Semantyka i weryfikacja programów

Bartosz Klin

(slajdy Andrzeja Tarleckiego)

Instytut Informatyki
Wydział Matematyki, Informatyki i Mechaniki
Uniwersytet Warszawski

http://www.mimuw.edu.pl/~klin pok. 5680
klin@mimuw.edu.pl

Strona tego wykładu:
http://www.mimuw.edu.pl/~klin/sem19-20.html

Program Semantics & Verification

Bartosz Klin

(slides courtesy of Andrzej Tarlecki)

Institute of Informatics
Faculty of Mathematics, Informatics and Mechanics
University of Warsaw

http://www.mimuw.edu.pl/~klin office: 5680
klin@mimuw.edu.pl

This course: http://www.mimuw.edu.pl/~klin/sem19-20.html

Goto's

- Let's replace exceptions by the full control-flow catastrophe.

$$S \in \mathbf{Stmt} ::= \ldots \mid L{:}S \mid \mathbf{goto}\ L$$
$$L \in \mathbf{LAB} ::= \ldots$$

- Labels are visible inside the block in which they are declared
- No jumps into a block are allowed; jumps into other statements are okay

Semantics — sketch

- Yet another environment:

$$\mathbf{LEnv} = \mathbf{LAB} \to (\mathbf{Cont} + \{??\})$$

- Semantic functions get another environment parameter as before:

$$\mathcal{S}: \quad \mathbf{Stmt} \to \underbrace{\mathbf{VEnv} \to \mathbf{PEnv} \to \mathbf{LEnv} \to \mathbf{Cont} \to \mathbf{Cont}}_{\mathbf{STMT}}$$

$$\mathcal{D}_P: \mathbf{PDecl} \to \underbrace{\mathbf{VEnv} \to \mathbf{PEnv} \to \mathbf{LEnv} \to \mathbf{Cont}_{\mathbf{D_P}} \to \mathbf{Cont}}_{\mathbf{PDECL}}$$

- Semantic clauses for declarations and statements of the "old" forms take the extra parameter and disregard it (passing it "down").

Goto's — sketch of the semantics continues

- We add a declaration-like semantics for statements:

$$\mathcal{D}_S : \mathbf{Stmt} \to \mathbf{VEnv} \to \mathbf{PEnv} \to \mathbf{LEnv} \to \mathbf{Cont} \to \mathbf{LEnv}$$

- With a few trivial clauses, like:

$$\mathcal{D}_S[\![x := e]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa = \rho_L$$

and similarly for **skip**, **call** p etc., and for **goto** L, where no visible labels can be introduced. Since we cannot jump into blocks, also:

$$\mathcal{D}_S[\![\mathbf{begin}\ D_V\ D_P\ S\ \mathbf{end}]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa = \rho_L$$

Goto's — sketch of the semantics continues

- And then a few not quite so trivial clauses follow:

$$\mathcal{D}_S[\![S_1; S_2]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa = \mathcal{D}_S[\![S_1]\!]\ \rho_V\ \rho_P\ \rho_L\ (\mathcal{S}[\![S_2]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa) + \mathcal{D}_S[\![S_2]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa$$

$$\mathcal{D}_S[\![\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa = \mathcal{D}_S[\![S_1]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa + \mathcal{D}_S[\![S_2]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa$$

$$\mathcal{D}_S[\![\textbf{while } b \textbf{ do } S]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa = \mathcal{D}_S[\![S]\!]\ \rho_V\ \rho_P\ \rho_L\ (\mathcal{S}[\![\textbf{while } b \textbf{ do } S]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa)$$

$$\mathcal{D}_S[\![L{:}S]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa = (\mathcal{D}_S[\![S]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa)[L \mapsto \mathcal{S}[\![S]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa]$$

The only extra thing to explain here is "updating":

$$(\rho_L + \rho'_L)\ L = \begin{cases} \rho_L\ L & \text{if } \rho'_L\ L = ?? \\ \rho'_L\ L & \text{if } \rho'_L\ L \neq ?? \end{cases}$$

Goto's — sketch of the semantics continues

- And finally we need new clauses for the (usual) semantics of labelled statements, of jumps (trivial now) and of blocks — rather complicated:

$$\mathcal{S}[\![L{:}S]\!] = \mathcal{S}[\![S]\!]$$

$$\mathcal{S}[\![\mathbf{goto}\ L]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa = \kappa_L \text{ where } \kappa_L = \rho_L\ L$$

$$\mathcal{S}[\![\mathbf{begin}\ D_V\ D_P\ S\ \mathbf{end}]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa =$$
$$\mathcal{D}_V[\![D_V]\!]\ \rho_V\ \lambda\rho'_V{:}\mathbf{VEnv}.\mathcal{D}_P[\![D_P]\!]\ \rho'_V\ \rho_P\ \rho_L\ \lambda\rho'_P{:}\mathbf{PEnv}.$$
$$\mathcal{S}[\![S]\!]\ \rho'_V\ \rho'_P\ \rho'_L\ \kappa \text{ where } \rho'_L = \mathcal{D}_S[\![S]\!]\ \rho'_V\ \rho'_P\ (\rho_L + \rho'_L)\ \kappa$$

. . . and perhaps not quite right?

- one should really check if the labels within a block are unique — this is easy!
- labels within a block should be visible within procedure declarations in this block

callcc

- short for **call-with-current-continuation**
- goto on steroids
- featured in **Scheme**, **SML**, **Haskell**, **Ruby**, but also in fancy libraries for **C++**
- We shall not define its semantics, as it does not mix very well with TINY, an imperative language with a trivial type system.
- Instead, we will explain how it works, rather informally and by example.
- This illustrates how knowledge of semantic concepts can help the programmer to learn a programming language concept.

Taxonomy of jumps

	Static (lexical)	Dynamic
Outward only	**break** **return**	**throw** **setjmp()/longjmp()**
Arbitrary	**goto**	**callcc**

callcc explained

- **callcc** is a function that takes one argument.
- As a programmer, you typically need to prepare such an argument (call it **f**).
- Your **f** should be a function that takes one argument (call it **k**).
- Normally you never need to prepare **k**; your **f** should be prepared to get one.
- You can expect **k** to be a function that takes one argument.
- If you call **callcc f**:
  - **f** is called with a magical argument **k**
  - if at some point you call **k** with an argument **v**, then **f** is immediately terminated and **callcc f** takes value **v**.
  - if **f** terminates normally and returns a value **v**, then **callcc f** takes value **v**.
- *The above works even if* **f** *returns* **k***!* If **k** is called after exiting from **f**, the call stack may need to be rebuilt.

Examples (in a functional-like pseudocode)

```
let f k = k 42
in callcc f
```

evaluates to 42

```
let f k = (k 42) + 25
in callcc f
```

evaluates to 42

```
let f k =
  for x in L do
    if test x then k x
in callcc f
```

returns the first element of L for which test holds

```
z := ...;
let f k = (z := k;  1)
in (callcc f) + 1;
```

returns 2 and stores the "+1" function in z

The yin-yang puzzle

Even the author of this program could not understand why it does what it does.

```
let
  yin  = ((\c -> (print 0) c) (callcc id))
  yang = ((\c -> (print 1) c) (callcc id))
in
  (yin yang)
```

So, what does this thing do and why?

"Standard semantics"

- continuations (to handle jumps of various kinds, and simplify notation)
- careful classification of various domains of values (assignable, storable, output-able, closures, etc) with the corresponding semantics of expressions (of various kinds)
- Scott domains and domain equations
- continuous functions only
- ...

... to be explained ...