Lecture Notes on Data Layout

15-417/817: HOT Compilation
Frank Pfenning

Lecture 15
March 13, 2025

1 Introduction

In today's lecture we discover that the Sax language we have been working with gives us quite direct access to understanding issues of data layout. It is still more abstract, say, then data layout specification in a typical *application binary interface* and that is important, because we don't want to specialize our compiler to a particular architecture at this point. Our approach is to say that abstract data layout is characterized by the ability to compute some addresses from others. For example, if we have the address $c : A \otimes B$, we should have a way to denote the address of A (which will end up being $c.\pi_1$) and also the address of B (which will be $c.\pi_2$). This is actually quite related to the level of abstraction in the specification of the C programming language. It is sufficient, for example, to distinguish between a flat layout of a pair, and the layout of a pair using two addresses. The significance of these choices has been investigated by Morrisett [1995] but not captured type-theoretically. We call the resulting language Snax as an amalgam of Sax and Snip (the latter being the characteristic new rules in our language).

We stumbled upon issues of address computation by investigating the metatheory of Sax [DeYoung et al., 2020], trying to recover a theorem that is analogous to cut elimination in sequent calculi [Gentzen, 1935] and normalization in natural deduction [Prawitz, 1965]. This path is mapped out by DeYoung and Pfenning [2022]. Here, we take a different approach, via *bidirectional typechecking for Sax* [Somayyajula and Pfenning, 2023, Somayyajula, 2024]. These lecture notes represent some recent joint work with Joanna Boyland.

2 Bidirectional Typechecking, More Abstractly

Bidirectional typechecking for terms based on natural deduction can be seen as originating from various sources. For one, it codifies some form of *flow of information* throughout the term, so the term itself does not need to carry types [Dunfield and Krishnaswami, 2022]. Another is that it identifies *verifications*, that is, proofs that only proceed by decomposing a given proposition without having to introduce new ones [Dummett, 1991, Martin-Löf, 1983]. This is a little stronger than simply the subformula property, because we could stick to subformulas without arriving at them purely by decomposition. Verifications also have the property that they never include the introduction of a connective followed by its elimination; often we say they are *normal*. From a programming language perspective this means we never have a *destructor* applied to a *constructor* for a type, that is, no computation can take place.

We can rescue the programming language perspective in more than one way. In ND, we have metavariables that stand for program expressions which are given explicit types at the top-level. The definitions of these metavariables can also be mutually recursive. A less drastic approach is the introduce a type-annotated expression $(e : A)$. One downside of the latter approach is that it makes it difficult to give multiple types to the same expression, which is very useful in the case of adjoint types where the same expression may have multiple modes.

Gentzen's sequent calculus represents an almost extremely point with respect to this discussion, because *bidirectional checking* for the sequent calculus is actually *unidirectional*. That's because all rules except cut decompose propositions when read from the conclusion to the premises. Cut then, is the sequent calculus's "way out" to allow for computation.

But what about the semi-axiomatic sequent calculus? Can we define a "bidirectional" system for Sax that (a) satisfies the idea behind verifications that a proof only decomposes the given proposition (and therefore also has the subformula property), and (b) implies a simple algorithm for typechecking for Sax programs without any additional type information (except as given by typing of metavariables)?

The answer is "yes"! We discuss in 5 what this has to do with data layout.

3 Bidirectional Typing for Sax

In natural deduction, we could write

$$x_1 \Rightarrow A_1, \ldots, x_n \Rightarrow A_n \vdash e \Leftarrow A$$
$$x_1 \Rightarrow A_1, \ldots, x_n \Rightarrow A_n \vdash e \Rightarrow A$$

That is, all variables synthesize their types, and expressions e could either check against a given type A or synthesize a type A. In the sequent calculus, we just have a single judgment form

$$x_1 \Rightarrow A_1, \ldots, x_n \Rightarrow A_n \vdash M \Leftarrow A$$

because we always know the type of the succedent and each antecedent. Here we wrote M for a hypothetical term assignment to the sequent calculus.

It turns out, in Sax each antecedent could check or synthesize, and so could the conclusion. That is, a Sax sequent has the form $\Gamma \vdash P :: \delta$ where

$$\begin{array}{llll} \text{Antecedents} & \Gamma & ::= & x \Rightarrow A \mid x \Leftarrow A \mid \Gamma_1, \Gamma_2 \mid \cdot \\ \text{Succedent} & \delta & ::= & x \Leftarrow A \mid x \Rightarrow A \end{array}$$

One might think of it as *quadridirectional* because we can separately have two directions on each side of the turnstile.

Let's now analyze the direction of decomposition for each of the rules for the positive connectives. We do this for the linear calculus because it is the simplest, but the analysis carries over to nonlinear and even adjoint deduction.

Tensor $A \otimes B$. The right rule $\otimes R$ from the sequent calculus is replaced by an axiom. The direction of decomposition says that if we know the succedent then we can deduce both premises.

$$\frac{}{a : A, b : B \vdash \mathbf{write}\ c\ (a, b) :: (c : A \otimes B)}\ \otimes X \qquad \frac{}{a \Leftarrow A, b \Leftarrow B \vdash \mathbf{write}\ c\ (a, b) :: (c \Leftarrow A \otimes B)}\ \otimes X$$

We could also argue that if $a \Rightarrow A$ and $b \Rightarrow B$ we can synthesize $c \Rightarrow A \otimes B$. However, this is not in the direction of *decomposition of a connective*, so it doesn't satisfy one of our design criteria.

The left rule of the sequent calculus remain a left rule.

$$\frac{\Gamma, x : A, y : B \vdash P :: \delta}{\Gamma, c : A \otimes B \vdash \mathbf{read}\ c\ ((x, y) \Rightarrow P) :: \delta}\ \otimes L$$

$$\frac{\Gamma, x \Rightarrow A, y \Rightarrow B \vdash P :: \delta}{\Gamma, c \Rightarrow A \otimes B \vdash \mathbf{read}\ c\ ((x, y) \Rightarrow P) :: \delta}\ \otimes L$$

If we are given the type of C in the conclusion, then the types of the components in the premis are also known. We may have to think a little about whether δ in the succedent should be either of the two rules or possibly just $d \Leftarrow D$.

We summarize the rules, omitting the proof terms to focus in on the essential aspects of the rules.

$$\frac{}{a \Leftarrow A, b \Leftarrow B \vdash c \Leftarrow A \otimes B}\ \otimes X \qquad \frac{\Gamma, x \Rightarrow A, y \Rightarrow B \vdash \delta}{\Gamma, c \Rightarrow A \otimes B \vdash \delta}\ \otimes L$$

We can almost do an example already, such as $c : A \otimes B \vdash d : B \otimes A$, but we lack the identity rule. So that is next.

Identity. The usual rule

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

admits three different specializations:

$$\frac{}{b \Rightarrow A \vdash a \Rightarrow A}\ \mathsf{id}{\Rightarrow}{\Rightarrow} \qquad \frac{}{b \Leftarrow A \vdash a \Leftarrow A}\ \mathsf{id}{\Leftarrow}{\Leftarrow} \qquad \frac{B = A}{b \Rightarrow B \vdash a \Leftarrow A}\ \mathsf{id}{\Rightarrow}{\Leftarrow}$$

In the first two, we propagate information from left to right or from right to left. In the third rule, we have to compare two sources of information, made explicit in an equality check explicit. In a system with subtyping, this would become $B \leq A$, as defined in the usual coinductive manner.

A possible fourth rule does not actually make sense, because in

$$\frac{}{b \Leftarrow A \vdash a \Rightarrow A}\ \mathsf{id}{\Leftarrow}{\Rightarrow}??$$

both sides would require us to know A already (which we don't, given the directions).

With the given rules, we think we should be able to prove $c \Rightarrow A \otimes B \vdash d \Leftarrow B \otimes A$, but actually we can't!

$$\frac{\overset{??}{a \Rightarrow A, b \Rightarrow B \vdash d \Leftarrow B \otimes A}}{c \Rightarrow A \otimes B \vdash d \Leftarrow B \otimes A}\ \otimes L$$

The problem is that the directions of a and b don't match the direction of the axiom $\otimes X$. We need one more rule which looks like a version of the cut rule, but does not introduce any new formulas. We therefore call it *snip*.

$$\frac{\Gamma \vdash x \Leftarrow A \quad \Delta, x \Leftarrow A \vdash \delta}{\Gamma\ ;\ \Delta \vdash \delta}\ \mathsf{snip}{\Leftarrow}{\Leftarrow}$$

This rule is acceptable because in the second premise we will be given D and we will be able to compute A (and according to out general conventions, x is fresh). We can then check the first premise. What we don't specify here is how to split the antecedents in the conclusion, but we

already know how to deal with that separately (for example, using the additive approach to resource management).

Note that this rule does not violate either of our two design principles. A will be a subformula of D or some formula in Δ, and, in fact, will be computed by some decomposition.

We can use snip to complete our little proof.

$$\dfrac{\dfrac{\dfrac{B = B}{y \Rightarrow B \vdash u \Leftarrow B}\,\mathsf{id}{\Rightarrow}{\Leftarrow} \quad \dfrac{\dfrac{A = A}{x \Rightarrow A \vdash w \Leftarrow A}\,\mathsf{id}{\Rightarrow}{\Leftarrow} \quad \dfrac{}{u \Leftarrow B, w \Leftarrow A \vdash d \Leftarrow B \otimes A}\,\otimes X}{x \Rightarrow A, u \Leftarrow B \vdash d \Leftarrow B \otimes A}\,\mathsf{snip}{\Leftarrow}{\Leftarrow}}{x \Rightarrow A, y \Rightarrow B \vdash d \Leftarrow B \otimes A}\,\mathsf{snip}{\Leftarrow}{\Leftarrow}}{c \Rightarrow A \otimes B \vdash d \Leftarrow B \otimes A}\,\otimes L$$

Written as a proof term, we would have:

```
proc swap (d : B * A) (c : A * B) =
read c (x, y)
snip u
  id u y
snip w
  id w x
write d (u, w)
```

Here, we have left out the directional information from **snip** and **id** because it is visible above and could easily be inferred.

If we didn't care about the direction of information flow, we could think of a snip as a cut (which is is, dynamically), and apply the cut/id optimization twice.

```
proc swap (d : B * A) (c : A * B) =
read c (x, y)
write d (y, x)
```

More Snips, and Cut. As for the identity, there are two more snips, and there is the real cut (which requires a type annotation).

$$\dfrac{\Gamma \vdash x \Rightarrow A \quad \Delta, x \Rightarrow A \vdash \delta}{\Gamma\,;\,\Delta \vdash \delta}\,\mathsf{snip}{\Rightarrow}{\Rightarrow}$$

$$\dfrac{\Gamma \vdash x \Rightarrow A \quad A = B \quad \Delta, x \Leftarrow B \vdash \delta}{\Gamma\,;\,\Delta \vdash \delta}\,\mathsf{snip}{\Rightarrow}{\Leftarrow}$$

In the last rule, $\mathsf{snip}{\Rightarrow}{\Leftarrow}$ we can replace $A = B$ with $A \leq B$ if we support subtyping.

The last rule is not a snip, but a proper cut in the sense the that cut formula A must be present in the syntax.

$$\dfrac{\Gamma \vdash x \Leftarrow A \quad \Delta, x \Rightarrow A \vdash \delta}{\Gamma\,;\,\Delta \vdash \delta}\,\mathsf{cut}_A$$

Sums $\oplus\{\ell : A_\ell\}_{\ell \in L}$. For the axiom, as for $A \otimes B$, the information flows from the succedent to the antecedent.

$$\dfrac{(k \in L)}{a \Leftarrow A_k \vdash c \Leftarrow \oplus\{\ell : A_\ell\}_{\ell \in L}}\,\oplus X$$

The left rule is a little trickier.

$$\frac{(\Gamma, x_\ell \Rightarrow A_\ell \vdash \delta) \quad (\forall \ell \in L)}{\Gamma, c \Rightarrow \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \delta}\ \oplus L??$$

The difficulty here is that if $\delta = (d \Rightarrow D)$ then each branch has to synthesize exactly the same type D. In the presence of subtyping, this doesn't make sense: we'd have to merge all the D_ℓ from all the branches. Also, if we think back to the sequent calculus, the succedent is always $d \Leftarrow D$, so it would be elegant if we can keep the same rule. And, indeed, we can!

$$\frac{(\Gamma, x_\ell \Rightarrow A_\ell \vdash d \Leftarrow D) \quad (\forall \ell \in L)}{\Gamma, c \Rightarrow \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash d \Leftarrow D}\ \oplus L$$

Now the problem from before goes away because we have D in the conclusion and can propagate it to all premises.

If empty sums are allowed, the general rule is even more problematic. Without a premise, we'd have

$$\frac{}{\Gamma, c \Rightarrow \oplus\{\,\} \vdash d \Rightarrow D}\ \mathbf{0}L??$$

and we see that is entirely indeterminate. It therefore would violate the principles of our construction.

For uniformity and simplicity we therefore universally restrict δ when it appears as a variable in the rules to be of the form $d \Leftarrow D$.

Unit 1. The unit doesn't present any interesting new issues.

$$\frac{}{\cdot \vdash c \Leftarrow \mathbf{1}}\ \mathbf{1}X \qquad \frac{\Gamma \vdash \delta}{\Gamma, c \Rightarrow \mathbf{1} \vdash \delta}\ \mathbf{1}L$$

4 Summary of Bidirectional Rules for Sax, Positive Types

The rules are summarized in Figure 1, formulated with subtyping. Not all of these rules appear to be needed. For example, only some of the rules appear when translating from the cut-free sequent calculus or bidirectional natural deductions. In particular, the rule snip$\Rightarrow\Leftarrow$ appears to be redundant because it can be composed from one of the other snips and the identity id$\Rightarrow\Leftarrow$. We therefore exclude if from consideration for now. And, of course, a true cut is not needed when translating from the pure sequent calculus. Looking at the negative connectives in the next lecture might shed more light on these questions. This is subject of ongoing research, and at this point we do not have all the answers.

5 Data Layout Using Subformulas

The key idea behind data layout is that from the address of a compound type like $A \otimes B$ we can compute the addresses of A and B. But that matches exactly the principles underlying our construction of the bidirectional rules: we decompose a type into into its components.

For this purpose we introduce the following syntax for addresses, which were previously entirely abstract. We use variables x for "roots" that are allocated by cuts. Note that we do not need

$$\frac{}{b \Rightarrow A \vdash a \Rightarrow A}\ \mathsf{id}{\Rightarrow}{\Rightarrow} \qquad \frac{}{b \Leftarrow A \vdash a \Leftarrow A}\ \mathsf{id}{\Leftarrow}{\Leftarrow} \qquad \frac{B \le A}{b \Rightarrow B \vdash a \Leftarrow A}\ \mathsf{id}{\Rightarrow}{\Leftarrow}$$

$$\frac{\Gamma \vdash x \Leftarrow A \quad \Delta, x \Leftarrow A \vdash \delta}{\Gamma\,;\,\Delta \vdash \delta}\ \mathsf{snip}{\Leftarrow}{\Leftarrow} \qquad \frac{\Gamma \vdash x \Rightarrow A \quad \Delta, x \Rightarrow A \vdash \delta}{\Gamma\,;\,\Delta \vdash \delta}\ \mathsf{snip}{\Rightarrow}{\Rightarrow}$$

$$\left[\ \frac{\Gamma \vdash x \Rightarrow A \quad A \le B \quad \Delta, x \Leftarrow B \vdash \delta}{\Gamma\,;\,\Delta \vdash \delta}\ \mathsf{snip}{\Rightarrow}{\Leftarrow}\ \right] \qquad \frac{\Gamma \vdash x \Leftarrow A \quad \Delta, x \Rightarrow A \vdash \delta}{\Gamma\,;\,\Delta \vdash \delta}\ \mathsf{cut}_A$$

$$\frac{}{a \Leftarrow A, b \Leftarrow B \vdash c \Leftarrow A \otimes B}\ \otimes X \qquad \frac{\Gamma, x \Rightarrow A, y \Rightarrow B \vdash \delta}{\Gamma, c \Rightarrow A \otimes B \vdash \delta}\ \otimes L$$

$$\frac{(k \in L)}{a \Leftarrow A_k \vdash c \Leftarrow \oplus\{\ell : A_\ell\}_{\ell \in L}}\ \oplus X \qquad \frac{(\Gamma, x_\ell \Rightarrow A_\ell \vdash \delta) \quad (\forall \ell \in L)}{\Gamma, c \Rightarrow \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \delta}\ \oplus L$$

$$\frac{}{\cdot \vdash c \Leftarrow \mathbf{1}}\ \mathbf{1}X \qquad \frac{\Gamma \vdash \delta}{\Gamma, c \Rightarrow \mathbf{1} \vdash \delta}\ \mathbf{1}L$$

Figure 1: Bidirectional Rules for Sax, δ is $d \Leftarrow D$

any projections for type $\mathbf{1}$, because the unit $(\,)$ has no components.

$$\begin{array}{lrcll} \text{Addresses} & a & ::= & x & \text{(root)} \\ & & \mid & a.\pi_1 \mid a.\pi_2 & (A \otimes B) \\ & & \mid & a.\overline{k} & (\oplus\{\ell : A_\ell\}_{\ell \in L} \end{array}$$

Pairs $A \otimes B$. We just follow the direction in which information is propagated.

$$\frac{}{c.\pi_1 \Leftarrow A, c.\pi_2 \Leftarrow B \vdash c \Leftarrow A \otimes B}\ \otimes X \qquad \frac{\Gamma, c.\pi_1 \Rightarrow A, c.\pi_2 \Rightarrow B \vdash \delta}{\Gamma, c \Rightarrow A \otimes B \vdash \delta}\ \otimes L$$

When we think about the expressions, something unexpected comes up. Consider the rule $\otimes X$. This rule doesn't actually *write* to c as before, when there were addresses a and b to be written. Here, it just *calculates addresses*! A similar observation holds for $\otimes L$: instead of reading from a memory cell, it just calculates the addresses of the components of the pair.

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

$$\frac{\Gamma, c.\pi_1 \Rightarrow A, c.\pi_2 \Rightarrow B \vdash P :: \delta}{\Gamma, c \Rightarrow A \otimes B \vdash \mathbf{read}\ c\ ((\_,\_) \Rightarrow P) :: \delta}\ \otimes L$$

Whether the reads and writes at type $A \otimes B$ are entirely silent depends on lower-level details of the implementation. For example, if the dynamics is parallel, they may represent a point of

synchronization where the **read** waits on the **write** to occur. If the dynamics is sequential, then the writes would appear before reads and the only effect of the operation is the calculation of component addresses.

Identity. For the third identity rule id⇒⇐ we have to addresses and we have to move data from one place to the other. How exactly this data movement occurs is a matter of a lower level of abstraction, but after it all projections of the address to be written should be defined, as long as all necessary projections of the address to be read are defined.

$$\frac{B \leq A}{b \Rightarrow B \vdash \mathbf{id}\ a\ b :: (a \Leftarrow A)}\ \mathsf{id}{\Rightarrow}{\Leftarrow}$$

On the other hand, the two unidirectional version of identity don't really accomplish anything. Unlike before, no data movement is involved because the data are already in the correct place.

$$\left[\ \frac{}{a \Rightarrow A \vdash a \Rightarrow A}\ \mathsf{id}{\Rightarrow}{\Rightarrow}\ \right] \qquad \left[\ \frac{}{a \Leftarrow A \vdash a \Leftarrow A}\ \mathsf{id}{\Leftarrow}{\Leftarrow}\ \right]$$

Snips. Looking at the rule snip⇐⇐ with proof terms, we see that it passes on the address computed in the second premise to the first premise. Unlike the cut, this rule does **not** allocate a new cell: the address a already preexists because it is some projection of d (if $\delta = (d \Leftarrow D)$). With negative types (see next lecture) it would also be a projection of some address in Δ.

$$\frac{\Gamma \vdash P :: a \Leftarrow A \quad \Delta, a \Leftarrow A \vdash Q :: \delta}{\Gamma\ ;\ \Delta \vdash \mathbf{snip}\ a\ P\ Q :: \delta}\ \mathsf{snip}{\Leftarrow}{\Leftarrow}$$

The snip⇒⇒ rule works symmetrically, transferring an address computed in the first premise to the second

$$\frac{\Gamma \vdash P :: a \Rightarrow A \quad \Delta, a \Rightarrow A \vdash A :: \delta}{\Gamma\ ;\ \Delta \vdash \mathbf{snip}\ a\ P\ Q :: \delta}\ \mathsf{snip}{\Rightarrow}{\Rightarrow}$$

Now we can rephrase the earlier example.

```
proc swap (d : B * A) (c : A * B) =
read c (x, y)
snip u
  id u y
snip w
  id w x
write d (u, w)
```

as

```
proc swap (d : B * A) (c : A * B) =
read c (_, _)
snip d.pi1
  id d.pi1 c.pi2
snip d.pi2
  id d.pi1 c.pi1
write d (_, _)
```

If we assume that address calculation do not need to be explicitly given, and write **snip** $a\ P\ Q$ as $P\ ;\ Q$ we can abbreviate this as

```
proc swap (d : B * A) (c : A * B) =
id d.pi1 c.pi2 ;
id d.pi1 c.pi1
```

which is exactly the intuitive data movement from source c to destination d.

Cut. Cut, as before, just allocates a fresh cell which we call root.

$$\frac{\Gamma \vdash P :: (x \Leftarrow A) \quad \Delta, x \Rightarrow A \vdash Q :: \delta}{\Gamma\,;\,\Delta \vdash \mathbf{cut}\ (x : A)\ P\ Q :: \delta}\ \mathsf{cut}^x$$

In order to remind ourselves of the freshness condition, we write α as a superscript of the rule. One point here: if we think about allocation at a lower level of abstraction, we need to allocate sufficient space for data of type A so that all projections of α (as predetermined by the type) can be written and read.

Sums. Are a little more complex than pairs, because the axiom/rule actually do have to write/read something: a label. When the small value at an address c in Sax is $k(a)$, then in Snax the address of a is $c.\overline{k}$. In concrete terms, we like to think of this as the tag k and the representation of the tagged data to be side by side.

$$\frac{(k \in L)}{a \Leftarrow A_k \vdash c \Leftarrow \oplus\{\ell : A_\ell\}_{\ell \in L}}\ \oplus X \qquad \frac{(\Gamma, x_\ell \Rightarrow A_\ell \vdash \delta) \quad (\forall \ell \in L)}{\Gamma, c \Rightarrow \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \delta}\ \oplus L$$

These become

$$\frac{(k \in L)}{a.\overline{k} \Leftarrow A_k \vdash \mathbf{write}\ c\ k(\_) :: c \Leftarrow \oplus\{\ell : A_\ell\}_{\ell \in L}}\ \oplus X$$

$$\frac{(\Gamma, c.\overline{\ell} \Rightarrow A_\ell \vdash P_\ell :: \delta) \quad (\forall \ell \in L)}{\Gamma, c \Rightarrow \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \mathbf{read}\ c\ \{\ell(\_) \Rightarrow P_\ell\}_{\ell \in L} :: \delta}\ \oplus L$$

6 Recursive Types

Consider the recursive type

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

The difficulty is that if we see **cut** $(x : \mathsf{nat})\ P\ Q$ we do not know how much space to allocate for x. A complex solution might be to allocate some initial space and then reallocate more space if it grows. A simpler solution is to require the recursion to be guarded so that somewhere within a recursive type there is a pointer. But what is a pointer here? Not surprisingly, in Snax it is just an *address*.

In the lecture on adjoint types, we have seen the type $\downarrow^k_m A_k$ which represents a way to include data of type A at mode k in data at mode m. Because the shift is positive, we can pattern match againt values $\langle v \rangle$ and we evaluate e in $\langle e \rangle$. We usurp this mechanism here. Even though A_k is a component of $\downarrow^k_m A_k$ we imagine a representation where it is inhabited by an explicit address, as in $\langle a \rangle$. For recursive types, this is mostly a shift from a mode to itself. For example, if we make modes explicit as with adjoint types

```
type nat[k] = +{'zero : 1, 'succ : <nat[k]>}
type list[m k] = +{'nil : 1, 'cons : <nat[k]> * <list[m k]>}
```

would desugar into

$$\mathsf{nat}_k = \oplus\{\underline{\mathsf{zero}} : \mathbf{1}, \underline{\mathsf{succ}} : \downarrow_k^k \mathsf{nat}_k\}$$
$$\mathsf{list}_m = \oplus\{\underline{\mathsf{nil}} : \mathbf{1}, \underline{\mathsf{cons}} : \downarrow_m^k \mathsf{nat}_k \otimes \downarrow_m^m \mathsf{list}_m\}$$

We can picture a concrete layout for type nat as

zero	XXX
succ	a

where XXX is unused because $() : \mathbf{1}$ would be size 0 allocation, and a is the address of a cell again of type nat. Then for lists

nil	XXX	XXX
cons	a	b

where a is an address of type nat and b is an address of type list.

We can achieve these layout conventions with the following, specializing the rules to a single linear mode as in the earlier parts of this lecture.

$$\frac{}{a \Leftarrow A \vdash \mathbf{write}\ c\ \langle a \rangle :: (c \Leftarrow \downarrow A)}\ \downarrow X \qquad \frac{\Gamma, x \Rightarrow A \vdash P :: \delta}{\Gamma, c \Rightarrow \downarrow A \vdash \mathbf{read}\ c\ (\langle x \rangle \Rightarrow P) :: \delta}\ \downarrow L$$

These are actually just the rules of bidirectional Sax because we don't replace the addresses inside $\langle - \rangle$ by underscores. The reason is that the type is inhabited by an explicit address, and we need to read or write the explicit address.

We discuss the upshift (as a negative connective) in the next lecture.

7 Translations and Justifications

How can we justify the particular set or subset of the rules? One way is to describe a particular algorithm for typechecking and prove suitable properties for it. On a related note, we could directly prove a subformula propery for cut-free derivations. Another would be to show that translations from standard systems like the sequent calculus or bidirectional natural deduction preserve typing. You can find some considerations along these lines in the literature [Somayyajula, 2024].

The translation from ND to Snax is actually relevant if you'd like to build a compiler, so we'll discuss this in the next lecture.

8 Summary

The rules for Snax are in Figure 2. With them, we have a sound type-theoretic foundation for compact layout of data structures. We can also see it as a *generalization* of Sax, because there is a uniform translation of Sax into Snax that (intuitively) preserves the simple Sax layout structure.

$$\frac{B \leq A}{b \Rightarrow B \vdash \mathbf{id}\ a\ b :: (a \Leftarrow A)}\ \mathsf{id}{\Rightarrow}{\Leftarrow}$$

$$\frac{\Gamma \vdash P :: (a \Leftarrow A) \quad \Delta, a \Leftarrow A \vdash Q :: \delta}{\Gamma\ ;\ \Delta \vdash \mathbf{snip}\ a\ P\ Q :: \delta}\ \mathsf{snip}{\Leftarrow}{\Leftarrow}$$

$$\frac{\Gamma \vdash P :: (a \Rightarrow A) \quad \Delta, a \Rightarrow A \vdash Q :: \delta}{\Gamma\ ;\ \Delta \vdash \mathbf{snip}\ a\ P\ Q :: \delta}\ \mathsf{snip}{\Rightarrow}{\Rightarrow}$$

$$\frac{\Gamma \vdash P :: (x \Leftarrow A) \quad \Delta, x \Rightarrow A \vdash Q :: \delta}{\Gamma\ ;\ \Delta \vdash \mathbf{cut}\ (x : A)\ P\ Q :: \delta}\ \mathsf{cut}^x$$

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

$$\frac{\Gamma, c.\pi_1 \Rightarrow A, c.\pi_2 \Rightarrow B \vdash P :: \delta}{\Gamma, c \Rightarrow A \otimes B \vdash \mathbf{read}\ c\ ((\_,\_) \Rightarrow P) :: \delta}\ \otimes L$$

$$\frac{}{\cdot \vdash \mathbf{write}\ c\ (\,) :: c \Leftarrow \mathbf{1}}\ \mathbf{1}X \qquad \frac{\Gamma \vdash P :: \delta}{\Gamma, c \Rightarrow \mathbf{1} \vdash \mathbf{read}\ c\ ((\,) \Rightarrow P) :: \delta}\ \mathbf{1}L$$

$$\frac{(k \in L)}{a.\overline{k} \Leftarrow A_k \vdash \mathbf{write}\ c\ k(\_) :: c \Leftarrow \oplus\{\ell : A_\ell\}_{\ell \in L}}\ \oplus X$$

$$\frac{(\Gamma, c.\overline{\ell} \Rightarrow A_\ell \vdash P_\ell :: \delta) \quad (\forall \ell \in L)}{\Gamma, c \Rightarrow \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \mathbf{read}\ c\ \{\ell(\_) \Rightarrow P_\ell\}_{\ell \in L} :: \delta}\ \oplus L$$

Figure 2: Typing Rules for Snax

References

Henry DeYoung and Frank Pfenning. Data layout from a type-theoretic perspective. In *38th Conference on the Mathematical Foundations of Programming Semantics (MFPS 2022)*. Electronic Notes in Theoretical Informatics and Computer Science 1, 2022. URL https://arxiv.org/abs/2212.06321v6. Invited paper. Extended version available at https://arxiv.org/abs/2212.06321v3.pdf.

Henry DeYoung, Frank Pfenning, and Klaas Pruiksma. Semi-axiomatic sequent calculus. In Z. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, pages 29:1–29:22, Paris, France, June 2020. LIPIcs 167.

Michael Dummett. *The Logical Basis of Metaphysics*. Harvard University Press, Cambridge, Massachusetts, 1991. The William James Lectures, 1976.

Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Computing Surveys*, 54(5):98:1–98:38, 2022.

Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. Notes for three lectures given in Siena, Italy. Published in *Nordic Journal of Philosophical Logic*, 1(1):11-60, 1996, April 1983. URL http://www.hf.uio.no/ifikk/forskning/publikasjoner/tidsskrifter/njpl/vol1no1/meaning.pdf.

Greg Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, December 1995. Available as Technical Report CMU-CS-95-226.

Dag Prawitz. *Natural Deduction*. Almquist & Wiksell, Stockholm, 1965.

Siva Somayyajula and Frank Pfenning. Dependent type refinements for futures. In M. Kerjean and P. Levy, editors, *39th International Conference on Mathematical Foundations of Programming Semantics (MFPS 2023)*, Bloomington, Indiana, USA, June 2023. Preliminary version.

Siva Kamesh Somayyajula. *Total Correctness Type Refinements for Communicating Processes*. Ph.D. thesis, Carnegie Mellon University, May 2024. Available as Technical Report CMU-CS-24-108.