [get_num()])
      | "DISJ1" =>
        let val n = [get_num()]
        in
          skip_item(); n
        end
      | "DISJ2" => (skip_item(); [get_num()])
      | "DISJCASES" => ([get_num(), get_num(), get_num()])
      | "EQIMPRULEL" => ([get_num()])
```
"EQIMPRULER" => ([get_num()])
"EQMD" => ([get_num(), get_num()])
"EQINTRO" => ([get_num()])
"EXISTS" => (skip_item(); (* skip a pair *) [get_num()])
"EXT" => ([get_num()])
"GEN" => (skip_item(); [get_num()])
"IMPANTISYMRULE" => ([get_num(), get_num()])
"IMPTTRANS" => ([get_num(), get_num()])
"INST" => (skip_item(); [get_num()])
"MKABS" => ([get_num()])
"MKCOMB" => ([get_num(), get_num()])
"MKEXISTS" => ([get_num()])
"NOTELIM" => ([get_num()])
"NOTINTRO" => ([get_num()])
"SPEC" => (skip_item(); [get_num()])
"SUBS" =>
  let val nl = parse_num_list ()
  in
    (get_num()):nl
  end
"SUBSOCCS" =>
  let
    val nl1 = parse_list (parse_subs_list) ()
    val n = get_num()
  in
    n :: (map #2 nl1)
  end
"SUBSTCONV" =>
  let
    val ntl = parse_list_pairs (get_num, (parse_item parse_term)) ()
    in
      skip_item (); skip_item ();
      (map #1 ntl)
    end
"SYM" => ([get_num()])
"TRANS" => ([get_num(), get_num()])
"_" => (skip_long_string false [RF]; [])
end

A little auxiliary parsing function for substitution lists.

and parse_subs_list () =
  ((parse_num_list ()), get_num())

The function parse_thm recognises a theorem and returns it.

and parse_thm () =
  (get_tag Keyword.THM;
   let val hyp = parse_list parse_term ()
       val concl = parse_item parse_term ()
   in
     if (debug > 2) then write_out("<THM>") else ();
     Hthm.mk_thm(hyp, concl)
   end)
4.2. THE IMPLEMENTATION

The function parse_term recognises a term. There are four kinds of terms, two of them, namely variables and constants, have type information attached. The appropriate term constructor is called to create a term, and it is returned as the value of this function.

```plaintext
and parse_term() =
  let val tag = get_name() in
  if (tag = Keyword.VAR) then
    let val name = get_name()
    val ty = parse_item parse_type()
    in (if (debug > 2) then write_out("-\"name\") else ())
    mk_var(name,ty)
  end
  else if (tag = Keyword.CONST) then
    let val name = get_const_name()
    val ty = parse_item parse_type()
    in (if (debug > 2) then write_out("\$\"name\") else ())
    mk_const(name,ty)
  end
  else if (tag = Keyword.APP) then
    let val rator = parse_item parse_term()
    val rand = parse_item parse_term()
    in (if (debug > 2) then write_out("\"\") else ())
    mk_comb(rator,rand)
  end
  else if (tag = Keyword.ABS) then
    let val v = parse_item parse_term()
    val body = parse_item parse_term()
    in (if (debug > 2) then write_out("\"\") else ())
    mk_abs(v, body)
  end
  else raise (PASS1_ERR(function="parse_term",
                      message="Unknown term \"-tag\")
  end
```

The function parse_type recognises a type. There are three kinds of types: type operators, type constants and type variables. The first two are represented by type operators in the HType module. The appropriate type constructor is called to create a type, and it is returned as the value of this function.

```plaintext
and parse_type() =
  let val tag = get_name() in
  if (tag = Keyword.TYVAR) then
    let val name = get_tyvar_name()
    in (if (debug > 2) then write_out("\"-\"name\") else ())
    mk_vartype name
  end
  else if (tag = Keyword.TYCONST) then
    let val name = get_name()
    in (if (debug > 2) then write_out("\"\"-name\") else ())
    mk_type {Tyargs=[], Tyop=name}
  end
  else if (tag = Keyword.TYOP) then
    let val name = get_name()
```
val ty1 = parse_list parse_type ()
in (if (debug > 2) then write_out("&"^name) else ()
    mk_type {Tyargs=(rev ty1),Tyop=name})
end
else raise (PASS1_ERR{function="parse_type",
                    message="Unknown type"^tag})
end

4.2.2 The user function This is the entry point of the Pass 1 parser. It takes the name of the proof file as its sole argument. This string should include all the suffixes and possibly the path of the file name. It calls the function mk_command in the Io module to work out any compressing and decompressing command if they are necessary. See Page 141 for details of how the file name is interpreted.

fun parse_file fname =
  (Parsing.init(); Proof.init();
   let val (outname, incmd, outcmd) = Io.mk_command fname
     in
     Io.open_input_socket fname incmd
     end;
   let val plst = parse_proof_file () in
     (Io.close_io_socket(); (plst))
     end) handle e =>
   (Io.close_io_socket(); raise e)
end; (* functor Pass1FUN *)
This is the main pass of the checker. It parses and checks the proof file. It has the same
signature as the module Pass1. It is repeated here for easy reference.

signature Pass_sig =
  sig
   val parse_file : string -> unit
 end

5.1 The implementation

The Pass 2 parser is implemented as the functor Pass2FUN with the same signature as the
Pass 1 parser.

functor Pass2FUN (structure Parsing :Parsing_sig and Henv: Henv_sig
  and Proof: Proof_sig and Hthm: Hthm_sig
  and Hterm: Hterm_sig and Htype: Htype_sig
  and Check :Check_sig and Report : Report_sig
sharing Hterm = Hthm.Hterm = Proof.Hterm = Check.Hterm
and Htype = Hterm.Htype = Hthm.Hterm.Htype =
    Proof.Htype = Henv.Htype = Check.Htype
and Hthm = Proof.Hthm = Check.Hthm
and Proof = Check.Proof) : Pass_sig =

struct
  structure Report = Report
  structure Parsing = Parsing
  structure Io = Parsing.Io
  structure Keyword = Parsing.Keyword
  structure Henv = Henv
  structure Proof = Proof
  structure Hthm = Hthm
  structure Hterm = Hterm
  structure Htype = Htype
  structure Check = Check

  open Parsing Htype Hterm Proof

  fun PASS2_ERR {function, message} =
      Exception.CHK_ERR{origin_structure = "Pass2",
        origin_function = function, message=message};
  val debug = Debug.get_debug("Pass2");
  fun write_out s = (output(std_err, s); flush_out std_err)
5.1.1 Parser The parser is a recursive descent parser. It is organised in the same way as the Pass 1 parser. The top expression is a proof file. The exact syntax of the file can be found in [10]. In this pass, the parser calls the checking function after recognising each proof line.

```
fun parse_proof_file () =
  (parse_item parse_ver ();
   parse_item parse_env ();
   parse_closure parse_proof ();
   ()
  )
and parse_ver () =
  (get_tag Keyword.VERSION;
   skip_sp ();
   if check_version_string() then ()
   else raise (PASS2_ERR
           {function = "parse_ver", message="incorrect format"});
  )
and parse_env () =
  (get_tag Keyword.ENV;
   let val name = get_name()
     val tyl = map Henv.mk_typeconst(parse_list_pairs (get_name, get_num) ()
     val cl = map Henv.mk_termconst
       (parse_list_pairs (get_name, (parse_item parse_type)) ()
      in
      if (debug > 0) then write_out("\nCurrent environment: ""name"="\n") else (;
      Henv.mk_proof_env(name, tyl, cl)
      end
  )

At the beginning of a proof, the function new_proof2 is called to initialise the internal data structures which hold the theorem reference table and the list of goals. Also, the opening tag of a proof is written to the log file. After parsing all lines, the USED list in the log file is closed. Then, chk_proof is called to verify that all goals have been proved. The proof in the log file is closed. See the Report module on Page 151 for details of the log file format.

and parse_proof () =
  (get_tag Keyword.PROOF;
   let val name = get_name()
     val thms = parse_list parse_thm()
    in
    if (debug > 0) then write_out("\nProof: ""name"="\n") else ();
    Report.write_line_opening2(Keyword.PROOF, name);
    Report.write_line_opening(Keyword.USED);
    Report.write_tok Keyword.LB;
    new_proof2 name;
    let
      val ls = parse_list parse_line ()
    in
    Report.write_tok Keyword.RB;
    Report.write_closing_line ();
    Check.chk_proof(name, thms, ls);
    Report.write_closing_line ()
    end
  )
```
5.1. THE IMPLEMENTATION

end;
()

The function parse_line processes a proof line. After parsing all fields, namely the line number \( n \), the justification \( just \) and the derived theorem \( th \), it calls chk_pline to check the line. A line number is returned by the checking function. If it is the same as the current line number \( n \), all the goals have been proved. All remaining proof lines can be ignored. The exception PARSE_DONE is raised to pass this signal up. Otherwise, this function returns the line number \( n \).

\[
\text{and parse\_line () = }
\begin{align*}
& \text{(get\_tag Keyword\_LINE;)} \\
& \text{let val n = get\_num ()} \\
& \text{val just = parse\_item parse\_just ()} \\
& \text{val th = parse\_item parse\_thm ()} \\
& \text{in} \\
& \text{if (debug > 1) then write\_out("\"nline \"-makestring n\) else ();} \\
& \text{if (Check.chk\_pline(n, just, th) = n)} \\
& \text{then raise PARSE\_DONE} \\
& \text{else n} \\
& \text{end}
\end{align*}
\]

This function parses the justification field in a proof line, and returns it. The name of the justification is case insensitive.

\[
\text{and parse\_just () =}
\begin{align*}
& \text{let} \\
& \text{val toupper = StringUtil.stringTrans} \\
& \quad \text{("abcdefghijklmnopqrstuvwxyz","ABCDEFGHIJKLMNOPQRSTUVWXYZ")} \\
& \text{in} \\
& \text{case toupper(get\_name()) of} \\
& \quad \text{"ABS" =>} \\
& \quad \text{let val tm = parse\_item parse\_term ()} \\
& \quad \text{val n = get\_num ()} \\
& \quad \text{in} \\
& \quad \text{(Abs (tm, n))} \\
& \quad \text{end} \\
& \quad | \text{"ASSUME" => (Assume (parse\_item parse\_term ()))} \\
& \quad | \text{"BETA\_CONV" => (BetaConv (parse\_item parse\_term ()))} \\
& \quad | \text{"DISCH" =>} \\
& \quad \text{let val tm = parse\_item parse\_term ()} \\
& \quad \text{val n = get\_num ()} \\
& \quad \text{in} \\
& \quad \text{(Disch (tm, n))} \\
& \quad \text{end} \\
& \quad | \text{"INST\_TYPE" =>} \\
& \quad \text{let val tytyl = parse\_list\_pairs} \\
& \quad \text{((parse\_item parse\_type), (parse\_item parse\_type)) ()} \\
& \quad \text{val n = get\_num ()} \\
& \quad \text{in} \\
& \quad \text{(InstType (tytyl, n))} \\
& \quad \text{end} \\
& \quad | \text{"MP" => (Mp (get\_num(), get\_num()))} \\
& \quad | \text{"REFL" => (Refl (parse\_item parse\_term ()))}
\end{align*}
\]
"SUBST" =>
let
val l = (parse_list_pairs (get_num,(parse_item parse_term)) ())
val tm = parse_item parse_term ()
in (Subst (1, tm, get_num ()))
end

"AXIOM" =>
let val s1 = get_name()
val s2 = get_name()
in (Axiom (s1,s2))
end

"DEFINITION" =>
let val s1 = get_name()
val s2 = get_name()
in (Definition (s1,s2))
end

"DEFEXISTSRULE" =>
(DefExistsRule (parse_item parse_term ()))

"HYPOTHESIS" => (Hypothesis)

"NEWAXIOM" =>
let val name = get_name()
val tm = (parse_item parse_term ()
in (NewAxiom (name,tm))
end

"NEWCONSTANT" =>
let val name = get_name()
val ty = (parse_item parse_type ()
in (NewConstant (name,ty))
end

"NEWTYPE" =>
let val n = get_num()
val name = get_name()
in (NewType (n,name))
end

"NUMCONV" =>
(Numconv (parse_item parse_term ()))

"STOREDEFINITION" =>
let val name = get_name()
val tm = (parse_item parse_term ()
in (StoreDefinition (name,tm))
end

"THEOREM" =>
let val s1 = get_name()
val s2 = get_name()
in
  (Theorem (s1,s2))
end

| "ADDASSUM" =>
  let
  val tm = parse_item parse_term ()
  val n = get_num ()
  in
  (AddAssum (tm,n))
end

| "ALPHA" =>
  let
  val tm1 = parse_item parse_term ()
  val tm2 = parse_item parse_term ()
  in
  (Alpha (tm1,tm2))
end

| "APTERM" => (ApTerm ((parse_item parse_term (),get_num())),get_num())
| "APTHM" => (ApThm ((get_num()),(parse_item parse_term ())))
| "CCONTR" => (Ccontr ((parse_item parse_term ()),get_num()))
| "CHOOSE" =>
  let val (tm,n1) = parse_pair ((parse_item parse_term),get_num) ()
  in
  (Choose (tm, n1, get_num()))
end

| "CONJ" => (Conj (get_num(),get_num()))
| "CONJUNCT1" => (Conjunct1 (get_num()))
| "CONJUNCT2" => (Conjunct2 (get_num()))
| "CONTR" => (Contr ((parse_item parse_term ()), get_num()))
| "DISJ1" => (Disj1((get_num()),(parse_item parse_term ())))
| "DISJ2" => (Disj2 ((parse_item parse_term ()), get_num()))
| "DISJCASES" => (DisjCases (get_num(), get_num(), get_num()))
| "EQIMPRULEL" => (EqImpRuleL (get_num()))
| "EQIMPRULER" => (EqImpRuleR (get_num()))
| "EQMP" => (EqMp (get_num(),get_num()))
| "EQTINTRO" => (EqIntro (get_num()))
| "ETACONV" => (EtaConv (parse_item parse_term ()))
| "EXISTS" => (Exists (parse_pair(parse_item parse_term,parse_item parse_term()),
                         (get_num())))
| "EXT" => (Ext (get_num()))
| "GEN" => (Gen ((parse_item parse_term ()), get_num()))
| "IMPANTISYMRULE" => (ImpAntisymRule (get_num(),get_num()))
| "IMPTRANS" => (ImpTrans (get_num(),get_num()))
| "INST" => (Inst ((parse_list_pairs
                      (parse_item parse_term,parse_item parse_term) (),
                      (get_num()))))
| "MKBABS" => (MkBabs (get_num()))
| "MKCOMB" => (MkComb (get_num(),get_num()))
"MKEXISTS" => (MkExists (get_num()))
"NDELM" => (NotElim (get_num()))
"NINTRO" => (NotIntro (get_num()))
"SPEC" => (Spec ((parse_item parse_term ()), (get_num())))
"SUBS" => (Subs ((parse_num_list ()), (get_num())))
"SUBSTOCCS" =>
    let
      val nln1 = parse_list (parse_subs_list ())
    in
      (SubstOcss (nln1, (get_num())))
    end
"SUBSTCONV" =>
    let
      val ntl = parse_list_pairs (get_num, (parse_item parse_term)) ()
      val tm1 = parse_item parse_term ()
      val tm2 = parse_item parse_term ()
    in
      (SubstConv (ntl, tm1, tm2))
    end
"SYM" => (Sym (get_num()))
"TRANS" => (Trans (get_num(), get_num()))
_ => raise (PASS2_ERR{function="parse_just",
          message="Unknown justification"})
end

A little auxiliary parsing function for substitution lists.

and parse_subs_list () =
  ((parse_num_list ()), get_num())

The functions for parsing theorems, terms and types, namely parse_thm, parse_term and
parse_type respectively, are the same as their counterparts in the module Pass1.

and parse_thm () =
  (get_tag Keyword.THM;
    let val hyp = parse_list parse_term ()
    val concl = parse_item parse_term ()
    in(if (debug > 2) then write_out("<THM>") else ();
      Hthm.mk_thm(hyp,concl)
    end )
and parse_term() =
  let val tag = get_name() in
    if(tag = Keyword.VAR) then
      let val name = get_name()
        val ty = parse_item parse_type ()
        in (if (debug > 2) then write_out("-"-name) else ();
            mk_var(name,ty))
      end
    else if (tag = Keyword.CONST) then
      let val name = get_const_name()
        val ty = parse_item parse_type ()
        in (if (debug > 2) then write_out("$"-name) else ();
            mk_const(name,ty))
      end
    end
else if (tag = Keyword.APP) then
    let val rator = parse_item parse_term ()
    val rand = parse_item parse_term ()
    in (if (debug > 2) then write_out("=") else ();
        mk_comb(rator, rand))
    end
else if (tag = Keyword.ABS) then
    let val v = parse_item parse_term ()
    val body = parse_item parse_term ()
    in (if (debug > 2) then write_out("\d") else ();
        mk_abs(v, body))
    end
else raise (PASS2_ERR{function="parse_term",
        message="Unknown term""tag"})
end

and parse_type () =
    let val tag = get_name() in
    if (tag = Keyword.TYVAR) then
        let val name = get_tyvar_name()
        in (if (debug > 2) then write_out("#" name) else ();
            mk_vartype name)
    end
else if (tag = Keyword.TYCONST) then
    let val name = get_name()
    in (if (debug > 2) then write_out("#" name) else ();
        mk_type {Tyargs=[], Tyop=name})
    end
else if (tag = Keyword.TYOP) then
    let val name = get_name()
    val ty1 = parse_list parse_type ()
    in (if (debug > 2) then write_out("&" name) else ();
        mk_type {Tyargs=[ty1], Tyop=name})
    end
else raise (PASS2_ERR{function="parse_type",
        message="Unknown type""tag"})
end

5.1.2 The user function This function is the entry point to the Pass 2 parser. It is similar to the entry function in the module Pass1. It takes the name of the proof file as its sole argument. This string should include all the suffixes and possibly the path of the file name. It calls the function mk_command in the Io module to work out the name of the log file and of any compressing and decompressing command if they are necessary. See Page 141 for details of how the file name is interpreted.

fun parse_file fname =
    (Parsing.init();
    Henv.init(); Proof.clear_proof(); Check.init();
    let val (outname, incmd, outcmd) = Io.mk_command fname
    in
    Io.open_input_socket fname incmd;
    Io.open_output_socket outname outcmd;
Report.write_log_preamble ()
end;
let val plst = parse_proof_file () in
  (Io.close_io_socket(); (plst))
end) handle e =>
  (Io.close_io_socket(); raise e)
end; (* functor Pass2FUN *)
The parsing functions

The Parsing module includes all the lexical analysis and parsing functions. The parsing of the proof files can be considered in two tiers: the lexical analysis and parsing. Due to the slowness of string functions in SML, we do not work on strings. Instead, the input bytes returned by the input function are converted to abstract characters represented by the type ctype. These characters are converted to strings only when it is necessary.

Functions provided by this module are used by both passes. They can be divided into three groups: the character functions which convert input bytes to abstract characters and are local to this module, the atom functions which recognise atoms, such as identifiers and numbers, and the parsing functionals which recognise larger syntactic structures, such as lists and pairs.

The module is implemented by a functor, ParsingFUN. The interface is specified by the signature Parsing_sig.

6.1 The specification

The signature Parsing_sig specifies the interface of the Parsing module. The structure Io provides the input functions to the parser. The structure Keyword defines the concrete input syntax.

The type ctype is an abstract representation of input characters. The parser works with abstract characters except in certain special situations.

signature Parsing_sig =

sig
  structure Io : IO_sig
  structure Keyword : Keyword_sig

exception PARSE_DONE

datatype ctype = ALPHA of string | NUM of int | SYM of string
  | LP | RP | LB | RB | LC | RC | PL | MI | EOF | SP;

The first group of functions provided by this module are for parsing atoms, e.g., tags, identifiers and numbers. Their names are all prefixed by get_.

val get_const_name : unit -> string
val get_name : unit -> string
val get_num : unit -> int
val get_string : unit -> ctype list
val get_symbolic_name : unit -> string
val get_tag : string -> unit
val get_tyvar_name : unit -> string

The function check_ver_string verifies the file format string.

val check_ver_string : unit -> bool

The function init initialises this module. It should be called at the beginning of every pass.

val init : unit -> unit
These functions are parsing functionals. They recognise larger syntactic structures, such as lists and pairs.

```haskell
val parse_closure : (unit -> 'a) -> unit -> 'a list
val parse_item : (unit -> 'a) -> unit -> 'a
val parse_list : (unit -> 'a) -> unit -> 'a list
val parse_num_list : unit -> int list
val parse_list_pairs : (unit -> 'a) * (unit -> 'b) -> unit -> ('a * 'b)list
val parse_pair : (unit -> 'a) * (unit -> 'b) -> unit -> 'a * 'b
```

These functions are for skipping certain input at high speed. They are used only in the first pass.

```haskell
val skip_item : unit -> unit
val skip_long_string : bool -> ctype list -> unit
val skip_sp : unit -> unit
end (* signature *)
```

### 6.2 The implementation

The functor ParsingFUN implements the Parsing module. It requires two structures: the first Io provides input and file management functions, and the second Keyword defines the concrete input syntax.

```haskell
functor ParsingFUN
  (structure Io:I0_sig and Keyword :Keyword_sig) : Parsing_sig =
  struct
    structure Io = Io
    structure Keyword = Keyword
  end
```

```haskell
exception PARSE_DONE

fun PARSING_ERR {function, message} =
  Exception.CHR_ERR{origin_structure = "Parsing",
                   origin_function = function, message=message};

val debug = Debug.get_debug("Parsing");

fun write_out s = (output(std_err, s); flush_out std_err);
```

#### 6.2.1 Abstract characters

The type `ctype` is an abstract internal representation of the input characters. The input structure `Io` has a function for inputting a fixed size block of bytes represented as a list of ASCII codes. It is very time-consuming to convert this into SML strings since the string operation is rather slow. So, we work on the abstract characters.

The type of abstract characters consists of four kinds of characters:

- **letters** represented by the constructor `ALPHA`;
- **digits** represented by the constructor `NUM`;
- **symbols** represented by the constructor `SYM`;
- **special characters** each of which has a constructor.

```haskell
datatype ctype = ALPHA of string | NUM of int | SYM of string
               | LP | RP | LB | RB | LC | RC | PL | MI | EOF | SP;
```
6.2. The Implementation

6.2.2 Mapping functions  The function byte_to_char maps a byte (ASCII code) to the abstract character of type ctyp e. Symbolic names are defined locally for some ASCII characters. Upper and lower case letters are mapped to ALPHA with single character string containing the letter. Digits are mapped to NUM with their corresponding numeric value. Other printable characters except ( ) [] {}<> are mapped to SYM with single character string containing the character. The special characters ( ) [] {}<-> are mapped to their respective symbolic names, LP, RP, LB, RB, LC, RC, FL and MI. All other input bytes are mapped to SP, the space character which is ignored in most cases. The special character EOF is used to indicate the end of file.

This function needs to be modified if the concrete syntax is changed.

```lisp
local
val ASCII_code_sp = ordof(" ",0)
and ASCII_code_0 = ordof("0",0)
and ASCII_code_9 = ordof("9",0)
and ASCII_code_a = ordof("a",0)
and ASCII_code_z = ordof("z",0)
and ASCII_code_A = ordof("A",0)
and ASCII_code_Z = ordof("Z",0);

in
fun byte_to_char n =
  if(n <= ASCII_code_sp) then (SP)
  else
    case n of
      40 => (LP) (* "(" *)
    | 41 => (RP) (* ")" *)
    | _ =>
      if(n < ASCII_code_0) then
        case n of
          43 => (PL) (* "+" *)
        | 45 => (MI) (* "-" *)
        | _ => (SYM (chr n))
      else if (n <= ASCII_code_9) then (NUM (n - ASCII_code_0))
      else if (n < ASCII_code_A) then (SYM (chr n))
      else if (n <= ASCII_code_z) then (ALPHA (chr n))
      else if (n < ASCII_code_a) then
        case n of
          91 => (LB) (* "[" *)
        | 93 => (RB) (* "]" *)
        | _ => (SYM (chr n))
      else if (n <= ASCII_code_z) then (ALPHA (chr n))
      else
        case n of
          123 => (LC) (* "{" *)
        | 125 => (RC) (* "}" *)
        | _ => (SYM (chr n))
    end (* local *)
```

The function char_to_str maps an abstract character to its name string. It is mainly used for debugging.

```lisp
fun char_to_str (LP) = "LP"
  | char_to_str (RP) = "RP"
  | char_to_str (LB) = "LB"
```
6.2.3 character functions  This section contains several low level functions which are used by the string functions and parsing functionals to obtain the next input character.

This module keeps an internal input buffer `char_buf` which is a byte array. The current length of the buffer is kept in `char_buf_len`. The identifier `charp` always points to the next available byte. The local function `chk_buf` is called every time a character is accessed. It calls the input function `Io.read_bytes` to get a new block of bytes if the end of the buffer is reached.

```plaintext
def chk_buf() =
    if (!charp >= !char_buf_len) then
        let val (len,cs) = Io.read_bytes () in
            (char_buf_len := len;
             char_buf := cs;
             charp := 0;
             if (debug > 1) then
                 write_out(    
                     ByteArray.extract((!char_buf), (!charp), (!char_buf_len)))
             else ()
             end ()
    else ()

local
    val char_buf = ref (ByteArray.array(256,0))
    and char_buf_len = ref 0
    and charp = ref 0;
```

The function `init` clears the internal input buffer. It should be called at the beginning of each pass.

```plaintext
def init() =
    (charp := 0;
     char_buf_len := 0;
     ()

The function `peek` reads the next byte without removing it from the buffer. It converts the byte into an abstract character and returns it.

```plaintext
def peek() =
    (chk_buf ();
     if (!char_buf_len = 0) then (EOF)
     else
         let
            val b = (ByteArray.sub(!char_buf, !charp))
     end ()
```

...
6.2. THE IMPLEMENTATION

```plaintext
in
  (if (debug > 3) then write_out( "\n"~(chr b)) else ();
  byte_to_char b)
end)

The function next removes the next byte from the buffer. It converts the byte into an
abstract character and returns it.

and next() =
  (chk_buf);
  if (!char_buf_len = 0) then (EOF)
  else let val c = (ByteArray.sub(!char_buf, !charp))
    in
      (charp :=(!charp) + 1;
      if (debug > 2) then write_out( "\n"~(chr c)) else ();
      byte_to_char c)
end )

The function unget puts its argument c which is an abstract character back into the buffer.
Since the contents of the buffer is not destroyed before the next call to the input function,
unget makes sure that c is the same as the character in the buffer. It should always be the
same since a character can only be put back immediately after it is removed from the buffer.
No intervening reading is allowed. It is not possible to put back a character at the beginning
of the buffer.

and unget c =
  if (!charp = 0) then
    raise (PARSING_ERR{function="unget",
    message="Can't unget--beginning of buffer")
  else if (c = byte_to_char(ByteArray.sub(!char_buf, (!charp-1)))) then
    charp := (!charp) - 1
  else raise (PARSING_ERR{function="unget",
    message="Can't unget--not the same char")

The following two functions, peekb and nextb, are the same as peek and next except their
return values are bytes instead of abstract characters.

and peekb() =
  (chk_buf);
  if (!char_buf_len = 0) then 0
else
  let val b = (ByteArray.sub(!char_buf, !charp))
    in
      (if (debug > 3) then write_out( "\n"~(chr b)) else ();
      b)
end)

and nextb() =
  (chk_buf);
  if (!char_buf_len = 0) then 0
else let val c = (ByteArray.sub(!char_buf, !charp))
    in
      (charp :=(!charp) + 1;
      if (debug > 2) then write_out( "\n"~(chr c)) else ();
      c)
end)
```
The function strip removes the next byte from the internal buffer. It is usually used after a peek which finds the desired character, thus avoiding using unget.

\[
\text{and strip() = (charp := (charp + 1))}
\]

end (* local *);

6.2.4 String Functions Functions in this section are for parsing atoms. We first describe the local functions. The function skip takes a predicate \( f \) as its argument. It skips all characters that satisfy \( f \), i.e., it removes all initial characters from the internal buffer until the first character \( c \) such that evaluating \( f c \) returns false.

The function get_str is similar to skip but it returns a list of characters which satisfy the predicate \( f \). The second argument \( l \) is used for cumulating the characters so the function is more efficient. NOTE: the resulting list is in reverse order. The function get_bytes is the same as get_str except the return value is a list of bytes.

The function get_long_str returns a list of character which may contain matched pairs of delimiters. The delimiter pairs it recognises are LP and RP, LB and RB, and LC and RC. It takes two arguments. The first is a list of right (closing) delimiters, and the second is a list for cumulating the characters. It returns its second argument if the delimiter list is empty. Otherwise, it scans through the input characters. If it finds a closing delimiter which is the same as the head of the delimiter list, the head is removed. If it sees an open delimiter, the matching closing delimiter is pushed on to the delimiter list. Then, it calls itself recursively.

local

fun skip f = while (f (peek())) do strip();

fun get_str f l =
  let val c = peek() in
  if (f c) then (strip(); get_str f (c::l)) else l
  end

and get_bytes f l =
  let val c = peekb() in
  if (f c) then (strip(); get_bytes f (c::l)) else l
  end

and get_long_str [] l = l
| get_long_str (dl as (delim::ds)) l =
  let val c = next() in
  if (delim = c) then get_long_str ds (c::l)
  else
    case c of
      LP => get_long_str (RP::dl) (c::l)
    | LB => get_long_str (RB::dl) (c::l)
    | LC => get_long_str (RC::dl) (c::l)
    | EOF => raise (PARSING_ERR{function="get_long_str",
                           message="Unexpected end of file"})
    | _ => get_long_str dl (c::l)
  end

Here is a list of predicates to be used in conjunction with the above functions to get input characters.

fun is_alpha (ALPHA c) = true
| is_alpha _ = false
and is_num (NUM c) = true
6.2. THE IMPLEMENTATION

```haskell
| is_num _ = false
and is_sym (SYM c) = true
| is_sym _ = false
and is_lp (LP) = true
| is_lp _ = false
and is_rp (RP) = true
| is_rp _ = false
and is_eof (EOF) = true
| is_eof _ = false
and is_sp (SP) = true
| is_sp _ = false
and is_notrp RP = true
| is_notrp _ = false
and is_delim (LP) = true
| is_delim (RP) = true
| is_delim (LC) = true
| is_delim (RC) = true
| is_delim (LB) = true
| is_delim (RB) = true
| is_delim _ = false
val not_alpha = (not o is_alpha)
```

Here are local copies of the special characters defined in the Keyword structure. They are all ASCII codes (bytes). `idchar1` is a list of non-letters allowed in identifiers. `tyvchar` is the first character of type variables. `specials` is a list of characters allowed to be the first character of a symbolic identifier.

```haskell
val idchar1 = map ord (explode Keyword.idchars)
and tyvchar = ord Keyword.tyvar
and specials = setify (map (fn s => ordof(s,0)) Keyword.symbols);
```

The following three predicates are used in conjunction with the string functions above to get alphanumerical identifiers, type variables and symbolic identifiers, respectively.

```haskell
fun is_id_charOrd c =
  (CTYPE.isAlphaOrd c) orelse (CTYPE.isDigitOrd c) orelse
  (exists (fn x => (c = x)) idchar1)
and is_tpv_charOrd c =
  (CTYPE.isAlphaOrd c) orelse (CTYPE.isDigitOrd c) orelse
  (c = tyvchar) orelse (exists (fn x => (c = x)) idchar1)
and is_spec_charOrd c = (exists (fn x => (c = x)) specials)
```

Here are the user functions. The function `skip_sp` skips all consecutive space characters. The function `skip_long_string` skips a string until all matching delimiters in the argument `list delim` are found. The string may contain matching delimiters as well. The first argument `flag` is a boolean value if it is false, the last delimiter is not removed from the internal buffer. The function `skip_item` skips the next delimited item which may be an item enclosed in parentheses, or a list or a pair.

```haskell
fun skip_sp () = skip is_sp
and skip_long_string flag delim =
  (get_long_str delim []; if flag then () else unget(hd delim))
and skip_item () =
  (skip_sp());
```
case next() of
  (LP) => get_long_str [RP] []
  | (LC) => get_long_str [RC] []
  | (LB) => get_long_str [RB] []
  | _ => [] ;
()

The function get_string returns a string containing only letters. The function get_num returns a number containing only digits and optionally preceded by a plus or minus sign.

fun get_string () = (rev(get_str is_alpha []))
and get_num () =
  let
    fun f sum =
      case peek() of
        (NUM n) => (strip(); f (sum * 10 + n))
        | _ => sum
      in
        (skip_sp();
        case (peek()) of
          (PL) => (strip(); f 0)
          | (MI) => (strip(); "(f 0))
          | (NUM n) => (f 0)
          | _ => raise (PARSING_ERR{function="get_num",
                                      message="Expecting a number"})
        end
      end

The function get_name returns an alphanumeric identifier. The function get_tyvar_name returns a type variable name.

fun get_name () =
  (skip_sp();
  let val name = (implode o rev)(map chr(get_bytes (is_id_charOrd [])))
  in
    (if (debug > 1) then write_out("|" ^ name ^ "|") else ()
     name)
  end
and get_tyvar_name () =
  (skip_sp();
  if (tyvchar =peekb()) then
    (implode o rev)(map chr(get_bytes (is_tyv_charOrd) [nextb()])))
  else raise (PARSING_ERR{function="get_tyvar_name",
                                      message="Illegal type variable name"})
)

The function get_const_name returns a string which is a valid constant name. It may be either alphanumeric or symbolic. The function get_symbolic_name returns a symbolic identifier.

and get_const_name () =
  (skip_sp();
  if(is_id_charOrd(peekb())) then
    (implode o rev)(map chr(get_bytes (is_id_charOrd [])))
  else if (is_spec_charOrd(peekb())) then
get_symbolic_name ()
  else raise (PARSING_ERR{function="get_const_name",
    message="Illegal constant name"})
}
and get_symbolic_name () =
let fun sym n lsym =
    let val c = peekb()
    val lsym' = ListUtil.filter
      (fn s => (if (ordof(s,n) = c) then SOME s else NONE)
       handle Ord => NONE) lsym
    in
    if (debug > 2) then
      (write_out("get_sym_name "+(makestring n));
       (write_out o implode o (map chr)) l; ()
    else ();
    if (null lsym') then ((implode o rev)(map chr l))
    else (strip(); sym (n+1) (c::l) lsym')
end
in
  sym 0 [] Keyword.symbols
end
handle e => raise
  (PARSING_ERR{function="get_symbolic_name",
    message="error"})

The function check_ver_string verifies the version string which identifies the file format.

fun check_ver_string () =
  let val s = rev (tl(get_long_str [RP] [])) in
    (unget RP;
     (s = (map (byte_to_char o ord) (explode Keyword.versionName))))
  end

The function get_tag takes a string str which is the tag expected to be read from the input. It scans through the input characters comparing each character with successive character of str until the end of it. If the end of str is reached and all corresponding characters agree, the required tag is found. Otherwise it raises an exception.

fun get_tag str =
  let
    fun f [] = true
    | f (x::xs) =
        case (peek()) of
          (ALPHA c) => if(c = x) then (strip(); f xs) else false
        | _ => false
    in
    if (f (explode str)) then ()
    else raise (PARSING_ERR {function="get_tag",
                               message="Expecting tag ("""str"""")"})
  end
end (* local *)

6.2.5 Parsing functionals Functions in this section are parsing functionals. They process the higher level objects, such as an item, a list or a pair. In general, they take parsing functions
as arguments. The functionals parse the delimiters of the object and use the supplied functions to parse the sub-objects.

An item is always enclosed in a pair of parentheses. The function parse_item takes a function kind which parses the contents between the parentheses. parse_item ignores any leading blanks before the open parenthesis LP. It applies the parsing function kind to process the contents after the LP. It then skips any blanks before the closing parenthesis RP and gets the RP. An exception is raised if neither the LP nor the RP is found. If the parsing function kind fails with the special exception PARSE_DONE, we skip to the end of the item and propagate the exception. This situation occurs when all goals are proved, so the remaining proof lines are skipped over.

```ocaml
fun parse_item kind () =  
  (skip_sp ();  
    if (LP = peek()) then 
      (strip ();  
        let 
          val x = kind () 
          handle PARSE_DONE => (skip_long_string true [RP]; 
            raise PARSE_DONE)  
          | e => raise e 
        in 
          (skip_sp ();  
            if (RP = next()) then x 
            else raise (PARSING_ERR 
              {function="parse_item", message="expecting RP")}) 
        end)  
    else raise (PARSING_ERR 
      {function="parse_item", message="expecting LP")}) 
  )
```

A pair is always enclosed in a pair of braces. The function parse_pair processes a pair. The argument is a pair of parsing functions which parse the first and second field of the pair, respectively.

```ocaml
and parse_pair (f1,f2) () =  
  (skip_sp ();  
    if (LC = peek()) then 
      (strip ();  
        let 
          val x1 = f1 () and x2 = f2 () 
        in 
          (skip_sp ();  
            if (RC = next()) then (x1,x2) 
            else raise (PARSING_ERR 
              {function = "parse_pair", message = "expecting RC")}) 
        end)  
    else raise (PARSING_ERR 
      {function = "parse_pair", message = "expecting LC")}) 
  )
```

A list is always enclosed in a pair of brackets. The function parse_list parses lists. Its argument kind is a parsing function for a single element. It uses the local function get_list_items to process the elements recursively.

A closure is a list without the enclosing brackets. The function parse_closure uses the same local function as parse_list to process the elements.
6.2. THE IMPLEMENTATION

NOTE: the list of items returned is in reverse order.

local

fun get_list_items f l =
  get_list_items f ((parse_item f ()::l)
    handle PARSE_DONE => (skip_long_string false [RB]; l)
    (* raise PARSE_DONE *)
    | CHK_ERR {message = "expecting LP",
             origin_function = "parse_item",
             origin_structure = "Parsing"} => l
    | e => raise e

fun get_list_num l =
  get_list_num ((get_num ()::l)
    handle CHK_ERR {message,origin_function,origin_structure} => l
    | e => raise e

in

fun parse_list kind () =
  (skip_sp();
   if (LB = peek()) then
     (strip();
      let val items = get_list_items kind [] in
      (skip_sp();
       if (RB = next()) then items
      else raise (PARSING_ERR
      {function = "parse_list", message = "expecting RB"}))
     end)
   else raise (PARSING_ERR
   {function = "parse_list", message = "expecting LB"}))

and parse_closure kind () =
  (skip_sp(); get_list_items kind [])

and parse_num_list () =
  (skip_sp();
   if (LB = peek()) then
     (strip();
      let val items = get_list_num [] in
      (skip_sp();
       if (RB = next()) then items
      else raise (PARSING_ERR
      {function = "parse_num_list", message = "expecting RB"}))
     end)
   else raise (PARSING_ERR
   {function = "parse_num_list", message = "expecting LB"}))

end (* local * )

The function parse_list_pairs is similar to parse_list_items except that the elements are pairs. NOTE: the list of pairs returned is in reverse order.

local

fun get_list_pairs f l =
  get_list_pairs f ((parse_pair f ()::l)
    handle PARSE_DONE => (skip_long_string true [RB];
raise PARSE_DONE
| CHK_ERR {message = "expecting LC",
  origin_function = "parse_pair",
  origin_structure = "Parsing"} => 1
| e => raise e
in
fun parse_list_pairs kind () =
  (skip_sp();
   if (LB = peek()) then
     (strip();
      let val items = get_list_pairs kind [] in
        (skip_sp();
         if (RB = next()) then items
         else raise (PARSING_ERR
                      {function = "parse_list", message = "expecting RB"}))
       end)
   else raise (PARSING_ERR
                {function = "parse_list", message = "expecting LB"}))
end;(* local *)
end;(* functor Parsing *)
The module keyword specifies the concrete strings of the proof file and the log file. Throughout
the program, symbolic names defined in this module are used instead of concrete strings. If the
formats are changed, only definitions in this module need to be modified.

7.1 The specification

The signature Keyword_sig specifies a structure containing all the keywords the lexical analyser
and the parser understand. These keywords constitute the concrete syntax of the proof format.

signature Keyword_sig =
  sig
    val alphas: string list (* Alphanumeric keywords *)
    and symbols: string list (* Symbolic identifiers *)
    and idchars: string (* Special characters in alphanumeric id *)
    and tyvar: string (* The leading character of type variables *)
    and LP: string (* Begin of an item *)
    and RP: string (* End of an item *)
    and LC: string (* Begin of a pair *)
    and RC: string (* End of a pair *)
    and LB: string (* Begin of a list *)
    and RB: string (* End of a list *)
    and PLUS: string
    and MINUS: string
    and PM: string list (* The optional signs before a number *)
    and versionName: string
    and log_versionName: string
    and TYOP: string
    and TVAR: string
    and TYCONST: string
    and VAR: string
    and CONST: string
    and APP: string
    and ABS: string
    and THM: string
    and LINE: string
    and PROOF: string
    and VERSION: string
    and ENV: string
    and TIMESTAMP: string
    and USED: string
    and PROVED: string
    and UNSOLVED: string
end;
7.2 The implementation

Here is the keyword structure. alphas is a list of tags, i.e., the first atom in an expression that identifies what kind of expression this is. symbols is a list of allowable symbolic identifiers. All initial substrings of these strings can be the names of constants. idchars are characters, in addition to letters and digits, allowed in alphanumeric identifiers. tyvar is the initial character of a type variable. LP and RP are the left and right parentheses. They enclose an expression. LC and RC are the left and right braces which enclose a pair. LB and RB are the left and right brackets which enclose a list. PM is a list of characters which may appear in front of a number, i.e., the optional signs. versionName is a string identifying the version of the input file format. log_versionName is a string identifying the version of the output file format.

VERSION is a special tag. All characters after the blanks following this tag until the closing RP are taken as a single string. A list of symbolic names are defined for the tags of all kinds of expressions. They are exported and used by the parser.

structure HOLProofKey :Keyword.sig =
struct

and symbols = ["=*", "*+", "*--", "<=--", "<-->", "<-->", "=>", "<>", "<>="]

and idchars = ", /%"

and tyvar = "+="

and LP = "(" and RP = ")"

and LC = "{" and RC = "}"

and LB = "[" and RB = "]"

and PLUS = "+=" and MINUS = "--"

and PM = ["--", "+="] (* There should be no space between these signs
 and the following digits. The first should be
 the minus sign, the second the plus. *)

and versionName = "PRF FORMAT 1.0 EXTENDED"

and log_versionName = "HOL CHECKER 0.1";

val idchar_codes = map ord (explode idchars)

and tyvar_code = ord tyvar;

val [TYOP, TYVAR, TYCONST, VAR, CONST, APP, ABS,
 THM, LINE, PROOF, VERSION, ENV,
 TIMESTAMP, USED, PROVED, UNSOLVED] = alphas;

end;
The checking rules

The module Check is the critical part of the checker. It implements the checking rules. For each basic inference rule basicrule, a checking function chk_basicrule is defined. The functions return true if the inference is correct, otherwise, they return false. If the theorem is not in the expected form, an exception is raised due to the failure in pattern matching.

8.1 The specification

signature Check_sig =
  sig
  structure Report: Report_sig
  structure Htype: Htype_sig
  structure Hterm: Hterm_sig
  structure Hthm: Hthm_sig
  structure Henv: Henv_sig
  structure Proof: Proof_sig
  structure HtermSet: ORD_SET

The main functions provided by this module are init which should be called to initialise the module at the beginning of a proof; chk_pline which is called for each proof line; and chk_proof which is called at the end of each proof.

  val init : unit -> unit

  val chk_pline: (int * Proof.justification * Hthm.hthm) -> int

  val chk_proof: (string * Hthm.hthm list * int list) -> unit

Below are checking functions for each kind of justification. They do not need to be exported. They should be removed from the signature when the program is stable.

  val chk_Abs: (int * Hterm.hterm * int * Hthm.hthm) -> bool

  val chk_Assume: (Hterm.hterm * Hthm.hthm) -> bool

  val chk_BetaConv: (Hterm.hterm * Hthm.hthm) -> bool

  val chk_Disch: (int * Hterm.hterm * int * Hthm.hthm) -> bool

  val chk_InstType:
      (int * (Htype.htype * Htype.htype)list * int * Hthm.hthm) -> bool

  val chk_Mp: (int * int * int * Hthm.hthm) -> bool

  val chk_Refl: (Hterm.hterm * Hthm.hthm) -> bool

  val chk_Subst:
      (int * (int * Hterm.hterm)list * Hterm.hterm * int * Hthm.hthm) -> bool

  val chk_Axiom: (string * string * Hthm.hthm) -> bool

  val chk_Definition: (string * string * Hthm.hthm) -> bool

  val chk_DefExistsRule: (Hterm.hterm * Hthm.hthm) -> bool

  val chk_Hypothesis: Hthm.hthm -> bool

  val chk_NewAxiom: (string * Hterm.hterm * Hthm.hthm) -> bool

45
val chk_NewConstant: (string * Htype.htype) -> bool
val chk_NewType: (int * string) -> bool
val chk_Numconv: (Hterm.hterm * Htm.hthm) -> bool
val chk_StoreDefinition: (string * Hterm.hterm * Htm.hthm) -> bool
val chk_Theorem: (string * string * Htm.hthm) -> bool

val chk_AddAssum: (int * Hterm.hterm * int * Htm.hthm) -> bool
val chk_Alpha: (Hterm.hterm * Hterm.hterm * Htm.hthm) -> bool
val chk_ApTerm: (int * Hterm.hterm * int * Htm.hthm) -> bool
val chk_ApThm: (int * int * Hterm.hterm * Htm.hthm) -> bool
val chk_Constr: (int * Hterm.hterm * int * Htm.hthm) -> bool
val chk_Choose: (int * Hterm.hterm * int * int * Htm.hthm) -> bool
val chk_Conj: (int * int * int * Htm.hthm) -> bool
val chk_Conjunct1: (int * int * Htm.hthm) -> bool
val chk_Conjunct2: (int * int * Htm.hthm) -> bool
val chk_Disj1: (int * int * Hterm.hterm * Htm.hthm) -> bool
val chk_Disj2: (int * Hterm.hterm * int * Htm.hthm) -> bool
val chk_DisjCases: (int * int * int * int * Htm.hthm) -> bool
val chk_EqImpRuleL: (int * int * Htm.hthm) -> bool
val chk_EqImpRuleR: (int * int * Htm.hthm) -> bool
val chk_EqMp: (int * int * int * Htm.hthm) -> bool
val chk_EqIntro: (int * int * Htm.hthm) -> bool
val chk_EtaConv: (Hterm.hterm * Htm.hthm) -> bool
val chk_Exists: (int * Hterm.hterm * Hterm.hterm * int * Htm.hthm) -> bool

val chk_Ext: (int * int * Htm.hthm) -> bool
val chk_Gen: (int * Hterm.hterm * int * Htm.hthm) -> bool
val chk_ImplantisymRule: (int * int * int * Htm.hthm) -> bool
val chk_ImplTrans: (int * int * int * Htm.hthm) -> bool
val chk_Inst: (int * (Hterm.hterm * Hterm.hterm) list * int * Htm.hthm) -> bool
val chk_MkAbs: (int * int * Htm.hthm) -> bool
val chk_MkComb: (int * int * int * Htm.hthm) -> bool
val chk_MkExists: (int * int * Htm.hthm) -> bool
val chk_NotElim: (int * int * Htm.hthm) -> bool
val chk_NotIntro: (int * int * Htm.hthm) -> bool
val chk_Spec: (int * Hterm.hterm * int * Htm.hthm) -> bool
val chk_Subs: (int * int list * int * Htm.hthm) -> bool
val chk_SubsOccs: (int * (int list * int) list * int * Htm.hthm) -> bool
val chk_SubstConv: (int * (int * Hterm.hterm) list * Hterm.hterm * Hterm.hterm * Htm.hthm) -> bool
val chk_Sym: (int * int * Htm.hthm) -> bool
val chk_Trans: (int * int * int * Htm.hthm) -> bool

end

8.2 The implementation

functor CheckFUN (structure Report: Report_sig and Htype:Htype_sig
and Hterm:Hterm_sig and Htm: Htm_sig
and Henv: Henv_sig and Proof: Proof_sig
and HtermSet: ORD_SET

8.2. THE IMPLEMENTATION

and Htype = Hterm.Htype = Hthm.Hterm.Htype =
Henv.Htype = Proof.Htype
and Hthm = Proof.Hthm
and HtermSet = Hthm.HtermSet
sharing type Hterm.hterm = HtermSet.item) : Check_sig =

struct
structure Report = Report;
structure Htype = Htype;
structure Hterm = Hterm;
structure Hthm = Hthm;
structure Henv = Henv;
structure Proof = Proof;
structure HtermSet = HtermSet;

open Htype Hterm Hthm Henv Proof;

fun Check_ERR(function,message) =
Exception.CHK_ERR(message = message,
origin_function = function,
origin_structure = "Check");

val debug = ref 0;
fun write_out s = (output(std_err, s); flush_out std_err);
fun init () = (
  debug := Debug.get_debug "Check";
  ()
);

8.2.1 Functions for outputting to log file These two functions are convenient for writing to the log file. The tag is the name of the justification. The function write_used is used for the justifications DEFINITION and THEOREM. The first string s1 is the name of the theory and the second string s2 is the name of the theorem. The function write_used2 is for the justifications NEWAXIOM and STOREDEFINITION.

local
open Report
open Keyword Io
in
fun write_used (tag,s1,s2,thm) =
  (write_output_string (LP " tag " " s1 " " s2);
   Hthm.pr_hthm thm;
   write_closing_line ());
and write_used2 (tag,s1,tm,thm) =
  (write_output_string (LP " tag " " s1);
   Hterm.pr_hterm tm;
   Hthm.pr_hthm thm;
   write_closing_line ());
end
8.3 Checking functions

All checking functions are described below in detail. They are grouped into three subsections: primitive rules, miscellaneous functions and derived rules. For each rule, the syntax in the proof file is given in a box first. It is followed by a specification, the name of the rule and the type of the ML function which implements the rule in HOL and a more detail description. The checking function is shown at the end. In the case of a derived rule, a proof in terms of primitive rules and other derived rules is given. Care has been taken not to introduce circular dependency in these proofs. Some of the proofs were adapted from [5] with some modifications.

8.3.1 Primitive rules

- Abstraction

\[ \Gamma \vdash t_1 = t_2 \quad \Rightarrow \quad \Gamma \vdash (\lambda x.t_1) = (\lambda x.t_2) \]

ABS : term -> thm -> thm

The term argument must be a variable, and must not be free in the assumptions of the hypothesis \( \Gamma \). The NUMBER refers to the theorem in a line having the given line number. This theorem must be an equation.

fun chk_Abs (line,term,n1,thm) =
  let val thml = (get_thm (line,n1))
      val hyp1 = hyp thml and con1 = concl thml
      val (left,right) = dest_eq (concl thm)
      val (lv,lbody) = dest_abs left and (rv,rbody) = dest_abs right
      val hyps = hyp thm
  in
    well_typed term andalso is_var term andalso
    is_eq con1 andalso (HtermSet.equal(hyp1, hyps)) andalso
    (lv = rv) andalso (term = rv) andalso
    not(mem lv (frees1 (HtermSet.listItems hyps))) andalso
    (lbody = (lhs con1)) andalso (rbody = (rhs con1))
  end;

- Assumption introduction

\[ t \vdash t \]

ASSUME : term -> thm

The term \( t \) must be of type : bool. There should be only a single assumption, and it must be the same as the conclusion.

fun chk_Assume (term, thm) =
  let val [ass] = HtermSet.listItems (hyp thm)
      val conc = concl thm
  in
    is_bool_ty term andalso (conc = term) andalso (conc = ass)
  end;
8.3. CHECKING FUNCTIONS

- **β-conversion**

  \[ \vdash (\lambda x.t_1)t_2 = t_1[t_2/x] \]

  \[ \text{BETA_CONV : term -> thm} \]

  The term argument must be a β-redex in the form \((\lambda x.t_1)t_2\). The right-hand side of the resulting theorem is obtained by substituting \(t_2\) for \(x\) in \(t_1\) with suitable renaming of variables in \(t_1\) to avoid variable capture.

  \[ \text{fun chk_BetaConv (term, thm) =} \]
  \[ \text{let val (rat,rnd) = dest_comb term} \]
  \[ \text{val (v,body) = dest_abs rat} \]
  \[ \text{val conc = concl thm} \]
  \[ \text{in} \]
  \[ \text{well_typed term andalso} \]
  \[ \text{is_eq conc andalso ((lhs conc) = term) andalso} \]
  \[ \text{(HtermSet.isEmpty (hyp thm)) andalso} \]
  \[ \text{term_subst_chk (((v,rnd),v)] body (rhs conc) body} \]
  \[ \text{end;} \]

- **Discharging an assumption**

  \[ \text{DISCH term NUMBER} \]

  \[ \frac{\Gamma \vdash t_2}{\Gamma - \{t_1\} \vdash t_1 \supset t_2} \]

  \[ \text{DISCH : term -> thm -> thm} \]

  The input term \(t_1\) must be of type : bool. The NUMBER refers to the theorem in the line having the given number. The expression \(\Gamma - \{t_1\}\) denotes the set subtraction of \(\{t_1\}\) from \(\Gamma\). If \(t_1\) is not in \(\Gamma\), the result of the subtraction is \(\Gamma\) itself.

  \[ \text{fun chk_Disch (line, term, n1, thm) =} \]
  \[ \text{let val thm1 = get_thm (line,n1)} \]
  \[ \text{val (ante,conc) = dest_imp (concl thm)} \]
  \[ \text{in} \]
  \[ \text{is_bool_ty andalso} \]
  \[ \text{(ante = term) andalso (conc = (concl thm1)) andalso} \]
  \[ \text{(HtermSet.equal(hyp thm),} \]
  \[ \text{(HtermSet.difference(hyp thm1),(HtermSet.singleton term)))})} \]
  \[ \text{end;} \]

- **Type instantiation**

  \[ \text{INSTTYPE type_type.list NUMBER} \]

  \[ \frac{\Gamma \vdash t}{\Gamma \vdash t[\sigma_1,\ldots,\sigma_n/\alpha_1,\ldots,\alpha_n]} \]

  \[ \text{INST_TYPE : (type # type)list -> thm -> thm} \]

  The first argument is a list of type pairs \([\sigma_1,\alpha_1];\ldots;[\sigma_n,\alpha_n]\) which specifies the simultaneous type substitutions to be made in the theorem referred to by NUMBER. The second fields \(\alpha_i\) of the pairs must be type variables, and \(\alpha_i\) must occur in any assumption in \(\Gamma\).
All occurrences of $\alpha_i$ in $t$ are replaced by the corresponding $\sigma_i$. Furthermore, if distinct variables in $t$ become identical after the instantiation, they will be renamed.

```haskell
fun chk_instType (line, type_type_list, n1, thm) = let
  val thml = get_thm (line,n1)
  val tyvar_list = map snd type_type_list
  val hypp = hyp thm
  val hypl = HtermSet.listItems hypp
  in
    every is_vartype tyvar_list andalso
    every (type_OK o fst) type_type_list andalso
    HtermSet.equal(hypp, (hyp thml)) andalso
    null(intersect tyvar_list (tyvars1 hypl)) andalso
    term_inst_chk type_type_list (frees1 hypl) (concl thm) (concl thml)
  end;
```

- **Modus Ponens**

<table>
<thead>
<tr>
<th>MP NUMBER NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma_1 \vdash t_1 \supset t_2 \quad \Gamma_2 \vdash t_3$</td>
</tr>
<tr>
<td>$\Gamma_1 \cup \Gamma_2 \vdash t_2$</td>
</tr>
</tbody>
</table>

**MP** : thm $\rightarrow$ thm $\rightarrow$ thm

The theorem referred to by the first NUMBER must be an implication. The theorem referred to by the second NUMBER must match the antecedent of the first theorem exactly.

```haskell
fun chk_Mp (line, n1,n2,thm) = let
  val thml = get_thm (line,n1) and thm2 = get_thm (line,n2)
  val (ante,conc) = dest_imp (concl thml)
  in
    (conc = concl thm) andalso (ante = concl thm2) andalso
    HtermSet.equal((hyp thm), (HtermSet.union((hyp thml),(hyp thm2))))
  end;
```

- **Reflexivity**

<table>
<thead>
<tr>
<th>REFL term</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\vdash t = t$</td>
</tr>
</tbody>
</table>

**REFL** : term $\rightarrow$ thm

The term $t$ must be of type : bool. The resulting theorem must be an equation with identical terms on both sides.

```haskell
fun chk_Refl (term,thm) = let
  val conc = concl thm
  in
    well_typed term andalso HtermSet.isEmpty(hyp thm) andalso
    is_eq conc andalso (lhs conc = term) andalso (rhs conc = term)
  end;
```
8.3. CHECKING FUNCTIONS

- **Substitution**

  \[
  \frac{\Gamma_1 \vdash t_1 = t'_1 \quad \cdots \quad \Gamma_n \vdash t_n = t'_n \quad \Gamma \vdash t[t_1, \ldots, t_n]}{\Gamma_1 \cup \cdots \cup \Gamma_n \cup \Gamma \vdash t[t'_1, \ldots, t'_n/t_1, \ldots, t_n]}
  \]

  \[
  \text{SUBST : (thm # term)list } \rightarrow \text{ term } \rightarrow \text{ thm } \rightarrow \text{ thm}
  \]

  The first argument is a list of pairs whose first fields are equational theorems \( t_i = t'_i \), and whose second fields are simple variables \( x_i \). The second argument is a term of the form \( t[x_1, \ldots, x_n] \). It should match the conclusion of the theorem referred to by NUMBER. The variables \( x_i \) in the term mark the places where substitutions using the theorems \( \vdash t_i = t'_i \) are to be done. The type of \( x_i \) must be the same as of \( t_i \). Variables in \( t \) may be renamed to avoid capture.

  ```
  fun chk_Subst (line, num_term_list, term, thm1, thm) = 
  let val thms = get_thm (line, ni) 
    val thms = map (fn n => get_thm(line,(fst n))) num_term_list 
    val hyps = fold HtermSet.union (map hyp thms) HtermSet.empty 
    val lrl = map (dest_eq o concl) thms 
    val (_,vl) = ListUtil.unzip num_term_list 
    in 
    every (is_eq o concl) thms andalso 
    HtermSet.equal(hyps,(HtermSet.union(hyps,(hyps, concl thm1)))) andalso 
    term_subst_chk (ListUtil.zip(lrl,vl)) term (concl thm) (concl thm1) 
    end;
  ```

8.3.2 Miscellaneous Functions

- **Retrieving an axiom**

  \[
  \text{AXIOM \ STRING \ STRING}
  \]

  \text{axiom : string } \rightarrow \text{ string } \rightarrow \text{ thm}

  This justification indicates that the theorem in this inference step is an axiom whose name is the second string and which is stored in the theory specified as the first string. The well-typedness of the axiom should be checked. An entry is written to the log file.

  ```
  fun chk_Axiom (s1,s2,thm) = 
  (write_used ("AXIOM", s1, s2, thm); 
   is_bool_ty(concl thm) andalso 
   every is_bool_ty (HtermSet.listItems(hyp thm)))
  ```

- **Retrieving a definition**

  \[
  \text{DEFINITION \ STRING \ STRING}
  \]

  \text{definition : string } \rightarrow \text{ string } \rightarrow \text{ thm}

  This justification indicates that the theorem in this inference step is a previously defined definition whose name is the second string and which is stored in the theory specified as the first string. The well-typedness of the theorem should be checked. An entry is written to the log file.

  ```
  fun chk_Definition (s1,s2,thm) = 
  (write_used ("DEFINITION", s1, s2, thm); 
   is_bool_ty(concl thm) andalso 
   every is_bool_ty (HtermSet.listItems(hyp thm)))
  ```
• Create a definitional theorem

    DEF_EXISTS_RULE term

    DEF_EXISTS_RULE : term -> thm

    This justification introduces a new definitional theorem. The input term must be an
    equation and both sides must be of the same type.

    fun chk_DefExistsRule (tm,thm) =
      let val (left,right) = dest_eq (snd(strip_forall tm))
           val (l,r) = dest_eq(snd(strip_forall(snd(dest_exists(concl thm))))))
           in
           (well_typed left) andalso (well_typed right) andalso
           ((type_of left) = (type_of right)) andalso
           (left = l) andalso (r = right)
           end

• Hypothesis

    HYPOTHESIS

    This justification indicates that the theorem is one of the initial theorems of the proof. It
    should have been proven in a previous proof. An entry is written to the log file.

    fun chk_Hypothesis thm =
      (write_used ("HYPOTHESIS", "", "", thm);
       is_bool_ty (concl thm))

• Introducing a new axiom

    NEWAXIOM STRING term

    \[ \forall x_1 \ldots x_n. t[x_1, \ldots , x_n] \]

    new_axiom : (string # term) -> thm

    This justification introduces a new axiom. It is in the form given as the term. The
    STRING is the name of the axiom. All free variables in the input term are automatically
    generalised. An entry is written to the log file.

    fun chk_NewAxiom (s,term,thm) =
      let val (vs,body) = (strip_forall (concl thm)) in
      write_used2 ("NEWAXIOM", s, term, thm);
      (term = body) andalso (every (fn x => mem x vs) (frees term))
      end

• Introducing a new constant

    NEWCONSTANT STRING type

    new_constant : string -> type -> void

    This justification declares a new constant whose name is the given STRING and whose type
    is the given type. The name should be unique, i.e., different from any existing constant,
    and the type should be well-formed. The current signature should be updated. There
    should be no conclusion (derived theorem) in this inference step. To satisfy the type
    checker, the theorem TRUTH is used as a dummy.

    fun chk_NewConstant (string,ty) =
      if (type_OK ty) then
8.3. CHECKING FUNCTIONS

let val _ = add_const(string,ty)
handle (CHK_ERR{origin_function=f,
                origin_structure=s,message=m}) =>
    raise (Check_ERR{function="chk_NewConstant",
                      message=(s"".""f"":"m")})
| e => raise e
in true end
else raise (Check_ERR{function="chk_NewConstant",
                      message="type of new constant not well-formed"})

• Introducing a new type

| NEWTYPE NUMBER STRING |

new_type : int -> string -> void

This justification declares a new type constructor whose name is the given STRING and
whose arity is the given NUMBER. The name should be unique, i.e., different from any
existing type constructor. The current type structure should be updated. The theorem
in this inference step is not used. To satisfy the type checker, the theorem TRUTH is used
as a dummy.

fun chk_NewType (num,string) =
let val _ = add_type(string,num)
handle (CHK_ERR{origin_function=f,origin_structure=s,message=m}) =>
    raise (Check_ERR{function="chk_NewType",
                      message=(s"".""f"":"m")})
| e => raise e
in true end

• Definition of non-zero numbers

| NUMCONV term |

\[ \vdash n = SUC m \]

num_CONV : term -> thm

The input term must be a constant denoting a non-zero natural number. m is a numeric
constant denoting the predecessor of n.

fun chk_Numconv (term,thm) =
let val (left,right) = dest_eq (concl thm)
val n = StringCvt.atoi(fst(dest_const left))
and (s,m) =
((fst o dest_const) ## (StringCvt.atoi o fst o dest_const))
(dest_comb right)
in
(term = left) andalso (n = (m + 1)) andalso (s = "SUC")
end

• Storing a definition

| STOREDEFINITION STRING term |

store_definition : (string # term) -> thm

This justification introduces a new definition. In fact, making a new definition is a three-
step process:
1. a theorem asserting the existence of the definition is derived with the justification
   DEFEXISTSRULE;

2. a new constant is declared with the justification NEWCONSTANT;

3. the definition is saved in the current theory with the justification STOREDEFINITION.

Since this file format allows only the four kinds of primitive terms, special syntactic status
of constants, i.e., infix or binder, are not recognised. The input term must be a series
of conjunctions. Each conjunct is an equation and both sides are of the same type. An
entry is written to the log file.

fun chk_StoreDefinition (s,term,thm) = 
  let
    fun chk_tm ((l,r),t) =
      t andalso (well_typed l) andalso (well_typed r) andalso
      (type_of l) = (type_of r))
    val conja = strip_conj (snd(strip_forall term))
    val lrlst = map (dest_eq o snd o strip_forall) conjs
    in
      write_used2 ("STOREDEFINITION", s, term, thm);
      (fold chk_tm lrlst true) andalso (term = (concl thm))
  end

- **Retrieving a theorem**

  ```
  THEOREM STRING
  theorem : string -> string -> thm
  ```

  This justification indicates that the theorem in this inference step has been derived pre-
  viously. It has been stored in the theory, whose name is the first string, under the name
  specified as the second string. The well-typedness of the theorem should be checked. An
  entry is written to the log file.

  ```
  fun chk_Theorem (s1,s2,thm) =
    (write_used ("THEOREM", s1, s2, thm);
     is_bool_ty(concl thm) andalso
     every is_bool_ty (HtermSet.listItems(hyp thm)))
  ```

8.3.3 Derived Rules

- **Adding an assumption**

  ```
  ADDASSUM term NUMBER
  ```

  $\frac{\Gamma \vdash t}{\Gamma, t' \vdash t}$

  ```
  ADD_ASSUM : term -> thm -> thm
  ```

  The input term is the new assumption $t'$ to be added to the theorem. It must be of type
  bool.

1. $t' \vdash t'$
2. $\Gamma \vdash t$ [ASSUME]
3. $\Gamma \vdash t' \supset t$ [Hypothesis]
4. $\Gamma, t' \vdash t$ [DISCH 2]
5. $\Gamma, t' \vdash t$ [MP 3,1]
fun chk_AddAssum (line, term, n, thm) = 
  let val thm1 = get_thm(line, n)
  in
    (concl thm = concl thm1) andalso (ty_is_bool(type_of term)) andalso
    (HtermSet.equal(HtermSet.delete((hyp thm), term)), (hyp thm1))
  end

• α-conversion

\[ \vdash t_1 = t_2 \]

**ALPHA** : term \(\rightarrow\) term \(\rightarrow\) thm

The input terms \(t_1\) and \(t_2\) must be α-equivalent, otherwise, it fails.

fun chk_Alpha (term1, term2, thm) = 
  let val (left,right) = dest_eq (concl thm)
  in
    well_typed term1 andalso well_typed term2 andalso
    aconv left right andalso (term1 = left) andalso (term2 = right)
  end

• **Application of a term to a theorem**

\[ \Gamma \vdash t_1 = t_2 \]

\[ \Gamma \vdash tt_1 = tt_2 \]

**AP_TERM** : term \(\rightarrow\) thm \(\rightarrow\) thm

The input term must have a function type whose domain is the type of the left-hand side
(or right-hand side as the type of both sides must be the same) of the input theorem.

1. \(\Gamma \vdash t_1 = t_2\) [Hypothesis]
2. \(\vdash tt_1 = tt_1\) [REFL]
3. \(\Gamma \vdash tt_1 = tt_2\) [SUBST 1,2]

fun chk_ApTerm (line, term, n, thm) = 
  let val ((lt,left), (rt,right)) = 
    (dest_comb ## dest_comb)(dest_eq(concl thm))
  val thm1 = get_thm(line, n)
  val (lft,rgt) = dest_eq (concl thm1)
  in
    (lt = rt) andalso (term = lt) andalso
    (lft = left) andalso (rgt = right) andalso
    (HtermSet.equal((hyp thm), (hyp thm1))) andalso
    (domain_of(type_of term) = (type_of left))
  end
• Application of a theorem to a term

\[ \Gamma \vdash t_1 = t_2 \]
\[ \Gamma \vdash t_1 = t_2 \]

AP_THM : thm → term → thm

The input term must have the same type as the domain of the left-hand side (or the right-hand side) of the input theorem.

1. \[ \Gamma \vdash t_1 = t_2 \] [Hypothesis]
2. \[ \vdash t_1 \vdash t_1 = t_1 \] [REFL]
3. \[ \Gamma \vdash t_1 = t_2 \] [SUBST 1,2]

fun chk_ApThm (line, n, term, thm) =
  let val ((left,lt), (right,rt)) =
    (dest_comb # dest_comb)(dest_eq(concl thm))
  in
  val thml = get_thm(line, n) val (lt,rgt) = dest_eq (concl thml)
  (lt = rt) andalso (lt = term) andalso
  (left = lift) andalso (right = rgt) andalso
  (HtermSet.equal((hyp thm), (hyp thml))) andalso
  (domain_of(type_of left) = (type_of term))
  end

• Classical contradiction rule

\[ \Gamma \vdash F \]
\[ \Gamma \vdash \neg \neg t \]

CCONTR : term → thm → thm

The theorem referred to by the NUMBER must have F as its conclusion. The input term \( t \) should be of type :bool, and the negation of it should occur in the assumption of the input theorem.
8.3. CHECKING FUNCTIONS

1. \( \vdash \neg \lambda b.b \supset F \)  
   [Definition of \( \neg \)]  
2. \( \vdash \neg t = (\lambda b.b \supset F)t \)  
   [AP_THM 1]  
3. \( \vdash (\lambda b.b \supset F)t = t \supset F \)  
   [BETA_CONV]  
4. \( \vdash \neg t = t \supset F \)  
   [TRANS 2,3]  
5. \( \Gamma \vdash F \)  
   [Hypothesis]  
6. \( \Gamma \vdash \neg t \supset F \)  
   [DISCH 5]  
7. \( \Gamma \vdash (t \supset F) \supset F \)  
   [SUBST 4,6]  
8. \( t = F \vdash t = F \)  
   [ASSUME]  
9. \( \Gamma, t = F \vdash (F \supset F) \supset F \)  
   [SUBST 8,7]  
10. \( F \vdash F \)  
    [ASSUME]  
11. \( \vdash F \supset F \)  
    [DISCH 10]  
12. \( \Gamma, t = F \vdash F \)  
    [MP 9,11]  
13. \( \vdash F = \forall b.b \)  
    [Definition of \( F \)]  
14. \( \Gamma, t = F \vdash \forall b.b \)  
    [SUBST 13,12]  
15. \( \Gamma, t = F \vdash t \)  
    [SPEC 14]  
16. \( \vdash \forall b.(b = T) \lor (b = F) \)  
    [Axiom EXCLUDED_MIDDLE]  
17. \( \vdash (t = T) \lor (t = F) \)  
    [SPEC 16]  
18. \( t = T \vdash t = T \)  
    [ASSUME]  
19. \( t = T \vdash t = T \)  
    [SYM 18]  
20. \( \vdash T \)  
    [Theorem TRUTH]  
21. \( t = T \vdash t \)  
    [EQ_MP 19,20]  
22. \( \Gamma \vdash t \)  
    [DISJ_CASES 17,21,15]

fun chk_Ccontr (line, term, n, thm) = 
  let val thm1 = get_thm(line, n)
      and not_term = 
        mk_comb(mk_const("\"",mk_functype(bool_ty,bool_ty)), term)
      and F = mk_const("F", bool_ty)
      val hyp1 = (HtermSet.delete((hyp thm1), not_term))
      handle NotFound => (hyp thm1)
  in
    (concl thm1 = F) andalso (concl thm = term) andalso
    (HtermSet.equal((hyp thm), hyp1))
  end

• 3-elimination

\[
\Gamma_1 \vdash \exists x. t[x] \quad \Gamma_2, t[v] \vdash t'
\]

\[
\Gamma_1 \cup \Gamma_2 \vdash t'
\]

CHOOSE : (term ≠ thm) → thm → thm

The input term must be a variable v and its type must be the same as the existentially quantified variable z in the first theorem. \( t[v] \) is a term occurring in the assumptions of the second theorem. It is the same as \( t[z] \), the body of the first theorem, up to \( \alpha \)-conversion. The variable v must not occur free in the conclusion of the first theorem, i.e., \( \exists x. t[x] \), and neither can it occur free in \( \Gamma_2 \) or \( t' \).
\[1. \Gamma \vdash \exists = \lambda P. P(\epsilon P) \quad \text{[Definition of } \exists^1] \\
2. \Gamma \vdash \exists(\lambda x. t[x]) = (\lambda P. P(\epsilon P))(\lambda x. t[x]) \quad \text{[AP_THEM 1]} \\
3. \Gamma_1 \vdash \exists(\lambda x. t[x]) \quad \text{[Hypothesis]} \\
4. \Gamma_1 \vdash (\lambda P. P(\epsilon P))(\lambda x. t[x]) \quad \text{[EQ_MP 2,3]} \\
5. \Gamma \vdash (\lambda P. P(\epsilon P))(\lambda x. t[x]) = (\lambda x. t[x])(\epsilon(\lambda x. t[x])) \quad \text{[BETA_CONV]} \\
6. \Gamma_1 \vdash (\lambda x. t[x])(\epsilon(\lambda x. t[x])) \quad \text{[EQ_MP 5,4]} \\
7. \Gamma \vdash (\lambda x. t[x])v = t[v] \quad \text{[BETA_CONV]} \\
8. \Gamma_1 \vdash t[v] = (\lambda x. t[x])v \quad \text{[SYM 7]} \\
9. \Gamma_2 \vdash t[v] = t' \quad \text{[Hypothesis]} \\
10. \Gamma_2 \vdash t[v] = t' \quad \text{[DISCH 9]} \\
11. \Gamma_2 \vdash (\lambda x. t[x])v = t' \quad \text{[SUBST 8,10]} \\
12. (\lambda x. t[x])v = (\lambda x. t[x])v \quad \text{[ASSUME]} \\
13. \Gamma_2 \vdash (\lambda x. t[x])v = t' \quad \text{[MP 11,12]} \\
14. \Gamma_1 \cup \Gamma_2 \vdash t' \quad \text{[SELECT_ELIM 6,13]} \]

fun chk_Choose(line, term, n1, n2, thm) = 
let val thm1 = get_thm(line, n1) and thm2 = get_thm(line, n2) 
  val (evar, ebody) = dest_exists(concl thm1) 
  val hyp1 = hyp thm1 and hyp2 = hyp thm2 
  val hyp21 = HtermSet.listItems hyp2 
  val wit = ListUtil.findOne (aconv ebody) hyp21 
  in 
  case wit of 
    NONE => false 
  | SOME tm =>
    let 
    val hyp2' = HtermSet.delete(hyp2, tm) 
    in 
      (is_var term) andalso not(mem term(frees concl thm1))) 
    andalso 
      (not (mem term(frees(concl thm2)))) 
      andalso 
      (not (mem term(frees(HtermSet.listItems hyp2')))) 
      andalso 
      (concl thm = concl thm2) 
      andalso 
      (HtermSet.equal(hyp thm), (HtermSet.union(hyp1, hyp2')))) 
    end 
  end

\[\text{\textbf{• \&-introduction}}\]

\[
\begin{align*}
\Gamma_1 \vdash t_1 & \quad \Gamma_2 \vdash t_2 \\
\Gamma_1 \cup \Gamma_2 \vdash t_1 \& t_2
\end{align*}
\]

CONJ : thm \(\rightarrow\) thm \(\rightarrow\) thm

The two \textsc{Numbers} refer to two theorems which are to be combined by the \& operator.

\textsuperscript{1}with suitable type instantiation.
8.3. CHECKING FUNCTIONS

1. \( \vdash \land = \lambda b_1 b_2. \forall b. (b_1 \supset (b_2 \supset b)) \supset b \)  
   [Definition of \( \land \)]
2. \( \vdash \$\Delta t_1 = \$\Delta t_1 \)  
   [REFL]
3. \( \vdash (\lambda b_1 b_2. \forall b. (b_1 \supset (b_2 \supset b)) \supset b)t_1 = \lambda b_2. \forall b. (t_1 \supset (b_2 \supset b)) \supset b \)  
   [BETA_CONV]
4. \( \vdash \$\Delta t_1 = (\lambda b_1 b_2. \forall b. (b_1 \supset (b_2 \supset b)) \supset b)t_1 \)  
   [SUBST 1,2]
5. \( \vdash \$\Delta t_1 = \lambda b_2. \forall b. (t_1 \supset (b_2 \supset b)) \supset b \)  
   [SUBST 3,4]
6. \( \vdash t_1 \land t_2 = t_1 \land t_2 \)  
   [REFL]
7. \( \vdash (\lambda b_2. \forall b. (t_1 \supset (b_2 \supset b)) \supset b)t_2 \)  
   = \( \forall b. (t_1 \supset (t_2 \supset b)) \supset b \)  
   [BETA_CONV]
8. \( \vdash t_1 \land t_2 = (\lambda b_2. \forall b. (t_1 \supset (b_2 \supset b)) \supset b)t_2 \)  
   [SUBST 5,6]
9. \( \vdash t_1 \land t_2 = \forall b. (t_1 \supset (t_2 \supset b)) \supset b \)  
   [SUBST 7,8]
10. \( t_1 \supset (t_2 \supset b) \vdash t_1 \supset (t_2 \supset b) \)  
    [ASSUME]
11. \( \Gamma_1 \vdash t_1 \)  
    [Hypothesis]
12. \( \Gamma_1, t_1 \supset (t_2 \supset b) \vdash t_2 \supset b \)  
    [MP 10,11]
13. \( \Gamma_2 \vdash t_2 \)  
    [Hypothesis]
14. \( \Gamma_1 \cup \Gamma_2, t_1 \supset (t_2 \supset b) \vdash b \)  
    [MP 12,13]
15. \( \Gamma_1 \cup \Gamma_2 \vdash (t_1 \supset (t_2 \supset b)) \supset b \)  
    [DISCH 14]
16. \( \Gamma_1 \cup \Gamma_2 \vdash \forall b. (t_1 \supset (t_2 \supset b)) \supset b \)  
    [GEN 15]
17. \( \vdash \forall b. (t_1 \supset (t_2 \supset b)) \supset b = t_1 \land t_2 \)  
    [SUBST 9,6]
18. \( \Gamma_1 \cup \Gamma_2 \vdash t_1 \land t_2 \)  
    [SUBST 17,16]

fun chk_Conj(line, n1, n2, thm) =
    let val thm1 = get_thm(line, n1) and thm2 = get_thm(line, n2)
        val hyp1 = hyp thm1 and hyp2 = hyp thm2
        and conclss = map concl [thm1, thm2]
        val (AND,conjs) = strip_comb (concl thm)
        val hypss = hyps thm
    in
        (well_typed AND) andalso (fst(dest_const AND) = "\\") andalso
        (every is_bool ty conclss) andalso (conjs = conclss) andalso
        (every is_bool ty (HtermSet.listItems hypss)) andalso
        (HtermSet.equal(hyps, (HtermSet.union(hyp1, hyp2))))
    end

• \( \land \)-elimination(left)

CONJUNCT1 NUMBER

\[
\begin{array}{c}
\Gamma \vdash t_1 \land t_2 \\
\hline
\Gamma \vdash t_1
\end{array}
\]

CONJUNCT1 : thm \rightarrow thm

This inference step extracts the left conjunct from the theorem referred to by NUMBER. The conclusion of the input theorem must be a conjunction.
1. $\Gamma \vdash \Lambda = \lambda b_1. b_2. \forall b. (b_1 \supset (b_2 \supset b)) \supset b$ [Definition of $\wedge$]
2. $\vdash \$\Lambda t_1 = \$\Lambda t_1$ [REFL]
3. $\vdash (\lambda b_1. b_2. \forall b. (b_1 \supset (b_2 \supset b)) \supset b)t_1$
   $= \lambda b_2. \forall b. (t_1 \supset (t_2 \supset b)) \supset b$ [BETA_CONV]
4. $\vdash \$\Lambda t_1 = (\lambda b_1. b_2. \forall b. (b_1 \supset (b_2 \supset b)) \supset b)t_1$
5. $\vdash \$\Lambda t_1 = \lambda b_2. \forall b. (t_1 \supset (b_2 \supset b)) \supset b$
6. $\Gamma \vdash t_1 \wedge t_2 = t_1 \wedge t_2$ [REFL]
7. $\vdash (\lambda b_2. \forall b. (t_1 \supset (b_2 \supset b)) \supset b)t_2$
   $= \forall b. (t_1 \supset (t_2 \supset b)) \supset b$ [BETA_CONV]
8. $\vdash t_1 \wedge t_2 = (\lambda b_2. \forall b. (t_1 \supset (b_2 \supset b)) \supset b)t_2$
9. $\vdash t_1 \wedge t_2 = \forall b. (t_1 \supset (t_2 \supset b)) \supset b$
10. $\Gamma \vdash t_1 \wedge t_2$ [Hypothesis]
11. $\Gamma \vdash \forall b. (t_1 \supset (t_2 \supset b)) \supset b$
12. $\Gamma \vdash (t_1 \supset (t_2 \supset t_1)) \supset t_1$ [SPEC 11]
13. $t_1 \vdash t_1$ [ASSUME]
14. $t_1 \vdash t_2 \supset t_1$ [DISCH 13]
15. $\vdash t_1 \supset (t_2 \supset t_1)$ [DISCH 14]
16. $\Gamma \vdash t_1$ [MP 12,15]

fun chk_Conjunct1(line, n, thm) =
let val thm1 = get_thm(line, n)
    val (AND,[conj1 conj2]) = strip_comb (concl thm1)
    and hyps = (hyps thm1)
in
  (every is_bool ty (Hterm.set.listItems hyps)) andalso
  (Hterm.set.equal(hyps, (hyps thm1))) andalso
  (well_typed AND) andalso (fst(dest_const AND) = "/\") andalso
  (conj1 = (concl thm))
end

• $\wedge$-elimination(right)

CONJUNCT2 NUMBER

\[
\frac{\Gamma \vdash t_1 \wedge t_2}{\Gamma \vdash t_2}
\]

CONJUNCT2 : thm → thm

This inference step extracts the right conjunct from the theorem referred to by NUMBER.
The conclusion of the input theorem must be a conjunction.
8.3. CHECKING FUNCTIONS

1. $\vdash \Lambda b_1 b_2. \forall b. (b_1 \supset (b_2 \supset b)) \supset b$
   [Definition of $\Lambda$
2. $\vdash \$s \wedge \$t_1 = \$s \wedge \$t_1$
   [REFL]
3. $\vdash (\lambda b_1 b_2. \forall b. (b_1 \supset (b_2 \supset b)) \supset \$t_1$
   $\lambda b_2. \forall b. (\$t_1 \supset (b_2 \supset b)) \supset b$]
4. $\vdash \$s \wedge \$t_1 = (\lambda b_1 b_2. \forall b. (b_1 \supset (b_2 \supset b)) \supset \$t_1$
   $\lambda b_2. \forall b. (\$t_1 \supset (b_2 \supset b)) \supset b$
5. $\vdash \$s \wedge \$t_1 = \$s \wedge \$t_2$
   [SUBST 1,2]
6. $\vdash \$t_1 \wedge \$t_2 = \$t_1 \wedge \$t_2$
   [SUBST 3,4]
7. $\vdash (\lambda b_2. \forall b. (\$t_1 \supset (b_2 \supset b)) \supset \$t_2$
   $\forall b. (\$t_1 \supset (\$t_2 \supset b)) \supset b$]
8. $\vdash \$t_1 \wedge \$t_2 = (\lambda b_2. \forall b. (\$t_1 \supset (b_2 \supset b)) \supset b)$
   $\forall b. (\$t_1 \supset (\$t_2 \supset b)) \supset \$t_2$
9. $\vdash \$t_1 \wedge \$t_2 = \forall b. (\$t_1 \supset (\$t_2 \supset b)) \supset b$
   [SUBST 5,6]
10. $\Gamma \vdash \$t_1 \wedge \$t_2$
    [SUBST 7,8]
11. $\Gamma \vdash \forall b. (\$t_1 \supset (\$t_2 \supset b)) \supset \$t_2$
    [HYPOTH]
12. $\Gamma \vdash (\$t_1 \supset (\$t_2 \supset \$t_2)) \supset \$t_2$
    [SUBST 9,10]
13. $\Gamma \vdash \$t_2$
    [SPEC 11]
14. $\Gamma \vdash \$t_2$
    [ASSUME]
15. $\Gamma \vdash \$t_2$
    [DISCH 13]
16. $\Gamma \vdash \$t_2$
    [DISCH 14]

fun chk_Conjunct2 (line, n, thm) =
  let val thml = get_thm(line, n)
     val (AND, [conj1, conj2]) = strip_comb (concl thml)
     and hyps = hyp thml
  in
     every is_boolTy (HtermSet.listItems hyps) andalso
     (HtermSet.equal(hyps, (hyp thm))) andalso
     (wellTyped AND) andalso (fst(dest_const AND) = "\"\") andalso
     (conj2 = concl thm)
  end

• Intuitionistic contradiction rule

CONTR term NUMBER

\[
\begin{array}{c}
\Gamma \vdash F \\
\Gamma \vdash t
\end{array}
\]

CONTR : term -> thm -> thm

The theorem referred to by NUMBER should have falsity (F) as its conclusion.

1. $\Gamma \vdash \forall t. F \supset t$
   [Theorem FALSITY]
2. $\Gamma \vdash F$
   [Hypothesis]
3. $\Gamma \vdash F \supset t$
   [SPEC 1]
4. $\Gamma \vdash t$
   [MP 3,2]

fun chk_Contr(line, term, n, thm) =
  let val thml = get_thm(line, n)
     and hyps = hyp thml
  in
     (wellTyped(concl thml)) andalso
     (fst(dest_const(concl thml)) = "F") andalso
     (every is_boolTy (HtermSet.listItems hyps)) andalso
(TermSet.equal(hyps, (hyp thm))) andalso
(is_bool ty term) andalso (term = concl thm)
end

• Right \lor-introduction

\[
\Gamma \vdash t_1 \\
\Gamma \vdash t_1 \lor t_2
\]

DISJ1 : thm \to term \to thm

The result of this inference rule is a disjunctive theorem whose second disjunct is the input term. This term must be of type :bool.

1. \vdash \forall \lambda b_1. \forall b. \lambda b. (b_1 \lor b) \supset (b_2 \supset b) \supset b
   [Definition of \forall]
2. \vdash \forall \forall t_1 = \forall \forall t_1
   [REFL]
3. (\lambda b_1. \forall b. (b_1 \lor b) \supset (b_2 \supset b) \supset b) t_1
   = \lambda b_2. \forall b. (t_1 \lor b) \supset (b_2 \supset b) \supset b
   [BETA_CONV]
4. \vdash \forall \forall t_1 = (\lambda b_1. \forall b. (b_1 \lor b) \supset (b_2 \supset b) \supset b) t_1
   [SUBST 1,2]
5. \vdash \forall \forall t_1 = \lambda b_2. \forall b. (t_1 \lor b) \supset (b_2 \supset b) \supset b
   [SUBST 3,4]
6. \vdash t_1 \lor t_2 = t_1 \lor t_2
   [REFL]
7. (\lambda b_2. \forall b. (t_1 \lor b) \supset (b_2 \supset b) \supset b) t_2
   = \forall b. (t_1 \lor b) \supset (t_2 \supset b) \supset b
   [BETA_CONV]
8. \vdash t_1 \lor t_2 = (\lambda b_2. \forall b. (t_1 \lor b) \supset (b_2 \supset b) \supset b) t_2
   [SUBST 5,6]
9. \vdash t_1 \lor t_2 = \forall b. (t_1 \lor b) \supset (t_2 \supset b) \supset b
   [SUBST 7,8]
10. \Gamma \vdash t_1
    [Hypothesis]
11. t_1 \supset b \vdash t_1 \supset b
12. \Gamma, t_1 \supset b \vdash b
    [MP 11,10]
13. \Gamma, t_1 \supset b \vdash (t_2 \supset b) \supset b
    [DISCH 12]
14. \Gamma \vdash (t_1 \supset b) \supset (t_2 \supset b) \supset b
    [DISCH 13]
15. \Gamma \vdash \forall b. (t_1 \lor b) \supset (t_2 \supset b) \supset b
    [GEN 14]
16. \forall b. (t_1 \lor b) \supset (t_2 \supset b) \supset b = t_1 \lor t_2
    [SUBST 9,6]
17. \Gamma \vdash t_1 \lor t_2
    [SUBST 16,15]

fun chk_Disj1(line, n, term, thm) =
  let val thm1 = get_thm(line, n)
    val (OR, [disj1, disj2]) = strip_comb(concl thm)
  in
    (is_bool ty disj1) andalso (is_bool ty disj2) andalso
    (well_typed OR) andalso ((fst(dest const OR)) = "\lor") andalso
    (concl thm1 = disj1) andalso (term = disj2) andalso
    (HtermSet.equal((hyp thm1), (hyp thm1)))
  end

• Left \lor-introduction

\[
\Gamma \vdash t_2 \\
\Gamma \vdash t_1 \lor t_2
\]

DISJ2 : term \to thm \to thm

The result of this inference rule is a disjunctive theorem whose first disjunct is the input term. This term must be of type :bool.
3.3. CHECKING FUNCTIONS

1. \( \vdash \forall v. (b_1 \supset b) \supset (b_2 \supset b) \supset b \)  
   \[ \text{Definition of } \forall \]
2. \( \vdash \forall v \vdash \forall v \)  
   \[ \text{REFL} \]
3. \( (\lambda b_1 b_2. \forall v. (b_1 \supset b) \supset (b_2 \supset b) \supset b) \eta_1 \)
   \( = \lambda b_2. \forall v. (\eta_1 \supset b) \supset (b_2 \supset b) \supset b \)
   \[ \text{BETA_CONV} \]
4. \( \vdash \forall v \vdash (\lambda b_1 b_2. \forall v. (b_1 \supset b) \supset (b_2 \supset b) \supset b) \eta_1 \)
   \[ \text{SUBST 1,2} \]
5. \( \vdash \forall v \vdash \lambda b_2. \forall v. (\eta_1 \supset b) \supset (b_2 \supset b) \supset b \)
   \[ \text{SUBST 3,4} \]
6. \( \vdash t_1 \forall t_2 = t_1 \forall t_2 \)
   \[ \text{REFL} \]
7. \( (\lambda b_2. \forall v. (\eta_1 \supset b) \supset (b_2 \supset b) \supset b) \eta_2 \)
   \( = \forall v. (\eta_1 \supset b) \supset (\eta_2 \supset b) \supset b \)
   \[ \text{BETA_CONV} \]
8. \( \vdash \forall v \vdash (\lambda b_2. \forall v. (\eta_1 \supset b) \supset (\eta_2 \supset b) \supset b) \eta_2 \)
   \[ \text{SUBST 5,6} \]
9. \( \vdash \forall v \vdash \forall v. (\eta_1 \supset b) \supset (\eta_2 \supset b) \supset b \)
   \[ \text{SUBST 7,8} \]
10. \( \Gamma \vdash \eta_2 \)
    \[ \text{Hypothesis} \]
11. \( t_2 \supset b \vdash \eta_2 \supset b \)
12. \( \Gamma, t_2 \supset b \vdash \eta_2 \)
    \[ \text{MP 11,10} \]
13. \( \Gamma, t_2 \supset b \vdash t_2 \supset b \)
    \[ \text{DISCH 12} \]
14. \( \Gamma \vdash (t_1 \supset b) \supset (t_2 \supset b) \supset b \)
    \[ \text{DISCH 13} \]
15. \( \Gamma \vdash \forall v. (t_1 \supset b) \supset (t_2 \supset b) \supset b \)
    \[ \text{GEN 14} \]
16. \( \vdash \forall v. (t_1 \supset b) \supset (t_2 \supset b) \supset b = t_1 \forall t_2 \)
    \[ \text{SUBST 9,6} \]
17. \( \Gamma \vdash t_1 \forall t_2 \)
    \[ \text{SUBST 16,15} \]

fun chk_Disj2(line, term, n, thm) = 
  let val thml = get_thm(line, n)
  val (OR, [disj1,disj2]) = strip_comb(concl thml)
in
  (is_bool_ty disj1) andalso (is_bool_ty disj2) andalso
  (well_typed OR) andalso ((fst(dest_const OR)) = "\lor") andalso
  ((concl thml) = disj2) andalso (term = disj1) andalso
  (HtermSet.equal((hyp thm), (hyp thml)))
end

• \( \forall \)-elimination

\[
\frac{\Gamma \vdash t_1 \lor t_2 \quad \Gamma_1, t_1 \vdash t \quad \Gamma_2, t_2 \vdash t}{\Gamma \cup \Gamma_1 \cup \Gamma_2 \vdash t}
\]

DISJCASES NUMBER NUMBER NUMBER NUMBER

The theorem referred to by the first NUMBER must be a disjunction. The assumptions of the second and the third theorems must include the first and second disjunct of the first theorem, respectively.
64

CHAPTER 8. THE CHECKING RULES

1. \( \vdash \forall x \, (b_1 \lor b) \supset (b_2 \supset b) \supset b \)  
   \[\text{Definition of } \lor\]

2. \( \vdash \forall x \, (b_1 \lor b) \supset (b_2 \lor b) \supset \top \)  
   \[\text{REFL}\]

3. \( (\lambda x \, (b_1 \lor b) \supset (b_2 \lor b) \supset ) t_1 = (\lambda x \, (b_1 \lor b) \supset (b_2 \lor b) \supset b) \)
   \[\text{BETA_CONV}\]

4. \( \vdash \forall x \, (\lambda x \, (b_1 \lor b) \supset (b_2 \lor b) \supset ) t_1 \)
   \[\text{SUBST 1,2}\]

5. \( \vdash \forall x \, (\lambda x \, (b_1 \lor b) \supset (b_2 \lor b) \supset b) \)
   \[\text{SUBST 3,4}\]

6. \( \vdash t_1 \lor t_2 = t_1 \lor t_2 \)
   \[\text{REFL}\]

7. \( (\lambda x \, (b_1 \lor b) \supset (b_2 \lor b) \supset ) t_2 \)
   \[\text{BETA_CONV}\]

8. \( \vdash t_1 \lor t_2 = (\lambda x \, (b_1 \lor b) \supset (b_2 \lor b) \supset ) t_2 \)
   \[\text{SUBST 5,6}\]

9. \( \vdash t_1 \lor t_2 = (\lambda x \, (b_1 \lor b) \supset (b_2 \lor b) \supset b) \)
   \[\text{SUBST 7,8}\]

10. \( \Gamma \vdash t_1 \lor t_2 \)
    \[\text{Hypothesis}\]

11. \( \Gamma \vdash (\forall x \, (b_1 \lor b) \supset (b_2 \lor b) \supset ) t_1 \)
    \[\text{SUBST 9,10}\]

12. \( \Gamma \vdash (t_1 \supset t) \supset (t_2 \supset t) \supset t \)
    \[\text{SPEC 11}\]

13. \( \Gamma_1 \vdash t \)
    \[\text{Hypothesis}\]

14. \( \Gamma_1 \vdash t \)
    \[\text{DISCH 13}\]

15. \( \Gamma \cup \Gamma_1 \vdash (t_2 \supset t) \supset t \)
    \[\text{MP 12,14}\]

16. \( \Gamma_2 \vdash t_2 \supset t \)
    \[\text{Hypothesis}\]

17. \( \Gamma_2 \vdash t_2 \supset t \)
    \[\text{DISCH 16}\]

18. \( \Gamma \cup \Gamma_1 \cup \Gamma_2 \vdash t \)
    \[\text{MP 15,17}\]

fun chk_DisjCases(line, n1, n2, n3, thm) =
let val thm1 = get_thm(line, n1)
   and thm2 = get_thm(line, n2) and thm3 = get_thm(line, n3)
val (val, OR, [disj1, disj2]) = strip_comb (concl thm1)
in
(well_typed OR) andalso (((fst(dest_const OR)) = "\lor") andalso
(HtermSet.member((hyp thm2), disj1)) andalso
(HtermSet.member((hyp thm3), disj2)) andalso
((concl thm2) = (concl thm3)) andalso
((concl thm) = (concl thm2)) andalso
(HtermSet.equal(hyp thm), (HtermSet.union((hyp thm1),
(HtermSet.union((HtermSet.delete((hyp thm2),disj1)),
(HtermSet.delete((hyp thm3),disj2)))))))))
end

• Implication from equality (left)

**EQIMPRULE. NUMBER**

\[
\Gamma \vdash t_1 = t_2 \\
\Gamma \vdash t_2 \supset t_1
\]

**EQ_IMP_RULE : thm \to (thm \# thm)**

The theorem referred to by **NUMBER** must be an equation, and both sides of it must be of type :bool. The resulting theorem of this justification is the second theorem returned by the function **EQ_IMP_RULE**. It is an implication whose antecedent is the right-hand side of the hypothesis and whose conclusion is the left-hand side of the hypothesis.

1. \( \Gamma \vdash t_1 = t_2 \)
   \[\text{Hypothesis}\]

2. \( t_2 \vdash t_2 \)
   \[\text{ASSUME}\]

3. \( \Gamma \vdash t_2 = t_1 \)
   \[\text{SYM 1}\]

4. \( \Gamma, t_2 \vdash t_1 \)
   \[\text{SUBST 3,4}\]

5. \( \Gamma \vdash t_2 \supset t_1 \)
   \[\text{DISCH 4}\]
fun chk_EqImpRuleL(line, n, thm) = 
let val thml = get_thm(line, n) 
  val (left,right) = dest_eq (concl thml) 
in 
  (is_bool_ty left) andalso (is_bool_ty right) andalso 
  ((right,left) = dest_imp(concl thm)) andalso 
  (HtermSet.equal((hyp thm), (hyp thml))) 
end

• Implication from equality (right)

\[
\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash t_1 \supset t_2}
\]

EQ_IMP_RULE : thm \rightarrow (thm \# thm)

The theorem referred to by NUMBER must be an equation, and both sides of it must be of

\text{type} : \text{bool}. The resulting theorem of this justification is the first theorem returned by

the function EQ_IMP_RULE. It is an implication whose antecedent is the left-hand side of

the hypothesis and whose conclusion is the right-hand side of the hypothesis.

1. \(\Gamma \vdash t_1 = t_2\) \hspace{1cm} \text{[Hypothesis]}
2. \(t_1 \vdash t_1\) \hspace{1cm} \text{[ASSUME]}
3. \(\Gamma, t_1 \vdash t_2\) \hspace{1cm} \text{[SUBST 1,2]}
4. \(\Gamma \vdash t_1 \supset t_2\) \hspace{1cm} \text{[DISCH 3]}

fun chk_EqImpRuleR(line, n, thm) = 
let val thmr = get_thm(line, n) 
  val (left,right) = dest_eq (concl thmr) 
in 
  (is_bool_ty left) andalso (is_bool_ty right) andalso 
  ((left,right) = dest_imp(concl thm)) andalso 
  (HtermSet.equal((hyp thm), (hyp thmr))) 
end

• Modus Ponens for equality

\[
\frac{\Gamma_1 \vdash t_1 = t_2 \quad \Gamma_2 \vdash t_1}{\Gamma_1 \cup \Gamma_2 \vdash t_2}
\]

EQ_MP : thm \rightarrow thm \rightarrow thm

The first theorem should be an equation. The second theorem should be identical to

the left-hand side of the first. The resulting theorem is the right-hand side of the first

theorem.

1. \(\Gamma_1 \vdash t_1 = t_2\) \hspace{1cm} \text{[Hypothesis]}
2. \(\Gamma_2 \vdash t_1\) \hspace{1cm} \text{[Hypothesis]}
3. \(\Gamma_1 \cup \Gamma_2 \vdash t_2\) \hspace{1cm} \text{[SUBST 1,2]}
fun chk_EqMp(line, n1, n2, thm) =  
  let val thm1 = get_thm(line, n1) and thm2 = get_thm(line, n2)  
  val (left,right) = dest_eq (concl thm1)  
  in  
    (left = (concl thm2)) andalso (right = (concl thm)) andalso  
    (H_termset.equal((hyp thm1), (H_termset.union((hyp thm1), (hyp thm2)))))  
  end  

• Equality-with-\(T\) introduction

\[ \frac{\Gamma \vdash t}{\Gamma \vdash t = T} \]

\text{EQT\_INTRO : thm -> thm} This inference introduces an equality. The left-hand side of the conclusion must be the same as the hypothesis. The right-hand side must be a single constant \(T\).

1. \(\vdash \forall b_1. (b_1 \supset b_2) \supset (b_2 \supset b_1) \supset (b_1 = b_2)\) \hspace{1cm} [IMP\_ANTISYM\_AX]  
2. \(\vdash \forall b_2. (t \supset b_2) \supset (b_2 \supset t) \supset (t = b_2)\) \hspace{1cm} [SPEC 1]  
3. \(\vdash (t \supset T) \supset (T \supset t) \supset (t = T)\) \hspace{1cm} [SPEC 2]  
4. \(\vdash T\) \hspace{1cm} [TRUTH]  
5. \(\vdash t \supset T\) \hspace{1cm} [DISCH 4]  
6. \(\vdash (T \supset t) \supset (t = T)\) \hspace{1cm} [MP 3,5]  
7. \(\Gamma \vdash t\) \hspace{1cm} [Hypothesis]  
8. \(\Gamma \vdash T \supset t\) \hspace{1cm} [DISCH 7]  
9. \(\Gamma \vdash t = T\) \hspace{1cm} [MP 6,8]

fun chk_EqIntro(line, n, thm) =  
  let val thm1 = get_thm(line, n)  
  val (lhs,rhs) = dest_eq (concl thm1)  
  and T = mk_const("T", bool_ty)  
  in  
    (rhs = T) andalso (lhs = (concl thm1)) andalso  
    (H_termset.equal((hyp thm1), (hyp thm)))  
  end  

• \(\eta\)-conversion

\[ \vdash (\lambda x'. t \ x') = t \]

\text{ETA\_CONV : term -> thm} The variable \(x'\) does not occur free in \(t\). The input term is the same as the left-hand side of the resulting theorem.

1. \(\vdash \forall f. (\lambda x. f \ x) = f\) \hspace{1cm} [ETA\_AX]  
2. \(\vdash (\lambda x. t \ x) = t\) \hspace{1cm} [SPEC 1]  
3. \(\vdash (\lambda x'. t \ x') = (\lambda x. t \ x)\) \hspace{1cm} [\(\alpha\)-conversion]  
4. \(\vdash (\lambda x'. t \ x') = t\) \hspace{1cm} [TRANS 3,2]
fun chk_EtaConv(term, thm) = 
  let val (left,right) = dest_eq (concl thm) 
  in 
    (left = term) andalso (null (HtermSet.listItems(hyp thm))) andalso 
    (right = (fst o dest_comb o snd o dest_abs)left) 
  end 

• \exists\text{-introduction} 

\exists\text{ exists term NUMBER} 

\begin{align*} 
\Gamma & \vdash t_2/x \\
\Gamma & \vdash \exists x.t[x] 
\end{align*} 

\exists\exists : (\text{term} \# \text{term}) \rightarrow \text{thm} \rightarrow \text{thm} 

The first term is an existentially quantified term which matches exactly the conclusion of the resulting theorem, i.e., \exists x.t[x]. The second term t₂ must be of the same type as the bound variable x. When t₂ is substituted into t for the bound variable x, it results in a theorem which is the same as the input theorem referred to by NUMBER.

1. \vdash (\lambda x.t_1[x])t_2 = t_1[t_2] \quad \text{[BETA_CONV]} 
2. \vdash t_1[t_2] = (\lambda x.t_1[x])t_2 \quad \text{[SYM 1]} 
3. \Gamma \vdash t_1[t_2] \quad \text{[Hypothesis]} 
4. \Gamma \vdash (\lambda x.t_1[x])t_2 \quad \text{[EQ_MP 2,3]} 
5. \Gamma \vdash (\lambda x.t_1[x])\epsilon(\lambda x.t_1[x]) \quad \text{[SELECT_INTO 4]} 
6. \vdash \exists = \forall P.P(e P) \quad \text{[Definition of \exists]} 
7. \vdash \exists(\lambda x.t_1[x]) = (\lambda P.P(\epsilon P))(\lambda x.t_1[x]) \quad \text{[AP_THM 6]} 
8. \vdash (\lambda P.P(\epsilon P))(\lambda x.t_1[x]) = (\lambda x.t_1[x])(\epsilon(\lambda x.t_1[x])) \quad \text{[BETA_CONV]} 
9. \vdash \exists(\lambda x.t_1[x]) = (\lambda x.t_1[x])(\epsilon(\lambda x.t_1[x])) \quad \text{[TRANS 7,8]} 
10. \vdash (\lambda x.t_1[x])(\epsilon(\lambda x.t_1[x])) = \exists(\lambda x.t_1[x]) \quad \text{[SYM 9]} 
11. \Gamma \vdash \exists(\lambda x.t_1[x]) \quad \text{[EQ_MP 10,5]}

fun chk_Exists(line, term1, term2, n, thm) = 
  let val thmi = get_thm(line, n) 
    val (x,body) = dest_exists (concl thm) 
  in 
    (term1 = (concl thm)) andalso 
    (term_subst_chk [(x,term2),x] body (concl thmi) body) andalso 
    (HtermSet.equal((hyp thm), (hyp thmi))) 
  end 

• Extensionality 

\exists \text{ exists NUMBER} 

\begin{align*} 
\Gamma & \vdash \forall x.t_1 x = t_2 x \\
\Gamma & \vdash t_1 = t_2 
\end{align*} 

\exists \exists : \text{thm} \rightarrow \text{thm} 

The variable x in the proof below is a new variable which does not occur free anywhere in the input theorem. Both the hypothesis and the conclusion must be equality. Both

\footnote{with appropriate type instantiation.}
sides of the hypothesis must be function applications, and their operands must be the same variable. Both sides of the conclusion must be the same as the function on the corresponding side of the hypothesis.

1. $\Gamma \vdash \forall x.t_1 = t_2 x$  \hspace{1cm} [Hypothesis]
2. $\Gamma \vdash t_1 x' = t_2 x'$  \hspace{1cm} [SPEC 1]
3. $\Gamma \vdash (\lambda x'.t_1 x') = (\lambda x'.t_2 x')$  \hspace{1cm} [ABS 2]
4. $\vdash (\lambda x'.t_1 x') = t_1$  \hspace{1cm} [ETA_CONV]
5. $\vdash t_1 = (\lambda x'.t_1 x')$  \hspace{1cm} [SYM 4]
6. $\Gamma \vdash t_1 = (\lambda x'.t_2 x')$  \hspace{1cm} [TRANS 5,3]
7. $\vdash (\lambda x'.t_2 x') = t_2$  \hspace{1cm} [ETA_CONV]
8. $\Gamma \vdash t_1 = t_2$  \hspace{1cm} [TRANS 6,7]

fun chk_Ext(line, n, thm) =
  let val (left,right) = dest_eq(concl thm)
    val thm1 = get_thm(line, n)
    val (x, con) = dest_forall (concl thm1)
    val (left',right') = dest_eq con
    in
      (mk_comb(left,x) = left') andalso
      (mk_comb(right,x) = right') andalso
      (HtermSet.equal(hyp thm), (hyp thm1))) andalso
      (not(mem x (frees1 (HtermSet.listItems(hyp thm))))) andalso
      (not (free_in x left)) andalso (not (free_in x right))
    end

• Generalisation($\forall$-introduction)

\[
\Gamma \vdash t \\
\Gamma \vdash \forall x.t
\]

GEN : term $\rightarrow$ thm $\rightarrow$ thm

The input term $x$ is a variable which does not occur free in the assumption $\Gamma$.

1. $\Gamma \vdash t$  \hspace{1cm} [Hypothesis]
2. $\Gamma \vdash t = T$  \hspace{1cm} [EQT_INTRO 1]
3. $\Gamma \vdash (\lambda x.t) = (\lambda x.T)$  \hspace{1cm} [ABS 2]
4. $\vdash \forall(\lambda x.t) = \forall(\lambda x.t)$  \hspace{1cm} [REFL]
5. $\vdash (\lambda P.P = (\lambda x.T))$  \hspace{1cm} [Definition of $\forall$]
6. $\vdash (\lambda x.T) = (\lambda P.P = (\lambda x.T))(\lambda x.t)$  \hspace{1cm} [SUBST 5,4]
7. $\vdash (\lambda P.P = (\lambda x.T))(\lambda x.t) = ((\lambda x.t) = (\lambda x.T))$  \hspace{1cm} [BETA_CONV]
8. $\vdash (\lambda x.T) = ((\lambda x.t) = (\lambda x.T))$  \hspace{1cm} [TRANS 6,7]
9. $\vdash ((\lambda x.t) = (\lambda x.T)) = \forall(\lambda x.T)$  \hspace{1cm} [SYM 8]
10. $\Gamma \vdash \forall(\lambda x.t)$  \hspace{1cm} [EQ_MP 9,3]

fun chk.Gen(line, term, n, thm) =
  let val thm1 = get_thm(line, n)
    val (x,body) = dest_forall(concl thm)

\[\text{with appropriate type instantiation.}\]
8.3. CHECKING FUNCTIONS

val fs = freesl (HtermSet.listItems(hyp thm))
in
  (body = concl thm1) andalso (not (mem x fs)) andalso
  (HtermSet.equal((hyp thm), (hyp thm1)))
end

• Deducing equality from implications

<table>
<thead>
<tr>
<th>IMIPSYSRULE NUMBER NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Γ₁ ⊢ t₁ ⊋ t₂ \ Γ₂ ⊢ t₂ ⊋ t₁</td>
</tr>
<tr>
<td>Γ₁ ∪ Γ₂ ⊢ t₁ = t₂</td>
</tr>
</tbody>
</table>

IMP_ANTISYM_RULE : thm -> thm -> thm

The two hypotheses should be implications. The antecedent of first must be the same as the conclusion of the second, and vice versa. The conclusion should be an equality whose left-hand side must be the same as the antecedent of the first hypothesis and whose right-hand side must be the same as the conclusion of the first hypothesis.

1. ⊢ ∀b₁ b₂. (b₁ ⊋ b₂) ⊋ (b₂ ⊋ b₁) ⊋ (b₁ = b₂)  [IMP_ANTISYM_AX]
2. Γ₁ ⊢ t₁ ⊋ t₂
3. Γ₂ ⊢ t₂ ⊋ t₁
4. ⊢ ∀b₂. (t₁ ⊋ b₂) ⊋ (b₂ ⊋ t₁) ⊋ (t₁ = b₂)  [SPEC]
5. ⊢ (t₁ ⊋ t₂) ⊋ (t₂ ⊋ t₁) ⊋ (t₁ = t₂)  [SPEC]
6. Γ₁ ⊢ (t₂ ⊋ t₁) ⊋ (t₁ = t₂)  [MP 5,2]
7. Γ₁ ∪ Γ₂ ⊢ t₁ = t₂  [MP 6,3]

fun chk_ImpAntisymRule(line, n1, n2, thm) =
  let val thm1 = get_thm(line, n1) and thm2 = get_thm(line, n2)
     val (left,right) = dest_eq (concl thm)
in
    ((left,right) = dest_imp(concl thm1)) andalso
    ((right,left) = dest_imp(concl thm2)) andalso
    (HtermSet.equal((hyp thm), (HtermSet.union((hyp thm1),(hyp thm2))))))
end

• Transitivity of implications

<table>
<thead>
<tr>
<th>IMTRANS NUMBER NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Γ₁ ⊢ t₁ ⊋ t₂ \ Γ₂ ⊢ t₂ ⊋ t₃</td>
</tr>
<tr>
<td>Γ₁ ∪ Γ₂ ⊢ t₁ ⊋ t₃</td>
</tr>
</tbody>
</table>

IMP_TRANS : thm -> thm -> thm

Both theorems referred to by the numbers must be implications. The conclusion of the first theorem must be the same as the antecedent of the second theorem.

1. Γ₁ ⊢ t₁ ⊋ t₂  [Hypothesis]
2. Γ₂ ⊢ t₂ ⊋ t₃  [Hypothesis]
3. t₁ ⊢ t₁  [ASSUME]
4. Γ₁ ∪ {t₁} ⊢ t₂  [MP 1,3]
5. Γ₁ ∪ Γ₂ ∪ {t₁} ⊢ t₃  [MP 2,4]
6. Γ₁ ∪ Γ₂ ⊢ t₁ ⊋ t₃  [DISECH 5]
fun chk_ImpTrans(line, n1, n2, thm) =
  let val thm1 = get_thm(line, n1) and thm2 = get_thm(line, n2)
  val (left', right'') = dest_imp(concl thm1)
  and (left'', right'') = dest_imp(concl thm2)
  val (left, right) = dest_imp(concl thm)
in
  (left = left') andalso (right = right'') andalso
  (HtermSet.equal((hyp thm), (HtermSet.union((hyp thm1), (hyp thm2))))
end

• Instantiation of free variables

\[
\Gamma \vdash t \\
\Gamma \vdash t[t_1, \ldots, t_n/x_1, \ldots, x_n]
\]

INST : (term # term) list -> thm -> thm

The term_term_list is a list of pairs. Their second fields are variables which do not occur free in the assumption \( \Gamma \). Their first fields are terms having the same type as the corresponding variables. They are substituted for the variables in the theorem referred to by the NUMBER.

1. \( \Gamma \vdash t \) \hspace{1cm} \text{[Hypothesis]}
2. \( \Gamma \vdash \forall x_1 \ldots x_n. t[x_1, \ldots, x_n] \) \hspace{1cm} \text{[GENL}\textsuperscript{3}]
3. \( \Gamma \vdash t[t_1, \ldots, t_n/x_1, \ldots, x_n] \) \hspace{1cm} \text{[SPECL}\textsuperscript{4}]

fun chk_Inst(line, ttl, n, thm) =
  let val thm1 = get_thm(line, n)
    val vars = map snd ttl
    val fs = freesl (HtermSet.listItems(hyp thm1))
    val coni = concl thm1
    val (tl1, tl2) = ListUtil.unzip ttl
    val tl1' = ListUtil.zip(tl2, tl1)
in
  (every (fn x => not (mem x fs)) vars) andalso
  (term_subst_chk
    (ListUtil.zip(tl1', vars)) coni (concl thm) coni) andalso
  (HtermSet.equal((hyp thm), (hyp thm1)))
end

• Abstraction introduction on equality

\[
\Gamma \vdash \forall x. t_1 = t_2 \\
\Gamma \vdash (\lambda x. t_1) = (\lambda x. t_2)
\]

MK_ABS : thm -> thm

The theorem referred to by the NUMBER must be an equality inside a single universal quantification.

\textsuperscript{3}GENL is an iterative version of GEN that applies GEN repeatedly to a list of variables.
\textsuperscript{4}SPECL is an iterative version of SPEC that applies SPEC repeatedly to a list of terms.
8.3. CHECKING FUNCTIONS

1. $\Gamma \vdash \forall x. t_1 = t_2$  \hspace{1cm} [Hypothesis]
2. $\vdash (\lambda x. t_1)x' = t_1[x'/x]$ \hspace{1cm} [BETA_CONV]
3. $\Gamma \vdash t_1[x'/x] = t_2[x'/x]$ \hspace{1cm} [SPEC 1]
4. $\vdash (\lambda x. t_2)x' = t_2[x'/x]$ \hspace{1cm} [BETA_CONV]
5. $\vdash t_2[x'/x] = (\lambda x. t_2)x'$ \hspace{1cm} [SYM 4]
6. $\Gamma \vdash (\lambda x. t_1)x' = t_2[x'/x]$ \hspace{1cm} [TRANS 2,3]
7. $\Gamma \vdash (\lambda x. t_1)x' = (\lambda x. t_2)x'$ \hspace{1cm} [TRANS 5,6]
8. $\Gamma \vdash \forall x'. (\lambda x. t_1)x' = (\lambda x. t_2)x'$ \hspace{1cm} [GEN 7]
9. $\Gamma \vdash (\lambda x. t_1) = (\lambda x. t_2)$ \hspace{1cm} [EXT 8]

fun chk_MkAbs(line, n, thm) =
let val thm1 = get_thm(line, n)
  val (x,body) = dest_forall(concl thm1)
  val (left,right) = dest_eq (concl thm)
in
  (left = mk_abs(x,lhs body)) andalso
  (right = mk_abs(x, rhs body)) andalso
  (HtermSet.equal((hyp thm), (hyp thm1)))
end

• Equality of combinations

\begin{center}
\begin{align*}
\Gamma_1 \vdash f = g & \quad \Gamma_2 \vdash x = y \\
\Gamma_1 \mathbin{\sqcup} \Gamma_2 \vdash f \, x = g \, y
\end{align*}
\end{center}

MK_COMB : (thm # thm) \to thm

Both theorems referred to by the NUMBERS must be equations. Both sides of the first theorem are functions whose domains are the type of \(x\) in the second theorem.

1. $\Gamma_1 \vdash f = g$  \hspace{1cm} [Hypothesis]
2. $\Gamma_2 \vdash x = y$ \hspace{1cm} [Hypothesis]
3. $\vdash f \, x = f \, x$ \hspace{1cm} [REFL]
4. $\Gamma_1 \vdash f \, x = g \, x$ \hspace{1cm} [SUBST 1,3]
5. $\Gamma_1 \vdash f \, x = g \, y$ \hspace{1cm} [SUBST 2,4]

fun chk_MkComb(line, n1, n2, thm) =
let val thm1 = get_thm(line, n1) and thm2 = get_thm(line, n2)
  val (f,g) = dest_eq(concl thm1) and (x,y) = dest_eq(concl thm2)
  val (left,right) = dest_eq (concl thm)
in
  (left = mk_comb(f,x)) andalso (right = mk_comb(g,y)) andalso
  (HtermSet.equal((hyp thm), (HtermSet.union((hyp thm1),(hyp thm2))))))
end

• Converting \(\forall\) to \(\exists\)

\begin{center}
\begin{align*}
\Gamma \vdash \forall x. t_1 = t_2 \\
\Gamma \vdash (\exists x. t_1) = (\exists x. t_2)
\end{align*}
\end{center}

MK.Exists : thm \to thm
The theorem referred to by the NUMBER must be an equality inside a single universal quantification. The terms \( t_1 \) and \( t_2 \) must be of type \( \text{bool} \).

1. \( \Gamma \vdash \forall x. t_1[x] = t_2[x] \)  
   \[ \text{[Hypothesis]} \]
2. \( \Gamma \vdash t_1[x'/x] = t_2[x'/x] \)  
   \[ \text{[SPEC 1]} \]
3a. \( \Gamma \vdash t_1[x'/x] \supset t_2[x'/x] \)  
   \[ \text{[EQ_IMP_RULE(right) 2]} \]
3b. \( \Gamma \vdash t_2[x'/x] \supset t_1[x'/x] \)  
   \[ \text{[EQ_IMP_RULE(left) 2]} \]
4. \( t_1[x'/x] \vdash t_1[x'/x] \)  
   \[ \text{[ASSUME]} \]
5. \( \Gamma \cup \{t_1[x'/x]\} \vdash t_2[x'/x] \)  
   \[ \text{[MP 3a,4]} \]
6. \( \Gamma \cup \{t_1[x'/x]\} \vdash \exists x. t_2[x] \)  
   \[ \text{[EXISTS 5]} \]
7. \( \exists x. t_1[x] \vdash \exists x. t_1[x] \)  
   \[ \text{[ASSUME]} \]
8. \( \Gamma \cup \{\exists x. t_1[x]\} \vdash \exists x. t_2[x] \)  
   \[ \text{[CHOOSE 7,8]} \]
9. \( \Gamma \vdash \exists x. t_1[x] \supset \exists x. t_2[x] \)  
   \[ \text{[DISCH 8]} \]
10. \( t_2[x'/x] \vdash t_2[x'/x] \)  
    \[ \text{[MP 3b,9]} \]
11. \( \Gamma \cup \{t_2[x'/x]\} \vdash \exists x. t_1[x] \)  
    \[ \text{[EXISTS 10]} \]
12. \( \exists x. t_2[x] \vdash \exists x. t_2[x] \)  
    \[ \text{[ASSUME]} \]
13. \( \Gamma \cup \{\exists x. t_2[x]\} \vdash \exists x. t_1[x] \)  
    \[ \text{[CHOOSE 12,11]} \]
14. \( \Gamma \vdash \exists x. t_2[x] \supset \exists x. t_1[x] \)  
    \[ \text{[DISCH 13]} \]
15. \( \Gamma \vdash \exists x. t_1[x] = \exists x. t_2[x] \)  
    \[ \text{[IMP_ANTISYM_RULE 9,14]} \]

fun chk_MkExists(line, n, thm) =
  let val thm1 = get_thm(line, n)
      val (x, body) = dest_forall (concl thm1)
      val (left',right') = dest_eq body
      val (left,right) = dest_eq (concl thm)
    in
      ((x, left') = dest_exists left) andalso
      ((x, right') = dest_exists right) andalso
      (HtermSet.equal((hyp thm1), (hyp thm1)))
    end

* →-elimination

| NOTELEM NUMBER |
\[
\Gamma \vdash \neg t \quad \Gamma \vdash t \supset F
\]

NOT_ELIM : thm → thm

The theorem referred to by NUMBER must be a negation.

1. \( \vdash \neg = \lambda b \ b \supset F \)  
   \[ \text{[Definition of \( \neg \)]} \]
2. \( \Gamma \vdash \neg t \)  
   \[ \text{[Hypothesis]} \]
3. \( \Gamma \vdash (\lambda b \ b \supset F)t \)  
   \[ \text{[SUBST 1,2]} \]
4. \( \vdash (\lambda b \ b \supset F)t = t \supset F \)  
   \[ \text{[BETA_CONV]} \]
5. \( \Gamma \vdash t \supset F \)  
   \[ \text{[SUBST 4,3]} \]

fun chk_NotElim(line, n, thm) =
  let val thm1 = get_thm(line, n)
      val t = dest_not (concl thm1)
      val (t',f) = dest_imp (concl thm)
    in
8.3. CHECKING FUNCTIONS

in
(t = t') andalso (well_typed f) andalso
(fst(dest Const f) = "F") andalso
(HtermSet.equal((hyp thm), (hyp thm1)))
end

• introduction
NOTINTRO NUMBER

\[ \Gamma \vdash t \supset F \quad \Gamma \vdash \neg t \]

NOT_INTRO : thm \rightarrow thm

The theorem referred to by NUMBER must be an implication whose conclusion is the constant F.

1. \[ \Gamma \vdash \neg = \lambda b. b \supset F \] [Definition of \( \neg \)]
2. \[ \Gamma \vdash t \supset F \] [Hypothesis]
3. \[ \Gamma \vdash \neg t = \neg t \] [REFL]
4. \[ \Gamma \vdash \neg t = (\lambda b. b \supset F)t \] [SUBST 1,3]
5. \[ \Gamma \vdash (\lambda b. b \supset F)t = t \supset F \] [BETA_CONV]
6. \[ \Gamma \vdash \neg t = t \supset F \] [SUBST 5,4]
7. \[ \Gamma \vdash t \supset F = \neg t \] [SUBST 6,3]
8. \[ \Gamma \vdash \neg t \] [SUBST 7,2]

fun chk_NotIntro(line, n, thm) =
let val thm1 = get_thm(line, n)
  val t = dest_not (concl thm)
  val (t',f) = dest_imp (concl thm1)
in
(t = t') andalso (well_typed f) andalso
(fst(dest Const f) = "F") andalso
(HtermSet.equal((hyp thm), (hyp thm1)))
end

• Specialisation (\forall\text{-elimination})

SPEC term NUMBER

\[ \Gamma \vdash \forall z.t \quad \Gamma \vdash t[t'/x] \]

SPEC : term \rightarrow thm \rightarrow thm

The theorem referred to by the NUMBER must be universally quantified. The input term must have the same type as the bound variable x of the theorem.
1. \( \vdash \forall = (\lambda P. P = (\lambda x. T)) \)  
   \[ \text{Definition of } \forall^6 \]  
2. \( \Gamma \vdash \forall(x.t) \)  
   \[ \text{Hypothesis} \]  
3. \( \Gamma \vdash (\lambda P. P = (\lambda x. T))(\lambda x. t) \)  
   \[ \text{[SUBST 1,2]} \]  
4. \( \vdash (\lambda P. P = (\lambda x. T))(\lambda x. t) = ((\lambda x. t) = (\lambda x. T)) \)  
   \[ \text{[BETA_CONV]} \]  
5. \( \Gamma \vdash (\lambda x. t) = (\lambda x. T) \)  
   \[ \text{[SUBST 4,3]} \]  
6. \( \vdash (\lambda x. t) t' = (\lambda x. t) t' \)  
   \[ \text{[REFL]} \]  
7. \( \Gamma \vdash (\lambda x. t) t' = (\lambda x. T) t' \)  
   \[ \text{[SUBST 5,6]} \]  
8. \( \vdash (\lambda x. t) t' = t'[t'/x] \)  
   \[ \text{[BETA_CONV]} \]  
9. \( \vdash (\lambda x. t) t' = (\lambda x. t) t' \)  
   \[ \text{[REFL]} \]  
10. \( \vdash t'[t'/x] = (\lambda x. t) t' \)  
   \[ \text{[SUBST 8,9]} \]  
11. \( \Gamma \vdash t'[t'/x] = (\lambda x. T) t' \)  
   \[ \text{[SUBST 10,7]} \]  
12. \( \vdash (\lambda x. T) t' = T \)  
   \[ \text{[BETA_CONV]} \]  
13. \( \Gamma \vdash t'[t'/x] = T \)  
   \[ \text{[SUBST 12,11]} \]  
14. \( \vdash t'[t'/x] = t'[t'/x] \)  
   \[ \text{[REFL]} \]  
15. \( \Gamma \vdash T = t'[t'/x] \)  
   \[ \text{[SUBST 13,14]} \]  
16. \( \vdash T \)  
   \[ \text{[Theorem TRUTH]} \]  
17. \( \Gamma \vdash t'[t'/x] \)  
   \[ \text{[SUBST 15,16]} \]

fun chk_Spec(line, term, n, thm) = 
  let val thml = get_thm(line, n) 
  val (x, body) = dest_forall (concl thml) 
  in 
    if (term_subst_chk [(x, term), x]) body (concl thm) body) 
    then true 
    else raise (Check_ERR(function="chk_Spec", 
                        message="Assumption set not equal")) 
    else raise (Check_ERR(function="chk_Spec", 
                        message="Substitution check failed")) 
  end

• Substitution (for all instances)

\[ \begin{align*} 
\Gamma_1 \vdash x_1 = t_1 & \ldots \Gamma_n \vdash x_n = t_n \quad \Gamma \vdash t \\
\Gamma_1 \cup \ldots \cup \Gamma_n \cup \Gamma \uplus t[x_1, \ldots, t_n/x_1, \ldots, x_n] \end{align*} \]

SUBS : thm list -> thm -> thm

This is a generalised version of the primitive rule SUBST. It replaces all occurrences of the variables \( x_i \) in \( t \) by the corresponding term \( t_i \).

Since the term substitution checking function term_subst_chk requires a template to indicate where the substitutions occur, we build one using the function mk_subs_templ.

fun chk_Subs(line, n1, n, thm) = 
  let val thms = map (fn n' => get_thm(line, n')) n1 
  and thml = get_thm(line, n) 
  val concl = concl thml 
  val vtl = map (dest_eq o concl) thms 
  val sub_list = 
    ListUtil.zip (vtl, (map (mk_gen_var o type_of o (#1)) vtl))

\textit{with appropriate type instantiation.}
8.3. CHECKING FUNCTIONS

val temp1 = mk_subs_temp1 sub_list con1
fun out_ppstream outstrm n =
  System.PrettyPrint.mk_ppstream(linewidth = n,
    flush = fn () => flush_out outstrm,
    consumer = outputc outstrm)
val ppstrm = out_ppstream std_out 78
in
  if (((debug) > 1) then
    (Hterm.pp_qterm ppstrm temp1;
     System.PrettyPrint.flush_ppstream ppstrm)
  else ()
  in
    (term_subst_chk sub_list temp1 (concl thm) con1) andalso
    (HtermSet.equal((hyp thm1),
      (HtermSet.union ((hyp thm1),
        (fold HtermSet.union
          (map hyp thms) HtermSet.empty))))))
  end

• Substitution (for some instances)

<table>
<thead>
<tr>
<th>SUBSOCCS NUMBER/list.NNUMBER/list NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma_1 \vdash x_1 = t_1 \ldots \Gamma_n \vdash x_n = t_n ) ( \Gamma \vdash t )</td>
</tr>
<tr>
<td>( \Gamma_1 \cup \ldots \cup \Gamma_n \cup \Gamma \vdash t[t_1, \ldots, t_n/x_1, \ldots x_n] )</td>
</tr>
</tbody>
</table>

SUBS_OCCS : ((num)list # thm)list -> thm -> thm

This is another version of the primitive rule SUBST. It is more selective than SUBS. The first argument is a list of pairs whose general form is

\( ([k_1; \ldots; k_m], \Gamma \vdash x_i = t_i) \)

This inference replaces only the occurrences of \( x_i \) in \( t \) specified by the numbers in the list \( [k_1; \ldots; k_m] \). The occurrences are numbered from left to right starting from \( 1 \). In addition to the similar checks to SUBS, checking of whether only the specific occurrences are replaced needs to be carried out. This is done using a template which is created by the function mk_subs_occu_template. This function replaces the occurrences of \( x_i \) in \( t \) which are to be substituted by a new variable, and it replaces the \( x_i \) in the substitution list with the same variable. Then it uses the standard term substitution checking function term_subst_chk to check the results.

The local function mk_vlist creates a list of new variables for the \( x_i \)s. The first argument is a triple \((nln, thms, fs)\) where \( nln \) is the NUMBER/list.NNUMBER/list argument to this inference rule acting as a counter, \( thms \) is a list of equational terms obtained from the substitution theorems \( \Gamma_1 \vdash x_1 = t_i \), and \( fs \) is the list of free variables occurring in all theorems. Its second argument is a list of triples whose first fields are the two side of the substitution theorem, whose second fields are the new variables and whose third fields are the occurrence lists. When reaching the end of the list \( nln \), it returns its second argument.

Because the function mk_subs_occu_temp1 which creates the template requires the occurrence list in ascending order and the function get_thm also requires the line number in ascending order, the list \( nln \) is sorted first. The substitution theorems are fetched and bound to \( thms \). allhyps is bound to the assumptions of all input theorems. \( fvs \) are the
free variables of all the assumptions and the conclusion of \( \text{thm1} \). The template is bound to \( \text{templ} \).

```
fun chk_SubsOcss(line, nlnl, n, thm) = 
  let
    fun mk_vlist ([], [], fs) l = l
      | mk_vlist (((n1,n)::nlnl), thms::thms, fs) l = 
        let val (x,tm) = dest_eq thm
        val v = variant fs x
        in
          mk_vlist (nlnl, thms, (v :: fs)) (((tm,x),v,nlnl)::l)
        end
  in
    mk_vlist ([], [], thms)
    end

fun mk_subsoccs_template (((tm,x),v,nlnl),tm') = 
  #i(mk_subsoccs_occu_temp1 tm' x v 0 nlnl)
  val nlnl' = 
    map (fn (n1,n) => ((ListMergeSort.sort (op>) n1),n))
    (ListMergeSort.sort(fn ((n1,n1),(n2,n2)) => n1 > n2)nlnl)
  val thms = map (fn ((l:unit list),n') => get_thm(line, n')) nlnl'
  and thm1 = get_thm(line, n)
  val concl thm1
  val allhyp = HTermSet.union ((hyp thm1),
    (fold HTermSet.union (map hyp thms) HTermSet.empty))
  val fvs = (frees concl) @ (freesl (HTermSet.listItems allhyp))
  val ol = mk_vlist(nlnl', (map concl thms), fvs) []
  val subst = map (fn ((tm,x),v,nlnl) => ((tm,x),v)) ol
  val templ = fold mk_subsoccs_template ol concl
  in
    let fun out_ppstream outstr n = 
      System.PrettyPrint.mk_ppstream
        (linewidth = n, 
          flush = fn () => flush_out outstr, 
          consumer = outputc outstr)
      val ppstrm = out_ppstream std_out 78
    in
      if (!debug > 1) then
        (Hterm.pp_qterm ppstrm templ;
         System.PrettyPrint.flush_ppstream ppstrm)
      else ();
      (term_subst chk subst templ (concl thm) concl) andalso 
      (HTermSet.equal((hyp thm), allhyp))
    end
  end

\* Substitution (conversion)

| SUBST_CONV | NUMBER | term | term |

\[
\Gamma_1 \vdash t_1 = t_1' \quad \cdots \quad \Gamma_n \vdash t_n = t_n' \\
\Gamma_1 \cup \cdots \cup \Gamma_n \vdash t[t_1, \ldots, t_n / x_1, \ldots, x_n] = t[t_1', \ldots, t_n' / x_1, \ldots, x_n]
\]

\text{SUBST\_CONV} : \ (\text{thm} \# \text{term})\text{list} \to \text{term} \to \text{term} \to \text{thm}

This is a conversion performing substitution in a term similar to the primitive rule \text{SUBST}. The elements of the first argument to this conversion have the following form:

\[
(\Gamma_i \vdash t_i = t_i', "x_i")
\]
8.3. CHECKING FUNCTIONS

The first term argument $term_1 = t[x_1, \ldots, x_n]$, contains the variables $x_i$ marking the places where substitution is required. The second term argument $term_2$ should match $term_1$ with occurrences of $x_i$ replaced by the corresponding $t_i$ which is the left-hand side of the corresponding theorem in the list. The term $term_2$ is the left-hand side of the resulting theorem. The right-hand side is obtained by replacing the occurrences of $x_i$ in $t$ by the corresponding $t_i$. The term $term_1$ is used as the template.

fun chk_SubstConv(line, ntl, term1, term2, thm) =
  let
    val thms = map (fn (n,tm) => get_thm(line, n)) ntl
    val allhyps = fold HtermSet.union (map hyp thms) HtermSet.empty
    val sublist = ListUtil.zip((map (dest_eq o concl) thms),
                              (map (#2) ntl))
    val conc = concl thm
    val (left,right) = dest_eq conc
    in
      (term_subst_chk sublist term1 right left) andalso
      (left = term2) andalso (HtermSet.equal((hyp thm), allhyps))
  end

• Symmetry of equality

\[
\begin{align*}
\Gamma \vdash t_1 &= t_2 \\
\Gamma \vdash t_2 &= t_1
\end{align*}
\]

SYM : thm -> thm

The theorem referred to by NUMBER must be an equation.

1. $\Gamma \vdash t_1 = t_2$ [Hypothesis]
2. $\vdash t_1 = t_1$ [REFL]
3. $\Gamma \vdash t_2 = t_1$ [SUBST 1,2]

fun chk_Sym(line, n, thm) =
  let val thml = get_thm(line, n)
      val (left,right) = dest_eq (concl thml)
    in
      ((right,left) = dest_eq (concl thm)) andalso
      (HtermSet.equal((hyp thm), (hyp thml)))
    end

• Transitivity of equality

\[
\begin{align*}
\Gamma_1 \vdash t_1 &= t_2 & \Gamma_2 \vdash t_2 &= t_3 \\
\Gamma_1 \cup \Gamma_2 \vdash t_1 &= t_3
\end{align*}
\]

TRANS : thm -> thm -> thm

Both theorems referred to by the NUMBERS must be equations. The right-hand side of the first theorem must be the same as the left-hand side of the second.

1. $\Gamma_1 \vdash t_1 = t_2$ [Hypothesis]
2. $\Gamma_2 \vdash t_2 = t_3$ [Hypothesis]
3. $\Gamma_1 \cup \Gamma_2 \vdash t_1 = t_3$ [SUBST 2,1]
8.3.4 Checking proof lines The function \texttt{chk\_pline} is the checking function dispatcher. It is called by the parser after recognising a line.

\begin{verbatim}
local
open System.PrettyPrint Report
fun out_ppstream outstrm n =
  mk_ppstream(linewidth = n,
    flush = fn () \Rightarrow flush_out outstrm,
    consumer = outputc outstrm);
val check_message = ref "";
in
fun chk\_pline (thisline, just, thm) =
  let
    val n' = get_last_lineNo ()
in
    if not((thisline >= n') orelse (n' = 0)) then
      raise (Check\_ERR \{function = "chk\_pline",
          message = "incorrect line number"\})
    else (* line number OK *)
      let
        val line\_OK =
          (case just of
            (Hypothesis) \Rightarrow chk\_Hypothesis thm
            | (Abs (term, n)) \Rightarrow chk\_Abs (thisline,term,n,thm)
            | (Assume term) \Rightarrow chk\_Assume (term, thm)
            | (BetaConv term) \Rightarrow chk\_BetaConv (term,thm)
            | (Disch (term,n)) \Rightarrow chk\_Disch (thisline,term,n,thm)
            | (InstType (ttl,n)) \Rightarrow chk\_InstType (thisline,ttl,n,thm)
            | (Mp (n1,n2)) \Rightarrow chk\_Mp (thisline,n1,n2,thm)
            | (Refl term) \Rightarrow chk\_Refl (term, thm)
            | (Subst (itmlist,term,n)) \Rightarrow
              (case just of
                (Axiom (s1,s2)) \Rightarrow chk\_Axiom(s1,s2,thm)
                | (Definition (s1,s2)) \Rightarrow chk\_Definition(s1,s2,thm)
                | (DefExistsRule term) \Rightarrow chk\_DefExistsRule(term,thm)
                | (NewAxiom (s,term)) \Rightarrow chk\_NewAxiom(s,term,thm)
                | (NewConstant (s,ty)) \Rightarrow chk\_NewConstant(s,ty)
                | (NewType (n,s)) \Rightarrow chk\_NewType(n,s)
                | (Nunconv term) \Rightarrow chk\_Nunconv(term,thm)
                | (StoreDefinition (s,term)) \Rightarrow chk\_StoreDefinition(s,term,thm)
                | (Theorem (s1, s2)) \Rightarrow chk\_Theorem(s1,s2,thm)
          )
          end)

\end{verbatim}
8.3. CHECKING FUNCTIONS

```
| (AddAssum (term,n)) => chk_AddAssum(thisline,term,n,thm) |
| (Alpha (term1,term2)) => chk_AltAlpha(term1,term2,thm) |
| (AppTerm(term,n)) => chk_AppTerm(thisline,term,n,thm) |
| (AppThm (n,term)) => chk_AppThm(thisline,n,term,thm) |
| (CCons (term,n)) => chk_CCons(thisline,term,n,thm) |
| (Choose (term,n1,n2)) => chk_Choose(thisline,term,n1,n2,thm) |
| (Conj (n1,n2)) => chk_Conj(thisline,n1,n2,thm) |
| (Conjunct1 n) => chk_Conjunct1(thisline,n,thm) |
| (Conjunct2 n) => chk_Conjunct2(thisline,n,thm) |
| (Contr (term,n)) => chk_Contr(thisline,term,n,thm) |
| (Disj1 (n,term)) => chk_Disj1(thisline,n,term,thm) |
| (Disj2 (term,n)) => chk_Disj2(thisline,term,n,thm) |
| (DisjCases(n1,n2,n3)) => chk_DisjCases(thisline,n1,n2,n3,thm) |
| (EqImpRuleL n) => chk_EqImpRuleL(thisline,n,thm) |
| (EqImpRuleR n) => chk_EqImpRuleR(thisline,n,thm) |
| (EqM (n1,n2)) => chk_EqM(thisline,n1,n2,thm) |
| (EqTIntro n) => chk_EqTIntro(thisline,n,thm) |
| (EqTConv term) => chk_EqTConv(term,thm) |
| (Exists ((tm1,tm2),n)) => chk_Exists(thisline,tm1,tm2,n,thm) |
| (Ext n) => chk_Ext(thisline,n,thm) |
| (Gen (term,n)) => chk_Gen(thisline,term,n,thm) |
| (ImpAntisymRule (n1,n2)) => |
|   chk_ImpAntisymRule(thisline,n1,n2,thm) |
| (ImpTrans (n1,n2)) => chk_ImpTrans(thisline,n1,n2,thm) |
| (Inst (ttl,n)) => chk_Inst(thisline,ttl,n,thm) |
| (MkAbs n) => chk_MkAbs(thisline,n,thm) |
| (MkConv (n1,n2)) => chk_MkConv(thisline,n1,n2,thm) |
| (MkExists n) => chk_MkExists(thisline,n,thm) |
| (NotElim n) => chk_NotElim(thisline,n,thm) |
| (NotIntro n) => chk_NotIntro(thisline,n,thm) |
| (Spec (term,n)) => chk_Spec(thisline,term,n,thm) |
| (Subs (n1,n)) => chk_Subs(thisline,n1,n,thm) |
| (Subs0Ocs (nln1,n)) => chk_Subs0Ocs(thisline,n1,n,thm) |
| (SubstConv(ntl,ttm1,ttm2)) => |
|   chk_SubstConv(thisline,ntl,ttm1,ttm2,thm) |
| (Sym n) => chk_Sym(thisline,n,thm) |
| (Trans (n1,n2)) => chk_Trans(thisline,n1,n2,thm) |

    ) handle (CHK_ERR(message=m,origin_function=f,...) => |
    (check_message := (f "":"":m);
       write_out(!check_message);
       false)
     | e => |
    (check_message := "Unexpected error"; false)

in |

(if ((debug) > 0) |
then (pp_pline (out_ppstream std_err 78) |
        (mk_pline (thisline, just, thm))) |
else ()
if line_OK then add_pline(thisline,just,thm) |
else(raise (Check_ERR(function = "chk_pline", |
    message =
The function `chk_proof` is called when the end of a proof is found in Pass 2. The first argument `name` is the name of the proof, which is a string. The second is a list of goals. It writes the result of checking the proof to the log file, i.e., whether all goals have been proved, and so on.

```plaintext
fun chk_proof (name, thmlst, alst) = 
  fun splitpp [] (lp,lu) = (lp,lu)
     | splitpp ((Goal{Thm,Proved}):gs) (lp,lu) = 
       if Proved then splitpp gs ((Thm::lp),lu) 
       else splitpp gs (lp, (Thm::lu))
     in
   (output(std_out, "Proof "-"name" has been checked\n");
    if (null thmlst) then
      (write_line_opening Keyword.PROVED;
       write_list Hthm.pr_hthm [Proof.get_final_thm()];
       write_closing_line())
    else
      let
        val (thms1,thms2) = splitpp (goal_table()) ([],[])
        val out_ppstr = out_ppstream std_out 78
      in
        (if (null thms1) then ()
         else if ((length thms1) > 1)
            then output(std_out,"\nFollowing theorems are proved:\n")
            else output(std_out,"\nFollowing theorem is proved:\n")
            map (fn thm => pp_hthm out_ppstr thm) thms1;
            System.PrettyPrint.flush_ppstream out_ppstr;
            write_line_opening Keyword.PROVED;
            write_list Hthm.pr_hthm thms1;
            write_closing_line()
          );
        if (null thms2) then ()
        else if ((length thms2) > 1)
          then output(std_out,"\nFollowing goals are NOT proved:\n")
          else output(std_out,"\nFollowing goal is not proved:\n")
          map (fn thm => pp_hthm out_ppstr thm) thms2;
          System.PrettyPrint.flush_ppstream out_ppstr;
          write_line_opening Keyword.UNSOLVED;
          write_list Hthm.pr_hthm thms1;
          write_closing_line();
          output(std_out, "\nUsing the following hypotheses:");
          Proof.pp_proof_thm out_ppstr)
      end;
      clear_proof ()
    end)
```
end

end(* local *)

end; (* functor *)
The proof module models HOL proofs. A proof is a sequence of inferences. Each element of this sequence is called a **proof line** which represents a single inference step. It has a line number, a justification indicating which inference rule is applied, and a theorem derived by this inference.

The interface of the proof module is specified by the signature `Proof_sig`. The module is implemented by the functor `ProofFUN`. It requires the modules of type, term and theorem.

Its main task is to maintain several internal data structures which contain information about the proof. They are used to communicate between the two passes. The functions available to other modules can be roughly divided into two groups: those used in Pass 1 and those used in Pass 2.

### 9.1 The specification

The signature `Proof_sig` specifies the interface of the proof module. A proof is represented by the type `proof`. Each proof is a triple `(name, goals, lines)`. The `name` of a proof is a string. The `goals` are conjectures to be proved in the proof, and they are represented by the type `Hthm list`. The field `lines` is a sequence of proof lines represented by a list whose elements are of type `pline`.

Each `pline` is again a triple `(int * justification * hthm)`. The first field is the line number, the second is the justification, and the third is the derived theorem. The justification is represented by the type `justification`. The representations of these two types, `proof` and `pline` are private to this module.

```ml
signature Proof_sig =
  sig
    structure Hthm: Hthm_sig
    structure Hterm: Hterm_sig
    structure Htype: Htype_sig

    type pline
    type proof

    datatype justification =
      Hypothesis
      | Assume of Hterm.hterm
      | Refl of Hterm.hterm
      | Subst of (int*Hterm.hterm)list * Hterm.hterm * int
      | BetaConv of Hterm.hterm
      | Abs of Hterm.hterm * int
      | InstType of (Htype.htype * Htype.htype)list * int
      | Disch of Hterm.hterm * int
```

82
9.1. THE SPECIFICATION

Mp of int * int
| Axiom of (string * string)
| Definition of (string * string)
| DefExistsRule of (Hterm.hterm)
| NewAxiom of (string * Hterm.hterm)
| NewConstant of (string * Htype.htype)
| NewType of (int * string)
| Numconv of Hterm.hterm
| StoreDefinition of (string * Hterm.hterm)
| Theorem of (string * string)
| AddAssum of Hterm.hterm * int
| Alpha of Hterm.hterm * Hterm.hterm
| ApTerm of Hterm.hterm * int
| ApThm of int * Hterm.hterm
| Ccontra of Hterm.hterm * int
| Choose of Hterm.hterm * int * int
| Conj of int * int
| Conjunct1 of int
| Conjunct2 of int
| Constr of Hterm.hterm * int
| Disj1 of int * Hterm.hterm
| Disj2 of Hterm.hterm * int
| DisjCases of int * int * int
| EqImpRuleL of int
| EqImpRuleR of int
| EqMp of int * int
| EqTIntro of int
| EtaConv of Hterm.hterm
| Exists of (Hterm.hterm * Hterm.hterm) * int
| Ext of int
| Gen of Hterm.hterm * int
| ImpAntisymRule of int * int
| ImpTrans of int * int
| Inst of (Hterm.hterm * Hterm.hterm) list * int
| MkAbs of int
| MkComb of int * int
| MkExists of int
| NotElim of int
| NotIntro of int
| Spec of Hterm.hterm * int
| Subs of int list * int
| SubsSucs of (int list * int) list * int
| SubstConv of (int * Hterm.hterm) list * Hterm.hterm
| Sym of int
| Trans of int * int

The functions below are for resetting and initialising the internal data structures at various points. The function init is for cold start, i.e., it should be called before processing a file. The function clear proof should be called when the second pass starts. The functions new proof1 and new proof2 should be called at the beginning of every proof in Pass 1 and Pass 2, respectively.
val init: unit -> unit
val clear_proof: unit -> unit
val new_proof1: (string * Hthm.hthm list) -> unit
val new_proof2: string -> unit

The type goal represents goals in proof tables. If the field Proved is true, the goals have been proved.

datatype goal = Goal of {Thm: Hthm.hthm, Proved: bool}

The function lazy_mode is used to set or clear the lazy mode. When the lazy mode is set, the checker compresses the goals with the theorem derived in each step. When a match is found, the goal is flagged as proved. When all goals in a proof are proved, the checker skips the remaining proof lines.

val lazy_mode : bool -> unit

The function goal_table returns the current goal list. The function last_pline returns the line number of the last line in the current proof. The function get_final_thm returns the theorem in the last proof line.

val goal_table: unit -> goal list
val last_pline: unit -> int
val get_final_thm: unit -> Hthm.hthm

The following three functions are used in Pass 1. The function add_pline_tab should be called for each proof line to update the pline table. The function add_pline_table should be called at the end of a proof to save the current pline table. The function print_prooftab outputs the current pline table. It is mainly for debugging. The structure and the use of pline tables will be explained below.

val add_pline_tab : (int * int list * Hthm.hthm) -> bool
val add_pline_table : string -> unit
val print_prooftab : unit -> unit

The functions below are for use in Pass 2. The function add_pline should be called for every proof line. It saves the proof line in the current proof. The function get_last_lineNo returns the line number of the previous line. The function get_thm is called to fetch a theorem. Given the line number of the required theorem, it retrieves it from the current proof. The function find_thm takes a theorem, and searches the current proof to find the theorem. If it finds it, SOME is returned, otherwise, NONE is returned.

val add_pline: (int * justification * Hthm.hthm) -> int
val get_last_lineNo: unit -> int
val get_thm: (int * int) -> Hthm.hthm
val find_thm : Hthm.hthm -> Hthm.hthm option

The functions mk_pline and mk_proof are constructors for the types pline and proof, respectively.

val mk_pline: (int * justification * Hthm.hthm) -> pline
val mk_proof: (string * Hthm.hthm list * pline list) -> proof

The functions pp_just and pp_pline are the pretty printers for the justifications and proof lines, respectively. The function pp_proof.thm is used only in debugging.

val pp_just : System.PrettyPrint.postream -> justification -> unit
val pp_pline : System.PrettyPrint.postream -> pline -> unit
val pp_proof.thm : System.PrettyPrint.postream -> unit
end;
9.2 The implementation

The functor ProofFUN implements the proof module. It requires three modules: type, term
and theorem.

```ml
functor ProofFUN(structure Htm: Htm_sig
                 and Hterm: Hterm_sig and Htype: Htype_sig
                 sharing Htype = Hterm.Htype = Htm.Hterm.Htype
                 and Hterm = Htm.Hterm): Proof_sig =

struct
  structure Htm = Htm;
  structure Hterm = Hterm;
  structure Htype = Htype;

The internal data structures in this module need the dynamic array structure and binary
dictionary structure. The functors creating these structures are from the SML/NJ library.
A dynamic array is an array which can grow in size. When an attempt to access an element
with an index greater than the current size, the dynamic array is enlarged to accommodate
the element, thus, it provides an unbounded storage space. It is used to store the theorem
references. Using dynamic arrays, there is no hard limit on the number of proof lines in a
proof.

The integer-keyed dictionary is used to stored theorems. The keys are the line numbers.
```n
```ml
structure Sarray = StaticArrayFUN(type elemType = int);
structure Darray = DynamicArray (Sarray);
structure Int_key : ORD_KEY =
  struct
    type ord_key = int
    fun cmpKey (i,j:ord_key) =
      if (i = j) then LibBase.Equal
      else if (i < j) then LibBase.Less
      else LibBase.Greater
    end;
  structure Dict = BinaryDict(Int_key);
```

open Darray;

9.2.1 Defining types The type justification represents the justifications. The mean-
ings of various fields are explained in the Check module in Chapter 8.

data{type justification =
  Hypothesis
  | Assume of Hterm.hterm
  | Refl of Hterm.hterm
  | Subst of (int*Hterm.hterm)list * Hterm.hterm * int
  | BetaConv of Hterm.hterm
  | Abs of Hterm.hterm * int
  | InstType of (Htype.htype * Htype.htype)list * int
  | Disch of Hterm.hterm * int
  | Wp of int * int
  | Axiom of (string * string)
  | Definition of (string * string)
  | DefExistsRule of (Hterm.hterm)
  | NewAxiom of (string * Hterm.hterm)
| NewConstant of (string * Hterm.hterm) |
| NewType of (int * string) |
| Numconv of Hterm.hterm |
| StoreDefinition of (string * Hterm.hterm) |
| Theorem of (string * string) |
| AddAssum of Hterm.hterm * int |
| Alpha of Hterm.hterm * Hterm.hterm |
| ApTerm of Hterm.hterm * int |
| ApThm of int * Hterm.hterm |
| Ccontr of Hterm.hterm * int |
| Choose of Hterm.hterm * int * int |
| Conj of int * int |
| Conjunct1 of int |
| Conjunct2 of int |
| Contr of Hterm.hterm * int |
| Disj1 of int * Hterm.hterm |
| Disj2 of Hterm.hterm * int |
| DisjCases of int * int * int |
| EqImpRuleL of int |
| EqImpRuleR of int |
| EqM of int * int |
| EqTIntro of int |
| EtaConv of Hterm.hterm |
| Exists of (Hterm.hterm * Hterm.hterm) * int |
| Ext of int |
| Gen of Hterm.hterm * int |
| ImpAntisymRule of int * int |
| ImpTrans of int * int |
| Inst of (Hterm.hterm * Hterm.hterm)list * int |
| MkAbs of int |
| MkComb of int * int |
| MkExists of int |
| NotElim of int |
| NotIntro of int |
| Spec of Hterm.hterm * int |
| Subs of int list * int |
| SubsOccs of (int list * int) list * int |
| SubstConv of (int * Hterm.hterm) list * Hterm.hterm * Hterm.hterm |
| Sym of int |
| Trans of int * int |

The record type pline represents proof lines with the following three fields:

Line — the line number;

Just — the justification;

Thm — the derived theorem.

datatype pline =
  Pline of {Line: int, Just: justification, Thm: Hterm.hterm};

The record type proof represents proofs with the following three fields:
Name — the name of the proof;
Goals — a list of theorems to be derived in this proof;
Proof — a list of proof lines.

```
datatype proof =
  Proof of {Name: string, Goals: Htm. Htm list, Proof: pline list};
```

The type `pline_tab` represents `pline` tables. They are used to store theorem reference information found in the first pass. They also store the goals of proofs. Every proof will have a `pline` table. Each table has the following fields:

Name — the name of the proof;
Last — the line number of the last proof line;
Goals — the goals of the proof;
TabHyp — hypothesis table;
TabLine — inference line table.

The Goals field is a list of goals. Each element of the list represents a single goal. It has two fields: Thm is the goal, and Proved is a boolean value indicating whether the theorem is found in one of the proof lines. The fields TabHyp and TabLine are dynamic arrays whose elements are line numbers. These arrays are used to record the highest line which refers to a theorem. For example, if TabLine[2] equals 5, then the last reference to the theorem in Line 2 is in Line 5. This means that the derived theorem of Line 2 is no longer needed after Line 5 is checked. Since the indexes must be non-negative numbers, two arrays are used. One for the hypotheses whose line numbers are negative, and the other for the remaining lines. The TabHyp index is the absolute value of the hypothesis line number, e.g., the element TabHyp[3] is the hypothesis line — 3. The default value of these array elements is 0.

```
datatype goal = Goal of {Thm: Htm. Htm, Proved: bool};
datatype pline_tab =
  PlineTab of {Name: string, Last: int, Goals: goal list,
              TabHyp: Darray.array, TabLine: Darray.array};
```

9.2.2 Exception This function defines the exception in this module.

```
fun PROOF_ERR {function, message} =
  Exception.CK_ERR {message = message,
                   origin_function = function,
                   origin_structure = "Proof"};
```

val debug = Debug.get_debug("Proof");

9.2.3 Internal data structures The main purpose of the internal data structures is to store information about the proofs to be found in the first pass so that it can be used in the later pass. There are four dynamic objects for working with the current proof. Their names are prefixed by current_, and their usages are described below. They are initialised at the beginning of a proof, i.e., when the tag PROOF is found. In the first pass, information is cumulated in three of the dynamic structures. At the end of a proof, a structure of type pline_tab is created to group all information in these three structures. It is then inserted into the list containing all tables of the proofs in the current file to be used by the next pass.
The identifier current_goal refers to a list of goals of the current proof. From the observation of some proof generated by the HOL system, it has been found that sometimes the proof continues after the theorem matching the goal has been derived. In such a case, it is not necessary to check the entire proof if all goals are proved before the end of the proof is reached. The derived theorem of each line is compared with the goals. If a goal is found to be equal to the theorem, the goal is deleted from the list of current goals. When the last goal is deleted, we can ignore any remaining lines in the proof. By including this mechanism, it is hoped that the checker may be able to save some time so it is more efficient. However, the usefulness of this mechanism in practical, large proofs is still not clear, because it may not save time as comparing the goals with the derived theorem in every line takes time.

The line number of the last line is recorded in the Last field of the pline table. However, if there is no goal, we are obliged to check the entire proof. This laziness feature is enabled if the value of lazy is true. The default value is false when the checker starts. The value of lazy should not be changed in the middle of a proof.

At the beginning of a proof, proof_goal is bound to the same list of goals as current_goal. In Pass 1, as the proof lines are processed, when a goal is found to be the same as a theorem, it is deleted from current_goal. At the end of the proof, what remains in the list current_goal will be unsolved. A pline table is built for the proof, in which Goals contains the list bound to proof_goal with all goals proved (i.e., not in current_goal) flagged as such.

The identifier current_p tab refers to a pair of dynamic arrays containing theorem reference information. This information is used in the same way as in the description of type pline_tab above.

The identifier current_pline refers to the line number of the latest line being processed.

The identifier current_proof refers to a dictionary of theorems keyed by their line numbers. This dictionary is used only in Pass 2. Initially, it is empty. When a proof line is processed, the derived theorem is added to the dictionary if it is referenced by later line(s), i.e., the value of the corresponding element in the pline table array is greater than the current line number. When the theorem is retrieved from the dictionary for the last line referencing it, it is deleted. In this way, the size of the dictionary is kept to a minimum.

The name final_thm is bound to the theorem of the last line in a proof. This is assumed to be the goal of the proof if there is no goal specified explicitly at the beginning of the proof.

```plaintext
val default_p tab = ((Dar ray.array(10,0)), (Dar ray.array(10,0)))
val current_p tab = ref default_p tab
and proof_goal = ref([]:Hthm.h thm list)
and current_goal = ref ([]:Hthm.h thm list)
and goal_tab = ref([]:goal list)
and pline_tables = ref ([]:pline_tab list)
and current_pline = ref 0
and last_line = ref 0
and dummy_thm = (Hthm.mk_thm([],Hterm.mk_const("T",Htype.bool_ty)))
and dummy_proof = (Dict.mkd graph):Hthm.h thm Dict.dict

val final_thm = ref dummy_thm
and current_proof = ref dummy_proof
val lazy = ref false
fun lazy_mode b = (lazy := b)
```

9.2.4 Functions for Pass 1 The function add_pline_tab processes a proof line in Pass 1. It takes a triple as its argument. The first field n is the line number, the second ks is a list of line numbers to which this line references, the last thm is the derived theorem. It returns a
boolean value true if all the goals of the current proof have been found, otherwise, false.

The current_pline table, i.e., the arrays pointed to by current_ptab, is updated. The values of the array elements whose indices occur in the line number list ks are updated to the current line number. This is performed by the local function update_tab. Likewise, the current line number current_pline is updated. The local function chk checks the current goal list. If one of the goals is the same as the theorem thm, it is deleted. If this is the last goal, true is returned to indicate that all the goals have been proved. This checking is done only if lazy mode is enabled.

```plaintext
fun add_pline_tab (n, ks, thm) = 
  let
    fun update_tab 0 = ()
    | update_tab k = 
      if k > 0 then update(#2(!current_ptab), k, n)
      else update(#1(!current_ptab), ~k, n)
    and check_goal thm = 
      let
        fun chk th [] = false
        | chk th (th' :: thms) = 
          if (Hthm.thm_eq th th') then 
            (current_goal := thms;
             if (null thms) then true else false)
          else chk th thms
        in
        chk thm (!current_goal)
      end
    in
      (map update_tab ks; current_pline := n;
       if (!lazy) then check_goal(thm) else false)
    end

The function add_pline_table is called at the end of a proof. It builds a pline table (of type pline_tab) and adds it to the pline table list pline_tables.

```plaintext
fun add_pline_table name = 
  let
    val gls = (!current_goal) 
    val g10 = null gls
    fun chk_goal th = 
      if g10 then true
      else (not(exists (fn th' => Hthm.thm_eq th th') gls))
    val (tabh,tabl) = !current_ptab
    val gs = 
      if not(!lazy) then []
      else map (fn th => Goal(Thm=th,Proved=chk_goal th)) (!proof_goal)
    in
      pline_tables := (PlineTab(Name=name,Last=(!current_pline),
        Goals = gs,
        TabHyp=tabh,TabLine=tabl)):!(pline_tables);
    ()
  end

The internal function ptab_value retrieves the value of an element from the current pline arrays.
The function print_prooftab outputs the current pline table arrays to the std_out stream. It is mainly for debugging.

```plaintext
fun ptab_value n = 
  if (n > 0) then sub(#2(!current_ptab), n)
  else sub(#1(!current_ptab), "n"

fun print_prooftab () = 
let
  val (hyptab,linetab) = (!current_ptab)
  val hb = bound hyptab
  and lb = bound linetab
  val i = ref hb
  in
  (while(!i > 0) do
    (output(std_out, 
      ("-" makestring (!i) ") " = " makestring (sub(hyptab,(!i)))"\n"
    ))
  i := (!i - 1); 
  while(!i < lb) do
    (output(std_out, 
      (makestring (!i) " = " makestring (sub(linetab,(!i)))"\n"))
  i := (!i + 1);
  ()
  end

fun reset_line_num () = (current_pline := 0);
```

9.2.5 Functions for Pass 2 Functions in this section are used in Pass 2. The function print_cur_proof prints out the current proof dictionary. It is mainly for debugging.

```plaintext
fun print_cur_proof () = 
let
  val thms = Dict.listItems(!current_proof)
  fun prt (ln,thm) = output(std_out,(makestring ln)" ")
  in
    output(std_out,("current_proof:[ ")
    map prt thms;
    output(std_out, "]\n")
  end
```

The function add_pline should be called for every proof line after it is checked. It inserts the derived theorem into the current proof dictionary if it is referenced by later line(s). It returns the line number of the expected last proof line in the proof.

```plaintext
fun add_pline (line,just,thm) = 
(if (ptab_value line > line) then
  (if (debug > 1) then
    output(std_out,"\n")
  else ()
  current_proof := Dict.insert( (!current_proof),line,thm);
  if (debug > 1) then print_cur_proof() else ()
```
else();
current_pline := line;
if (line = (!last_line)) then final_thm := thm else ();
(!!last_line));

fun get_last_lineNo () = !current_pline;

The function get_thm retrieves a theorem from the current proof dictionary. It takes two
line numbers as its arguments. The first one ln is the line number of the required theorem.
The second n is the line number of the line which uses the theorem. If n is the last line that
references the theorem in line ln, the theorem is removed from the dictionary. However, no
hypothesis will be removed as they are reported at the end of the proof.

local
fun get_th n [] = NONE
  | get_th n ((l as Pline(Line=line,...)) :: thmlst) =
    if (n = line) then (SOME l) else get_th n thmlst
fun find_th th [] = NONE
  | find_th th ((Pline(Thm=tmh,...)) :: thmlst) =
    if (Htm.thm_eq th tmh) then (SOME thm) else find_th th thmlst
in
fun get_thm (ln, n) =
  if (ln < (ptab_value n)) orelse (n < 0) then
    Dict.find((!current_proof),n)
  else
    let
      val (dict,th) = Dict.remove((!current_proof), n)
    in
      current_proof := dict;
      th
    end
handle Dict.NotFound =>
  raise (PROOF_ERR {function="get_thm",
                     message="Line ":{(makestring n)"
                     ("not found")})
  | e => raise e
  and find_thm thm = NONE (* find_thm (!current_proof) *)
end;

9.2.6 Constructors and field selector These two functions, mk_pline and mk_proof,
are the constructors for the types pline and proof, respectively.
The functions pline_No, pline_Just and pline_Thm return the respective fields of a pline.

fun mk_pline (line,just,tmh) =
  Pline{Line=line, Just=just, Thm=tmh}
and mk_proof (name, goals, lines) =
  Proof{Name=name, Goals=goals, Proof=lines};

fun pline_No (Pline{Line,...}) = Line
and pline_Just (Pline{Just,...}) = Just
and pline_Thm (Pline{Thm,...}) = Thm;

9.2.7 Initialisation The function init initialises the internal data structures. It should be
called at the beginning of the first pass. The function new_proof1 initialises the current goal
list and pline table. It should be called at the beginning of every proof in Pass 1.

```
fun init () =
  (current_proof := dummy_proof;
   current_ptab := ((Darray.array(10,0)), (Darray.array(10,0)));
   pline_tables := [];
   ());

and new_proof1 (name, thms) =
  (current_goal := thms;
   proof_goal := thms;
   current_ptab := ((Darray.array(10,0)), (Darray.array(10,0)));
   ());
```

The next two functions, clear_proof and new_proof2, are used in the second pass to initialise the internal data structures in a way similar to the above first pass functions.

```
fun clear_proof () =
  (current_proof := dummy_proof; ())

and new_proof2 name =
  let
    fun find_ptab name [] =
      raise (PROOF_ERR {function="new_proof2",
                         message="Proof "$name" not found"})
    in
      find_ptab name
        ((tab as PlineTab{Name,Goals,Last,TabHyp,TabLine}):xs) =
        if (name = Name) then tab
        else find_ptab name xs
    end

    val (PlineTab ptab) = find_ptab name (!pline_tables)

    val current_ptab := ((#TabHyp ptab),(#TabLine ptab));
    goal_tab := (#Goals ptab);
    last_line := #Last ptab;
    current_proof := dummy_proof;
    current_pline := "(Darray.bound(#1(!current_ptab)));"
    ()
  end
```

The following three functions are provided to allow other modules to access some internal data. The function goal_table returns the current goal list. The function last_pline returns the line number of the last line in the current proof. The function get_final_thm returns the theorem in the last proof line.

```
and goal_table () = (!goal_tab)
and last_pline () = (!last_line)
and get_final_thm () = (!final_thm)
```

9.2.8 Pretty Printer  The function pp-proof-thm prints the theorems in the current_proof theorem table. Each theorem is preceded by its line number. This is used in debugging. The functions pp-just and pp-pline print a justification and a proof line, respectively, to a ppstream. They use the system pretty printer structure.

```
local
  open System.PrettyPrint;
  fun with_ppstream ppstrm =
    (add_string = add_string ppstrm,
```

```
add_break = add_break ppstrm,
begin_block = begin_block ppstrm,
end_block = fn () => end_block ppstrm,
flush_ppstream = fn () => flush_ppstream ppstrm);

in

fun pp_proof_thm ppstrm =
  let
    val {add_string, add_break,
         begin_block, end_block, flush_ppstream} =
        with_ppstream ppstrm
    and pp_hthm = Hthm.pp_hthm ppstrm
    and thms = Dict.listItems(!current_proof)
    fun pp_thmno n =
      (add_string "\n<";
       add_string (Format.format "%d" [(Format.INT n)]);
       add_string ">")
    and prt (ln,thm) =
      (pp_thmno ln; add_string " "; pp_hthm thm; add_break (1,0))
    in
      if (debug > 1) then
        output(std_out, ("pp-proof-thm:"~(makestring (length thms))~"\n"))
      else ()
      if null thms then ()
      else (begin_block CONSISTENT 0;
            map prt thms;
            end_block ()
            add_string "\n";
            flush_ppstream())
    end

9.2.9 Pretty printer for justification  Each justification is enclosed in a pair of parentheses. The name of the justification is printed first followed by the other fields. A theorem is printed as a line number surrounded by angle brackets. Terms are printed using the term pretty printer.

fun pp_just ppstrm just =
  let
    val {add_string, add_break,
         begin_block, end_block, flush_ppstream} =
        with_ppstream ppstrm;
    val pp_hthm = Hthm.pp_hthm ppstrm
    and pp_qterm = Hterm.pp_qterm ppstrm
    and pp_htype = Htype.pp_htype ppstrm
    fun pp_thmno n =
      (add_string "<";
       add_string (Format.format "%d" [(Format.INT n)]);
       add_string ">")
    and pp arity n =
      if (a = 0) then ()
      else (add_string "("; add_string (makestring n);
            add_string ")")
fun pp_list pf l =
  let
    fun plist pf [] = ()
    | plist pf [x] = pf x
    | plist pf (x::(l as (y::ys))) =
      (pf x; add_string";"; add_break(1,0); plist pf l)
  in
    (begin_block CONSISTENT 0;
     add_string "[";
     plist pf l;
     add_string "]";
     end_block ());
  end

and pp_pair (f1,f2) (x1,x2) =
  (begin_block CONSISTENT 0;
   add_string(""; f1 x1; add_string",";
   f2 x2; add_string")";
   end_block ());

in
case just of
  Hypothesis => add_string "Hypothesis"
  | (Assume term) => (add_string "ASSUME "; pp_qterm term)
  | (Ref1 term) => (add_string "REFL "; pp_qterm term)
  | (Subst (itml,tm,n)) => (add_string "SUBST ";
    pp_list (pp_pair (pp_thmno,pp_qterm)) itml;
    add_break (1,3);
    pp_qterm tm; add_string";";
    pp_thmno n)
  | (BetaConv term) => (add_string "BETA_CONV "; pp_qterm term)
  | (Abs (term,n)) => (add_string "ABS ";
    pp_qterm term; add_string";"; pp_thmno n)
  | (InstType (ttl,n)) => (add_string "INST_TYPE ";
    pp_list (pp_pair (pp_hotype,pp_hotype)) ttl;
    pp_thmno n)
  | (Disch (term,n)) => (add_string "DISCH ";
    pp_qterm term; add_string";"; pp_thmno n)
  | (Mp (n1,n2)) => (add_string "MP ";
    pp_thmno n1; add_string";"; pp_thmno n2)
  | (Axiom (s1,s2)) => (add_string "AXIOM ";
    add_string s1; add_string"."; add_string s2)
  | (Definition (s1,s2)) => (add_string "DEFINITION ";
    add_string s1; add_string"."; add_string s2)
  | (DefExistsRule term) => (add_string "DEF_EXISTS_RULE";
    pp_qterm term)
  | (NewAxiom (s,term)) => (add_string "NEW_AXIOM ";
    pp_qterm term)
  | (NewConstant (s, ty)) => (add_string "NEW_CONSTANT ";
    pp_qterm (Hterm.mk_const(s,ty)))
  | (NewType (n,s)) => (add_string "NEW_TYPE ";
    pp_arity n; add_string s)
9.2. THE IMPLEMENTATION

| (Numconv term) => (add_string "num_CONV ";
| pp_qterm term) |
| (StoreDefinition (s,term)) => (add_string "STORE_DEFINITION ";
| add_string s; pp_qterm term) |
| (Theorem (s1,s2)) => (add_string "THEOREM ";
| add_string s1; add_string "."; add_string s2) |
| (AddAssum (term, n)) => (add_string "ADDASSUM ";
| pp_qterm term; pp_thmno n) |
| (Alpha (term1,term2)) => (add_string "ALPHA ";
| pp_qterm term1; pp_qterm term2) |
| (ApTerm (term,n)) => (add_string "APTERM ";
| pp_qterm term; pp_thmno n) |
| (ApThm (n,term)) => (add_string "APTHM ";
| pp_thmno n; pp_qterm term) |
| (Ccontr (term,n)) => (add_string "CCONTR ";
| pp_qterm term; pp_thmno n) |
| (Choose (term,n1,n2)) => (add_string "CHOOSE ";
| pp_qterm term; pp_thmno n1; pp_thmno n2) |
| (Conj (n1,n2)) => (add_string "CONJ ";
| pp_thmno n1; pp_thmno n2) |
| (Conjunct1 n) => (add_string "CONJUNCT1 "; pp_thmno n) |
| (Conjunct2 n) => (add_string "CONJUNCT2 "; pp_thmno n) |
| (Contr (term,n)) => (add_string "CONTR ";
| pp_qterm term; pp_thmno n) |
| (Disj1 (n,term)) => (add_string "DISJ1 ";
| pp_thmno n; pp_qterm term) |
| (Disj2 (term,n)) => (add_string "DISJ2 ";
| pp_qterm term; pp_thmno n) |
| (DisjCases (n1,n2,n3)) => (add_string "DISJCASES ";
| pp_thmno n1; pp_thmno n2; pp_thmno n3) |
| (EqImpRuleL n) => (add_string "EQUIPRULEL "; pp_thmno n) |
| (EqImpRuleR n) => (add_string "EQUIPRULER "; pp_thmno n) |
| (EqMp (n1,n2)) => (add_string "EQMP "; pp_thmno n1; pp_thmno n2) |
| (EqIntro n) => (add_string "EQINTRO "; pp_thmno n) |
| (EtaConv term) => (add_string "ETA_CONV "; pp_qterm term) |
| (Exists ((term1,term2),n)) => (add_string "EXISTS ";
| pp_qterm term1; pp_qterm term2; pp_thmno n) |
| (Ext n) => (add_string "EXT "; pp_thmno n) |
| (Gen (term,n)) => (add_string "GEN ";
| pp_qterm term; pp_thmno n) |
| (ImpAntisymRule (n1,n2)) => (add_string "IMPAntisymRule ";
| pp_thmno n1; pp_thmno n2) |
| (ImpTrans (n1,n2)) => (add_string "IMPTRANS ";
| pp_thmno n1; pp_thmno n2) |
| (Inst (ttl,n)) => (add_string "INST ";
| pp_list (pp_pair (pp_qterm, pp_qterm)) ttl;
| add_break(1,3);
| pp_thmno n) |
| (MkAbs n) => (add_string "MKAbs "; pp_thmno n) |
| (MkComb (n1,n2)) => (add_string "MKComb ";
| pp_qterm n1; pp_qterm n2) |
9.2.10 Pretty printer for proof line  The proof lines are printed in the following form:

\[ \text{linenumber): (justification) (theorem) } \]

fun pp_pline ppstrm (Pline(Line,Just,Thm)) =
  let
    val {add_string, add_break, begin_block, end_block, flush_ppstream} =
      with_ppstream ppstrm;
    val pp_hthm = Hthm.pp_hthm ppstrm
    and pp_just = pp_just ppstrm;
    fun mk_str n = Format.format "\%d: " [(Format.INT n)]
  in
    (begin_block CONSISTENT 0;
     add_string (mk_str Line);
     pp_just Just;
     add_break (1,6);
     case Just of
       (NewType _) => ()
       | (NewConstant _) => ()
       | _ => pp_hthm Thm;
     add_string "\n";
     end_block ();
     flush_ppstream())
  end
end;  (* end of functor ProofFUN *)
The proof checker works within an environment which consists of a type structure and a constant structure. The type structure contains all currently known type operators and their arities. The constant structure contains all known constants and their types. The proof file format prf specifies that every proof file has an environment expression at the beginning. It gives the type and constant structures to be used in checking the proofs in the file. There are three default environments. The checker starts with the default environment HOL which contains the same types and constants as when the HOL system starts. This default environment will be augmented first by the environment specified at the beginning of a proof file. Then, as checking progresses, new types and constants may be added by the justifications new_type and new_constant, respectively.

10.1 The specification

The environment module has the following signature:

```
signature Henv_s =
  sig
    structure Htype : Htype_s
    structure Hterm : Hterm_s
```

10.1.1 New types The type htypeConst represents an element in the type structure. The type structure is part of the current environment of the checker. A type constant has a name and an arity.

The type htermConst represents a constant in the signatures (or constant structure) of the current environment. It contains a set of constants known to the current theory.

The type henv represents the current environment. It consists of the type structure and the constant structure.

```
type htypeConst
type htermConst
type henv
```

10.1.2 Functions and identifiers The function mk_proof_env creates a new proof environment. It is called when an ENV expression is found by the parser. It takes a triple (name, tylist, clist) as its argument. The name envList is bound to the list of default environments.

```
val mk_proof_env : (string * htypeConst list * htermConst list) -> unit
val envList : (string * htypeConst list * htermConst list) list
val mk_typeconst : (string * int) -> htypeConst
val mk_termconst : (string * Htype.h) -> htermConst
```

The function init should be called to initialise the environment before starting reading a proof file.

```
val init: unit -> unit
```

The function envName returns the name of the current environment. A list of all type operators in the current environment is returned by the function knownTypes. Similarly, the function knownConstants returns a list of constants in the current environment.
val envName : unit -> string
val knownTypes : unit -> htypeConst list
val knownConstants : unit -> htermConst list

The functions add_type and add_const add a new type operator and a new constant to
the current environment, respectively.

val add_type : (string * int) -> unit
val add_const : (string * Htype.htype) -> unit

10.1.3 Checking type well-formedness The function type_OK takes a type ty. It checks
whether ty is well-formed under the current environment. A type is well-formed if and only
if (i) it is a type variable; or (ii) its name appears in the type structure and the number of
arguments equals to the arity given in the structure. The function well_typed returns true if
its argument term is well-typed in the current environment.

val type_OK : Htype.htype -> bool
val well_typed : Hterm.hterm -> bool
end; (* signature *)

10.2 The implementation

functor HenvFUN (structure Htype : Htype_sig
and Hterm : Hterm_sig
sharing Htype = Hterm.Htype) : Henv_sig =

struct
structure Htype = Htype;
structure Hterm = Hterm;

10.2.1 Types The type htypeConst represents type constructors. It is a pair consisting of
a string and a number. The string is the name of the type constructor. The number is its arity.
A type structure is a list of type constructors.

datatype htypeConst = TYCON of {Name:string, Arity:int};
The type htermConst represents Constant terms. The type henv represents a proof envi-
ronment which consists of a type structure and a constant structure.

datatype htermConst = CCON of {Name:string, Ty:Htype.htype};
datatype henv = HENV of {Name: string,
    Tys: htypeConst list,
    Tms: htermConst list};

open ListUtil Htype Hterm;

10.2.2 Error handling The function ENV.ERR defines exceptions of this module.

fun ENV.ERR{function,message} =
    Exception.CHK.ERR{origin_structure="Henv",
        origin_function = function, message = message};

val debug = ref 0;

10.2.3 Constructors and destructors These are the constructors and destructors of the
type operators and constant terms.

fun mk_typeconst (name,arity) = TYCON {Name=name, Arity=arity}
and mk_termconst (name,ty) = CCON {Name=name, Ty=ty}
and dest_typeconst (TYCON{Name,Arity}) = (Name, Arity)
and dest_termconst (CCON{Name,Ty}) = (Name, Ty)

10.2.4 Current environment An empty environment is bound to the name dummy_env. The
internal variable current_env is always pointing to an environment to be used as the
current environment. Initially, the current environment is the dummy environment. This is set
in the function init:

val dummy_env = (HENV {Name="", Tys=[], Tms=[]});
val current_env = ref dummy_env;

fun init () =
  (debug := Debug.get_debug "Henv";
   current_env := dummy_env; ());

The functions envName, knownTypes and knownConstants return respectively the name,
type structures and constant structures of the current environment. The function init should
be called to clear the current environment when the checker begins.

fun envName () =
  let val HENV {Name, ...} = !current_env in Name end
and knownTypes () =
  let val HENV {Tys, ...} = !current_env in Tys end
and knownConstants () =
  let val HENV {Tms, ...} = !current_env in Tms end

10.2.5 Default environments We first initialise the default environments. To improve
the readability of these environments, some identifiers are defined locally. Amongst these local
identifiers, some are symbolic, and they stand for type variables, such as *, and type operators,
such as -->. Others are normal identifiers. They stand for type constants and type operators,
such as bool and list and so on.

local
  nonfix *;

val * = mk_vartype "*"
and ** = mk_vartype "**"
and *** = mk_vartype "***";

val bool = bool_ty
and num = mk_type {Tyargs=[],Tyop="num"}
and ind = mk_type {Tyargs=[],Tyop="ind"}
and one = mk_type {Tyargs=[],Tyop="one"}
and tree = mk_type {Tyargs=[],Tyop="tree"};

infixr 3 -->;
infix 4 ++;
infix 5 ##;

fun ty1 --> ty2 = mk_funtype (ty1,ty2)
and ty1 ++ ty2 = mk_type {Tyargs=[ty1,ty2],Tyop="sum"}
and ty1 ## ty2 = mk_type {Tyargs=[ty1,ty2],Tyop="prod"}
and list ty = mk_type {Tyargs=[ty],Tyop="list"}
and ltree ty = mk_type {Tyargs=[ty],Tyop="ltree"};
in

There are three default environments, MIN, LOG and HOL. See Section 16.4 of [5] for the
definition of the first two environments. Each environment in the list includes all the types and
constants of its preceding environments, e.g., HOL includes all the types and constants of MIN
and LOG. HOL is the initial environment when the HOL system starts.

val envlist = [
  ("HOL", map mk_typeconst
   [ ["at", 2], ["prod", 2], ["num", 0], ["list", 1],
     ["tree", 0], ["ltree", 1], ["sum", 2], ["one", 0]]),
  map mk_termconst
   [ ["=", (num --> (num --> num))],
     ["<", (num --> (num --> num))],
     [">", (num --> (num --> bool))],
     ["==", (num --> (num --> bool))],
     ["??", (num --> (num --> num))],
     ["ABS_list", (num --> list(*))],
     ["ABS_ltree", (tree --> list(*))],
     ["ABS_sum", (num --> (num --> num))],
     ["ABS_tree", (num --> tree)]],
  ("if", (num --> (num --> bool)))]
]
10.2. THE IMPLEMENTATION

("IS_PAIR", (( * -- ( ** -- bool)) -- bool)),
("IS_SUMREP", (bool -- (( * -- ( ** -- bool))) -- bool)),
("IS_list_coeff", (num -- * ) # num -- bool)),
("T", ( * -- * ))),
("Is_ltree", (tree # list( * ) -- bool)),
("Is_tree_coeff", (num -- bool)),
("K", ( * -- ( ** -- * ))),
("LENGTH", (list( * ) -- num)),
("LET", (( * -- * ) -- ( * -- * ))),
("MAP2", (( * -- ( ** -- ** )) -- (list( * ) -- (list( ** ))) -- list( ** ))),
("MAP", (( * -- * ) -- (list( * ) -- list( ** )))),
("MH_PAIR", ( * -- ( ** -- ( * -- ( ** -- bool)))))},
("MOD", (num -- (num -- num))},
("NIL", (list( * ))),
("NULL", (list( * ) -- bool)),
("Node", ( * -- (list(ltree( *)) -- ltree( * )))),
("ODD", (num -- bool)),
("OUT", ( * + * -- * )),
("OUTR", ( * + * -- * )),
("PART", (list(num) -- (list( * ) -- list(list( * ))))),
("PRE", (num -- num)),
("PRIMA_REC_FUN", ( * -- (( * -- (num -- * )) --
(num -- (num -- * ))))),
("PRIMA_REC", ( * -- (( * -- (num -- * )) -- (num -- * )))),
("REP_list", (list( * ) -- (num -- * ) # num))},
("REP_ltree", (ltree( * ) -- tree # list( * ))),
("REP_num", (num -- ind)),
("REP_prod", ( * * -- ( * -- ( ** -- bool)))),
("REP_sum", ( * * -- (bool -- ( * -- ( ** -- bool)))))},
("REP_tree", (tree -- num)),
("RES_ABSTRACT", (( * -- bool) -- ( ( * -- ) --
( * -- )))),
("RES_EXISTS", (( * -- bool) -- ( ( * -- bool) -- bool))),
("RES_FORALL", (( * -- bool) -- ( ( * -- bool) -- bool))),
("RES_SELECT", (( * -- bool) -- ( ( * -- bool) -- * ))),
("SIMP_REC_FUN", (( * -- ( * -- )) -- (num -- (num -- * ))))),
("SIMP_REC_REL", ((num -- * ) -- ( * -- ( ( * -- * ) --
(num -- bool)))))),
("SIMP_REC", ( * -- ( ( * -- * ) -- (num -- * )))),
("SND", ( * * ** -- ** ))),
("SPLIT", (num -- (list( * ) -- list( * ) # list( * ))),
("SUCстоль", (ind -- ind)),
("SUC", (num -- num)),
("SUM", (list(num) -- num)),
("S", (( * -- ( ** -- * )) --
( ( * -- ** ) -- ( * -- ** ))))},
("Size", (tree -- num)),
("T", (list( * ) -- list( * ))),
("TRP", (( * -- (list(ltree( *)) -- bool)) --
(ltree( * ) -- bool)))},

...
("UNCURRY", (( * --> ( ** --> *** )) --> ( * ## ** --> *** ))),
("ZERO_REP", (ind)),
("bht", (num --> (tree --&gt; bool))),
("dest_node", (tree --&gt; list(tree))),
("node_REP", (list(num) --&gt; num)),
("node", (list(tree) --&gt; tree)),
("o", (( ** --&gt; *** ) --&gt; (( * --&gt; **) --&gt; ( * --&gt; *** )))),
("one", (one)),
("trf", (num --&gt; ((list( **) --&gt; **) --&gt; (tree --&gt; ** ))))]
)

("LOG", []),
map mk_termconst
[(["!", (( * --&gt; bool) --&gt; bool))],
(["?", (( * --&gt; bool) --&gt; bool))],
(["T", (bool)]),
(["F", (bool)]),
(["\-", (bool --&gt; bool))],
(["/\", (bool --&gt; (bool --&gt; bool))],
(["/\", (bool --&gt; (bool --&gt; bool))],
(["ONE_ONE", (( * --&gt; **) --&gt; bool))],
(["ONTO", (( * --&gt; **) --&gt; bool))],
(["TYPE_DEFINITION", (( * --&gt; bool) --&gt; (( ** --&gt; *) --&gt; bool))]]
),

("MIN", map mk_typeconst ["bool", 0], "ind", 0]
map mk_termconst
[(["==>/", (bool --&gt; (bool --&gt; bool))],
(["==>/", (bool --&gt; (bool --&gt; bool))],
(["==>/", (bool --&gt; (bool --&gt; bool))]]
)
end(* local *)

10.2.6 Setting up the environment When an ENVironment expression is found in the input, the function mk_proof_env is called. This function sets up the current proof environment. The expression, represented as a triple (name, tylst, clst), is passed to this function as its argument.

If name is the name of one of the default environments in the list envList, tylst and clst are ignored. The current environment will consist of all the type and constant structures of the environment(s) in the default list from the first element up to and including the element having the same name as name.

If name is not in the default environment, it is taken as a new environment built on top of the default environment HOL. The type and constant structures in the argument, together with the default structures, are set up as the current environment.

local
fun mk_env ((n,tyl',conl'),(tyl,conl)) =
((tyl' @ tyl), (conl' @ conl))
in
fun mk_proof_env (name, tyl, conl) =
let val (l1,l2) =
  splitp (fn (n,l1,l2) => (n = name)) envList
in
  if (null l2) then
let val (tys,tms) =  
  fold mk_env ((name,tyl,conl):envList) ([][],[])  
in  
current_env := HENV{Name=name, Tys=tys, Tms=tms};  
if (!debug > 0)  
  then output(std_out, ("Current environment: "~name~\n");  
else ()  
end  
else let val (tl,cl) = fold mk_env 12 ([] [],[])  
in  
current_env := HENV{Name= #1(hd 12), Tys=tl, Tms=cl};  
if (!debug > 0)  
  then output(std_out,  
    ("Current environment: "~(#1(hd 12))~\n")  
else ()  
end  
end

10.2.7 Well-formedness and well-typedness  A type is well-formed under an environment if it is a type variable, or if the name of the type operator appears in the type structure env and all its arguments are also well-formed.

A variable is well-typed under the environment (tysl,constsl) if and only if its type is well-formed. A constant is well-typed if and only if it occurs in the constant structure constsl and its type is an instance of the type given in the constant structure. Numerical constants are exceptions since they are not in the constant structure. They are well-typed if and only if their type is num and their names are strings consists only of digits. Terms which are function applications, are well-typed if and only if the operator is of function type and is well-typed, and the operand is well-typed and in the domain of the operator. For abstractions, the bound variables and the bodies should be well-typed.

The predicate type_OK returns true if its argument is a well-formed type under the current environment. Similarly, the predicate wellTyped returns true if its argument is a well-typed term under the current environment.

local

fun in_consts cl (name,cty) =  
  let val found = findOne (fn (s,ty) => (s = name)) cl  
in  
case found of  
  NONE => false  
  | SOME(_,ty) => is_type_inst cty ty  
end;

fun type_OK ty =  
  if (is_vartype ty) then true  
else  
  let  
    val {Tyop,Tyargs} = dest_type ty  
    val tysl = map dest_typeconst (knownTypes())  
    val found = findOne (fn (s,a) => (s = Tyop)) tysl  
in  
case found of
NONE => false
| SOME (_,arity) =>
    (arity = length Tyargs) andalso (every type_OK Tyargs)
end;

fun well_typed tm = let
    fun is_digit_str s = every (CTYPE.isDigitOrd o ord) (explode s)
    val constsl = map dest_termconst (knownConstants())
in
    if (is_var tm) then
        let val ty = #2(dest_var tm) in (type_OK ty) end
    else if (is_const tm) then
        let val (name,ty) = dest_const tm in
            if (is_digit_str name) then (ty_is_num ty)
            else (in_consts constsl (name,ty)) end
    andalso (type_OK ty)
end

else if (is_comb tm) then
    let
        val (rator,rand) = dest_comb tm
        val (ranty,rtty) = dest_funtuple (type_of rator)
    in
        (well_typed rator) andalso (well_typed rand) andalso
        (ranty = (type_of rand))
    end
handle CHK_ERR _ => false

else if (is_abs tm) then
    let val (bv,body) = dest_abs tm
    in
        (well_typed bv) andalso (well_typed body)
    end

else false
end

end; (* local *)

10.2.8 Augmenting the environment: The functions add_type and add_const add a new type and a new constant to the current environment, respectively. We need to check whether the new type or the new constant is already in the environment. When adding constants, we also need to check the well-formedness of its type.

fun add_type (tname, arity) = let val HENV = {!current_env
    val found = findOne (fn (ty,ar) => (ty = tname))
    (map dest_typeconst tys)
in
    case found of
        SOME (_, ar) =>
            if (ar = arity)
            then raise (ENV_ERR {function="add_type",
                message="type already in environment"})
            else raise (ENV_ERR {function="add_type",}
message="type name crash")
| NONE =>
    (current_env := HENV{Name=n,
                         Tys=((TYCON{Name=tname,Arity=arity}):tys),
                         Tms=tms});

| ()
end

and add_const (name, ty) =
let val HENV {Name=n,Tys=tys,Tms=tms} = !current_env
    val found = findOne (fn (n,ty) => (n=name)) (map dest_term const tms)
in
  case found of
    SOME (_,typ) =>
      if (typ = ty)
        then raise (ENV_ERR {function="add_const",
                              message="constant already in environment")}
      else raise (ENV_ERR {function="add_const",
                              message="constant name crash")})

| NONE =>
    if (type_OK ty) then
      (current_env := HENV{Name=n, Tys=tys,
                        Tms=((CCON{Name=name,Ty=ty}):tms)};

    | ()
      else raise (ENV_ERR {function="add_const",
                              message="type of constant badly formed"})

end;(* functor HenvFun *)
11 HOL theorems

The abstraction of HOL theorems is in the module Hthm. The signature Hthm_sig specifies this abstraction.

11.1 The specification

The type hthm represents theorems. A theorem has two parts: the set of hypotheses and the conclusion. These two parts are accessed by the functions hyp and concl, respectively. The function mk_thm creates a theorem when supplied a list of hypotheses and a conclusion. All of these must be boolean terms.

The function thm_eq returns true when applied to two theorems if the conclusions are identical and their sets of hypotheses are equivalent.

The pretty printer for theorems is pp_hthm.

signature Hthm_sig =
  sig
    structure Report : Report_sig
    structure Hterm : Hterm_sig
    structure HtermSet : ORD_SET

    type hthm

    val concl : hthm -> Hterm.hterm
    val hyp : hthm -> HtermSet.set
    val mk_thm : (Hterm.hterm list * Hterm.hterm) -> hthm
    val thm_eq : hthm -> hthm -> bool
    val pr_hthm : hthm -> unit
    val pp_hthm : System.PrettyPrint.ppstream -> hthm -> unit end

11.2 The implementation

The functor HthmFUN generates a structure for theorems when applied to the correct arguments. The argument structure Hterm represents terms, and the argument structure HtermSet represents sets of terms.

functor HthmFUN (structure Report : Report_sig
  and Hterm: Hterm_sig and HtermSet: ORD_SET
  sharing type Hterm.hterm = HtermSet.item
  and Report = Hterm.Report) : Hthm_sig =

struct
  structure Report = Report;
  structure Hterm = Hterm;
  structure HtermSet = HtermSet;

  fun THM_ERR{function,message} =
11.2. THE IMPLEMENTATION

Exception.CHK_ERR{origin_structure = "Hthm",
origin_function = function, message = message};

11.2.1 The type hthm The type representing theorems consists of a record of two fields. The field Hyp is a set of terms representing the hypotheses. The field Concl is a term representing the conclusion.

datatype hthm = THM of {Hyp: HtermSet.set, Concl: Hterm.hterm};

11.2.2 The constructor The function mk_thm creates a theorem from a list of terms as assumptions and a term as conclusion. It checks to make sure all terms in the hypotheses and the conclusion are of boolean type.

fun mk_thm (hypst, concl) =
  if (every Hterm.is_bool_ty (concl::hypst))
    then
      let
        val hyp =
          fold (fn (i, s) => HtermSet.add(s, i)) hypst (HtermSet.empty)
        in
          (THM {Hyp=hyp, Concl=concl})
        end
      else raise (THM_ERR {function="mk_thm",
       message="terms are not all of boolean type"});

11.2.3 The destructors The functions concl and hyp extract the conclusion and the hypotheses of their argument theorem, respectively.

fun concl (THM {Concl,...}) = Concl
and hyp (THM {Hyp,...}) = Hyp

11.2.4 The comparator The theorems are equal if and only if their conclusions are identical and their sets of hypotheses are equivalent. As equal compares the elements of the sets, corresponding hypotheses must be identical. α-equivalent theorems are considered not equal by this function.

fun thm_eq (THM{Concl=con1,Hyp=hyp1}) (THM{Concl=con2,Hyp=hyp2}) =
  (con1 = con2) andalso (HtermSet.equal(hyp1, hyp2))

11.2.5 Output function The function pr_thm outputs a theorem to the default output stream in the proof file format.

local
  open Report
in
  fun pr_thm (THM {Hyp, Concl}) =
    (write_opening(Keyword.THM);
     write_list Hterm.pr_hterm (HtermSet.listItems Hyp);
     Hterm.pr_hterm Concl;
     write_closing_line())
end

11.2.6 Pretty Printer The function pp_thm is a pretty printer for theorems. It outputs its second argument to the pretty printing stream given as the first argument. The conclusion of a theorem is always prefixed by the turnstile \(\vdash\). The hypotheses are enclosed in square brackets and are printed before the turnstile if they occur. The individual hypotheses and the conclusion are printed by the term pretty printer.
local

open System.PrettyPrint;

fun with_ppstream ppstrm =
  {add_string = add_string ppstrm,
   add_break = add_break ppstrm,
   begin_block = begin_block ppstrm,
   end_block = fn () => end_block ppstrm,
   flush_ppstream = fn () => flush_ppstream ppstrm};

in

fun pp_hthm ppstrm (THM {Hyp,Concl}) =
  let
    val {add_string, add_break, begin_block, end_block, flush_ppstream} =
      with_ppstream ppstrm;
    val pp_hterm = Hterm.pp_hterm ppstrm
    and pp_qterm = Hterm.pp_qterm ppstrm
    fun pp_hyp [] = ()
      | pp_hyp terml =
          let
            fun p_hyp [] = ()
              | p_hyp [tm] = pp_qterm tm
              | p_hyp (tm::(tm1 as (x::i))) =
                  (pp_qterm tm;
                   add_string "\;"; add_break (1,1);
                   p_hyp tm1)
            in
              (begin_block CONSISTENT 0;
               add_string"["; p_hyp terml;
               add_string"]";
               end_block ());
          end
    val hyps = HtermSet.listItems Hyp
  in
    (begin_block CONSISTENT 0;
     pp_hyp hyps;
     add_break (1,1); add_string "|- ";
     pp_hterm Concl;
     end_block ())
  end

end(* local *)

end; (* functor HthmFUN *)
The Hterm module is the abstraction of HOL terms. It is created by applying the functor HtermFUN.

12.1 The specification

The signature Hterm_sig is the specification of the term structure. It contains the type structure Htype and the report generation module Report.

signature Hterm_sig =
  sig
    structure Report : Report_sig
    structure Htype : Htype_sig

12.1.1 Types
The type hterm represents HOL terms. It admits equality so two terms are considered equal if their representations are the same. There are four kinds of primitive terms: variables, constants, λ-abstractions and combinations (also known as function applications). The actual representation of different kinds of terms within the type hterm is local to this module.

eqtype hterm

12.1.2 Constructors and destructors
The following functions are provided for constructing, destructing and testing primitive terms. All constructors have prefix mk_. They all take a pair as their argument. All destructors have prefix dest_. They all return a pair. All term testers have prefix is_.

val mk_var : string * Htype.htype -> hterm
val mk_const : string * Htype.htype -> hterm
val mk_comb : hterm * hterm -> hterm
val mk_abs : hterm * hterm -> hterm

val dest_var : hterm -> string * Htype.htype
val dest_const : hterm -> string * Htype.htype
val dest_comb : hterm -> hterm * hterm
val dest_abs : hterm -> hterm * hterm

val is_var : hterm -> bool
val is_const : hterm -> bool
val is_comb : hterm -> bool
val is_abs : hterm -> bool

12.1.3 More destructors and testers
These are destructors for higher level terms, i.e., for terms of certain common structures, for instance equalities. They follow the naming convention described above.

val dest_eq : hterm -> hterm * hterm
val dest_forall : hterm -> hterm * hterm
val dest_exists : hterm -> hterm * hterm
val dest_imp : hterm -> hterm * hterm
val dest_not : hterm -> hterm
val dest_conj : hterm -> hterm * hterm
val is_eq : hterm -> bool
val is_conj : hterm -> bool

Listed below are special destructors. rev_dest_eq is the same as dest_eq except the order of the fields returned is reversed. The destructors whose names have the prefix strip_ are the repetitive version of their simple counterparts.

val rev_dest_eq : hterm -> (hterm * hterm)
val strip_comb : hterm -> (hterm * hterm list)
val strip_forall : hterm -> (hterm list * hterm)
val strip_exists : hterm -> (hterm list * hterm)
val strip_conj : hterm -> hterm list
val lhs : hterm -> hterm
val rhs : hterm -> hterm

12.1.4 Term comparison The function cmpTerm is an ordering function for terms. The expression cmpTerm tm1 tm2 evaluates to Equal if tm1 and tm2 are exactly the same. Otherwise, it is either Less or Greater depending on their kinds and names.

val cmpTerm : (hterm * hterm) -> LibBase.relation

12.1.5 Type of a term This group of functions deal with types of a term. When applied to a term tm, the function type_of returns the type of tm. The expression tyvars tm evaluates to a list of distinct type variables occuring in tm. The function tyvars1 is similar to tyvars but it takes a list of terms.

The expression type_in ty tm evaluates to true if tm or any of its subterms has the type ty.

The expression vty_occurs vty tm evaluates to true if the type variable vty occurs in any subtype of any subterm of tm. This is a combination of the functions type_in and type_in_type in Htype with the restriction that the type to be checked must be a type variable.

val type_of : hterm -> Htype.htype
val tyvars : hterm -> Htype.htype list
val tyvars1 : hterm list -> Htype.htype list
val type_in : Htype.htype -> hterm -> bool
val vty_occurs : Htype.htype -> hterm -> bool

12.1.6 α-conversion The predicate aconv takes two terms as its arguments. It returns true if they are α convertible from each other.

val aconv : hterm -> hterm -> bool

12.1.7 Type instantiation The function term_is_type_inst tests two terms to see whether one is the result of type-instantiating the other. The first argument is a list of type pairs specifying the type instantiations.

val term_is_type_inst : (Htype.htype * Htype.htype)list -> hterm -> hterm -> bool

The function term_is_tty_inst is a bounded version which takes an extra argument of a list of variable pairs. This list specifies the variables renamed when the term is instantiated.

val term_is_tty_inst : (Htype.htype * Htype.htype)list -> (hterm * hterm) list -> hterm -> hterm -> bool
The function `term_inst_renames` returns the list of variable pairs which are renamed in the instantiation.

```ml
val term_inst_renames : (Htype.htype * Htype.htype)list -> hterm -> hterm -> (hterm * hterm)list
```

The function `term_inst_chk` takes an extra list of variables. These are assumed to be the free variables in the assumptions of a theorem. Any variables occurring in this list should not be renamed. The new variables should not be the same as any of these variables.

```ml
val term_inst_chk : (Htype.htype * Htype.htype)list -> (hterm)list -> hterm -> hterm -> bool
```

### 12.1.8 Free and bound variables

The function `frees` returns a list of the free variables in the input term. The function `frees1` is similar but takes a list of terms. The resulting list is treated as a set, i.e., every free variable appears only once in the list if it occurs one or more times in the term.

The expression `frees_in tm1 tm2` evaluates to true if `tm1` occurs free in `tm2` where `tm1` must be a variable. (Note that this is different from the function having the same name in HOL88.)

The expression `variant ul x` returns a variable which is a variant of `x` and is different from any of the variables in the list `ul`. The variant is created by appending a prime (') to the name of `x`.

```ml
val frees : hterm -> hterm list
val frees1 : hterm list -> hterm list
val frees_in : hterm -> hterm -> bool
val variant : hterm list -> hterm -> hterm
```

### 12.1.9 Substitutions

A substitution of a term `tm[t_1, \ldots , t_n]` is a term

\[ tm[t'_1, \ldots , t'_n / t_1, \ldots , t_n] \]

in which free occurrences of `t_i` are replaced by `t'_i` for `i = 1, \ldots , n`. The type of `t'_i` must be the same as `t_i`. If free occurrences of variables in `t'_i` will be captured, the corresponding bound variables will be renamed. A substitution can be specified as a pair `(t'_i, t_i)`.

A list of substitutions `[(t'_1, t_1), \ldots , (t'_n, t_n)]` is consistent if and only if

1. all the `t'_i`'s are distinct, or
2. if `t_i = t_j`, then `t'_i = t'_j`.

The function `term_subst_chk` checks the correctness of a substitution. It takes four arguments in the following order:

- a *substitution list* whose elements are triples, `((t_i, x_i), v_i)`, specifying the substitutions;
- a *template* which specifies the positions where substitutions are performed;
- a *substituted term* which is the result of the substitutions;
- an *original term* which is to be substituted.

Positions of substitutions are indicated in the template by the occurrences of dummy variable `v_i`. The corresponding positions of the original term should be free occurrences of the variable `x_i`. They are replaced by the term `t_i` in the substituted term. The function `term_subst_chk` returns
true if the substituted term is a consistent substitution according to the above description. Otherwise, it returns false.

val termsubstchk : (hterm * hterm) * hterm list -> hterm -> hterm -> hterm -> bool

The function mk_subs_temp is used in the check module. They generate a substitution template.

val mksubsTemp : (hterm * hterm) * hterm list -> hterm
val mksubsOccuTemp : hterm -> hterm -> hterm -> int -> int list
-> (hterm * int * int list)

12.1.10 Generating unique variables The function mk_gen_var creates a unique variable every time it is called. This is used when the checker needs an internal general variable, e.g., for renaming.

val mk_gen_var : Htype.hterm -> hterm

12.1.11 Special types Since it is frequently required to check whether a term has boolean type, the function is_bool ty is defined for this purpose. Similarly, the function is_num ty returns true if applied to a term whose type is num.

val is_boolTy : hterm -> bool
val is_numTy : hterm -> bool

12.1.12 Output function This function prints a term to the default output stream in the prf format.

val pr_hterm : hterm -> unit

12.1.13 Pretty Printer The function pp_qterm is a pretty printer for quoted terms. The function pp_hterm is a pretty printer for terms without being enclosed in double quotes. They use the system pretty-printer module to print terms.

val pp_qterm : System.PrettyPrint.pppstream -> hterm -> unit
val pp_hterm : System.PrettyPrint.pppstream -> hterm -> unit

end; (* signature Hterm_sig *)

12.2 The implementation

The functor HtermFUN takes three structures: Report the report generation module, Htype the abstraction of HOL types and their operations and HtypeCmp the ordering of HOL types. The last two have to share the type representing HOL types. The Report structure should be the same as the Report structure in the type module.

functor HtermFUN (structure Report : Report_sig
    and Htype: Htype_sig and HtypeCmp: ORD_KEY
        sharing type Htype.hterm = HtypeCmp.ord_key
        and Report = Htype.Report) : Hterm_sig =

struct
    structure Report = Report;
    structure Htype = Htype;
    structure HtypeCmp = HtypeCmp;

    val debug = ref 0;
    fun write_out s = (output(std_err, s); flush_out std_err);
fun init () = (
    debug := Debug.get_debug "Hterm";
    ()
);

12.2.1 Type for HOL terms  The type representing HOL terms is hterm. In addition to the
four kinds of primitive terms, an extra kind of term BVAR is allowed. It represents the bound
variable in an abstraction. This follows the de Bruijn’s name-free representation [3]. The integer
in a BVAR is the depth (or level) of λ’s between the bound variable and its binder.

type atom = {Name: string, Ty: Htype.htype};

datatype hterm = BVAR of int
    | VAR of atom
    | CONST of atom
    | COMB of {Rator: hterm, Rand: hterm}
    | ABS of {Bvar: hterm, Body: hterm};

12.2.2 Exception  This function generates exception for this module.

fun TERM_ERR{function, message} =
    Exception.CK_ERR{origin_structure = "Hterm",
    origin_function = function, message = message};

12.2.3 Ordering function  The function cmpTerm compares two terms. By convention,
variables are Less than constants, which are in turn Less than function applications, and
which are in turn Less than λ-abstraction. If two terms are both variables, or constants, their
names are compared in alphabetical order. If two terms are both function applications, the
operators are compared first. If they are the same, the operands are compared. If two terms are
both λ-abstractions, the bound variables are compared first, then the bodies. This function is
required to create the HtermComp structure which is used by the ordered-set of terms structure.

local open LibBase

    fun cmpTerm (tm1, tm2) =
        case tm1 of
            (BVAR n1) =>
                (case tm2 of
                    (BVAR n2) =>
                        if (n1 = n2) then Equal
                        else if (n1 < n2) then Less else Greater
                    | _ => Less)
                | (VAR {Name=name, Ty=ty}) =>
                    (case tm2 of
                        (VAR {Name=name2, Ty=ty2}) =>
                            if (name < name2) then Less
                            else if (name > name2) then Greater
                            else Htype_cmpType (ty, ty2)
                        | _ => Less)
                | (CONST {Name=name, Ty=ty}) =>
                    (case tm2 of
                        (VAR _) => Greater
                        | (CONST {Name=name2, Ty=ty2}) =>
                            if (name < name2) then Less
                            else if (name > name2) then Greater
                        | _ => Less)
        | (COMB of {Rator: tm1, Rand: tm2}) =>
            cmpTerm tm1 tm2
        | (ABS of {Bvar: tm1, Body: tm2}) =>
            if (cmpTerm tm1 tm2) then Greater
            else AbsLess tm1 tm2
        | (CONJ of {Left: tm1, Right: tm2}) =>
            if (cmpTerm tm1 tm2) then Greater
            else ConjLess tm1 tm2
        | (DISJ of {Left: tm1, Right: tm2}) =>
            if (cmpTerm tm1 tm2) then Greater
            else DisjLess tm1 tm2
        | (NOT of tm) =>
            cmpTerm tm tm
        | (IF of {Pred: tm1, Cond: tm2, Then: tm3, Else: tm4}) =>
            if (cmpTerm tm1 tm2) then Greater
            else IfLess tm1 tm2 tm3 tm4
        | _ => NonTermLess tm1 tm2

in
else Htype.cmpType (ty,ty2)
  | _ => Less
| (COMB {Rator=rat, Rand=rnd}) =>
  (case tm2 of
   (ABS _) => Less
   | (COMB {Rator=rat2, Rand=rnd2}) =>
     let val c = cmpTerm(rat,rat2)
     in
     if not(c = Equal) then c else cmpTerm(rnd,rnd2)
     end
   | _ => Greater)
| (ABS {Bvar=var, Body=body}) =>
  (case tm2 of
   (ABS {Bvar=var2, Body=body2}) =>
     let val c = cmpTerm(var,var2)
     in
     if not(c = Equal) then c else cmpTerm(body,body2)
     end
   | _ => Greater)
end ;

12.2.4 Term constructors The term constructors create terms like their counterparts in HOL, namely taking the appropriate arguments and returning a term. A special case is the \( \lambda \) term constructor since de Bruijn's name-free representation is used. The function mk_bind binds the variable in the body of an abstraction. It is called when a \( \lambda \) term is created. It recursively descends into the body of the abstraction to change the free occurrences of the variable into BVARS. A variable is considered to be the same as the bound variable if its name and its type are identical to those of the bound variable. The integer argument \( n \) is incremented when an abstraction is entered.

fun mk_var (name, ty) = VAR {Name = name, Ty = ty};
fun mk_const (name, ty) = CONST {Name = name, Ty = ty};
fun mk_comb (rator, rand) = COMB {Rator = rator, Rand = rand};

fun mk_bind (n, (bv as VAR{Name=bvname,Ty=bvty}),
  (v as VAR{Name=name,Ty=ty})) =
  if ((bvname = name) andalso (bvty = ty)) then (BVAR n) else v
  | mk_bind (n, (bv as VAR{Name=bvname,Ty=bvty}),
    (COMB{Rator=rator,Rand=rand})) =
    (COMB{Rator=mk_bind(n,bv,rator), Rand=mk_bind(n,bv,rand)})
  | mk_bind (n, (bv as VAR{Name=bvname,Ty=bvty}),
    (ABS {Bvar=v,Body=body})) =
    (ABS {Bvar=v,Body=mk_bind((n+1),bv,Body)})
  | mk_bind (n, (bv as VAR{Name=bvname,Ty=bvty}), tm as _) = tm
  | mk_bind (n,_,_) =
    raise (TERM_ERR {function="mk_bind",
      message="not a variable"});

fun mk_abs (bv as VAR _, body) =
  ABS {Bvar = bv, Body = (mk_bind(0,bv,body))}
  | mk_abs _ = raise TERM_ERR {function = "mk_abs",
      message = "first term not a variable"};
12.2. THE IMPLEMENTATION

12.2.5 System generated variables The names of all system generated unique variables begin with the string gv% and are followed by a number. The variable gen_var_count keeps a count of the number of variable generated.

```ml
val gen_var_prefix = "gv%"
and gen_var_count = ref 0
fun mk_gen_var ty =
  (gen_var_count := (!gen_var_count) + 1;
   mk_var((gen_var_prefix ^ (makestring (!gen_var_count))), ty))
```

12.2.6 Term destructors The function dest_bind performs the inverse process of mk_bind. It is called by dest_abs. It recursively descends into the body of the abstraction and converts the bound variable into a free variable. The integer argument n indicates the depth of the nested As. If it encounters a bound variable of the same level, it changes it to free variable.

```ml
fun dest_var (VAR {Name=s, Ty=ty}) = (s,ty)
  | dest_var _ = raise TERM_ERR {function = "dest_var",
                                message = "not a variable"};

fun dest_const (CONST {Name=s, Ty=ty}) = (s,ty)
  | dest_const _ = raise TERM_ERR {function = "dest_const",
                                 message = "not a constant"};

fun dest_comb (COMB {Rator=ra,Rand=rnd}) = (ra,rnd)
  | dest_comb _ = raise TERM_ERR {function = "dest_comb",
                                  message = "not an application"};

fun dest_bind (n, (bv as VAR{Name=bvname,Ty=bvty}),
               (BVAR n')) =
  if (n = n') then bv else (BVAR n')
  | dest_bind (n, (bv as VAR{Name=bvname,Ty=bvty}),
                (COMB{Rator=rator,Rand=rand})) =
    (COMB{Rator=(dest_bind(n,bv,rator)), Rand=(dest_bind(n,bv,rand))})
  | dest_bind (n, (bv as VAR{Name=bvname,Ty=bvty}),
               (ABS {Bvar=v,Body=body})) =
    (ABS {Bvar=v,Body=(dest_bind(n+1),bv, body)})
  | dest_bind (n, (bv as VAR{Name=bvname,Ty=bvty}), tm as _) = tm
  | dest_bind (n,_,_) =
    raise TERM_ERR {function="dest_bind",
                    message="not a variable");

fun dest_abs (ABS {Bvar=v, Body=b}) = (v, (dest_bind(0,v,b)))
  | dest_abs _ = raise TERM_ERR {function = "dest_abs",
                                message = "not an abstraction"};
```

12.2.7 Term testers The primitive term testers are implemented using the primitive term destructors in the previous section.

```ml
val is_var = can dest_var;
val is_const = can dest_const;
val is_comb = can dest_comb;
val is_abs = can dest_abs;
```

12.2.8 More destructors and testers There are also destructors for special terms, such as equations, negations, universal and existential quantifiers, implications and conjunctions.

```ml
fun dest_eq (COMB {Rator=(COMB {Rator=eq,Rand=lhs}), Rand=rhs}) =
```
let
  val (eqname,ety) = dest_const eq
  handle CHK_ERR _ => ("",(Htype.mk_var_type""))
in
  if (eqname = "=") then (lhs,rhs)
  else raise (TERM_ERR {function="dest_eq",
                        message="not an equality"})
end

| dest_eq _ =
  raise (TERM_ERR {function="dest_eq",
                   message="not an equality"});

fun rev_dest_eq tm =
  let val (l,r) = dest_eq tm in (r,l) end
and lhs tm =
  let val (l,r) = dest_eq tm in l end
and rhs tm =
  let val (l,r) = dest_eq tm in r end;

val is_eq = can dest_eq;

fun dest_forall (COMB {Rator=(CONST u), Rand=abs as (ABS {Bvar,Body})}) =
  if ((#Name u) = "!") then dest_abs abs
  else raise (TERM_ERR {function="dest_forall",
                        message="Not a universal quantification"})
| dest_forall _ =
  raise (TERM_ERR {function="dest_forall",
                   message="Not a universal quantification"});

fun dest_exists (COMB {Rator=(CONST u), Rand=abs as (ABS {Bvar,Body})}) =
  if ((#Name u) = "?") then dest_abs abs
  else raise (TERM_ERR {function="dest_exists",
                        message="Not an existential quantification"})
| dest_exists _ =
  raise (TERM_ERR {function="dest_exists",
                   message="Not an existential quantification"});

fun dest_imp (COMB {Rator=(COMB {Rator=eq,Rand=ante}), Rand=conc}) =
  let
    val (eqname,ety) = dest_const eq
    handle CHK_ERR _ => ("",(Htype.mk_var_type""))
in
    if (eqname = "=>") then (ante,conc)
    else raise (TERM_ERR {function="dest_imp",
                          message="not an implication"})
end
| dest_imp _ =
  raise (TERM_ERR {function="dest_imp",
                   message="not an implication"});

fun dest_not (COMB {Rator=(CONST{Name=name,...}), Rand=body}) =

if (name = "") then body
else raise (TERM_ERR {function="dest_not",
message="not a negation"})
| dest_not _ =
raise (TERM_ERR {function="dest_not",
message="not a negation"});

local
fun str_comb (tm,l) =
  if is_comb tm then
    let val (rator,rand) = dest_comb tm
    in
      str_comb (rator, (rand::l))
    end
  else (tm,l)
  in
    fun strip_comb tm = str_comb (tm,[]) end;

local
fun it_right f (l,tm) =
  if (can f tm) then
    let val (fst,snd) = f tm
    in
      it_right f ((fst::l), snd)
    end
  else (l,tm)
  in
    fun strip_forall tm =
    let val (vs,b) = it_right dest_forall ([],tm) in ((rev vs),b) end
    fun strip_exists tm =
    let val (vs,b) = it_right dest_exists ([],tm) in ((rev vs),b) end
    in
    end;

fun dest_conj (COMB {Rator=(COMB {Rator=conj,Rand=lhs}), Rand=rhs}) =
  let
    val (cname,cty) = dest_const conj
    handle CHK_ERR _ => ("",(Htype.mk_vartype"*"))
in
  if (cname = "/\") then (lhs,rhs)
  else raise (TERM_ERR {function="dest_conj",
message="not a conjunction"})
| dest_conj _ =
raise (TERM_ERR {function="dest_conj",
message="not a conjunction"})

val is_conj = can dest_conj

fun strip_conj term =
  let
fun stripc tm lrl =  
  if (is_conj tm) then 
    let 
      val (l,r) = dest_conj tm 
    in 
      stripc l (stripc r lrl) 
    end 
  else (tm :: lrl) 
  end 
  in 
  stripc term [] 
end

12.2.9 Free variables The task of finding free occurrences of variables is made simple by the name-free representation. All we need to do is to recursively descend into the subterms to pick up all the VAR nodes since bound occurrences are represented by BVAR.

The local function freevars does the actual job of finding free variables. Its second argument is used to cumulate the free variables found in the term. This improves the performance.

local 
  fun freevars ((tm as VAR _), l) = tm :: l 
  | freevars ((CONB{Rator, Rand}), l) = 
    (freevars (Rator, (freevars (Rand, l)))) 
  | freevars ((ABS{Body,...}), l) = freevars (Body, l) 
  | freevars (_, l) = l 
  in 
  fun frees tm = setify (freevars (tm, [])) 
  and freevl tms = setify (fold freevars tms []) 
  and free_in (v as VAR _) tm = mem v (freevars (tm, [])) 
  | free_in _ tm = 
    raise (TERM_ERR{function="freein", 
                      message="argument not a variable"})
end;

12.2.10 Variant of variable This function generates a variant of the given variable by appending prime(') on to it. The new variant is tested against the list vl. More primes are added if required.

    fun variant [] x = x 
    | variant vl (x as VAR{Name,Ty}) = 
      if (mem x vl) then variant vl (VAR{Name="\""",Ty=Ty}) 
      else x 
    | variant vl _ = raise (TERM_ERR {function="variant", 
                                          message="not a variable"});

12.2.11 Type of a term The types of variables and constants are carried in them explicitly. The types of combinations and abstractions are worked out dynamically. The operator of a combination must be of a function type. If it is not, dest_functype will fail. The type of the operand must be the same as the domain of the operator. An exception is raised if this is not the case. The type of an abstraction is guaranteed to be a function type. Its domain is the type of the bound variable, and its range is the type of the body. dest.Abs is called to convert the bound occurrences of the variable into free occurrences before the type of the body can be worked out.

    fun type_of (VAR {Ty,...}) = Ty
12.2. THE IMPLEMENTATION

| type_of (CONST {Ty,...}) = Ty
| type_of (COMB {Rator,Rand}) = let
|          val (randty,rty) = (Htype.dest_funtype (type_of Rator))
in
|          if (randty = type_of Rand) then rty
|          else raise (TERM_ERR {function="type_of",
|                          message="incompatible types in application")}
end
| type_of (tm as (ABS _)) = let val (bv, body) = dest_abs tm
|          in Htype.mk_funtype ((type_of bv), (type_of body))
end
| type_of _ = raise (TERM_ERR {function="type_of",
|                          message="type_of bound variable"));

12.2.12 Special types testers These functions return true if the term is of type bool or num, respectively.

    fun is_bool_ty tm = ((type_of tm) = Htype.bool_ty)
    and is_num_ty tm = ((type_of tm) = Htype.num_ty)

12.2.13 Type variables in term The function tyvars, when applied to a term, returns a list of the distinct type variables occurring in the term. These include the type variables occurring in any subterm of the input term. The resulting list is like a set, i.e., no repeated variables.

    The function tyvars1 is similar but taking a list of terms.

    local
        fun ty_vars ((VAR {Ty,...}), v1) = union (Htype.type_tyvars Ty) v1
        | ty_vars ((CONST {Ty,...}), v1) = union (Htype.type_tyvars Ty) v1
        | ty_vars ((ABS {Bvar,Body}), v1) = union (Htype.type_tyvars (type_of Bvar)) (ty_vars(Body, v1))
        | ty_vars ((COMB {Rator,Rand}), v1) = ty_vars(Rand, (ty_vars(Rator, v1)))
        | ty_vars (_, v1) = v1
    in
        fun tyvars tm = ty-vars(tm, [])
        and tyvars1 tml = fold ty_vars tml []
    end;

12.2.14 Type in a term The function type_in returns true if its first argument ty is the type of the term or the type of any of its proper subterm. If the term is a combination, its type is the range of the operator. We find the types of the operator and the operand separately. If ty is equal to the range of the operator, we require the type of the operand to be the same as the domain of the operator. We also find the types of the bound variable and the body separately if the term is an abstraction.

    fun type_in ty (VAR {Ty,...}) = (ty = Ty)
    | type_in ty (CONST {Ty,...}) = (ty = Ty)
    | type_in ty (COMB {Rator, Rand}) = let val (aty,xty) = Htype.dest_funtype (type_of Rator)
                   and ranty = type_of Rand
                   in
((ty = rty) andalso (rty = rnty)) orelse
(type_in ty Rator) orelse (type_in ty Rand)
end
| type_in ty (tm as ABS _) =
let val (bv, body) = dest_abs tm
  val bty = type_of body and vty = type_of bv
in
  (ty = Htype.mk_funttype(vty, bty)) orelse
  (ty = vty) orelse (ty = bty) orelse
  (type_in ty body)
end
| type_in ty _ = false;

12.2.15 Occurrence of type variables The expression \texttt{vty\_occurs vty\_tm} evaluates to true if the type variable \texttt{vty} occurs in any subtype of any subterm of \texttt{tm}. This is implemented by first finding all type variables occurring in the type of the term or the types of any subterms using \texttt{tyvars}; and then checking whether \texttt{vty} is one of these type variables.

\texttt{fun vty\_occurs vty\_tm =}
  \texttt{if (Htype.is_vartype vty) then mem vty (tyvars tm)}
  \texttt{else raise (TERM\_ERR \{function=\"vty\_occurs\",}
                   \texttt{message=\"not a type variable\")
      \}}

12.2.16 Checking type instantiation The function \texttt{term\_is\_type\_inst} calls the local function \texttt{is\_ty\_inst} to perform the type instantiation checking. The local function takes, as its arguments, two terms \texttt{tm1} and \texttt{tm2}, a list of type pairs \texttt{tytys} and a list of pairs variables \texttt{nl}. It checks to see if \texttt{tm1} is an instance of \texttt{tm2} according to \texttt{tytys}. The pairs of variables in \texttt{nl} are the renaming found in the checking. The function will raise an exception if inconsistent renaming is found, i.e., two occurrences of a variable are renamed differently or two distinct variables become identical.

The checking uses the fact that the structure of term trees of \texttt{tm1} and \texttt{tm2} must be the same if one is a type instantiation of the other, i.e., the corresponding nodes are of the same kind. The complication due to renaming only occurs at the nodes which are free variables. There are five different cases corresponding to the five kinds of primitive terms:

- For bound variable nodes, they must be at the same level of nested abstraction.
- For constant nodes, they must be the same constant and their types must be a consistent instantiation according to the list \texttt{tytys}.
- For free variable nodes, their names must be the same and their types must be a consistent instantiation according to the list \texttt{tytys} if renaming does not occur. Otherwise, the renaming is checked against the list \texttt{nl}. This is performed by the local function \texttt{chk\_rename}. It takes, as arguments, the pair of variables \texttt{\langle v1, v2 \rangle} which are the current nodes, and the list \texttt{nl} which contains all the renaming found so far. If \texttt{v2} has not been seen before, then \texttt{v1} must not have been seen either. Then, the pair \texttt{\langle v1, v2 \rangle} is added into \texttt{nl}. If \texttt{v2} is not seen before but \texttt{v1} is found in the list \texttt{nl}, we have a renaming that makes two variables identical. The function returns \texttt{false}. If \texttt{v2} is found in \texttt{nl}, then \texttt{v1} must be the same as the first field of the pair found in the list, otherwise, we find an inconsistent renaming. In the latter case, the function returns \texttt{false}.
- For \texttt{CONB} nodes, we check the rator (left) subtree first. If it is \texttt{OK}, we check the rand (right) subtree.
• For ABS nodes, the type \( ty_1 \) of a bound variable in \( tm_1 \) must be a consistent instantiation of \( ty_2 \). This variable may be renamed, but it does not matter what it is renamed to as long as the name is not the name of any free variable in the body. The renaming does not need to be recorded in \( nl \).

```
local
  open Htype ListUtil
  fun chk_rename (v1,v2) nl =
    let
      val found1 = ListUtil.findOne (fn (v,u) => (v2 = u)) nl
    in
      case found1 of
        NONE =>
          let val found2 = ListUtil.findOne (fn (v,u) => (v1 = v)) nl
          in
            case found2 of
              NONE => (true, ((v1,v2)::nl))
              SOME _ => (* renaming makes two variables identical *)
                (false, nl)
            end
        SOME(v,u) => (* variable already seen *)
              ((v1 = v), nl)
    end
  end

fun is_ty_inst (BVAR n1) (BVAR n2) tytyl nl =
  ((n1 = n2), tytyl, nl)
| is_ty_inst (CONST{Name=n1,Ty=ty1}) (CONST{Name=n2,Ty=ty2}) tytyl nl =
  (((n1 = n2) andalso (is_ty_inst tytyl ty1 ty2)), tytyl, nl)
| is_ty_inst (v1 as VAR{Name=n1,Ty=ty1}) (v2 as VAR{Name=n2,Ty=ty2}) tytyl nl =
  if (is_tytyl_inst tytyl ty1 ty2) then
    let val (nameok, nl') = chk_rename (v1,v2) nl
    in
      (nameok, tytyl, nl')
    end
  else (false, tytyl, nl)
| is_ty_inst (COMB{Rator=rat1,Rand=ran1})
  (COMB{Rator=rat2,Rand=ran2}) tytyl nl =
  let val (rat,tytyl',nl') = is_ty_inst rat1 rat2 tytyl nl
  in
    if rat then is_ty_inst ran1 ran2 tytyl' nl'
    else (false, tytyl, nl)
  end
| is_ty_inst (ABS{Bvar=(v1 as VAR{Name=lv1,Ty=ty1}),Body=bo1})
  (ABS{Bvar=(VAR{Name=lv2,Ty=ty2}),Body=bo2}) tytyl nl =
  if ((is_tytyl_inst tytyl ty1 ty2) andalso
    ((lv1 = lv2) orelse (not (free_in v1 bo1))))
  then (is_ty_inst bo1 bo2 tytyl nl)
  else (false, tytyl, nl)
| is_ty_inst tm1 tm2 tytyl nl =
  raise (TERM_ERR{function="is_ty_inst"},
```
Here comes the type instantiation checking function and its variants.

```haskell
fun term_is_type_inst tytyl tml tm2 =
  let
    val (OK, ttl, nnl) = is_ty_inst tml tm2 tytyl []
  in
    OK
  end

fun term_is_tyty_inst tytyl n1 tml tm2 =
  let
    val (OK, ttl, nnl) = is_ty_inst tml tm2 tytyl []
  in
    OK andalso ((length n1) = (length nnl))
  end

fun term_inst_renames tytyl tml tm2 =
  let
    val (OK, ttl, nnl) = is_ty_inst tml tm2 tytyl []
    ListUtil.remove (fn (v, u) => (v = u)) nnl
  end

fun term_inst_chk tytyl v1 tml tm2 =
  let
    val (OK, ttl, nnl) = is_ty_inst tml tm2 tytyl []
    val (n1, v1') = unzip(remove (fn (v, u) => (v = u)) nnl)
  in
    (OK andalso (null(intersect v1 v1')) andalso
     (null(intersect v1 n1)))
  end

end
```

12.2.17  \(\alpha\)-conversion  Two terms are \(\alpha\) equivalent if and only if

1. they are of the same kind of term; and

2. all corresponding subterms are \(\alpha\) equivalent. This can be divided into the following cases:

- if they are combinations the operators and the operands are \(\alpha\) equivalent, respect-ively;
- if they are abstractions, the bound variables must be of the same type and the bodies must be \(\alpha\) equivalent;
- otherwise they must be the same constant, or the same free variable (the same in name and type), or they must both be bound variables associated with the bounder of the same level, i.e., with the same depth index number.

```haskell
fun aconv (COMB{Rator=M1, Rand=M2}) (COMB{Rator=N1, Rand=N2}) =
aconv M1 M1 andalso aconv M2 N2
| aconv (ABS{Bvar=VAR(Ty=ty1,...), Body=body1})
  (ABS{Bvar=VAR(Ty=ty2,...), Body=body2}) =
  (ty1=ty2) andalso (aconv body1 body2)
| aconv tml tm2 = (tml=tm2);
```
12.2.18 Pretty Printer The pretty printer prints a term to the pretty printing stream given as the first argument. No special syntactic treatment is performed on the term since the checker knows nothing about special syntactic status of constants, such as infix or binder status. Although it is entirely primitive, a term printed by the pretty printer is a legal HOL input term, i.e., it is accepted by the HOL system parser. Therefore, one can cut a pretty printed term from the checker and paste it to HOL. The latter will print the term in the proper infix and binder form.

```ml
local
  open System.PrettyPrint;
  fun with_ppstream ppstrm =
    {add_string = add_string ppstrm,
     add_break = add_break ppstrm,
     begin_block = begin_block ppstrm,
     end_block = fn () => end_block ppstrm,
     flush_ppstream =fn () => flush_ppstream ppstrm};

in
  fun pp_hterm ppstrm tm =
    let
      val {add_string, add_break,
           begin_block, end_block, flush_ppstream} =
        with_ppstream ppstrm;
      val pp_htype = Htype.pp_htype ppstrm;
      fun pp_term (VAR {Name,...}) = add_string Name
          | pp_term (CONST {Name,...}) =
            (if not(CType.isAlphaNum(Name,0))
             then add_string"$" else ()
             add_string Name)
          | pp_term (COMB {Rator,Rand}) =
            (begin_block CONSISTENT 0;
             add_string "(";
             pp_term Rator; add_break (1,1);
             pp_term Rand;
             add_string ")";
             end_block ())
          | pp_term (ABS {Bvar,Body}) =
            let val btm = dest_bind(0,Bvar,Body)
            in
              (begin_block CONSISTENT 0;
               add_string "\\"
               pp_term Bvar;
              add_string ","; add_break(1,2);
              pp_term btm;
              add_string"
               end_block ()
              end
          | pp_term _ =
            raise (TERM_ERR {function="pp_term",
                             message="Unknown term"})
    in
      (begin_block CONSISTENT 0;
     pp_term tm; add_break(1,1));
```
pp_htype (type_of tm);
end_block () handle e => (flush_ppstream();
clear_ppstream ppstrm; raise e)
end

and pp_qterm ppstrm tm =
let
val {add_string, add_break, begin_block, end_block, flush_ppstream} =
  with_ppstream ppstrm;
in
(begin_block CONSISTENT 0;
add_string "\n";
pp_hterm ppstrm tm;
add_string "\n";
end_block () handle e => (flush_ppstream();
clear_ppstream ppstrm; raise e)
end

end(* local *)

12.2.19 Substitutions  The local function sublst_chk checks the consistency of the substitution list subl which is in the following form: \([...,((t_i, t'_i), v_i),...]\). The \(v_i\)'s must all be variables. Their types must be the same as the corresponding \(t_i\) and \(t'_i\).

local

open ListUtil

fun sublst_chk [] = true
  | sublst_chk (((t, t'), (VAR(Ty,...))):l) =
      (Ty = (type_of t) andalso (Ty = (type_of t')) andalso (sublst_chk l))
  | sublst_chk (((t, t'), _):l) =
      raise (TERM_ERR(function="sublst_chk",
                       message="substitution list contains non-variable");)

The local function mk_sublst converts the substitution list subl into a pair of lists of pairs. The second fields of the list elements are the dummy variables \(v_i\). The first fields of the first list are the terms or variables to be replaced \(t_i\). The first fields of the second list are the new terms \(t'_i\).

and mk_sublst ttvl =
  let val (l,vl) = unzip ttvl
    val (tl, tl') = unzip l
  in
    (zip(tl',vl), zip(tl,vl))
  end

The local function is_tm_subst checks two terms \(tm_1\) and \(tm_2\) to see if the first is a substitution of the second according to the list sublst. The elements of the list are pairs of the form \((tm_i, v_i)\) where the \(v_i\)'s are variables and the \(tm_i\)'s are terms to replace the free occurrences of the corresponding \(v_i\) in \(tm_2\).

The term trees should be the same except where substitutions occur and bound variables are renamed. Substitutions only occur at the VAR nodes of \(tm_2\). For any VAR nodes \(v\) in \(tm_2\), if there is a pair \((t, v)\) in the list sublst, then the corresponding node in \(tm_1\) must be the same
as \( t \). For bound variables and constants, the corresponding nodes in the two terms must be the same. For function applications, the left subtrees are checked, then the right subtrees. For abstractions, only the bodies are checked against each other. The bound variable may have been renamed. Their names can be different, but their types should be identical.

```fun is_tm_subst tm1 (v as VAR{Name,Ty}) sublist = 
  let
    val found = findOne (fn (t,v') => (v = v')) sublist in
    case found of
      NONE => (tm1 = v)
    | SOME (t,v') => (aconv tm1 t)
  end
  | is_tm_subst (BVAR n1) (BVAR n2) sublist = (n1 = n2)
  | is_tm_subst (c1 as CONST{...}) (c2 as CONST{...}) sublist = (c1 = c2)
  | is_tm_subst (COMB{Rator=rat1,Rand=ran1}) (COMB{Rator=rat2,Rand=ran2}) sublist =
    (is_tm_subst rat1 rat2 sublist) andalso
    (is_tm_subst ran1 ran2 sublist)
  | is_tm_subst (ABS{Bvar=(VAR{Ty=ty1,...}),Body=bo1})
    (ABS{Bvar=(VAR{Ty=ty2,...}),Body=bo2}) sublist =
    (ty1 = ty2) andalso (is_tm_subst bo1 bo2 sublist)
  | is_tm_subst tm1 tm2 sublist = false
```

The user function `term_subst_chk` returns `true` if \( tm_1 \) is a substitution of \( tm_2 \) according to the template \( tmpl \) and the substitution list \( ttvl \). It first verifies the substitution list by calling the local function `sublst_chk`. It then calls `is_tm_subst` to check whether \( tm_1 \) is a substitution of the template and also \( tm_2 \) is a substitution of the template.

```in
fun term_subst_chk ttvl tmpl tm1 tm2 = 
  if (sublst_chk ttvl) then
    let
      val (sublst1,sublst2) = mk_sublst ttvl
      fun out_ppstream outstr n =
        System.PrettyPrint.mk_ppstream{linewidth = n, 
        flush = fn () => flush_out outstr, 
        consumer = outputc outstr} 
      val pstrm = out_ppstream std_err 78 
    in
      if (is_tm_subst tm1 tmpl sublst1) 
        then if (is_tm_subst tm2 tmpl sublst2) 
          then true 
          else raise (TERM_ERR{function="term_subst_chk", 
                        message="second check failed"})
        else (pp_qterm pstrm (#1(hd sublst1)); 
        pp_qterm pstrm (#2(hd sublst1)); 
        pp_qterm pstrm tm1; 
        pp_qterm pstrm tmpl; 
        System.PrettyPrint.flush_ppstream pstrm; 
        raise (TERM_ERR{function="term_subst_chk", 
                        message="first check failed"})
```
end
else false
end

12.2.20 Create template for substitution The function mk_subs_occu_temp builds a
template for use in the substitution checking function. It takes five arguments: \( t \) is the original
term to which the substitution is done. \( x \) is the variable to be replaced. \( v \) is a new variable
which indicates where the required substitutions are. It should not occur anywhere in \( t \). \( seen \)
is the number of times \( x \) is encountered during the left-to-right scan of \( t \). \( l \) is a list of numbers
indicating the occurrences of \( x \) in \( t \) where the substitutions are required. This list should be
sorted in ascending order.

It returns a triple. The first is the template. The second is the number of times \( x \) is found.
The third is the list of occurrences.

The algorithm of this function is to perform a preorder scan on the term \( t \) while recording
the number of times \( x \) is found. When the occurrence matches the required one in the list \( l \), \( x \)
is replaced by the new variable \( v \). This should only occur at a \texttt{VAR} node.

\[
\text{fun mk_subs_occu_temp} \ t \ x \ v \ seen \ [\] = (t, seen, [\])
| mk_subs_occu_temp (COMB \{Rator, Rand\}) \ x \ v \ seen \ l =
  let
  val (tm', s, nl) = mk_subs_occu_temp Rator \ x \ v \ seen \ l
  val (tm'', s',nl') = mk_subs_occu_temp Rand \ x \ v \ s \ nl
  in
  (COMB\{Rator=tm', Rand=tm''\}, s', nl')
end
| mk_subs_occu_temp (ABS\{Bvar, Body\}) \ x \ v \ seen \ l =
  let
  val (tm,s, nl) = mk_subs_occu_temp Body \ x \ v \ seen \ l
  in
  (ABS\{Bvar=Bvar, Body=tm\}, s, nl)
end
| mk_subs_occu_temp (t as VAR\{Name, Ty\}) \ x \ v \ seen \ l =
  if (t = x) then
    let val seen' = seen + 1
    in
      if (seen' = (hd l)) then (v, seen', (tl l))
      else (t, seen', l)
    end
  else (t, seen, l)
| mk_subs_occu_temp \t \ x \ v \ seen \ l = (t, seen, l)
\]

The function \texttt{mk_subs_temp} is similar to the previous one except it scans the term to find
all free occurrences of variables specified in the substitution list \texttt{sub_list}.

\[
\text{fun mk_subs_temp} \ [\] \ tm = \tm 
| mk_subs_temp \ sub_list \ tm =
  (case ListUtil.findOne (fn ((t,t'),v) => (t = tm)) \ sub_list of
    SOME((t,t'),v) => (v)
    NONE =>
      case \tm\ of
        (COMB\{Rand, Rator\}) =>
          let
\]


12.3. STRUCTURE HTERM_CMP

    val rand = mk_subs_temp sub_list Rand
    and rator = mk_subs_temp sub_list Rator
    in
      (COMB(Rand=rand,Rator=rator))
    end
    | (ABS{Bvar,Body}) =>
      (ABS{Bvar=Bvar,Body=mk_subs_temp sub_list Body})
    | _ => tm

12.2.21 Output function  The function pr_hterm outputs a term to the default output
stream in the prf format.

    local
      open Report
    in
      fun pr_hterm (VAR {Name,...}) = write_sitem(Keyword.VAR,Name)
      | pr_hterm (CONST {Name,...}) = write_sitem(Keyword.CONST,Name)
      | pr_hterm (COMB {Rator,Rand}) =
        (Io.write_output_string(Keyword.LP ^ Keyword.APP ^ " ");
         pr_hterm Rator; pr_hterm Rand;
         Io.write_output_string Keyword.RP)
      | pr_hterm (ABS {Bvar,Body}) =
        let
          val btm = dest_bind(0,Bvar,Body)
        in
          Io.write_output_string(Keyword.LP ^ Keyword.APP ^ " ");
          pr_hterm Bvar; pr_hterm btm;
          Io.write_output_string Keyword.RP
        end
    end;

end; (* functor Hterm *)

12.3 Structure HtermCmp

This structure encapsulates the term ordering function cmpTerm. It is used to create ordered
sets of terms.

    functor HtermCmpFUN(Hterm : Hterm_sig) : ORD_KEY =
    struct
      type ord_key = Hterm.hterm

      fun cmpKey (tm1,tm2) = Hterm.cmpTerm (tm1, tm2)
    end;
13 HOL types

The structure Htype is the abstraction for HOL types. It defines a type htype for representing HOL types. There are a number of constructors and destructors. Their names are the same as their counterpart in the HOL system if they perform the same operation.

13.1 The specification

The signature Htype_sig is the specification of the type structure.

signature Htype_sig =
  sig
      structure Report : Report_sig
  end

13.1.1 Types The type htype represents HOL types. It admits equality as two types are equal if and only if their representations are the same. There are two kind of HOL types: type variables and type operators. Although atomic types and function types are treated separately in Section 15.2 of [5], they are treated as special cases of type operators. Atomic types are type operators taking no arguments, i.e., having an arity of 0. Function types are type operators having name fun and arity of 2.

eqtype htype

13.1.2 Constructors and destructor The functions mk_type and mk_vartype return a type operator and a type variable, respectively. The string argument to these functions is the name of the type. The type list argument of mk_type is the list of argument types of the type operator.

val mk_type : {Tyargs:htype list, Tyop:string} -> htype
val mk_vartype: string -> htype

The function dest_type and dest_vartype return the constituent part(s) of the type. If an incorrect kind of type is supplied to these functions, they will raise an exception.

val dest_type : htype -> {Tyargs:htype list, Tyop:string}
val dest_vartype: htype -> string

As function types are used very often, a constructor and a destructor are provided for creating and taking apart function types. Their names are mk_functype and dest_functype, respectively. The function domain_of takes a function type and returns the type of its domain.

val mk_functype: (htype * htype) -> htype
val dest_functype : htype -> (htype * htype)
val domain_of : htype -> htype

Two predicates is_vartype and is_functype are provided for testing whether a type is a type variable or a function type respectively.

val is_vartype : htype -> bool
val is_functype : htype -> bool

128
13.1. THE SPECIFICATION

13.1.3 Type comparison The function cmpType is an ordering function for types. The expression cmpType \( t_{y1}, t_{y2} \) evaluates to Equal if \( t_{y1} \) and \( t_{y2} \) are exactly the same. Otherwise, it is either Less or Greater depending on their kinds and names. Type variables are Less than type operators. Types of the same kind are ordered according to their names. If they are of the same type operator, the order depends on their arguments.

\[
\text{val cmpType : (hype * hype) -> LibBase\_relation}
\]

13.1.4 Getting type variables The expression type\_tyvars \( t_y \) evaluates to a list of all distinct type variables occurring in the type \( t_y \). The function ty\_tyvars1 is similar except it takes a list of types.

\[
\text{val type\_tyvars: hype -> hype list}
\]

\[
\text{val type\_tyvars1: hype list -> hype list}
\]

13.1.5 Test for sub-types The expression type\_in\_type \( t_{y1}, t_{y2} \) evaluates to true if \( t_{y1} \) is equal to \( t_{y2} \) or any sub-type of it.

\[
\text{val type\_in\_type : hype -> hype -> bool}
\]

13.1.6 Special types Some types are used very frequently. Symbolic names are bound to these types for convenience.

\[
\text{val bool\_ty : hype}
\]

\[
\text{val ty\_is\_bool : hype -> bool}
\]

\[
\text{val num\_ty : hype}
\]

\[
\text{val ty\_is\_num : hype -> bool}
\]

13.1.7 Type instantiation The expression is\_type\_inst \( t_{y1}, t_{y2} \) evaluates to true if \( t_{y1} \) is an instance of \( t_{y2} \). An instantiation of a type \( \sigma \) is defined as

\[
\sigma' = \sigma[\tau_1, \ldots, \tau_n/\beta_1, \ldots, \beta_n]
\]

where \( \tau_i \) for \( i = 1, \ldots, n \) are types and \( \beta_i \) are type variables in \( \sigma \). \( \sigma' \) is the result of simultaneous substitution of \( \tau_i \) for \( \beta_i \) in \( \sigma \).

The function is\_tyty\_inst is a bounded version of is\_type\_inst. It takes an extra argument of type (hype * hype) list. This specifies the type variables to be instantiated and the corresponding types. The type instantiations must be compatible with this list.

\[
\text{val is\_type\_inst : hype -> hype -> bool}
\]

\[
\text{val is\_tyty\_inst : (hype * hype) list -> hype -> hype -> bool}
\]

When applied to two types \( t_{y1} \) and \( t_{y2} \), the function type\_inst1 returns a list of type instantiations in the form

\[
[\ldots, (\tau_i, \sigma_i), \ldots]
\]

if \( t_{y1} \) is an instance of \( t_{y2} \). Otherwise it fails. If the list is null, no instantiation is necessary to unify the two types, i.e., they are exactly the same. The first fields of the list elements \( \tau_i \) are types. The second fields \( \sigma_i \) are type variables. If \( \tau_i \) for all \( i \) are substituted into \( t_{y2} \) for corresponding \( \sigma_i \), the result is \( t_{y1} \). This Substitution can be performed by the function inst\_type.

The expression type\_compat \( t_{y1}, t_{y2} \) evaluates to true if the types \( t_{y1} \) and \( t_{y2} \) are compatible. The definition of compatible is that they have similar structure, i.e., the shape of the tree representing the two are the same. A type variable is compatible to any type. Two type operators are compatible if and only if they are the same operator and their arguments are pair-wise compatible.

\[
\text{val type\_inst1 : hype -> hype -> (hype * hype) list}
\]
val inst_type : (htype * htype) list -> htype -> htype
val type_replace : (htype * string) list -> htype -> htype
val type_compat : htype -> htype -> bool

13.1.8 Output function The function pr_htype outputs a type to the default output stream in the proof file format prf.
val pr_htype : htype -> unit

13.1.9 Pretty Printer The function pp_htype is a pretty printer for types. It uses the system pretty-printer module to print types.
val pp_htype : System.PrettyPrint.ppstream -> htype -> unit
end;

13.2 The implementation

The functor HtypeFUN implements HOL types which are represented by the datatype htype. It has two cases:

1. TV represents type variables whose name is the argument string;
2. TO represents type operators.

The type operators consist of a record. The Tyop field is the name of the operator, and the Tyargs is the arguments of the operator which is a list of types. The length of this list is the arity of the operator. The atomic types in HOL are treated as type operators with 0-arity.

Since this representation has the same semantics as the abstract HOL type under equality, it admits equality. However, the type constructors are hidden.

functor HtypeFUN (structure Report : Report_sig) : Htype_sig =
  struct
    structure Report = Report
    datatype htype = TV of string
                | TO of {Tyop: string, Tyargs: htype list};

13.2.1 Exception This function generates exceptions for this module.

fun TYPE_ERR{function, message} =
  Exception.CHK.ERR{message = message,
    origin_function = function,
    origin_structure = "HType"};

13.2.2 cmpType This function is the ordering function for htype. The convention is that if two types are both variables, their names are compared. If two types are of the same type operator, their arguments are paired up and compared by calling the function cmp_Type_list which recursively calls cmpType. We can stop as soon as a non-equal pair is found.

open LibBase ListUtil
fun cmpType ((TV s), ty2) =
  (case ty2 of
     (TV s2) =>
      if (s < s2) then Less
      else if (s > s2) then Greater
      else Equal
     | _ => Less)
13.2. THE IMPLEMENTATION

```
| cmpType ((T0 {Tyop=tyop, Tyargs=tyargs}), ty2) =
|     (case ty2 of
|         (T0 {Tyop=tyop2, Tyargs=tyargs2}) =>
|             if (tyop < tyop2) then Less
|             else if (tyop > tyop2) then Greater
|             else cmp_type_list (zip(tyargs,tyargs2))
|         _ => Greater)
| and cmp_type_list [] = Equal
| | cmp_type_list ((ty1,ty2)::tys) =
|         let val c = cmpType (ty1,ty2)
|         in
|             if not(c = Equal) then c else cmp_type_list tys
|         end

13.2.3 type_tyvars and type_tyvars1  These functions return a list of the type variables occurring in their arguments. The local function tyvars does the actual work. The argument
vlist is used to cumulate the type variables.

local
        fun tyvars (v as (TV _)) vlist =
            if (mem v vlist) then vlist else v::vlist
        | tyvars (TO(Tyargs,...)) vlist = tyvars1 Tyargs vlist
        and tyvars1 tylist vlist = rev_itlist tyvars tylist vlist
        in
            fun type_tyvars ty = rev(tyvars ty [])
            fun type_tyvars1 tyl = rev(tyvars1 tyl [])
        end;

13.2.4 Type constructors, destructors and testers  These are the interface functions to the datatype htype. The real constructors are hidden so that the type appears as an abstract

type and its implementation may be changed without affecting the user. Note that an exception
is raised if a destructor is applied to the wrong kind of argument.

fun mk_type {Tyop, Tyargs} = T0 {Tyop=Tyop, Tyargs=Tyargs};
fun mk_vartype s = TV s;
fun mk_funtype (ratty, randty) =
            T0 {Tyop = "fun", Tyargs=[ratty, randty]};

fun dest_type (T0 r) = r
| dest_type _ = raise TYPE_ERR{function="dest_type",
                            message="Not compound type");

fun dest_vartype (TV v) = v
| dest_vartype _ = raise TYPE_ERR{function="dest_vartype",
                            message="Not type variable");

fun dest_funtype (T0 {Tyop="fun", Tyargs=[ratty, randty]}) =
            (ratty, randty)
| dest_funtype _ =
        raise (TYPE_ERR {function="dest_funtype",
                         message="Not a function type");

val is_vartype = can dest_vartype;
fun is_funtype (T0 {Tyop,Tyargs}) =
            (Tyop = "fun") andalso (length Tyargs = 2)
```
is_funttype _ = false;

fun domain_of ty = #1(dest_funttype ty);

13.2.5 type_in_type  We use cmpType to test whether two types are equal. If they are type operators, their arguments are compared.

fun type_in_type ty (TV v2) = (cmpType (ty, (TV v2)) = Equal)
  | type_in_type ty (TO {Tyargs=tyargs2,Tyop=tyop2}) =
      (case ty of
          (TV v) => exists (type_in_type ty) tyargs2
        | (TO {Tyargs,Tyop}) =>
          ((Tyop = tyop2) andalso
            ((cmp_type_list (zip(Tyargs,tyargs2))) = Equal))
        | else
          (exists (type_in_type ty) tyargs2));

13.2.6 Special type constants  Since :bool and :num types are used very frequently, we define the following constructors and predicates for them.

val bool_ty = (TO {Tyop="bool",Tyargs=[]});
val num_ty = (TO {Tyop="num",Tyargs=[]});
fun ty_is_bool (TO {Tyop="bool",Tyargs=[]}) = true
  | ty_is_bool _ = false

fun ty_is_num (TO {Tyop="num",Tyargs=[]}) = true
  | ty_is_num _ = false

13.2.7 Type instantiation  The local function inst_compat checks a type instantiation, specified as a pair (ty, vty) against an instantiation list, instl. vty should be a type variable. If no such variable appears in the list, then this is a new instantiation, it can be added to the list. If it does appear in the list, the type ty must be equal to the corresponding type in the list. Otherwise, it is incompatible and the function fails.

The local function is_ty_inst takes a pair of types (ty1, ty2) and a list of type pairs. It does the actual checking of the type instantiation. It returns true if ty1 is an instance of ty2. The list instl is used to cumulate the type instantiations found so far. It calls inst_compat to check the compatibility of instantiation. It fails if ty1 is not an instance of ty2. The function is_ty_inst1 checks a list of type pairs.

local
exception TY_INST

fun inst_compat instl (ty,vty) =
  case (findOne (fn (ty',v')=> (vty = v')) instl) of
    NONE => (true, ((ty,vty)::instl))
  | SOME (ty',v') =>
    if (ty = ty') then (true, instl)
    else raise TY_INST (* TYPE_ERR {function="inst_compat",
      message="incompatible instantiation"} *)

fun is_ty_inst (ty, v as (TV s)) instl1 =
  (inst_compat instl1 (ty,v))
  | is_ty_inst (ty2,(TO {Tyop=tyop1,Tyargs=tyargs1})) instl1 =
    (case ty2 of
      (TO {Tyop=tyop2,Tyargs=tyargs2}) =>
if not (tyop1 = tyop2)
  then raise TY_INST
    (* TYPE_ERR {function="is_ty_inst",
      message="different type op"} *)
  else is_ty_instl instl (zip(tyargs2,tyargs1))
 | _ => raise TY_INST)
      (* TYPE_ERR {function="is_ty_inst",
      message="different types"} *)
and is_ty_instl instl [] = (true, instl)
| is_ty_instl instl (tt::ttl) =
  let
    val (f,l) = is_ty_inst tt instl
  in
    if f then is_ty_instl l ttl
    else raise TY_INST
      (* TYPE_ERR {function="is_ty_instl",
      message="different type argument"} *)
  end
in
fun is_type_inst ty1 ty2 = can (is_ty_inst (ty1,ty2)) []
and is_tty_inst ttyp1 ty1 ty2 =
  let val (OK,ilst) = (is_ty_inst (ty1,ty2) ttyp1) 
    handle TY_INST => (false,ttyp1) | e => raise e
  in
    OK andalso (ilst = ttyp1)
  end
and type_instl ty1 ty2 = #2(is_ty_inst (ty1,ty2) [])
end

13.2.8 Perform type instantiation The function inst_type performs the substitution of types in its second argument. Its first argument is a type-pair list specifying the substitution. In the case that its second argument is a type variable, the function looks for the same variable in the second fields of type pairs in the substitution list. If this is found, the corresponding type is returned. Otherwise, no substitution is performed. In the case the second argument is a type operator, the function recurses down to its arguments.

fun inst_type ttyp1 (v as (TV s)) =
  let
    val typ = findOne (fn (ty, tv) => (s = dest_vartype tv)) ttyp1
  in
    case typ of
    NONE => v
    | SOME (ty, tv) => ty
  end
| inst_type ttyp1 (TV {Tyop=tyop,Tyargs=tyargs}) =
  (TV {Tyop=tyop, Tyargs=(map (inst_type ttyp1 tyargs))});

The function type_replace is the same as inst_type except it takes a (type, string)-pair as its first argument. The string is the variable name.

fun type_replace ttypnamel (v as (TV s)) =
  let
    val typ = findOne (fn (ty,tvname) => (s = tvname)) ttypnamel
  in
    case typ of
    NONE => v
    | SOME (ty, tv) => ty
  end
| type_replace ttypnamel (TV {Tyop=tyop,Tyargs=tyargs}) =
  (TV {Tyop=tyop, Tyargs=(map (type_replace ttypnamel tyargs))});
in
  case typ of
    NONE => v
  | SOME (ty, tv) => ty
end
| type_replace tytypename (TU {Tyop=tyop, Tyargs=tyargs}) =
  (TU {Tyop=tyop, Tyargs=(map (type_replace tytypename) tyargs)});

13.2.9 Compatibility of types This implements the function having the same name in
JWV’s theory. The local function ty_compat compares the structure of the two types.

  local
    fun ty_compat (ty, (TV s)) = true
    | ty_compat ((TU {Tyop=tyop1, Tyargs=tyargs1}), (TU {Tyop=tyop2, Tyargs=tyargs2})) =
      (tyop1 = tyop2) andalso (length tyargs1 = length tyargs2) andalso
      (every ty_compat (zip(tyargs1, tyargs2)))
    | ty_compat _ = false
in
  fun type_compat ty1 ty2 = ty_compat(ty1, ty2)
end

13.2.10 Output function The function pr_htype prints its argument to the default out-
put stream in the same format as in the proof file.

  local
    open Report
  in
    fun pr_htype (TV name) = write_sitem (Keyword.TYVAR, name)
    | pr_htype (TU(Tyop,Tyargs)) =
      if (null Tyargs) then write_sitem (Keyword.TYCONST, Tyop)
      else write_item_list (Keyword.TYOP, Tyop, (pr_htype, Tyargs))
  end

13.2.11 Pretty Printer The function pp_htype is a pretty printer for htypes. It out-
puts its second argument to the pretty printing stream specified by the first argument. A type is
always prefixed by a colon (:). If the type is a type variable, its name is printed. If the type is
an atomic type, the name of the type operator is printed. If a type is a type operator having
one or more arguments, its arguments are printed first, which are enclosed in parentheses and
separated by commas, then the name of the type operator are printed. The exceptions are
that if the type operator is a function type, a product type or a disjoint sum type, their name
become infix, for instance, a function type taking a num to bool is printed as :num -> bool
instead of : (num, bool)fun.

  local
    open System.PrettyPrint;
    fun with_ppstream ppstrm =
      {add_string = add_string ppstrm,
       add_break = add_break ppstrm,
       begin_block = begin_block ppstrm,
       end_block = fn () => end_block ppstrm,
       flush_ppstream = fn () => flush_ppstream ppstrm};
in
fun pp_htype ppstrm ty = 
  let
    val \{add_string, add_break, 
      begin_block, end_block, flush_ppstream\} = 
      with/ppstrm ppstrm;

  fun pp_type (TV s) = add_string s
    | pp_type (TO \{Tyop="fun",Tyargs=[ty1,ty2]\}) = 
      (begin_block CONSISTENT 0;
       add_string "(";
       pp_type ty1;
       add_string " ->"; add_break(1,1);
       pp_type ty2;
       add_string ")";
       end_block ());
    | pp_type (TO \{Tyop="prod",Tyargs=[ty1,ty2]\}) = 
      (begin_block CONSISTENT 0;
       add_string "(";
       pp_type ty1;
       add_string ","; add_break(1,1);
       pp_type ty2;
       add_string ")";
       end_block ());
    | pp_type (TO \{Tyop="sum",Tyargs=[ty1,ty2]\}) = 
      (begin_block CONSISTENT 0;
       add_string "(";
       pp_type ty1;
       add_string "+"; add_break(1,1);
       pp_type ty2;
       add_string ")";
       end_block ());
    | pp_type (TO \{Tyop,Tyargs\}) = 
      (begin_block CONSISTENT 0;
       pp_type_list Tyargs;
       add_string Tyop;
       end_block ());
  and pp_type_list [] = ()
    | pp_type_list tyl = 
      (begin_block CONSISTENT 0;
       add_string "(";
       pp_type tyl;
       add_string ")";
       end_block ());
  and pp_tyl [] = ()
    | pp_tyl [ty] = pp_type ty
    | pp_tyl (ty::tyl) = 
      (pp_type ty;
       add_string ",";
       add_break(1,2);
       pp_tyl tyl)
  in
(begin_block CONSISTENT 0;
   add_string "::");
pp_type ty;
end_block () handle e => (flush_ppstream();
   clear_ppstream ppstrm; raise e)

end

end; (* functor HtypeFUN *)

13.3 Structure HtypeCmp

This structure encapsulates the type ordering function cmpType. It is used to create ordered sets of types.

functor HtypeCmpFUN (Htype:Htype_sig) : ORD_KEY =
struct

  type ord_key = Htype.htype

  fun cmpKey (ty1,ty2) = Htype.cmpType (ty1,ty2)

end(* functor HtypeCmpFUN *);
14 Error handling

Error handling can be divided into two parts: debugging and exception handling. Debugging is for use while developing the program. Exception handling is used to trap abnormal inputs while the program is in operation. They are implemented as two separate structures.

14.1 Debugging utilities

The debugging utilities consist of a centralised database for storing debugging settings and three functions to access the data. Each individual module in the program has an entry in the debugging database. An entry is keyed by module name, and the associated data is an integer. The higher the value, the more debugging information is printed. The default value is 0 which means no debugging information is printed. This provides a flexible way of controlling the amount of information.

There are 3 levels of debugging settings. Level 0 suppresses any messages. Level 1 will show informative messages, such as the name of the inference rule of the current proof line. Level 2 provides more messages including the names of the functions.

The function set_debug sets a new debugging level for a module. It takes the module name and the new value as arguments. The function get_debug returns the current setting of a module. The function print_flags shows all the entries in the database.

However, a new value stored into the database will not take effect immediately. It is up to each individual module to decide when to update its own private copy of the setting. Generally, this happens when the module's initialisation function is called, or the structure is created by applying the functor. Otherwise, the user has no way of updating the private copies of the settings in individual modules directly.

signature Debug_sig =

  sig
    (* structure Dict : DICT *)
    val set_debug : (string * int) -> unit
    val get_debug : string -> int
    val print_flags : unit -> unit
  end

14.1.1 The implementation  The debugging setting database is implemented as a dictionary whose signature is DICT (from the SML/NJ library). It requires a structure implementing a key comparison function. The StrCmp structure is for this purpose. It has the signature ORD_KEY as specified in the library. It has a function for comparing two strings.

structure StrCmp : ORD_KEY =

  struct
    open LibBase
    type ord_key = string
    fun cmpKey (s1:ord_key,s2:ord_key) =
      if (s1 < s2) then Less
      else if (s1 = s2) then Equal else Greater;
  end;
The structure Debug implements the debugging database in its private structure Dict and the data accessing functions. An empty dictionary is created initially.

```haskell
structure Debug : Debug_sig =
  struct
    structure Dict = BinaryDict(StrCmp);

    val status_dict = ref ((Dict.mkDict()):(int) Dict.dict)
  end
```

Since the dictionary entries cannot be updated in place, to assign a new value to an entry, one has to remove the entry first and then insert a new entry with the new value.

```haskell
fun set_debug (name, value) =
    let
        val dic = #1(Dict.remove(!status_dict), name))
        handle NotFound => (!status_dict)
    in
        (status_dict := Dict.insert(dic, name, value); ());
    end
```

If no entry having the key name is found in the dictionary, the default value of 0 is returned.

```haskell
and get_debug name =
    case Dict.peek(!status_dict, name) of
    NONE => 0
    | SOME n => n
```

The function print_flags simply lists all the entries in the dictionary.

```haskell
and print_flags () =
    let
        val lst = Dict.listItems(!status_dict)
    in
        map (fn (s,n) => output(std_err,(s":"(makestring n)"\n"))) lst;
    end
end;(* struct *)
```

14.1.2 Initialisation  The database is initialised with the following values when this file is loaded.

```haskell
map Debug.set_debug
[("Io", 0), ("Htype", 0), ("Hterm", 0), ("Htm", 0), ("Henv", 1),
 ("Proof", 0), ("Parsing", 0), ("Check",2), ("Pass1", 0), ("Pass2", 1)];
```

14.2 Exception handling

This is the same as the HOL90 exception handling system\(^1\). It defines a single exception format for the entire checker. There is one exception constructor:

```haskell
signature Exception_sig =
  sig
    exception CHK_ERR of {message:string,
                            origin_function:string,
                            origin_structure:string}
```

\(^1\)Thanks to Elsa Gunter who wrote the HOL90 exception handling structure.
14.2. EXCEPTION HANDLING

It takes three strings: the structure name, the routine name, and the message. There is an (assignable) function for printing CHK_ERRs, plus a function Raise that will print out exceptions at the site of occurrence.

The function Raise can be used to raise an exception instead of the SML operation raise. By default, it prints the three fields of a CHK_ERR exception in a readable form. Suppose that the evaluation of the expression exp raises an exception, this can be handled as

```exn handle e => Raise e```

which will print the fields of the exception.

```val Raise : exn -> 'a```

The boolean value print_exceptions is for controlling the display of exceptions. The (assignable) function output_CHK_ERR is for printing the exception messages. Its default value is to print the three fields of CHK_ERR. The function print_CHK_ERR prints a checker exception package.

```val print_exceptions : bool ref```
```val output_CHK_ERR : (message:string,
origin_function:string,
origin_structure:string) -> unit) ref```
```val print_CHK_ERR : exn -> unit```
```end;```

14.2.1 The structure Exception The implementation of the structure Exception is as follows:

```structure Exception : ExceptionSig = structure``
```exception CHK_ERR of {origin_structure:string,
origin_function:string,
message:string}```
```val print_exceptions = ref true;```

(* Assignable function for printing errors *)
```val output_CHK_ERR = ref (fn {message,origin_function,origin_structure} =>
 (output(std_err, ("\nException raised at ""-origin_structure"".""-origin_function"
 (if (origin_structure = ")
 then "
 else ":\n"
 message="\n")

 flush_out std_err));```
```fun print_CHK_ERR (CHK_ERR ss) = !output_CHK_ERR ss```
```fun print_CHK_ERR _ = print_CHK_ERR(CHK_ERR{origin_structure="Exception",
origin_function="print_CHK_ERR",
message="not a checker error"})```
```
```fun Raise (e as CHK_ERR ss) =```
```  ( if (!print_exceptions)```
.then !output_CHK_ERR sss
else ()
  raise e
| Raise (e as _) = raise e

end (* Exception *)
15 Input and output management

The input and output of the checker is managed by the Io module. This provides a simple interface between the file system and the core of the checker. Conceptually, the checker accepts input from a single input stream which is connected to a named proof file. The checker outputs to a single output stream connected to a log file. The connections are made via sockets.

15.1 The specification

The signature Io_sig specifies the interface of the input/output manager. It consists of a small set of functions for opening/closing a file, reading from an opened file via an input socket or writing to an opened file via an output socket.

This arrangement provides a convenient way of filtering the input and output text. Since the proof file is expected to be stored in compressed form, instead of decompressing the file then reading the decompressed file directly, a decompressor can be run as a filter which outputs the decompressed text into a socket, and the checker gets its input from the socket.

The IO manager maintains a pair of sockets, as the current input/output sockets, one to input from and one to output to. After the sockets are opened, all subsequent input is read from the input socket, and output is written to the output socket.

The functions open_input_socket and open_output_socket open a socket to input from and output to, respectively. They take two strings as their arguments. The first string is the name of the file. The second string is the name of the filter. In the case that no filter is required, the second argument should be a null string. The function close_io_socket closes both the input and output sockets.

signature Io_sig =

    sig
    val open_input_socket : string -> string -> unit
    val open_output_socket : string -> string -> unit
    val close_io_socket : unit -> unit

    There are two functions for reading the input file. The function read_bytes reads a fixed number of bytes from the input socket. It returns a pair whose first field is an integer giving the actual number of bytes read and whose second field is a ByteArray containing the input bytes. The function read_input_line reads a line (terminated by carriage return or newline) from the input socket, and returns it as a string. The function write_output_string writes its argument string to the default output socket.

    val read_bytes : unit -> (int * ByteArray.ByteArray)
    val read_input_line : unit -> string
    val write_output_string : string -> unit

    The function mk_command takes a file name as its sole argument. It creates and returns three strings: the name of the log file, and the command strings of the input and output filters. The rules for deriving the output file name are as follows:

    • if the input file name ends with suffix .gz, it is assumed to be a compressed file. The output file will be compressed. The decompression command is gzip, and the compression command is gzip. The output file name will also end with the suffix .gz.
• if the input file name ends with suffix .prf after deleting the .gz suffix if it occurs, then the output file name is the same as the input file name with the string prf replaced by the string clg.

• otherwise, the suffix .clg is appended to the input name to obtain the output name.

The rules are applied in the order listed.

val mk_command : string -> string * string * string
end;

15.2 The structure Io

This structure implements the input/output manager. It uses the low level input/output functions in the structure System.Unsafe.SysIo in the SML/NJ library because we need to create a separate filter process and to manage sockets.

structure Io : IO_sig =
  struct
    fun IO_ERR{function,message} =
      Exception.CHK_ERR{message = message,
        origin_function = function,
        origin_structure = "Io"};
  end

15.2.1 Local variables A set of internal variables is used by the IO manager. The variables in_sname and out_sname are assigned the names of the input and output sockets, respectively.

val in_sname = ref ""
and out_sname = ref ""

The sockets are accessed through file descriptors. After the sockets are opened, the file descriptors are assigned to the identifiers default_ifp and default_ofp. When the sockets are not opened, the names are assigned a null string and the default file descriptors are the invalid descriptor which has the symbolic name NULL.

val NULL = ("1":System.Unsafe.SysIo.fd)
val default_ifp = ref NULL
and default_ofp = ref NULL

The name of the socket server is bound to the identifier server. This server is a separate program which creates the sockets and the filter process, and connects them together. Refer to Appendix C for detailed description of the server.

val server = "so_server"

str_buf is the input buffer. The default input block size is bound to default_block_size.

val str_buf = ref ""
and default_block_size = 80

The local variable debug holds the current debugging setting for this module. The function write_out simply writes a message to the standard error stream. It is used for displaying debugging messages.

val debug = Debug.get_debug("Io");
fun write_out s = (output(std_err,s);flush_out std_err);
15.2.2 Data conversions Because the low level system IO functions work with bytearrays rather than strings, conversion functions are needed. The function `str_to_bytearray` takes a string and its length as its arguments. It returns a byte array containing a sequence of ASCII codes representing the string. Since the element type of a byte array must by integer, and the elements can only be updated one-by-one, the input string is converted into a list of integers (character codes) first; then, they fill the byte array one-by-one.

```ml
local
  fun str_to_bytearray (s,n) =
    let
      val len = ref n
      val si = (map ord (explode s));
      val barray = ByteArray.array(n, 0)
    in
      (while (!len > 0) do
        (len := (!len) - 1;
         ByteArray.update(barray, (!len), (nth(si, (!len)))));
        barray
      end)
    end
```

The function `bytearray_to_string` does the reverse conversion. It takes a byte array and returns a string. The conversion from byte array to string is simpler since the SML/NJ library has a function for this task.

```ml
fun bytearray_to_string ba =
  ByteArray.extract(ba, 0, ByteArray.length(ba))
```

15.2.3 Reading input file The actual reading of a file is carried out by the function `read_string`. It takes a file descriptor and an integer indicating the required number of bytes. It returns a pair whose first field `len` is an integer giving the actual number of characters read, and whose second field is a string containing the input characters. On reaching the end of file, `len` is 0.

```ml
fun read_string fp n =
  let
    val barray = ByteArray.array(n, 0)
    val len = System.Unsafe.SysIO.read(fp, barray, n)
  in
    (len, ByteArray.extract(barray, 0, len))
  end
```

15.2.4 Input buffering Because the logical input unit of the lexical analyser is the line, but the system input function is byte oriented, a local string buffer, namely `str_buf`, is managed. It can hold at most one string. When a block of characters is read, it is split into two parts just after the first end of line character (`\n`) if such a character occurs. The first part is returned whilst the second part is saved in the buffer `str_buf`.

When an input line is needed, the function `get_string` is called. It checks the string buffer first. If it is not empty, the string is returned. Otherwise, the function `read_string` is called to read a block of characters from the input file.

```ml
fun get_string fp =
  if (!str_buf = "") then (read_string fp default_block_size)
  else
    let
      val s = !str_buf
```

in
  (str_buf := ""; ((size s),s))
end;

The function unget_string puts a string back in the string buffer provided the latter is empty. Otherwise, it raises an exception.

fun unget_string s = 
  if (!str_buf) = "" then str_buf := s
  else raise
    (IO_ERR{function="unget_string", message="Buffer not empty"});

15.2.5 Splitting a string  This function split_string takes two strings (cs,s) as its arguments. It returns a triple (found,s1,s2). If one of the characters in cs, say c, occurs in s, then s is split into two sub-strings s1 and s2. The last character of s1 is c. found is set to true. If none of the characters in cs occurs in s, found is set to false, s1 becomes s and s2 is a null string.

fun split_string chrs s = 
  let
    val n = size s
    val (found,ix) = ((true, String#Util.index chrs (s,0) )
                      handle String#Util.NotFound => (false,n))
    val (s1,s2) = if (ix = n) then (s, "")
                  else (substring(s,0,ix+1), substring(s,ix+1, n-ix-1))
  in
    (found, s1, s2)
  end;

15.2.6 Input functions  The function read_line reads a line from the file pointed to by the file descriptor fp. A line is a string whose last character is the 'end of line' character ("\n" by convention). If the end of file is reached, get_string returns 0 to indicate that no character is read. This function returns a null string. Otherwise, the input string is checked for an end of line by calling split_string. If none is found, another string is read, the process is repeated until an end of line or end of file is found. If an end of line is found, the characters after it are pushed back to the string buffer.

fun read_line fp = 
  let
    val outstr = ref ""
    val done = ref false
    in
      while not (!done) do
        let
          val (n,s) = get_string fp
        in
          if (n = 0) then (done := true)
          else
            let val (found, s1,s2) = split_string "\n" s
            in
              outstr := !outstr ^ s1;
              if found
                then (done := true;
if not(size s2 = 0) then unget_string s2 else ()
  else done := false
end
end;
(!outstr)
end;

The function read_input_line reads an input line from the default input file. If no default input file is opened, it raises an exception.

fun read_input_line () =
  if (!default_ifp) = NULL
    then raise (IO_ERR(function="read_input_line",
                      message="No default input stream")

  else read_line (!default_ifp);

The function read_bytes reads a block of bytes from the default input stream. The size of the block is determined by the value in default_block_size. The checker now works directly on bytes, i.e., ASCII codes, therefore, the local functions for converting bytes to strings and buffering are not needed. They have not been deleted in case they are needed in the future. See Chapter 6 for details of how the checker recognises ASCII codes.

fun read_bytes () =
  if (!default_ifp) = NULL
    then raise (IO_ERR(function="read_bytes",
                        message="No default input stream")

  else
    let
      val array = ByteArray.array(default_block_size,0)
      val len = System.Unsafe.SysIO.read
                   (!default_ifp), array, default_block_size
    in
      (len, array)
    end

15.2.7 Output functions  The function write_string writes its second argument str to
the file pointed by the file descriptor fp.

fun write_string fp str =
  let val len = size str
    val ba = str_to bytearray(str,len)
    val n = makestring(ByteArray.length ba) and n0 = makestring len
  in
    if (debug > 3) then
      write_out("write_string:"
               "n0"
               "str"
               "n")
    else ()
    System.Unsafe.SysIO.write(fp, ba, len)
  end;

The function write_output_string writes its argument str to the default output file. If
no default output file is opened, it raises an exception.

fun write_output_string str =
  if (!default_ofp) = NULL
    then raise (IO_ERR(function="write_output_string",
                        message="No default output stream"))

else write_string (!default_ofp) str;  
end(* local *)

15.2.8 Closing sockets The function close_socket takes the file descriptor fp and the 
file name sname of a socket as its argument. It closes the socket and removes the socket file 
without any checking.

fun close_socket (fp,sname) =  
  (System.Unsafe.SysIO.closef fp;  
  System.Unsafe.SysIO.unlink sname)  
handle e => raise IO_ERR {function="close_socket", 
  message= "Error in closing socket"};

The function close_io_socket closes the default IO sockets. It checks to ensure at least 
one socket is currently opened. If no socket is opened, it raises an exception.

fun close_io_socket () =  
  if (!default_ifp = NULL) andalso (!default_ofp = NULL) 
    then raise (IO_ERR {function="close_io_socket", 
        message= "No open socket"})  
  else (if not(!default_ifp = NULL)  
        then (close_socket (!default_ifp), (!in_sname)); 
        default_ifp := NULL; 
        in_sname := ""; ();  
    else ();  
    if not(!default_ofp = NULL)  
      then (close_socket (!default_ofp), (!out_sname)); 
      default_ofp := NULL; 
      out_sname := ""; ();  
    else ()
  );

15.2.9 File access checking The function check_file takes a file name and an access 
mode. Both arguments are strings. It performs the following actions according to the value of 
the access mode string:

- mode = "R" — return true if the file fname exists and can be read;
- mode = "W" — return true if the file fname exists and can be written or if the file fname 
can be created with write permission;
- mode = anything else — return true if the file fname exists.

fun check_file (fname, mode) =  
  case mode of  
    "R" =>  
      (System.Unsafe.SysIO.access(fname,[System.Unsafe.SysIO.A_READ]]) | "W" =>  
      if (System.Unsafe.SysIO.access(fname,[System.Unsafe.SysIO.A_WRITE]]) 
        then true 
        else let 
          val fp =  
          handle e => ("1")
in
  if (fp = -1) then false
  else
    (System.Unsafe.SysIO.close fp;
     System.Unsafe.SysIO.unlink fname; true)
  end
| _ =>
    (System.Unsafe.SysIO.access(fname, []));

15.2.10 Delay function  This function is used to delay the program in retrying to open a
socket. It sleeps for sec seconds before it returns.

  fun sleep sec =
    let val cmd = System.Unsafe.CInterface.c_string("sleep " ^ sec) in
    System.Unsafe.CInterface.system cmd
    end;

15.2.11 Opening sockets  There are four functions for opening sockets arranged into two
tiers. All of these take two strings as their arguments. The first string is the name of the file to
be opened. The second string is the filter to be run. The lower tier functions open_in_socket
and open_out_socket return a file descriptor to the socket and the name of the socket as a
pair. The higher tier functions open_input_socket and open_output_socket return a unit
on successful opening of a socket and make it the default input or output file, respectively.

Conceptually, the input file whose name is given as the first argument to open_input_socket
becomes the default input file. In fact, the file is opened for reading and connected to the input
end of the filter specified as the second argument. The filter sends its output to the socket.
The socket is created with a unique name sname by a socket server server. The default socket
server is an auxiliary program whose name is bound to the identifier server. The new socket
is connected by calling the system function connect_unix which returns a file descriptor to
the socket. This becomes the default input file descriptor and is used to read all inputs. The
process of creating the filter process and socket and connecting them up is carried out by the
lower tier function open_in_socket. The output channel is managed in the same way as the
input.

  fun open_in_socket fname proc =
    let
      val pid = System.Unsafe.CInterface getpid()
      val sname = "RTMP" ^ (makestring pid)
      val cmd' = server " R " sname
      val cmd = if (proc = "") then (cmd' " < " fname " &")
        else (proc " < " fname " | " cmd' " &")
      val ifp = ref NULL
      val n = ref 10
    in
      if not(check_file (fname, "R"))
        then raise (IO_ERR(function="open_in_socket",
            message = "Cannot read file " ^ fname))
        else
          (System.Unsafe.CInterface.system
            (System.Unsafe.CInterface.c_string cmd)
            handle e => raise (IO_ERR(function="open_in_socket",
                message = "Error in creating socket"));
while ((!n > 0) andalso (!ifp = NULL)) do
  (sleep " 2 ";
  n := !n - 1;
  ifp := (System.Unsafe.SysI0.connect_unix sname handle e => NULL));

  if (!ifp = NULL)
    then raise (IO_ERR {function="open_in_socket",
      message= "Error in connecting socket"})
  else
    (!ifp, sname)
  end;
end;

fun open_out_socket fname proc =
  let
    val pid = System.Unsafe.CInterface getpid()
    val sname = "WTMP" (make_string pid)
    val cmd' = server " W " "sname
    val cmd = if (proc = "") then (cmd' " > " "fname" ")
      else (cmd' " | " " proc " > " "fname" ")
    val NULL = ("1":System.Unsafe.SysI0 fd)
    val ofp = ref NULL
    val n = ref 10
  in
    if not (check_file (fname, "W"))
      then raise (IO_ERR {function="open_out_socket",
        message = "Cannot create file " "fname")
    else
      (System.Unsafe.CInterface system
        (System.Unsafe.CInterface c_string cmd)
        handle e => raise (IO_ERR {function="open_out_socket",
          message = "Error in creating socket"});

    while ((!n > 0) andalso (!ofp = NULL)) do
      (sleep " 2 ";
      n := !n - 1;
      ofp := (System.Unsafe.SysI0.connect_unix sname handle e => NULL));

    if (!ofp = NULL)
      then raise (IO_ERR {function="open_out_socket",
        message = "Error in creating socket"})
    else
      (!ofp, sname)
    end;
end;

fun open_input_socket fname proc =
  if not((default_ifp) = NULL)
    then raise (IO_ERR {function="open_input_socket",}
message="socket already opened")})
else
  let
    val (fp, sname) = (open_in_socket fname proc
      handle e => raise (IO_ERR{function="open_input_socket",
                                  message="error in opening socket"}))
  in
    (default_ifp := fp;
     in_sname := sname;
     str_buf := ""; ()
   end;

fun open_output_socket fname proc =
  if not(!default_ofp = NULL)
    then raise (IO_ERR{function="open_output_socket",
                        message="socket already opened")})
  else
    let
      val (fp, sname) = (open_out_socket fname proc
        handle e => raise (IO_ERR{function="open_output_socket",
                                   message="error in opening socket"}))
    in
      (default_ofp := fp;
       out_sname := sname; ()
    end;

15.2.12 Local variables for file name parsing  The names compr_cmd and decompr_cmd
are bound to the shell commands for compressing and decompressing files, respectively. They
are the filtering processes connecting the sockets to the files.

  val compr_cmd = "gzip - "
  and decompr_cmd = "gzcat"

The names prf_suff, log_suff and compr_suff are bound to strings which are the file
name suffixes of proof files, log files and compressed files, respectively. The name suf.sep is
the suffix separator.

  val prf_suff = "prf"
  and log_suff = "clg"
  and compr_suff = "gz"
  and suf.sep = "," 

The names log_suf, compr_suf and log_compr_suf are bound to the suffixes with separ-
ators of the log files, compressed files and compressed log files, respectively.

  val log_suf = (suf.sep ^ log_suff)
  and compr_suf = (suf.sep ^ compr_suff)
  and log_compr_suf = (suf.sep ^ log_suf ^ suf.sep ^ compr_suff)

15.2.13 File name parsing  The function mk_command parses its string argument and returns
a triple (outname, incmd, outcmd) where outname is a string to be used as the output
file name, incmd and outcmd are strings to be used to invoke the input and output filter pro-
cesses, respectively. These filters are decompression and compression programs. The rules for
generating these strings are described in the signature section.

  fun mk_command in_name =
let
  fun imploded l =
    let fun f(x,s) = x""""s in fold f (butlast l) (last l) end
  fun do_path name =
    let
      val i = StringUtil.revindex "/" (name,(size name) -1)
      handle NotFound => ~1
          | e => raise e
    in
      if (i < 0) then ("",name)
      else (substring(name,0,i+1), StringUtil.suffix(name,i+1))
    end
  val (path,fname) = do_path in_name
  val len = StringUtil.size fname
  val l = StringUtil.tokenize suf_sep (fname,len-1)
  val n = length l
  val compressed = ((nth(l,n-1)) = compr_suff)
  val (outname,incmd,outcmd) =
    case n of
      1 => ((fname ~ log_suf), "", "")
    | 2 =>
      if compressed then ((hd l)"log_compr_suf", decompr_cmd, compr_cmd)
      else if ((nth(l,n-1)) = prf_suff) then
        ((hd l) ~ log_suf), "", "")
      else 
        ((imploded l) ~ log_suf), "", "")
    | 0 => ("checker.clg", "", "")
    | _ =>
      if compressed then
      if (last(butlast l) = prf_suff) then
        ((imploded(butlast(butlast l))) ~ log_compr_suf, 
        decompr_cmd, compr_cmd)
      else 
        ((imploded(butlast l)) ~ log_compr_suf, 
        decompr_cmd, compr_cmd)
      else if (last l) = prf_suff then
        ((imploded(butlast l)) ~ log_suf, "", "")
      else ((imploded l) ~ log_suf, "", "")
      in
      if (path = "") then (outname,incmd,outcmd)
      else (path ~ outname,incmd,outcmd)
    end
end; (* end of structure Io *)
The Report module contains several utility functions for outputting various objects to the log file. The log file is assumed connected to the default output stream managed by the Io module. All writings targeted to this file are performed by the function `write_output_string` in Page 15.2.7.

The format of the log file is essentially the same as the proof file. For each proof, all the hypotheses, theorems and definitions which have been used are written into the log file.

The complete syntax of the log file format in an augmented BNF is as below:

```
log_file ::= version time stamp proof_logs ;;

version ::= LP VERSION STRING RP ;;

timestamp ::= LP TIMESTAMP STRING RP ;;

proof_logs ::= proof_log | proof_logs proof_log ;;

proof_log ::= LP PROOF STRING used proved unsolve RP ;;

used ::= LP USED just_list RP ;;

proved ::= LP PROVED thm_list RP ;;

unsolved ::= LP UNSOLVED thm_list RP ;;

just ::= LP justi RP ;;

justi ::= HYPOTHESIS thm
     | DEFINITION STRING thm
     | AXIOM STRING thm
     | THEOREM STRING thm
     | STOREDEFINITION STRING STRING thm
     | NEWAXIOM STRING thm ;;
```

Below is a log file generated by the checker when checking a simple proof named `ap_term`. In this proof, one hypothesis is used and one theorem is proved.

```
(VERSION HOL CHECKER 0.1)

(PROOF ap_term
 (USED [[HYPOTHESIS (THM [ ])(A (A (C =)(A (A (C +)(V m))(V n))))
   (A (A (C +)(V m))(V m))))
 )
)
\[ \text{(PROVED [(THM [])(A (A (C =) (A (C SUC)(A (A (C +)(V m))(V n)))(V n))))}(A (C SUC)(A (A (C +)(V n))(V m))))] \]

16.1 The specification

The signature `Report_sig` is the specification of the report generation module.

\[
\text{signature Report_sig =}
\]

\[
\begin{align*}
\text{sig} & \quad \text{structure Io : IO_sig} \\
\text{structure Keyword : Keyword_sig} & \\
\text{val write_tok : string -> unit} & \\
\text{val write_opening : string -> unit} & \\
\text{val write_line_opening : string -> unit} & \\
\text{val write_opening2 : string * string -> unit} & \\
\text{val write_line_opening2 : string * string -> unit} & \\
\text{val write_closing : unit -> unit} & \\
\text{val write_closing_line : unit -> unit} & \\
\text{val write_item_list : string * string * (('a -> 'b) * 'a list) -> unit} & \\
\text{val write_list : ('a -> 'b) -> 'a list -> unit} & \\
\text{val write_sitem : string * string -> unit} & \\
\end{align*}
\]

The function `timeofday` returns a string containing the current date and time in the format as returned by the UNIX shell command date. It is used by the function `write_log_preamble` which writes the version and timestamp to the log file.

\[
\begin{align*}
\text{val timeofday : unit -> string} & \\
\text{val write_log_preamble : unit -> unit} & \\
\end{align*}
\]

16.2 The implementation

The functor takes two structures: Io, the input/output manager and Keyword which specifies the concrete syntax of the file format.

\[
\text{functor ReportFUN (structure Io : IO_sig and Keyword : Keyword_sig) : Report_sig =}
\]

\[
\begin{align*}
\text{struct} & \quad \text{structure Io = Io} \\
\text{structure Keyword = Keyword} & \\
\end{align*}
\]

\[
\begin{align*}
\text{open Io Keyword} & \\
\end{align*}
\]

16.2.1 Simple items The function `write_sitem` outputs a simple item to the log file. A simple item is an item which has a tag and a string as its only fields. The tag and the string is enclosed in a pair of parentheses and separated by a blank.

\[
\text{fun write_sitem(tag,s) =} \\
\text{Io.write_output_string (Keyword.LP ~ tag ~" ~ s ~ Keyword.RP)}
\]
16.2. THE IMPLEMENTATION

16.2.2 A list of items The function `write_list` outputs a list of items. Each individual item is output by the function `f` which is the first argument to this function. The entire list is enclosed in a pair of brackets.

```latex
and write_list f xs = (write_output_string LB;
  map f xs;
  write_output_string RB)
```

16.2.3 Item having a list The function `write_item_list` outputs an item which contains a list. It takes a triple whose first and second field are the tag and a string, respectively. The tag identifies the type of the item. The last field of the triple is itself a pair whose first field is an output function and whose second field is the list of items which are output by the function `f`.

```latex
and write_item_list (tag,name,(f,args)) =
  (write_output_string (LP ^ tag ^ " " ^ name);
   write_list f args;
   write_output_string RP)
```

16.2.4 Item opening and closing The function `write_opening` outputs the open parenthesis and the tag of an item. The function `write_opening2` outputs the open parenthesis and the first two fields of an item. The first field of an item is always a tag. The second field accepted by this function is a string.

The function `write_closing` simply outputs a closing parenthesis. The function `write_closing_line` is a variant of `write_closing`. It closes the item and adds a new line. There are also variants of opening functions which begin a new line before writing the opening parenthesis.

There should always be a matching call to one of the closing functions for every call to an opening function.

```latex
and write_line_opening_tag =
  write_output_string ("\n" ^ LP ^ tag ^ "")
and write_opening_tag =
  write_output_string (LP ^ tag ^ "")
and write_line_opening2(tag, s) =
  write_output_string ("\n" ^ LP ^ tag ^ " " ^ s)
and write_opening2(tag, s) =
  write_output_string (LP ^ tag ^ " " ^ s)
and write_closing () =
  write_output_string RP
and write_closing_line () =
  write_output_string (RF ^ "\n")
val write_tok = write_output_string
```

16.2.5 Time of day The function `timeofday` is implemented by calling the UNIX command date. The output of the command is written in a temporary file whose name consists of the current process number (so it is unique). The output is a string showing the current date and time, and its terminated by a newline character. This string is read from the file. The newline character is replaced by a space character. The string is returned as the value of the function.

```latex
fun timeofday () =
  let
    val fname = "CHK" ^ (makestring(System.Unsafe.CInterface getpid ()))
    val n = System.Unsafe.CInterface.system
            (System.Unsafe.CInterface.c_string("date > " ^ fname))
```
val inf = open_in fname
val date = input_line inf
in
  close_in inf;
  System.Unsafe.Sys10.unlink (System.Unsafe.CInterface.c_string fname);
  StringUtil.stringTrans("\n"," ") date
end

16.2.6 Writing the preamble This function simply writes out the version and the time stamp.

fun write_log_preamble () =
  (write_opening2 (VERSION, log_versionName);
   write_closing_line ();
   write_opening2 (TIMESTAMP, (timeofday()));
   write_closing_line ()

end (* functor Report * )
The functor StaticArrayFUN is an implementation of the signature STATIC_ARRAY in the SML/NJ library. This is in turn used by the library functor DynamicArray to implement dynamic arrays.

A static array is a data structure whose size is fixed at the time the array is created. An attempt to access an element using an out-of-bound index raises the exception Subscript. In contrast, a dynamic array is a data structure whose size can grow when required. Its elements can be accessed randomly using their indices like an ordinary static array. When an out-of-bound element is accessed, the array grows to a size larger than the given index so that the element will be in the array. This feature may be unsafe, but provides an unbounded data structure so there is no hard limit imposed on the size of data.

Dynamic arrays are used in the checker to hold theorem reference information. See Chapter 9 for more detail.

A.1 The static array functor

The static array module is similar to the Array structure in the pervasive environment. Actually, it is implemented using the Array structure.

functor StaticArrayFUN (type elemType) : STATIC_ARRAY =
    struct
        type elem = elemType
        type array = elem Array.array

        exception Size
        exception Subscript

        fun array (n,e:elem) =
            (Array.array (n,e)
             handle Array.Size => raise Size | exn => raise exn);

        fun sub (a:array,n) =
            (Array.sub (a,n)
             handle Array.Subscript => raise Subscript | exn => raise exn);

        fun update (a:array,n,e) =
            (Array.update (a,n,e)
             handle Array.Subscript => raise Subscript | exn => raise exn);

        fun length (a:array) = Array.length a;
        fun tabulate (n,f:int -> elem) = Array.tabulate (n,f);
        fun arrayoflist (l:elem list) = Array.arrayoflist l;
    end
This appendix contains the SML file join1.sml for loading and linking the modules. To use the checker, one simply loads this file in an SML session by typing the command

use "join1.sml";

After SML processes this file, a function check_proof will be defined at the top level as the entry point to the checker. This function takes a single string as its argument. The string is the name of a proof file.

(* loading some libraries from the SML/NJ library *)

val libpath = "~/compstaff/wwong/lib/sml/smlnj-lib/";

fun load_libs libs = app (fn lib => use (libpath ^ lib)) libs;

load_libs
["lib-base.sig.sml", "lib-base.sml",
 "charset.sig.sml", "charset.sml",
 "ctype.sig.sml", "ctype.sml",
 "makestring.sig.sml", "makestring.sml",
 "list-util.sig.sml", "list-util.sml",
 "listsort.sig.sml", "list-mergesort.sml",
 "string-util.sig.sml", "string-util.sml",
 "string-cvt.sig.sml", "string-cvt.sml",
 "format.sig.sml", "format.sml",
 "static-array.sig.sml", "dynamic-array.sig.sml",
 "dynamic-array.sml", "binary-array.sig.sml",
 "ord-key.sig.sml", "binary-dict.sml",
 "dict.sig.sml", "binary-set.sml",
 "ord-set.sig.sml", "binary-set.sml"];

use "array.sml";
use "genfun.sml";
use "debug.sml";
open Exception;

use "io.sml";
use "keyword.sml";
use "report.sml";
use "parsing.sml";
use "type.sml";
use "term.sml";
use "thm.sml";
use "proof.sml";
use "env.sml";
use "check.sml";
use "pass1.sml";
use "pass2.sml";

structure Report =
  ReportFUN(structure Io = Io and Keyword = HOLProofKey);

structure Parsing =
  ParsingFUN(structure Io = Io and Keyword = HOLProofKey);

structure Htype = HtypeFUN(structure Report = Report);
structure HtypeCmp = HtypeCmpFUN(Htype);

structure Hterm = HtermFUN(structure Report = Report
  and Htype = Htype and HtypeCmp = HtypeCmp);
structure Termcmp = HtermCmpFUN(Hterm);
structure HtermSet = BinarySet(Termcmp);

structure Hthm = HthmFUN(structure Report = Report
  and Hterm = Hterm and HtermSet = HtermSet);

structure Hproof = ProofFUN(structure Hthm = Hthm
  and Hterm = Hterm and Htype = Htype);

structure Henv = HenvFUN(structure Htype = Htype and Hterm = Hterm);

structure Check = CheckFUN(structure Report = Report and Htype = Htype
  and Hterm = Hterm and Hthm = Hthm
  and Proof = Hproof and Henv = Henv
  and HtermSet = HtermSet);

structure Pass1 =
  Pass1FUN(structure Parsing = Parsing and Henv = Henv
  and Proof = Hproof and Hthm = Hthm
  and Hterm = Hterm and Htype = Htype);

structure Pass2 =
  Pass2FUN(structure Parsing = Parsing and Henv = Henv
  and Proof = Hproof and Hthm = Hthm
  and Hterm = Hterm and Htype = Htype
  and Check = Check and Report = Report);

fun check_proof fname =
  (Pass1.parse_file fname; Pass2.parse_file fname);
This appendix contains a small C program which manages sockets and spawns external processes for the checker. It should be invoked with two arguments: the first is a single character which can be either 'R' for reading or 'W' for writing; the second is the name for the socket.

C.1 The program

This is a rather simple program. It starts by including some standard header files and defining a number of symbolic constants.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define STDIN 0
#define STDOUT 1

#define READ 0
#define WRITE 1

#define NAME "mysocket"
```

C.1.1 The main function Here comes the beginning of the main function.

```
main(argc, argv)
int argc;
char *argv[];
{
    int sock, msgsock, rval, rc;
    int rw, nchars;
    struct sockaddr_un server;
    char buf[BUFSIZE], *bp;

    The command line arguments are checked first.
    if (argc != 3)
    {
        fprintf(stderr, "(so_server) command line: number of arguments\n");
        exit(1);
    }
    if (*argv[1] == 'R') rw = READ;
    else if (*argv[1] == 'W') rw = WRITE;
    else
    {
        fprintf(stderr, "(so_server) command line: direction\n");
        exit(1);
    }
```
C.1. THE PROGRAM

The system function socket is called to open a socket. If an error occurs, a negative value is returned.

/* create a socket */
sock = socket(AF_UNIX, SOCK_STREAM, 0);
if(sock < 0)
{
    perror("Opening stream socket");
    exit(1);
}

The name specified by the command line is bound to the socket. The socket appears to the program just as a UNIX file.

server.sun_family = AF_UNIX;
strcpy(server.sun_path, argv[2]);
if(bind(sock, (struct sockaddr *)&server,
        sizeof(struct sockaddr_un)) < 0)
{
    perror("Binding stream socket");
    exit(1);
}

fprintf(stderr,"socket name:%s\n", server.sun_path);

Now, we are ready to accept a connection to the socket. When an incoming request is processed, accept returns a value to indicate the outcome.

listen(sock, 1);

msgsock = accept(sock, (struct sockaddr *)&0, (int *)&0);
fprintf(stderr,"SERVER:accept socket No.%d\n", msgsock);

If the connection is successful, we initialise the input/output buffer buf and start passing characters. If the socket was opened for reading, we pass everything from the standard input to the socket until the end of the file is found, in this case, read returns 0. If the socket was opened for writing, we pass anything from the socket to the standard output until read returns 0.

if(msgsock == -1) perror("accept");
else
{
    bzero(buf, sizeof buf);
    if (rw == READ)
    {
        while((nchars = read(STDIN, buf, BUFSIZ)) > 0)
        {
            if((rval = write(msgsock, buf, nchars)) < 0)
            {
                perror("Writing stream message");
                rc = 1;
                break;
            }  
        }  
    }
}
else
{
    do
    {
        if((rval = read(msgsock, buf, (BUFSIZE-1))) < 0)
            perror("Reading stream message");
        else if (rval == 0)
            /* printf("Ending connection\n"); */
            rc = 1;
        else{
            *(buf + rval) = '\0';
            write(STDOUT, buf, rval);
        }
    }while (rval > 0);
}

All sockets are closed before exit.
close(msgsock);
close(sock);
/*
   unlink(NAME); */
exit(rc);
References


Index

prf file, 5
accept, 159
aconv, 110
add_const, 98, 104
add_pline, 84, 90
add_pline_tab, 84, 88
add_pline_table, 84, 89
add_type, 98, 104
ALPHA, 32, 33
alphas, 44
Array, 155

barray_to_string, 143
basic, 5
buf, 159
byte_to_char, 33
ByteArray, 141

char_buf, 34
char_buf_len, 34
char_to_str, 33
chr, 34
Check, 45
check_file, 146
check_proof, 8, 156
check_ver_string, 31, 39
chk, 89
chk_buf, 34
CHK_ERR, 139
chk_pline, 25, 45, 78
chk_proof, 24, 45, 80
chk_rename, 120
clear_proof, 83, 92
close_io_socket, 141, 146
close_socket, 146
cmp_Type_list, 130
cmpTerm, 110, 113, 127
cmpType, 129, 130, 132, 136
cmpr_cmd, 149
cmpr_suff, 149
cmpr_suff, 149

Concl, 107
conc, 106, 107
conclusion, 5
core, 5
core checker, 7
core version, 4
ctype, 31–33
current_env, 99
current_goal, 88
current_pline, 88, 89
current_proof, 88, 92
current_ptab, 88, 89

Debug, 138
debug, 142
decompr_cmd, 149
default_block_size, 142, 145
default_ifp, 142
default_ofp, 142
derived theorem, 6
dest_abs, 115, 118
dest_bind, 115
dest_eq, 110
dest_funtpe, 118, 128
dest_type, 128
dest_vartype, 128
DICT, 137
Dict, 138
domain_of, 128
DynamicArray, 155

efficient checker, 7
efficient version, 4
evlist, 102
evName, 97, 99
EUF, 33
Equal, 110, 129
equal, 107
extended, 5

F, 56, 57, 61, 72, 73
false, 88

162
sub_list, 126
sublist_chk, 124, 125
Subscript, 155
SUC, 53
suf_sep, 149
SYM, 32, 33
symbols, 44

T, 57, 66, 68, 74
term_inst_chk, 111
term_inst_renames, 111
term_is_type_inst, 110, 120
term_is_tyty_inst, 110
term_subst_chk, 74, 75, 111, 125

theorem provers, 3
thm_eq, 106
timeofday, 152, 153
toupper, 19
true, 84, 88, 106, 125, 129
ty_compat, 134
ty_tyvars, 129
tyvarl, 129
type_compat, 129
type_in, 110, 119
type_in_type, 110, 129
type_instl, 129
type_of, 110
type_OK, 98, 103
type_replace, 133
type_tyvars, 129
tyvar, 44
tyvars, 110, 119, 120, 131
tyvarl, 110, 119
tyvchar, 37
unget, 35, 36
unget_string, 144
update_tab, 89

variant, 111
VERSION, 44
versionName, 44
vty_occurs, 110, 120

well.Typed, 98, 103
write_closing, 153
write_closing_line, 153
write_item_list, 153
write_list, 153
write_log_preamble, 152
write_opening, 153
write_opening2, 153
write_out, 142
write_output_string, 141, 145, 151
write_sitem, 152
write_string, 145
write_used, 47
write_used2, 47