Lecture 26 Review

PLAN

- OCaml syntax
  - symbols, expressions, terms, types, values
- OCaml computation rules
  - $t \downarrow t'$, rules for reduction, substitution model
  - can we have eval t n ?
  - eager and lazy evaluation
- state and mutable variables
  - while bexp do body done
  - state as a record
  - tail recursion
- OCaml typing rules
  - type checking is decidable
- OCaml SL types
  - type checking is undecidable
- Research question example
  - which recursive types make sense for OCaml SL ?
- Recursion and Induction
- Real numbers and Modules
- If time, another Research Question.

OCaml Syntax

symbols $\Sigma = \{ a, b, c, \ldots, 0, 1, 2, \ldots |, ;, \rightarrow, \ldots \}$

expressions – recognized by the parser

terms – expressions that have types

true, 1+1, 2*3, 2.*3.

Is type a value ?

types – expressions recognized by types, also defined type atype = ...

– polymorphic type expressions

$\alpha * \beta$, $\alpha \rightarrow \beta$, Id_1 of α | Id_2 of β

– in OCaml SL these polymorphic expressions give rise to polymorphic logic

$\alpha * \beta \rightarrow \alpha$ $\quad$ $\alpha \rightarrow \alpha \mid \beta$

$x : \alpha \rightarrow \beta(x)$

OCaml Computation Rules

$t \downarrow t'$ is our notation for evaluation (reduction rules)

e.g. $bexp_1 \downarrow true \vdash bexp_1 \,||\, bexp_2 \downarrow true$

Note how if bexp then exp_1 else exp_2 evaluates, it is lazy! What does this mean?

Function evaluation $f\ a$

What is the standard order?

What is lazy evaluation?

fix vs efix

$$fix\ f \downarrow f\,(fix\ f)$$

$$efix\ f\ x \downarrow f\,(efix\ f)\ x$$

It would be nice to have

eval t n that evaluates t for n steps

Can this be eager in t?

Recursive functions can be defined by fix and efix.

What is tail recursion?

Consider the <u>while</u> bexp <u>do</u> exp <u>od</u>

(in OCaml <u>while</u> bexp <u>do</u> exp <u>done</u>)

OCaml Computation Rules continued

while bexp do exp done is used for mutable variables, but we can also think of state as a record

$\{x_1 : t_1 ; x_2 ; t_2 ; \cdots ; x_n : t_n\}$

We can "simulate" mutable variables by operations on state that are functional.

This leads to state monads that we did not cover. It's how Haskell deals with state.

Exercise: write the square root example from Lecture 19 using the while loop as a recursive function on a functional record.

Specify the square root problem with a dependent type.

OCaml Typing Rules

We provided some explicit typing rules in Lecture 2

$$exp_1 : int,\ exp_2 : int \vdash exp_1 + exp_2 \in int$$

$$f : \alpha \to \beta,\ a : \alpha \vdash f\ a \in \beta$$

Exercise Give typing rules for fix and efix.

Note that recursive types are limited

type $\alpha = \alpha \to \alpha$ cyclic!

type $\alpha = \alpha * int$ cyclic

But OCaml allows

type α = Left of int | Right of $\alpha \to int$

This type "does not make sense" mathematically.

Research Question What OCaml SL recursive types make sense?

We investigate this using the subtype relation,

$$\alpha \sqsubseteq \beta$$

that needs to be defined for a full OCaml SL.

Recursion and Induction

We have studied in detail the idea that recursion and induction are closely related.

recursion is terminating induction.

If we use dependent types we can say that induction is specified by a dependent type inhabited (realized) by a recursive function – see Lecture 18, page 6

```
let rec list_ind (l:α list) (base : P[ ])
(step: u:α → v: α list → w: P v → P u::v)
: P l  =
match l with [ ] → base

| h::t → step h t list_ind t base step
```

We noted that this is fold_right.

Note, we can type P as α list → type.

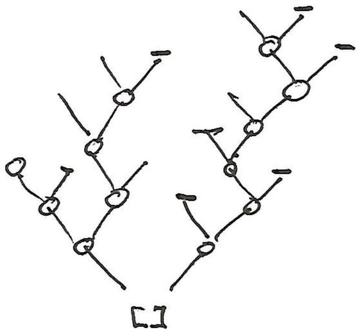
Real Numbers

If we define $e = \sum_{i=1}^{\infty} 1/i!$, this provides a simple basis for writing e as a real number. You should be able to define $e+e$ and $e * e$.

Lecture 26

Research Question   Is the Fan Theorem valid in OCaml SL

Fan specification



$\bar{\alpha}(n) = [\alpha 0, \dots, \alpha n]$

$\alpha : (nat \to bool) \to (n : nat \text{ where } R\ \bar{\alpha}\ n) \to$

$z : (nat * \alpha : (nat \to bool)) \to (x : nat \text{ where } x \le z.1\ \&\ R\ \bar{\alpha}\ x)$

$$\underline{x : nat \text{ where } x \le z.1\ \&\ R\ \bar{\alpha}\ x}$$

The z is a uniform bound on the height of the tree.