Specification of imperative languages using operational semantics in Coq

Internship with Freek Wiedijk

Jean-Marie Madiot

ENSL, RU Nijmegen

ENS Lyon, September 8, 2010

Motivations

Certification of **existing** languages:

- critical systems
- low-level languages
- C

Motivations

Certification of **existing** languages:

- critical systems
- low-level languages
- C

$\rightarrow$ C specification

Motivations

Certification of **existing** languages:

- critical systems
- low-level languages
- C

$\rightarrow$ C specification
$\rightarrow$ specification of the C standard

Motivations

Certification of **existing** languages:

- critical systems
- low-level languages
- C

$\rightarrow$ C specification
$\rightarrow$ specification of the C standard (C99)

Motivations

Certification of **existing** languages:

- critical systems
- low-level languages
- C

$\rightarrow$ C specification
$\rightarrow$ specification of the C standard (C99)
with a formal semantics

Existing semantics

Existing semantics: why not?

- no nondeterminism, no undefined behavior
- certifying programs that crashes with GCC
- certifying programs that could crash according to C99

Existing semantics

Existing semantics: why not?

- no nondeterminism, no undefined behavior
- certifying programs that crashes with GCC
- certifying programs that could crash according to C99

but: PhD thesis of Michael Norrish

Semantics

Reminder:

- Denotational:

$$\left[\!\left[\text{let } \begin{array}{l} \text{f 0} = 1 \\ \text{f n} = \text{n * f (n-1)} \end{array} \text{ in f}\right]\!\right] = \{(n, n!) \mid n \in \mathbb{N}\}$$

- Operational:

$$\frac{\langle\ c_1, s\ \rangle \Rightarrow s' \quad \langle\ c_2, s'\ \rangle \Rightarrow s''}{\langle\ c_1; c_2, s\ \rangle \Rightarrow s''}$$

- Axiomatic:

$$\{x = 3\}\ x := x + 1\ \{x = 4\}$$

Operational

Why operational?

- how the program is supposed to behave
  (not the function it computes)

Why small-step?

- full non-determinism:

$$(a + b) + c \to (a' + b) + c \to (a' + b) + c' \to (a' + b') + c'$$

- more precisely, interleaving: $f() + g()$
- non-termination

Approach

- top-down: formalize C from scratch
  C is too big (heavy syntax, preprocessor, function pointers. . .)
  too risky
- bottom-up: start with toy languages
  - imperative programs
  - side effects in expressions
  - procedures
  - functions with `return`, `exit` statements, with a stack
  - non determinism
  - . . .
  - C syntax
  - preprocessor
  - several possible instanciations of the architecture
  - . . . , . . . , . . .

The WHILE language

($\simeq$ Yves Bertot in coq contribs)

$$
\begin{aligned}
e \quad &::= \quad n \;\mid\; x \;\mid\; e \odot e \;\mid\; \Box e && (1)\\
i \quad &::= \quad \texttt{skip} \mid i\ ;\ i \mid \texttt{if}\ e\ \texttt{then}\ i\ \texttt{else}\ i && (2)\\
&\qquad \mid \texttt{while}\ e\ \texttt{do}\ i \mid x := e && (3)
\end{aligned}
$$

Semantics of expressions: $\langle\ e, s\ \rangle \Rightarrow n$ or $\langle\ e, s\ \rangle \to e'$
Semantics of instructions: $\langle\ i, s\ \rangle \Rightarrow s$ or $\langle\ i, s\ \rangle \to \langle\ i', s'\ \rangle$

Adding side effects

$\textsc{While}$ + side effects in expressions

$$\begin{aligned} e &::= n \mid x \mid x := e \mid e \odot e \mid \Box e && (4)\\ i &::= \texttt{skip} \mid i \,;\, i \mid \texttt{if } e \texttt{ then } i \texttt{ else } i && (5)\\ &\quad\ \mid \texttt{while } e \texttt{ do } i \mid e && (6) \end{aligned}$$

Semantics of expressions: $\langle\, e, s \,\rangle \Rightarrow \langle\, n, s' \,\rangle$ or $\langle\, e, s \,\rangle \to \langle\, e', s' \,\rangle$
Semantics of instructions: $\langle\, i, s \,\rangle \Rightarrow s$ or $\langle\, i, s \,\rangle \to \langle\, i', s' \,\rangle$

Example of difference: the semantics of `while`

Adding procedures

$\textsc{While}$ + side effects in expressions + procedures

$$e \quad ::= \quad n \mid x \mid x := e \mid e \odot e \mid \Box e \tag{7}$$

$$i \quad ::= \quad \texttt{skip} \mid i \,;\, i \mid \texttt{if}\ e\ \texttt{then}\ i\ \texttt{else}\ i \tag{8}$$

$$\mid \texttt{while}\ e\ \texttt{do}\ i \mid e \mid \color{red}{\texttt{call}\ x\ \texttt{with}\ (e, \ldots, e)} \tag{9}$$

- procedures in the context
- body of procedures compiled
- arguments on the stack
- explicit stack size

```
procedure "f" [arg1, arg2, arg3, ..]
  body of the procedure
main
  call "f" with [1, 4]
```

Adding functions: Return

$$
\begin{aligned}
e \quad ::= \quad & \underline{n} \mid x \mid e \odot e \mid \Box e & (10)\\
& x := e \mid f(e_1, \ldots, e_n) \mid & (11)\\
& e \mathbin{?} e : e \mid & (12)\\
& \texttt{wait}(i) & (13)\\
& & (14)\\
i \quad ::= \quad & \texttt{skip} \mid e \mid i; i \mid & (15)\\
& \texttt{while } e \texttt{ do } i \mid \texttt{if } e \texttt{ then } i \texttt{ else } i \mid & (16)\\
& \texttt{return } e \mid & (17)\\
& \texttt{exit } e & (18)
\end{aligned}
$$

The RETURN language – example

$P = (f, [a, b], a := a + 1; \texttt{return}\ (a + b)) : nil$

$P \vdash$

The RETURN language – example

$$P = (f, [a, b], a := a + 1; \texttt{return}\ (a + b)) : nil$$

$$P \vdash \quad \langle \quad f(x+2, 1) \quad , \quad (x, 1) : s \ \rangle$$

The RETURN language – example

$$P = (f, [a, b], a := a + 1; \texttt{return}\ (a + b)) : nil$$

$$\begin{array}{llcr} P \vdash & \langle & f(x+2,1) & , & (x,1) : s \rangle \\ & \rightarrow_e\ \langle & f(1+2,1) & , & (x,1) : s \rangle \end{array}$$

The RETURN language – example

$$P = (f, [a, b], a := a + 1; \texttt{return}\ (a + b)) : nil$$

$$\begin{array}{llccc}
P \vdash & & \langle & f(x+2,1) & , & (x,1) : s \ \rangle \\
& \rightarrow_e & \langle & f(1+2,1) & , & (x,1) : s \ \rangle \\
& \rightarrow_e & \langle & f(3,1) & , & (x,1) : s \ \rangle
\end{array}$$

The RETURN language – example

$$P = (f, [a, b], a := a + 1; \mathtt{return}\ (a + b)) : nil$$

$$\begin{array}{rlccr}
P \vdash & \langle & f(x+2,1) & , & (x,1) : s \rangle \\
\rightarrow_e & \langle & f(1+2,1) & , & (x,1) : s \rangle \\
\rightarrow_e & \langle & f(3,1) & , & (x,1) : s \rangle \\
\rightarrow_e & \langle & \mathtt{wait}(a := a + 1; \mathtt{return}\ (a+b)) & &
\end{array}$$

The RETURN language – example

$$P = (f, [a, b], a := a + 1; \texttt{return}\ (a + b)) : nil$$

$$\begin{array}{rcccl}
P \vdash & \langle & f(x+2,1) & , & (x,1) : s \rangle \\
\rightarrow_e & \langle & f(1+2,1) & , & (x,1) : s \rangle \\
\rightarrow_e & \langle & f(3,1) & , & (x,1) : s \rangle \\
\rightarrow_e & \langle & \texttt{wait}(a := a + 1; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \rangle
\end{array}$$

The RETURN language – example

$$P = (f, [a, b], a := a + 1; \texttt{return}\ (a + b)) : nil$$

$$\begin{array}{llccr}
P \vdash & \langle & f(x+2,1) & , & (x,1) : s \rangle \\
\rightarrow_e & \langle & f(1+2,1) & , & (x,1) : s \rangle \\
\rightarrow_e & \langle & f(3,1) & , & (x,1) : s \rangle \\
\rightarrow_e & \langle & \texttt{wait}(a := a+1; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \rangle \\
\rightarrow_e & \langle & \texttt{wait}(a := 3+1; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \rangle
\end{array}$$

The RETURN language – example

$$P = (f, [a, b], a := a + 1; \texttt{return}\ (a + b)) : nil$$

$$\begin{array}{llccr}
P \vdash & \langle & f(x+2,1) & , & (x,1) : s \,\rangle \\
\rightarrow_e & \langle & f(1+2,1) & , & (x,1) : s \,\rangle \\
\rightarrow_e & \langle & f(3,1) & , & (x,1) : s \,\rangle \\
\rightarrow_e & \langle & \texttt{wait}(a := a+1; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \,\rangle \\
\rightarrow_e & \langle & \texttt{wait}(a := 3+1; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \,\rangle \\
\rightarrow_e & \langle & \texttt{wait}(a := 4; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \,\rangle
\end{array}$$

$$P = (f, [a, b], a := a + 1; \mathtt{return}\ (a + b)) : nil$$

$$\begin{array}{rlccc}
P \vdash & \langle & f(x + 2, 1) & , & (x, 1) : s \ \rangle \\
\rightarrow_e & \langle & f(1 + 2, 1) & , & (x, 1) : s \ \rangle \\
\rightarrow_e & \langle & f(3, 1) & , & (x, 1) : s \ \rangle \\
\rightarrow_e & \langle & \mathtt{wait}(a := a + 1; \mathtt{return}\ (a + b)) & , & (a, 3) : (b, 1) : \mathtt{PROC} : (x, 1) : s \ \rangle \\
\rightarrow_e & \langle & \mathtt{wait}(a := 3 + 1; \mathtt{return}\ (a + b)) & , & (a, 3) : (b, 1) : \mathtt{PROC} : (x, 1) : s \ \rangle \\
\rightarrow_e & \langle & \mathtt{wait}(a := 4; \mathtt{return}\ (a + b)) & , & (a, 3) : (b, 1) : \mathtt{PROC} : (x, 1) : s \ \rangle \\
\rightarrow_e & \langle & \mathtt{wait}(4; \mathtt{return}\ (a + b)) & &
\end{array}$$

The RETURN language – example

$$P = (f, [a, b], a := a + 1; \texttt{return}\ (a + b)) : nil$$

$$\begin{array}{llccr}
P \vdash & \langle & f(x+2,1) & , & (x,1) : s \rangle \\
\rightarrow_e & \langle & f(1+2,1) & , & (x,1) : s \rangle \\
\rightarrow_e & \langle & f(3,1) & , & (x,1) : s \rangle \\
\rightarrow_e & \langle & \texttt{wait}(a := a+1; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \rangle \\
\rightarrow_e & \langle & \texttt{wait}(a := 3+1; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \rangle \\
\rightarrow_e & \langle & \texttt{wait}(a := 4; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \rangle \\
\rightarrow_e & \langle & \texttt{wait}(4; \texttt{return}\ (a+b)) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \rangle
\end{array}$$

The RETURN language – example

$$P = (f, [a, b], a := a + 1; \texttt{return}\ (a + b)) : nil$$

$$\begin{array}{rcccl}
P \vdash & \langle & f(x+2,1) & , & (x,1) : s \ \rangle \\
\rightarrow_e & \langle & f(1+2,1) & , & (x,1) : s \ \rangle \\
\rightarrow_e & \langle & f(3,1) & , & (x,1) : s \ \rangle \\
\rightarrow_e & \langle & \texttt{wait}(a := a+1; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
\rightarrow_e & \langle & \texttt{wait}(a := 3+1; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
\rightarrow_e & \langle & \texttt{wait}(a := 4; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
\rightarrow_e & \langle & \texttt{wait}(4; \texttt{return}\ (a+b)) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
\rightarrow_e & \langle & \texttt{wait}(\texttt{return}\ (a+b)) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle
\end{array}$$

$$P = (f, [a, b], a := a + 1; \mathtt{return}\ (a + b)) : nil$$

$$\begin{array}{llcll}
P \vdash & \langle & f(x+2,1) & , & (x,1) : s \rangle \\
\rightarrow_e & \langle & f(1+2,1) & , & (x,1) : s \rangle \\
\rightarrow_e & \langle & f(3,1) & , & (x,1) : s \rangle \\
\rightarrow_e & \langle & \mathtt{wait}(a := a+1; \mathtt{return}\ (a+b)) & , & (a,3) : (b,1) : \mathtt{PROC} : (x,1) : s \rangle \\
\rightarrow_e & \langle & \mathtt{wait}(a := 3+1; \mathtt{return}\ (a+b)) & , & (a,3) : (b,1) : \mathtt{PROC} : (x,1) : s \rangle \\
\rightarrow_e & \langle & \mathtt{wait}(a := 4; \mathtt{return}\ (a+b)) & , & (a,3) : (b,1) : \mathtt{PROC} : (x,1) : s \rangle \\
\rightarrow_e & \langle & \mathtt{wait}(4; \mathtt{return}\ (a+b)) & , & (a,4) : (b,1) : \mathtt{PROC} : (x,1) : s \rangle \\
\rightarrow_e & \langle & \mathtt{wait}(\mathtt{return}\ (a+b)) & , & (a,4) : (b,1) : \mathtt{PROC} : (x,1) : s \rangle \\
\rightarrow_e & \langle & \mathtt{wait}(\mathtt{return}\ (4+b)) & , & (a,4) : (b,1) : \mathtt{PROC} : (x,1) : s \rangle
\end{array}$$

The RETURN language – example

$$P = (f, [a, b], a := a + 1; \texttt{return}\ (a + b)) : nil$$

$$
\begin{array}{rlccl}
P \vdash & \langle & f(x+2,1) & , & (x,1) : s \rangle \\
\rightarrow_e & \langle & f(1+2,1) & , & (x,1) : s \rangle \\
\rightarrow_e & \langle & f(3,1) & , & (x,1) : s \rangle \\
\rightarrow_e & \langle & \texttt{wait}(a := a+1; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \rangle \\
\rightarrow_e & \langle & \texttt{wait}(a := 3+1; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \rangle \\
\rightarrow_e & \langle & \texttt{wait}(a := 4; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \rangle \\
\rightarrow_e & \langle & \texttt{wait}(4; \texttt{return}\ (a+b)) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \rangle \\
\rightarrow_e & \langle & \texttt{wait}(\texttt{return}\ (a+b)) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \rangle \\
\rightarrow_e & \langle & \texttt{wait}(\texttt{return}\ (4+b)) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \rangle \\
\rightarrow_e & \langle & \texttt{wait}(\texttt{return}\ (4+1)) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \rangle
\end{array}
$$

The RETURN language – example

$$P = (f, [a, b], a := a + 1; \texttt{return}\ (a + b)) : nil$$

$$
\begin{array}{lllcll}
P \vdash & & \langle & f(x+2,1) & , & (x,1) : s \ \rangle \\
& \rightarrow_e & \langle & f(1+2,1) & , & (x,1) : s \ \rangle \\
& \rightarrow_e & \langle & f(3,1) & , & (x,1) : s \ \rangle \\
& \rightarrow_e & \langle & \texttt{wait}(a := a+1; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
& \rightarrow_e & \langle & \texttt{wait}(a := 3+1; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
& \rightarrow_e & \langle & \texttt{wait}(a := 4; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
& \rightarrow_e & \langle & \texttt{wait}(4; \texttt{return}\ (a+b)) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
& \rightarrow_e & \langle & \texttt{wait}(\texttt{return}\ (a+b)) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
& \rightarrow_e & \langle & \texttt{wait}(\texttt{return}\ (4+b)) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
& \rightarrow_e & \langle & \texttt{wait}(\texttt{return}\ (4+1)) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
& \rightarrow_e & \langle & \texttt{wait}(\texttt{return}\ 5) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle
\end{array}
$$

The RETURN language – example

$$P = (f, [a, b], a := a + 1; \texttt{return}\ (a + b)) : nil$$

$$
\begin{array}{llccr}
P \vdash & \langle & f(x+2,1) & , & (x,1) : s \rangle \\
\to_e & \langle & f(1+2,1) & , & (x,1) : s \rangle \\
\to_e & \langle & f(3,1) & , & (x,1) : s \rangle \\
\to_e & \langle & \texttt{wait}(a := a+1; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \rangle \\
\to_e & \langle & \texttt{wait}(a := 3+1; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \rangle \\
\to_e & \langle & \texttt{wait}(a := 4; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \rangle \\
\to_e & \langle & \texttt{wait}(4; \texttt{return}\ (a+b)) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \rangle \\
\to_e & \langle & \texttt{wait}(\texttt{return}\ (a+b)) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \rangle \\
\to_e & \langle & \texttt{wait}(\texttt{return}\ (4+b)) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \rangle \\
\to_e & \langle & \texttt{wait}(\texttt{return}\ (4+1)) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \rangle \\
\to_e & \langle & \texttt{wait}(\texttt{return}\ 5) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \rangle \\
\to_e & \langle & \texttt{wait}(\texttt{return}\ 5) & , & (b,1) : \texttt{PROC} : (x,1) : s \rangle
\end{array}
$$

The RETURN language – example

$$P = (f, [a, b], a := a + 1; \texttt{return}\ (a + b)) : nil$$

$$
\begin{array}{rcccl}
P \vdash & \langle & f(x+2,1) & , & (x,1) : s \ \rangle \\
\rightarrow_e & \langle & f(1+2,1) & , & (x,1) : s \ \rangle \\
\rightarrow_e & \langle & f(3,1) & , & (x,1) : s \ \rangle \\
\rightarrow_e & \langle & \texttt{wait}(a := a+1; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
\rightarrow_e & \langle & \texttt{wait}(a := 3+1; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
\rightarrow_e & \langle & \texttt{wait}(a := 4; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
\rightarrow_e & \langle & \texttt{wait}(4; \texttt{return}\ (a+b)) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
\rightarrow_e & \langle & \texttt{wait}(\texttt{return}\ (a+b)) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
\rightarrow_e & \langle & \texttt{wait}(\texttt{return}\ (4+b)) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
\rightarrow_e & \langle & \texttt{wait}(\texttt{return}\ (4+1)) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
\rightarrow_e & \langle & \texttt{wait}(\texttt{return}\ 5) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
\rightarrow_e & \langle & \texttt{wait}(\texttt{return}\ 5) & , & (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
\rightarrow_e & \langle & \texttt{wait}(\texttt{return}\ 5) & , & \texttt{PROC} : (x,1) : s \ \rangle
\end{array}
$$

The Return language – example

$$P = (f, [a, b], a := a + 1; \texttt{return}\ (a + b)) : nil$$

$$
\begin{array}{rlccr}
P \vdash & \langle & f(x+2,1) & , & (x,1) : s \ \rangle \\
\to_e & \langle & f(1+2,1) & , & (x,1) : s \ \rangle \\
\to_e & \langle & f(3,1) & , & (x,1) : s \ \rangle \\
\to_e & \langle & \texttt{wait}(a := a+1; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
\to_e & \langle & \texttt{wait}(a := 3+1; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
\to_e & \langle & \texttt{wait}(a := 4; \texttt{return}\ (a+b)) & , & (a,3) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
\to_e & \langle & \texttt{wait}(4; \texttt{return}\ (a+b)) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
\to_e & \langle & \texttt{wait}(\texttt{return}\ (a+b)) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
\to_e & \langle & \texttt{wait}(\texttt{return}\ (4+b)) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
\to_e & \langle & \texttt{wait}(\texttt{return}\ (4+1)) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
\to_e & \langle & \texttt{wait}(\texttt{return}\ 5) & , & (a,4) : (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
\to_e & \langle & \texttt{wait}(\texttt{return}\ 5) & , & (b,1) : \texttt{PROC} : (x,1) : s \ \rangle \\
\to_e & \langle & \texttt{wait}(\texttt{return}\ 5) & , & \texttt{PROC} : (x,1) : s \ \rangle \\
\to_e & \langle & 5 & , & (x,1) : s \ \rangle
\end{array}
$$

The RETURN language – example

$$P = (f, [a, b], a := a + 1; \texttt{return}\ (a + b)) : nil$$

$$
\begin{array}{llccr}
P \vdash & \langle & f(x+2,1) & , & (x,1):s \rangle \\
\to_e & \langle & f(1+2,1) & , & (x,1):s \rangle \\
\to_e & \langle & f(3,1) & , & (x,1):s \rangle \\
\to_e & \langle & \texttt{wait}(a := a+1; \texttt{return}\ (a+b)) & , & (a,3):(b,1):\texttt{PROC}:(x,1):s \rangle \\
\to_e & \langle & \texttt{wait}(a := 3+1; \texttt{return}\ (a+b)) & , & (a,3):(b,1):\texttt{PROC}:(x,1):s \rangle \\
\to_e & \langle & \texttt{wait}(a := 4; \texttt{return}\ (a+b)) & , & (a,3):(b,1):\texttt{PROC}:(x,1):s \rangle \\
\to_e & \langle & \texttt{wait}(4; \texttt{return}\ (a+b)) & , & (a,4):(b,1):\texttt{PROC}:(x,1):s \rangle \\
\to_e & \langle & \texttt{wait}(\texttt{return}\ (a+b)) & , & (a,4):(b,1):\texttt{PROC}:(x,1):s \rangle \\
\to_e & \langle & \texttt{wait}(\texttt{return}\ (4+b)) & , & (a,4):(b,1):\texttt{PROC}:(x,1):s \rangle \\
\to_e & \langle & \texttt{wait}(\texttt{return}\ (4+1)) & , & (a,4):(b,1):\texttt{PROC}:(x,1):s \rangle \\
\to_e & \langle & \texttt{wait}(\texttt{return}\ 5) & , & (a,4):(b,1):\texttt{PROC}:(x,1):s \rangle \\
\to_e & \langle & \texttt{wait}(\texttt{return}\ 5) & , & (b,1):\texttt{PROC}:(x,1):s \rangle \\
\to_e & \langle & \texttt{wait}(\texttt{return}\ 5) & , & \texttt{PROC}:(x,1):s \rangle \\
\to_e & \langle & 5 & , & (x,1):s \rangle
\end{array}
$$

DONE!

Adding nondeterminism

- same grammar
- some rules differ

Forced left to right: $$\frac{\langle\ e_2 \mid s\ \rangle \to \langle\ e_2' \mid s'\ \rangle}{\langle\ \underline{n_1} + e_2 \mid s\ \rangle \to \langle\ \underline{n_1} + e_2' \mid s'\ \rangle}$$

Nondeterminism: $$\frac{\langle\ e_2 \mid s\ \rangle \to \langle\ e_2' \mid s'\ \rangle}{\langle\ e_1 + e_2 \mid s\ \rangle \to \langle\ e_1 + e_2' \mid s'\ \rangle}$$

Validation

Sanity checks:

- stating theorems
- writing programs
- executing programs
- proving the statements
- certifying the programs

Example of a program in RETURN

```
proc "gcd" with [a, b] begin
  If a = 0 then
    return b
  else
    If b = 0 then
      return a
    else
      If a > b then
        return (call "gcd" with [a - b, b])
      else
        return (call "gcd" with [b - a, a]).
end
main
  ignore (res := call "gcd" with [15, 6])
```

Dynamic small step!

These small-step semantics are:

- built to be deterministic
- small
- $\rightarrow$ automatic:
  a tactic in Ltac `sos_small_step`
  - on goals of the forms $\langle\ \_,\ \_\ \rangle \rightarrow^?_? \langle\ \_,\ \_\ \rangle$
  - apply the most obvious reduction
  - try to reduce the simple cases

visualizing the behavior of the program
some proofs are just “execution” of a program with just this tactic.

Functionality

Proof of the determinism in the RETURN language:

$$\forall a\ a_1\ a_2\ \ (a \to a_1) \wedge (a \to a_2) \Rightarrow a_1 = a_2$$

```
Lemma sos_functionality : forall p i e s s1 s2,
  (forall i1 i2,
    p ⊢ ⟨ i | s ⟩ →i ⟨ i1 | s1 ⟩ ->
    p ⊢ ⟨ i | s ⟩ →i ⟨ i2 | s2 ⟩ ->
    i1 = i2 /\ s1 = s2) /\
  (forall e1 e2,
    p ⊢ ⟨ e | s ⟩ →e ⟨ e1 | s1 ⟩ ->
    p ⊢ ⟨ e | s ⟩ →e ⟨ e2 | s2 ⟩ ->
    e1 = e2 /\ s1 = s2).
```

Functionality

About the proof:

- big proof, bigger proof term! (a lot of `inversion`)
- great sanity check (for an SOS behaviour)
- $=$ determinism of the language
- possible extraction of an interpreter

Functionality

About the proof:

- big proof, bigger proof term! (a lot of `inversion`)
- great sanity check (for an SOS behaviour)
- $=$ determinism of the language
- possible extraction of an interpreter

Interpreter

Again, for the return language

- statement:

```
Definition eval_partial p (t : toeval) (s : env) :
  match t with
    | Instr i =>
      { i's' | p ⊢ ⟨ i | s ⟩ →i  ⟨ fst i's' | snd i's' ⟩ } +
      { forall i' s', p ⊢ ⟨ i | s ⟩ →i  ⟨ i' | s' ⟩ -> False}
    | Expr  e =>
      { e's' | p ⊢ ⟨ e | s ⟩ →e ⟨ fst e's' | snd e's' ⟩ } +
      { forall e' s', p ⊢ ⟨ e | s ⟩ →e ⟨ e' | s' ⟩ -> False}
  end.
```

- quite long but why? Different ways to do so:
  - exhaustively (have to decide recursively)
  - use the functionality? (not enough, not directly decidable – probably)
- correspondence with an efficient algorithm?

Decidable equality

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

- `decide equality` does not work:
  $\leftarrow$ `expr` has a constructor having `list expr` as a premise
  Try with a mutually defined inductive list?
  $\leftarrow$ `expr` and `instr` are mutually recursive
- sanity check (no infinite objects in the syntax!)

Example of a certified program

Certification of the recursive version of the GCD algorithm

```
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) }.
```

Stack handling

$$p \vdash \langle \quad e \quad | \quad \underset{\uparrow}{s} \quad \rangle \rightarrow_e \langle \quad e' \quad | \quad \underset{\uparrow}{s'} \quad \rangle$$

these are stacks

- scope and variables implemented as an explicit stack
- can add a premise on $\rightarrow$ to consider the maximum length of s

Towards a C semantics

Beyond toy languages: for a complete C specification of the C99 standard

- making technichal choices:
  - step size
  - considering files, streams of I/O
  - preprocessor
  - ...
- interpreting the C99 standard ambiguities
  - interleaving
  - stack
  - ...
- a long work of translation from English to Coq
- validation of the semantics: readable grammar, readable rules
- proofs that existing semantics are sound according to this specification

Of course: a lot of work

There is a PhD proposal pending on this subject.

Towards a C semantics

Some approached points?

- lazy evaluation
  `0 && e`, `1 || e`

Towards a C semantics

Some approached points?

- lazy evaluation
  `0 && ` e, `1 || ` e
- Sequences points and undefinedness
  sequence points in `,`, `;`, `_(_)`, `&&`, `||`, `?:`

Towards a C semantics

Some approached points?

- lazy evaluation
  `0 && e`, `1 || e`
- Sequences points and undefinedness
  sequence points in `,`, `;`, `_(_)`, `&&`, `||`, `?:`
  undefined:
  `j = (i=0) + (i=0);`

Towards a C semantics

Some approached points?

- lazy evaluation
  `0 && e`, `1 || e`
- Sequences points and undefinedness
  sequence points in `,`, `;`, `_(_)`, `&&`, `||`, `?:`
  undefined:
  `j = (i=0) + (i=0);`
  (as in Norrish) multiset of side effects

Towards a C semantics

Some approached points?

- lazy evaluation
  `0 && e`, `1 || e`
- Sequences points and undefinedness
  sequence points in `,`, `;`, `_(_)`, `&&`, `||`, `?:`
  undefined:
  `j = (i=0) + (i=0);`
  (as in Norrish) multiset of side effects
- Stack handling
  function of "stack" on which the semantics depends. This function can be different for different semantics, as long as it is authorized by the formalized specification.

Done for the C semantics

What I have done:

- C syntax (casts: `(char)i`, ++, calls, `?:`)

Done for the C semantics

What I have done:

- C syntax (casts: `(char)i`, `++`, calls, `?:`)
- unchecked C semantics for most basic constructors (not `return`, `break`, `continue`)

Done for the C semantics

What I have done:

- C syntax (casts: `(char)i`, `++`, calls, `?:`)
- unchecked C semantics for most basic constructors (not `return`, `break`, `continue`)
- context trick!
  hard manipulation of expression:

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

Done for the C semantics

What I have done:

- C syntax (casts: `(char)i`, ++, calls, `?:`)
- unchecked C semantics for most basic constructors (not `return`, `break`, `continue`)
- context trick!

  hard manipulation of expression:

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

  just have to do:

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

Done for the C semantics

What I have done:

- C syntax (casts: `(char)i`, ++, calls, `?:`)
- unchecked C semantics for most basic constructors (not `return`, `break`, `continue`)
- context trick!

  hard manipulation of expression:

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

  just have to do:

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

  the expressions then look like `C[[e/-]]`

  then apply the context rule for $\rightarrow_e$

Of course, not complete. At all.

Remaining to do

- more examples in RETURN (and with nondeterminism) (most advanced / usable language for now)

Remaining to do

- more examples in RETURN (and with nondeterminism) (most advanced / usable language for now)
- extract a real interpreter for RETURN (ML)

Remaining to do

- more examples in RETURN (and with nondeterminism) (most advanced / usable language for now)
- extract a real interpreter for RETURN (ML)
- extract a interpreter for RETURN with nondeterminism (argument $\mathbb{N} \to \{0, 1\}$ and proof of completeness)

Remaining to do

- more examples in RETURN (and with nondeterminism) (most advanced / usable language for now)
- extract a real interpreter for RETURN (ML)
- extract a interpreter for RETURN with nondeterminism (argument $\mathbb{N} \to \{0, 1\}$ and proof of completeness)
- stack handling examples for RETURN and C

Remaining to do

- more examples in RETURN (and with nondeterminism) (most advanced / usable language for now)
- extract a real interpreter for RETURN (ML)
- extract a interpreter for RETURN with nondeterminism (argument $\mathbb{N} \rightarrow \{0, 1\}$ and proof of completeness)
- stack handling examples for RETURN and C
- complete the specification (not now!)

Remaining to do

- more examples in RETURN (and with nondeterminism)
  (most advanced / usable language for now)
- extract a real interpreter for RETURN (ML)
- extract a interpreter for RETURN with nondeterminism
  (argument $\mathbb{N} \to \{0, 1\}$ and proof of completeness)
- stack handling examples for RETURN and C
- complete the specification (not now!)
- work on differences with the compiler:
  evaluation of expression: stack or registers? (possible to avoid)

Remaining to do

- more examples in RETURN (and with nondeterminism)
  (most advanced / usable language for now)
- extract a real interpreter for RETURN (ML)
- extract a interpreter for RETURN with nondeterminism
  (argument $\mathbb{N} \to \{0, 1\}$ and proof of completeness)
- stack handling examples for RETURN and C
- complete the specification (not now!)
- work on differences with the compiler:
  evaluation of expression: stack or registers? (possible to avoid)
- virtual machine corresponding to the C semantics
  (really good sanity check, non-deterministic, but hard)

Questions

?