Specification of imperative languages using operational semantics in Coq

Jean-Marie Madiot
internship under the supervision of Freek Wiedijk

2010, June and July
Radboud University of Nijmegen

1 Introduction

In software verification, *formal* verification is increasingly used to provide guarantees for behaviors in various domains. To correctly use formal verification one needs formal specifications describing the environment in order to state mathematical propositions about a system. Such specifications are currently and actively developed for hardware, micro-kernel or programming languages.

Formalizing compilation of programs has interests in several domains and it needs several layers of formalization. One needs a specification of the low-level language – generally assembly – to assume hypotheses on the actual behavior of the compiled programs. Moreover, one also needs a specification of the high-level language (i.e. the source code language) to be able to formulate the intended behavior of the program.

We focus on specifying a low-level language, C, in a way considering stack overflow, undefined behavior and nondeterminism. Indeed these aspects of the C programming language are generally omitted in specifications whereas they allow compilers to perform optimizations. The goal we pursue in the following is to provide a formal specification of the C progamming language faithful to the C standard [ISO99].

The development of the specification will use the proof assistant Coq. Indeed using a computer for verbose specification is a good way to avoid errors by automated checking. We use the flexibility of Coq and also sometimes we use the constructive paradigm of Coq.

1.1 Nondeterminism and undefined behavior

Undefined behavior and nondeterminism were designed to provide the compiler designers with some flexibility in the implementation and mostly to allow them to optimize the compiled programs. For instance, in the functional programming language Objective CAML the order of evaluation of the arguments of a function is not specified and it can be used by the compiler to use less memory. Undefined behavior lets the compiler implement some optimization without considering all possible cases with could lead to a slower produced program.

Nondeterminism can be justified in some situations as followed:

```
int main ()
{
  f() + g();
}
```

In the previous source code, a function call to f can use less stack or memory than to g. Depending on this, the result of this evaluation has to be stored and the remaining space for the computation of $g()$ would be smaller, so the compiler can choose the best way to order these operations. This kind of phenomenon especially appears in recursive calls.

However if nondeterminism appears harmless – as it is just about the order of evaluation – there are situations where the difference has more significant consequences:

```
int main ()
{
  int a, b;
  b = (a = 0) + (a = 1);
  return a;
}
```

In the previous program, the value of b at the end of the function is always 1 but the function can return either 0 or 1, depending on the evaluation order. In fact this is so problematic that the C standard specifies the behavior this program is *undefined* in the sense it can really do anything. The C standard defined several kinds of special behaviors:

- implementation-defined behavior: the implementation of the compiler chooses a behavior in a set of authorized behaviors
- unspecified behavior: at runtime the program has to respect a set of possible behaviors
- undefined behavior: the program can do anything, for example crashing

Undefined behavior cannot be empirically verified as it can be not reproducible and to consider these aspects of the C standard it seems better to have a full formal specification of the system. Indeed, formal verification can be avoided when one can enumerate all the possible ways a program can behave, and undefined behavior and nondeterminism seem to prevent from being able to test all the alternatives.

One can wonder if static and syntactic analysis can avoid heavy specification by simply considering variable names in the source code. Of course this is impractical in general as C allows pointers manipulation. In the expression `(*a = *b ++)` if `a` and `b` have the same value, then `*a` and `*b` will be the same variable and the evaluation of the expression is then undefined.

1.2 Call stack handling

The existing formal descriptions of C does not consider the function call stack – and thus stack overflow – whereas most implementations use the call stack to significantly improve performances. Unfortunately the C standard is quite evasive about the stack[1].

For example the following C program calls recursively 2^{25} times the function `f` and each recursive call causes some allocation for the local variables making allocation summing up to about 130 megabytes [2].

```
int f (int c)
{
  return c && f(c-1);
}
int main ()
{
  return f(1 << 25);
}
```

Allocating that on the heap with functions as `malloc` is very possible, even if it can fail. However using this amount of memory on the stack is generally causing a *stack overflow* because the stack is generally of a moderated size. A stack overflow cannot be caught and it causes a segmentation fault, whereas an allocation done with `malloc` won't crash. (It will return `NULL` instead of a correct pointer.)

Formalization of the C specification usually does not consider the stack and the fact that the C99 standard does not talk about it. It could result that we can certify a program (like the one above) that fails with most real-world implementation of the specification (the compilers). Even if certified compilers do their job – which is providing a program that respects a given semantics under some assumptions – we would like to be able to provide a specification considering the call stack in different ways.

1.3 Semantics

The specification of programming language is usually done through a natural language sometimes leading to ambiguities of interpretation, and C is not an exception. A formal specification is usually done using a formal semantics.

A formal semantics of a programming language defines its meaning. This meaning can be expressed in another language (as mathematical functions, or another language – generally lower-level). We would call such a semantics *denotational*.

However, this meaning can also be directly expressed in terms of execution and then we deal with *operational* semantics. There is mainly two kinds of

[1]in fact the C99 standard never mentions the call stack

[2]on usual machines where `sizeof(int)` equals to 4

operational semantics: *structural* operational semantics and *natural* operational semantics.

Structural operational semantics (SOS) (also called *small-step semantics*) explains rather syntactically the steps of execution of a program. Natural operational semantics (also called *big-step semantics*) defines the value which the program will eventually have, if any.

Example: in the figure 1 we present both the big-step semantics and the small-step semantics rules for the evaluation of an expression with + as top-level connective. The representation of the integers are underlined ($\underline{n}$) and $\underline{n_1 + n_2}$ corresponds to the representation of the integer $n_1 + n_2$.

$\dfrac{e_1 \to e_1'}{e_1 + e_2 \to e_1' + e_2}$	$\dfrac{e_2 \to e_2'}{\underline{n_1} + e_2 \to \underline{n_1} + e_2'}$	$\dfrac{}{\underline{n_1} + \underline{n_2} \to \underline{n_1 + n_2}}$	$\dfrac{e_1 \Rightarrow \underline{n_1} \quad e_2 \Rightarrow \underline{n_2}}{e_1 + e_2 \Rightarrow \underline{n_1 + n_2}}$
small-step			big-step

Figure 1: Operational semantics: rules for +

Since in the following we will mostly talk about imperative languages the state of the machine running the program will be the main notion of value. We will note $\langle i, s\rangle \Rightarrow s'$ for the meaning "the program i with the value s evaluates into the value s'" which is rather a natural operational semantics notion. We will also note $\langle i, s\rangle \to \langle i', s'\rangle$ the notion of "the program i associated with the state s transforms itself in one step into the other program i' with the state s' (we now talk about small-step semantics).

The programs are generally composed of both expressions and instructions and we will respectively note the associated transitions $\to_e$ and $\to_i$. s will designate the state of the machine, which can be for example a map between variable names x and values n. We will note as usual $s[n/x]$ the state s in which the value of x is now n. We will also note $\to^*$ the reflexive and transitive closure of $\to$. For example $\langle x \times y + 3, (x \mapsto 1, y \mapsto 2)\rangle \to_e^* \langle 5, (x \mapsto 1, y \mapsto 2)\rangle$.

2 Bottom-up approach

Developing a semantics for a programming language can be some hard work and to avoid as many mistakes as possible we choose to approach the system starting with a simple and well-known one, and then progressively add features to it in order to have a system working throughout the development. We acknowledge [Plo81] for inspiring the first drafts of the semantics.

All steps that follow have been implemented in Coq as different languages with their own semantics.

2.1 The While language

The While language has expressions e and instructions i, defined by the following grammars:

$$e \quad ::= \quad n \mid x \mid e \odot e \mid \Box e \tag{1}$$

$$i \quad ::= \quad \texttt{skip} \mid i \;;\; i \mid \texttt{if } e \texttt{ then } i \texttt{ else } i \mid \texttt{while } e \texttt{ do } i \mid x := e \tag{2}$$

where

- $\odot$ is a binary operator ($\odot$ is one of the $+, *, -$ or any comparison $\leq, \geq, <, >, =, \neq$)
- $\Box$ is a unary operator ($\Box$ can here be only the negation $\neg$)
- n the representation of an integer
- x a variable name

There is no typing system and there is no need for it as expressions are all of the type `int`. This is a small extension of the language defined in [Ber07] which is given a small-step semantics – among others.

Intuitively the semantics of the expressions is as usual and saying that if x is not defined in s, it is in $s[n/x]$. The semantics of While we defined in Coq was quite simple. The memory is simply addressed by the variable names and the evaluation of expressions has no side effects so it can be done independently. Indeed, the first implementation considered the evaluation of expressions as a single "small-step" $\Rightarrow$. The semantics for the instructions is described below.

$$\frac{\langle i_1, s\rangle \to \langle i_1', s'\rangle}{\langle i_1; i_2, s\rangle \to \langle i_1'; i_2, s'\rangle}\ ;_{sub}$$

$$\frac{}{\langle \texttt{skip}\ ; i_2, s\rangle \to \langle i_2, s\rangle}\ ;_{skip}$$

$$\frac{\langle b, s\rangle \Rightarrow n \quad n \neq 0}{\langle \texttt{while } b \texttt{ do } i, s\rangle \to \langle i; \texttt{while } b \texttt{ do } i, s\rangle}\ \texttt{while}_T$$

$$\frac{\langle b, s\rangle \Rightarrow 0}{\langle \texttt{while } b \texttt{ do } i, s\rangle \to \langle \texttt{skip}, s\rangle}\ \texttt{while}_F$$

$$\frac{\langle b, s\rangle \Rightarrow n \quad n \neq 0}{\langle \texttt{if } b \texttt{ then } i_1 \texttt{ else } i_2, s\rangle \to \langle i_1, s\rangle}\ \texttt{if}_T$$

$$\frac{\langle b, s\rangle \Rightarrow 0}{\langle \texttt{if } b \texttt{ then } i_1 \texttt{ else } i_2, s\rangle \to \langle i_2, s\rangle}\ \texttt{if}_F$$

$$\frac{\langle e, s\rangle \Rightarrow n}{\langle x := e, s\rangle \to \langle \texttt{skip}, s[n/x]\rangle}\ :=$$

$\langle \cdot, \cdot\rangle \Rightarrow \cdot$ and $\cdot[\cdot/\cdot]$ are explicitly defined in the Coq source code and this is enough for such a small language.

2.2 Adding side effects

One characteristic of the C programming language is that expression can contain side effects – even most side effects are contained in expressions. We then develop another toy language with this property. The grammar below is almost enough to see the modifications of the language. The affectation is now a part of the expression and not of the instructions.

$$e \quad ::= \quad n \;\mid\; x \;\mid\; x := e \;\mid\; e \odot e \;\mid\; \Box e \tag{3}$$

$$i \quad ::= \quad \texttt{skip} \mid i \;;\; i \mid \texttt{if}\ e\ \texttt{then}\ i\ \texttt{else}\ i \mid \texttt{while}\ e\ \texttt{do}\ i \mid e \tag{4}$$

One big difference with the previous semantics is that we need an explicit handling of the state in the semantics for the expressions, we then note the reduction relations for the expression and the instructions respectively $\rightarrow_e$ and $\rightarrow_i$. We detail the semantics for the expressions.

Nevertheless, in order to take into account the modification of the state by the evaluation of the expressions we have to add rules for constructors which have expressions as parameters. For example we transform the rules $\texttt{if}_T$ and $\texttt{if}_F$ into three rules $\texttt{if}_T$, $\texttt{if}_F$ and $\texttt{if}_{sub}$. The first two transform a statement of the form $\texttt{if}\ \underline{0}\ \texttt{then}\ i_1\ \texttt{else}\ i_2$ in i_2 and $\texttt{if}\ \underline{n}\ \texttt{then}\ i_1\ \texttt{else}\ i_2$ in i_1 if $n \neq 0$.

$$\frac{n \neq 0}{\langle \texttt{if}\ \underline{n}\ \texttt{then}\ i_1\ \texttt{else}\ i_2, s\rangle \rightarrow_i \langle i_1, s'\rangle}\ \texttt{if}_T$$

$$\frac{}{\langle \texttt{if}\ \underline{0}\ \texttt{then}\ i_1\ \texttt{else}\ i_2, s\rangle \rightarrow_i \langle i_2, s'\rangle}\ \texttt{if}_F$$

The last one is detailed below:

$$\frac{\langle b, s\rangle \rightarrow_e \langle b', s'\rangle}{\langle \texttt{if}\ b\ \texttt{then}\ i_1\ \texttt{else}\ i_2, s\rangle \rightarrow_i \langle \texttt{if}\ b'\ \texttt{then}\ i_1\ \texttt{else}\ i_2, s'\rangle}\ \texttt{if}_{sub}$$

Note that we have to handle the `while` differently as if we transform the conditional sub-expression (b) into a value then we will have to retrieve the original expression, so we change the rules for `while` into a single syntactic one:

$$\frac{}{\langle \texttt{while}\ b\ \texttt{do}\ i, s\rangle \rightarrow_i \langle \texttt{if}\ b\ \texttt{then}\ i; \texttt{while}\ b\ \texttt{do}\ i\ \texttt{else}\ \texttt{skip}, s\rangle}\ \texttt{while}$$

As instructions can contain expressions whose result are ignored – like in C where in the term `a=2;` the subterm `a=2` is an expression – we add a rule of inheritance of $\rightarrow_i$ from $\rightarrow_e$ and a rule transforming the instruction ignoring a value into `skip`. Note that the constructor `ignore` is here hidden for convenience.

$$\frac{\langle e, s\rangle \rightarrow_e \langle e', s'\rangle}{\langle e, s\rangle \rightarrow_i \langle e', s'\rangle}\ \texttt{ignore}_{sub}$$

$$\frac{}{\langle \underline{n}, s\rangle \rightarrow_i \langle skip, s\rangle}\ \texttt{ignore}$$

For testing and feasibility purposes we write a little program that computes the GCD of two numbers in this language. Then we prove that this program will terminate on any non-negative arguments. In fact the exact statement is that it can terminate (i.e. end on the `skip` instruction) and does not talk about any other possible sequence – even if the semantics is expected to be deterministic.

```
Lemma gcd_terminating: forall a b, { d |
  ⟨ gcd "a" "b" | ("a", Z_of_nat a) :: ("b", Z_of_nat b) :: nil ⟩ →*
  ⟨ skip        | ("a", Z_of_nat d) :: ("b", 0)          :: nil ⟩}.
```

Note that the statement is constructive: the `{ d | ... }` statement means that there exists some `d` such that "…" but also that Coq is able to produce such a `d`. And indeed we can extract from this proof a ML term that computes the GCD of two natural integers.

2.3 Adding procedures

To add procedures to the language we have to handle the environment differently. As we want to keep the language and the implementation simple we'll make a simple system of scope: a variable is accessible since it has been defined and local variable of procedures are destructed when leaving the procedure, which is necessary to handle recursion.

$$e \quad ::= \quad n \mid x \mid x := e \mid e \odot e \mid \Box e \tag{5}$$
$$i \quad ::= \quad \texttt{skip} \mid i \,;\, i \mid \texttt{if } e \texttt{ then } i \texttt{ else } i \tag{6}$$
$$\mid \texttt{while } e \texttt{ do } i \mid e \mid \texttt{call } x \texttt{ with } (e, \ldots, e) \tag{7}$$

Implementing the procedures in the semantics implies a notion of environment for the procedures themselves, which will likely be the case in the more general case of C. The environment is now a stack which contains the value of all defined variables but this does not fundamentally change the Coq implementation. At each procedure call the special token *PROC* is pushed on the stack, then followed by the arguments.

Leaving a function call will cause all arguments to be popped until a *PROC* is reached. (If no *PROC* is found then the execution ends). We do not detail the rules for this semantics given that adding functions is more relevant in that the rules are more general so that the next section will suffice. Also for this quite simple language we develop a Coq function that performs a small-step reduction – if possible.

2.4 Adding functions: the Return language

We now talk about functions which can return a value. Unlike functional programming language we want that returning a value means to use the keyword `return` at any moment in the function. As we use small-step semantics a call to a function could be transformed into instructions whereas a call to a function is an expression. (as in `x:=gcd(3,4)+3`).

We have one main difference here which is that function calls can be made inside expressions. Then the expressions/instructions grammars have to be mutually recursive (because intermediate states of an expression containing a function call will contain instructions of the function called). But also – and mainly – the expressions/instructions semantics have to be mutually recursive.

For convenience we will call this language the RETURN language. The grammar is the following:

$$\begin{aligned} e \quad &::= \quad n \;\mid\; x \;\mid\; x := e \;\mid\; e \odot e \;\mid\; \Box e && (8)\\ &\quad\;\; \mid\; e \;?\; e : e \;\mid\; id(e,\ldots,e) \;\mid \mathrm{wait}(i) && (9)\\ i \quad &::= \quad \texttt{skip} \mid i \;;\; i \mid \texttt{if}\ e\ \texttt{then}\ i\ \texttt{else}\ i && (10)\\ &\quad\;\; \mid \texttt{while}\ e\ \texttt{do}\ i \mid e \mid \texttt{return}\ e \mid \texttt{exit}\ e && (11) \end{aligned}$$

The main changes are all in the constructor $\cdot(\,\cdot\,,\ldots,\,\cdot\,)$ (the function call) and the `return` instruction (for leaving a function call with a value). We add a constructor wait for embedding instructions in expressions when a function call is encountered, but this constructor does not need to be in the original programming language's syntax.

We also add the ternary operator ? : and the `exit` function to apprehend these new notions since we want to approach the C language. The `exit` function is easily definable given the way we encode function calls, since it just have to skip the *PROC* elements on the stack.

Like the previous language with procedures we have to consider an environment for global variables like the body of the functions. As usual in the operational semantics we note

$$p \vdash \langle\; i \mid s \;\rangle \rightarrow^* \langle\; i' \mid s' \;\rangle$$

for a reduction of where p contains the global information that will not be modified all along the execution. We could just insert this kind of information into the state s but this is another chance to generate mistakes.

We detail some of the 45 inference rules of the semantics we build for the RETURN language below.

Final step of the result of a binary operator:

$$\frac{}{c \;\vdash\; \langle\; \underline{n_1} \;\odot\; \underline{n_2} \;,\; s \;\rangle \;\rightarrow_e\; \langle\; \underline{eval_\odot(n_1, n_2)} \;,\; s \;\rangle}\ \text{bop\_calc}$$

Forcing the evaluation from left to right for binary operators:

$$\frac{c \;\vdash\; \langle\; e_2 \;,\; s \;\rangle \;\rightarrow_e\; \langle\; e_2' \;,\; s' \;\rangle}{c \;\vdash\; \langle\; \underline{n_1} \;\odot\; e_2 \;,\; s \;\rangle \;\rightarrow_e\; \langle\; \underline{n_1} \;\odot\; e_2' \;,\; s' \;\rangle}\ \text{bop\_subr}$$

Below is the case of the function call when all arguments are values. We see here that that find_proc is a Coq function which can return `None` if the function name is not in the global environment c and in case of success return the body of the function and the name of the arguments. In the same way get_num returns `None` if an argument is not a value and the list of the values in the other case. Then mapargs build an environment from the names and the values and the instructions of the body of the function are embedded in the "wait" constructor.

$$\frac{\begin{array}{c}\text{find\_proc } c\ f \ = \ \text{Some } (\vec{x},\ \mathit{body}) \\ \text{get\_num } \mathit{args} \ = \ \text{Some } \vec{n}\end{array}}{c \ \vdash \ \langle \text{ call } f \text{ with } \mathit{args} \ , \ s \ \rangle \ \rightarrow_e \ \langle \text{ wait } \mathit{body} \ , \ \text{mapargs } \vec{x}\ \vec{n} \ + + \ \text{PROC} \ :: \ s \ \rangle}$$

When `return` is reached with a value ($\underline{n}$) the next step is popping every local variable from the stack until reaching $PROC$.

$$\frac{}{p \ \vdash \ \langle \text{ return } \underline{n} \ , \ \text{MEM}(y, vy) \ :: \ s \ \rangle \ \rightarrow_i \ \langle \text{ return } \underline{n} \ , \ s \ \rangle} \text{ return\_pop}$$

And when a $PROC$ is encountered with a value then the value is ready and the "wait" unleash the value contained by the `return`.

$$\frac{}{c \ \vdash \ \langle \text{ wait (return } \underline{n}) \ , \ \text{PROC} \ :: \ s \ \rangle \ \rightarrow_e \ \langle \ \underline{n} \ , \ s \ \rangle} \text{ wait\_return}$$

We also developed a Coq tactic (written in Ltac) which tries to apply the correct rules to prove a goal which top-level connector is $\rightarrow_i, \rightarrow_e, \rightarrow_i^*$ or $\rightarrow_e^*$. The tactic, `sos_small_step`, is written with the underlying objective that its behavior should be practically the same as an interpreter. Indeed the use of this tactic often manifest as visualizing the steps of the computation of a given program.

Measuring the stack is also quite easy the way we built the semantics. Indeed at any moment the size of the stack is the length of the data structure representing the state. Thus to consider stack overflow – although this notion is arbitrary in such a toy language – one just need to add a predicate to consider the maximum length of the state used in a sequence of reduction.

2.5 Adding nondeterminism

We only add the nondeterminism now because, for a deterministic system, finding mistakes and debugging is easier. The proofs of properties specific to the deterministic case obtained on the previous systems made the systems trustworthy and we can then use them as a model. Indeed non determinism is almost straightforward. The rules that are modified are those about binary operators and expressions in arguments of function calls.

For instance the following rule of the Return language:

$$\frac{c \ \vdash \ \langle \ e_2 \ , \ s \ \rangle \ \rightarrow_e \ \langle \ e_2' \ , \ s' \ \rangle}{c \ \vdash \ \langle \ \underline{n_1} \ \odot \ e_2 \ , \ s \ \rangle \ \rightarrow_e \ \langle \ \underline{n_1} \ \odot \ e_2' \ , \ s' \ \rangle} \text{ bop\_subr}$$

become this new rule:

$$\frac{c \vdash \langle\, e_2 \,,\, s \,\rangle \rightarrow_e \langle\, e_2' \,,\, s' \,\rangle}{c \vdash \langle\, e_1 \odot e_2 \,,\, s \,\rangle \rightarrow_e \langle\, e_1 \odot e_2' \,,\, s' \,\rangle}\ \text{bop\_subr}$$

which does not force the left operand to be a value to let the right one to evolve.

For illustrate the nondeterminism of this variant of the RETURN language, we also give an example of a program that can evolves in an infinite number of ways and we proved it can:

```
Lemma non_functionality_infinite :
  let i := "i" in
  let c := "c" in
  let loop :=
    (while
      bop Mul (bop Add (i <- #0) (i <- #1)) !i
    do
      ignore (c <- bop Add !c #1))
  in
  forall n : nat,
  nil ⊢
    ⟨ loop | MEM "i" 0 :: MEM "c" 0 :: nil ⟩ →*
    ⟨ skip | MEM "i" 0 :: MEM "c" (Z_of_nat (S n)) :: nil ⟩
.
```

3 Validation

The choice of the inference rules for the semantics is fundamental in the sense that it has to correspond to what it means to be correct. Indeed formalization, before proving anything, implies defining what the proven theorem will mean.

The idea of the De Bruijn criterion – saying trusting a system should mean trusting a small kernel of this system – also applies for the formalization itself: if all the definitions are small and clear the meaning of the theorems is also clear and trustworthy.

Unfortunately defining an operational semantics can be verbose and errors can be made when writing a semantics. This is why proving likelihood theorems about the system helps finding errors which could be not just syntactic errors or oversights, but also more fundamental misconceptions.

3.1 Functionality

A common expected property is *functionality*, meaning a program can evolve in only one way at each step. This may also be called *determinism*. We may remark that another wanted property is confluence which is weaker but also often more desired, as it ensures that a program each time eventually does the same thing even if it can behave differently before it terminates.

This property can be expressed as: “if the program i in the state s can evolve into a program $i1$ in the state s_1 or into a program i_2 in the state s_2, then $i_1 = i_2$ and $s_1 = s_2$”. The statements of the version for both instructions and expressions in Coq is described below.

```
Lemma sos_instr_functionality : forall p i i1 i2 s s1 s2,
  p ⊢ ⟨ i | s ⟩ → ⟨ i1 | s1 ⟩ ->
  p ⊢ ⟨ i | s ⟩ → ⟨ i2 | s2 ⟩ ->
  i1 = i2 /\ s1 = s2.

Lemma sos_expr_functionality : forall p e e1 e2 s s1 s2,
  p ⊢ ⟨ e | s ⟩ >> ⟨ e1 | s1 ⟩ ->
  p ⊢ ⟨ e | s ⟩ >> ⟨ e2 | s2 ⟩ ->
  e1 = e2 /\ s1 = s2.
```

Where $\rightarrow$ is the reduction relation for the instructions and `>>` is the reduction relation for the expressions. This theorem has been proven for the hence deterministic language RETURN.

The proof of these two properties invokes a more general lemma which prove this property for both expressions and instructions, using an induction on their size. This lemma has a proof which uses many times inversion methods[3] because the proof is mainly about things that “cannot happen”. (And the resulting proof term is quite big)

This proof helped finding errors about a few rules, some about `exit`, `return` and also stack handling. Indeed when writing the rules of the semantics one can make choices that are not errors strictly speaking. For example one can let some operation step possible by several rules at once. It is also possible to make mistakes simply on the variable names for example using s' instead of s in the `while` rule and thus authorizing any modification of the environment. This kind of errors are spotted when attempting to prove the functionality.

The proof functionality could be used to build a proof of certified interpreter. It is tempting to state the excluded middle on the existence of a successor, i.e. either there is a successor of $\langle i|s\rangle$ by $\rightarrow_i$ or there is not. That formulation seems easy and immediate but is not directly constructive. Having a constructive proof of it would be very interesting, as it would provide a certified interpreter for the language, which in case of error, would provide a proof of the impossibility to reduce.

3.2 Decidable equality

The equality of the expression and the instructions is normally decidable for any programming language. This is equivalent to say that given the internal representation of two expressions or instructions, a function definable in Coq can decide whether they are equal or not. Such a property is usually true for terms

[3]The tactic `inversion`

with a finite representation. The statements for this property are presented below.

```
Lemma instr_eq_dec : forall a b : instr, { a = b } + { a <> b }.
Lemma expr_eq_dec :  forall a b : expr,  { a = b } + { a <> b }.
```

This property is not too hard to show[4]. It also forces not to put infinite objects in the syntax of the language, for example the syntax cannot contain binary operators containing the associated function. (Because the equality between two functions of type `nat -> nat -> nat` is not decidable.)

3.3 Example of a certified program

We give an example of a certified *program* in the Return language using return instructions and recursive calls (of the same function). It is a functional version of the calculus of the GCD of two numbers.

In the following we write a program in the Return language that computes the result of $gcd(16, 9)$ thanks to a recursive procedure and then assigns this value to the variable `res`. The source code below is written with Coq notations we add to the semantics.

```
proc "gcd" with ["a", "b"]
begin
  If bop Eq !"a" #0 then
    return !"b"
  else
    If bop Eq !"b" #0 then
      return !"a"
    else
      If bop Gt !"a" !"b" then
        return (call "gcd" with [bop Sub !"a" !"b", !"b"])
      else
        return (call "gcd" with [bop Sub !"b" !"a", !"a"]).
end
main
  ignore "res" <- call "gcd" with [#15, #6]
```

Some explanations:

- `bop` is the constructor for binary operators
- `Sub` is the syntactic binary operator for the subtraction
- `Eq` is the one for the equality test

[4]but of course, the tactic `decide equality` does not work – mainly for two reasons:

- `expr` contains a constructor having `list expr` as a premise
- `expr` and `instr` are mutually recursive

- `#n` stands for $\underline{n}$
- `!"b"` stands for the variable addressed by the string "b"
- `call` is the function call
- `[., .]` is the list of arguments passed to the function
- `ignore` is the instruction constructor ignoring an expression
- `<-` is the assignment of an expression to a variable

Thanks to the tactic we have previously written we can see all the small operational steps the program goes through inside Coq itself, just by applying the tactic. But what is more interesting is that we can state some propositions about the behavior of this program.

For example we have proven that the recursive function calculating the GCD is correct and terminates in the Return language. The Coq statement is detailed below where `gcd_body` designates the body of the function of the program written above, `prod` is the construction conjunction in Coq, `is_gcd` a predicate we define below.

```
Definition divide (a b : Z) : Type := { q | b = a * q }.

Definition is_gcd d a b :=
  prod (divide d a)
    (prod (divide d b)
      (forall x, divide x a -> divide x b -> divide x d)).

Lemma gcd_correct :
  forall a b s, Z0 <= a -> Z0 <= b ->
    { d |
      prod (
        [("gcd", ["a", "b"], gcd_body)]
          ⊢
        ⟨ gcd_body  | MEM "a" a :: MEM "b" b :: PROC :: s ⟩ →*
        ⟨ return #d | MEM "a" a :: MEM "b" b :: PROC :: s ⟩
      )
      (is_gcd d a b) }.
```

In plain English the lemma states that for all non-negative integers a and b, there exists d such that the function will return d and d is the greatest [5] divisor of both numbers.

The process of writing the program and mostly the one of proving properties on it spotted some bad conceptions of some semantics rules. It also strengthened confidence in this semantics as the program did what we intended it did.

After approaching the aimed C language with quite smaller ones we try to build a semantics for a specification of C.

[5] in the sense of the divisibility

4 Towards a C semantics

We try to answer basic but necessary questions about a semantics for the C specification. Obviously a complete specification of the C standard goes beyond the framework of this internship but we will try to answer questions that came up when attempting to develop this specification in Coq. This work acknowledges previous works on the subject as [Nor98] which represents a tremendous advance in this direction.

4.1 Step size

The question of which kind of operational semantics we will use is quite important as it will in either way decides which parts of both the specification and its usage will be hard to handle. The big-step semantics are handy for various reasons when defining a semantics for an imperative language. However for a full description of the C standard specification we choose a small-step semantics for several reasons.

Full nondeterminism: one can think nondeterminism can be simulated in a big-step semantics by giving rules saying that an argument or an operand can be evaluated before or after another, with rules like those following (for the + operator):

$$\frac{\langle e_1, s\rangle \Rightarrow \langle \underline{n_1}, s'\rangle \quad \langle e_2, s'\rangle \Rightarrow \langle \underline{n_2}, s''\rangle}{\langle e_1 + e_2, s\rangle \Rightarrow \langle \underline{n_1 + n_2}, s''\rangle}\ +_{lr}$$

$$\frac{\langle e_2, s\rangle \Rightarrow \langle \underline{n_2}, s'\rangle \quad \langle e_1, s'\rangle \Rightarrow \langle \underline{n_1}, s''\rangle}{\langle e_1 + e_2, s\rangle \Rightarrow \langle \underline{n_1 + n_2}, s''\rangle}\ +_{rl}$$

We recall the small-step version:

$$\frac{\langle e_1, s\rangle \rightarrow \langle e_1', s'\rangle}{\langle e_1 + e_2, s\rangle \rightarrow \langle e_1' + e_2, s'\rangle}\ +_l$$

$$\frac{\langle e_2, s\rangle \rightarrow \langle e_2', s'\rangle}{\langle e_1 + e_2, s\rangle \rightarrow \langle e_1 + e_2', s'\rangle}\ +_r$$

$$\frac{}{\langle \underline{n_1} + \underline{n_2}, s\rangle \rightarrow \langle \underline{n_1 + n_2}, s\rangle}\ \underline{+}$$

These simple rules suffice to bring a difference between big-step and small-step. For example in an expression of the form $(a+b)+c$ the big-step semantics groups the evaluation of the subexpressions whereas in small-step one can transform a into a', then c into c', then b into b' (which is impossible in the previous big-step semantics). As we want to be able to express such subtleties small-step semantics seems to suit our need of expressiveness.

Note that for that kind of problems we can refine the answer saying that we need a small-step semantics only for expressions for now.

Non-termination: having a general specification implies handling correctly non-termination of programs. One has to be able to state that a program will in any case terminate, that it can terminate or that it cannot terminate. Those notions differ only because we talk about non-deterministic operations. A big-step semantics gives a relation between a program (e), a state (s), and a value (v, which generally contains a state). Hence one can only express the "possible termination" ($\exists v\ \langle e, s\rangle \Rightarrow v$) and its contrary ($\forall v\ \langle e, s\rangle \not\Rightarrow v$).

Small-step semantics let us be able to manipulate more precisely $\rightarrow$ as a relation. For example "always terminating" can be translated as "there is no infinite $\rightarrow$-sequence starting from the initial program and state". Observe that we can use more tools from rewriting systems with the small-step's $\rightarrow$ and not with the big-step's $\Rightarrow$. Note that it is possible to use a big-step semantics to describe non-terminating computations as described in [LG09] but it involves coinduction and we prefer keeping simple fundamental concepts of proof theory to define a formal specification.

Special value $\mathcal{U}$: the specification has to consider undefined behavior, which can appear in only some possible evaluations and the C standard says if one path of evaluation leads to undefined behavior then the instruction itself leads to undefined behavior. Using big-step semantics makes necessary to have a special value which we note $\mathcal{U}$ to distinguish it from non-termination.

Interleaving: the nondeterminism in expressions can have other consequences. Expressions can contains function calls and the question if in the expression $f() + g()$, the two function calls can interleave is not clearly defined in the specification. A big-step semantics would be likely not to allow interleaving and a small-step semantics can express both alternatives. As a technical question in the formal specification should not influence this kind of decision we choose to develop a small-step semantics.

Additionally it seems easier in a small-step semantics to visually see the execution steps of a program. Note that if we emphasize the reasons that motivated this choice, it's because a small-step semantics as its flaws, even if they are mostly about how more difficult or verbose it is to develop.

4.2 Context trick

As used in [Nor98] we develop a technique which reduce the number of rules – and thus the number of errors – about the expressions and which can make easier their use. The principle is to specify which operand and which subterms can be altered. For example in the expression `a ? b : c` the `a` is the only part that can evolve on its own whereas in `a + b` both operands can evolve.

To avoid writing a lot of verbose rules which only differ of a few characters only to precise these priorities, we build contexts to precise where the reduction can happen.

We also write a tactic taking a subterm as an argument which transform the main term into a context containing the remaining of the term.

For instance to avoid difficult handling of big expressions, the following goal

```
P ⊢ ⟨ e1, s ⟩ →e ⟨ e1', s' ⟩ ->
P ⊢ ⟨ expr_call f [e2, e1 , e3], s ⟩ →e
    ⟨ expr_call f [e2, e1', e3], s' ⟩.
```

can be proven thanks to the tactics

```
context_extract e1.
context_extract e1'.
```

The expressions then look like `C[[e/-]]` and we just have to apply the context rule for $\rightarrow_e$:

$$\frac{P \ \vdash \ \langle\ e, s\ \rangle \ \rightarrow_e \ \langle\ e', s'\ \rangle}{P \ \vdash \ \langle\ C[e/-], s\ \rangle \ \rightarrow_e \ \langle\ C[e'/-], s'\ \rangle}\ \text{expr\_sub}$$

4.3 Sequence points

Sequence points are a notion related to undefined behavior. It's mainly use to forbid some not well-defined practices of programming. Sequence points are some syntactic transitions as followed '`,`', '`;`','`&&`','||','·()' (before a function call) and '`return`'.

Between two sequence points only one writing of the same memory address can be done and a value can be read only to compute a value on the right-hand side of an assignment operator (for example, not for computing the address where the value will be stored[6]). The compiler ensures all side effects are performed before crossing the sequence point.

The implementation of the notion of sequence point is not complete at all and a previous thorough work on how to model the memory map is necessary. However, as the notion of sequence point may change ore be replaced in a near future, it may be interesting to directly consider the new notion.

4.4 Stack handling

Even if stack handling is not (yet) a part of the standard and may belong to the list of the implementation-defined behavior we want to consider stack handling in the specification even if there could have different ways to behave:

- no stack (allocation with `malloc`)
- predefined stack (trying to stick to `gcc`)
- function of stack

[6]"Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be accessed only to determine the value to be stored." [ISO99]

We chose the last option which will give, given some properties of the function say how much will be allocated in the stack or a range for this values. To specify the behavior of a compiler one will then have to give a function "of stack handling" giving the amount of memory the compiler guarantees not to exceed.

4.5 Differences with a compiler

There is some questions one can ask in relation to the role of a compiler. These questions help understanding which aspect of the stack can be present in a future formal specification.

The role of the specification is only describing the possible computations, not the way the intermediate values are stored for instance. For this very reason we cannot consider registers in the specification, at least not the same way compilers do. A consequence of this is that since the intermediate values can be stored on the stack rather than on the registers, handling registers differently can alter the size of the size.

One possible workaround is considering there is no registers. Indeed if every intermediate value is stored on the stack then it could be simulate best the compilers that archive the values of the registers on the stack before calling any other function.

5 Conclusion

The bottom-up approach were successful in the sens that the Coq implementation is working, we can even use them to visually see the programs and the environment evolving. There are also some results and proofs about it. This approach can be used to build semantics that can be used to simulate with (very useful for error checking).

If they are only used for small languages it is probably a good way to proceed to reach a bigger problem.

Then we started a semantics for the specification for C. Of course the duration of the internship was too short, you can't expect formalizing such a big specification in a language more verbose than English, Coq. However we raised useful questions and we tried to answer them from a practical and technical point of view. We have a working but quite incomplete implementation of these semantics. We also encountered many technical problems with Coq itself.

Even if the objectives could be considered way too optimistic we have to say that the work engaged with this internship is the subject of a PhD thesis proposal[7]. There is hence hope a more complete specification of the C99 standard could be achieved.

The development of proofs certification with this kind of semantics could be long and tedious. Nevertheless once this semantics well-established one could

[7]Formalizing the C99 standard in HOL, Isabelle and Coq, Research proposal, Radboud University Nijmegen

generate proofs and proof obligations from techniques of Hoare logic which are easier to handle.

All the implementation developed during the internship will be available under LGPL to further examination online among other things [Mad10].

References

[Ber07] Yves Bertot. Theorem proving support in programming language semantics. Research Report RR-6242, INRIA, 2007.

[ISO99] ISO. Iso c standard 1999. Technical report, 1999. ISO/IEC 9899:1999 draft.

[LG09] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009.

[Mad10] Jean-Marie Madiot. Notes on a formal specification of the C standard, 2010. `http://perso.ens-lyon.fr/jeanmarie.madiot/cspec/`.

[Nor98] Michael Norrish. C formalised in HOL. Technical report, University of Cambridge, 1998.

[Plo81] G. D. Plotkin. A structural approach to operational semantics, 1981.