Adjoint Functional Programming

NICHOLAS COLTHARP, ANTON LORENZEN, WESLEY NUZZO, and XIAOTIAN ZHOU

These are the lecture notes for Frank Pfenning's course at OPLSS 2024.

1 LECTURE 1: LINEAR FUNCTIONAL PROGRAMMING

Linear logic: 1987 or earlier. Relevance has been known for a while, but implementation takes a while. But see, eg, Rust: things are changing.

1.1 SNAX

In this course, we use the SNAX programming language (its name derives from proof theory). It has the following features:

- Substructural programming (linear + non-linear)
- Inference
- Overloading (between linear/non-linear versions)
- Intermediate language also based in proof theory: it is a proof-theoretic compiler
- pretty decent performance comparable to MLTon

1.2 Types

$$\frac{}{() : 1} \qquad \frac{}{() \text{ value}}$$

$$\frac{e : A}{\mathsf{inl}\ e : A + B} \qquad \frac{v \text{ value}}{\mathsf{inl}\ v \text{ value}}$$

$$\frac{e : B}{\mathsf{inr}\ e : A + B} \qquad \frac{v \text{ value}}{\mathsf{inr}\ v \text{ value}}$$

And we define

$2 = 1 + 1$
$\mathsf{true} = \mathsf{inl}\ ()$
$\mathsf{false} = \mathsf{inr}\ ()$

1.2.1 Sum Types. But in fact, we'll use *labels*: $+\{l : A_l\}$ for $L \neq \emptyset$, finite (L has to be non-empty due to implementation issues).

$A + B = +\{\mathsf{inl} : A, \mathsf{inr} : B\}$

$\mathsf{bool} = +\{\mathsf{true} : 1, \mathsf{false} : 1\}$
$\mathsf{true}\ () : \mathsf{bool}$
$\mathsf{false}\ () : \mathsf{bool}$

1.2.2 Equireqursive Types. We can form the natural numbers using an equirecursive (*not* isorecursive) approach.

$\mathsf{nat} = +\{\mathsf{zero} : 1, \mathsf{succ} : \mathsf{nat}\}$
$\lceil 0 \rceil = \mathsf{zero}\ ()$
$\lceil 1 \rceil = \mathsf{succ}\ (\mathsf{zero}\ ())$
$\lceil 2 \rceil = \mathsf{succ}\ (\mathsf{succ}\ (\mathsf{zero}\ ()))$

$$\frac{(\mathsf{K} \in L) \quad e : A_l}{\mathsf{K}(e) : +\{\, l : A_l \,\}_{l \in L}} \qquad \frac{v \text{ value}}{\mathsf{K}(v) \text{ value}}$$

$\mathsf{list} = +\{ \mathsf{nil} : 1,\ \mathsf{cons} : \mathsf{nat} \times \mathsf{list} \}$

1.2.3 Pairs.

$$\frac{e_1 : A \quad e_2 : B}{(e_1,\ e_2) : A \times B} \qquad \frac{v_1 \text{ value} \quad v_2 \text{ value}}{(v_1,\ v_2) \text{ value}}$$

1.2.4 Computation. Barely, we are not introducing function types.

```
not (x : bool)
not x = match x with
  | true a => false a
  | false a => true a
```

Note: It would *not* be legal to use `()` instead of `a` on the right-hand side! That would not be *linear*.

1.2.5 Match construct.

$$\frac{e : +\{\, l : A_l \,\}_{l \in L} \quad x : A_l \vdash e_l : C\ (\forall l \in L)}{\mathsf{match}\ e\ \mathsf{with}\ (l(x) \Rightarrow e_l)_{l \in L} : C}$$

For pairs, we need to use a different approach to avoid re-using variables:

$$\frac{\Gamma \vdash e_1 : A \quad \Delta \vdash e_2 : B}{\Gamma, \Delta \vdash (e_1,\ e_2) : A \times B} \qquad \frac{}{x : A \vdash x : A}$$

We can only apply this rule if `x` is the only variable in the context. Otherwise, there could be unused variables.

Note: variables must be unique in a context.

$$\frac{\Delta \vdash e : A \times B \quad \Gamma,\ x : A,\ y : B \vdash e' : C}{\Delta, \Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ (x, y) \Rightarrow e' : C}$$

How to implement? The obvious solution is to just check the preconditions and make sure that they partition the variables. However, this does not perform well. Better approaches will be covered in the next lecture.

Back to the type rules for other types:

$$\frac{}{\vdash () : 1}$$

We need to use the empty context, since this term consumes no variables.

$$\frac{\Delta \vdash e : +\{\, l : A_l \,\}_{l \in L} \quad \Gamma,\ x : A_l \vdash e_l : C\ (\forall l \in L)}{\Delta, \Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ (l(x) \Rightarrow e_l)_{l \in L} : C}$$

Note that every branch of the match must have the same context Γ. This is just to say that every branch must use the same set of variables.

```
plus (x : nat) (y : nat) : nat
plus x y = match x with
  | zero () => y
  | succ x' => succ (plus x' y)
```

Note that in the first arm, a pattern like `zero u` would not work, since `u` would be unused.