Limits of Computation

6 - Programs as Data Objects

Bernhard Reus

So far...

- "effective procedure" = `WHILE`-program
- introduced `WHILE`-language with binary tree data type ...
- ... that can also be viewed as a type of (arbitrary deeply) nested lists
- and extended `WHILE` for convenience

WHILE-programs as lists

THIS TIME

- We show how WHILE-programs can be ***data objects*** usable in another WHILE-program

```
[0,
 [[:=,1,[quote,nil]],
  [while,[var,0],
      [  [:=,1,[cons,[hd,[var,0]],[var,1]]],
         [:=,0,[tl,[var,0]]]
      ]
 ]],
 1]
```

A WHILE-program abstract syntax tree encoded as list

Programs as Input or Output

- Compiler
  program transformer which takes a program and translates it into an *equivalent* program, most likely in another language;
- Interpreter
  takes a program *and* its input data, and returns the result of applying the program to that input.
- Program Specialiser
  takes a *program with two inputs and* one data for one of the inputs and *partially evaluates* the program with the one given data producing a new program with one input only (more on that later).

Programming Languages

our notion, formally

Definition 6.1. A *programming language* L consists of

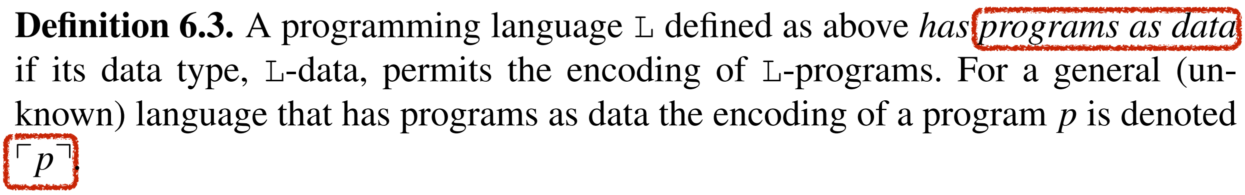
1. two sets: L-programs (the set of L-programs) and L-data (the set of data values described by the datatype used by this language)[1].
2. A function $[\![\_]\!]^{\mathtt{L}} : \mathtt{L}\text{-programs} \to (\mathtt{L}\text{-data} \to \mathtt{L}\text{-data}_{\bot})$ which maps L-programs into their semantic behaviour, namely a partial function mapping inputs to outputs, which are both in L-data.

PL with Pairing

Definition 6.2. A programming language L defined as above *has pairing* if its data type, L-data, permits the encoding of pairs. For a general (unknown) language that has pairing we denote pairs (a,b), i.e. using parenthesis and a comma.

Does WHILE have pairing?

PL with Programs As Data



Definition 6.3. A programming language `L` defined as above *has programs as data* if its data type, `L`-data, permits the encoding of `L`-programs. For a general (unknown) language that has programs as data the encoding of a program p is denoted $\ulcorner p \urcorner$.

The purpose of this session is
to show that `WHILE` has programs as data.

Programs as Data

- If language `L` has "***programs as data***" we can write compilers, interpreters, and specialisers in `L`.
- We want `WHILE` to have "*programs as data*".
- Thus we need a representation of `WHILE` programs as binary tree
- It is natural to use ***abstract syntax trees***

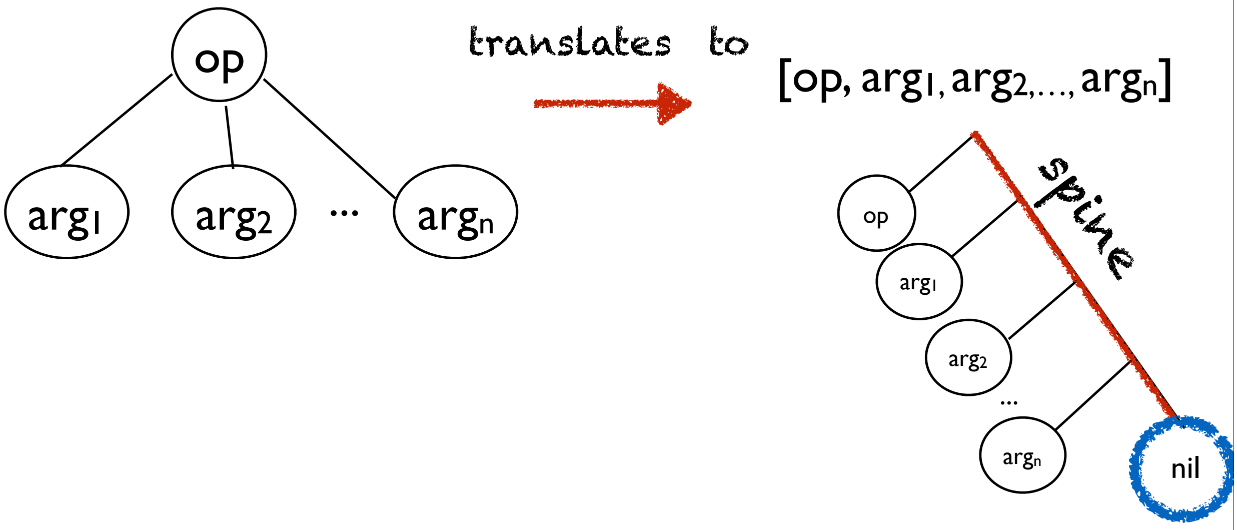
Interpreter

our notion, formally

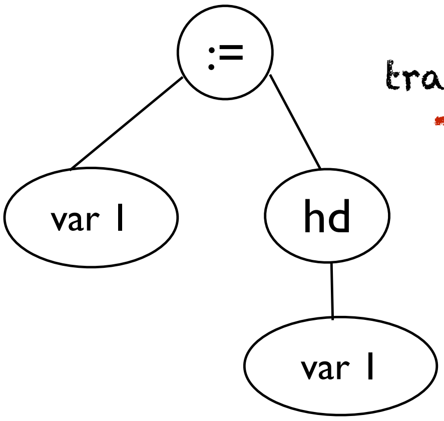
Definition 6.4. Assume S has programs as data, S-data $\subseteq$ L-data and L has pairing. An interpreter `int` for a language S written in L must fulfil the following equation for any given S-program p and $d \in$ S-data:

$$\boxed{[\![\texttt{int}]\!]^{\mathtt{L}}(\ulcorner p \urcorner, d) = [\![p]\!]^{\mathtt{S}}(d)} \tag{6.1}$$

Abstract Syntax Trees as lists

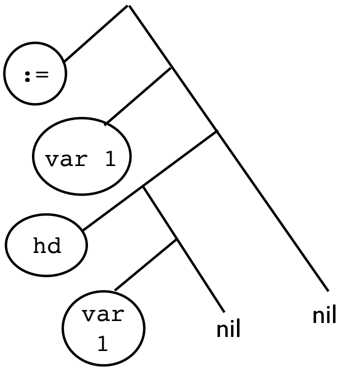


AST as list `Y:= hd Y` (Y is 1st variable)



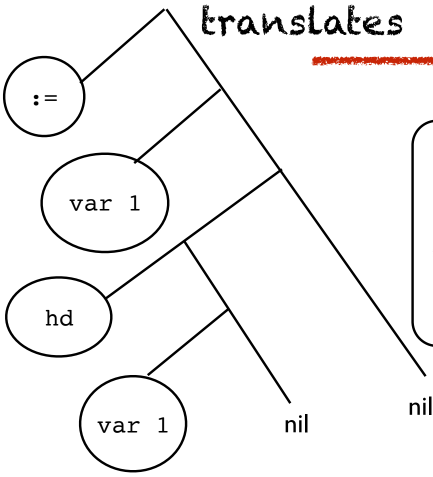
translates to →

`[:=,var1,[hd,var1]]`



var I needs to be unfolded as list too, see next slide

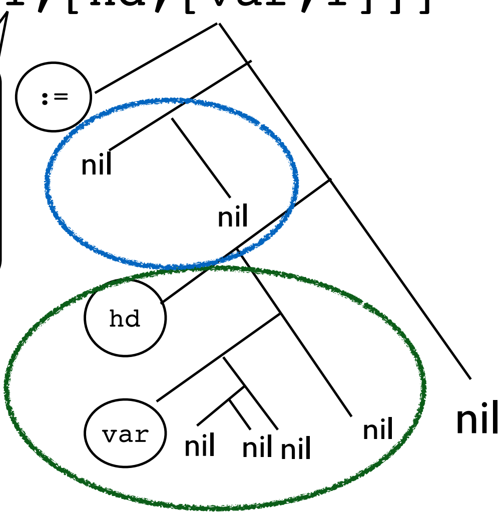
AST as list `Y:= hd Y` (Y is `var 1`)



translates to →

`[:=,1,[hd,[var,1]]]`

Simplification: we do only need to store the variable name (i.e. number), as we can only assign to variables



What to do with var etc?

```
[:=,1,[hd,[var,1]]]
```

These are not yet trees/lists:

:= var hd

Answer: either introduce them as *additional atoms* or *encode them* (uniquely) as numbers.

Programs as data in `WHILE`

- We are now in a position to define more exactly how the list encoding of abstract syntax trees work.
- Lists are themselves encoded as binary trees.
- Let's go:

WHILE programs in $\mathbb{D}$

$\ulcorner \mathtt{progname\ read\ X\ \{S\}\ write\ Y} \urcorner = [\mathit{varnum}_{\mathtt{X}}, \ulcorner\{\mathtt{S}\}\urcorner, \mathit{varnum}_{Y}]$

commands

$\ulcorner \mathtt{while\ E\ B} \urcorner = [\mathtt{while}, \ulcorner \mathtt{E} \urcorner, \ulcorner \mathtt{B} \urcorner]$

$\ulcorner \mathtt{X := E} \urcorner = [:=, \mathit{varnum}_{\mathtt{X}}, \ulcorner \mathtt{E} \urcorner]$

$\ulcorner \mathtt{if\ E\ B_T\ else\ B_E} \urcorner = [\mathtt{if}, \ulcorner \mathtt{E} \urcorner, \ulcorner \mathtt{B_T} \urcorner, \ulcorner \mathtt{B_E} \urcorner]$

$\ulcorner \mathtt{if\ E\ B} \urcorner = [\mathtt{if}, \ulcorner \mathtt{E} \urcorner, \ulcorner \mathtt{B} \urcorner, [\,]]$

$\ulcorner \{\mathtt{C_1; C_2; \ldots; C_n}\} \urcorner = [\ulcorner \mathtt{C_1} \urcorner, \ulcorner \mathtt{C_2} \urcorner, \ldots, \ulcorner \mathtt{C_n} \urcorner]$

expressions

$\ulcorner \mathtt{nil} \urcorner = [\mathtt{quote}, \mathtt{nil}]$

$\ulcorner \mathtt{X} \urcorner = [\mathtt{var}, \mathit{varnum}_{\mathtt{X}}]$

$\ulcorner \mathtt{cons\ E\ F} \urcorner = [\mathtt{cons}, \ulcorner \mathtt{E} \urcorner, \ulcorner \mathtt{F} \urcorner]$

$\ulcorner \mathtt{hd\ E} \urcorner = [\mathtt{hd}, \ulcorner \mathtt{E} \urcorner]$

$\ulcorner \mathtt{tl\ E} \urcorner = [\mathtt{tl}, \ulcorner \mathtt{E} \urcorner]$

Example

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

```
X is var 0
Y is var 1
```

translate program into data

```
[0,
 [[:=,1,[quote,nil]],
  [while,[var,0],
      [  [:=,1,[cons,[hd,[var,0]],[var,1]]],
         [:=,0,[tl,[var,0]]]
      ]
 ]],
 1]
```

Programs-as-data in *hWhile*

- We can now write compilers, interpreters, specializers in `WHILE` using abstract syntax trees in list notation ("programs-as-data") instead of string representation.
- Thus we do not have to care about parsing programs.
- In *`hwhile`* (see Canvas) we can use the -u flag to produce this list representation:

```
hWhile -u reverse.while
```

```
[ 0
,
    [ [@:=, 1, [@quote, nil]]
    ,
        [ @while, [@var, 0]
        ,
            [ [@:=, 1, [@cons, [@hd, [@var, 0]], [@var, 1]]]
            , [@:=, 0, [@tl, [@var, 0]]]
            ]
        ]
    ]
, 1
]
```

`hWhile` uses @ to indicate special atoms

A note on `hWhile` output

- *hWhile* output by default is given as binary tree:

```
./hwhile add [3,4]
<nil.<nil.<nil.<nil.<nil.<nil.<nil.nil>>>>>>>
```

- *use flags to determine the "type" in which it is presented*

```
./hwhile -i add [3,4]
7
```

integer

```
./hwhile -l add [3,4]
[nil,nil,nil,nil,nil,nil,nil]
```

list of trees

```
./hwhile -li add [3,4]
[0, 0, 0, 0, 0, 0, 0]
```

list of integers

A note on `hWhile` output

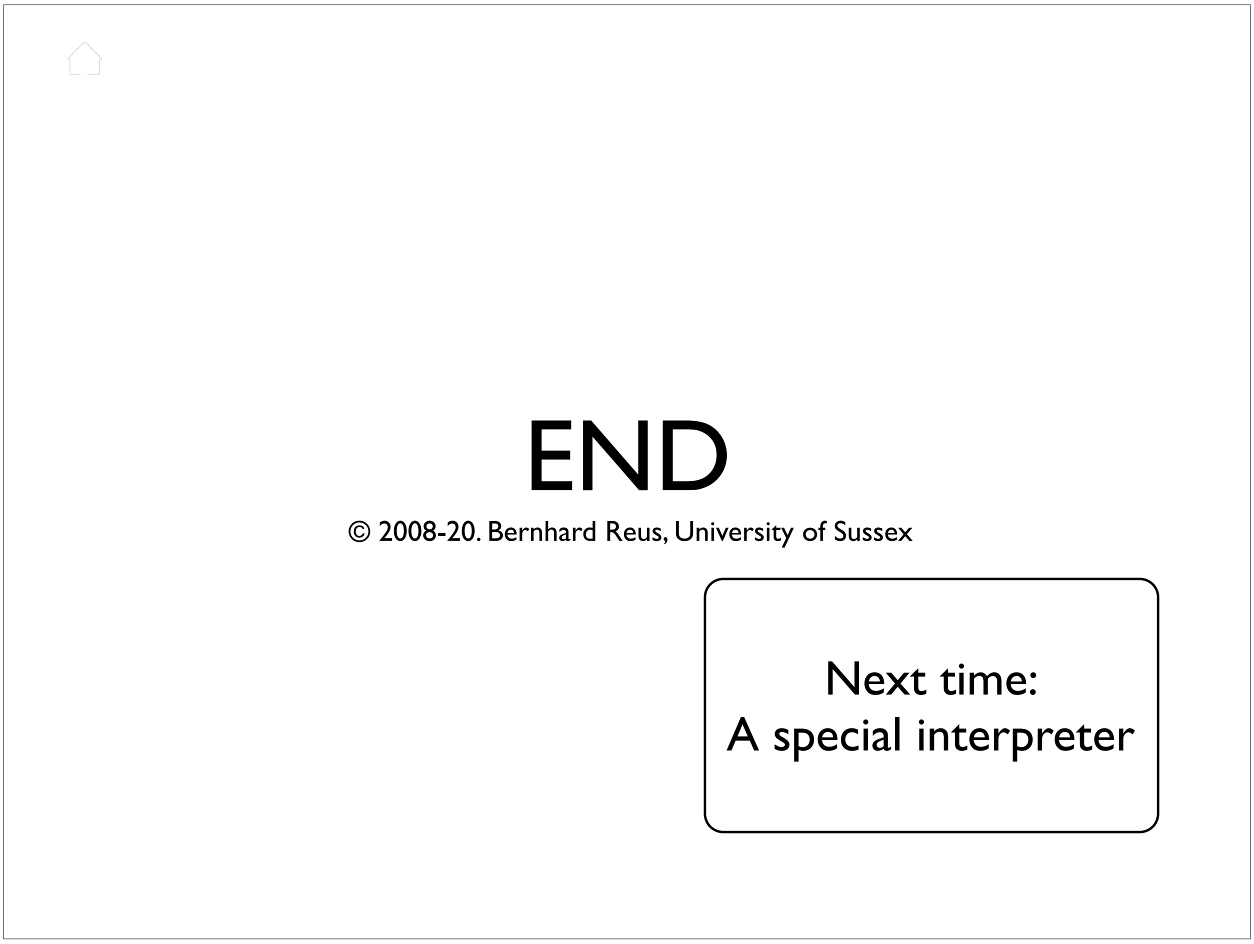
- *There are more output formats, to see them all run:*

```
./hwhile -h
```

- *Look at this one, can you explain it?*

```
/hwhile -La  add [3,4]
@doWhile
```

`-La` ?



END

Next time:
A special interpreter