Implementing a Deduction System for a Guarded Commands Procedural Language Using a Generic Theorem Prover

Pedro Quaresma de Almeida
Departamento de Matemática da Universidade de Coimbra,
3000 COIMBRA — PORTUGAL
e-mail: pedro@mat.uc.pt phone: +351-39-28097

August 2, 1994

Abstract

Two main issues have to be dealt with when buiding formal proofs with any form of computational system: the possibility of doing formal proofs in different deduction systems and the possibility of different user/program interaction strategies.

If we want to emphasize efficiency we will adopt a system based in a predetermined deduction system and an *automatic theorem proving* [6] style of doing proofs.

If we want a flexible "workbench" we will adopt a *generic theorem prover i.e.* a system were one can implement different deduction systems and were one can adopt different strategies of doing proofs.

We discuss here the second approach, analysing two different systems, Isabelle [14, 15, 16] and 2OBJ [17].

Finally we present an example of a deduction system for an axiomatic definition of a guarded commands procedural language (GCPL) [1, 3, 7, 10, 11, 13], showing how the two systems implement this deduction system and their capability of conducting proofs within this new deduction system.

1 Introduction

Because many different forms of logical systems are applicable to computer science it is highly desirable to work with a flexible "workbench". Such a computacional system must support for a wide variety of logical systems.

The computational system must also provide support for different styles of conducting proofs in order to choose the best approach for a particular problem. It should support either a proof style, which we will designate by *automatic theorem proving* [6], in which the interaction user/program is minimal, or a proof style, which we will designate by *proof checking* [6] with a greater degree of user/program interaction.

In the first approach, typically, a generic proof strategy (*e.g.* resolution principle, see [2, 4]) is implemented and the user is confined to, submitting some axioms and the formula to be proved, then adjusting some parameters of the strategy to control its method of search, and eventually, responding to requests for help from the system during its search for a proof.

In the second approach the user provides an alleged proof, step by step, and the program checks each step. In the most extreme form of proof checking each step consists in the application of a primitive rule of inference althought, and in the following of the work of Michael Gordon *et al.* [6], many proof checking systems allow complex inferences in the form of *tactics and tacticals*[1], they even allow the programming of strategies of the first style described above.

We will now describe, briefly, two generic theorem provers suitable for the description of a logical systemrepresenting a guarded commands procedural language and the corresponding deduction system in both programs, and finally we will discuss the results of some proofs conducted with both programs.

2 Defining a Deduction System

To define a deduction system we must be able of defining the syntax of a logical system: formula-, term- and rule-schemes, the management of binding operators and substitution. We must also be able of defining a representation for formal proofs and to provide the mechanisms associated with proof checking and proof construction [9].

The systems Isabelle and 2OBJ are two computational systems that satisfy those requirements.

The two systems are based in generic programming languages, *Standard ML* [8] in the case of Isabelle, and *OBJ3* [5] in the case of 2OBJ. They both support the implementation of logical systems and corresponding deduction systems and the programming of different styles of conducting proofs, *i.e.* they both are *generic theorem provers.*

We will now briefly describe the two systems.

2.1 Isabelle

Isabelle [14, 15, 16] is a generic theorem prover supporting formal proofs in a variety of logics. It has already been instantiated to support reasoning in the following object-logics [16]: first-order logic, intuitionistic and classical; higher-order logic; Zermello-Fraenkel set theory; Constructive type theory; Classical first-order sequent calculus, LK; modal logics T, S4 and S43; Logic for computable functions and Barendrgt λ-cube.

Isabelle is based in Standard ML, it represents syntax in a polymorphic typed λ-calculus and its meta-logic is the higher-order logic.

[1] By analogy with functions and functionals.

Isabelle's proof procedures go from single-step proof construction (proof checking) to a complete automatization of the proof (automatic theorem proving).

Isabelle's user interface is character-based. It does not display the proof-tree.

2.1.1 Syntax

In Isabelle the syntax of a deduction system is encoded as a *Theory*.

A theory definition in Isabelle extends one or more existing theories (if we want to define an entire logic we must use the *Pure* theory containing only Isabelle's meta-logic), with new classes, types arities, consts and rules.

The classes, types and arities declarations define the syntatic classes of the logical system. We must declare an identifier (a type) for each syntactic class and we must say to which class it belongs. Isabelle's classes, which are partially ordered by the sub-class relation, are used to control the polymorphism.

The consts (constants) declarations define the symbols of the logical language. For each one of them we must declare an identifier (a constant or an n-curried function) and specify its arity and co-arity.

The rules declarations define the inference rules. They are written, in a simplified form, using the Isabelle's meta-level connectives and its concrete form depends on the Isabelle's chosen logical subsystem.

Isabelle's theories are written in a meta-syntax that is automatically translated in Isabelle's internal syntax, which is a typed λ-calculus. The variable binding and substitution are performed by λ-abstraction and β-reduction.

Isabelle's meta-logic is intuitionistic higher-order logic. The meta-level connectives are implication, the universal quantifier and equality: the implication expresses logical entailment; the quantification expresses generality in rules and axioms schemes; the equality expresses definition.

To relate the meta-level with the object-logic level Isabelle declares the constant `Trueprop :: o => prop`, reading `Trueprop(P)` as *"P is true at object-level"*.

2.1.2 Proof construction

Proof construction in Isabelle is done using tactics, which perform primitive refinement, and tacticals wich combine tactics.

Isabelle principal proof method (tactic) is resolution. It works in a backward proof by unifying a goal with the conclusion of a rule, whose premises become new subgoals, and in a forward proof by unifying theorems with premises of a rule, deriving a new theorem.

In order to have a full automatization of proofs Isabelle provides the tacticals depth-first search, breath-first search and best-first search. In the tactical best-first search an heuristic function selects the best state to explore.

If we want to program simpler tactics we can do it using the basic tacticals *THEN*, *ORELSE*, and *REPEAT*.

2.2 2OBJ

The 2OBJ [17] is a generic theorem prover that can be programmed to prove theorems in any desired logical system. As a theorem prover 2OBJ is a library of modules in OBJ3 that support the encodings of logics and the definition and execution of proof tactics for such encoded logics. The logical system must be a variant of a sequent calculus. 2OBJ has already instantiated `fopc.obj` (*first-order predicate calculus*).

2OBJ encodes a given logical system as an abstract data type within OBJ3. 2OBJ meta-logic is equational logic.

2OBJ's proof procedures are single-step proof construction (proof checking) but, via tactics and tacticals, more complex proof procedures can be programed. It does not provide unification nor any generic proof strategy.

The X2OBJ tool is a X–Windows tool supporting interactive editing of proofs in the encoded logics. It displays the proof-tree.

2.2.1 Syntax

In 2OBJ the syntax of an deduction system is encoded as an OBJ3 equational theory.

Each syntactic class is encoded by an OBJ3 sort, while the symbols of the logic are denoted by OBJ3 operators with the correct arities and co-arities. Polymorphism is supported via the OBJ3 order-sorted type mechanism.

Each inference rule is identified by a distinct OBJ3 operator of co-arity *Rule*. Its arity corresponds to the OBJ3 sorts of the parameters that need to be supplied when the inference rule is applied.

The form of each inference rule is defined by a set of OBJ3 equations giving the formulae that would need to be proved as subgoals to prove a given formula by application of the inference rule.

2.2.2 Proof construction

Proof construction in 2OBJ is done using tactics and tacticals. 2OBJ does not provide any built-in proof method, but it provides the basic tacticals: *THEN*, *THENL*, *ELSE*, *idtac* and *REPEAT* that allow to program proof methods. The `fopc.obj` module is not well suited as a base for the construction of proof methods, because for apllying most of its tactics (inference rules) one or more arguments are needed.

As a solution for this problem we wrote the `cppo.obj` module, an implementation of Gallier's G calculus [4] for the intuitionistic case.

Subsequently we wrote the tactic *ProofProc* which automatizes the propositional proofs. 2OBJS's lack of unification prevents the automatization of first-order proofs.

3 The GCPL language and deduction system

In order to compare the two systems and justify our defence of the generic theorem provers, we will describe now a deduction system for an axiomatic definition of a guarded commands procedural language (GCPL).

3.1 The GCPL syntax and informal semantics

Before we present the GCPL deduction system we must describe how each instruction of GCPL should be executed [1, 7, 3, 10, 11, 13].

Skip (skip) execution of skip does nothing;

Assignment ($\mathsf{atr}(x, e)$, where x is a variable and e is an expression) execution of $\mathsf{atr}(x, e)$ consists of evaluating e and storing the resulting value in the location named x.

Sequential Composition ($\mathsf{alg}\ (S_1, P_1); (S_2, P_2); \ldots; (S_n, P_n)\ \mathsf{endalg}$) execute first S_1 then S_2, etc. and finally execute S_n. The P_i $(1 \leq i \leq n)$ are assertions that are needed for the proof of correctness;

Alternative ($\mathsf{if}\ B_1 \rightarrow S_1[] \ldots []B_n \rightarrow S_n\ \mathsf{endif}$) execution consists of choosing, at random, a guarded command $B_i \rightarrow S_i$ with true guard B_i and execute S_i;

Iterate ($\mathsf{do}\ B_1 \rightarrow S_1[] \ldots []B_n \rightarrow S_n\ \mathsf{enddo}$) iterate the following until all B_i $(1 \leq i \leq n)$ are false. Choose a true guard B_i and execute S_i;

Functions

function declaration ($\mathsf{function}\ f(x)\ \mathsf{vars}\ (aux)\{P\}S\{Q\}\ \mathsf{endfunction}$)

function call ($f[a]$, in an expression)

the effect of the invocation of a function is the execution of the body, S, with the actual parameters "replacing" the formal parameters. It is assumed that no assignments are made within S to global variables and that all the parameters are passed with a "call-by-value" mechanism. The function name is used as a variable to store the result of a function to be returned to the place of the function call;

Procedures

procedure declaration ($\mathsf{subprg}\ p(x) : (y)\ \mathsf{vars}\ (aux)\{P\}S\{Q\}\ \mathsf{endsubprg}$)

procedure call ($\mathsf{call}\ p(x) : (y)\ \mathsf{endcall}$)

execution of the procedure call causes the procedure body, S, to be executed. We shall consider that all variables of the procedure body (other then locally declared variables) are formal parameters of the procedure. Other restrictions to the formal and actual parameters will be discussed when we present the inference rules.

Pre-Post formulae ($\{P\}S\{Q\}$) whose meaning is: if the assertion P is true before S is executed, then Q will hold upon completion of execution of S.

3.2 The GCPL deduction system

The logical language of GCPL is a multi-sorted first order language with equality [4]. Following the notational conventions of [4] we have:

$S = \{\text{names}, \text{expr}, \text{inst}\}$; $CS_{\text{elems}} = \{n, c_1, \ldots\}$; $CS_{\text{inst}} = \{\text{skip}\}$; FS ={opr, atr, if, do, comp,deffunc,func,defproc,proc}; $PS = \{\text{prepost}\}$.

The arities and co-arities of the elements of FS and PS are the following:

$$
\begin{array}{rcl}
\text{opr} & : & \text{expr expr} \rightarrow \text{expr} \\
\text{skip} & : & \rightarrow \text{inst} \\
\text{atr} & : & \text{expr expr} \rightarrow \text{inst} \\
\text{if} & : & (\text{bool inst})^{+} \rightarrow \text{inst} \\
\text{do} & : & (\text{bool inst})^{+} \rightarrow \text{inst} \\
\text{comp} & : & \text{inst bool inst} \rightarrow \text{inst} \\
\text{deffunc} & : & \text{names } (\text{expr})^{+}\ (\text{expr})^{+} \text{ bool inst bool} \rightarrow \text{bool} \\
\text{func} & : & \text{names } (\text{expr})^{+} \rightarrow \text{expr} \\
\text{defproc} & : & \text{names } (\text{expr})^{+}\ (\text{expr})^{+}\ (\text{expr})^{+} \text{ bool inst bool} \rightarrow \text{bool} \\
\text{proc} & : & \text{names } (\text{expr})^{+}\ (\text{expr})^{+} \rightarrow \text{bool} \\
\text{prepost} & : & \text{bool inst bool} \rightarrow \text{bool}
\end{array}
$$

The rules of inference of GCPL [1, 3, 7, 10, 11, 13] are an extension of a first-order with equality deduction system, here we present then as an extension of LJ_{e} [4, 12]. We will present the instructions operators with a different syntax from above for a easier reading:

$$\frac{H\vdash\{P\}S\{R\} \quad H\vdash R\Rightarrow Q}{H\vdash\{P\}S\{Q\}} \vdash \text{consr(R)} \qquad \frac{H\vdash P\Rightarrow R \quad H\vdash\{R\}S\{Q\}}{H\vdash\{P\}S\{Q\}} \vdash \text{consl(R)}$$

$$\frac{H\vdash\{P_0\}S_1\{P_1\} \ H\vdash\{P_1\}S_2\{P_2\}}{H\vdash\{P_0\}S_1\{P_1\}S2\{P_2\}} \vdash \text{algR} \qquad \frac{}{H\vdash\{P\}\mathsf{skip}\{P\}} \vdash \text{skipR}$$

$$\frac{H\vdash\{P\wedge B_1\}S_1\{P\} \ \ldots \ H\vdash\{P\wedge B_n\}S_n\{P\}}{H\vdash\{P\}\mathsf{do}\ B_1\rightarrow S_1[]\ldots[]B_n\rightarrow S_n\ \mathsf{enddo}\{P\wedge\neg(B_1\vee\ldots\vee B_n)\}} \vdash \text{doR} \qquad \frac{}{H\vdash\{P_e^x\}x:=e\{P\}} \vdash \text{atrR}$$

$$\frac{H\vdash P\wedge\neg(B_1\vee\ldots\vee B_n)\Rightarrow Q \ \ H\vdash\{P\wedge B_1\}S_1\{Q\} \ \ldots \ H\vdash\{P\wedge B_n\}S_n\{Q\}}{H\vdash\{P\}\mathsf{if}\ B_1\rightarrow S_1[]\ldots[]B_n\rightarrow S_n\ \mathsf{endif}\{Q\}} \vdash \text{ifR}$$

$$\frac{H\vdash\{P\}S\{Q\}}{H\vdash\forall_x(P\Rightarrow Q^{f}_{f(x)})} \vdash \text{funR(i)}$$

$$(H_i = \mathsf{function}\ \text{f (x)}\ \ \mathsf{vars}\ \text{(aux)}\ \{\text{P}\}\ \text{S}\ \{\text{Q}\}\ \ \mathsf{endfunction})$$

$$\frac{H\vdash\{P\}S\{Q\}}{H\vdash\{P^{x,y}_{a,e}\}\mathsf{call}\ p(a){:}(e)\ \mathsf{endcall}\{Q^{x,y}_{a,e}\}} \vdash \text{callR(i)}$$

$$(H_i = \mathsf{subprg}\ \text{p(x):(y)}\ \ \mathsf{vars}\ \text{(aux)}\{\text{P}\}\text{S}\{\text{Q}\}\ \ \mathsf{endsubprg})$$

If necessary new variables are created to avoid clashes between local and global variables. We have that: x – list of non-local variables subject to change; y – list of non-local variables not subject to change; aux – list of local variables; a – list of distinct variables; e – list of expressions not containing any of the variables in a;

4 Comparing the two systems

In order to compare the two systems we have implemented GCPL in both systems: *gcpl.thy* (Isabelle) and *gcpl.obj* (2OBJ). We performed several proofs with the two implementations.

4.1 Syntax

The GCPL syntax implementation was carried out in both systems without great difficulties. We feel that *gcpl.thy* has, a cleaner syntax then *gcpl.obj*, due to the fact that in Isabelle we write the pretended system in a "meta-syntax" that will be translate automatically to the internal syntax (Standard ML). Isabelle "borrows" from OBJ3 some of its most interesting syntactic features: mixfix syntax and the order sorted types allowing overloading identifiers.

In 2OBJ we write our logical system in OBJ3 language and we must be aware to the potential conflicts that may arise from this fact.

gcplt.thy ...

```
types
   expr, inst 0
arities
   expr, inst :: term
...
const
   skip    :: "inst"
   atr     :: "[expr,expr] => inst" ("_:=_")
   comp    :: "[inst,o,inst] => inst" ("_{_}_")
...
   prepost :: "[o,inst,o] => o" ("{_}_{_}" [0,0,0] 200)
...
```

gcpl.obj ...

```
sorts Expr Inst .
sorts Var-Expr Var-Inst .
subsorts Var-Expr < Var .
...
op skip : -> Inst .
op atr : Expr Expr -> Inst .
op alg_endalg : IClist -> Inst .
...
op {_}_{_} : Sent Inst Sent -> Prop .
...
```

4.2 Logical systems

We have used Isabelle's LK subsystem as a base for our system so we may compare the two implementations. Once again the Isabelle "meta-syntax" allows a cleaner syntax than 2OBJ.

gcpl.thy ...

```
rules
  skipR "$H |- $E,{P}skip{P},$F"
  atrR    "$H |- $E,{P(e)}x:=e{P(x)},$F"
  compR   "[| $H |- $E,{P}S1{R},$F ; $H |- $E,{R}S2{Q},$F|]  ==> \
\             $H |- $E,{P}S1{R}S2{Q},$F"
...
```

gcpl.obj ...

```
vars P P0 P1 Pn : Sent .
var Slist : ICList .
vars S1 S2 : Inst .
ops skipR atrR algR : -> Rule .
eq skipR ( H |- {P}skip{P} ) = [] .
cq atrR ( H |- {Q}atr(X,Y){P} ) = []
         if Q == ( P o Y / X ) .
eq algR ( H |- {P0}alg ([]) endalg{Pn}) = [] .
eq algR ( H |- {P0}alg (<< S1 ; P1 >> ; Slist) endalg{Pn}) =
        ( H |- {P0}S1{P1}),
        ( algR ( H |- {P1}alg Slist endalg{Pn})) .
...
```

4.3 Proof construction

In Isabelle we may use all the built-in procedures for proof construction in a very simple manner. Isabelle provides "packs", pairs of sets, the first of them will contain the safe rules and the second the unsafe rules (an unsafe rule may reduce a provable goal to an unprovable set of subgoals). These "packs" will be used by the proof tactics.

In *gcpl.ML* we have defined the *gcpl_pack* adding to *prop_pack* (propositional pack) the safe rules: *skipR, atriR,* ..., and the unsafe rules *consl* and *consr*. We have also implemented, using tacticals, *gcpl_tac*, a tactic that allow us, as we will show in the next section, an almost complete automatization of the proofs.

In 2OBJ, as we already said there are no built-in proof procedures. We have based our *gcpl.obj* in *cppo.obj* and we have implemented, using tacticals, two proof procedures *ProcProof* for proofs in the propositional calculus and *WithoutSubprg* for proofs in the GCPL system (which uses the first). The lack of unification/resolution does not allow a complete automatization of the proofs.

The use of OBJ3 as a base for a theorem prover allows the acess to the OBJ3 facilities, namely, equational reasoning.

4.4 Proofs in GCPL

We have chosen two GCPL programs in order to compare the two systems.

The first example is the algorithm for integer division using only additions and subtractions. The second is Euclide's algorithm for the calculus of g.c.d. of two natural numbers.

First example:

Division.thy

```
Divisao = GCPL +
end
```

Division.ML

```
open Divisao;
goal GCPL.thy "|- {r .=. r+y*0}     \
\                 x:=r              \
\                 {x .=. r+y*0}     \
\                 q:=0              \
\                 {x .=. r+y*q}     \
\                 do (y <= r,       \
\                   r:=r-y          \
\                   {x .=. r+y+y*q} \
\                   q:=q+1) :GCNil  \
\                 od {x .=. r+y*q & ~ y<=r}"
```

Proof

```
- by (gcpl_tac GCPL_pack 1);
...
 1. x .=. r + y * q , y <= r |- x .=. r - y + y + y * q
 2. x .=. r + y + y * q  |- x .=. r + y * (q + 1)
```

Isabelle was able to prove automatically the correctness of the program up to the point were it's lack of equality rules prevents it from proving sub-goals 1 and 2.

Division.obj

```
obj DIVISAO is
 pr GCPL .
ops x y r q : -> Var-Elems .
let Divisao = ( * |- |A| x . |A| y . |A| r . |A| q .
{(x = (x + (y * 0)))}
alg
<< atr(r,x) ; x = (r + y * 0) >> ;
<< atr(q,0) ; x = (r + y * q) >> ;
<< do
   << y <= r ;
      alg
      << atr(r,r - y) ; x = (r + y + y * q) >> ;
      << atr(q,q + 1) ; x = (r + y * q) >>
      endalg >>
   enddo ; (x = (r + (y * q)) ^ (~ (y <= r))) >>
endalg
{x = (r + (y * q))}) .
vars z1 z2 : Var-Elems .
[axiom,div1] eq z1 + (- z1) = 0 .
[axiom,div2] eq - z1 + z1 = 0 .
[axiom,div3] eq z1 + z1 * z2 = z1 * (z2 + 1) .
endo
```

Proof

```
WithoutSubprg
1 consl(x = (r + (- y + y) + y * q))
1 1 reducef('div3)
1 1 1 PropProof
1 2 atrR
2 consl(x = (r + y * (q + 1)))
2 1 reduce
2 1 1 PropProof
2 2 atrR
Proof is COMPLETE
```

Using OBJ3 rewrite mechanism (tactics *reducef* and *reduce*) we were able to complete the proof. The lack of unification/resolution does not allow us a greater automatization of the proof.

Second example:

Euclides.thy

```
Euclides = GCPL +
consts
 gcd :: "[expr,expr] => expr"
rules
 abm0 "$H, A > 0, $I, B > 0, $J, A > B, $K |- $E, A - B > 0,$F"
 bam0 "$H, A > 0, $I, B > 0, $J, B > A, $K |- $E, B - A > 0,$F"
 gcd1 "$H, gcd(A,B) .=. gcd(X,Y), $I, A > B, $K \
\              |- $E, gcd(A - B,B) .=. gcd(X,Y),$F"
 gcd2 "$H, gcd(A,B) .=. gcd(X,Y), $I, B > A, $K \
\              |- $E, gcd(A,B - A) .=. gcd(X,Y),$F"
end
```

Euclides.ML

```
open Euclides;
val Euclides_pack = gcpl_pack add_safes [abm0,bam0,gcd1,gcd2];
goal Euclides.thy "|- {x > 0 & y > 0 & gcd(x,y) .=. gcd(x,y)} \
\ a:=x                                                        \
\ {a > 0 & y > 0 & gcd(a,y) .=. gcd(x,y)}                     \
\ b:=y                                                        \
\ {(a > 0 & b > 0 & gcd(a,b) .=. gcd(x,y))}                   \
\ do < ~ (a .=. b),                                           \
\   do <a > b,                                                \
\     a:=a-b> : GCNil                                         \
\   od                                                        \
\   {(a > 0 & b > 0 & gcd(a,b) .=. gcd(x,y)) & (~ (a>b))}     \
\   do < b > a,                                               \
\     b:=b-a> : GCNil                                         \
\   od > : GCNil                                              \
\ od {(a > 0 & b > 0 & gcd(a,b) .=. gcd(x,y)) & ~(~(a .=. b))}";
```

Proof

```
by (safe_goal_tac gcpl_pack 1);
by (consL_tac "(a > 0 & b > 0 & gcd(a,b) .=. gcd(x,y) )" 1);
by (consL_tac "(a > 0 & b > 0 & gcd(a,b) .=. gcd(x,y) )" 3);
by (consR_tac "(a > 0 & b > 0 & gcd(a,b) .=. gcd(x,y)) & ~ b > a" 4);
by (gcpl_tac Euclides_pack 1);
No subgoals!
```

Isabelle was able to complete the proof of correctness of the program but not in an complete automatic way. The dificulty resides in the need of using, in the same proof, the unsafe rules *consl* and *consr*. The *consL_tac* and *consR_tac* allow us to give the necessary formula to "complete" the corresponding rule. They are not strictly necessary, Isabelle can guess the correct instantiation, nevertheless the proof must be directed by the user because the order of aplication and the rule to aplly are crucial points in this proof.

Euclides.obj

```
obj EUCLIDES is
 pr GCPL .
ops x y a b : -> Var-Elems .
ops gcd : Expr Expr -> Elems [comm] .
let Euclides =
( * |- |A| x . |A| y . |A| a . |A| b .
{(x > 0) ^ (y > 0) ^ (gcd(x,y) = gcd(x,y))}
alg
<< atr(a,x) ; (a > 0) ^ (y > 0) ^ (gcd(a,y) = gcd(x,y)) >> ;
<< atr(b,y) ; (a > 0) ^ (b > 0) ^ (gcd(a,b) = gcd(x,y)) >> ;
<< do
 << ~ (a = b) ;
  alg
   << do
    << (a > b).Sent ; atr(a,a - b) >>
   enddo ; ((a > 0) ^ (b > 0) ^ gcd(a,b) = gcd(x,y)) ^ ~(a > b) >> ;
   << do
    << b > a ; atr(b,b - a) >>
   enddo ; ((a > 0) ^ (b > 0) ^ gcd(a,b) = gcd(x,y)) ^ ~(b > a) >>
  endalg >>
 enddo ; ((a > 0) ^ (b > 0) ^ gcd(a,b) = gcd(x,y)) ^ ~(~(a = b)) >>
endalg
{(a > 0) ^ (b > 0) ^ (gcd(a,b) = gcd(x,y)) ^ ~ ~(a = b)}) .
ops lemma1 lemma2 lemma3 lemma4 : -> Lemma .
eq lemma lemma1 =
  ( * |- |A| a . |A| b . (a > b) -> gcd(a,b) = gcd((a - b).Expr,b)) .
eq lemma lemma2 =
  ( * |- |A| a . |A| b . gcd(a,b) = gcd(b,a)) .
eq lemma lemma3 =
  ( * |- |A| a . |A| b . ((a > b) ^ (a > 0)) -> ((a - b).Expr > 0)) .
eq lemma lemma4 =
  ( * |- |A| a . |A| b . ((b > a) ^ (b > 0)) -> ((b - a).Expr > 0)) .
endo
```

Proof

```
WithoutSubprg
```

```
1 consl((a > 0) ^ (b > 0) ^ (gcd(a,b) = gcd(x,y)))
1 1 PropProof
1 2 doR
1 2 1 consl((a + - b) > 0 ^ b > 0 ^ gcd(a + - b,b) = gcd(x,y))
1 2 1 1 reducee(none,gcd(a + - b,b) =' gcd(a,b) [a ;; b | 'gcd1])
1 2 1 1 1 reducee(none,(a + - b > 0).Sent =' True [a ;; b | 'mz1])
1 2 1 1 1 1 PropProof
1 2 1 2 atrR
2 consl((a > 0) ^ (b > 0) ^ (gcd(a,b) = gcd(x,y)))
2 1 PropProof
2 2 doR
2 2 1 consl(a > 0 ^ (b + - a) > 0 ^ gcd(a,b + - a) = gcd(x,y))
2 2 1 1 reducee(none,gcd(a,b + - a) =' gcd(a,b) [a ;; b | 'gcd2])
2 2 1 1 1 reducee(none,(b + - a > 0).Sent =' True [a ;; b | 'mz2])
2 2 1 1 1 1 PropProof
2 2 1 2 atrR
Proof is COMPLETE
```

We can introduce lemmas in 2OBJ proofs using the tactical *LEMMA* but due to the lack of unification is simpler to use the underlying rewrite mechanism of OBJ3.

Once again the degree of automatization is far less than that of Isabelle.

5 Conclusions

The implementation of a proof system for an axiomatic definition of a procedural language with guarded commands (GCPL) in both programs shows the kind of problems that require generic theorem provers. We have been able to define a deduction system for GCPL, and programe some proof strategies. During the construction of such a system two problems were detected: 2OBJ's lack of mechanisms for a complete automatization of the proof, namely the unification, and Isabelle's lack of equality rules.

The proof of correctness of programs is an interesting and complex problem. In order to construct a system to conduct proofs in it we must put together the deduction system for the instructions of the programming language (pre-post conditions), the deduction systems for all the data structures that can be manipulated by the programming language and the necessary lemmas for each program. This is, in our opinion, an almost impossible task.

One of the solutions to this problem would be the construction of the deduction system with only the necessary and sufficient information to do the proof of correctness of a single program, and not all the proofs of all the programs. This could be achieved by a parametrised modular construction of the deduction system. The user of such a system would create an object/theory binding the program (theorem) to be proved to the necessary lemmas an the instantiated GCPL theory.

Both programs are based in programming languages having the capability of parameterised modular construction of programs. The use of such capability for the construction of a complex proof systems, seems to be the obvious path forward.

We have, at least, three different possibilities:

Parameterised modular construction under the same logical system and the same deduction system. For example: with a first order logic sequent calculus deduction system for gcpl programs, integers and lists we would built a first order logic sequent calculus for gcpl programs dealing with lists of integers.

Parameterised modular construction under different logical systems with the same deduction system. For example: a first order logical system for gcpl programs and a higher-order system for integers.

Parameterised modular construction under different deduction systems (and different logical systems). In this case the proof tree would be composed of branches of different kinds. We have done something of this sort when we used OBJ3 rewriting mechanism (equational inference) in our sequent calculus proofs. Eventually we can also foresee the possibility of combining different theorem provers, specialised in specific mechanisms, in a single deductive system.

References

[1] Alagić, S., Arbib, M. A.: The Design of Well-Structured and Correct Programs. Springer-Verlag. New York (1978)

[2] Chang, C., Lee, R. C.: Symbolic Logic and Mechanical Theorem Proving. Academic Press, Inc.. Orlando (1973)

[3] Clint, M., Hoare, C. A. R.: Program proving: Jumps and functions. Acta Informatica, **2** (1972) 214–224

[4] Gallier, J. H.: Logic For Computer Science: Foundations of Automatic Theorem Proving. Computer Science and Technology Series. John Wiley & Sons. New York (1987)

[5] Goguen, J., Winkler, T.: Introducing OBJ3. SRI International, Computer Science Lab. (1993)

[6] Gordon, M. J., Milner, A. J., Wadsworth C. P.: Edinburgh LCF. Lecture Notes in Computer Science **78** (1979)

[7] Gries, D.: The Science of Programming. Springer-Verlag. New York (1981)

[8] Harper, R.: Introduction to Standard ML. University of Edinburgh (1986)

[9] Harper, R., Honsel, F. and Plotkin G.: A Framework for Defining Logics. Technical Report CMU-CS-89-173. School of Computer Science, Carnegie Mellon University

[10] Hoare, C. A. R. : Procedures and parameters: An axiomatic approach. Lecture Notes in Mathematics **188** (1971) 102–106

[11] Hoare, C. A. R. : An axiomatic definition of the programming language Pascal. Acta Informatica **2** (1973) 335–355

[12] Szabo, M. E. : The Collected Papers of Gerahrd Gentzen Studies in Logic and the Foundations of Mathematics. North-Holland Publishing Company. Amsterdam (1969)

[13] Meyer B.: Introduction to the Theory of Programming Languages. Prentice Hall International. New York (1990)

[14] Paulson, L. C.: Introduction to Isabelle. Computer Laboratory University of Cambridge (1992)

[15] Paulson, L. C.: Isabelle's Object-Logics. Computer Laboratory University of Cambridge (1993)

[16] Paulson, L. C.: The Isabelle Reference Manual. Computer Laboratory University of Cambridge (1993)

[17] Stevens A., Hobley, K.: Mechanised Theorem Proving with 2OBJ: A Tutorial Introduction. draft edition, (February 1993)