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 4: SEMI-AXIOMATIC SEQUENT CALCULUS (SAX)

First, some recap and remarks from last lecture:

$$A_m,\ B_m ::= 1_m \mid A_m \times B_m \mid +\{\, l : A_l \,\}_{l \in L} \mid {\downarrow_m^k}\ A_k\ (k \geq m)$$
$$\mid A_m \to B_m \mid \&\{\, l : A_l \,\}_{l \in L} \mid {\uparrow_i^m}\ A_i\ (i \leq m)$$

1.0.1 Comparison to OCaml. The new work on OCaml has modes not in SNAX like stack allocation. A question last time was whether you can use linear resources to construct something unrestricted:

$f : A_L \to B_L,\ x : A_L \vdash C_U$

You can not, since our judgement is:

$\Gamma \vdash e : A_m$ presupposes $\Gamma \geq m$

Instead, you can construct something inside a `down` constructor. That unlocks the elements from your context that you can to use. Eg. write `match f x with down ...` in this case.

1.0.2 Other Logics. We can model linear logic (with just $U > L$) as: $!A = {\downarrow_L^U}\,{\uparrow_L^U}\ A$. Our calculus generalizes LNL Benton '95.

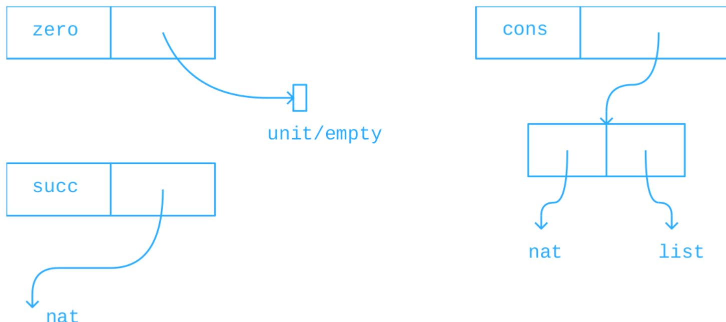
We can also model other logics such as IS_4, where $\mathsf{V} > T$ as: $\Box A = {\downarrow_T^{\mathsf{V}}}\,{\uparrow_T^{\mathsf{V}}}\ A$. This is a *comonad*. You can also do monadic programming by having two modes $T > L$ as: $\circ A = {\uparrow_L^T}\,{\downarrow_L^T}\ A$. This is a (strong) *monad*.

In practice, we only use a fragment of the expressive power. You can also model proof irrelevance; not all modes have to do with linearity.

- *Q:* What is contraction? What is weakening?
- *A:* Contraction is a proof theoretic term for assumptions that can be duplicated. Weakening is a proof theoretic term for assumptions that can be forgotten. In SNAX, `strict` values can not be forgotten, `affine` values can not be duplicated, `linear` values can not be duplicated or forgotten.
- *Q:* If you have non-linear values, do you need a garbage collector?
- *A:* Yes, but this is not yet implemented.

1.1 Explicating Store

In this Section, we want to make the store explicit. Normally, you would use something like separation logic for this. However, we want to give a more higher level view. Our heap layout looks something like this:



For example, a `Cons(1, list)` would be laid out by allocating a `Cons` constructor that points to a pair pointing to `1` and `list`. How do we describe this semantically? Using addresses a, b, we can lay this out in the store as:

$(a, b) : A \times B$

Our rules for the store and the logical rules are respectively:

$$\frac{}{a : A, b : B \vdash (a, b) : A \times B} \qquad \frac{}{A, B \vdash A \times B}$$

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

$$\frac{k \in L}{a : A_k \vdash k(a) : +\{ l : A_l \}_{l \in L}} \qquad \frac{}{A \vdash A \oplus B}$$

$$\frac{}{B \vdash A \oplus B}$$

The shifts will come later, but they turn out not be terribly interesting. We can design one half as axioms, but we need to design the other half as rules. For the elimination rule of pairs, we need to read from memory:

$$\frac{\Gamma, x : A, y : B \vdash P : \gamma}{\Gamma, c : A \times B \vdash \text{read } c \; ((x, y) \Rightarrow P) : \gamma}$$

What does that mean logically?

$$\frac{A, B \vdash C}{A \otimes B \vdash C}$$

This makes sense, even if you don't think about memory. Again, we can compare the store rules to the logical rules:

$$\frac{\Gamma, x : A, y : B \vdash P : \gamma}{\Gamma, c : A \times B \vdash \text{read } c \; ((x, y) \Rightarrow P) : \gamma} \qquad \frac{\Gamma, A, B \vdash C}{\Gamma, A \times B \vdash C}$$

$$\frac{\Gamma \vdash P : \gamma}{\Gamma, c : 1 \vdash \text{read } c \; (() \Rightarrow P) : \gamma} \qquad \frac{\Gamma \vdash C}{\Gamma, 1 \vdash C}$$

$$\frac{\Gamma, x : A_l \vdash P_l : \gamma \quad \forall l \in L}{\Gamma, c : +\{ l : A_l \}_{l \in L} \vdash \text{read } c \; (l(x) \Rightarrow P_l) : \gamma} \qquad \frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \oplus B \vdash C}$$

1.2 SAX

How do we interpret this judgement?

$\Gamma \vdash P : \gamma$

The elements of Γ are addresses and the program P can read from it. There are two rules in sequent calculus: identity and cut. Logically:

$$\frac{\Delta \vdash A \quad \Gamma, A \vdash C}{\Delta,\ \Gamma \vdash C}$$

- Q: Does this become trivially, since we have replaced half the rules by axioms?
- A: No.

What is that computationally?

$$\frac{\Delta \vdash P :: (x : A) \quad \Gamma,\ x : A \vdash Q :: (c : C)}{\Delta,\ \Gamma \vdash \mathsf{cut}_A\ x\ P;\ Q :: C}$$

`P` has to write into `x`. Each program returns a destination that it writes into.

$$\frac{}{A \vdash A}$$

In programming terms, this is a move from `b` to `a`:

$$\frac{}{b : A \vdash \mathsf{id}\ \alpha : A}$$

- Q: What does the double colon mean?
- A: Just syntax since there is also another colon for the address.
- Q: Is the `cut` a let-binding?
- A: Yes, but it writes to the destination instead of returning a value.

If we use destinations, we need to change the pair rule. But how do we write into the destination?

$$\frac{}{\alpha : A,\ b : B \vdash \mathsf{write}\ c\ (\alpha,\ b) :: (c : A \times B)}$$

$$\frac{}{\cdot \vdash \mathsf{write}\ c\ () :: (c : 1)}$$

$$\frac{k \in L}{\alpha : A_k \vdash \mathsf{write}\ c\ (k(\alpha)) :: (c : +\{\ l : A_l\ \}_{l \in L})}$$

The whole language is now explicit, even though logically it is just a simple sequent calculus. Our complete syntax is for programs:

$P := \mathsf{write}\ d\ \mathsf{V}$
$\ \ \mid \mathsf{read}\ d\ \mathsf{K}$
$\ \ \mid \mathsf{id}\ \alpha\ b$
$\ \ \mid \mathsf{cut}\ x\ P;\ Q$

Values:

$\mathsf{V} := () \mid (\alpha,\ b) \mid k(\alpha)$

Continuations:

$\mathsf{K} := () \Rightarrow\ P \mid (x,\ y) \Rightarrow\ P \mid (l(x) \Rightarrow\ P_l)_{l \in L}$

1.3 Compiler

The compiler translates from natural deduction to sequent calculus – this is a proof-theoretic question! We compile:

$\Gamma \vdash e : A$

into an expression that writes into a new destination d:

$\Gamma \vdash [\![e]\!] d :: (d : A)$

Let's say we translate pairs:

$$\begin{aligned}[\![(e_1, e_2)]\!] d \;&= \text{cut } d_1 \; [\![e_1]\!] d_1; \\ &\quad \text{cut } d_2 \; [\![e_2]\!] d_2; \\ &\quad \text{write } d \; (d_1, d_2)\end{aligned}$$

How do I compile a match?

$$\begin{aligned}[\![\; \textit{match } e \textit{ with } ((x, y) \Rightarrow e') \;]\!] d' \;&= \text{cut } d \; [\![e]\!] d \\ &\quad \text{read } d \; ((x, y) \Rightarrow [\![e']\!] d')\end{aligned}$$

How do I compile a variable?

$[\![x]\!] d \; = \; \text{id } d \; x$

- *Q:* What does this buy us?
- *A:* We can very easily compile this program to C.
- *Q:* Can there be superfluous moves?
- *A*: Yes! The compiler has two optimizations for this:

$\text{cut } x \; (\text{id } x \; y); \; Q(x) \; = \; Q(y)$

This is a logical rule: Cut and identity are opposites! And you can also reuse reads that you have read before.

- *Q:* Does linearity make it easier to convert to SSA?
- *A:* Maybe? We leave this to C here, even if it does not know about linearity.
- *Q:* Is `x` in the translation already allocated?
- *A:* Yes, the variables of the source language become the addresses of the target language.
- *Q:* Would you have to change the rules if you want to change the data-layout?
- *A:* Yes, this will be in the next lecture!
- *Q:* Do you want to do cut-elimination?
- *A:* For linear values, this would be free, but for non-linear values it might explode the size of the code.

1.4 In Action

We continue with live coding:

```
type bin[m] = +{'b0 : <bin[m]>, 'b1 : <bin[m]>, 'e : 1}

decl inc (x : bin[m]) : bin[m]
defn inc x = match x with
  | 'b0 <x> => 'b1 <x>
  | 'b1 <x> => 'b0 <inc x>
  | 'e() => 'b1 <'e()>

mode L linear

inst inc (x : bin[L]) : bin[L]
```

Let's look at how this is compiled:

```
proc inc/0($0:bin[L]) (x:bin[L]) =
  read x =>
  | 'b0($1) =>
    read $1 <$2> =>
    cut $3:down[L] bin[L]
        write $3 <$2>
    write $0 'b1($3)
  | 'b1($5) =>
    read $5 <$6> =>
    cut $7:down[L] bin[L]
      cut $8:bin[L]
        call inc/0 $8 $6
      write $7 <$8>
    write $0 'b0($7)
  | 'e($11) => ...
```

But this is without the reuse optimization! With that, we get:

```
proc inc/0($0:bin[L]) (x:bin[L]) =
  read x =>
  | 'b0($1) =>
    read $1 <$2> =>
    cut $3 = $1 : down[L] bin[L] % reuse
      write $3 <$2>
    write $0 'b1($3)
  | 'b1($5) =>
    read $5 <$6> =>
    cut $7 = $5 : down[L] bin[L] % reuse
      cut $8 = x : bin[L] % reuse
        call inc/0 $8 $6
      write $7 <$8>
    write $0 'b0($7)
  | 'e($11) => ...
```

This is the general purpose idea, that in linear logic you can reuse the memory. Nothing fancy here. One of Frank's students proved the correctness of that optimization in their senior thesis.

We want to improve our implementation:

```
type bin[m] = +{'b0 : <bin[m]>, 'b1 : <bin[m]>, 'e : 1}

type std[m] = +{'b0 : <pos[m]>, 'b1 : <std[m]>, 'e : 1}
type pos[m] = +{'b0 : <pos[m]>, 'b1 : <std[m]>        }
```

We can give another type signature to `inc`:

```
decl inc (x : bin[m]) : bin[m]
decl inc (x : std[m]) : pos[m]

defn inc x = match x with
  | 'b0 <x> => 'b1 <x>
  | 'b1 <x> => 'b0 <inc x>
  | 'e()    => 'b1 <'e()>
```

This is super cool: We can give multiple types to a function! We can also implement decrement:

```
decl dec (x : bin[m]) : bin[m]
decl dec (x : pos[m]) : bin[m]
defn dec x = match x with
  | 'b0 <x> => 'b1 <dec x>
  | 'b1 <x> => 'b0 <x>
  | 'e()    => 'e()
```

But we can not give the signature:

```
decl dec (x : std[m]) : std[m]
```

since in the `'b0` case we would have to turn a `std` into a `pos`. Another possible implementation:

```
decl dec (x : std[m]) : std[m]
defn dec x = match x with
  | 'b0 <x> => 'b1 <dec x>
  | 'b1 <'e()> => 'e()
  | 'b1 <'b0 <x>> => 'b0 <'b0 <x>>
  | 'b1 <'b1 <x>> => 'b0 <'b1 <x>>
  | 'e() => 'e()
```

Created with Madoko.net.