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 5: DATA LAYOUT

Recap:

Programs:	Small Values:	Continuations:	Types:
P := write c V	V := $(\alpha,\ b)$	K := $(x,\ y) \Rightarrow\ P$	$A \times B$
| read c K	| ()	| $() \Rightarrow\ P$	1
| cut x P; Q	| $k(\alpha)$	| $(l(x) \Rightarrow\ P_l)_{l \in L}$	$+\{l\colon A_l\}_{l \in L}$
| id α b	| <a>	| <x> $\Rightarrow\ P$	$\downarrow A$
| call f $\overline{a}$			

1.1 Dynamics

In the style of SSOS.

cell α_1 V_1, cell α_2 V_2, ..., proc P_1, proc P_2, ...

In substructural operational semantics, you write down:

proc (cut x $P(x)$; $Q(x)$) $\longrightarrow$ cell α $\square$; proc($P(\alpha)$); proc ($Q(\alpha)$)
cell α $\square$; proc (write α S) $\longrightarrow$ cell α S
cell α $\square$; cell b S; proc (id α b) $\longrightarrow$ cell α S

We do not have to note down the things that stay the same, this is more modular. In the third line, the cell b is de-allocated since we assume that everything is linear here.

- *Q:* Do we allow mutating cells that contain values already?
- *A:* No, we only write to empty cells. We will discuss reuse later.

cell c S, proc (read c S′) $\longrightarrow$ proc (S $\triangleright$ S′)
$(\alpha,\ b) \triangleright ((x,\ y) \Rightarrow\ P(x,\ y)) = P(\alpha,\ b)$
$() \triangleright (() \Rightarrow\ P) = P$
$k(\alpha) \triangleright (l(x) \Rightarrow\ P_l(x))_{l \in L} = P_k(\alpha)$
<a> $\triangleright$(<x> $\Rightarrow\ P(x)) = P(\alpha)$

- *Q:* Why is this substructural?
- *A:* Since you can read the rules in logical form, where , is the linear conjunction and $\longrightarrow$ is a linear function arrow.

Let's consider functions:

$$\frac{\Gamma,\ A \vdash B}{\Gamma \vdash A \multimap B}$$

How would we interpret this in the store rules?

$$\frac{\Gamma,\ x\ :\ A \vdash P\ ::\ (y\ :\ B)}{\Gamma \vdash \text{write } c\ ((x,\ y) \Rightarrow\ P)\ ::\ (c\colon A \multimap B)}$$

The axiom for the usual left-rule of functions is:

$$\frac{}{A,\ A \multimap B \vdash B}$$

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

We pass (α, b) to the continuation in the cell c. To achieve this, we say that a store variable is either a value or a continuation:

$\mathsf{S} \coloneqq \mathsf{V} \mid \mathsf{K}$

We then change:

	Small Values:	Continuations:	Types:
$P \coloneqq \mathsf{write}\ c\ \mathsf{S}$	$\mathsf{V} \coloneqq (\alpha,\ b)$	$\mathsf{K} \coloneqq (x,\ y) \Rightarrow P$	$A \times B,\ A \to B$
$\mid \mathsf{read}\ c\ \mathsf{S}$	$\mid ()$	$\mid () \Rightarrow P$	1
$\mid \mathsf{cut}\ x\ P;\ Q$	$\mid k(\alpha)$	$\mid (l(x) \Rightarrow P_l)_{l \in L}$	$+\{l : A_l\}_{l \in L},\ \&\{l : A_l\}_{l \in L}$
$\mid \mathsf{id}\ \alpha\ b$	$\mid \langle a \rangle$	$\mid \langle x \rangle \Rightarrow P$	$\downarrow A,\ \uparrow A$
$\mid \mathsf{call}\ f\ \overline{a}$			

Why do we not a have counter-part for 1? Because it would be bottom. It happens not to be too important, because it is not inhabited.

1.2 Negatives

Logical rules for the lazy record:

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B} \qquad \frac{}{A \,\&\, B \vdash A} \qquad \frac{}{A \,\&\, B \vdash B}$$

Process rules:

$$\frac{(\forall l \in L) \quad \Gamma \vdash P_l :: (x : A_l)}{\Gamma \vdash \mathsf{write}\ c\ (l(x) \Rightarrow P_l(x)) :: (c : \&\{l : A_l\}_{l \in L})}$$

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

Remember: *Right rules write.*

1.2.1 Reuse. How do we do reuse? One way: when we read from a cell, rather than deleting it, but it on the other side with content $\square$. Then we can reuse the cell for another one of the same type. Using the same types, ensures that the cells have the same size at runtime.

1.2.2 No garbage. In the final configuration, we only have cells and no more processes. There is no garbage, if everything can be reached from the final destination. Non-linear things might still be around and could be unreachable.

Reference counting is not very compatible with parallelism, because there can be contention when several threads access a reference count. For this reason, we will stick with a more traditional garbage collector.

1.3 Cuts and Snips

The data layout we have considered so far is quite pointer-intensive. For example, $A \otimes (B \otimes C)$ would be laid out as two allocations, but in practice it should be laid out as one allocation. But this is hard to express logically.

How can we derive the associativity of $\otimes$? It turns out that we need a cut rule somewhere:

$$\dfrac{\dfrac{\dfrac{A,\ B \vdash A \otimes B \quad A \otimes B,\ C \vdash (A \otimes B) \otimes C}{A,\ B,\ C \vdash (A \otimes B) \otimes C}}{A,\ B \otimes B \vdash (A \otimes B) \otimes C}}{A \otimes (B \otimes C) \vdash (A \otimes B) \otimes C}$$

So we can not have cut-elimination. So how can we recover? Use *snips* instead of cuts. We mark subformulas by an underline:

$$\dfrac{}{\underline{A},\ \underline{B} \vdash A \otimes B} \qquad \dfrac{}{\cdot \vdash 1}$$

$$\dfrac{}{\underline{A} \vdash A \oplus B} \qquad \dfrac{}{\underline{B} \vdash A \oplus B}$$

The snip rule is:

$$\dfrac{\Delta \vdash A \quad \Gamma,\ \underline{A} \vdash C}{\Delta,\ \Gamma \vdash C}$$

Then, instead of allocating:

$$\dfrac{}{\alpha : \underline{A},\ b : \underline{B} \vdash \mathsf{write}\ c\ (\alpha,\ b) :: (c : A \otimes B)}$$

… we can just write the addresses. This performs no computation at runtime:

$$\dfrac{}{c.\pi_1 : A,\ c.\pi_2 : B \vdash \mathsf{write}\ c\ (\_,\ \_) :: (c : A \otimes B)}$$

Snips correspond to address computation and cuts correspond to allocation. When you have a subformula, then you can compute the address of the subformula. For the application rule:

$$\dfrac{}{\underline{A},\ A \to B \vdash \underline{B}} \qquad \dfrac{}{A \,\&\, B \vdash \underline{A}}$$

This determines a calling convention for functions. This happens to be a good way to represent lambdas. This is currently unpublished, but supported in the compiler.

1.4 SNAX backend

Continuing in the file from last lecture.

```
type list[m k] = +{nil: 1, cons: <std[k]> * <list[m k]>}

decl append (xs : list[m k]) (ys : list[m k]) : list[m k]
defn append xs ys = match xs with
     | 'nil => ys
     | 'cons(<x>, <xs> => 'cons(<x>, <append xs ys>)

inst append (xs : list[L U]) (ys : list[L U]) : list[L U]
```

The compiled code is (without reuse):

```
proc append/0 ($0 : list[L U]) (xs : list [L U]) (ys : list[L U]) =
  read xs =>
  | 'nil(_) =>
    read xs.nil () =>
    id:list[L U] $0 ys % : list[L U]
  | 'cons(_) =>
    read xs.cons (_, _) =>
    read xs.cons.pi1 <$1> =>
    read xs.cons.pi2 <$2> =>
    write $0.cons.pi1 <$1>
    cut $4:list[L U]
      call append/0 $4 $2 ys
    write $0.cons.pi2 <$4>
    write $0.cons (_,_)
    write $0 'cons(_)
```

With reuse:

```
proc append/0 ($0 : list[L U]) (xs : list [L U]) (ys : list[L U]) =
  read xs =>
  | 'nil(_) =>
    read xs.nil () =>
    id:list[L U] $0 ys % : list[L U]
  | 'cons(_) =>
    read xs.cons (_, _) =>
    read xs.cons.pi1 <$1> =>
    read xs.cons.pi2 <$2> =>
    write $0.cons.pi1 <$1>
    cut $4 = xs : list[L U] % reuse
      call append/0 $4 $2 ys
    write $0.cons.pi2 <$4>
    write $0.cons (_,_)
    write $0 'cons(_)
```

Notice that we do an unnecessary write before the cut, since the element is already in the right place. In future work, this will be eliminated.