Limits of Computation

4 - WHILE-Semantics
5- Extensions of the WHILE language

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;                (* initi
    while X {                (* run t
      Y := cons hd X Y;      (* appen
      X := tl X              (* remov
     }
  }
  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)

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;                (* initialise accumulator Y *)
    while X {                (* run through input list X *)
      Y := cons hd X Y;      (* append first element of X to Y *)
      X := tl X              (* remove first element from 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
```

```
(* X is first argument  m *)
(* Y is second argument  n *)
(* run through X *)
(* Y: = Y+1 *)
(* X: = X-1 *)
(* finally, Y is m+n *)
```

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

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

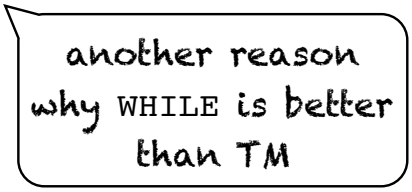
p on input d diverges (so no result)

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`).



Stores for WHILE

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

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

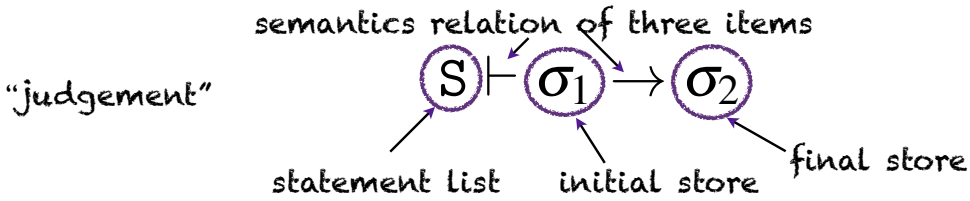
$$\{\mathtt{X}_1 : d_1, \mathtt{X}_2 : d_2, \ldots, \mathtt{X}_n : d_n\}$$

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 $StatementList \times Store \times Store$

that describes the operational semantics of S

defined inductively over the syntax (details in book)

Semantics of WHILE-programs

Definition 4.5. 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:

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

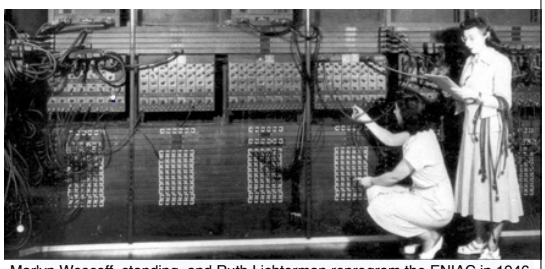
lookup result

- 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.

WHILE–extensions (Syntax sugar)

Programming Convenience

- we present some extensions to the core WHILE-language
- for convenience and ease of programming
- these extensions **do not require** additional semantics, they are "syntax sugar"
- i.e. they can be translated (away) into core WHILE



Marlyn Wescoff, standing, and Ruth Lichterman reprogram the ENIAC in 1946.

In the early days of computers, "programming" meant "plugging", not very convenient!

www.quora.com/How-was-the-very-first-programming-software-made

Core WHILE

- no built-in equality (only test for nil)
- no procedures or macros
- no number literals (nor Boolean literals, tree literals)
- no built in list notation
- no case/switch statement
- only one "atom" at leaves of trees: nil

Equality

- Equality needs to be programmed (exercises) in core WHILE
- Extended WHILE uses a new expression:

$$\begin{aligned}\langle expression\rangle ::= &\ \ldots \\ &\mid \langle expression\rangle = \langle expression\rangle \\ &\vdots\end{aligned}$$

e.g. `if X=Y { Z:= X } else {Z:= Y}`

Literals

- literals abbreviate constant values
- on our case: *natural* numbers or Boolean values
- Extended `WHILE` uses new expressions:

$$\langle \textit{expression} \rangle ::= \ldots \mid \langle \textit{number} \rangle \mid \vdots$$

$$\langle \textit{expression} \rangle ::= \ldots \mid \texttt{true} \mid \texttt{false} \mid \vdots$$

e.g. `if true { Z:= cons 3 cons 1 nil }`

Lists

- lists can be encoded in our datatype but explicit syntax is nicer
- Extended `WHILE` uses new expressions:

$$\begin{array}{lll}
\langle \textit{expression} \rangle & ::= \ldots & \\
& \mid [\,] & \text{(empty list constructor)} \\
& \mid [\,\langle \textit{expression-list} \rangle\,] & \text{(nonempty list constructor)} \\
& \vdots & \\
\langle \textit{expression-list} \rangle & ::= \langle \textit{expression} \rangle & \text{(single expression list)} \\
& \mid \langle \textit{expression} \rangle\,,\,\langle \textit{expression-list} \rangle & \text{(multiple expression list)}
\end{array}$$

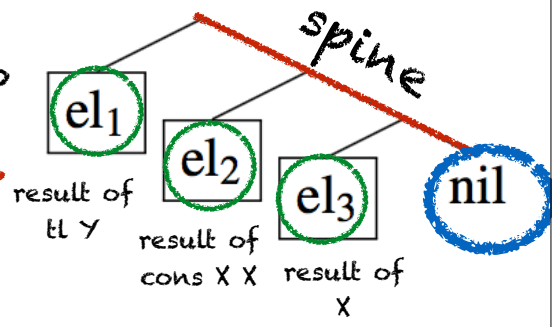
List Example

```
[tl Y, cons X X, X]
```

translates to

```
cons (tl Y)
  (cons (cons X X)
        cons X nil)
```

evaluates to



Macro Calls

- We don't have procedures but we can implement "macro calls" that allows one:
  - to write more readable code
  - to write modular code as macro code can be replaced without having to change the program.

Procedures provide abstraction & modularisation

Syntax of Macro Calls

- Macro calls use angle brackets <…> around the name of the program called
- and one argument (programs have one argument)
- Extended `WHILE` uses new assignment command:

$$\begin{aligned}\langle command \rangle ::= & \ldots \\ & \mid \langle variable \rangle := < \langle name \rangle > \langle expression \rangle \\ & \vdots\end{aligned}$$

Macro Calls Example

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

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

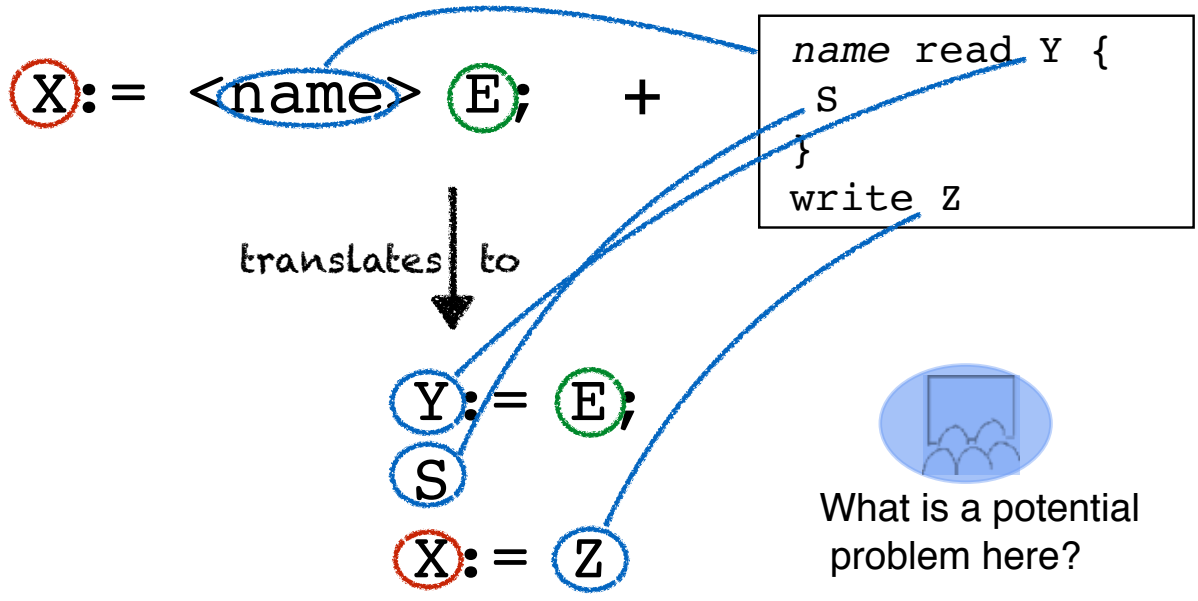
```
add read L {
  X:= hd L;
  Y:= hd tl L;
  while X {
        X := <pred> X;
        Y := <succ> Y
        }
}
write Y
```

Semantics of Macro Calls

`X:= <name> E`

1. Evaluate argument `E` (expression) to obtain d
2. Run the body of macro `name` with input d
3. And obtain as result r
4. Assign the value r to variable `X`

Translate Macros Calls



Switch Statement

luxurious form of if-then-else cascade

⟨*command*⟩ ::= ...
| `switch` ⟨*expression*⟩ { ⟨*rule-list*⟩ }
| `switch` ⟨*expression*⟩ { ⟨*rule-list*⟩
`default` : ⟨*statement-list*⟩ }
⋮

⟨*rule*⟩ ::= `case` ⟨*expression-list*⟩ : ⟨*statement-list*⟩

⟨*rule-list*⟩ ::= ⟨*rule*⟩
| ⟨*rule*⟩ ⟨*rule-list*⟩

Switch Example

```
switch X  {
 case 0           :  Y := 0
 case 1, 3        :  Y := 1
 case cons 2 nil  :  Y := 2
 }
```

evaluates to [2]

translates to

```
if X = 0
  { Y := 0 }
else { if X = 1
        { Y := 1 }
       else
          { if X = 3
              { Y := 1 }
            else
                {  if X = cons 2 nil
                     { Y := 2 }
                }
          }
      }
```

Extra Atoms

- Atoms are the "labels" at the leaves of binary trees.
- So far only one atom: nil
- Add more to simplify encodings.
- Extended `WHILE` uses new expression(s):

$$\langle expression \rangle ::= \ldots \mid \texttt{a} \quad (a \in Atoms) \mid \vdots$$

Only finitely many.
Why?

END

Next time:
`WHILE`-programs as
`WHILE`-data