Reasoning with Executable Specifications

Yves Bertot and Ranan Fraer

INRIA – Sophia Antipolis,
2004, route des Lucioles,
06902 Sophia Antipolis, France
e-mail: {bertot,rfraer}@sophia.inria.fr

Abstract. Starting from the specification of a small imperative programming language, and the description of two program transformations on this language, we formally prove the correctness of these transformations. The formal specifications are given in a single format, and can be compiled into both executable tools and collections of definitions to reason about into a theorem prover. This work is a case study of an environment integrating executable tool generation and formal reasoning on these tools.

1 Introduction

Two important areas of computer science have shown some interest in formal specifications of programming languages. One area is that of programming environment generators like Centaur [Jac92], the Synthesizer Generator [RT88], or ASF+SDF [Kli93] where one is interested in deriving programming tools from an abstract description of the programming language. Users of the programming environment generators can thus provide short descriptions of tools, which are then obtained by some compilation process from these descriptions.

In the other area interested in formal specifications, the central tool is a proof tool like Coq [DFH+93], HOL [MT92], or Elf [Pfe89] rather than a programming environment. Here, the actual derivation of programming tools is less relevant than the ease with which one can describe a programming language, feed this description to a theorem proving system, and get this theorem proving system to produce a proof for statements about the language.

Because of the distance between these two areas, there is no insurance that the same notion of "formal specification" is shared by speakers from all sides. The work described in these notes attempts to unite the two sides: we are going to provide formal specifications that will be both integrated in practical programming environments and abstract enough to reason about formally. In this respect we follow Berry's "What you prove is what you execute" principle [Ber89]. The result of our work is a practical environment where users can edit, run and debug programs and trigger transformations by clicking the mouse. The same environment also provides support for proof checked by a machine. We used Centaur and Coq to realize the experiment described in this paper.

in: *Conference on Theory and Practice of Software Development (TAPSOFT)*, Aarhus, Danemark, May, 1995.

2 Related work

A wide variety of programming environment generators use formal descriptions of programming languages. The best known systems use Attribute Grammars, Conditional Rewriting Systems or Natural Semantics. Only in rare cases does the specification formalism adapt to some kind of formal proof.

Attribute Grammars, used in the Synthesizer Generator, exemplify the case of specification formalisms where the formal convenience is sacrificed to efficiency. Attribute Grammars can be compiled into very efficient programs, with good incrementality properties (incrementality is the holy grail of interactive environments). However, Attribute Grammars are limited in scope, so that language designers are left on their own to describe key aspects of their language's environment like dynamic semantics. When it comes to formal reasoning, the situation is even worse, as to our knowledge the only work done in this direction is by Reetz and Kropf [RK94] but their approach is very limited in scope.

Conditional Rewriting Systems, used in ASF+SDF, may be more adapted to formal reasoning than attribute grammars, but, to our knowledge there are no experiments done on coupling ASF+SDF with proof tools.

The specification formalism we use in Centaur is often referred to as Natural Semantics [Kah87]. It draws upon earlier work of Plotkin who introduced Structured Operational Semantics [Plo81]. This style of specification is well suited to execution, as it can easily be related with Prolog, Attribute Grammars or functional evaluation. However, this style of specification is also suitable for formal reasoning, as it has its foundations in meta-mathematics and logics. The natural semantics is widely used for the formal description of type systems or program execution.

Another area of related work is formal semantics and proof tools. The work closest to the experience described in this document is that of Pfenning and Rohwedder [PR92]. They use the LF Logical Framework [HHP91] to specify the meta-theory of deductive systems and the meta-language Elf to implement it. Our study improves on their results in two respects: first, we present an integrated environment, where exactly the *same* specification is executed to perform transformations and to *mechanically* produce the necessary theorem prover input; second, Coq provides general tools related to inductive definitions as in [PM93], while in LF the induction principle is not internalized in the system. A drawback of our approach with respect to theirs is that we are not able to use higher order abstract syntax. Despeyroux and Hirschowitz are currently doing research in this direction [DH94]. However, they prove properties of functional languages, and it is unclear whether higher order abstract syntax can also be useful in reasoning about imperative languages.

There are also case studies using HOL for reasoning about semantic properties of programming languages. Very close to our work is [RN91], where Roxas and Newey study simple program transformations on roughly the same programming language. However, they do not have a complete environment and they do not use induction (as they prove only the correctness of local transformation rules).

Camilleri and Zammit [CZ94] study a way to execute formal specifications inside the theorem prover HOL, and to use this symbolic execution as a proof tool. This kind of tool could have been useful in our experiment.

Recently, Buth presented a transformation of operational and denotational semantics definitions into rewrite rules [But94]. These rules can be used as proof input by the Larch Prover [GJ93]. As the transformation is automatic this approach is similar to ours. However Buth doesn't provide a tool for executing the semantic specifications.

3 Starting little

In this section, we present the formal specification of a small imperative programming language, little, that only handles boolean and integer values.

3.1 Syntactic specifications

The abstract syntax describes how to construct syntactic trees for the programs of a language, while the concrete syntax describes their textual form. The abstract syntax consists of *sorts* and *operators*. Operators represent primitive tree patterns, while sorts represent tree categories, for instance instructions or expressions. Each sort is defined as the set of head operators accepted for trees in this sort. The syntactic description of the main part of the language is as follows, where operators are given in lower case characters and sorts are in upper case.

```
program  -> DECLS INST ;    PROGRAM  ::= program ;
decls  -> DECL + ... ;      DECLS  ::= decls ;
decl  -> ID VAL ;           DECL  ::= decl ;
assign  -> ID EXP ;         INST  ::= assign sequence if while ;
sequence  -> INST * ... ;   SEQUENCE  ::= sequence ;
if  -> EXP INST INST ;
while  -> EXP INST ;
```

This description states that a program is made of a list of declarations and an instruction. Each declaration assigns an initial value to some variable of the program. An instruction can be an assignment, an if conditional instruction, a while loop or a sequence of instructions. In the following we will often use skip as a shorthand for sequence[].

The abstract syntax description of a language can be used to implement a structure editor for this language. This tool provides a more user-friendly environment for editing programs in this language. We can also give specifications for other tools of the environment, like a pretty-printer to a more readable textual form and a parser that allows editing fragments of programs as text.

3.2 Semantic Specifications

We describe semantic aspects of the programming language in a natural semantics style [Kah87]. More exactly we define relations between abstract syntax trees and provide inference rules for proving instances of these relations. These inference rules have the following form:

$$\frac{premise_1 \quad \cdots \quad premise_n}{conclusion}$$

The meaning of such a rule is that the conclusion holds if all the premises hold. There is an implicit universal quantification for all the variables occurring in the rule.

Dynamic Semantics We can use this kind of inference rules to specify the *dynamic semantics* of programming languages. We do this with a judgement of the form

$$dynamics(\vdash P \to D)$$

that states that the execution of program P terminates and returns a list of declarations D, a judgement of the form

$$exec(D \vdash I \to D')$$

that states that the execution of an instruction I in the environment D terminates and returns a list of declarations D', a judgement of the form

$$eval(D \vdash E \mapsto V)$$

that states that the evaluation of expression E in the environment D returns the value V, and a judgement of the form

$$update(id, V \vdash D \to D_1)$$

stating that we obtain the new environment D_1 from the environment D by associating the value V to the identifier id.

For instance, the rule describing the execution of a program has the following form:

$$\frac{exec(D \vdash I \to D')}{dynamics(\vdash \mathsf{program}(D, I) \to D')} \tag{1}$$

This rule says that if the execution of I terminates in the environment D and returns the environment D' then the execution of $\mathsf{program}(D, I)$ terminates and returns the same environment. The rule describing the execution of an assignment has the following form:

$$\frac{eval(D \vdash E \mapsto V) \qquad update(id, V \vdash D \to D_1)}{exec(D \vdash \mathsf{assign}(id, E) \to D_1)} \tag{2}$$

It states that to execute an assignment we evaluate the expression E and associate the obtained value V to the identifier id in the new environment.

The rule describing the execution of a sequence of instructions has the following form:

$$\frac{exec(D \vdash I_1 \to D_1) \qquad exec(D_1 \vdash I_2 \to D_2)}{exec(D \vdash \mathsf{sequence}[I_1 . I_2] \to D_2)} \tag{3}$$

It states that the execution of the two instructions has to terminate for the execution of a sequence of two instructions to terminate, and it gives the relation between the various computed environments.

Compiling the dynamic semantics specification into a Prolog program yields an interpreter that can be used to run programs [Des84]. If the programming environment generator also provides subject tracking and breakpointing facilities, a complete debugger can also be derived from this specification [Ber91].

Static Semantics In the same way that we specified the execution of programs, we can express the property that programs respect a type discipline. We will consider a program to be *well typed* if every variable used in the program is declared in the initial list of declarations and if all subsequent uses of this variable are coherent with its declaration. The specification of the static semantics can be described with Natural Semantics rules and compiled into a type-checker for the programs of the language. In the case of static semantics, the target language of the compilation may also be Attribute Grammars [Att88], so that an incremental evaluation of static properties may be achieved.

4 Program transformations

We distinguish two important classes of transformations. Global transformations require a full pass on the program; local ones replace a piece of code with another. In this section we will study a global transformation, constant propagation, and a local one, code simplification.

4.1 Constant propagation

Constant propagation builds on a data flow analysis technique used by compilers in the optimization phase. Its goal is to discover variables whose value is constant on all possible executions and propagate these values as far as possible in the program. For example, this transformation can perform relevant simplifications on a program fragment that has the following form:

```
a := true;
if a then
 begin  x := 2; y := z + (x + 4); z := x + y; end
else
 x := 0
```

and replace it with the following equivalent fragment, inferring the way the conditional instruction and the various assignments will be performed.

```
y:= z + 6; z:= 2 + y; a:= true; x:= 2;
```

As before, we specify constant propagation by describing a number of judgements on programs, instructions, expressions, etc. The idea is to compute in each point of the program a list of bindings associating some variables with their values at that point whenever these values are the same for every possible execution of the program. We use meta-variables named B, B' to denote such lists.

We have a judgement

$$propagate\_program(P \to P')$$

that states that the program P' is the result of constant propagation applied to the program P.

We also have a judgement

$$propagation(B \vdash I \to I' + B')$$

that states that the instruction I' is obtained from the instruction I by simplifying it with respect to the bindings given by B and that B' denotes new bindings derived from B and I. Note that $+$ in judgements is merely a syntactic separator symbol.

The rule used to propagate constants in a program is:

$$\frac{propagation(\mathsf{bindings}[] \vdash I \to I' + B') \quad conversion(B' \mapsto S')}{propagate\_program(\mathsf{program}(D, I) \to \mathsf{program}(D, \mathsf{sequence}[I'.S']))} \tag{4}$$

It says that if the body I of the initial program is simplified with respect to the empty list of bindings, giving us the instruction I' and the new list of bindings B', then the new program will have the body $I'; S'$ where S' is the sequence of assignments $(I_1 := V_1); \ldots; (I_n := V_n)$ corresponding to the list of bindings $B' = (I_1, V_1); \ldots; (I_n, V_n)$. We have a judgement

$$propag\_eval(B \vdash E \mapsto E')$$

that states that the expression E' is obtained from the expression E by simplifying it with respect to a list of bindings of values to variables given by B.

We have a judgement

$$partial\_update(id \backslash val \vdash B \to B')$$

that states that B' is the same list of bindings as B except that it maps the identifier id to the value val.

We also have a judgement

$$remove(id \vdash B \to B')$$

that states that B' is the same list of bindings as B except that it does not associate any value to the identifier id.

For instance, the rules describing constant propagation over assignments have the following form:

$$\frac{propag\_eval(B \vdash E \mapsto val) \quad partial\_update(id \backslash val \vdash B \to B')}{propagation(B \vdash \mathsf{assign}(id, E) \to \mathsf{skip} + B')} \tag{5}$$

$$\frac{propag\_eval(B \vdash E \mapsto E') \quad remove(id \vdash B \to B')}{propagation(B \vdash \mathsf{assign}(id, E) \to \mathsf{assign}(id, E') + B')} \tag{6}$$
$$\text{provided } E' \text{ is not a value}$$

The first rule states that if the expression E simplifies to a value val in the context of the list of bindings B and if updating the list B with the couple (id, val) yields the new list B', then the assignment $\mathsf{assign}(id, E)$ reduces to an empty instruction together with the list B'. This rule is valid only when E simplifies to a fixed value, since partial_update is only defined for such immediate values. The second rule states that if the expression E simplifies to another expression E', which is not a fixed value, then the assignment $\mathsf{assign}(id, E)$ must be transformed into the assignment $\mathsf{assign}(id, E')$ together with a new list of bindings B' where the identifier id is not assigned a value.

The rule describing constant propagation over a sequence of instructions has the following form:

$$\frac{propagation(B \vdash I_1 \to I_1' + B_1) \quad propagation(B_1 \vdash I_2 \to I_2' + B_2)}{propagation(B \vdash \mathsf{sequence}[I_1.I_2] \to \mathsf{sequence}[I_1'.I_2'] + B_2)} \tag{7}$$

It states that the first instruction is simplified with respect to the initial list of bindings and the second instruction is simplified with respect to the list of bindings resulting from the propagation on the first instruction. This transformation specification can also be compiled to Prolog or Attribute Grammars to obtain an executable tool.

4.2 Code simplification

After constant propagation, programs present some unpleasant characteristics. They contain pieces of useless code like if(E,skip,skip). We use code simplification to get rid of such useless fragments. The specification of this local transformation is divided into two parts. First, we define a judgement

$$replace(I \to I')$$

expressing that the instruction I is locally replaced with the instruction I'. The following two rules describe how the local replacements are performed:

$$replace(\mathsf{if}(E, \mathsf{skip}, \mathsf{skip}) \to \mathsf{skip}) \tag{8}$$

var $S1$: SEQUENCE

$$\frac{append(S_1 + S_2 = S)}{replace(\mathsf{sequence}[S_1, S_2] \to S)} \tag{9}$$

The first one detects pieces of unused code and suppresses them. The second one restructures some parts of the code into a simpler form. More exactly it flattens a sequence of sequences into one sequence. The variable declaration is used to check that the variable S_1 is already a sequence.

Then we introduce the judgement

$$rewrite(I \rightarrow I')$$

as the closure of the relation *replace* under the operators if,while and sequence. Two of the rules defining this judgement are:

$$\frac{replace(I \rightarrow I')}{rewrite(I \rightarrow I')} \tag{10}$$

$$\frac{rewrite(I \rightarrow I')}{rewrite(\mathsf{while}(E, I) \rightarrow \mathsf{while}(E, I'))} \tag{11}$$

Finally, code simplification is the transitive closure of the *rewrite* relation.

In an interactive environment, local transformations may be applied on request using the mouse, by selecting the location in the program and the rule to apply in a "Transformation menu". Such transformation menus are easy to define in programming environment generators such as the Cornell Synthesizer Generator or Centaur.

5 Using the Coq proof assistant

Formal specifications of programming languages are objects that one should be able to reason about. The reasoning process can be done informally, but it can also be done formally, in a way that can be mechanically checked, by using a proof assistant like Coq. This proof assistant is a system in itself, *a priori* independent from our programming environment generator, but tools have been developed to increase the cooperation between the two systems. So far, these tools provide the following features:

- Abstract syntax specification and semantic specification can be compiled into data-type declarations and axioms for the Coq system [Ter94],
- Programming tools as found in our programming environment generator can improve the usability of the Coq system and make some proofs easier [BKT92].

The type theory provided by the Coq system seems to be a good candidate for representing programming languages, but the real gain of using this system is in the specialized proof tactics provided for manipulating inductive types.

5.1 Translating Specifications

Translating the Abstract Syntax We illustrate the translation of abstract syntax specifications towards Coq data structures on the example of the language little. All the sorts of the language are merged into one inductive type called little and each operator is translated into a constructor of this type: [1]

[1] Another possibility would be to associate a type to each sort. This approach, requires mutually inductive types in the general case. This feature was not available in Coq at the time of the experiment.

```
Inductive Definition little : Set =
    program : little → little → little
  | assign : little → little → little
  | if : little → little → little → little
  | sequence : little → little → little
  | nul_sequence : little
  | while : little → little → little
  | ...
```

In this definition $A \rightarrow B \rightarrow C$ is read as $A \rightarrow (B \rightarrow C)$. List operators are encoded using two operators, a binary one (sequence) and an atomic one (nul_sequence).

To forbid the manipulation of exotic terms (not respecting the syntactic constraints) we define the set of sorts of our language, such as little_PROGRAM and little_EXP, and we define a function little_is which takes two arguments, a term t and a sort p and determines whether t is a syntactically correct tree in the sort p.

Translating Semantic Specifications Semantic specifications are translated into collections of axioms. Each inference rule is compiled into a universally quantified formula. For example, the rule describing the execution of an assignment is translated into the following statement:

```
exec_assign : ∀D, E, id, V : little.(eval D E V) ⇒ (update id V D D1) ⇒
                (exec D (assign id E) D1)
```

The statement $\forall x : A.B$ reads as "for all x of type A, B is true". Also, $A \Rightarrow B \Rightarrow C$ is understood as $A \Rightarrow (B \Rightarrow C)$. Coq also supports inductive definitions of relations. Given a collection of statements about a relation, an inductive definition expresses that the defined relation is the *least* relation, in terms of set inclusion, that verifies all these statements. Actually, semantic specifications correspond to inductive definitions. For instance, in the case of our exec property, we will write down the following inductive definition:

```
Inductive Definition exec : little → little → little → Prop =
      exec_assign : ∀D, E, id, V : little.(eval D E V) ⇒ (update id V D D1) ⇒

                  (exec D (assign id E) D1)
  | exec_sequence : ∀I1, I2, D, D1, D2 : little.
                    (little_is I2 little_SEQUENCE) ⇒ (exec D I1 D1) ⇒
                    (exec D1 I2 D2) ⇒ (exec D (sequence I1 I2) D2)
  | ...
```

5.2 Proof Methods

The Coq system provides a variety of methods for manipulating inductive definitions. Proof of properties of programs often rely on proofs by induction of various forms [Des86,Win93]. More precisely:

- From the definition of an inductive set the Coq system automatically generates a *structural induction principle*. This principle states that this set is the least one closed under the given operators.
- From the definition of an inductive relation the Coq system automatically generates a *principle of induction on the structure of the proof*. This principle states that this relation is the least one closed under the given axioms and inference rules.

Sometimes a degenerate form of induction is sufficient. For example an argument by cases on the structure of expressions will do when a property is true of all expressions simply by virtue of the different forms expressions can take, without having to use the fact that the property holds for subexpressions.

Thanks to the work of Terrasse [Ter94], the relevant inductive definitions are automatically generated from the specification of the language. In turn, the induction theorems are automatically generated by the Coq system. A precise definition of the translation to Coq and a proof of its correctness is given in [Ter94]. This proof guarantees that correctness proofs in Coq imply correctness of the underlying natural semantics style specifications.

6 Proving transformations correct

Once we have translated the syntactic and semantic specifications into Coq definitions, we can start proving that the given program transformations preserve the meaning of programs.

6.1 Validating the constant propagation

We saw that the execution of a program returns a list of pairs *(variable,value)*. To prove the transformation correct it is enough to show that the execution of the transformed program returns a list that associates the same values to the variables. In fact it is easier to prove a stronger result, namely that the transformed program returns the same list as the initial program.

More exactly, we prove the *soundness* and the *completeness* of the transformation. The soundness property states that there is no result that the transformed program returns but that the initial program cannot return.

$$\begin{aligned}&\forall \mathsf{P}_1, \mathsf{P}_2, \mathsf{D}' : \mathsf{little}.(\mathsf{propagate\_program}\ \mathsf{P}_1\ \mathsf{P}_2) \Rightarrow\\ &\qquad\qquad(\mathsf{dynamics}\ \mathsf{P}_2\ \mathsf{D}') \Rightarrow (\mathsf{dynamics}\ \mathsf{P}_1\ \mathsf{D}')\end{aligned}$$

The completeness property states that any result returned by the initial program can also be returned by the transformed program.

$$\begin{aligned}&\forall \mathsf{P}_1, \mathsf{P}_2, \mathsf{D}' : \mathsf{little}.(\mathsf{propagate\_program}\ \mathsf{P}_1\ \mathsf{P}_2) \Rightarrow\\ &\qquad\qquad(\mathsf{dynamics}\ \mathsf{P}_1\ \mathsf{D}') \Rightarrow (\mathsf{dynamics}\ \mathsf{P}_2\ \mathsf{D}')\end{aligned}$$

As the proofs of the two properties are very similar we shall limit our presentation only to the proof of the soundness property. Using the rules (1), (4) and (3) the soundness property can be reduced to the following one:

$\forall \mathsf{I}, \mathsf{I}', \mathsf{B}', \mathsf{S}', \mathsf{D}, \mathsf{D}_1, \mathsf{D}' : \mathsf{little}.$
$\quad (\mathsf{propagation\ nul\_bindings\ I\ I'\ B'}) \Rightarrow (\mathsf{conversion\ B'\ S'}) \Rightarrow$
$\quad (\mathsf{exec\ D\ I'\ D_1}) \Rightarrow (\mathsf{exec\ D_1\ S'\ D'}) \Rightarrow (\mathsf{exec\ D\ I\ D'})$

This property is a particular case of a more general one, propagation_sound, which has the following statement:

$\forall \mathsf{I}, \mathsf{I}', \mathsf{B}, \mathsf{B}', \mathsf{S}, \mathsf{S}', \mathsf{D}, \mathsf{D}_1, \mathsf{D}' : \mathsf{little}.$
$\quad (\mathsf{propagation\ B\ I\ I'\ B'}) \Rightarrow (\mathsf{wf\_bindings\ B}) \Rightarrow (\mathsf{conversion\ B\ S}) \Rightarrow$
$\quad (\mathsf{conversion\ B'\ S'}) \Rightarrow (\mathsf{exec\ D\ I'\ D_1}) \Rightarrow (\mathsf{exec\ D_1\ S'\ D'}) \Rightarrow$
$\quad \exists \mathsf{D}_2 : \mathsf{little}.(\mathsf{exec\ D\ S\ D_2}) \wedge (\mathsf{exec\ D_2\ I\ D'})$

What matters here is that we replace nul_bindings by a more general value B, with a specific constraint, (wf_bindings B), expressing that the list B contains at most one occurrence of each variable. This kind of generalization step seems difficult to automatize and justifies the use of an interactive proof assistant.

The property propagation_sound can be better understood as expressing the commutativity of the following diagram (where the dashed lines represent properties that must be proved):

$$\begin{tikzcd}
\mathsf{D} \arrow[r, "\mathsf{I}'"] \arrow[d, dashed, "\mathsf{B} \rightsquigarrow \mathsf{S}"'] & \mathsf{D}_1 \arrow[d, "\mathsf{B}' \rightsquigarrow \mathsf{S}'"'] \\
\mathsf{D}_2 \arrow[r, dashed, "\mathsf{I}"] & \mathsf{D}'
\end{tikzcd}$$

For this diagram we have the following assumptions:

- the instruction I' is obtained from the instruction I by simplifying it with respect to the bindings given by B, and B' is a new list of bindings derived from B and I,
- S and S' are sequences of assignments corresponding to the lists of bindings B and B', and
- B is well-formed.

We prove the property propagation_sound by induction on the structure of the proof of (propagation B I I' B'), followed by case reasoning on the proof of (exec D I' D_1). We obtain several cases, one for each inference rule given in the definition of the predicate propagation.

We present the proof of one of these cases. It is the case corresponding to the rule (7) when $\mathsf{I} \equiv \mathsf{sequence}(\mathsf{I}_1, \mathsf{I}_2)$ and $\mathsf{I}' \equiv \mathsf{sequence}(\mathsf{I}'_1, \mathsf{I}'_2)$. In this case, we must prove the commutativity of the following diagram:

$$\begin{tikzcd}
\mathsf{D} \arrow[r, "\mathsf{I}'_1"] \arrow[d, dashed, "B \rightsquigarrow S"'] & \mathsf{D}'_1 \arrow[r, "\mathsf{I}'_2"] \arrow[d, dashed, "B_1 \rightsquigarrow S_1"'] & \mathsf{D}_1 \arrow[d, "B_2 \rightsquigarrow S_2"'] \\
\mathsf{D}_2 \arrow[r, dashed, "\mathsf{I}_1"] & \mathsf{D0} \arrow[r, dashed, "\mathsf{I}_2"] & \mathsf{D}'
\end{tikzcd}$$

We begin by applying the induction assumption corresponding to the proof of the hypothesis (propagation $\mathsf{B_1}$ $\mathsf{I_2}$ $\mathsf{B_2}$ $\mathsf{I'_2}$), where $\mathsf{B_1}$ is the list of bindings obtained by applying the propagation on $\mathsf{I_1}$. We have to show that the list $\mathsf{B_1}$ is well-formed: (wf_bindings $\mathsf{B_1}$). We do this by using the assumption that the initial list B is well-formed and applying an auxiliary lemma, propagate_well_formed saying that the propagation transforms a well-formed list into another well-formed list:

$$\forall \mathsf{B}, \mathsf{I}, \mathsf{B'}, \mathsf{I'} : \mathsf{little}.(\mathsf{propagation\ B\ I\ I'\ B'}) \Rightarrow (\mathsf{wf\_bindings\ B}) \Rightarrow (\mathsf{wf\_bindings\ B'})$$

Interestingly, propagation_sound expresses that the propagation preserves dynamic properties of instructions, while propagate_well_formed expresses that the propagation also preserves static properties, like well-formedness. After this first step the intermediary state can be represented by the following partially completed diagram:

$$\begin{tikzcd}
\mathsf{D} \arrow[r, "\mathsf{I'_1}"] \arrow[d, dashed, "B \rightsquigarrow S"'] & \mathsf{D'_1} \arrow[r, "\mathsf{I'_2}"] \arrow[d] \arrow[dl, phantom, "B_1 \rightsquigarrow S_1"] & \mathsf{D_1} \arrow[d] \arrow[dl, phantom, "B_2 \rightsquigarrow S_2"] \\
\mathsf{D_2} \arrow[r, dashed, "\mathsf{I_1}"] & \mathsf{D0} \arrow[r, "\mathsf{I_2}"] & \mathsf{D'}
\end{tikzcd}$$

The proof of the case can be completed by using the induction assumption corresponding to the proof of (propagation B I1 $\mathsf{B_1}$ $\mathsf{I'_1}$).

6.2 Validating the code simplification

As before we want to prove the soundness and the completeness of the transformation. The proof closely follows the specification of the transformation, being also divided into two parts.

Let us first define the relation equiv on the instructions of the language as the equivalence induced by the dynamic semantics:

$$\forall \mathsf{I}, \mathsf{I'} : \mathsf{little}.(\mathsf{equiv\ I\ I'}) \stackrel{\mathrm{def}}{=} \forall \mathsf{D}, \mathsf{D_1} : \mathsf{little}.(\mathsf{exec\ D\ I\ D_1}) \Leftrightarrow (\mathsf{exec\ D\ I'\ D_1})$$

In the first part, we prove that each local replacement transforms an instruction into an equivalent one:

$$\forall \mathsf{I}, \mathsf{I'} : \mathsf{little}.(\mathsf{replace\ I\ I'}) \Rightarrow (\mathsf{equiv\ I\ I'})$$

In the second part we prove that the relation rewrite transforms an instruction into an equivalent one:

$$\forall \mathsf{I}, \mathsf{I'} : \mathsf{little}.(\mathsf{rewrite\ I\ I'}) \Rightarrow (\mathsf{equiv\ I\ I'})$$

This proof is done by induction on the structure of the proof of the judgement (rewrite I I'). Actually it reduces to a proof of the fact that the relation equiv is a congruence with respect to the operators if, while and sequence. For instance, one of the subgoals to prove is:

$$\forall \mathsf{I}, \mathsf{I}', \mathsf{E} : \mathsf{little}.(\mathsf{equiv}\ \mathsf{I}\ \mathsf{I}') \Rightarrow (\mathsf{equiv}\ (\mathsf{while}\ \mathsf{E}\ \mathsf{I})\ (\mathsf{while}\ \mathsf{E}\ \mathsf{I}'))$$

We should stress that the second part of the proof is *generic*, as it can be shared by all the proofs of local transformations.

Another interesting remark about this proof is that it must be done under the assumption that the manipulated programs are well typed. We need this assumption in order to prove that the instructions skip and if E then skip else skip are equivalent. But skip can be executed in any environment whereas if E then skip else skip can only be executed in an environment where E is a boolean expression. This shows the need to prove several facts which relate the static and the dynamic semantics of the language, such as:

- the evaluation of a well typed expression always terminates and returns a value having the same type as the expression.
- types of variables do not change during the execution of a well typed instruction.

7 Conclusions and future work

The work described in this paper aims at providing a uniform environment for the design and study of well defined programming languages. The originality of this work is not in the way formal specifications are executed and integrated in an interactive environment or in the proof techniques that have been used to establish the coherence results between the various specifications. Rather, this work is original as it connects several aspects of formal descriptions of programming languages which have so far been politely ignoring each other. Connecting these various research domains is not only interesting *per se*, it is also relevant when considering the evolution of software engineering tools. First, the wide variety of applications of computer technology leads to the design of numerous special purpose languages for which trustworthy compilers and programming tools are needed. It is sensible to provide tools to the designers of these languages to assist the task of writing these compilers and verifying their correctness. Second, transformation tools can become a major feature of programming environments, since such tools can assist software engineers not only in their task of writing new programs and optimizing them, but also in the task of maintening old software and adapting it to new architectures. The transformations we have studied in our work exemplify two kinds of transformation tools: one performs a complete pass on the program, without interaction from the user, while the other describes a transformation that can be piloted by the engineer in an interactive environment.

This work was very labor intensive, especially due to the lack of programmability of the proof system we used (an old version of Coq). The issue whether this technique will scale up to more powerful languages cannot be fairly estimated before evaluating the progress obtained with a programmable proof system. In this respect we have remarked that many proofs followed a similar pattern which could make them amenable to an automatic treatment. For scaling up to real programming languages it is also important to be able to reuse previous semantic descriptions and proofs as one adds a new feature to the language. We suspect that the work of Felty and Howe [FH94] brings pertinent answers to this problem.

All these problems hide a more general one: compiling semantic specifications to definitions for the proof assistant and then using the proof assistant for manipulating these definitions forces the user down to the lowest level of abstraction. A more user-friendly environment would allow the user to reason directly with the concepts available at the level of the formal specification, or even with concepts available at the level of the programming language. Finding the correct representation for concepts during the proof is an interesting but difficult goal.

References

[Att88] I. Attali. Compiling Typol with Attribute Grammars. In *Programming Language Implementation and Logic Programming*, Orléans, France, 1988. Springer Verlag, LNCS.

[Ber89] G. Berry. Real-time programming: General purpose or special-purpose languages. In G. Ritter, editor, *Information Processing 89*, pages 11–17. Elsevier Science Publishers P.V, 1989.

[Ber91] Y. Bertot. *Une Automatisation du Calcul des Résidus en Sémantique Naturelle.* PhD thesis, Université de Nice-Sophia Antipolis, 1991.

[BKT92] Y. Bertot, G. Kahn, and L. Théry. Real Theorem Provers Deserve Real Interfaces. In *5th ACM Symposium on Software Development Environments*, Washington, 1992. Also available as INRIA Research Report, RR-1684.

[But94] K. Buth. *Techniques for Modelling Structured Operational and Denotational Semantics Definitions with Term Rewriting Systems.* PhD thesis, Christian-Albrechts University, Kiel, 1994.

[CZ94] J. Camilleri and V. Zammit. Symbolic Animation as a Proof Tool. In *HOL Theorem Proving System and its Applications.* Springer-Verlag LNCS 859, 1994.

[Des84] T. Despeyroux. Executable Specifications of Static Semantics. In *International Symposium on Semantics of Data Types*, 1984. Springer-Verlag LNCS 173.

[Des86] J. Despeyroux. Proof of Translation in Natural Semantics. In *Proceedings of the first ACM-IEEE Symp. on Logic In Computer Science, Cambridge, Ma, USA, June 1986*, pages 193–205, 1986. also available as a Research Report RR-514, Inria-Sophia-Antipolis, France, April 1986.

[DFH+93] G. Dowek, A. Felty, H. Herbelin, G. Huet, Ch. Paulin, and B. Werner. The Coq Proof Assistant User's guide, Version 5.8. Technical Report 154, INRIA, Rocquencourt, May 1993.

[DH94] J. Despeyroux and A. Hirschowitz. Higher-Order Abstract Syntax and Induction in Coq. In *Proceedings of the* 5^{th} *Int. Conf. on Logic Programming and Automated Reasoning*, July 1994.

[FH94] A. Felty and D. Howe. Generalization and Reuse of Tactic Proofs. In *Proceedings of the 5^{th} Int. Conf. on Logic Programming and Automated Reasoning*, July 1994.

[GJ93] J.V. Guttag and J.J.Horning, editors. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[HHP91] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. Technical Report 162, LFCS, University of Edinburgh, June 1991.

[Jac92] I. Jacobs. The Centaur 1.2 Manual. Technical report, INRIA, Sophia-Antipolis, 1992.

[Kah87] G. Kahn. Natural Semantics. In *Proceedings of the Symp. on Theorical Aspects of Computer Science, Passau, Germany*, 1987. Also available as Research Report RR-601,INRIA, Sophia-Antipolis, February 1987.

[Kli93] Paul Klint. A Meta-environment for Generating Programming Environments. In *ACM Transaction on Software Engineering and Methodology*, number 2 in 2, pages 176–201, 1993.

[MT92] M.J.C. Gordon and T.Melham. *HOL: a Proof Generating System for Higher-order Logic.* Cambridge University Press, 1992.

[Pfe89] F. Pfenning. Elf: A Language for Logic Definition and Verified Meta-Programming. In *Proceedings of the 4^{th} International Symposium on Logic in Computer Science*, June 1989.

[Plo81] G.D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Aarhus, 1981.

[PM93] C. Paulin-Mohring. Inductive Definitions in the System Coq: Rules and Properties. In Mark Bezem and Jan-Friso Groote, editors, *Typed Lambda Calculi and Applications*, pages 328–345. Springer-Verlag, March 1993.

[PR92] F. Pfenning and E. Rohwedder. Implementing the Meta-Theory of Deductive Systems. In D. Kapur, editor, *Proceedings of the 11^{th} International Conference on Automated Deduction*, Saratoga Springs, New York, June 1992.

[RK94] R. Reetz and T. Kropf. Simplifying Deep Embedding: A Formalised Code Generator. In *HOL Theorem Proving System and its Applications.* Springer-Verlag LNCS 859, 1994.

[RN91] R. Roxas and M. Newey. Proof of Program Transformations. In *HOL'91, HOL Theorem Proving System and its Applications*, pages 223–230. IEEE Computer Society Press, 1991.

[RT88] T. Reps and T. Teitelbaum. *The Synthesizer Generator: a System for Constructing Language Based Editors.* Springer Verlag, 1988. (third edition).

[Ter94] D. Terrasse. Encoding Natural Semantics in Coq. Submitted to AMAST'95. Also available by anonymous ftp to babar.inria.fr: pub/croap/terrasse:NSinCoq.dvi, 1994.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages, an Introduction.* Foundations of Computing. The MIT Press, 1993.