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

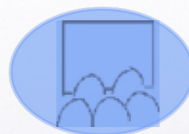**Definition** A *programming language* L consists of

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) .
2. A function $[\![\_]\!]^L : \text{L-programs} \to (\text{L-data} \to \text{L-data}_\perp)$ 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 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?

Answer: Yes, use [a,b] or <a.b>

PL with Programs As Data

Definition 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

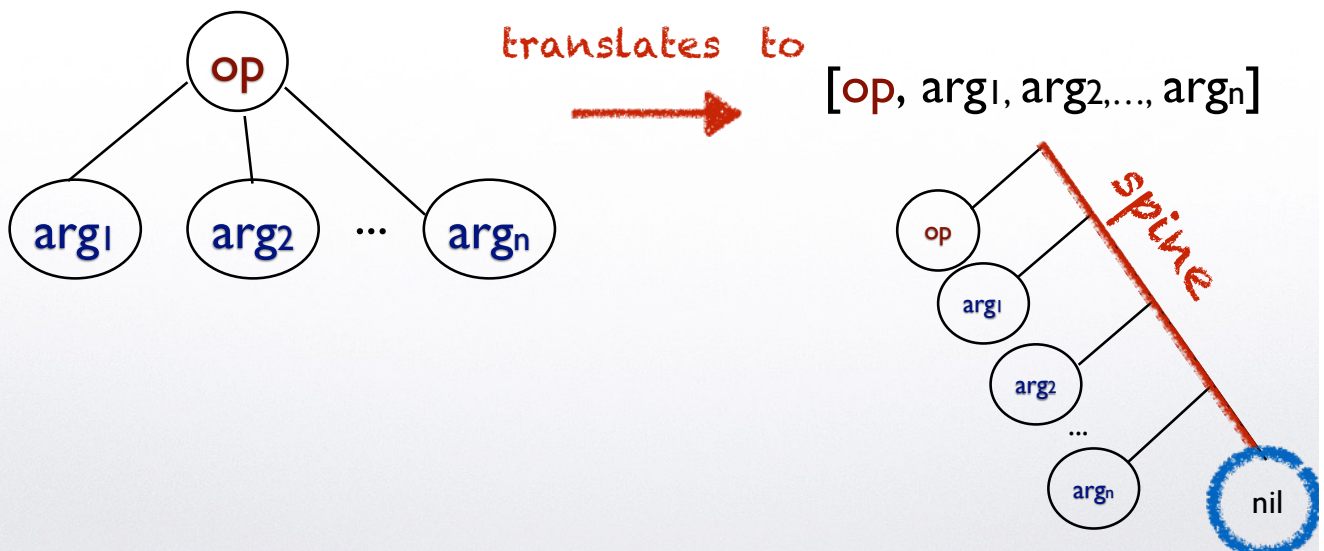- It is natural to use ***abstract syntax trees***

Interpreter

our notion, formally

Definition   Assume S has programs as data, S-data ⊆ 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:

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

Abstract Syntax Trees as lists

translates to

[op, arg1, arg2,…, argn]

spine

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

translates to

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

:=

var 1   hd

var 1

var 1

needs to be unfolded as list too, see next slide

:=
var 1
hd
var 1   nil   nil

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

translates to

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

:=

var 1

hd

var 1   nil

nil

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

:=
nil
nil

hd

var   nil   nil nil   nil   nil
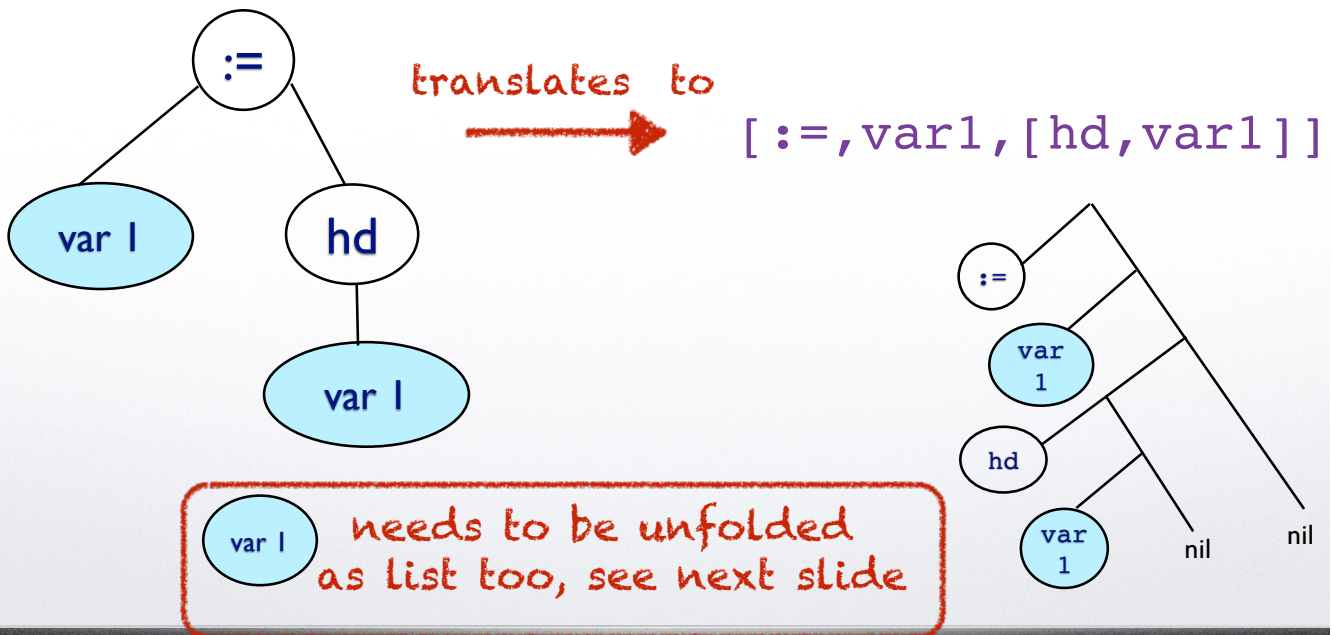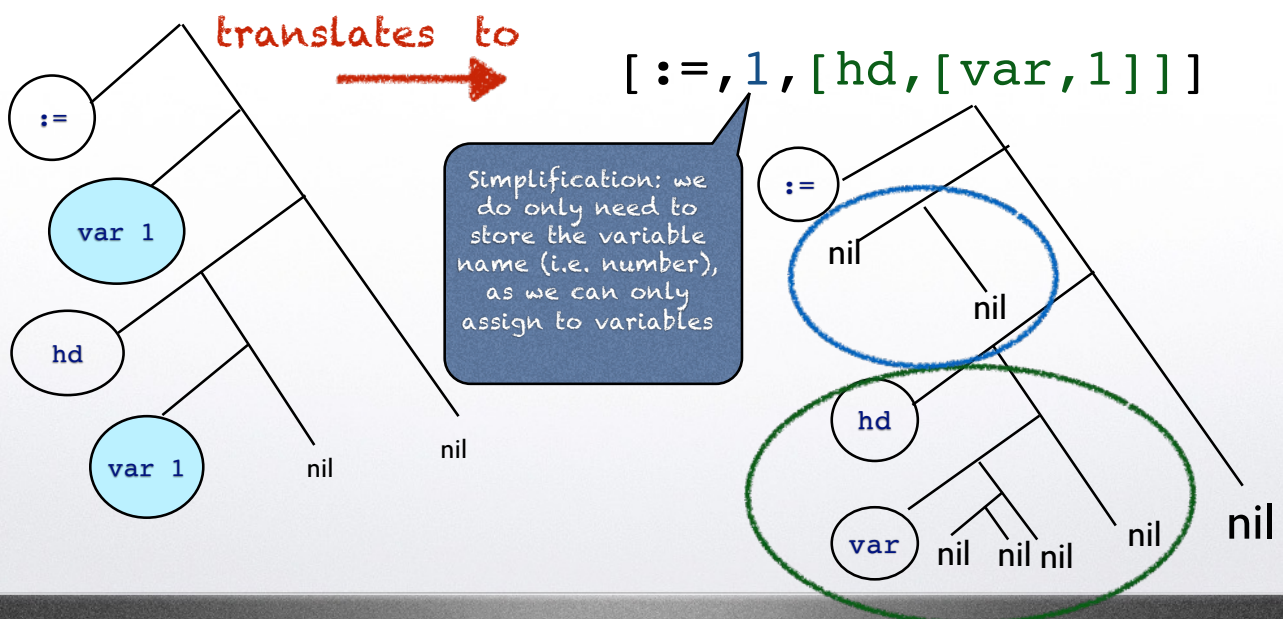
What to do with (var) etc?

`[:=,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:

$\ulcorner\texttt{progname read X \{S\} write Y}\urcorner =$ $\left[varnum_X, \ulcorner\{\!|\mathbf{S}|\!\}\urcorner, varnum_Y\right]$

commands

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

$\ulcorner\texttt{X} := \texttt{E}\urcorner$ $=$ $[:=, varnum_X, \ulcorner\texttt{E}\urcorner]$

$\ulcorner\texttt{if E } \texttt{B}_T \texttt{ else } \texttt{B}_E\urcorner$ $=$ $[\texttt{if}, \ulcorner\texttt{E}\urcorner, \ulcorner\texttt{B}_T\urcorner, \ulcorner\texttt{B}_E\urcorner]$

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

$\ulcorner\{\texttt{C}_1; \texttt{C}_2; \dots; \texttt{C}_n\}\urcorner$ $=$ $[\ulcorner\texttt{C}_1\urcorner, \ulcorner\texttt{C}_2\urcorner, \dots, \ulcorner\texttt{C}_n\urcorner]$

expressions

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

$\ulcorner\texttt{X}\urcorner$ $=$ $[\texttt{var}, varnum_X]$

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

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

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

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:*
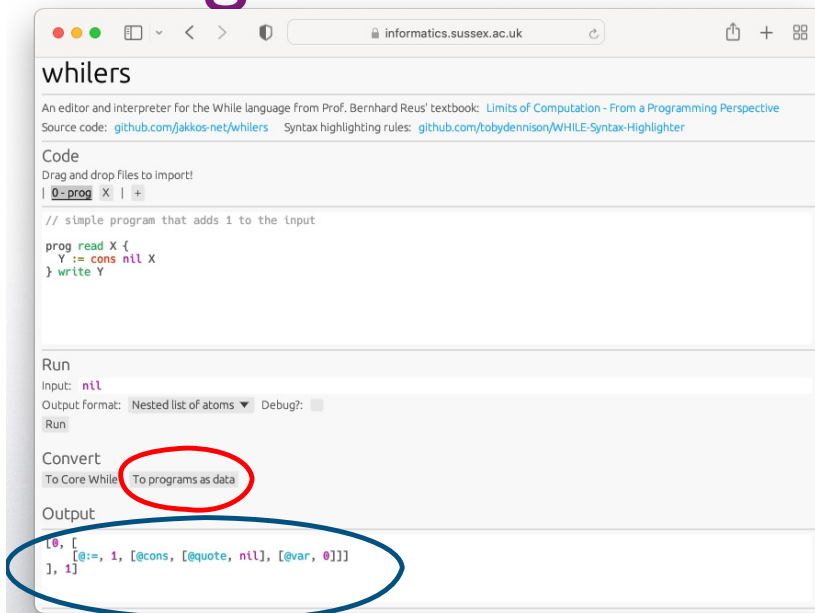
```
./hwhile -h
```

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

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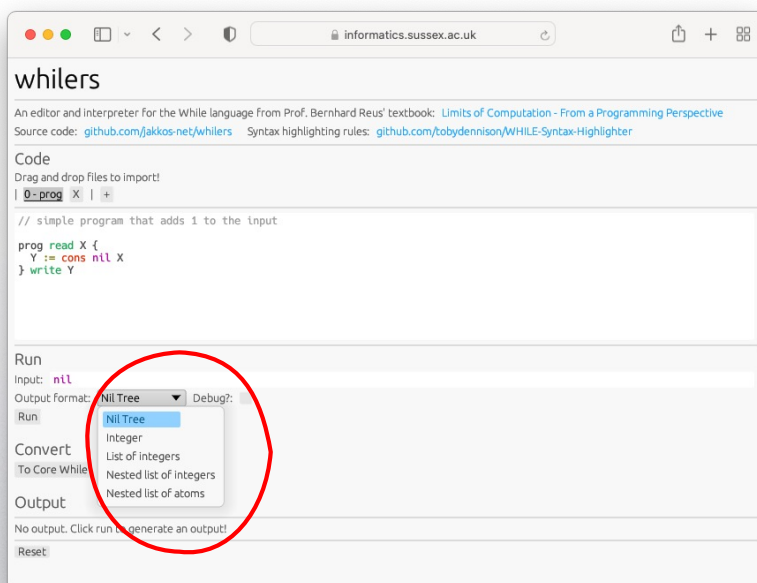-La ?

Programs-as-data in *whilers*

- New this year: *whilers* (see Canvas), a web interface for `WHILE`.

- Button to produce program as data

- View program-as-data in output window

On whilers output

- Output is by default also given as binary tree.

- *Use menu to determine the "type" in which it is presented.*

- *Options:*
  - *Nil tree*
  - *Integer*
  - *List of Integers*
  - *Nested list of integers*
  - *Nested list of atoms*

hwhile / whilers
Demo
(if there is time)

END

Next time:
A special interpreter