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

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

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

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

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 $L \neq \varnothing$, 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.

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

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

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

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.

2 LECTURE 2: FROM PL TO LOGIC AND BACK (x2?)

We will go back and forth between logic and PL. Logic will inform our PL approach. It is important to be aware of the connection: it is inevitable post-hoc but these features may be confusing to implement without the knowledge.

The rules from last lecture, summarized:

$$\frac{}{x : A \vdash x : A}$$

$$\frac{}{\cdot \vdash () : 1} \qquad \frac{\Delta \vdash e : 1 \quad \Gamma \vdash e' : C}{\Delta, \Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ () \Rightarrow e' : C}$$

$$\frac{\Delta \vdash e_1 : A \quad \Gamma \vdash e_2 : B}{\Delta, \Gamma \vdash (e_1, e_2) : A \times B} \qquad \frac{\Delta \vdash e : A \times B \quad \Gamma, x : A, y : B \vdash e' : C}{\Delta, \Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ (x, y) \Rightarrow e' : C}$$

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

We have to use every variable exactly once.

2.1 Reduction relation

In the lecture, we will see the intuition for the theorems, but not include proofs. You can do them yourself if you want.

Next, we will look at a reduction relation for the language. In the dynamic semantics, we want to show type soundness and also something different: we want to show that there is no garbage at the end of the evaluation.

We can set up a close correspondence between the static rules and the dynamic rules. The proof will be easier if the rules are very close. What should then be our runtime interpretation of a judgement such as:

$\Gamma \vdash e : A$

We interpret the expression e as the program getting evaluated, the type A will not be carried around and Γ will be a variable map. We write the variable map as η, and define a judgement for it as $\eta : \Gamma$ with:

$$\frac{}{(\cdot) : (\cdot)} \qquad \frac{\eta : \Gamma \quad \cdot \vdash v : A}{\eta, x \mapsto v : (\Gamma, x : A)}$$

We say that η is an *environment* and Γ is a *context*.

Under these preconditions, we want to run the program as $\eta \vdash e \hookrightarrow v$. But splitting the context for pairs will create a problem:

$$\frac{? \vdash e_1 \hookrightarrow v_1 \quad ? \vdash e_2 \hookrightarrow v_2}{\eta \vdash (e_1, e_2) \hookrightarrow (v_1, v_2)}$$

How will we split the environment and fill in the "?" in the rule? We can not split η, because then we would always have to traverse e_1 (at runtime!) to see which the variables are to figure out how to do the split.

2.1.1 The subtractive approach. However, we do not actually have to split η: under the assumption that this type checks, we can put η on both sides, since we know this will be well-formed. One way to do this: use the *subtractive* approach. After e_1 is finished, we get back an η_1 of variables that are unused. We then pass η_1 to the evaluation of e_2 and get back an empty environment. But actually, we have to do this everywhere: e_2 returns an environment η_2, which we return from the rule:

$$\frac{\eta \vdash e_1 \hookrightarrow v_1 \setminus \eta_1 \quad \eta_1 \vdash e_2 \hookrightarrow v_2 \setminus \eta_2}{\eta \vdash (e_1,\ e_2) \hookrightarrow (v_1,\ v_2) \setminus \eta_2}$$

This also corresponds to how the type checker might check that variables are only used once.

- Q: Is there are more logical way to do this? It seems like much gets hidden here?
- A: Yes, taking some shortcuts here. In the subtractive approach we would write the type rule as:

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

Can we write the rest of the rules now? Yes, let's look at variables:

$$\frac{}{\Gamma,\ x : A \vdash x : A \setminus \Gamma}$$

2.1.2 The additive approach. However, it is much better to do things *additive*. Subtractive has an issue: It forces left-to-right evaluation, where you have to look at e_1 before you look at e_2.

In the additive approach: We have a context $\Gamma \vdash e : A \setminus \Omega$ where Ω are the variables that are *actually used*. In contrast, in the subtractive approach we return the *remainder*. The pair rule becomes:

$$\frac{\Gamma \vdash e_1 : A \setminus \Omega_1 \quad \Gamma \vdash e_2 : B \setminus \Omega_2}{\Gamma \vdash (e_1,\ e_2) : A \times B \setminus (\Omega_1,\ \Omega_2)}$$

The result $\Omega_1,\ \Omega_2$ is undefined if there is any overlap between Ω_1 and Ω_2 (eg. if they share a variable). Note that in the rule above, Γ can go into both of the preconditions. That is, because we treat Γ purely as a typing context now, while Ω returns the used variables.

We can relate our new judgement to the old one:

- Soundness: If $\Gamma \vdash e : A \setminus \Omega$ then $\Omega \vdash e : A$ and $\Omega \subseteq \Gamma$.
- Completeness: If $\Omega \vdash e : A$ and $\Omega \subseteq \Gamma$, then $\Gamma \vdash e : A \setminus \Omega$.

The corresponding rule in the semantics is:

$$\frac{\eta \vdash e_1 \hookrightarrow v_1 \setminus \omega_1 \quad \eta \vdash e_2 \hookrightarrow v_2 \setminus \omega_2}{\eta \vdash (e_1,\ e_2) \hookrightarrow (v_1,\ v_2) \setminus (\omega_1,\ \omega_2)}$$

If our expression type-checks then ω_1 and ω_2 will have disjoint domains. If we add non-linear variables, these can occur on both sides.

- Q: Where would our semantics get stuck if a program does not type-check?
- A: First off, not every program that does not type-check will get stuck. But if it gets stuck: this can be because the ω_1 and ω_2 might have overlapping domains, where it would get stuck.
- Q: Is Ω an over-approximation of the variables that are used?
- A: No, we want Ω to be *exactly* the variables used in e.
- Q: If we were to set out to try and prove this, would we need two different versions of the typing rules?

- A: Yes, you would show that for each derivation of one of them, you get a derivation of the other. This is *rule induction.*
- Q: Isn't there an overapproximation in the relation to the old judgement where we write $\Omega \subseteq \Gamma$?
- A: No, since our old judgement is always precise in its typing context.
- Q: Why can Ω and Γ not be the same?
- A: Induction would fail in the pair rule, since even if $\Gamma = \Omega_1, \Omega_2$, then $\Gamma \neq \Omega_1$ or $\Gamma \neq \Omega_2$.

2.1.3 Soundness of additive approach. What do we want the program to satisfy? We have to change our soundness theorem:

Theorem 1. (*Soundness (1)*)
If $\Gamma \vdash e : A \setminus \Omega$ and $\eta : \Gamma$ and $\omega : \Omega$ then $\eta \vdash e \hookrightarrow v \setminus \omega$ (and $v : A$).

- Q: Are the v and the ω existentially quantified in this statement?
- A: Yes, great question! We do not know that e evaluates to v, since that would imply termination. We want to additionally quantify over the v and ω:

Theorem 2. (*Soundness (2)*)
If $\Gamma \vdash e : A \setminus \Omega$ and $\eta : \Gamma$ and $\eta \vdash e \hookrightarrow v \setminus \omega$, then $\omega : \Omega$ (and $v : A$).

Since we defined them in the same way, we can now relate them in the same way. We can prove this theorem with this kind of dynamics.

How do we know that in the end there is no garbage? We write $\eta \vdash e \hookrightarrow v \setminus \eta$ at the toplevel so that everything in η is actually used. We can prove this for the new dynamics. This gives us both soundness and that there will be no garbage in the end.

- Q: Since we do not have recursion in this language, we can always assume that terms terminate, right?
- A: Yes, that is true, but then we can not prove that using induction since termination is a stronger property.
- Q: Don't we use the evaluation as a precondition in the second soundness theorem and thus can not catch stuckness?
- A: Yes, this is no longer type soundness. We will use a different approach next lecture.
- Q: Can you explain what $\eta : \Gamma$ means?
- A: η is a map from variables to values. If the variable has type A in Γ, then the value has type A in η.
- Q: Will you show a small-step semantics?
- A: Not for this language, but next lecture.
- Q: How does the merge of Ω_1, Ω_2 handle top-level variables?
- A: We treat them like non-linear variables.

2.1.4 Affine types. We can play a small game: how can we make this system *affine* so that variable are used at most once? What happens to the merge operator?

At the top-level we have to check for $\Gamma \vdash e : A \setminus \Omega$ that $\Gamma = \Omega$ in the linear version to ensure that everything in Γ is used exactly once. In an affine setting, we can allow $\Omega \subseteq \Gamma$.

- Q: It seems like the typing tree is equal to the evaluation tree?
- A: Yes, since we have not looked at interesting rules. Inviting comments: is the derivation tree the same as the evaluation tree? Student: For matches there is a difference, since we pick a branch of each match in the evaluation tree.

2.1.5 Further typing rules.

$$\frac{}{\cdot \vdash () \hookrightarrow () \setminus \cdot}$$

$$\frac{\eta \vdash e \hookrightarrow (v_1,\ v_2) \setminus \omega \quad \eta,\ x \mapsto v_1,\ y \mapsto v_2 \vdash e' \hookrightarrow v' \setminus (\omega',\ x \mapsto v_1,\ y \mapsto v_2)}{\eta \vdash \mathsf{match}\ e\ \mathsf{with}\ (x,\ y) \Rightarrow\ e' \hookrightarrow v' \setminus (\omega,\ \omega')}$$

2.1.6 Top-level definitions. Not much is happening in the computation. A purely linear type system, does not allow you to write many interesting programs: we need top-level definitions. However, studying the whole language is very complicated, so we study the simple case here.

- Q: What is a top-level definition?
- A: For example:

```
plus (x : nat) (y : nat) : nat
plus x y = ...
```

These top-level definitions also have something important to tell us logically. I will tell you at the end of the lecture.

2.2 Back to Logic

We write a natural deduction system, where $\Gamma \vdash A$ says that the assumptions in Γ can prove A. $\Delta,\ \Gamma\ := \cdot\ \mid \Gamma,\ A$. However, each assumption needs to be used exactly once, giving us linear logic.

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

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B} \qquad \frac{\Delta \vdash A \otimes B \quad \Gamma,\ A,\ B \vdash C}{\Delta, \Gamma \vdash C}$$

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

Historically, this was the first formulation of linear logic. From it, people later developed linear type systems. Our operators so far are:

$A,\ B\ := 1\ \mid A \otimes B \mid A \oplus B$

2.2.1 The operators of linear logic. In linear logic, there is one more operator: “of course A”, written $!A$. This allows us to reuse assumptions. We can model an ordinary function $A \Rightarrow\ B$, as $!A \multimap B$ (we will introduce this formally later). Then we have judgements of the form $\Sigma;\ \Gamma \vdash\ e\ :\ A$, where Σ has reused hypothesis and Γ is linear. The full syntax is:

$$\begin{aligned} A,\ B := &\ 1\ \mid A \otimes B \mid A \oplus B \\ &\mid A \multimap B \mid A \mathbin{\&} B \\ &\mid !A \end{aligned}$$

The first row is *positive* and the second row is *negative*. The first row can be duplicated by copying, eg. $1\ \oplus 1 \vdash\ (1\ \oplus 1)\ \otimes (1\ \oplus 1)$, where we duplicate the boolean $1\ \oplus 1$.

- Q: How do you prove this using the logic?
- A: Use the sum-elimination rule, where $1\ \oplus 1 \vdash\ 1\ \oplus 1$. Then we have to show that $1 \vdash\ (1 \oplus)\ \otimes (1\ \oplus 1)$. We use unit elimination so that we have to show $\cdot \vdash\ (1\ \oplus 1)\ \otimes (1\ \oplus 1)$. Then we use the introduction rules to obtain the term.

2.2.2 Linear Functions. Lastly, we will show the rules for $A \multimap B$ and $A \mathbin{\&} B$.

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

We will not be able to prove $A \multimap (B \multimap A)$ in general, since B is not used in the result. This is different from the rules we have seen so far, since we just plug the terms together without modifying the contexts.

How do we model this in our linear programming language? We use just one arrow $\rightarrow$ and we distinguish regular from linear functions using the arguments.

$$\frac{\Gamma,\ x : A \vdash e : B}{\Gamma \vdash \lambda x.e : A \rightarrow B} \qquad \frac{\Delta \vdash e_1 : A \multimap B \quad \Gamma \vdash e_2 : A}{\Delta, \Gamma \vdash e_1\ e_2 : B}$$

You can not pattern-match against a lambda expression. This is a fundamental distinction between the positive and the negative types.

2.2.3 Lazy pairs. You can not match on take a function and match on it. Instead you can only apply it and see what happens. Based on this, what do you think the elements of the $A \,\&\, B$ type should be? Let's look at the logical rule:

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

Oh, we duplicate Γ on both sides! How can this be sound?

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

We can only extract one of them! This is similar to the match-construct for sums. In the programming language:

$$\frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : B}{\Gamma \vdash (e_1,\ e_2) : A \,\&\, B} \qquad \frac{\Gamma \vdash e : A \,\&\, B}{\Gamma \vdash e.\pi_1 : A} \qquad \frac{\Gamma \vdash e : A \,\&\, B}{\Gamma \vdash e.\pi_2 : B}$$

This is a *lazy pair*: we do not evaluate the components when constructing the pair. We can only evaluate one of them when we deconstruct the pair. We can generalize this to $\&\{\ l : A_l\ \}_{l\ \in L}$.

$$\frac{\Gamma \vdash e_l : A_l\ (\forall l\ \in L)}{\Gamma \vdash (l\ =\ e_l) : \&\{\ l : A_l\ \}_{l\ \in L}} \qquad \frac{\Gamma \vdash e : \&\{\ l : A_l\ \}_{l\ \in L}}{\Gamma \vdash e.k : A_k}$$

We will use the lazy records for object oriented programming.

The main takeaway: In linear logic we have positive and negative types. We can deconstruct the positive types. We can not actually deconstruct the negative types.

3 LECTURE 3: ADJOINT TYPES (& MODES)

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

3.1 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`.

3.2 Types

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

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

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 ${\downarrow}_m^k$ 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.

3.3 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.

3.4 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:

$$\frac{\Gamma \geq k \quad \Gamma \vdash e : A_k}{\Delta_W, \Gamma \vdash \ <e> \ : \ \downarrow^k_m \ A_k}$$

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:

$$\frac{\Delta \vdash e : \downarrow^k_m \ A_k \quad \Delta \geq m \geq r \quad \Gamma, x : A_k \vdash e' : C_r}{\Delta, \Gamma \vdash \text{match } e \text{ with } \ <x> \Rightarrow \ e' : C_r}$$

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

4 LECTURE 4: SEMI-AXIOMATIC SEQUENT CALCULUS (SAX)

First, some recap and remarks from last lecture:

$$\begin{aligned} A_m, B_m ::= \ & 1_m \mid A_m \times B_m \mid +\{ l : A_l \}_{l \in L} \mid \downarrow^k_m \ A_k \ (k \geq m) \\ & \mid A_m \rightarrow B_m \mid \&\{ l : A_l \}_{l \in L} \mid \uparrow^m_i \ A_i \ (i \leq m) \end{aligned}$$

4.0.1 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:

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

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.

4.0.2 Other Logics. We can model linear logic (with just $U > L$) as: $!A = \downarrow^U_L \uparrow^U_L \ A$. Our calculus generalizes LNL Benton '95.

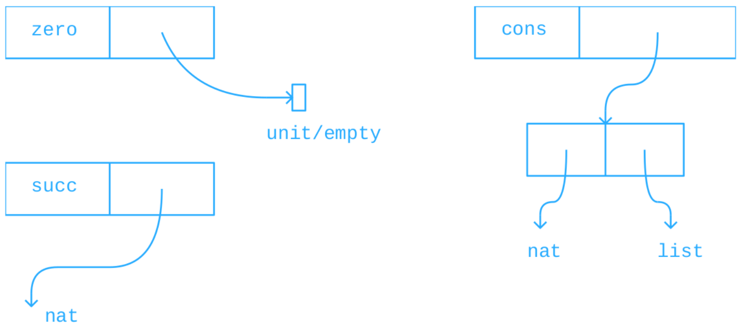
We can also model other logics such as IS_4, where $\mathsf{V} > T$ as: $\Box A = \downarrow^{\mathsf{V}}_T \uparrow^{\mathsf{V}}_T \ A$. This is a *comonad*. You can also do monadic programming by having two modes $T > L$ as: $\circ A = \uparrow^T_L \downarrow^T_L \ A$. This is a (strong) *monad*.

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.

4.1 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 α, b, we can lay this out in the store as:

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

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

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

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

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

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

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:

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

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

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

4.2 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`:

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

- *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?

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

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

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

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

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

Values:

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

Continuations:

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

4.3 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:

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

How do I compile a match?

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

How do I compile a variable?

$[\![x]\!]d\ =\ \mathsf{id}\ d\ x$

- *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:

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

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.

4.4 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()
```

5 LECTURE 5: DATA LAYOUT

Recap:

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

5.1 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.

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

- *Q:* Why is this substructural?
- *A:* Since you can read the rules in logical form, where , is the linear conjunction and → 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?

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

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

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

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

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:

$\text{S} := \text{V} \mid \text{K}$

We then change:

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

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.

5.2 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:

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

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

Remember: *Right rules write.*

5.2.1 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 □. 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.

5.2.2 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.

5.3 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:

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

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

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

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

The snip rule is:

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

Then, instead of allocating:

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

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

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

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:

$$\frac{}{\underline{A},\ A \longrightarrow B \vdash \underline{B}} \qquad\qquad \frac{}{A \mathbin{\&} B \vdash \underline{A}}$$

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.

5.4 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.