Lecture Notes on
Data Layout

15-836: Substructural Logics
Frank Pfenning

Lecture 17
November 9, 2023

1 Introduction

Data layout is a critical component in the efficient compilation of functional languages (see, for example, [Morrisett, 1995, Weeks, 2006, Vollmer et al., 2017, 2019]). Yet, in implementations of functional languages data layout decisions are left to the compiler rather than being available to the programmer. In today's lecture we design a type system in which certain high-level data layout decisions are explicit in the types, while lower-level details are still left to a compiler.

The surprising property of the type system is that it corresponds directly to a fragment of adjoint logic in its semi-axiomatic formulation which we call SNAX. In other words, SNAX provides a logical explanation for issues of data layout! Petersen et al. [2003] was an early attempt at characterizing data layout using an *ordered* type system. While this worked as far as it went, it did not generalize further. The root cause seems to be that the proofs-as-programs interpretation for ordered logic does not capture *adjacency* because the general rule of cut must apply to a proposition in the middle of ordered antecedents.

The line of research on SAX provided a surprising twist. SAX itself does not satisfy traditional cut elimination, as explained in a prior lecture. Certain cuts may be allowed if the cut formula arises from a use of a new axiom and is therefore a subformula of the goal sequent. We call these cuts *snips* and prove a new version of cut elimination [DeYoung et al., 2020] in which snips are allowed to remain. But we didn't tackle the question what the *computational* meaning of snips might be. It turns out that while a *cut* allocates a new memory cell, a *snip* merely computes an address relative to an existing address.

The fundamental connection between semi-axiomatic proofs and data layout is developed for nonlinear futures by DeYoung and Pfenning [2022]. Since this is largely consistent with the approach and notation of this course, we will not repeat

repeat it in these notes but provide a link to the extended version of the paper paper.[1]

These notes then will cover only the connection between *partial focusing* and data layout in the SNAX source language under the shared memory interpretation, which was discovered (during this course) in analogy to message sequences.

2 Data Layout: Compound Values

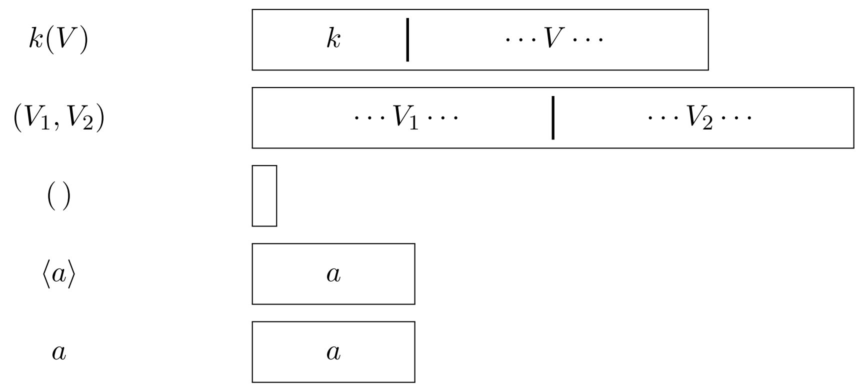
Message sequences were defined to model asynchronous communication along buffered channels. We just consider the positive types, because we won't be specific about how negative types are laid out (they are not directly observable, after all).

$$\begin{array}{llcll}
\text{Messages Sequences} & \overline{M} & ::= & k(\overline{M}) & (\oplus) \\
 & & \mid & (y, \overline{M}) & (\otimes) \\
 & & \mid & (\,) & (\mathbf{1}) \\
 & & \mid & \langle x' \rangle & (\downarrow) \\
 & & \mid & x' & \text{cont. channel}
\end{array}$$

When we think about memory layout, we do not need the first component of a pair to be an address—it could just be another value. The continuation channel x' is replaced by an address x, but in the typing rules to come later we will restrict such address to be of negative type. The reason is that we would like to statically allocate the space for a value. We just write V and K instead of $\overline{V}$ and $\overline{K}$ since the restricted case is just a special case.

$$\begin{array}{llcll}
\text{Values} & V & ::= & k(V) & (\oplus) \\
 & & \mid & (V_1, V_2) & (\otimes) \\
 & & \mid & (\,) & (\mathbf{1}) \\
 & & \mid & \langle x \rangle & (\downarrow) \\
 & & \mid & x & (\multimap, \&, \uparrow)
\end{array}$$

We picture the layout as follows:



[1] https://arxiv.org/abs/2212.06321v3.pdf

We imagine that the unit doesn't actually take any space, but we still display it as a narrow box.

Let's look at two recursive types:

$$\begin{aligned}&\mathsf{nat} = \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\}\\ &\mathsf{list} = \oplus\{\mathsf{nil} : \mathbf{1}, \mathsf{cons} : \mathsf{nat} \otimes \mathsf{list}\}\end{aligned}$$

Because these types are recursive and purely positive, their layout would be unbounded in size. This is the same problem as posed by (possibly mutually) recursive structs in C. In C, as here, the solution is to require an indirection via a pointer/address, which has a fixed size representation.

The indirection can be either through a downshift $\downarrow A$ or through a negative type A^-. For natural numbers, there are two obvious options:

$$\begin{aligned}&\mathsf{nat} = \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : {\downarrow}\mathsf{nat}\} &&\text{% eager}\\ &\mathsf{nat} = \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : {\uparrow}\mathsf{nat}\} &&\text{% lazy}\end{aligned}$$

The first would be the ordinary (eager) natural numbers, observable in their entirety. The second would be the *lazy natural numbers* because the successor a number would be $\mathsf{succ}\ \langle a\rangle$ where at the address is a continuation that can compute the tail.

But something doesn't seem right, because shifts in mixed linear/nonlinear logic go between structural and linear types. We are saved by the generality of adjoint logic, where $\downarrow^{\ell}_{m} A$ only requires that $\ell \geq m$. If we are working just with linear natural numbers, the downshift would be $\downarrow^{\mathsf{L}}_{\mathsf{L}}\mathsf{nat}$. For structural natural numbers it would probably $\downarrow^{\mathsf{S}}_{\mathsf{S}}\mathsf{nat}$. Since for the moment we are just working in purely linear logic, we just write $\downarrow A$ for $\downarrow^{\mathsf{L}}_{\mathsf{L}} A$.

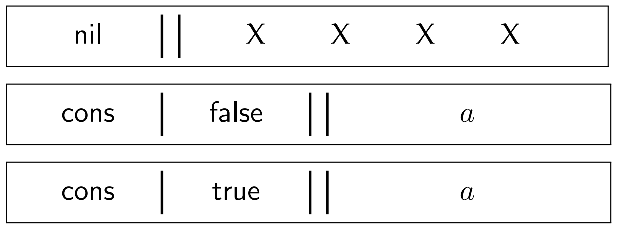
The ordinary eager lists might have pointers to natural numbers.

$$\mathsf{list} = \oplus\{\mathsf{nil} : \mathbf{1}, \mathsf{cons} : {\downarrow}\mathsf{nat} \otimes {\downarrow}\mathsf{list}\}$$

The representation might be more compact for lists of booleans by "inlining" them instead of having a pointer to a Boolean.

$$\begin{aligned}&\mathsf{bool} = \oplus\{\mathsf{false} : \mathbf{1}, \mathsf{true} : \mathbf{1}\}\\ &\mathsf{listbool} = \oplus\{\mathsf{nil} : \mathbf{1}, \mathsf{cons} : \mathsf{bool} \otimes {\downarrow}\mathsf{listbool}\}\end{aligned}$$

When a pointer to the element is embedded in a list it is called a *boxed representation*; otherwise it is said to be *unboxed*. Data of the unboxed type $\mathsf{listbool}$ might be layed out as on of the following, where a is the address for the tail of the list.



The extra unused space for nil is there because all values of type listbool should be laid out with the same width.

3 Partial Focusing Revisited

As can be seen from the development above, values still arise from partial focusing but with slightly different criteria for partiality. We begin with the rules for writing with positive types.

$$\frac{\Delta \vdash V : \lceil A \rceil}{\Delta \vdash \mathbf{write}\ x\ V :: (x : A)}\ \mathsf{write}$$

Now we have rules for each of the positive types with the corresponding values.

$$\frac{\Delta \vdash V : \lceil A_k \rceil}{\Delta \vdash k(V) : \lceil \oplus\{\ell : A_\ell\}_{\ell \in L} \rceil}\ \oplus R$$

$$\frac{\Delta_1 \vdash V_1 : \lceil A \rceil \quad \Delta \vdash V_2 : \lceil B \rceil}{\Delta_1, \Delta_2 \vdash (V_1, V_2) : \lceil A \otimes B \rceil}\ \otimes R \qquad \frac{}{\cdot \vdash (\,) : \lceil \mathbf{1} \rceil}\ \mathbf{1}R$$

When we encounter a downshift or a negative type we end the partial focusing phase, either with the corresponding axiom or an identity.

$$\frac{}{x : A \vdash \langle x \rangle : \lceil \downarrow A \rceil}\ \mathsf{down}X \qquad \frac{}{x : A^- \vdash x : \lceil A^- \rceil}\ \mathsf{id}^-$$

Pattern matching works symmetrically. The pattern has to be deep enough to cover all well-typed values of a given type. Inversion now has to continue on both sides of a pair, so we need to generalize to allow the patterns to be nested.

Pattern Sequence	$\overline{V}$	$::=$	$V \cdot \overline{V} \mid (\cdot)$
Continuations	K	$::=$	$(\overline{V} \Rightarrow P \mid K) \mid \cdot$

The sequence of nested patterns match the ordered context in $\Delta\ ;\ \Omega \vdash K :: \delta$. The judgment is started with Ω being a singleton.

$$\frac{\Delta\ ;\ \lceil A \rceil \vdash K :: \delta}{\Delta, x : A \vdash \mathbf{read}\ x\ K :: \delta}\ \mathsf{read}$$

$$\frac{\Delta \,;\, A\; B\; \Omega \vdash K \mathbin{@} (\_,\_) :: \delta}{\Delta \,;\, (A \otimes B)\; \Omega \vdash K :: \delta}\;\otimes L \qquad \frac{\Delta \,;\, \Omega \vdash K \mathbin{@} (\,) :: \delta}{\Delta \,;\, \mathbf{1}\; \Omega \vdash K :: \delta}\;\mathbf{1}L$$

$$\frac{\Delta \,;\, A_\ell\; \Omega \vdash \overline{K} \mathbin{@} \ell(\_) :: \delta \quad (\forall \ell \in L)}{\Delta \,;\, \oplus\{\ell : A_\ell\}_{\ell \in L}\; \Omega \vdash K :: \delta}\;\oplus L$$

$$\frac{\Delta, x : A \,;\, \Omega \vdash \overline{K} \mathbin{@} \langle x \rangle :: \delta}{\Delta \,;\, ({\downarrow}A)\; \Omega \vdash K :: \delta}\;{\downarrow}L \qquad \frac{\Delta, x : A^- \,;\, \Omega \vdash \overline{K} \mathbin{@} x :: \delta}{\Delta \,;\, A^-\; \Omega \vdash K :: \delta}$$

$$\frac{\Delta \vdash P :: \delta}{\Delta \,;\, \cdot \vdash (\cdot) \Rightarrow P :: \delta}$$

In the definition below we don't explicate failure conditions (for example, if there no branches for a given tag, or if there is a mismatch between the projection p and the pattern).

$$\begin{array}{llcl}
((V_1, V_2) \cdot \overline{V} \Rightarrow P \mid K) & @ \;\; (\_,\_) & = & (V_1 \cdot V_2 \cdot \overline{V} \Rightarrow P) \mid (K \mathbin{@} (\_,\_)) \\
((\,) \cdot \overline{V} \Rightarrow P \mid K) & @ \;\; (\,) & = & (\overline{V} \Rightarrow P) \mid (K \mathbin{@} (\,)) \\
(\ell(V) \cdot \overline{V} \Rightarrow P \mid K) & @ \;\; \ell(\_) & = & (V \cdot \overline{V} \Rightarrow P) \mid (K \mathbin{@} \ell(\_)) \\
(k(V) \cdot \overline{V} \Rightarrow P \mid K) & @ \;\; \ell(\_) & = & K \mathbin{@} \ell(\_) \qquad \text{for } k \neq \ell \text{ and } k \in L \\
(\langle x \rangle \cdot \overline{V} \Rightarrow P(x) \mid K) & @ \;\; \langle y \rangle & = & (\overline{V} \Rightarrow P(y)) \mid (K \mathbin{@} \langle y \rangle) \\
(x \cdot \overline{V} \Rightarrow P(x) \mid K) & @ \;\; y & = & (\overline{V} \Rightarrow P(y)) \mid (K \mathbin{@} y) \\
(\cdot) & @ \;\; p & = & (\cdot)
\end{array}$$

4 Example: Append with Three Types

We give three different examples, one for appending two lists of pointers to natural numbers, and one for appending lists of (unboxed) booleans.

In the first example we pass memory contents directly instead of pointers.

```
type nat = +{'zero : 1, 'succ : <down> nat}
type list = +{'nil : 1, 'cons : <down> nat * <down> list}

proc append (R : list) (L : list) (K : list) =
  read L ( 'nil() => move R K
         | 'cons(<x>, <L'>) => R' : list <- call append R' L' K
                               write R 'cons(<x>, <R'>) )
```

In the next version, we pass pointers instead of the layout structures. Because in this version we have to match all the way until we encounter an address, there is a slight awkwardness in the recursive calls: we might prefer not to decompose and recompose the tail of the list.

```
type nat = +{'zero : 1, 'succ : nat_ptr}
type nat_ptr = <down> nat

type list = +{'nil : 1, 'cons : <down> nat * list_ptr}
type list_ptr = <down> list

proc append (R : list_ptr) (L : list_ptr) (K : list_ptr) =
  read L ( <'nil()> => move R K
         | <'cons(<x>,<L'>)> => R' : list_ptr <- call append R' <L'> K
                                write R 'cons(<x>, R') )
```

In the unboxed example we see that partial matching could be quite helpful, because without that we need to match the entirety of the boolean instead of being able to leave it as a variable.

```
type bool = +{'false : 1, 'true : 1}
type boollist = +{'nil : 1, 'cons : bool * <down>boollist}

proc append (R : boollist) (L : boollist) (K : boollist) =
  read L ( 'nil() => move R K
         | 'cons('false(),<L'>) => R' : list <- call append R' L' K
                                   write R 'cons('false(), <R'>)
         | 'cons('true(),<L'>) => R' : list <- call append R' L' K
                                  write R 'cons('true(), <R'>) )
```

It would be easy to accommodate partial matches by removing the restriction on variables in values to be of negative type.

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.

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

Leaf Petersen, Robert Harper, Karl Crary, and Frank Pfenning. A type theory for memory allocation and data layout. In G. Morrisett, editor, *Conference Record of the 30th Annual Symposium on Principles of Programming Languages (POPL'03)*, pages 172–184, New Orleans, Louisiana, January 2003. ACM Press. Extended version available as Technical Report CMU-CS-02-171, December 2002.

Michael Vollmer, Sarah Spall, Buddhika Chamith, Laith Sakka, Chaitanya Koparkar, Milind Kulkarni, Sam Tobin-Hochstadt, and Ryan R. Newton. Compiling tree transformas to operate on packed representations. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, pages 26:1–26:29, Barcelona, Spain, June 2017. LIPIcs 745.

Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, and Milind Kulkarni. LoCal: A language for programs operating in serialized data. In Kathryn McKinley and Kathleen Fisher, editors, *40th Conference on Programming Language Design and Implementation (PLDI 2019)*, pages 48–62, Phoenix, Arizona, June 2019. ACM.

Stephen Weeks. Whole-program compilation in MLton. In Andrew Kennedy and Franųis Pottier, editors, *Proceedings of the Workshop on ML*, Portland, Oregon, September 2006. ACM. Slides available at `http://www.mlton.org/References.attachments/060916-mlton.pdf`.