Acta Univ. Sapientiae, Informatica, **1**, 1 (2009) 15–34

Meta programming on the proof level

Gergely Dévai
Eötvös Loránd University, Faculty of Informatics,
Department of Programming Languages and
Compilers
email: deva@elte.hu

Abstract. Computer aided proof generation is used for many reasons from formalization of mathematics to formal computer program development. Our research concentrates on completely declarative style proofs used to develop imperative programs in a refinement-based model (i.e. deriving the algorithm from the specification).

In this paper we investigate why and how to use meta programming techniques for proof development. We examine techniques already used in programming languages if they are applicable for proof construction and point out the specialities caused by the different application area. It is also shown that while meta programming techniques are often dangerous when used to develop programs, they are safe tools for proof development.

1 Introduction

1.1 Human-readable proofs

There is a wide range of theorem provers from completely automatic ones to proof checkers. Examining the history of formal program development and automatic theorem proving, it seems hopeless to create a system that proves the correctness of industrial sized programs without considerable human effort. It is crucial that users of a proof system can easily understand the given proof situations if the machine is not able to complete the proof automatically. For

AMS 2000 subject classifications: 68Q60
CR Categories and Descriptors: F.3.1. [Specifying and Verifying and Reasoning about Programs]
Key words and phrases: formal program verification, meta programming, proof generation

```
true() => abs(x) >= 0
select
{
  x >= 0 => abs(x) >= 0
  {
    x >= 0 => x = abs(x);
    x = abs(x) & x >= 0 => abs(x) >= 0;
  }
  !(x >= 0) => abs(x) >= 0
  {
    !(x >= 0) => 0-x = abs(x);
    !(x >= 0) => 0-x >= 0;
    0-x = abs(x) & 0-x >= 0 => abs(x) >= 0;
  }
}
```

Figure 1: The non-negativity of the absolute value function

that reason, there is a growing interest in human-readable proof languages. The proof language of the *Mizar* proof assistant [1] was designed to be similar to textbook proofs. Similar style is used in the *Isar* [2] language for *Isabelle* [3] and in a declarative style proof language for *Coq* [4].

In our case proofs are used mainly to specify the behaviour of imperative programs and to develop the algorithm by refining the original specification. When refining a specification statement, one gives a set of more detailed statements. The refinement is sound, if every program that corresponds to the detailed specification also fulfils the requirements of the original one. We can say that the original specification is a theorem about the behaviour of the resulting program and the refinement steps are the proof. In this proof style, instead of indicating the proof actions, one breaks up the original theorem into several smaller theorems.

As an example we show a toy proof about the non-negativity of the absolute value function on figure 1. The first line states the theorem to prove. The select keyword indicates case-distinction, the cases $x \geq 0$ and $\neg(x \geq 0)$ are inside the pair of curly braces. Each case is refined further: In the case where $x \geq 0$ holds we first conclude that $x = \mathtt{abs}(x)$ and from this (and the previous knowledge $x \geq 0$) we get that $\mathtt{abs}(x) \geq 0$. The second case is similar. The unrefined statements are accepted by the proof checker based on previously

defined axioms and tactics.

In this system there are two ways to refine a statement (both shown in the previous example): sequence and case-distinction. Sequence introduces intermediate steps in the reasoning, while case-distinction splits up the proof into several cases. For more information on the refinements and the techniques to check them the reader is referred to [5].

1.2 Programming by proof

According to the programming paradigm we use, one first writes the specification of the program, then refines the specification in several steps. Unlike traditional refinement systems [6], during the refinement process we do not introduce program fragments. The proof tree (consisting of the specification as the theorem and the refinements as the proof steps) is complete when one reaches specifications of primitive instructions in the leaves.

In the following example on figure 2, the specification states that the program swaps the values of the two variables `x` and `y`. Instead of the operator `=>` that we used for classical logic statements in the previous example, here we use the `>>` temporal operator to express that the program proceeds from the first condition to the second one. We use the parameters `xVal` and `yVal` to denote the values of `x` and `y` respectively in the pre-state. The variable `ip` (instruction pointer or program counter) is used explicitly in specification statements. (This makes the specification of control statements, like jumps and procedure calls, much easier.)

The specification is refined by a sequence of three statements that describes how do the values of variables change during the execution of the program.

The compiler of this proof language has two tasks: it first checks the soundness of the proof, then it collects the primitive instructions whose specifications are used in the proof, and generates the program in the target language. In case of the current example the following instructions are extracted from the proof.

```
A: t = x;
B: x = y;
C: y = t;
D:
```

The resulting program is guaranteed to be correct with respect to the specification, provided that the specifications of the primitive instructions were sound.

```
variable(x,Integer);
variable(y,Integer);
parameter(xVal,Integer);
parameter(yVal,Integer);

ip = A & x=xVal & y=yVal >> ip = D & x=yVal & y=xVal
{
  variable(t,Integer);
  ip = A & x = xVal >> ip = B & t = xVal;
  ip = B & y = yVal >> ip = C & x = yVal;
  ip = C & t = xVal >> ip = D & y = xVal;
}
```

Figure 2: Swapping the values of two variables

The system is independent of the target language. One can add support for a new programming language by specifying (part of) its instruction set in the system and writing a code generator module for that language.

This paradigm differs from the classical *program extraction from proofs* [7]. In that case by developing a constructive proof for the existence of a mathematical object M, one can extract a program that evaluates M. In our case the program is extracted from its own correctness proof and there is no restriction on the logic used.

2 Meta programming

2.1 Minimal trusted base

Is the output program really sound? It depends on the correctness of the proof checking algorithm used by the compiler and on the correctness of the specifications of primitive instructions. This is called the *trusted base* of the system. To reduce the risk of errors in it, the trusted base should be minimal.

Our currently supported target language is *C++*, a language that is extremly rich in high level language construts like different kinds of loops, variable scopes, argument passing modes, classes, inheritance and a lot more. If we wanted a formal system supporting all these features using built-in rules, the programming model would be very complex, harly exendable and target language dependent.

In order to keep the trusted base of our system minimal and general, we elected to reduce the programming model as much as possible. We consider a program as a set of simple state transitions. Instructions that perform a simple state transition are easy to specify.

Although this programming model is very simple, it is expressive enough. We were able to specify for example pointer instructions [8] and vector operations [9] of *C++* in it.

2.2 Motivation for meta programming

Formal program development in the model presented above is like programming in assembly languages: No high-level language constructs are available, only a set of elementary instructions. We have discussed above that hardwiring the verification conditions for high level constructs in the system is not desirable. The questions is, how to enable the user to extend the system with these high level constructs without affecting the minimality of the trusted base?

A possible answer is meta programming, which was already used in case of assembly languages in the form of macros. For assembly programmers macros are useful to emulate instructions that are not part of the instruction set, generalize often-used program fragments using arguments and emulate high-level language constructs (loops, conditional branching, etc.). We use meta programming techniques for the same reasons *on the proof level*: We generalize often used proof parts. We call these proof fragments *proof templates* and they may be both classical logic proofs (like the schema of indirect proofs) or temporal logic proofs (like the schema of proving the correctness of a loop).

While meta programming techniques in assembly languages are quite low level features (simple text-based replacement of arguments, for example), the techniques we use are more sophisticated. We apply type checking for arguments and perform their substitutions in the syntax tree instead of the error-prone text-based replacement.

The user is also allowed to define own proof templates and it is possible to build libraries of them to help the work of other users. In traditional program development, a library consisting of a great number of functions can help the developer to make the code shorter and more understandable while it also rises the efficiency of the development process. The same is true for proof development: It is a general observation that proof systems with a huge set of tactics are more efficient.

If the user defines a proof template, it does not become part of the trusted

base. The compiler checks the proofs inside templates. This check occurs either when the template is defined or when it is instantiated depending on the type of the template. If the template was wrong or was used in an inappropriate situation, an error report is generated during proof checking.

2.3 Naming conventions

Meta programming is a general notion. In this paper we use it to name techniques to define meta language entities that are transformed to object level entities during a preprocessing phase after which some kind of compilation of the object language entities takes place. In case of traditional programming, assembly macros or *C++* templates (meta language) are first transformed to pure assembly or *C++* (object language) and compiled further by an assembler or a *C++* compiler. In our case, the meta language consists of proof templates. Template calls are transformed to pure proofs which are then checked and compiled to a traditional programming language.

In the literature the terms *proof schema*, *proof sketch*, *proof template* are used in various senses. In [10] *formal proof sketches* are defined to be shortened formal proofs which have gaps from the point of view of mechanical proof checking but are easier to understand. In formal program verification it happens quite often that the proof attempt fails because the program is wrong. In that case, after correcting the program, one can reuse parts of the previous proof attempt. In [11] these reused proofs are called *proof templates* and in [12] generalized proofs to replay are called *proof schemas*. Our proof templates are similar to these techniques in the sense that they are generalized and reusable proof fragments with the goal making proofs shorter and more understandable, but are completely different in the way of their definition and application.

2.4 Overview of techniques

In the following we examine the meta programming techniques that we have found useful for the purposes of proof-development.

2.4.1 Arguments

We can generalize a proof fragment by giving it a name and replacing parts of it by arguments. This way we obtain a meta-proof that we call a *template*. It turned out to be useful to syntactically distinguish formal template arguments

from program variables and parameter variables. (We start them by a sharp symbol: `#`.)

To define a template called `example` having two arguments of types `Integer` and `Character` we write:

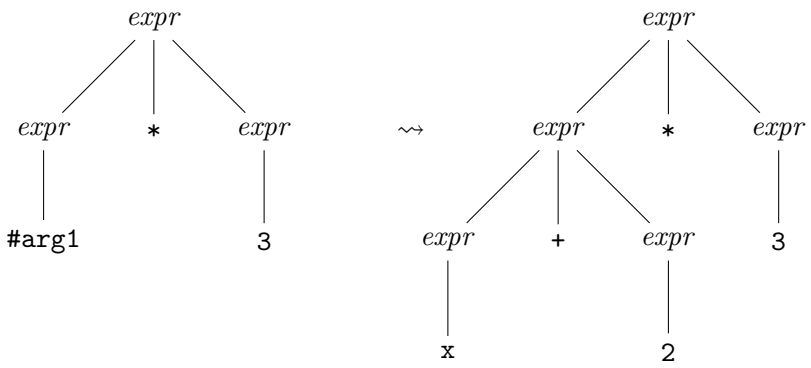
```
template example( Integer #arg1, Character #arg2 )
{
  // template body
}
```

In the body of the template we can write a proof fragment containing the formal template arguments. We instantiate the template using actual arguments in the following way:

```
example( x+2, 'a' );
```

The template call is type checked. The compiler instantiates the template by replacing the formal arguments in the body by the actual ones.

Simple text-based replacement of arguments would lead to surprising results in some cases. A well-known example of low level macros is the expression `#arg1*3` which becomes `x+2*3` in the previous template call. By the precedence rules of operators this means `x+(2*3)` instead of `(x+2)*3` which, supposedly, was the intention of the programmer. To avoid this pitfall in our system, instantiation of templates takes place after parsing. As shown in the figure below, the replacement is done in the syntax tree of the expression and produces the correct result.



An other common feature of low level macros is that symbols used inside the macro may not be declared at the point of the macro definition. The

meaning of the symbols depend on the declarations visible at the point of the instantiation. This can induce many errors that are hard to recover. In case of our templates any symbol inside the template body must be declared and they are bound to the declaration visible at the point of the template definition. Even if an other declaration hides the original declaration of the symbol at the point where one calls the template, the symbol in the instantiated proof will belong to the original declaration.

It is quite important that the proofs obtained by instantiating a template may not be sound in general. Templates should be thought of as proof attempts or proof schemas instead of theorems with their proofs. That is, when a template is defined, the soundness of its body is not checked. But, each time the template is called, the resulting proof is verified and the errors are reported. In section 2.4.5 we introduce a special kind of template that is verified as soon as it is defined, so that there is no need for further checks when one instantiates it.

2.4.2 Compile time conditions

Meta programming makes it possible to perform computations in compile time. The result of the meta-level computations can influence the generated object-level code. For example, conditions expressed in the meta language can decide whether a piece of code is included or omitted. These techniques turned out to be useful also in case of proof generation.

A common situation is that different proofs are needed depending on the form of a template argument. In the following example the argument of the template is examined by *compile time conditions*. These conditions decide which variant of the proof should be used.

```
template variants( Integer #arg )
{
  constant( #arg ) :
    // proof in case of a constant argument
  variable( #arg ) :
    // proof in case of a variable argument
}
```

In case the template call is `variants( 2 )`, the first proof is the result of the instantiation, while the call `variants( x )` results in the second one.

We can increase the expressive power of these conditions if we enable pattern matching for their arguments. In the following example, the condition checks

whether the argument is a compound expression with addition as the top level operator and bounds the two sub-expressions to the arguments `#left` and `#right`. These can be used in the proof.

```
template patternMatcher( Integer #arg )
{
  equals( #arg, #left + #right ) :
    // proof that can use #left and #right
}
```

In case of the template call `patternMatcher( x+2 )`, the compiler replaces `#arg`, `#left` and `#right` by `x+2`, `x` and `2` respectively in the proof.

In the current implementation of our system there is a fixed set of such conditions. We have noticed that it would be useful to give the user the ability to define new conditions based on the old ones. We plan to carry out a *Prolog*-style implementation of this feature.

2.4.3 Axioms and instruction specifications

It seems natural to use the template features of the language to define the axioms that are used to close the branches of proof trees. We mark these templates with special keywords so that the compiler accepts the statements obtained by instantiating these templates without any further refinement.

There are two kinds of axioms in the system: One states properties of mathematical functions used in specifications and the other describes temporal properties of instructions in the target language. We call the former ones *axioms* and the latter ones *atoms*.

For example we show the temporal axiom of the instruction incrementing a variable. The arguments of the template will be the variable to increment (`#var`), an expression describing the value of the variable before the instruction (`#val`), and the labels before and after the instruction (`#before` and `#after`). The temporal axiom is basically the following:

```
ip = #before & #var = #val & #val < maxInt()
>> ip = #after & #var = #val + 1;
```

But this statement is not sound for all the possible combinations of the template arguments. For example, if we instantiate the template using the variable `x` both for the arguments `#var` and `#val`, the statement becomes invalid. The following conditions should be checked: the labels `#before` and `#after` have

```
atom increment( Integer #var, spec Integer #val,
                Label #before, Label #after )
{
  constant(#before) & constant(#after) & variable(#var)
  & independent(#var,#val) & independent(ip,#val) :
    ip = #before & #var = #val & #val < maxInt()
    >> ip = #after & #var = #val + 1;
}
```

Figure 3: Specification of the increment operation

to be label constants (this is needed for code generation), `#var` must be a variable and `#val` must not depend on `ip` and `#var`. These checks can be easily implemented using compile time conditions introduced in the previous section. The definition of the temporal axiom[1] is on figure 3.

There is one more thing to mention about this template: The `spec` keyword before the template argument `#val`. This marking informs the compiler that `#val` is not used for code generation, it is needed for specification purposes only. This means that the generated instruction does not depend on the actual expression provided for the argument `#val` in the template call. In section 2.4.5 we will see that it is sometimes important for the compiler to know which arguments does and which does not affect the generated instructions.

2.4.4 Proof fragments as arguments

Arguments of macros or templates are usually expressions or types in most systems using meta programming. In this section we show that allowing complete chunks of object level code (proofs in our case) as arguments rises the flexibility of the meta language.

To demonstrate this feature we construct a template that generates indirect proofs. To prove $P \Rightarrow Q$ by induction, we have to show $P \wedge \neg Q \Rightarrow \mathtt{false}()$. In our system, indirect proof is not a refinement possibility, we have sequence and case distinction only. We have to implement the indirect proof with these tools.

[1] Here we give only a progress property of this instruction. To make the axiom more useful, we could add a safety property describing which variables are affected by this operation. As the formal programming model behind this specification is not the main topic of this paper, we elected to simplify the example by omitting the safety property.

```
P => Q
select
{
  Q & P => Q
  {}
  !(Q) & P => Q
  {
    !(Q) & P => false()
    {
      // proof of contradiction
    }
  }
}
```

Figure 4: An indirect proof

The trick, shown on figure 4, is to perform a case distinction on Q. The case where Q holds is trivial, so an empty refinement is enough to complete it. In the second case we can use the proof of contradiction. As soon as `false`() is proved, the compiler also accepts Q.

As any indirect proof can be transformed to the same form, it is worth creating a template that does this transformation. Such a template must get the proof of contradiction as an argument. We use the `block` keyword in the argument list of a template to denote that a proof fragment has to be passed for that argument. Using this feature we can write our template shown on figure 5.

The proof inside the template is organised as we have discussed above. The argument marked by the `block` keyword can appear in any position where a proof is needed.

We also need a bit of special syntax to pass these special arguments when the template is called. It is done by writing the proof fragment to pass between curly braces after the template call (which is not terminated by a semicolon). The indirect proof calling the template we have just constructed is on figure 6. This template call results in the same proof shown on figure 4, but this variant is shorter and easier to understand.

In section 2.1 we have mentioned the advantages of the minimalistic programming model without built-in rules for programming constructs like conditional branching or loops. As a result, a simple *if-then-else* construct consist

```
template indirect( Boolean #hypothesis, Boolean #goal,
                   block #proof )
{
  #hypothesis => #goal
  select
  {
    #goal & #hypothesis => #goal
    {}
    !(#goal) & #hypothesis => #goal
    {
      !(#goal) & #hypothesis => false()
      {
        #proof;
      }
    }
}
```

Figure 5: Template for indirect proofs

of several instructions: First the condition is to be evaluated, then a conditional jump instruction follows to jump to the label of the *then-branch* if the condition was true or to continue at the label of the *else-branch* otherwise. At the end of both branches an instruction is needed to jump to the instruction that follows the branching.

It is possible to prove the correctness of such a low-level branching algorithm in the system, however, it is not desirable to force the programmer to develop a rather complex proof each time when using an *if-then-else* construct. Fortunately it is possible to generalise the proof and hide the details (that are same for every conditional branch) using a template. The condition of the branch and the proofs for the branches will be the arguments of the template, which can be called in the following way.

```
if( condition )
{
  // proof of the then-branch
}
{
  // proof of the else-branch
}
```

```
P => Q
{
  indirect( P, Q )
  {
    // proof of contradiction
  }
}
```

Figure 6: Indirect proof calling the `indirect` template

This means that there is no need to hard-wire techniques like indirect proofs or verification conditions for programming constructs into the system core. It is possible to define templates that provide the same convenience, without making the kernel of the proof system unnecessarily complex.

2.4.5 'Check once, use many times'

The templates we have seen so far could generate completely different proofs for different actual arguments and there was no guarantee that these proofs were sound. That is why the compiler had to instantiate and check the templates every time. If a template can be checked independently of the actual arguments, it is possible to validate it when it is defined. When such a template is called, the compiler can accept its top-level statement without instantiating and re-validating the whole proof inside it. Practically, a template of this kind contains a theorem and its proof.

As these templates are not dynamically instantiated and checked at every call, we call them *static templates*, and use the `static` keyword to introduce them. Arguments of static templates are not allowed to appear in compile time conditions, because that would make it impossible to validate the proof regardless of the actual arguments.

No matter how many times we call a static template, it is validated only once, and this is not only an efficiency issue. We can use it to implement induction: If a static template calls itself recursively, the recursive call is not expanded and checked, but its specification is used as an induction hypothesis. The well-foundness of the induction is ensured by the first argument of the recursive call: It must be an integer expression proved to be non-negative and strictly less then the first argument of the template containing the recursive call.

```
axiom p0( Integer #n )
{
  #n = 0 => p(#n);
}

axiom pNext( Integer #n )
{
  #n > 0 & p(#n-1) => p(#n);
}
```

Figure 7: Axioms for an inductive proof

For showing this recursive schema, let us have a logical function `p` on integers with the axioms on figure 7.

The static template on figure 8 proves `p` for all non-negative integers by induction. As in a usual induction proof, we have a base case (`#n=0`, solved by the first axiom) and an inductive case (`#n>0`). In the inductive case we first make it explicit that the induction is well-founded, then use the induction hypothesis by calling the template recursively with the argument `#n-1`. Then, by the second axiom we complete the proof.

Static templates are meaningful also for temporal logic proofs. In this case, not only the soundness of the refinements inside the template has to be independent of the actual template arguments, but also the instructions extracted from the proof. This additional condition practically means, that arguments of static templates must not be passed as arguments to atoms, if that argument influences the generated instruction. In section 2.4.3 we have introduced the `spec` keyword to denote that an argument of an atom is not used for code generation. That is, static template arguments are allowed to be passed to atoms only in `spec` arguments.

When should one place a piece of temporal proof into a static template? Every time the specification of that code is used more that once. For example, the proof of a procedure should be implemented in a static template. Each time the procedure is called, one can call the static template to use the specified properties of the procedure. An other example is the proof of a loop, as we usually need induction for that. The refinements describing the loop body form the static template, and when the program jumps back to the beginning of the loop, we call the template recursively.

```
static pAll( Integer #n )
{
  #n >= 0 => p(#n)
  select
  {
    #n = 0 => p(#n)
    {
      p0( #n );
    }
    #n > 0 => p(#n)
    {
      #n > 0 => #n-1 >= 0 & #n-1 < #n;
      pAll( #n-1 );
      p(#n-1) & #n-1 >= 0 => p(#n)
      {
        pNext( #n );
      }
    }
  }
}
```

Figure 8: A template implementing an inductive proof

2.4.6 Templates defined in templates

In section 2.4.4 we have seen a template that generates indirect proofs. We want to create a similar one for inductive proofs by generalising the example in the previous section. That is, we will pass the function `p`, the proof of the base case as well as the proof of the inductive case as arguments to that template, and it will generate the inductive proof seen in the previous section.

Our template will generate a name for the static template to define, using the compile time condition `templatename(#name)`. When evaluating this condition, the compiler will bound a fresh templatename to `#name`. The static template defined inside our template is the same as in the previous section, except the proofs of the base and the inductive cases, because these are arguments.

```
block( Integer #x )
{
  p0( #x );
}
```

Figure 9: An unnamed proof accepting an argument

Let us have a look on these proofs, that should now be passed as arguments. In the previous section we used

```
p0( #n );
```

as the base case and

```
pNext( #n );
```

as the inductive case. Notice that we use the argument `#n` of the static template in these proofs. That is, we can not pass these proofs 'as is', because the argument `#n` is not usable outside the static template. A solution is to pass these proofs accepting an argument. The syntax for this[2] is shown on figure 9. We will pass this block as an argument to our template and that will use it inside the static template and pass `#n` to it as an argument.

When the static template is defined, our template should also immediately call it to use the theorem just proved in the static template. This *template-defining-template*, called `induction`, can be seen on figure 10.

Having the `induction` template defined, we can write our inductive proof in a much more elegant way. For arbitrary parameter value `k`, we can prove `p(k)` as on figure 11.

3 Safety considerations

There are several programming errors that make the careless application of meta programming techniques dangerous, when the object-level code is a traditional programming language. This is especially true for low-level meta programming features.

A common example is, when one repeats a piece of code many times in the program using meta programming techniques instead of writing a loop. This

[2]The ability to write proofs accepting arguments (i.e. unnamed templates) is currently under development.

```
template induction( Integer --> Boolean #fun, Integer #arg
                    block(Integer #p) #base,
                    block(Integer #p) #induct )
{
  templatename( #name ) :
  block
  {
    static #name( Integer #n )
    {
      #n >= 0 => #fun(#n)
      select
      {
        #n = 0 => #fun(#n)
        {
          #base( #n );
        }
        #n > 0 => #fun(#n)
        {
          #n > 0 => #n-1 >= 0 & #n-1 < #n;
          #name( #n-1 );
          #fun(#n-1) & #n-1 >= 0 => #fun(#n)
          {
            #induct( #n );
          }
        }
      }
    }
    #name( #arg );
  }
}
```

Figure 10: Template for induction

```
k >= 0 => p(k)
{
  induction( p )
    block( Integer #x )
    {
      p0( #x );
    }
    block( Integer #x )
    {
      pNext( #x );
    }
}
```

Figure 11: An inductive proof using the `induction` template

can result in an extremely large program. The same happens when the programmer inserts the same instructions at several points in the code instead of defining a procedure and calling that each time it is needed. Confusing procedures with syntactically similar meta programming features can also mess up the code so that it produces erroneous behaviour. Compilers of traditional programming languages does not complain on these errors or inefficient solutions, because the program is syntactically correct.

After all this, is it safe to use meta programming techniques for proof generation? Using the techniques presented in this paper carelessly can lead to erroneous proofs. But all the errors in the proof are reported by the proof checker, and in that case, there is no program generated that could be compiled further to an executable. That is, each time the checker accepts the proof, the generated program conforms to its specification.

The difference between traditional program development and programming by proofs is that in the latter case the compiler can check also the behavioural semantics of our program, not just the syntax and static semantics. This additional check makes meta programming a safe tool for proof development.

In addition to this, our meta programming features are high level language constructs. As we have already discussed in section 2.4.1, instantiation of templates is not text-based, but uses the syntax tree provided by the parser. Visibility of arguments, variables and parameter variables are also correctly handled in conformation with the block structure of the proof. Compile time conditions use techniques usually applied in high-level declarative languages.

These features help avoiding the common pitfalls of low level meta programming and makes construction of sound proofs easier.

4 Conclusions and future work

We can conclude, that meta programming techniques are applicable for construction of declarative style proofs. They make the proofs considerably shorter and easier to understand and to maintain. A great advantage of this solution is that there is no need to make the proof system complex in order to provide rules for common proof patterns. Techniques like indirect proving, induction, or temporal logic patterns like proofs for loops or conditional branches can be implemented using templates instead of hard-wiring them into the system core. This improves the reliability of the proof system.

We have also shown that the dangers of meta programming are not a real risk in case of proof development, as proofs are checked anyway and the errors are discovered already in compile time.

We have implemented the features described in this paper, except the possibility of writing unnamed templates (see the footnote in section 2.4.6). The current implementation is done in *C++*. We use the system to specify instructions of imperative programming languages (including more complex ones, like vector or pointer operations) and to develop verified program code using the refinement techniques of the system. The library of tactics is not yet comparable with that of leading theorem provers available [3, 13], but the templates of our language turned out to be useful techniques in building such libraries.

Our plans include further development of the language and to experiment with refinement techniques not only for imperative programs, but also for functional ones. We also intend to use our system to specify and to formally develop DFA-s[3] or Petri-nets.

References

[1] M. Muzalewski, An Outline of PC Mizar, *Fondation Philippe le Hodey*, Brussels, 1993.

[2] M. Wenzel, F. Wiedijk, A Comparison of Mizar and Isar, *Journal of Automated Reasoning*, **29**, 3–4 (2002) 389–411.

[3] Deterministic Finite Automaton

[3] T. Nipkow, L. Paulson, M. Wenzel, Isabelle/HOL – A Proof Assistant for Higher-Order Logic, *LNCS 2283, Springer*, 2002.

[4] P. Corbineau, A declarative proof language for the Coq proof assistant, *Types for Proofs and Programs, LNCS 4941., Springer*, 2008.

[5] G. Dévai, Programming language elements for correctness proofs, *Acta Cybernetica* **18**, (2008) 403–425.

[6] J.-R. Abrial, The B-book: assigning programs to meanings, *Cambridge University Press*, 1996.

[7] U. Berger, H. Schwichtenberg, Program extraction from classical proofs, *LNCS 960, Springer*, (1995) 77–97.

[8] G. Dévai, Z. Csörnyei, Separation logic style reasoning in a refinement based language, *Proceedings of the 7th International Conference on Applied Informatics*, 2007.

[9] G. Dévai, N. Pataki, Towards verified usage of the C++ Standard Template Library, *Proceedings of the 10th Symposium on Programming Languages and Software Tools*, (2007) 360–371.

[10] Freek Wiedijk, Formal proof sketches, *LNCS 3085, Springer*, (2004) 378–393.

[11] B. Beckert, V. Klebanov, Proof reuse for deductive program verification, *Proceedings of Second International Conference on Software Engineering and Formal Methods*, (2004) 77–86.

[12] C. Hunter, P. Robinson, P. Strooper, Flexible Proof Reuse for Software Verification, *LNCS 3116, Springer*, (2004) 211–225.

[13] Y. Bertot, P. Castéran, Interactive theorem proving and program development. Coq'Art: The calculus of inductive constructions, *Texts in Theoretical Computer Science, Springer*, 2004.

Received: October 1, 2008