A Gentle Introduction to Isabelle

Sara Kalvala (sk@cl.cam.ac.uk)
University of Cambridge, Computer Laboratory

June 1994

Abstract

This document provides a starting point to the use of the Isabelle system. The main purpose is to introduce the method for developing proofs supported by Isabelle, through simple examples. It is intended to be used while running an Isabelle session, and emphasizes getting a feel for the system rather than a full understanding of how it works.

1 Introduction

Theorems provers (and particularly those in the LCF-style family, such as Isabelle) are often thought of as having a steep learning curve: many people find it very difficult to get started, and get used to the methodology involved. The payoff comes when the user is able to exploit the characteristics of the methodology, such as the functions and term language characterizing the prover and the distinction between the meta-language (the programming language in which the theorem-prover is implemented and which is accessible to the user).

This tutorial tracks the main parts of a proof endeavor. The first step is to specify a proof problem: this is done by defining a *theory* of definitions as illustrated in Section 3. The next step is to set up a proof goal and work on in interactively, as described in Section 4 and using the actual functions described in Section 5. It is expected that this short tutorial will provide a basis with which to peruse the more complete documentation, such as the more complete introduction to the system found in [Pau93a].

1.1 A family of provers

The basic Isabelle system provides a framework for developing proof systems. This framework, called *Pure* Isabelle, can be instantiated with rules, axioms, and syntax for many different logics, in the form of Isabelle *theories*. These logics can then be used as deductive systems within the particular logic used. The logics which are implemented in Isabelle make use of a *meta-logic* defined in Pure Isabelle, with a syntax for theorems and a set of rules for manipulating theorems. Underlying the use of Isabelle, then, is the concept of a meta-level logic and an object-level logic. (The meta-logic of Isabelle is intuitionistic higher-order logic.)

Several instantiations of Isabelle to particular object logics are part of the Isabelle distribution, providing off-the-shelf theorem-proving environments. Also, while many users would like to use Isabelle to mechanize logics, it is easier to learn how to use the system by starting with an existing logic. Once it is understood how to use Isabelle to model problems within a logic, it becomes easier to extend the system with other logics. Some of the in-built logics—the ones which are well developed and documented are

- FOL: predicate calculus, in both intuitionistic and classical versions,
- HOL: higher-order logic, typed, with quantification over functions and functionals,
- ZF: Zermelo-Fraenkel's axiomatization of set theory, and
- CTT: Martin-Löf's constructive type theory, with extensional equality.

These object logics are described in [Pau93b]. In this paper, HOL will be the object logic used: it is expressive and quite straightforward to understand, and therefore well-suited for general theorem-proving.

1.2 Running Isabelle

Isabelle runs on Standard ML, and can be built using the instructions supplied with the distribution. The main implementations of SML are Poly/ML and SML of New Jersey.

At Cambridge we mostly use the Poly/ML implementation. The Poly/ML interpreter can be invoked by running the command "poly" with a given *database*. For example, Poly/ML is invoked below with the database that contains the definition of the HOL logic: [1]

```
$ poly -r /usr/groups/theory/isabelle/sun4/HOL
```

Isabelle does not directly incorporate facilities to store results of a session to be used at another session, relying instead on methods provided by the implementation of SML. With Poly/ML, the resulting state can be saved for later continuation by using a writable database without the `-r` option above. With SML of New Jersey, a new core file can be created using the `exportML` function.

2 Terms and theorems

In the interaction with Isabelle, object level "sentences" are entered as ML strings, which are then parsed into *terms*. Usually this parsing is completely transparent to the user. The process of parsing a string into a term depends on a 'theory' (this concept is explained below). Thus, some operators, such as the logical conjunction (`&`) and implication (`-->`), are defined in the HOL theory, while others such as less-than (`<`) are not. Therefore, in the context of the theory called `HOL.thy` the string `"a & b --> b & a"` is parsed successfully, but on the other hand the string `"m < n"` results in a parsing error. In most cases users need not worry about specifying the theory under which a string will be parsed: this is taken care of automatically by the Isabelle functions that incorporate parsing. As a note, a multi-line string is input by entering a backslash (\) as the last character of a line and the first character of the next line.

When attempting to write out terms in any of Isabelle's object logics, it is important to keep in mind the distinction between the object-level and the meta-level. The operators in the meta-logic are: universal quantification (`!!`), implication (`==>`), and equality (`==`). On the other hand, the object logic (in this case HOL) has its own operators: universal and existential quantification (`!` and `?`), conjunction and disjunction (`&` and `|`), equality (`=`), negation (`~`) and implication (`-->`). This distinction seems small but is sometimes quite important. For example, the operators `==>` and `-->` both denote implication; the first one is used to show an *inference* from what is on the l.h.s. to the proposition on the r.h.s., and signifies a derivation. The latter, though, is used to describe a property in the object-level. (In practice, it is best to use meta-level implication while setting out proof goals, reserving object-level implication only when needed, e.g. when doing induction, which

[1] At Cambridge, the executable `poly` can be found in the directory `/usr/groups/theory/poly2.04/$ARCH/poly` and the databases in the directory `/usr/groups/theory/isabelle/sun4`.

has to be in the object-level.) The meta-level implication is important in the proof procedures existing in Pure Isabelle, as explained further on.

Theorems are objects in Isabelle that are constructed from terms, and are entered as axioms or produced by proof. An example of a theorem is:

```
> zero_less_Suc;
val it = "0 < Suc(?n)" : thm
```

The symbol `?` in the above theorem denotes a (meta-level) variable that can be unified with any expression of the appropriate type. In the above, `Suc` is a constant denoting a function on natural numbers.

The topic of how to build theorems—the essential activity in theorem proving—will be covered in the next sections, the first of which describes how new theorems are postulated and must be accepted at face value and the second which describes how theorems are derived from others.

3 Theories

Theories are modules that contain information on a logic, or in the case of applications definitions that describe the problem being modelled.

Theories are often defined as *extensions* of other theories. When using Isabelle for reasoning about some domain, one writes theories as extensions to existing logics, and theories in which underlying types and data structures are made available. As an illustration, the theory `Nat.thy` contains the specification for natural numbers, and the less-than relation (<) on natural numbers. The theory `Arith.thy` contains syntax and axiomatization for arithmetic operations on numbers such as addition and subtraction. Thus, when `Nat.thy` is used as a context the string `"0 < m"` is parsed, but not the string `"0 < m div n"`, as `+` is defined in (`Arith.thy`), which contains but is not contained by `Nat.thy`.

3.1 Constants and rules

Simple theories extend a proof environment with definitions. In Figure 1 is an instance of such a theory. The text in the figure represents a theory which is an *extension* of the theory of Arithmetic (in the higher-order logic) by the constants `divides` and `GCD`, defined as given. In this theory, called `gcd_theory`, two logical constants are defined. The theory also states two axioms (theorems), called `divides_def` and `GCD_def`.

A file with the text describing a theory can be loaded into an Isabelle session using the `use_thy` function, that takes a file containing such stylized notation and then generates appropriate ML objects:

```
> use_thy "gcd";

structure gcd : sig
   val GCD_def : thm
   val thy : theory
   val divides_def : thm
   val ax : string -> thm
   end
```

The information that is contained in the ML structures returned by `use_thy` is:

```
gcd = Arith +

consts

        divides :: "[nat,nat]=>bool"                 (infixl 50)
        GCD :: "[nat,nat,nat]=>bool"

rules

divides_def  "y divides x == (? c. x = c * y)"

GCD_def      "GCD(a, b, c) == (c divides a) & (c divides b) &     \
\       (!d. ((d divides a) & (d divides b)) -->     \
\                                (d divides c))"

end
```

Figure 1: A theory for GCDs

- constants: declaration of their names and types;
- axioms: declaration of their names and a string giving their bodies.

Theories do not contain theorems derived from the axioms; such theorems are obtained by proof later on, using the methods shown in the next section.

Other examples of theory definitions can be found in the Isabelle sources, as most logics are added to Isabelle in this form. In fact, browsing through the theory files defining object logics can be quite instructive, and the form of these files makes them almost self-documenting.

3.2 Other entries

Apart from the declaration of constants and definitions as above, theories can be used to modify the language accepted by the Isabelle parser. One way to do this is to declare infixes or change the binding New syntax for objects can be defined in a theory, as in the following code that declares the string `1` to be equivalent to the expression `Suc(0)`:

```
translations "1" == "Suc(0)"
```

For many simple applications, it is enough to create theories which contain definitions and axioms (as above) and develop proofs (as explained in the next section). For more complex applications, one often has to define particular *types*, especially in the case of strongly typed logics such as `HOL`. The types can be completely new ones, or they can be type *constructors*. For an example of the former, natural numbers (in `HOL`) are represented as a new type, with the following specification:

```
types nat 0
arities nat :: term
        nat :: ord
```

which is part of `nat.thy` in the `HOL` logic. The numeral in the `types` declaration represent the number of (type) arguments by which the new type is parametrized; the `arities` represent the possible *classes* of types that the argument and resulting types can belong to, and provide a way of introducing polymorphism. (Classes are also declared in theory files, particularly when declaring new logics, but they will not be described here.

Types can also be parametrized by other types. For example, sets must be sets *of* something, and therefore they are defined as being parametrized by an underlying types of elements:

```
types set 1
arities
  set :: (term) term
  set :: (term) ord
  set :: (term) minus
```

(The lines specifying arities above, as mentioned in the case of `nat`, allow polimorphism. They will not be explained here.)

4 Proofs

Theorems can either be axioms (as shown above) or they can be obtained by combining already existing theorems. To understand this derivation of theorems we must examine the way theorems can be manipulated by the meta-level proof functions.

4.1 Rules and Unification

In Isabelle, inference rules are theorems that contain premises and a conclusion, linked by a meta-level implication. They are created through proof or as axioms, just like any other theorems. The prescribed fashion for the use of logical rules is in natural deduction style: operators in the logic are given rules for their introduction and elimination. For example, for the object-level (HOL) implication, we can think of the following rules:

$$\frac{\begin{array}{c}[P]\\ \vdots \\ Q\end{array}}{P \to Q} \qquad\qquad \frac{P \to Q \qquad P}{Q}$$

These rules, belonging to the axiomatization of HOL, provide a method for reasoning about theorems containing an implication. They are translated into Isabelle using the meta-level implication:

```
> impI;
val it = "(?P ==> ?Q) ==> ?P --> ?Q" : thm
> mp;
val it = "[| ?P --> ?Q; ?P |] ==> ?Q" : thm
```

Here the meta-level implication (`==>`) separates the premises from the conclusion of the rule. The symbols `[|` and `|]` denote multiple premises, which are separated by semicolons. This format for rules allows theorems to be matched to either a premise or a conclusion. The matching as such is done by the use of higher-order unification. Details of this matching can be found elsewhere [Pau93a]. Ways of using such rules to generate new theorems will be described below.

4.2 Forward resolution

Forward proof consists in combining theorems to create new ones. The typical way to do this is by resolution. Figure 2 shows how resolution using the `RS` function works. The consequence of the

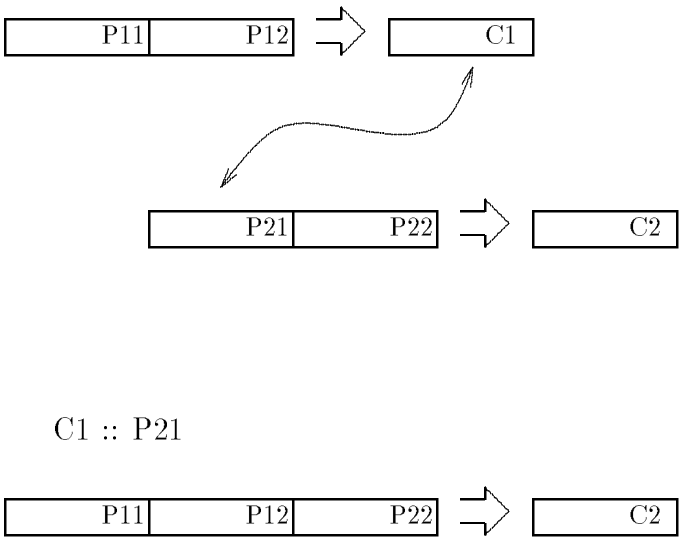

Figure 2: Forward proof by resolution

first theorem is unified with the first premise of the second theorem (unification is represented in the figures as `::`) , resulting in the third theorem.

This use of resolution can also be understood by examining the following examples:

```
> sym;
val it = "?s = ?t ==> ?t = ?s" : thm
> trans;
val it = "[| ?r = ?s; ?s = ?t |] ==> ?r = ?t" : thm
> sym RS trans;
val it = "[| ?s = ?r; ?s = ?t |] ==> ?r = ?t" : thm
> trans RS sym;
val it = "[| ?s = ?s1; ?s1 = ?t |] ==> ?t = ?s" : thm
```

To obtain the result of `sym RS trans`, the conclusion of `sym` is unified with the first premise of `trans`, resulting in the substitutions `t` $\mapsto$ `r` and `s` $\mapsto$ `s`. The premise of `sym` is then used instead of the first premise of `trans`. The other resolution, `trans RS sym`, unifies resulting in the substitutions `r` $\mapsto$ `s` and `t` $\mapsto$ `t`. The variable `s` in `trans` is renamed to `s1`. The two premises of `trans` substitute the premise of `sym`.

The function `RS` always attempts to unify with the *first* premise of the second theorem. A more general function is `RSN`, which accepts an integer argument specifying which premise to unify with. Another related function is `compose` which unifies with the *whole* of the first theorem rather than only its conclusion.

Resolution provides a method to join theorems in an easy way. However, it is difficult to use it to derive very complex theorems. For this, the subgoal package can be used to develop proofs, as described next.

4.3 Backward proofs

Proofs in Isabelle consist in setting a goal and applying tactics to reduce goals into subgoals or prove a subgoal.

The use of the subgoal package will be illustrated through an example. The first step consists in declaring a proof goal. This is done below:

```
> goalw gcd.thy [divides_def] "b divides 0";
Level 0
b divides 0
 1. ? c. 0 = c * b
val it = [] : thm list
```

The function `goalw` sets a goal and expands the definitions given. The actual subgoal that the user must then prove is output. There is only one subgoal to be proved, and it is numbered as `1`. When there is no need to expand any definitions, one may use the simpler function `goal` to start a goal.

A goal is set within a theory, which specifies constants, types, axioms, etc and also the syntax which can be used in the proof. Thus, the information in this theory is used for parsing and for maintaining consistency.

The next step is to expand this subgoal. The theorem `exI` represents the introduction of an existentially quantified variable. In backward proof it replaces the quantified variable by a generic variable.

```
> exI;
val it = "?P(?x) ==> ? x. ?P(x)" : thm
```

Note above the difference between the strings `?  x` and `?x`; the former represents a variable which is quantified *in the object level* while the latter represents a schematic (or *logical*) variable.

This variable can be instantiated when used in the proof, as below where it is instantiated to `0` during the application of a tactic:

```
> by (res_inst_tac [("x","0")] exI 1);
Level 1
b divides 0
 1. 0 = 0 * b
val it = () : unit
```

Now, it is simple to solve this subgoal with a lemma about multiplication. Finally, the function `result` returns the last theorem proved through the subgoal package.

```
> by (simp_tac arith_ss 1);
Level 2
b divides 0
No subgoals!
val it = () : unit
> result();
val it = "?b divides 0" : thm
```

Once the tactic to prove a certain goal is known, the corresponding theorem can be produced without recourse to the subgoal package:

```
> val b_divides_0 =
#    prove_goalw gcd.thy [gcd.divides_def] "b_divides 0"
#    (fn _ => [res_inst_tac [("x","0")] exI 1,
#              simp_tac arith_ss 1]);

val b_divides_0 = "?b divides 0" : thm
```

In using the tactics, it is important to be aware that there is a distinction between assumptions and premises. When entering a goal of the form `[|` a `,` b `|] ==>` c, the actual subgoal presented to the user is just c, while the terms a and b are returned to the user in the form of a list of theorems, which are the *premises*. The premises generated are available as theorems throughout the proof steps for this goal. On the other hand, when a subgoal of the form `[|` a `,` b `|] ==>` c is generated *during* a proof, the terms a and b become the *assumptions* at this proof state, and cannot be accessed as theorems.

The following example should make the creation of assumptions clear:

```
> val prems = goal Arith.thy "a = 0 ==> a + b = b";

Level 0
a + b = b
 1. a + b = b
val prems = ["a = 0  [a = 0]"] : thm list

> hd(prems);
val it = "a = 0  [a = 0]" : thm
```

Here, we see that the term `a = 0` is returned as a theorem. In fact, the theorem states that `a = 0` is true under the assumption (in square brackets) that `a = 0`, which is, of course, a theorem. More specifically, the variable `prems` is bound to the list of such premises.

The next goal differs in using an object level implication.

```
> val prems = goal Arith.thy "a = 0 --> a + b = b";
Level 0
a = 0 --> a + b = b
 1. a = 0 --> a + b = b
val prems = [] : thm list
```

Here, the list of premises is empty, as the implication is in the object level. It can be expanded into a meta-level implication easily:

```
> by (resolve_tac [impI] 1);
Level 1
a = 0 --> a + b = b
 1. a = 0 ==> a + b = b
val it = () : unit
```

The goal here looks similar to the previous goal, but there is a difference: the term `"a = 0"` now constitutes an *assumption* of the goal. Several tactics (`asm_simp_tac`, `dres_inst_tac`, etc.) manipulate assumptions, but not premises (which must be treated as theorems, and manipulated in a forward direction).

Proofs in Isabelle have no focus on any one subgoal at any point. In fact, tactics must contain an explicit numeric parameter to specify which subgoal should be affected by the tactic.

5 Proof methods

In the previous section the facilities for interacting an ongoing proof are shown; in this section some of the actual methods for changing the state of the proof are described.

5.1 Resolution tactics

There are three main ways to use resolution in backward proofs:

- resolution: the simplest kind of resolution is to unify the conclusion of the goal with the conclusion of the rule, and generating subgoals to prove each of the (unified) premises of the rules. If each of these subgoals can be proved at the current proof state, then (because there exists a theorem from these premises to the conclusion) the original conclusion is also provable.
- elim-resolution: variation of resolution in which not only the conclusion part of the goal but also one of the assumptions are unified with the rule being used.
- destruct-resolution: allows forward reasoning from the assumptions.

These are illustrated in Figure 3, simplified by not considering unification and renaming.

An important detail to keep in mind is *backtracking*. Essentially, unification can produce several results, so Isabelle chooses one of them. However, this may not be what is desired; the way to produce another unifier is to use the `back` function. Another is to instantiate variables explicitly in the unification, this is accomplished by the related tactics `res_inst_tac`, etc.

Many other tactics exist. Among them are the more sophisticated tools described next.

5.2 Simplifier

Simplification consists in rewriting a goal using a set of equational theorems. Non-equational theorems are converted to equality (with `True` or `False`) when added into a simplification set (of ML type `simpset`).

In the example of Section 4.3 one may observe the use of `simp_tac` given the simplification set embodying several arithmetic lemmas.

Rules used for simplification are contained in so-called "simplification sets"; some are built in (in HOL, for example, the main ones are `HOL_ss`, `nat_ss`, `arith_ss`), but often they have to be built for proof attempts. New rules can be added simply using the `addsimps` function.

There are several powerful tactics which use simplification; they are

- `simp_tac`: takes as argument a simplification set, and uses it to rewrite the conclusion part of the subgoal.
- `asm_simp_tac`: as above, but also joins the assumptions of the subgoal into the simplification set used.
- `asm_full_simp_tac`: performs rewriting of the assumptions themselves with the simplification set as well as other assumptions before rewriting the conclusion as in `asm_simp_tac`.

A problem with simplification is that it can lead to infinite rewriting. An example of a rule which does this is the commutativity of addition: `a + b = b + a`. One solution to this problem is to specialize a theorem to work only in some cases, using the functions `read_instantiate` and `read_instantiate_sg`. However, there are problems that arise due to scope of variable and constant bindings. Another option consists in changing the equality into an implication:

```
> add_commute;
?m + ?n = ?n + ?m
```

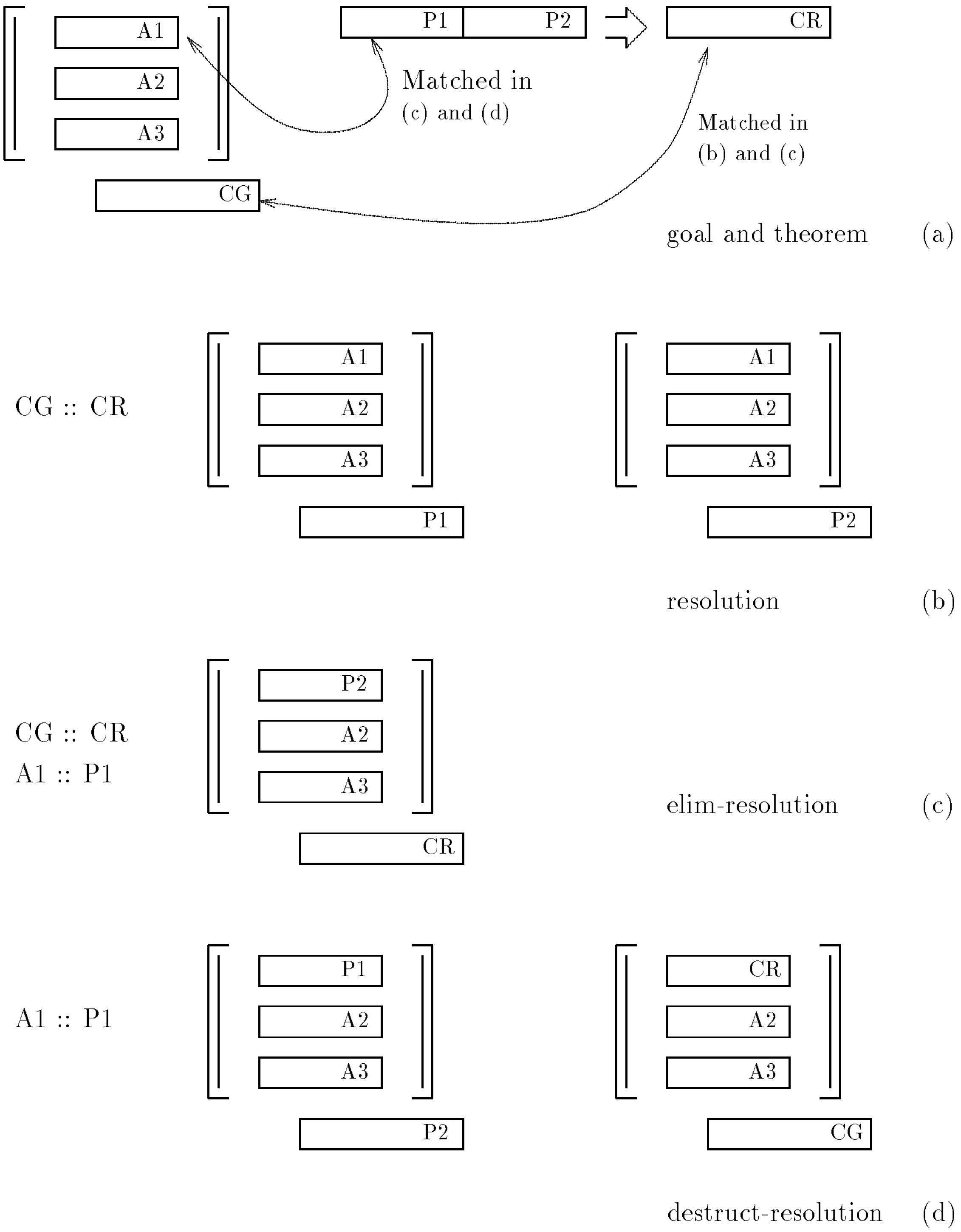

Figure 3: Resolution tactics

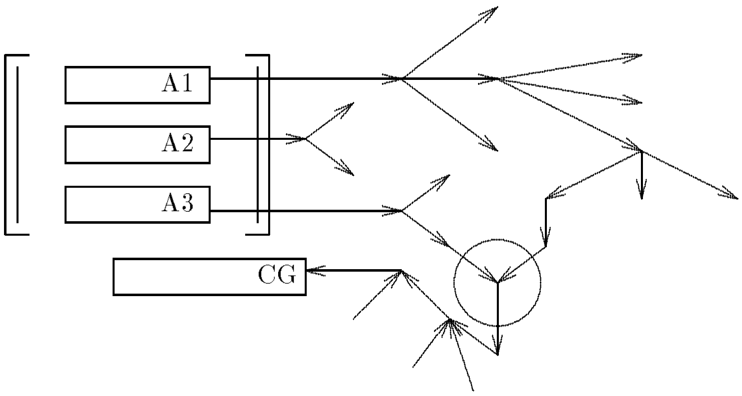

Figure 4: Use of the classical prover

```
val it =  : thm
> add_commute RS subst;
?P(?m1 + ?n1) ==> ?P(?n1 + ?m1)
val it =  : thm
```

This new theorem can be resolved against the goal, rather than using the simplifier. However, in most cases the simplifier is adequate, and becomes a powerful tool for proof.

5.3 The classical prover

Isabelle provides a level of automation of proof in the form of proof search tactics such as `fast_tac`, which exploit most of the in-built rules for the pre-defined logical connectives. These tactics can prove a surprisingly large number of goals. The use of the classical provers is illustrated in Figure 4. Rules are applied to both assumptions and the conclusion of the goal, generating subgoals successively, until the subgoals are proved by assumption. In Figure 4, the circle indicates the point where (forward) expansions of the assumptions and (backward) expansions of the conclusion converge at a theorem.

As several rules may be applicable to a subgoal, there are nodes in the proof where it must be decided which to apply and then which choice of subgoals to explore further, with the possibility of backtracking if a proof is not found at the particular branch taken. The search method can be simple depth-first (in the case of `fast_tac`) or can incorporate heuristics (such as `best_tac`). In either case, they use rules which are contained in "classical sets" (ML type `claset`). Classical sets store rules in three different ways, depending on how they will be used: if they will be used to expand the conclusion, one of the assumptions, or both at the same time. Theorems are added in these three groups using `addIs`, `addDs`, and `addEs`, respectively. Also, rules are classified regarding whether they are safe or not, that is, on whether they generate subgoals in a safe way or not.

The code below shows a simple example of use.

```
> goal HOL.thy "(~ a & ~ b) --> ~ (a|b)";
Level 0
~ a & ~ b --> ~ (a | b)
 1. ~ a & ~ b --> ~ (a | b)
val it = [] : thm list

> by (fast_tac HOL_cs 1);
Level 1
~ a & ~ b --> ~ (a | b)
```

```
No subgoals!
val it = () : unit
```

When a subgoal consists in a logical tautology, one is better off by initially trying to solve it using the appropriate classical prover. However, the tactic `fast_tac` will either prove the goal or fail; it will not produce a new subgoal to be proved.

6 Interface

The interactions shown in this document have shown the input being directly typed into the ML session. In practice, most users prefer to maintain Isabelle code in a text file, and select portions of it to be sent to the process. This can be done using window and mouse based "cut-and-paste".

One may also use a more automated approach based on running a process under the Emacs editor. A window being displayed can have the text file being edited, and another can mirror the output from the Isabelle process. The code distributed with SML of New Jersey is quite convenient: it provides specialized editing features as well as commands to interact with the ML process. Simple sequences of keyclicks can trigger several different actions, such as start and end the SML process, send the whole file being edited or only a marked area of the window buffer to the SML process, and track down the parts of the file which result in syntax errors.

In addition, a more sophisticated interface has been developed by David Aspinall. It takes care of much of the communication between the file where the proof is being logged and the current status of the proof is displayed, as well as many other help facilities.

While learning how to use this methodology may take some time (very short for those already used to Emacs) it pays off in the long run in speeding up the coding process.

References

[Pau93a] Lawrence Paulson. Introduction to Isabelle. Technical Report 280, University of Cambridge, Computer Lab, 1993.

[Pau93b] Lawrence Paulson. Isabelle's object logics. Technical Report 286, University of Cambridge, Computer Lab, 1993.