Limits of Computation

4 - WHILE-Semantics

Bernhard Reus

Last time

- we introduced WHILE,
- a simple imperative untyped language,
- which has a built-in data type of binary trees (lists)
- that can be used to encode other data types.

WHILE Language Semantics

- Like Turing for his machines, we need to define what it *means to execute* WHILE *programs,*
- i.e. we need to define an exact, finitely expressible (operational) semantics…
- …proving WHILE *programs* can be used as "effective procedures"!

THIS TIME

```
program  read X {
    Y := nil;
    while X {
      Y := cons hd X Y;
      X := tl X
     }
  }
  write Y
```

a WHILE program, what is its semantics?

Recall: Syntax of WHILE

Expressions

⟨*expression*⟩	::=	⟨*variable*⟩	(variable expression)
	|	`nil`	(atom nil)
	|	`cons` ⟨*expression*⟩ ⟨*expression*⟩	(construct tree)
	|	`hd` ⟨*expression*⟩	(left subtree)
	|	`tl` ⟨*expression*⟩	(right subtree)
	|	`(` ⟨*expression*⟩ `)`	(right subtree)

Statement (lists)

⟨*block*⟩	::=	`{` ⟨*statement-list*⟩ `}`	(block of commands)
	|	`{ }`	(empty block)
⟨*statement-list*⟩	::=	⟨*command*⟩	(single command list)
	|	⟨*command*⟩ `;` ⟨*statement-list*⟩	(list of commands)
⟨*elseblock*⟩	::=	`else` ⟨*block*⟩	(else-case)
⟨*command*⟩	::=	⟨*variable*⟩ `:=` ⟨*expression*⟩	(assignment)
	|	`while` ⟨*expression*⟩ ⟨*block*⟩	(while loop)
	|	`if` ⟨*expression*⟩ ⟨*block*⟩	(if-then)
	|	`if` ⟨*expression*⟩ ⟨*block*⟩ ⟨*elseblock*⟩	(if-then-else)

Programs

⟨*program*⟩	::=	⟨*name*⟩ `read` ⟨*variable*⟩ ⟨*block*⟩ `write` ⟨*variable*⟩

A sample program

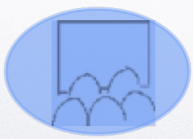
```
program  read X {
    Y := nil;
    while X {
      Y := cons hd X Y;
      X := tl X
     }
  }
  write Y
```

What does `program` do?

Sample Programs on Numbers

```
prog1 read X {
   X := cons nil X
}
write X
```

```
prog2 read X {
   X := tl X
}
write X
```



- what do `prog1` and `prog2` compute on (encoded) numbers?

Convention

- we can only have one input, so if we need more than one argument …
- … we always wrap them in a list.
- We could also wrap one argument in a list (then it would be totally uniform) but we won't do that for simplicity.

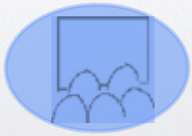
Sample Programs on Numbers

```
prog read L {
    X := hd L;
    Y := hd tl L;
    while X {
          Y := cons nil Y;
          X: = tl X
          }
 }
 write Y
```

L is argument list

supposed to contain (at least) two elements X and Y to encode two arguments

What does prog compute?



Semantics of WHILE

Semantics of Programs

$$[\![\_]\!]^{\mathtt{WHILE}} : \mathbb{D} \to \mathbb{D}_\perp$$

"undefined"

semantic brackets, semantic function

"function space"

map input to output

$$[\![p]\!]^{\mathtt{WHILE}}(d) = e$$

p on input d returns (writes) e

$$[\![p]\!]^{\mathtt{WHILE}}(d) = \perp$$

p on input d diverges (so no result)

The semantics of a program is its behaviour i.e.

the description of the program's output for any given input.

Stores for `WHILE`

- programs manipulate variables,
- so to determine a program's meaning (semantics) we need a store holding the values of its variables.
- This corresponds to the tapes used in Turing Machines to store values (not directly addressable like in `WHILE`).

another reason why `WHILE` is better than TM

Stores for WHILE

- Stores are sets of (key,value)-pairs, where
- key is an identifier (variable name)
- value is data element (binary tree)

$$Store = Set(VariableName \times \mathbb{D})$$

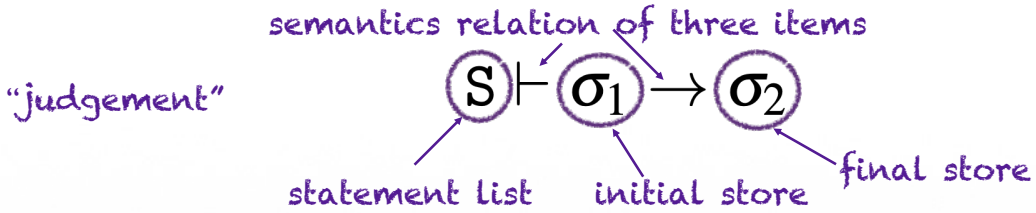
$$\{\mathtt{X}_1 : d_1, \mathtt{X}_2 : d_2, \ldots, \mathtt{X}_n : d_n\}$$

Store Operations for WHILE

- lookup X $\quad \sigma(\mathtt{X})$
- update X with d $\quad \sigma[\mathtt{X} := d] = \sigma \setminus \{(\mathtt{X}, val)\} \cup \{(\mathtt{X}, d)\}$
- initial store for input d $\quad \sigma_0^p(d) = \{\mathtt{X} : d\}$

X has value d, the other variables are implicitly initialised with nil

Semantics of Commands



"cool" notation for $(S, \sigma_1, \sigma_2) \in R_{\text{SemanticsStmtList}}$

i.e. a ternary relation on $\mathit{StatementList} \times \mathit{Store} \times \mathit{Store}$

that describes the operational semantics of S

defined inductively over the syntax (details in book)

We won't define this relation formally, but it can be done.

Semantics of WHILE-programs

Definition Let `p read X {S} write Y` be a WHILE-program where S is a statement list. The semantics of p is defined as follows:

$$[\![p]\!]^{\text{WHILE}}(d) = \begin{cases} e & \text{if } S \vdash \sigma_0^p(d) \to \sigma \text{ and } \sigma(Y) = e \\ \bot & \text{otherwise} \end{cases}$$

lookup result

- In other words: the output is e if executing the program's body S in the initial store (with the input variable set to d) terminates, and the output variable in the result state has value e; otherwise it is undefined.

END

Next time:
Extended
`WHILE`-programs for
convenience