Adjoint Functional Programming

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

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

1 LECTURE 3: ADJOINT TYPES (& NODES)

Since there was an evening lecture on modes in OCaml yesterday, we will switch up the lectures and talk about adjoint types first. They allow us to reason about linearity in SNAX, just like in OCaml. Not yet in SNAX: stack allocation, because we have not found the right logic yet.

- Negation
- Mixing linear & non-linear programming
- Mode checking & inference

1.1 Programming

```
type nat = +{'zero : 1, 'succ : nat}

type list = +{'nil : 1, 'cons : nat * list}

decl map (f : nat -> nat) (xs : list) : list
defn map f xs = match xs with
  | 'nil() => 'nil()
  | 'cons(x, xs) => 'cons(f x, map f xs)
```

This `map` function should not compile: we don't use `f` in the `nil` branch and use it twice in the `cons` branch.

We introduce the following lattice of *modes*:

$$\begin{tikzcd}
 & U \arrow[dl, dash] \arrow[dr, dash] & \\
A \arrow[dr, dash] & & S \arrow[dl, dash] \\
 & L &
\end{tikzcd}$$

where:

- `U` is *unrestricted*.
- `A` is *affine* (used at most once).
- `S` is *strict* (used at least once).
- `L` is *linear* (used exactly once).

The reason our `map` example fails is that SNAX assumes the `L` mode by default. Our `map` example checks if we instruct it to use the `U` mode as default. Before we introduce non-linear values logically, though, we will present this example using iterators:

```
type iterator = &{'next : nat -> nat * iterator,
                  'done : 1}

decl iterate (iter : iterator) (xs : list) : list
defn iterate iter xs = match xs with
  | 'nil() => (match iter.'done with | () => 'nil())
  | 'cons(x, xs) => (match iter.'next(x) with | (y, iter) =>
                       'cons('succ y, iterate iter xs))
```

We would have the same problem with `iterate` if we generalized `'succ` to an arbitrary function `f`.

1.2 Types

To support the *modes* above, we parameterize types by modes:

$$\begin{aligned} A_m,\ B_m := 1\ &|\ A_m \otimes B_m\ |\ +\{\ l : A^l_m\ \}_{l\ \in L}\ |{\downarrow}^k_m\ A_k\ (k\ \geq m) \\ &|\ A_m \multimap B_m\ |\ \&\{\ l : A^l_m\ \}_{l\ \in L}\ |{\uparrow}^m_i\ A_i\ (i\ \leq m) \end{aligned}$$

Remember: the first row is *positive* and the second row is *negative*. In contrast to last lecture, we have now added shift operators to the types to change modes. Shifting down with ${\downarrow}^k_m$ moves us down in the lattice from mode k to mode m and up-shift moves up from mode i to mode m.

Operationally, the downarrow will correspond to a pointer, while the uparrow corresponds to a lazy thunk.

To use modes in our programming example, we can use a mode variable `k` on the type declaration:

```
type nat[k] = +{'zero : 1, 'succ : nat[k]}
```

This allows `nat` to exist at each mode `k`. However, for recursive types like `nat`, we actually need to *guard* the recursion. The reason for this is the runtime layout: SNAX tries to pack datatypes as tightly as possible. This is a problem for recursive types, because fully expanding a recursive type would yield a value of infinite size! We can instruct SNAX to use a pointer indirection instead of inlining the data, by using the `down` operator. Our `nat` type becomes:

```
type nat[k] = +{'zero : 1, 'succ : down[k] nat[k]}
```

This allows us to be explicit about data layout. But we can do even more: the down operator corresponds to the $\downarrow$ type above and allows us to shift the mode of a term. For example, we can use a different mode parameter for the elements of the list:

```
type list[m k] = +{'nil : 1, 'cons : down[k] nat[k] * down[m] list[m k]}
```

Hidden behind this syntax, this imposes an constraint that `m <= k`. This extra constraint is necessary, since you can not have an unrestricted list of linear elements. However, it is super useful to have a `linear` list where the elements are non-linear.

- Q: How is that ensured?
- A: This is part of the typing rules for `down`. The outer `m` in the `list` type gives the mode of the term. The first mode index is special: it is the mode of the whole list.
- Q: Is `down[k]` a pointer or a shift in modes?
- A: Both! The two ends of a pointer might not have the same mode, but they might well have.

Let's fix the `map` example:

```
decl map (f : [mf] up[k] (nat[k] -> nat[k])) (xs : list[m k]) : list[m k]
defn map f xs = match xs with
  | 'nil() => 'nil()
  | 'cons(<x>, <xs>) => 'cons(<f.force x>, <map f xs>)
```

Here, we use $< x >$ to construct a downshifted value and $f.force$ to eliminate an upshifted function. Since the downshift is a positive type, we can pattern-match on it. Unlike in OCaml, we actually have mode polymorphism.

SNAX does not have our lattice of modes built-in, so we define the preorder here:

```
mode U structural :> S A L
mode S strict :> L
mode A affine :> L
mode L linear
```

Then we can instantiate the `map` function with different modes:

```
decl map (f : [U] up[L] (nat[L] -> nat[L])) (xs : list[L L]) : list[L L]
defn map f xs = match xs with
  | 'nil() => 'nil()
  | 'cons(<x>, <xs>) => 'cons(<f.force x>, <map f xs>)
```

This gives a `map` function for linear lists with linear elements using an unrestricted function `f`. Other possible instantiations are:

```
decl map (f : [U] up[L] (nat[L] -> nat[L])) (xs : list[L L]) : list[L L]
decl map (f : [U] up[A] (nat[A] -> nat[A])) (xs : list[A A]) : list[A A]
decl map (f : [U] up[U] (nat[U] -> nat[U])) (xs : list[U U]) : list[L U]
```

The last line is fine, since we construct a new list, which can only be used linearly.

- Q: Does the compiler actually accept the last line? Because it seems to instantiate `m` with both `L` and `U`.
- A: Yes, this is actually not a bug. The first type we gave restricted the output list of the same mode as the input list. That checks. The last type we gave allows the output mode to be different from the input mode. That also checks. The fact that the last is not an instance of the first is perfectly okay, because the function definition is checked separately against each type.

1.3 Relation to Linear Logic

We can define $!A$ as $\downarrow_L^U \uparrow_L^U \ A_L$. Some people also write $(\Box A_T)_T \ = \downarrow_T^U \uparrow_T^U \ A$. This is a *comonad*.

- Q: Why is it better to use up- and down-shifts than use the bang operator $!A$?
- A: Because with the bang operator you basically write a linear program and you always have to deconstruct the bang operator all the time. In SNAX, you can mix linear and non-linear programming.

1.4 Typing rules

For the typing rules, we will have to constrain the context to ensure that all variables bound in it have a mode that permits at least certain operations. We write $\Gamma \ \geq \ m$ to say that all variables in Γ have a mode that is at least m in the preorder. Then we can give a typing rule for the downshift operator, where we ask that all elements in the context have a mode that is at least k:

$$\frac{\Gamma \ \geq k \qquad \Gamma \vdash \ e \ : \ A_k}{\Delta_W, \ \Gamma \vdash \ < e > \ : \ \downarrow_m^k \ A_k}$$

In this rule, Δ_W is a context of variables that can be weakened (eg. are not linear or strict). We can eliminate the downshift operator with the following rule:

$$\frac{\Delta \vdash \ e \ : \ \downarrow_m^k \ A_k \qquad \Delta \ \geq m \ \geq r \qquad \Gamma, \ x \ : \ A_k \vdash \ e' \ : \ C_r}{\Delta, \Gamma \vdash \ \text{match } e \text{ with } \ < x > \Longrightarrow \ e' \ : \ C_r}$$

The full typing rules are given in Jang et al. 2024.

Created with Madoko.net.