Acta Cybernetica **18** (2008) 403–425.

Programming Language Elements for Correctness Proofs*

Gergely Dévai†

Abstract

Formal methods are not used widely in industrial software development, because the overhead of formally proving program properties is generally not acceptable. In this paper we present an ongoing research project to make the construction of such proofs easier by embedding the proof system into a compiler.

Using the introduced new programming language, the programmer writes formal specification first. The specification is to be refined using stepwise refinement which results in a proof. The compiler checks this proof and generates the corresponding program in a traditional programming language. The resulting code automatically fulfills the requirements of the specification.

In this paper we present language elements to build specification statements and proofs. We give a short overview on the metaprogramming techniques of the language that support the programmer's work. Using a formal model we give the semantics of specification statements and refinements. We also prove the soundness of the basic algorithms of the compiler.

1 Introduction

1.1 Motivation

The study of formal methods to reason about program properties is getting a more and more important research area, as a considerable part of a software product's lifecycle is testing and bug-fixing. The theoretical basis — such as formal programming models and reasoning rules [16, 15, 20, 7, 17] — has been developed so far, but these are rarely used in industry [6]. The main reason for this fact is that formally proving a program property usually takes much more time than writing the program itself.

The goal of this research is to use programming language elements to make the construction of these proofs easier. The basic idea is to develop a new programming language where the source code contains the formal specification and

*This work is supported by "Stiftung Aktion Österreich-Ungarn (OMAA-ÖAU 66öu2)" and "ELTE IKKK (GVOP-3.2.2-2004-07-0005/3.0)".

†Department of Programming Languages and Compilers, Faculty of Informatics, Eötvös Loránd University, Budapest, E-mail: `deva@elte.hu`

the correctness proof of the implementation. The proofs are built up using *stepwise refinement* [23, 22], as this technique provides *correctness by construction*, and also helps the programmers to make the right decisions during software development. The compiler of the language has to check the soundness of the proof steps and to generate the program code in a target language using the information of the proof.

Similarly to programs, proofs also contain schematic fragments. These can be managed efficiently using *proof templates* that have the same role in proof construction as procedures have in traditional program development. This leads to a special kind of *metaprogramming* [9]: by the instantiation of the templates a proof is constructed (and checked) in compile time and from the proof a target language program is generated which automatically fulfills all the requirements stated in the specification.

1.2 Related work

We summarize existing solutions for formally verified software development and point out how the system presented in this paper differs from these approaches.

The most obvious solution is to embed the programming model in the framework of a theorem prover. Once the program is written, one can (automatically) generate its representation in the prover, formulate the desired properties (specification) and discharge them using the tools of the prover. Theorem provers like *Coq* [5] or *Isabelle/HOL* [24] can be used for this purpose. The problem is, that the program often does not fulfill the specification. In that case one has to start the whole procedure from the beginning by fixing the error in the program code and reconstructing parts of the proof. In contrast, our approach uses the *correctness by construction* principle: the programmer writes the specification first and refines it towards the implementation. Using this method one can discover design errors in an early stage of the development.

Another approach is to extend a programming language by annotations (*JML and ESC/Java2* [8], *SPARK* [4], *FPP* [25]) which express the specification and possibly the key elements of the proof. If the source code is extended by the specification statements only, we need additional tools to discharge the proof-obligations: the previously mentioned problems arise again. If the elements of the proof appearing in the code are detailed enough to enable automatic check of soundness, the source code becomes redundant: a complete proof of correctness usually contains all the necessary information needed to reconstruct the algorithm. This is exactly what is done in the system presented in this paper: the programmer writes specification and proof only, the instructions of the algorithm are "extracted" from the proof by the compiler automatically.

Functional and logic programming have a tight relation to formal methods as programs in these languages can be considered as *executable specifications.* For example in a functional language a program that sums the elements of a list reflects the "natural" definition of the problem very well. In contrast, it is not the case if we consider for example the problem of sorting a list. The "natural" way of specifying it, is to state that we seek a sorted permutation of the original sequence, while (effective) implementations[1] in functional and logic programming languages are closer to the different sorting algorithms used also in imperative programming. This problem motivates several projects developing theorem provers especially for functional programming languages, like the *Lisp*-based *ACL2* [19], and *Sparkle* [10] for the *Clean* language. Theorems proved in these systems express relations between the functions implemented in the functional languages. This means that the construction of the proof takes place after the implementation of the functions. This has the same drawback mentioned before. Furthermore there are essentially "imperative" programming problems (consider for example the IO processes) that are hard to deal with in a purely functional environment. The proof of the soundness of these program fragments require further sophisticated methods [18]. In the language presented in this paper it is possible to specify the problem on an abstract level without any constraints to be "executable", while it also gives the possibility to fully control the effectiveness of the implementation. If we consider the example of sorting, the specification just states that a sorted permutation is needed, and this can be refined towards any of the effective sorting algorithms. It is even possible to choose assembly as the target language and to apply robust optimizations still keeping the program proved correct.

The most similar projects to the one presented in this paper are the *B-method* [3] and *Specware* [21], as both of them uses refinement. The *B-method* uses abstract machines specified by pre- and postconditions of its operations and invariants. The abstract machines are refined towards the implementation, and proof-obligations are generated to each refinement. Additional tools help to construct the proofs. *Specware* uses essentially the same schema, but it uses category theory as its basis. The goal of the project presented here is to keep the essence of these successful approaches while keeping the specification language and the refinement rules as simple and straightforward as possible, eliminating proof assistants and unifying the different levels of refinement.

The main characteristics and contribution of our approach can be summarized as follows:

- It is a refinement-based method to ensure correctness by construction.
- The resulting target language program is generated automatically, the programmer has to develop the specification and its refinements only.
- The compiler and the language is independent of the target language, because code generation is done by a separate module. The compiler's output is a set of state transitions. Any language which is able to implement these transitions can be a target language.
- The proofs are completed using the features of the new language only, no external tools are needed.

[1] It may be also possible to implement the sorting algorithm directly as a search for a sorted permutation (for example in Prolog), but the resulting algorithm is extremely ineffective.

- Reasoning in temporal and classical logic is unified.
- Proof strategies are not hard-wired in the compiler, but can be developed using the metaprogramming techniques provided by the new language.

1.3 Current state of the presented project

The programming language and the underlying system described in this paper is implemented in C++. The compiler currently consists of more than 6000 lines of source code. There are also hundreds of source files written in the new language to test the compiler, and several example programs implementing simple but useful algorithms [1, 13] are constructed for demonstration reasons. A small utility library is developed that contains templates to ease reasoning about loops, conditionals and can automatically construct proofs for expression evaluation. The currently supported target language is C++, but in a previous version the NASM assembly was supported.

As a relatively young project, the system's automated reasoning capabilities are not comparable yet with the power of the leading interactive theorem provers. However the advanced metaprogramming capabilities of the language are quite promising: it makes possible to reuse often used proof parts and to develop own proof strategies.

2 Examples

In this section we informally present the main features of the language using several small examples. The specification language and the refinement possibilities will be presented in more detail (2.1, 2.2), to help the reader to understand their formalization described in sections 3 and 4. On the metaprogramming features of this language and code generation issues we give a brief overview (2.3, 2.4).

2.1 Temporal properties

Expressions of the language are formulas of typed first order logic used to describe states of the program. The instruction pointer (*ip*) is considered as a normal variable and it may also appear in the formulas. For example the expression

```
ip = B & s = "Hello!"
```

states that the program execution is at the label B and the string "$Hello$!" is stored in the variable s. Using the $\gg$ symbol one can connect two such statements to build a *temporal progress property*:

```
ip = A >> ip = B & s = "Hello!";
```

This expresses that whenever the program execution is at label A, it has to reach after some (finite but not certainly bounded) steps label B and then $s = "Hello!"$ must hold.

We can express classical pre- and postconditions of Hoare logic [16] using the reserved labels *Start* and *Stop* instead of A and B in the example. Moreover, the explicit usage of the *ip* variable makes it possible to specify non-terminating programs. Like in the following example

```
ip = A >> ip = A & s = "Hello!";
```

where the "postcondition" implies the "precondition", stating that the program repeatedly returns to the same state or does not leave it.

This temporal property is close to the *leads to* property of *Unity* [7] and its relational version [17], but without supposing any kind of *fairness* of the scheduling. (As this language is currently designed for sequential languages, fairness is not a point.)

In many formal specification systems connections between pre- and postconditions are expressed using *auxiliary variables* (also called *parameter variables*). We also use this technique. For example the parameter variable j is used in the following property

```
ip = Start & i = j >> ip = Stop & i = j+1;
```

to state that this program increments the value of the variable i. Program variables and parameter variables are not distinguished syntactically but their declarations are different. As usually, parameter variables are not allowed to appear in the program, only in specification and proof.

It is also possible to express *safety properties* of the program: these are formulas enclosed between the '[' and ']' symbols. A safety property concerns one or more progress properties (the ones that are in the scope of it, for exact definition see section 3.4).

```
[ i > 0 ];
ip = A >> ip = B;
```

This example means that while the program proceeds from the label A to B, if $i > 0$ becomes true, it remains true at least until $ip = B$ is reached.

If a formula holds throughout a program fragment, in Hoare logic style proofs one has to repeat it in all intermediate steps. In our system one can use a safety property instead.

The *always* operator of temporal logic [20] and *invariant* notion of [17] are too strong, stating that a property must hold during the overall program. Our safety property is closer to the *(weak) unless* operator and the *unless* property of *Unity* and [17]. The difference is that our property must hold "between" a pre- and a postcondition.

2.2 Refining the specification

Progress properties of specifications can be refined by two constructs: *sequence* and *case analysis*. Safety properties are not to be refined, they are checked by the compiler automatically.

A *sequence* breaks a progress property into consecutive steps. In the following example the first line is the property to be refined and the two refining statements are enclosed by the braces.

```
ip = Start >> ip = Stop & s = "Hello!"
{
  ip = Start >> ip = A & s = "Hello!" { ... }
  ip = A & s = "Hello!" >> ip = Stop & s = "Hello!" { ... }
}
```

This refinement states that the original progress property is fulfilled by the program such that it first sets the desired value of the variable s while it steps to the label A, and then it terminates.

Note, that this example contains lots of unnecessary details. The algorithm actually used by the the compiler to check its soundness (see section 4.1) enables us to omit most of this redundancy. Furthermore such a simple refinement can be automatically constructed by proof tactics implemented in the language, so that the only line of the specification would be enough.

A *case analysis* can be used to implement conditionals. In the following example we want to compute the factorial of i. The program first computes the condition $i = 0$ and commits a conditional jump. The precondition of the following example describes the state of the program after this jump: it is at label A if $i = 0$, while it is at the label B if $i \neq 0$. The result is computed differently in each of the two cases, that is why we use case analysis. This is denoted by the *select* keyword.

```
(ip = A & i = 0) | (ip = B & !(i = 0))
>> ip = C & f = fact(i)
select
{
  ip = A & i = 0 >> ip = C & f = fact(i) { ... }
  ip = B & !(i = 0) >> ip = C & f = fact(i) { ... }
}
```

The soundness of such a refinement is checked by the algorithm presented in section 4.2.

These two refinement constructs are very close to the *Hoare logic* rules [16] for program sequences and *if* statements. Similar rules are the *transitivity* and *disjunctivity* of the *leads to* operator of *Unity* [7, 17].

2.3 Template features

The basic idea of this language is that the programmer builds specification and proof using the previously presented properties and refinement constructs, then the compiler checks their soundness and generates the corresponding program in a target language. The programmer's work is supported by a metaprogramming layer of the language consisting of *templates* and *compile-time conditions*.

Templates are often used or valuable proof parts which are parametrized. These templates can be called by the programmer with arguments to obtain a concrete proof fragment.

Template arguments can be examined by *compile-time conditions*. Depending on these conditions a template call may result in different proof fragments. For example we have constructed a template to generate proof for expression evaluation. The expression to be computed by the program is an argument of the template. Compile time conditions examine whether this expression is a constant, a variable or a function application etc. Depending on the kind of the expression, a proof of an assignment instruction or a proof of a function call is produced by the template.

Formally defining the semantics of templates is not in the scope of this paper. We give a brief overview of template substitutions. If a template is called, the arguments are type checked first, then every occurrence of the formal arguments in the template definition is replaced by the corresponding actual ones. Compile-time conditions are evaluated next. Proof parts with false conditions are left out, and the template call is replaced by the remaining parts.

Templates and compile-time conditions are similar to the *macro features* of *macro assemblers* like MASM [2]. However, our templates are type checked. Similarly to macro assemblers, our metaprogramming constructs can also be used to simulate higher level programming constructs like control structures, procedures, exceptions etc. While macros of a macro assembler generate assembly instructions implementing the constructs, our templates generate their proofs. In order to achieve this goal we developed several kinds of templates. In the following we give a brief overview of them.

2.3.1 Temporal and classical axioms

Templates marked with the *axiom* or *atom* keywords contain classical or temporal axioms respectively. The programmer is able to declare functions and predicates to use in specifications and proofs and can state their mathematical properties in axiom templates. Atom templates contain temporal properties of instructions of the target language, like an assignment or procedure call.

2.3.2 Tactics

The *tactic* keyword introduces a template that can be called by the compiler automatically. These templates have to have exactly two *boolean* arguments. If the compiler finds a non-refined progress property (which is not in an axiom or atom) it calls the available tactics with the pre- and postcondition of the progress property as arguments. If none of the tactics provide a valid refinement for the property, an error-message is generated.

2.3.3 Static templates

If the programmer marks a template by the *static* keyword, the compiler checks the soundness of its refinements regardless of its arguments. The soundness of these refinements and the set of program instructions generated from them are not allowed to depend on the actual arguments of the template. If this is violated, the compiler generates an error message.

As a result, when a static template is called, there is no need to check its contents again. This makes it possible to implement induction with static templates. Proofs of loops and procedures are usually placed inside static templates, as induction is often needed to prove their soundness.

2.3.4 Passing proof fragments as arguments

Templates usually get expressions as arguments, but it turned out to be quite useful to pass complete blocks of refinements too. Using this possibility we were able to define templates that generate proofs for *if-statements* and for different kinds of *loops*. The following example is a sketch of computing the absolute value of i. We call the *if* template, which gets two "simple" arguments: the condition of the branch, and the postcondition that is to be established. It also has two "special" arguments: the proofs of the *if-* and *else-branch*.

```
if( i < 0, j = abs(i) )
{
  // proof of if-branch
}
{
  // proof of else-branch
}
```

2.3.5 Templates declared in templates

It is possible to declare templates inside other templates. For example we were able to write a template that can be used to declare procedures: when it is called, it generates two other templates, one static template with the proof of the procedure itself, and another template with the proof of the procedure call.

2.4 Code generation

When the compiler checks the refinements and finds a temporal progress property axiom, it saves the corresponding atom template call to a set. This set of template calls is the compiler's output. A separate code generation module converts it to the syntax of the target language.

Most template calls in the set contain the label of the corresponding instruction and the label of the following instruction. Template calls coresponding to instructions like *goto* and *return* contain their own label only, because these instructions do not pass the control flow to the instruction after them.

That is, these labels define a *partial order* on the set of template calls. The code generator sorts the instructions according to this order and generates the target language code.

3 Semantics of the specification language

In this section we present the model that is the semantic domain of the specification statements of the language. We use this model to prove certain properties of the temporal statements. In section 4 these properties will be used to prove the correctness of the algorithms used by the compiler to check refinements of the language.

3.1 Expressions and logic

Expressions in this language are typed first order logic formulas. The free variables of the formulas are program variables and parameter variables. These variables define a state space that the formulas are interpreted on. The programming model introduced in section 3.2 uses this state space to describe the behavior of programs.

The detailed presentation of the syntax and semantics of the expressions of this language can be found in [11].

3.2 Underlying programming model

The semantics of the safety and progress properties is given using a relational programming model, that we present in this section. The rules that the stepwise refinement is based on are also proved in this model.

3.2.1 State space, program

Let A be an arbitrary set, the *state space.* A *program* over A is a set of *state transitions*:

$$S \subseteq A \times A$$

In case of $(a, b) \in S$, the program S can change its state form a to b.

In this model the *instruction pointer* of a program is a component of the state space, just as all other variables. For example the program

```
K:  b = true;
L:  b = false;
M:
```

operates on a two-component state space, $A = \{K, L, M\} \times \{true, false\}$ and has two variables, *ip* and b respectively. It has four state transitions:

$$S = \{((K, false), (L, true)), ((K, true), (L, true)),$$

$$((L, false), (M, false)), ((L, true), (M, false))\}.$$

3.2.2 Operation of programs

The operation of a program can be described by the state sequences that the program follows during its execution. We use the notation A^{**} for the set of all (finite or infinite) nonempty sequences over the set A. The *operation* of program S over state space A is the following subset of A^{**}[2]:

$$r_S = \{\alpha \in A^{**} \mid \forall i \in [1..|\alpha|-1] : (\alpha_i, \alpha_{i+1}) \in S \wedge (|\alpha| < +\infty \rightarrow \alpha_{|\alpha|} \notin \mathcal{D}_S)\}$$

This definition states that the program changes its state according to its transitions and it stops iff there is no applicable state-transition. Note that each α' postfix of a sequence $\alpha \in r_S$ is in r_S too.

For example, the sequences

$$< (K, false), (L, true), (M, false) >$$

and

$$< (L, true), (M, false) >$$

are valid for the example program presented in 3.2.1, but the sequences

$$< (L, false), (M, true) >$$

and

$$< (K, false), (L, true) >$$

are not.

The notation $F(Q, \alpha)$ is used for the *first occurrence* of an element in the sequence $\alpha \in A^{**}$ for which the statement Q holds.

$$F(Q,\alpha) = \begin{cases} i \in \mathcal{D}_\alpha & \text{if } Q(\alpha_i) \wedge \forall j \in [1..i-1] : \neg Q(\alpha_j) \\ +\infty & \text{if } \forall j \in \mathcal{D}_\alpha : \neg Q(\alpha_j) \end{cases}$$

This notation will be used to define the temporal properties of programs.

3.2.3 Temporal properties of programs

Let S be a program and P, Q and K be statements over the state space A. S *leads to* Q from P ($P \gg_S Q$), iff

$$\forall \alpha \in r_S : (P(\alpha_1) \rightarrow F(Q, \alpha) < +\infty).$$

That is, if the program is in a state for which the statement P holds it will reach some state where Q holds after a finite (but not certainly bounded) number of state transitions. The statement K is a *safety property* of S between P and Q ($[K]_S^{P,Q}$) iff

$$\forall \alpha \in r_S : (P(\alpha_1) \rightarrow \forall j \in [F(K,\alpha)..F(Q,\alpha)] \cap \mathcal{D}_\alpha : K(\alpha_j)).$$

[2] $\mathcal{D}_S = \{a \in A \mid \exists b \in A : (a,b) \in S\}$ is the domain of the relation S.

That is, if the program reaches some state where K holds while it proceeds form P to Q, then K remains true at least until Q is reached.

For example the properties

$$ip = K \gg_S ip = M$$

and

$$[b = true]_S^{(ip=K),(ip=L)}$$

hold for the example program in 3.2.1.

3.2.4 Temporal properties with parameters

Recall the example of section 2, where we used a parameter variable to express a progress property for each integer j:

```
ip = Start & i = j >> ip = Stop & i = j+1;
```

In general, let S be a program over state space A, and B be an arbitrary set, the *parameter space*, $C = A \times B$, and P, Q and K be statements over C. If $b \in B$, we use the notation P^b for the statement over A for which

$$\lceil P^b \rceil = \{a \in A \mid (a, b) \in \lceil P \rceil\}$$

holds. We say, that

$$P \gg_S Q \text{ and } [K]_S^{P,Q}$$

is true iff for every $b \in B$

$$P^b \gg_S Q^b \text{ and } [K^b]_S^{P^b,Q^b}$$

hold respectively.

3.2.5 Refinement rules

Using the relational model we introduce rules of the temporal properties. These rules are the basis for the algorithm that the compiler uses to check the refinement steps in the source code.

The proofs of these rules are not really difficult but rather technical. You can find them in the technical report [12]. Here we give short informal proofs and explanations.

In the following we suppose that S is an arbitrary program over the state space A, the parameter space is B, and $C = A \times B$. Furthermore we suppose that P, Q, R and K are arbitrary statements over C.

Rule of consequence

If $P \Rightarrow Q$ then $P \gg_S Q$ and $[K]_S^{P,Q}$.

To show this rule, we must take the sequences from r_S starting with an element satisfying P. But these elements also satisfy Q and using the definitions of the temporal properties we get what the rule states.

The condition of the rule states that each time the precondition holds, the postcondition also holds immediately. That is why any program can be used to reach the postcondition from the precondition. The same rule is present in the *Unity* based models [7, 17] for the *leads to* operator. In [16] Hoare had two such rules: one for the precondition and one for the postcondition. Both of those rules can be derived from our one and the *rule of sequence*.

Rule of sequence

If $P \gg_S Q$ and $Q \gg_S R$ then $P \gg_S R$.
If $[K]_S^{P,Q}$ and $[K]_S^{Q,R}$ then $[K]_S^{P,R}$.

To deal with the claim about the progress properties is quite simple: in each sequence starting with an element satisfying P, we can find an element for that Q holds, because of the first hypothesis. And then, by the second hypothesis we know that there is an element in the sequence for which R is true.

To prove the second statement we must explore cases depending on the order of the first occurrence of Q, R and K. In each case by using one or two of the hypothesis we can prove the statement.

This rule is essentially the rule of *Hoare logic* for program sequences and the transitivity of *leads to* in *Unity*.

Rule of case analysis

If $P \gg_S R$ and $Q \gg_S R$ then $P \vee Q \gg_S R$.
If $[K]_S^{P,R}$ and $[K]_S^{Q,R}$ then $[K]_S^{P \vee Q,R}$.

To prove this rule it is enough to consider, that if the first element of a sequence satisfies $P \vee Q$, then it satisfies either P or Q. In each case we can use the corresponding hypothesis to prove the claim. This reasoning can be applied for both statements.

This rule can be used to build proof for conditionals in a program. It splits the precondition into parts and allows the programmer to reach the postcondition in different ways from these parts. The corresponding rules are the disjunctivity of *leads to* and the *Hoare rule* for *if statements*.

Rule of safety property application

If $P \gg_S Q$ and $[K]_S^{P,Q}$ then $(P \wedge K) \gg_S (Q \wedge K)$.
If $[I]_S^{P,Q}$ and $[K]_S^{P,Q}$ then $[I]_S^{P \wedge K, Q \wedge K}$.

The core of both statements is that if K is a safety property between P and Q, then starting from $P \wedge K$, if we reach Q, then $K \wedge Q$ will hold. In the first statement we additionally suppose that Q is surely reached, which means $Q \wedge K$ is reached. Similar reasoning applies for the second statement.

In Hoare logic proofs all the unchanged parts of the assertions are present in every step of the proof. Using the rule described here we can "save" these unnecessary parts to safety properties and "put them back" into the progress properties when necessary. In *Unity* a similar rule describes the connection between the *leads to* operator and *invariants* of the program.

Rule of composition

Suppose that $S = S_1 \cup S_2$ and $\mathcal{D}_{S_1} \cap \mathcal{D}_{S_2} = \emptyset$.
If $P \gg_{S_1} Q$ then $P \gg_S Q$.
If $[K]_{S_1}^{P,Q}$ and $P \gg_{S_1} Q$ then $[K]_S^{P,Q}$.

Because the program S_1 reaches Q from P, every state on this way must be in the domain of S_1. Thus, by the crucial condition that the domains of the two composed programs are disjoint, these states can not be in the domain of S_2. From this we get that the compound program does exactly the same from P to Q as S_1 does. From this follows both claims of this rule.

Note that the disjointness can easily be fulfilled in case of sequential programs, but it is much harder for parallel/concurrent ones. Similar rules are established in *Unity.* In *Hoare logic*, this rule is implicitly present in each of its rules, as they are all compositional. The Hoare-style sequencing rule can be emulated in this model by first applying our *composition rule* for both programs and then applying our *sequencing rule.*

3.3 Syntax of proofs

In this paper we do not deal with the formal description of the operation of templates. After processing the meta programming elements in the code, the resulted proof consists of specification statements and their refinements. In this section we present the syntax of these elements.

In the grammar the following notations are used: non-terminal symbols are enclosed between the < and > symbols, alternatives are divided by the | symbol, the [and] symbols enclose optional parts, while [and]* denotes iteration (0, 1 or more times), terminals appear between single quotes.

```
<proof> ::=
  [ <safety property> | <temporal axiom> | <classical axiom>
  | <sequence> | <case analysis>
  | <conclusion refinement> ]*

<safety property> ::=
  '[' <expression> ']' ';'

<temporal axiom> ::=
  [ [ <condition> ':' ] <safety property> ';' ]*
  <expression> '>>' <expression> ';'

<classical axiom> ::=
  <expression> '=>' <expression> ';'

<sequence> ::=
  <expression> '>>' <expression> '{' <proof> '}'

<case analysis> ::=
  <expression> '>>' <expression> 'select'
  '{' [ <sequence> | <case analysis>
      | <conclusion refinement> ]* '}'

<conclusion refinement> ::=
  <expression> '=>' <expression> [ 'select' ]
  '{' [ <conclusion axiom> | <conclusion refinement> ]* '}'
```

That is, the proof is a sequence of safety and progress properties and conclusions. Each progress property and conclusion has to be refined, unless it is a progress property axiom or a conclusion axiom. These axioms are always produced by a template containing temporal or first order logic axioms. Conditions in safety property axioms are special expressions that can be computed in compile time.

3.4 Semantics of statements

Now we connect the statements of the language with the model presented in section 3.2. First, we define the state space, that the formulas are interpreted on. Let the $\mathcal{A} = \{v_1, ..., v_n\}$ be the set of program variables and $\mathcal{B} = \{p_1, ..., p_m\}$ be the set of parameter variables in the proof, and let V_i and P_j denote the sets of values corresponding to the types of v_i and p_j respectively. Then the formulas are interpreted on the space $(V_1 \times ... \times V_n) \times (P_1 \times ... \times P_m)$.

Let S denote the model of the specified program on state space $V_1 \times ... \times V_n$. A progress property $Q \gg R$ of the proof specifies that $Q \gg_S R$ has to be fulfilled by S.

In the grammar of section 3.3 the sequence of statements directly derived from the $< proof >$ symbol is called a block. The scope of a safety property consists of the statements from the location of the safety property to the end of the innermost block that contains it. If the progress properties $Q_1 \gg R_1$, ..., $Q_n \gg R_n$ are in the scope of the safety property $[K]$, it specifies, that S fulfills $[K]_S^{Q_1,R_1}$, ..., $[K]_S^{Q_n,R_n}$.

A safety property axiom $c : [K]$; may contain expression variables. We say that $[K']$ is stated by the axiom if it is possible to assign expressions to the expression variables such that replacing them in K results in K', and the condition c is true for this assignment. If the temporal axiom consists of $c_1 : [K_1]$; $\ldots c_n : [K_n]$; $P \gg Q$; then the axiom specifies $P \gg_S Q$, and for each $[L]$ that is stated by one of the safety property axioms, $[L]_S^{P,Q}$ is specified too.

We say that a refinement is sound, if each program that fulfills the refining properties, also fulfills the refined properties. In the next section we present algorithms to check refinements, and prove that each refinement accepted by these algorithms is sound in the sense of the previous definition.

4 Algorithms to check refinements

In this section we present the algorithms of the compiler used to check the soundness of refinement steps. In the pseudo codes we use the following conventions. Parameters are always passed by value, that is, the procedures do not have side-effects, results are given by return values only. We use set variables with the usual set operations, and stacks with *push*, *pop* and *top* operations. In section 4.5 the function *sizeof* is also used to give the number of elements in a stack. If T is a progress property, we use the notations $pre(T)$ and $post(T)$ to denote the pre- and postconditions of T respectively.

In the algorithms the procedure $infer(P, Q)$ is called. This can be any algorithm that tries to infer the formula Q from P. The only requirement is, that it has to be sound, that is, if it returns true then $P \Rightarrow Q$ has to hold. Of course, this procedure can not be complete, because first order logic is not decidable.

An other algorithm, $GCNF(P)$ is also used in the algorithms. It transforms the formula P to a *generalized conjunctive normal form.* The exact form of this GCNF and the *infer* algorithm currently used in the compiler are described in [11].

4.1 Processing sequential refinements

Algorithm: $process - sequent(Stm, \mathcal{K}, V)$
Parameters: *Stm*: the statement to process, $\mathcal{K}$: set of formulas, V: stack of formulas
Local variables: T: statement, P: formula
Return value: stack of formulas

```
1. $V := push(V, GCNF(pre(Stm)))$; $T$ :=the first refining statement;

2. if $T$ is the first statement of an axiom then call
   $V, T := process - axiom(T, \mathcal{K}, V)$;
   go to step 8;
```

```
3. if $T$ is a safety property $[K]$ then $\mathcal{K} := \mathcal{K} \cup \{K\}$; go to step 8;
4. if $infer(top(V), pre(T))$ returns $false$ then return ERROR;
5. if $T$ is a sequential refinement then call
   $V := process - sequent(T, \mathcal{K}, V)$;
   go to step 7;
6. if $T$ is a refinement by case analysis then call
   $V := process - select(T, \mathcal{K}, V)$;
   go to step 7;
7. $P := top(V)$; $V := pop(V)$; $V := push(V, GCNF(P \& post(T)))$;
8. if $T$ is the last refining statement in $Stm$,
   a) then go to step 9;
   b) else $T$ :=the next statement of the refinement; go to step 2;
9. if $infer(top(V), post(Stm))$ returns $false$ then return ERROR;
10. return $pop(V)$;
```

4.2 Processing refinements by case analysis

Algorithm: $process - select(Stm, \mathcal{K}, V)$
Parameters: Stm: the statement to process, $\mathcal{K}$: set of formulas, V: stack of formulas
Local variables: T: statement, P: formula
Return value: stack of formulas

```
1. $D$ :=empty disjunction; $T$ :=the first refining statement;
2. $V := push(V, GCNF(pre(Stm)))$; $D := D \mid pre(T)$;
3. if $T$ is a sequential refinement then call
   $V := process - sequent(T, \mathcal{K}, V)$;
   go to step 5;
4. if $T$ is a refinement by case analysis then call
   $V := process - select(T, \mathcal{K}, V)$;
   go to step 5;
5. $P := top(V)$; $V := pop(V)$; $V := push(V, GCNF(P \& post(T)))$;
6. if $infer(top(V), post(Stm))$ returns $false$ then return ERROR;
7. $V := pop(V)$;
8. if $T$ is the last refining statement in $Stm$,
        a) then go to step 9;
        b) else $T$ :=the next statement of the refinement; go to step 2;
 9. if $infer(pre(Stm), D)$ returns $false$ then return ERROR;
10. return $V$;
```

4.3 Processing axioms

Algorithm: $process - axiom(Stm, \mathcal{K}, V)$
Parameters: *Stm*: the first statement to process, $\mathcal{K}$: set of formulas, V: stack of formulas
Local variables: $\mathcal{M}$: set of statements, U: formula, W: stack of formulas
Return value: stack of formulas, statement

```
1. $\mathcal{M} := \emptyset$; $W$ :=empty stack;
2. if $Stm$ is a safety property axiom $[M]$
    a) then $\mathcal{M} := \mathcal{M} \cup \{M\}$; $Stm$ :=the next statement; go to step 2;
    b) else go to step 3;
3. if $infer(top(V), pre(Stm))$ returns $false$ then return ERROR;
4. for each element $K \in \mathcal{K}$: if $\exists M \in \mathcal{M}$: $check - safety - property(K, M)$
   returns $false$ then return ERROR;
5. for each element $F = F_1\&...\&F_n$ of $V$ (from the bottom to the top):
    a) for each $F_i$ ($i \in [1..n]$):
       if $\exists M \in \mathcal{M}$: $check - safety - property(F_i, M)$ returns $false$ then
       $F$ :=remove $F_i$ from $F$;
    b) $W := push(W, F)$;
6. $U := top(W)$; $W := pop(W)$; $W := push(W, GCNF(U\&post(Stm)))$;
7. return $W$, $Stm$;
```

4.4 Using safety property axioms

Algorithm: $check - safety - property(K, L)$
Parameters: K: formula, L: safety property axiom statement (of the form $c : [I]$)
Return value: boolean

```
1. Try to assign an expression to the expression variables in $I$ such than $K$ and
   $I$ match. If it is not possible then return $false$;
2. Evaluate the condition $c$ with the assigned expressions. If it is $true$ return
   $true$, else return $false$.
```

4.5 Soundness of the algorithms

In this section we present a theorem that states the soundness of the presented algorithms, and three lemmas that are used in the proof of the theorem. The proofs can be found in appendix **A.**

Theorem. If the refinements in a (finite) proof are accepted by the algorithms presented in sections 4.1–4.4, and a program fulfills all the axioms used in the proof, then the program fulfills all the temporal properties appearing in the proof.

Lemma 1. If a program S fulfills an axiom with properties $c : [I]$; and $P \gg Q$ and $check - invariant(K, c : [I])$ returns $true$, then $[K]_S^{P,Q}$ also holds.

Lemma 2. If the call $W, Stm' := process - axiom(Stm, \mathcal{K}, V)$ processes the axiom consisting of statements $c_1 : [I_1](= Stm)$, $c_2 : [I_2]$, ..., $c_n : [I_n]$, $P \gg Q$ without returning an error and the program S fulfills these properties, then

- $top(V) \gg_S top(W)$,
- $\forall K \in \mathcal{K} : \ [K]_S^{top(V),top(W)}$,
- $\forall i \in [1..sizeof(pop(V)) - 1]$ for the i^{th} elements F_i of $pop(V)$ and G_i of $pop(W)$: $F_i \Rightarrow G_i$ and $[G_i]_S^{top(V),top(W)}$ is true.

Lemma 3. If the call $W := process - sequent(Stm, \mathcal{K}, V)$ or $W := process - select(Stm, \mathcal{K}, V)$ accepts a refinement without error, and the program S fulfills all the properties inside the refinement, then the following hold:

- $pre(Stm) \gg_S post(Stm)$,
- $\forall K \in \mathcal{K} : \ [K]_S^{pre(Stm),post(Stm)}$,
- $\forall i \in [1..sizeof(V)]$ for the i^{th} elements F_i of V and G_i of W: $F_i \Rightarrow G_i$ and $[G_i]_S^{pre(Stm),post(Stm)}$.

5 Summary

The project presented in this paper experiments with two aspects of formal methods:

- embedding of a refinement based calculus into a compiler to produce code correct by construction,
- and using metaprogramming techniques to make proof construction easier.

In the current paper we discussed the first aspect. Semantics of specification statements were presented as well as the basic refinement-checking algorithms of the compiler together with their proofs of correctness.

Further research efforts have been issued to test the flexibility of our programming model and specification language. We embedded a model to reason of dynamic memory management and pointers [13], and also some datatypes of the C++ Standard Template Library and their basic operations with iterators were specified in this language [14]. These embeddings were possible without modifying the compiler and language design. Therefore we concluded that it is flexible and expressive enough.

In this paper we gave only a brief overview of the metaprogramming toolset of this language. Our current research concentrates on supporting the programmers' work by these tools. We investigate how to emulate higher level proof rules by templates.

There are also interesting research areas for later development of this work. It would be useful to extend the expressive power of the specification statements, for example to specify randomized algorithms, parallel programs, resource consumption of the program etc.

In its current state this system in already applicable to specify programming problems and to derive not-too-complicated algorithms as verified solutions. The limitation is clearly the non-sufficient automatic reasoning capabilities of the system. We experiment with the metaprogramming features of the language to overcome this limitation. While most formal methods use their built-in provers as black boxes, in our case most of the proof strategies are implemented not in the compiler but using the language itself. They are accessible and extendable.

A Proof of theorems of section 4.5

A.1 Proof of the theorem

We prove the theorem by structural induction on the structure of the proof tree.
In the base case we observe a refinement where all the refining statements are axioms. By the assumption of the theorem, the program S fulfills all these axioms. From this, by lemma 3 we get that S fulfills the refined properties too.
In the inductive case, by the induction assumption the program fulfills all the properties inside a refinement. From this, by lemma 3 we get that S fulfills the refined properties too.

A.2 Proof of lemma 1

If the call returned *true*, it means that it was possible to assign expressions to the expression variables such that K and I matched and the condition was also *true*. Using the semantics described in section 3.4 it means that K is stated by $c : [I]$, and $[K]_S^{P,Q}$.

A.3 Proof of lemma 2

Step 2 of the algorithm collects all the safety property axioms into the set $\mathcal{M}$. Because the algorithm processed the axioms without error, by step 4 we know that $\forall K \in \mathcal{K} : [K]_S^{P,Q}$.

Step 5 copies the elements of V to W in such a way that it removes certain parts of the conjunctive chains, thus $\forall i \in [1..sizeof(V)] : F_i \Rightarrow G_i$, where F_i and G_i are the i^{th} elements of V and W respectively. By lemma 1 we get that the removed parts are those that are not safety properties of the axiom. It means that $\forall G \in W : [G]_S^{P,Q}$. In the following steps the algorithm modifies only the top element of W, so at the end we have $\forall i \in [1..sizeof(pop(V))] : F_i \Rightarrow G_i$ and $[G_i]^{P,Q}$, where F_i and G_i are the i^{th} elements of $pop(V)$ and $pop(W)$ respectively.

Let us denote the value of $top(W)$ by T at the end of step 5. Thus, we also have $top(V) \Rightarrow T$ and $[T]_S^{P,Q}$. Because the call did not return an error, by step 3 we know that $top(V) \Rightarrow P$, and because of $top(V) \Rightarrow T$ also $top(V) \Rightarrow P\&T$. By the assumption of the lemma we know that $P \gg_S Q$ holds. Using the rule of safety property application we get that $P\&T \gg_S Q\&T$. In step 6 the algorithm changes $top(W)$ from T to $Q\&T$, that is we have $P\&T \gg_S top(W)$. From this and from $top(V) \Rightarrow P\&T$ by the rules of conclusion and sequence we get $top(V) \gg_S top(W)$. Using the safety property parts of the same rules give:

$\forall G \in pop(W) : [G]_S^{top(V),top(W)}$ and $\forall K \in \mathcal{K} : [K]_S^{top(V),top(W)}$.

A.4 Proof of lemma 3

We prove the lemma by structural induction on the structure of the proof tree.

The base case is a refinement where all the refining statements are axioms. From the syntax presented in section 3.3 follows that such a refinement is a sequential one. First, we prove, that each time when the algorithm $process-sequent(Stm', \mathcal{K}', V')$ is at step 2, then the following loop invariants hold: $pre(Stm') \gg_S top(V)$, $\forall K \in \mathcal{K}' : [K]_S^{pre(Stm'),top(V)}$ and $\forall i \in [1..sizeof(V')]$ for the i^{th} element $F_i \in V'$ and $G_i \in V$: $F_i \Rightarrow G_i$ and $[G_i]_S^{pre(Stm'),top(V)}$. When the algorithm is at step 2 for the first time $top(V) = pre(Stm')$, because of the initialization in step 1, so the loop invariants hold because of the rule of conclusion. By lemma 2 we get that step 2 preserves these invariants. As, according to our assumption, there are axioms in this refinement only, step 2 and 8 are repeated until we reach the end of the refinement. Then by step 9 we get that $top(V) \Rightarrow post(Stm')$. From this and the loop invariants, using the rules of conclusion and sequence we get the properties that we wanted to prove.

In the inductive case of the proof we have two cases: the cases of refinements by sequence and case analysis.

If the refinement is sequential, then the proof is similar to the base case including the loop invariant, but we have to deal with steps 3-7 too. In step 3 $\mathcal{K}$ is changed but in the loop invariant $\mathcal{K}'$ is present, so that is preserved. If the execution is at step 4 then we know that T is a progress property, and that $top(V) \Rightarrow pre(T)$. Then, depending on the type of the refinement of T step 5 or 6 is executed. We use the induction assumption and a proof similar to the end of the proof A.3 to show that the loop invariant is preserved.

If the refinement is a case analysis, let V_i^1 and V_i^2 denote the value of the stack V at the end of step 2 and step 6 respectively at the i^{th} refining statement T_i. By the induction assumption at steps 3 and 4 we get that $\mathit{pre}(T_i) \gg_S \mathit{post}(T_i)$, $\forall K \in \mathcal{K} : [K]_S^{\mathit{pre}(T_i),\mathit{post}(T_i)}$ and $\forall i \in [1..\mathit{sizeof}(V_i^1)]$ for the j^{th}element $F_j \in V_i^1$ and $G_j \in V_i^2$: $F_j \Rightarrow G_j$ and $[G_j]_S^{\mathit{pre}(T_i),\mathit{post}(T_i)}$. Thus we have $\mathit{pre}(\mathit{Stm}) \Rightarrow \mathit{top}(V_i^2)$ and $[\mathit{top}(V_i^2)]_S^{\mathit{pre}(T_i),\mathit{post}(T_i)}$. From the latter one by the rule of safety property application we get $\mathit{pre}(T_i)\&\mathit{top}(V_i^2) \gg_S \mathit{post}(T_i)\&\mathit{top}(V_i^2)$, $\forall K \in \mathcal{K} : [K]_S^{\mathit{pre}(T_i)\&\mathit{top}(V_i^2),\mathit{post}(T_i)\&\mathit{top}(V_i^2)}$ and $\forall G \in V_i^2$:$[G]_S^{\mathit{pre}(T_i)\&\mathit{top}(V_i^2),\mathit{post}(T_i)\&\mathit{top}(V_i^2)}$. As step 5 changes $\mathit{top}(V)$ from $\mathit{top}(V_i^2)$ to $\mathit{post}(T_i)\&\mathit{top}(V_i^2)$ by step 6 we get that $\mathit{post}(T_i)\&\mathit{top}(V_i^2) \Rightarrow \mathit{post}(\mathit{Stm})$. From this by the rules of consequence and sequence we have $\mathit{pre}(T_i)\&\mathit{top}(V_i^2) \gg_S \mathit{post}(\mathit{Stm})$, $\forall K \in \mathcal{K} : [K]_S^{\mathit{pre}(T_i)\&\mathit{top}(V_i^2),\mathit{post}(\mathit{Stm})}$ and $\forall G \in V_i^2$:$[G]_S^{\mathit{pre}(T_i)\&\mathit{top}(V_i^2),\mathit{post}(\mathit{Stm})}$, which is true for each statement T_i in the refinement. Using the rule of disjunction $n-1$ times, we get $(\mathit{pre}(T_1)\&\mathit{top}(V_1^2)) \mid ... \mid (\mathit{pre}(T_n)\&\mathit{top}(V_n^2)) \gg_S \mathit{post}(\mathit{Stm})$ and the similar safety properties.

Because of step 2 we know that at step 9 $D = \mathit{pre}(T_1) \mid ... \mid \mathit{pre}(T_n)$, and step 9 checks that $\mathit{pre}(\mathit{Stm}) \Rightarrow D$. From this, by $\mathit{pre}(\mathit{Stm}) \Rightarrow \mathit{top}(V_i^2)$ we get that $\mathit{pre}(\mathit{Stm}) \Rightarrow (\mathit{pre}(T_1)\&\mathit{top}(V_1^2)) \mid ... \mid (\mathit{pre}(T_n)\&\mathit{top}(V_n^2))$ is also true. Using the rule of consequence and the rule of sequence for this and for the previous result we get the properties of the lemma.

References

[1] Home of LaCert: http://deva.web.elte.hu/LaCert.

[2] Home of MASM: http://masm32.com/.

[3] J.-R. Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, New York, NY, USA, 1996.

[4] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison Wesley, 2003.

[5] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[6] J. P. Bowen and M. G. Hinchey. Ten commandments revisited: a ten-year perspective on the industrial application of formal methods. In *FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 8–16, New York, NY, USA, 2005. ACM Press.

[7] K. M. Chandy and J. Misra. *Parallel Program Design, A Foundation.* Addison-Wesley, 1988.

[8] D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362/2005, pages 108–128. Springer, 2005.

[9] M. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[10] M. de Mol, M. van Eekelen, and R. Plasmeijer. Theorem proving for functional programmers, Sparkle: A functional theorem prover. *LNCS*, page 55, 2001.

[11] G. Dévai. Programming language elements for proof construction. In *Volume of abstracts of the 6th Joint Conference on Mathematics and Computer Science*, 2006.

[12] G. Dévai. Refinement rules of LaCert. Technical report, Dept. of Programming Languages and Compilers, Fac. of Informatics, ELTE University, 2007.

[13] G. Dévai and Z. Csörnyei. Separation logic style reasoning in a refinement based language. In *Proceedings of the 7th International Conference on Applied Informatics (to appeare)*, 2007.

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

[15] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[17] Z. Horváth. *A Relational Model of Parallel Programs (in Hungarian)*. PhD thesis, Phd School in Informatics, Eötvös Loránd University, Budapest, Hungary, 1996.

[18] Z. Horváth, T. Kozsik, and M. Tejfel. Extending the Sparkle core language with object abstraction. *Acta Cybernetica*, 17:419–445, 2005.

[19] M. Kaufmann, J. S. Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[20] F. Kröger. *Temporal Logic of Programs*. Springer, Berlin, Heidelberg, 1987.

[21] J. McDonald and J. Anton. Specware - producing software correct by construction, 2001.

[22] C. Morgan. *Programming from specifications*. Prentice Hall International (UK) Ltd., second edition, 1994.

[23] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comput. Program.*, 9(3):287–306, 1987.

[24] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[25] J. Winkler. The frege program prover FPP. *Internationales Wissenschaftliches Kolloquium*, 42:116–121, 1997.