Preprint.
Presented at the E-CAP2004.

Defining and using deductive systems with Isabelle

F. Miguel Dionísio, Paula Gouveia, João Marcos

{fmd,mpg,jmarcos}@math.ist.utl.pt
Center for Logic and Computation, Department of Mathematics, IST
Lisbon, Portugal

There are several computerized tools to automatically and/or interactively prove theorems of given deductive systems. In general, such tools deal with only one specific deductive system or a small class thereof (for example, Otter deals with classical first-order logic with equality, NUPRL deals with intuitionistic type theory, ModLeanTAP deals with a range of propositional modal logics). The software Isabelle is a generic theorem-proving environment that allows for the definition and use of deductive systems for many different logics. The deduction rules may be specified in different formats: natural deduction rules, Hilbert-style axioms and rules, sequent-style rules, tableau rules, etc. In this way, using Isabelle, it is possible to define and experiment with different logics, since the user may implement the deduction rules she sees fit.

Obviously, some initial training is needed for the task of using Isabelle in order to define and experiment with new logics. However, it is our experience that only a few main concepts are, in fact, essential. The authors have been involved in teaching a logic course for undergraduate students. The system Isabelle was used for representation and use of natural deduction systems for propositional, first-order and modal logics. Students learned how to define logics and how to prove theorems and check inferences in those logics. From what they learned in one semester, most students were able to successfully deliver the final assignment that included the definition of a (new) hybrid logic, involving quantifiers and modalities. In this paper you will find the basic concepts needed to define and experiment with a logic in Isabelle.

1 Introduction

At the moment, there is a sizable number of models of logic software available, and seemingly there is still a strong impulse at the logic community to go and design their own personalized proof assistant, for their own preferred deductive systems. Some of the already existing logic software are highly flexible, and are intended as generic environments for doing a number of activities you would like the computer to help you with: doing interactive proofs, writing some formally verified mathematics, writing and checking formal specifications of all sorts, and, why not, doing some mechanized reasoning, that old dream of Leibniz.

Such highly flexible models of logic software have a strong potentiality still largely to be unleashed in their use in computer-based learning and their integration to the standard set of teaching strategies and resources. Used as laboratories for experimentation with abstract entities, the right software can turn a computer into a sort of 'bubble chamber' where ideas can be tested and improved —or rejected. Moreover, in the provocative words of Edward Feigenbaum (Gleick, 1987), computers can also help us in 'creating intuition' about certain subjects. This outlook converts the computer into a genuine tool for doing philosophical and mathematical research.

We had a simple goal in mind: To use the computer to teach logic to undergraduate students. Our main aim, however, was not (only) to teach the students about this and that deductive

system, but to provide them with some expertise on using an extensively customizable tool in which they would be able to define and work with *their own logics*, if that be the case. After some prospecting and experimentation with the existing proof assistants, Isabelle was chosen. Logic is carefully built-in in the design and implementation of this software. We are talking about a logical framework whose meta-*logic* is Intuitionistic Higher-Order Logic with three main components: (i) a meta-*implication* that can be used in laying down the object-logic rules (see section 2.2) of the specific object-logic being thereby represented and that takes care of the application of those rules and the discharge of assumptions; (ii) a meta-*universal quantification* that can be used in laying down a number of object-language quantifiers (see section 2.6); (iii) a meta-*equality* that can be used in laying down abbreviations as rewrite rules (see section 2.3). We are talking about a mechanizable theorem prover (see section 2.5) where: (i) object-logic formulas are λ-terms disambiguated by way of a priority grammar (see section 2.1); (ii) rules of the object-language are represented not as functions but as formulas of the underlying higher-order logic; (iii) the combination and application of those rules is performed by way of a uniform method of inference — higher-order resolution; (iv) tactics are implemented independently of the object-logic being represented (see section 2.4). We are talking about a simply typed environment where object-language formulas can be heavily structured (see sections 2.7 and 3), and very precise specifications can be met with.

Finally, it should be noted that Isabelle is written over ML (Paulson, 1996), a functional programming language that can come to help at any point where even more expressivity is needed. ML was designed precisely to serve as an implementation language for theorem provers.

This paper is not about the logic behind Isabelle (for that you may consult the appropriate handbooks), but about how Isabelle can be used to look ahead into a number of new user-defined logics. We will not be worried here about proving properties about our object-logics, but about how these very logics can be defined, changed, and used. To that intent, the following sections will systematically illustrate the USE of Isabelle in entirely pedestrian terms.

2 Defining logics in Isabelle

The system Isabelle (Nipkow, Paulson and Wenzel, 2002) has been developed by Lawrence C. Paulson (Univ. of Cambridge, UK) and Tobias Nipkow (Technical Univ. of Munich, DE) and is freely available on the web at `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/`. The relevant references about this system are also available on that site. For other mechanized reasoning systems see `http://www-formal.stanford.edu/clt/ARS/systems.html`, where an extensive commented list is available.

Isabelle is a generic theorem-proving environment that allows for the representation and use of deductive systems for many different logics. In the following we briefly describe the basic concepts needed to define and experiment with a logic. We begin with simple examples concerning classical propositional logic. In particular we will refer to natural deduction, Hilbert- and sequent-style systems for this logic (Troelstra and Schwichtenberg, 1996). More involved examples will be presented later on.

2.1 Language

The definition of the language and deductive system of a given logic constitutes a *theory* of Isabelle's meta-logic. Each theory must be specified in one file (with extension `.thy`).

Connectives are internally represented using λ-calculus and, therefore, understood as functions that for given argument formulas return a new formula. For example, `conj(A,B)` represents the conjunction of `A` and `B`. In this way `conj` is a function that, given two argument formulas,

returns another a formula. The definition of this function is `conj::[o,o]=>o`, where `o` is the type of formulas. The more usual notation `A&B` may also be used (and is internally translated to `conj(A,B)`). Isabelle's code and notation for this connective is:

```
conj ::  [o, o] => o        ("_&_" [36,35] 35)
```

The values `[36,35] 35` specify (using a priority grammar) the priority of conjunction (with respect to other connectives) and also that it is right-associative. In a priority grammar, priorities are assigned to occurrences of non-terminal symbols in a production, e.g. $A^{35} \Rightarrow A^{36} \wedge A^{35}$ (that corresponds to the priorities in rule `conj` above). Terminal symbols are assigned priority ∞. Derivations are as usual, with the difference that the occurrence of a non-terminal symbol can only be substituted using productions whose left-hand symbol has greater or equal priority. For example, in $A^{36} \wedge A^{35}$ only A^{35} can be substituted using $A^{35} \Rightarrow A^{36} \wedge A^{35}$ resulting in $A^{36} \wedge \boxed{A^{36} \wedge A^{35}}$. This process disambiguates the grammar and, in this example, justifies that conjunction is right-associative. Other connectives are dealt with in a similar way and the assignment of different priorities to different connectives sets its precedence. For example, the fact that conjunction has precedence over disjunction is coded by `disj::[o,o]=>o ("_|_" [31,30] 30)`.

2.2 Meta-logic and object propositional logics

Isabelle provides the logic `Pure`, a higher-order intuitionistic logic, as the framework for defining new logics. The logic `Pure` is called the *meta-logic*. Each new logic is called an *object-logic*. The deduction rules of each new logic must be coded in the meta-logic, using meta-implication. Consider, for example, a natural deduction elimination rule for conjunction:

$$\frac{\begin{array}{c}\mathcal{D}\\ \varphi_1 \wedge \varphi_2\end{array}}{\varphi_1}$$

This rule is represented by the metaformula `P&Q ==> P` where meta-implication (`==>`) has the intuitive meaning that the consequent can be proved provided that the antecedent has been proved. On the other hand, the metaformula `P==>(Q==>P&Q)` means that `P&Q` can be proved provided that both `P` and `Q` have been proved. It can also be written as `[|P;Q|]==> P&Q` and it is a representation of the introduction rule for conjunction.

2.3 Example theories

The previous concepts are enough for defining, for example, a natural deduction system for classical propositional logic, or a Hilbert system for the same logic. We begin by displaying the theory `PROPOSITIONAL`, representing a natural deduction system for propositional logic. Note that: (i) this theory is an extension of `Pure`; (ii) the type `o` of formulas has to be declared; (iii) Isabelle must recognize the object-language formulas (that have type `o`) as atomic metaformulas (that have type `prop`) and this is achieved by declaring a function (`Trueprop`) that assigns to each object formula its corresponding metaformula (in fact, itself); (iv) rules must have names; (v) abbreviations may be defined as rewrite rules, using Isabelle's built-in meta-equality rules. The function `Trueprop` has a strong semantic intuition behind it: it provides a way of internalizing Tarski's truth-predicate, so as to allow for the talk about the truth of the object-language formulas at the meta-logical level.

```
PROPOSITIONAL = Pure +
types
o
arities
o ::  logic
consts
Trueprop ::  o => prop ("(_)" 5)

(* Connectives *)
verum, falsum ::  o
neg ::  o => o ("∼ _" [40] 40)
conj ::  [o, o] => o ("_&_" [36,35] 35)
disj ::  [o, o] => o ("_|_" [31,30] 30)
imp ::  [o, o] => o ("_-->_" [26,25] 25)
iff ::  [o, o] => o ("_<->_" [26,25] 25)

rules (* Natural deduction rules *)
conjI "[| P; Q |] ==> P&Q"
conjEr "P&Q ==> P"
conjEl "P&Q ==> Q"
disjIr "P ==> P|Q"
disjIl "Q ==> P|Q"
disjE "[| P|Q; P ==> R; Q ==> R |] ==> R"
impI "(P ==> Q) ==> P-->Q"
impE "[| P-->Q; P |] ==> Q"
abs "((P --> falsum) ==> falsum) ==> P"

(* Abbreviations *)
verum_def "verum == falsum-->falsum"
neg_def "∼P == P-->falsum"
iff_def "P<->Q == (P-->Q) & (Q-->P)"
end
```

The theory `Hilbert.thy` that codes a Hilbert system for this logic can be written in a similar way. We omit the definition of the language (for simplicity we consider only implication and falsum as primitive connectives) and present a set of suitable axiom schemas and the accompanying inference rule.

```
Hilbert = Pure +
...
rules
ax1 "A --> (B --> A)"         (* Axioms *)
ax2 "(A-->(B-->C))-->((A-->B)-->(A-->C))"
ax3 "falsum-->A"
ax4 "((A-->falsum)-->falsum)-->A"
mp "[|A-->B;A|]==>B"          (* Rule *)
...
```

2.4 Using theories

In general, proofs in Isabelle are *backward proofs*, meaning that the user starts by stating the goal and applies the inference rules backwards (from the conclusion to the premises). Each premise constitutes a new subgoal to be proved. The proof ends when no further subgoals remain to be proved. The initial goal can be either a formula (e.g. `A-->(B-->A)`) or a metaformula (e.g. `[| A&B; B-->C |] ==> C`). In the first case the goal corresponds to a theorem and in the second to a derived rule of the deductive system.

Syntactic equation solving is called *unification*. The concept of unification is fundamental to understanding how rules can be applied to a (sub)goal. Rules are represented (internally) as schema metaformulas. For example the rule `conjI`, i.e. `[| P;Q |] ==> P&Q`, is, in fact, represented as `[| ?P;?Q |] ==> ?P&?Q`. The variables `?P` and `?Q` are schema variables, that is, variables that may be instantiated by any formula. This represents the fact that this rule can be applied to any formula of the form `?P&?Q`. In order to check whether a given formula `F` has that form, an equation has to be solved, namely the equation `?P&?Q=F`. The equation `?P&?Q=A&(B-->C)` has the solution `?P=A` and `?Q=B-->C`. In this way, the mechanism underlying the application of a rule to a goal corresponds to the unification of the conclusion of the rule with the goal. New subgoals appear corresponding to the premises of the rule (where appropriate schema variables have been replaced by the corresponding solution of the equation). This mechanism is called *resolution*. In the following example we establish `[| A; C |] ==> A & (B-->C)` using the theory `PROPOSITIONAL`.

```
> Goal "[| A;C |] ==> A&(B-->C)";
Level 0 (1 subgoal)
[| A; C |] ==> A&(B-->C)
1.  [| A; C |] ==> A&(B-->C)
val it = [] :  Thm.thm list

> by (resolve_tac [conjI] 1);
Level 1 (2 subgoals)
[| A; C |] ==> A&(B-->C)
1.  [| A; C |] ==> A
2.  [| A; C |] ==> B-->C
val it = () :  unit

> by (assume_tac 1);
Level 2 (1 subgoal)
[| A; C |] ==> A&(B-->C)
1.  [| A; C |] ==> B-->C
val it = () :  unit

> by (resolve_tac [impI] 1);
Level 3 (1 subgoal)
[| A; C |] ==> A&(B-->C)
1.  [| A; C; B |] ==> C
val it = () :  unit

> by (assume_tac 1);
Level 4
[| A; C |] ==> A&(B-->C)
No subgoals!
```

The command `by (resolve_tac [conjI] number)` applies the rule `conjI` to the subgoal identified by *number* (similarly for `by (resolve_tac [impI] number)`). Some subgoals do not need rules to be proved since they are premises or unifiable with premises. These are proved using `by (assume_tac number)`. In general, (sub)goals are proved *by* applying a *tactic*. The most important tactics are the previously referred tactics of resolution and assumption, the tactic `resolve_tac [rule] number` and the tactic `assume_tac number`. Moreover, `rtac` abbreviates `resolve_tac` and `atac` abbreviates `assume_tac`. Also, `br rule i` abbreviates `by (rtac [rule] i)` and `ba i` abbreviates `by (atac i)`. The variable `it` contains the value of the presently evaluated expression. We will omit the corresponding output line whenever it is not relevant.

It is worth noticing that by establishing `[| A; C |] ==> A&(B-->C)` one has established a derived rule of the current deductive system. This rule can be used in other proofs. For that purpose a name has to be assigned to it using `qed "newName";`. Afterwards one may use `by (resolve_tac [newName] number)` to apply the new rule to a subgoal. The rule will only be available in the current session. A simple way of making new rules available in all sessions is to write down their proofs in a file with extension `ML`, named after the current theory, in this case `PROPOSITIONAL.ML`.

It is also possible to prove more complex metaformulas. For example the metaformula `[|[|P==>Q;P|]==>R; P==>Q; P|]==>R` is a rule having other rules (e.g. `[|P==>Q;P|]==>R`) as premises. Such rules may be taken as primitive rules in extensions of natural deduction systems like those proposed in Schroeder-Heister (1984). Whenever the antecedent contains non-atomic metaformulas the result of `Goal` is the list of metarules associated to the premises.

```
> Goal "[|[|P==>Q;P|]==>R; P==>Q; P|]==>R";
Level 0 (1 subgoal)
R
1.  R
val it = ["[| P ==> Q; P |] ==> R" [.], "P ==> Q" [.], "P" [.]] :  Thm.thm list
```

Note that the goal to be established is simply `R` and there are three metarules, each one associated with a corresponding premise. These can also be recovered with `premises()`. Each metarule states that each premise is derivable from itself and is of the form `[Pr] Pr` (with the conclusion to the right). The previous output hides the metaformulas within `[]` (to see them use `set show_hyps;`). It is useful to assign a name to each metarule in the list, so that can be referred to later on. This can be achieved using `val [pr1,pr2,pr3] = premises();` that assigns to the first element of the list the name `pr1`, to the second the name `pr2` and to the third `pr3`. This can also be achieved directly by using

```
> val [pr1,pr2,pr3]=Goal "[|[|P==>Q;P|]==>R; P==>Q; P|]==>R";
Level 0 (1 subgoal)
R
1.  R
val pr1 = "[| P ==> Q; P |] ==> R" [.] :  Thm.thm
val pr2 = "P ==> Q" [.] :  Thm.thm
val pr3 = "P" [.] :  Thm.thm
```

The proof is as expected, noting that the names of the premises are used.

```
> br pr1 1;
Level 1 (2 subgoals)
R
1.  P ==> Q
```

```
2.  P
> br pr2 1;
Level 2 (2 subgoals)
R
1.  P ==> P
2.  P
> ba 1;
Level 3 (1 subgoal)
R
1.  P
> br pr3 1;
Level 4
R
No subgoals!
```

To establish a goal involving abbreviations, e.g. ∼∼P<->P, the abbreviated expressions must be rewritten (using the convenient rewrite rules, in this case `neg_def` and `iff_def`). One way of doing this is by using the command `Goalw [def1,def2,...] metaformula` that rewrites the abbreviations in `metaformula` using the definitions in the argument list:

```
> Goalw [neg_def,iff_def] "∼∼P <-> P";
Level 0 (1 subgoal)
∼∼ P<->P
1.  (((P-->falsum)-->falsum)-->P)&(P-->(P-->falsum)-->falsum)
```

The proof can now be done as usual.

2.5 Sequents and automated deduction

Sequent calculus can provide an easy decision procedure for validity (in classical propositional logic). All one has to do is to repeatedly apply the rules to the goal sequent.

Sequents are represented in Isabelle by pairs of lists of formulas. For example,

```
A, B-->C |- D, E&F
```

denotes a sequent with antecedent `A,B-->C` and consequent `D, E&F`. Variables prefixed by `$` are list variables. The sequent

```
"$H, P, $G |- $E, P, $F"
```

represents an axiom schema and

```
"[| $H, $G |- $E, P; $H, Q, $G |- $E |] ==> $H, P-->Q, $G |- $E"
```

codes the left-introduction rule for implication. Lists and sequents are provided by the theory `Sequents.thy` available in Isabelle's distribution. The following theory `SPROP.thy` defines the sequent calculus for classical propositional logic by providing the syntax of formulas and the sequent rules.

```
SPROP = Sequents +
consts
Trueprop ::  "two_seqi"
"@Trueprop" ::  "two_seqe" ("((_)/ |- (_))" [6,6] 5)
```

```
(* Connectives *)
falsum,verum ::  o
neg ::  o => o ("∼_" [40] 40)
conj ::  [o, o] => o ("_&_" [36,35] 35)
disj ::  [o, o] => o ("_|_" [31,30] 30)
imp ::  [o, o] => o ("_-->_" [26,25] 25)
iff ::  [o, o] => o ("_<->_" [26,25] 25)

rules (* Sequent Rules*)
axS "$H, P, $G |- $E, P, $F"
falsumL "$H, falsum, $G |- $E"
conjR "[| $H|- $E, P, $F; $H|- $E, Q, $F |] ==> $H|- $E, P&Q, $F"
conjL "$H, P, Q, $G |- $E ==> $H, P & Q, $G |- $E"
disjR "$H |- $E, P, Q, $F ==> $H |- $E, P|Q, $F"
disjL "[| $H, P, $G |- $E; $H, Q, $G |- $E |] ==> $H, P|Q, $G |- $E"
impR "$H, P |- $E, Q, $F ==> $H |- $E, P-->Q, $F"
impL "[| $H,$G |- $E,P; $H, Q, $G |- $E |] ==> $H, P-->Q, $G |- $E"

(* Abbreviations *)
verum_def "verum == falsum-->falsum"
iff_def "P<->Q == (P-->Q) & (Q-->P)"
neg_def "∼P == P-->falsum"
end

ML
val parse_translation = [("@Trueprop",Sequents.two_seq_tr "Trueprop")];
val print_translation = [("Trueprop",Sequents.two_seq_tr' "@Trueprop")];
```

The reason for the use of both `Trueprop` and `@Trueprop` is that there are, in fact, two different representations of lists, the one referred above (and called the external representation) and a functional representation of lists (called the internal representation). `@Trueprop(A,B)`, also written `A|-B`, is a sequent where the lists involved are written in the external representation. `Trueprop(alpha,beta)` is the sequent where the lists involved are written in the internal representation. The two lines in `ML` code at the end of the above theory provide functions that translate between the two representations of sequents.

In order to use this theory (with `use_thy`) the system must load the theory `Sequents`:

```
> isabelle Sequents
...
use_thy "SPROP";
```

Before illustrating the use of this theory with an example, we present the notion of *tactical*. Tacticals are forms of combining tactics. Useful tacticals are the iterative combinator of tactics `REPEAT` and the alternative combinator `ORELSE`. The tactic `REPEAT` *`tactic`* corresponds to the repeated application of the argument tactic until it fails. The tactic *`tactic1`* `ORELSE` *`tactic2`* corresponds to *`tactic1`* alone when the application of *`tactic1`* succeeds. Otherwise it corresponds to *`tactic2`* alone. Tacticals are useful in interactive proofs and they are the fundamental tools in developing automatic proving techniques.

The proof of `|-(A-->(B-->C))-->((A-->B)-->(A-->C))` begins with three applications of resolution with `impR` to the first subgoal. These three steps may be simplified by using `REPEAT (rtac impR 1)`.

```
> Goal "|- (A-->(B-->C))-->((A-->B)-->(A-->C))";
Level 0 (1 subgoal)
|- (A --> B --> C) --> (A --> B) --> A --> C
1.  |- (A --> B --> C) --> (A --> B) --> A --> C

> by (REPEAT (rtac impR 1));
Level 1 (1 subgoal)
|- (A --> B --> C) --> (A --> B) --> A --> C
1.  A --> B --> C, A --> B, A |- C
```

The next two steps are easy to follow.

```
> br impL 1;
Level 2 (2 subgoals)
|- (A --> B --> C) --> (A --> B) --> A --> C
1.  A --> B, A |- C, A
2.  B --> C, A --> B, A |- C

> br axS 1;
Level 3 (1 subgoal)
|- (A --> B --> C) --> (A --> B) --> A --> C
1.  B --> C, A --> B, A |- C
```

The final steps are by resolution with either `axS` or `impL`.

```
> by (REPEAT ((rtac axS 1) ORELSE (rtac impE 1)));
Level 4
|- (A --> B --> C) --> (A --> B) --> A --> C
No subgoals!
```

It may be helpful to exhibit parentheses making formulas more readable. The command `set show_brackets;` sets the corresponding flag to `true`.

```
> set show_brackets; Goal "|- (A-->(B-->C))-->((A-->B)-->(A-->C))";
Level 0 (1 subgoal)
( |- ((A --> (B --> C)) --> ((A --> B) --> (A --> C))))
1.  ( |- ((A --> (B --> C)) --> ((A --> B) --> (A --> C))))
```

As referred above, sequent calculus provides a decision procedure for validity (in classical propositional logic). In fact, each (backward) application of the rules decreases the complexity of the formulas involved in the sequent. In the end, subgoals consist of axioms or sequents without connectives that encode a counter-example for the original goal.

With this decision procedure in mind, one can easily write down a tactic that succeeds whenever the original goal is a valid sequent and fails otherwise. First one has to define a tactic that corresponds to the application of *some* rule. This tactic should try the rules in some order until it finds one that unifies. In this deductive system, the order of application of rules is irrelevant. It is more efficient to prefer rules that eliminate subgoals or introduce less new subgoals over other rules. One possible choice is to try application of rules in this order: `axS,falsumL,impR,conjL,disjR,impL,conjR,disjL`. The tactic

```
(rtac axS 1) ORELSE ... ORELSE (rtac disjL 1)
```

applies the axiom or some rule to the first subgoal and fails only if none is applicable. Repeated application of this tactic is achieved by

```
REPEAT ((rtac axS 1) ORELSE ... ORELSE (rtac disjL 1)).
```

Given a valid sequent, the application of this tactic succeeds, since the first subgoal is repeatedly simplified until an instance of the axiom `axS` is obtained and eliminated. At this point, if there are further subgoals, the second becomes the first and this process goes on until no subgoal is left. If the original sequent is not valid, the tactic fails in the first subgoal consisting of a sequent without connectives that is not an axiom. In this case more subgoals can be left unworked. In the following example the original sequent is valid. The above tactic is given the name `tacSeq1`.

```
> val tacSeq1= REPEAT ((rtac axS 1) ORELSE ... ORELSE (rtac disjL 1));
> Goal "|- (A --> B --> C) --> (A --> B) --> A --> C";
Level 0 (1 subgoal)
|- (A --> B --> C) --> (A --> B) --> A --> C
1.  |- (A --> B --> C) --> (A --> B) --> A --> C
> by tacSeq1;
Level 1
|- (A --> B --> C) --> (A --> B) --> A --> C
No subgoals!
```

The next example corresponds to a sequent that is not valid.

```
> Goal "|- (A --> D --> C) --> (A --> B) --> A --> E";
Level 0 (1 subgoal)
|- (A --> D --> C) --> (A --> B) --> A --> E
1.  |- (A --> D --> C) --> (A --> B) --> A --> E
> by tacSeq1;
Level 1 (2 subgoals)
|- (A --> D --> C) --> (A --> B) --> A --> E
1.  B, A |- E, D
2.  C, A --> B, A |- E
```

It is not difficult to improve the previous tactic to also work out the remaining subgoals. In this way, for invalid sequents, the remaining subgoals will encode the counter-examples. For that purpose one simply has to rewrite the previous tactic in such a way that it can be applied to subgoals other than the first. The first step is to define a function that, to each subgoal `i` associates the tactic `(rtac axS i) ORELSE ... ORELSE (rtac disjL i)`. This is achieved by `val tacSeqfun = fn i => (rtac axS i) ORELSE ... ORELSE (rtac disjL i).`

In the previous command the intended function (`fn i => (rtac axS i) ...`) is given the name `tacSeqfun`. Finally, the improved tactic is given by `REPEAT_FIRST tacSeqfun` that repeatedly applies `tacSeqfun` to each subgoal (starting with the first). When no further applications of `tacSeqfun` are possible, the next subgoal is worked out. For convenience, the name `tacSeq` is given to `REPEAT_FIRST tacSeqfun`. Note that the behavior of the new tactic on valid sequents is the same as the old tactic. We illustrate the new tactic using the previous (invalid) sequent.

```
> Goal "|- (A --> D --> C) --> (A --> B) --> A --> E";
Level 0 (1 subgoal)
|- (A --> D --> C) --> (A --> B) --> A --> E
1.  |- (A --> D --> C) --> (A --> B) --> A --> E
```

```
> by tacSeq;
Level 1 (2 subgoals)
|- (A --> D --> C) --> (A --> B) --> A --> E
1.  B, A |- E, D
2.  C, B, A |- E
```

2.6 Meta-universal quantification and first-order logic

Until now, different deductive systems for classical propositional logic have been presented. For the predicative version one needs terms, predicates and quantifiers. Terms belong to a type different from that of formulas and can be built in the usual way using variables and function symbols. Both formulas and terms are λ-calculus terms, i.e. λ-abstraction and λ-application are also available. Predicates are represented as functions that, to each term, associate a formula. For example, $\lambda x.P(x)$ associates to each term x the formula $P(x)$. The λ-term $\lambda x.P(x)$ is written `%x. P(x)`. Quantifiers are represented as functions that associate a formula to a λ-term. For example, `all(%x. P(x))` is a universally quantified formula. An alternative syntax (and a priority value) can be stated by `(binder "ALL " 10)`. This means that `ALL x. P(x)` will abbreviate `all(%x. P(x))`.

For the purpose of defining propositional systems only meta-implication was needed. In the predicative context the introduction and elimination rules for quantifiers need the meta-universal quantification (of the built-in logic `Pure`). The metaformula `!!x. P(x)` means that `P(t)` is true for any arbitrary term `t`. In particular, the generalization (introduction) rule is represented by the metaformula `(!!x. P(x))==>ALL x. P(x)` and is represented internally by `(!!x. ?P(x))==>ALL x. ?P(x)`. The rule can be read as follows: In order to prove `ALL x. P(x)` one has to prove `P(x)` for arbitrary `x`. The elimination rule for the universal quantifier is `(ALL x. P(x))==>P(t)` and is represented internally by `(ALL x. ?P(x))==>?P(?t)`. The substitution of `x` by `t` in `P` corresponds, in the context of λ-calculus, to λ-application. Possible problems related to substitutions are dealt with by application of α-reduction (variable renaming) and β-reduction (function application).

Next we present the theory representing classical first-order logic.

```
CLASSIC = PROPOSITIONAL +
classes
term < logic

consts (* Quantifier functions *)
all ::  ('a::term => o) => o (binder "ALL " 10)
ex ::  ('a::term => o) => o (binder "EX " 10)

rules (* Quantifiers *)
allI "(!!x. P(x)) ==> (ALL x. P(x))"
allE "(ALL x. P(x)) ==> P(t)"
exI "P(t) ==> (EX x. P(x))"
exE "[| EX x. P(x); !!x. (P(x) ==> R) |] ==> R"
end
```

The theory uses the connectives and rules of classical propositional logic and adds quantified formulas and their rules. It is noteworthy that `term` is not a type but a class to which any term type must belong. For instance, noting that `'a` is a variable ranging over types, the function `all :: ('a::term=>o)=>o` is a function that associates a formula to a predicate (on its turn, a function that associates a formula to a term). Finally, in rule `exE`, the fact that `R` does not depend on `x` codes the condition that `x` cannot occur free in `R`. In the next example we want to establish

`[| ALL x.Q(f(x)); ALL x.(P(x)-->Q(x)) |] ==> ALL x.Q(f(g(x)))` using the theory `CLASSIC`. It seems easy, at first glance, since the conclusion follows from the first premise. We first apply rule `allI` and then rule `allE`.

```
> Goal "[| ALL x.Q(f(x)); ALL x.(P(x)-->Q(x)) |] ==> ALL x.Q(f(g(x)))";
Level 0 (1 subgoal)
[|ALL x.Q(f(x)); ALL x.P(x)-->Q(x)|] ==> ALL x.Q(f(g(x)))
1.  [| ALL x.Q(f(x)); ALL x.P(x)-->Q(x) |] ==> ALL x.Q(f(g(x)))
> br allI 1;
Level 1 (1 subgoal)
[| ALL x.Q(f(x)); ALL x.P(x)-->Q(x) |] ==> ALL x.Q(f(g(x)))
1.  !!y.  [| ALL x.Q(f(x)); ALL x.P(x)-->Q(x) |] ==> Q(f(g(y)))
> br allE 1;
Level 2 (1 subgoal)
[| ALL x.Q(f(x)); ALL x.P(x)-->Q(x) |] ==> ALL x.Q(f(g(x)))
1.  !!y.[| ALL x.Q(f(x)); ALL x.P(x)-->Q(x) |] ==> ALL x.Q(x)
> ba 1;
** by:  tactic failed Exception- ERROR raised
```

When applying `allE`, the unification of its conclusion `%P.%t.P(t)` with `Q(f(g(y)))` has more than one solution. Isabelle displays first the result of β-reducing `%P.%t.P(t)` into `%t.Q(t)` and the latter into `Q(f(g(y)))`. In this case, however, this is not the suitable solution since `ALL x.Q(x)` does not unify with the hypothesis `ALL x.Q(f(x))`. Isabelle's unifying procedure is clever in producing all possible solutions (and in first-order logic there can be an infinite number of them) by using lazy evaluation (evaluation on demand). To demand Isabelle to consider another solution we use the command `back()`.

```
> back();
Level 2 (1 subgoal)
[| ALL x.Q(f(x)); ALL x.P(x)-->Q(x) |] ==> ALL x.Q(f(g(x)))
1.  !!y.[| ALL x.Q(f(x)); ALL x.P(x)-->Q(x) |] ==> ALL x.Q(f(x))
> ba 1;
Level 3
[| ALL x.Q(f(x)); ALL x.P(x)-->Q(x) |] ==> ALL x.Q(f(g(x)))
No subgoals!
```

The β-reduction of `%P.%t.P(t)` into `%t.Q(f(t))` and the latter into `Q(f(g(y))` is now considered. The rest of the proof is straightforward.

An alternative approach uses the tactic `res_inst_tac [("`*v1*`","`*f1*`"),...,"`*vn*`","`*fn*`")] `*rule number*, that corresponds to the use of resolution with *rule* to the subgoal identified by *number* forcing the schema variable *v1* to be instantiated into *f1* ... and *vn* into *fn*. In this case we can use `res_inst_tac` just after the application of the rule `allI`:

```
...
> by (res_inst_tac [("t","g(y)")] allE 1);
Level 2 (1 subgoal)
[| ALL x.Q(f(x)); ALL x.P(x)-->Q(x) |] ==> ALL x.Q(f(g(x)))
1.  !!y.[| ALL x.Q(f(x)); ALL x.P(x)-->Q(x) |] ==> ALL x.Q(f(x))
```

The last step of the proof is as before, using `ba 1`.

2.7 Labeled modal logics

The theories previously described are closely related to similar theories provided by Isabelle's distribution. Although theories for modal logics (including sequent calculus) are also available in that distribution we choose to present theories based on labeled deduction, as in Viganò (2000). Labeled deduction internalizes semantic notions in the object-language and deductive system. Deductive systems for normal modal logics can thus become modular in the sense that they are obtained simply by adding suitable rules (representing the properties of the accessibility relation) to the deductive system of modal logic K. With labeled deduction as the choice of formalization for our modal languages we can escape the traditional difficulties that surface when the corresponding logical systems are represented, as advanced in (Paulson, 1990).

Formulas are obtained by prefixing a term (usually a variable) to a modal formula. For instance, $x : (\Box\Diamond\phi \rightarrow \psi)$ is a formula. Informally, this formula means that $(\Box\Diamond\phi \rightarrow \psi)$ holds in the world denoted by x. These are called *generalized modal formulas*. Formulas $x\,Rel\,y$ state that the world denoted by y is accessible from the world denoted by x and are called *relational modal formulas*. More generally, they may also involve terms as in $t_1\,Rel\,t_2$. By the present construction, strings such as $(x : \phi) \vee (y : \psi)$ and $(\forall x)(x\,Rel\,x)$ are *not* formulas of K —but they *will* be formulas of the hybrid logic presented in the next section.

The theory for the minimal normal modal logic K is presented below. There are three types of syntactic entities involved in that theory: `tm` for terms (that prefix modal formulas), `sbf` for modal (sub)formulas and `o` for formulas. The latter are either generalized formulas (built with `gf`) or relational formulas (built with `rf`). The rules for $\Box$ (`[]`) and $\Diamond$ (`<>`) are similar to the rules for the universal and the existential quantifiers. For instance, the rule `boxI` (introduction of $\Box$) asserts that `x:[]P` follows from the proof that `P` holds in `y` for arbitrary `y` related to `x`. We omit most of the propositional fragment as straightforward.

```
K = Pure +
classes
term < logic
types
tm (* the type of terms for labels *)
sbf (* the type of modal subformulas *)
o (* the type of formulas *)
arities
tm ::  term
sbf ::  logic
o ::  logic
consts (* Formula Generators *)
gf ::  [tm,sbf] => o ("_:_" [0,0] 10)
rf ::  [tm,tm] => o ("_ Rel _" [0,0] 10)
Trueprop ::  o => prop ("(_)" 5)
(* Modal (sub)formulas *)
and ::  [sbf, sbf] => sbf ("_&_" [36,35] 35)
...
box ::  sbf => sbf ("[]_" [40] 40)
dia ::  sbf => sbf ("<>_" [40] 40)
rules
(* Propositional *)
conjI "[| x:P; x:Q |] ==> x:P&Q"
```

```
...
(* Modal Operators *)
boxI "(!!y. (x Rel y ==> y:P)) ==> x:[]P"
boxE "[| x:[]P; x Rel y|] ==> y:P"
diaI "[| x Rel y; y:P|] ==> x:<>P"
diaE "[| x:<>P; (!!y. [|x Rel y; y:P|]==> z:Q)|]==> z:Q"
(* Abbreviations *)
verum_def "x:verum == x:falsum-->falsum"
neg_def "x:~P == x:P-->falsum"
iff_def "x:P<->Q == x:(P-->Q) & (Q-->P)"
```

The theory for KT can be obtained from the theory for K by adding the axiom (a rule with no premises) of reflexivity (`x Rel x`). The theory for KB adds to the theory for K the rule of symmetry (`x Rel y ==> y Rel x`), and the theory for $S5$ adds to the theory for K both the axiom of reflexivity and the rule of symmetry, together with the rule of transitivity (`[|x Rel y; y Rel z|] ==> x Rel z`). Many other modal systems can be formalized in a similar way. To define more complex systems such as the normal modal logic of confluence, one might make use of terms in the labels. In this case, we add to the theory for K a Skolem function as a new constructor of the form `h :: [tm,tm,tm] => tm`, and we add also the rules `[|x Rel y; x Rel z|] ==> y Rel h(x,y,z)` and `[|x Rel y; x Rel z|] ==> z Rel h(x,y,z)`.

3 A hybrid logic

The issues addressed before are part of the syllabus of a course in logic for undergraduate students in Mathematics and in Computer Science, including the logics previously referred. The final assignment included the definition of a theory in Isabelle representing the hybrid logic described in the following. To learn more about hybrid languages, see Blackburn and Seligman (1998).

> The hybrid logic to be defined, using only `Pure`, is a labeled modal logic where quantification over worlds is allowed. Formulas like $\forall x(x\ Rel\ x) \rightarrow \forall x(x : \Box\psi \rightarrow x : \psi)$ can be written in the logic. In this logic the atomic formulas are generalized modal formulas and relational modal formulas, defined as in section 2.7. However, falsum, negation, disjunction and $\Box$ are the only connectives herein considered as primitive for the construction of the atomic formulas (verum, conjunction, implication, equivalence and $\Diamond$ are taken as abbreviations). The set of formulas is obtained by closing the latter set of atomic formulas with negation, disjunction and the universal quantifier (conjunction, implication, equivalence and the existential quantifier are taken as abbreviations). In particular, strings such as $(\forall x)\phi$ and $\Box(x : \phi)$ will *not* constitute formulas of the present language. Note that the propositional connectives occur in formulas and also within atomic formulas. For example, in the atomic formula $x : (\phi \vee \psi)$, disjunction involves two modal subformulas whereas in the (non-atomic) formula $(x : \phi) \vee (x : \psi)$ it involves two formulas of the hybrid logic. The intended interpretation assigns to both formulas the same meaning (justification: $(\phi \vee \psi)$ is true at the world x if and only if either ϕ is true at x or ψ is true at x) and the deductive system must be able to prove them equivalent. Moreover, the derived rules for all symbols defined by abbreviation should be made available by the student.

Most students were able to successfully deliver the corresponding Isabelle theory. One possible solution, using natural deduction, follows. In this solution we use *polymorphism* to define

the connectives that are common to the two different types of 'formulas', the modal subformulas and the formulas themselves. For example, `not::('a::logic) => 'a` defines a polymorphic operation `not` that takes any type of class `logic` and returns a value of that type. In our case we have two possible types, namely `sbf` (for modal subformulas) and `o` (for formulas). In this way `not::('a::logic) => 'a` simultaneously defines the two functions (connectives) `not::sbf => sbf` and `not::o => o`. In general, one has to provide introduction and elimination rules for both connectives. However, our solution provides rules only for `not::o => o` and provides further rules to transform subformulas into formulas and vice-versa (when applicable). For example, in order to conclude `x:∼P` one may first conclude `∼(x:P)` and then apply the rule that transforms it into `x:∼P`. The new rules for negation are `negOut "x:(∼P) ==> ∼(x:P)"` and `negIn "∼(x:P) ==> x:(∼P)"`. By using such transformation rules the introduction and elimination rules for `not::sbf => sbf` can be derived in this system. Similar considerations apply to other connectives.

```
HYBRID = Pure +
classes
term < logic
default
term
types
tm (* the type of terms for labels *)
sbf (* the type of modal subformulas *)
o (* the type of formulas *)
arities
tm ::  term
sbf ::  logic
o ::  logic
consts
(* Formula Generators *)
labf ::  [tm,sbf] => o ("_:_" [0,0] 45)
relf ::  [tm,tm] => o ("_ Rel _" [0,0] 45)
Trueprop ::  o => prop ("(_)" 5)
(* For modal subformulas only *)
verum, falsum ::  sbf
box ::  sbf => sbf ("[]_" [50] 50)
dia ::  sbf => sbf ("<>_" [50] 50)
(* Quantifiers (for formulas only) *)
all ::  ('a => o) => o (binder "ALL " 10)
ex ::  ('a => o) => o (binder "EX " 10)
(* Connectives for both modal subformulas and formulas *)
not ::  'a::logic => 'a ("∼_" [40] 40)
and ::  ['a::logic, 'a] => 'a ("_&_" [36,35] 35)
or ::  ['a::logic, 'a] => 'a ("_|_" [31,30] 30)
imp ::  ['a::logic, 'a] => 'a ("_-->_" [26,25] 25)
iff ::  ['a::logic, 'a] => 'a ("_<->_" [26,25] 25)
rules
(* Connectives *)
abs "(∼P ==> y:falsum) ==> P"
negE "[| ∼P; P |] ==> Q"
negI "(P ==> y:falsum) ==> ∼P"
```

```
negOut "x:(∼P) ==> ∼(x:P)"
negIn "∼(x:P) ==> x:(∼P)"
disjIr "P ==> P|Q"
disjIl "Q ==> P|Q"
disjE "[| P|Q; P ==> R; Q ==> R |] ==> R"
disjOut "x:(P|Q) ==> (x:P)|(x:Q)"
disjIn "(x:P)|(x:Q) ==> x:(P|Q)"
(* Modal Operators *)
boxI "(!!y. (x Rel y ==> y:P)) ==> x:[]P"
boxE "[| x:[]P; x Rel y |] ==> y:P"
(* Quantifier *)
allI "(!!y. P(y)) ==> (ALL x. P(x))"
allE "(ALL x. P(x)) ==> P(z)"
(* Abbreviations *)
verum_def "verum == falsum-->falsum"
conj_def "P&Q == ∼((∼P)|(∼Q))"
imp_def "P-->Q == (∼P)|Q"
iff_def "P<->Q == ∼(∼((∼P)|Q) | ∼(P|(∼Q)))"
dia_def "<>P == ∼([]∼P)"
ex_def "EX x. P(x) == ∼(ALL x. ∼P(x))"
end
```

Note a difference between the abbreviations of the theory `HYBRID` and those of the theory `K`, in section 2.7. In the present theory, a connective like `<->` can be used to generate both atomic formulas (as in `P<->Q`) and formulas of the above hybrid logic (as in `x Rel y <-> z:[]P`). Therefore, the rewrite rules must now be laid down so as to apply to both situations.

We illustrate the theory with some examples. In the first one we prove the derived rule for the elimination of disjunction of modal subformulas. Recall from section 2.4 that when the antecedent contains non-atomic metaformulas it is convenient to associate a name to each of the elements of the result of `Goal`.

```
> val [mt1,mt2,mt3]= Goal "[|x:(P|Q); x:P==>y:R; x:Q==>y:R|]==> y:R";
Level 0 (1 subgoal)
y:R
1.  y:R
val mt1 = "x:P|Q" [.] :  Thm.thm
val mt2 = "x:P ==> y:R" [.] :  Thm.thm
val mt3 = "x:Q ==> y:R" [.] :  Thm.thm
> br disjE 1;
Level 1 (3 subgoals)
y:R
1.  ?P|?Q
2.  ?P ==> y:R
3.  ?Q ==> y:R
> br mt2 2; br mt3 3; ba 2; br disjOut 1; br mt1 1;
...
No subgoals!
> qed "disjmE";
val disjmE = "[| ?x:?P|?Q; ?x:?P ==> ?y:?R; ?x:?Q ==> ?y:?R |] ==> ?y:?R"
```

Other examples follow. Only the used rules are shown. In the very last example, we use the tactic `rotate_tac` *number steps*, that corresponds to a left permutation of the subgoal identified by *number* by a number of *steps*.

```
Goal "(x:(A&B))<->(x:A)&(x:B)";
br eqI 1; br conjOut 1; ba 1; br conjIn 1; ba 1;

Goal "(ALL x. (x:P))-->(ALL x. (x:[][]P))";
br impI 1; br allI 1; br boxI 1; br boxI 1; br allE 1; ba 1;

Goal "ALL x.  ((x Rel x) --> ((x:[]P)-->(x:<>P)))";
br allI 1; br impI 1; br impI 1; br diaI 1; ba 1; br boxE 1; ba 1; ba 1;

Goal "(ALL x y. ((x Rel y) --> (y Rel x)))-->(ALL x. ((x:P)-->(x:[]<>P)))";
br impI 1; br allI 1; br impI 1; br boxI 1; br diaI 1; br impE 1;
br allE 1; br allE 1; by (REPEAT (atac 1));

Goal "(ALL x y z. ((x Rel y) & (y Rel z) --> (x Rel z)))-->(ALL x. (x:([]P-->[][]P)))";
br impI 1; br allI 1; br impIn 1; br impI 1; br boxI 1; br boxI 1; br boxE 1;
ba 1; br impE 1; br conjI 2; by (rotate_tac 2 2); by (rotate_tac 3 3); ba 2; ba 2;
by (res_inst_tac [("z","yb")] allE 1); by (res_inst_tac [("z","ya")] allE 1);
by (res_inst_tac [("z","y")] allE 1); ba 1;
```

4 Concluding remarks

The system Isabelle is an appropriate tool for the definition and experimentation of new logics. The fundamental concepts underlying the definition of new Isabelle theories and their use are not too difficult to master. In this way, users can easily start to use the system for their own purposes. We have presented the most important concepts, illustrated by examples that undergraduate logic students are able to develop.

Obviously, there are other important features of Isabelle not described herein. There are many logics and useful tactics already provided by the distribution. There is a growing open collection of Isabelle proof libraries and examples at `http://afp.sourceforge.net/`. Moreover, proofs can be developed in more user-friendly environments, including management of theories and standard graphical notation for connectives and operators (see `http://proofgeneral.inf.ed.ac.uk/` for the ProofGeneral tool and its support for Isabelle). The language Isar provides syntactic sugar for developing proofs and is now becoming standard. Isar's manual and other relevant documentation are freely available on-line at `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/docs.html`.

Acknowledgements

This work was partially supported by FCT and FEDER, namely, via the Project FibLog POCTI/MAT/372 39/2001 of CLC (Centro de Lógica e Computação). The third author is partially supported by FCT grant SFRH / BD / 8825 / 2002.

References

Blackburn, P. and Seligman, J.: 1998, What are hybrid languages?, *in* M. Kracht, M. de Rijke and H. Wansing (eds), *Advances in Modal Logic, Volume 1*, CSLI Publications, Stanford, California, pp. 41–62.

Gleick, J.: 1987, *Chaos, Making a New Science*, Penguin Books, New York, NY.

Nipkow, T., Paulson, L. C. and Wenzel, M.: 2002, *Isabelle/HOL*, Vol. 2283 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin. A proof assistant for higher-order logic.

Paulson, L. C.: 1990, Isabelle: The next 700 theorem provers, *in* P. Odifreddi (ed.), *Logic and Computer Science*, Academic Press, pp. 361–386.

Paulson, L. C.: 1996, *ML for the Working Programmer*, Cambridge Univ. Press. 2nd edition.

Schroeder-Heister, P.: 1984, A natural extension of natural deduction, *The Journal of Symbolic Logic* **49**(4), 1284–1300.

Troelstra, A. and Schwichtenberg, H.: 1996, *Basic Proof Theory*, Cambridge Univ. Press.

Viganò, L.: 2000, *Labelled Non-Classical Logics*, Kluwer Academic Publishers.