Lecture Notes on Calling Conventions

15-417/817: HOT Compilation
Frank Pfenning

Lecture 16
March 18, 2025

1 Introduction

In the last lecture we looked at data layout for positive types. Negative types correspond to computation, so one might think that data layout does not apply. On the other hand, the subformula property of bidirectional Sax should carry over from the positive to the negatives connectives. The outcome is a set of "calling conventions" for functions or lazy records. These are still pretty abstract, and we leave translation to a lower level open.

Before, we also test our intuition for the rules by describing compilation for the positives, later to be augmented by the negatives.

2 Compilation for Positive Types

We have two judgments to compile:

$$\Gamma \vdash e \Longleftarrow A$$
$$\Gamma \vdash s \Longrightarrow A$$

where s is the syntactic class of expressions that synthesize their type.

The result should be a bidirectionally well-typed Snax program. We should check the theorems we'd like to prove about the translation to guide our intuition. The first one is easy:

If $\Gamma \vdash e \Longleftarrow A$ *then* $\Gamma^{\Rightarrow} \vdash [\![e]\!]\, d :: (d \Leftarrow A)$. *(not quite correct)*

Here we write $\Gamma^{\Rightarrow}$ for the context where every variables synthesizes its type. We postpone the case for $s \Longrightarrow A$ for a few moments.

Pairs $A \otimes B$. We define

$$\begin{array}{rcl} [\![(e_1, e_2)]\!]\, d & = & \mathbf{snip}^{\Leftarrow}\ d.\pi_1 \\ & & \quad [\![e_1]\!]\, d.\pi_1 \\ & & \mathbf{snip}^{\Leftarrow}\ d.\pi_2 \\ & & \quad [\![e_2]\!]\, d.\pi_2 \\ & & \mathbf{write}\ d\ (\_, \_) \end{array}$$

We see that it make sense for the final write operation to be silent, since $[\![e_1]\!]\, d.\pi_1$ promises to write to $d.\pi_1$ and $[\![e_2]\!]\, d.\pi_2$ promises to write to $d.\pi_2$. Together, they already fill the pair.

The left rule requires us to consider how to translate synthesis, because in

$$\mathbf{match}\ s\ ((x, y) \Rightarrow e)$$

the expression s synthesizes.

The first natural attempt fails.

If $\Gamma \vdash s \Longrightarrow A$ *then* $\Gamma^{\Rightarrow} \vdash [s]\, d :: (d \Rightarrow A)$. *(incorrect)*

This does not work because the destination d is determined by the translation of s rather than given a priori. For example, if s is a variable x, then there is some address associated with x that must be taken into account.

We can use the standard technique of providing $[e]$ with a *continuation* [Reynolds, 1972]. In our case, the continuation expects a destination as argument and returns a procedure P. Then the correctness is a little harder to formulate, but not inordinately so.[1]

If $\Gamma \vdash s \Longrightarrow A$ and $\Delta, a \Rightarrow A \vdash \underline{K}(a) :: \delta$ then $\Gamma^{\Rightarrow}\ ;\ \Delta \vdash [s]\, \underline{K} :: \delta$. (almost correct)

This still doesn't work. Let's try (writing $\underline{\lambda}$ for a meta-level abstraction to form a continuation).

$$[\![\mathbf{match}\ s\ ((x, y) \Rightarrow e)]\!]\, d \quad = \quad [s](\underline{\lambda}c.\, \mathbf{read}\ c\ ((\_, \_) \Rightarrow [\![e]\!]\, d))$$

The problem here is that the source variable x in e is represented by address $z.\pi_1$, and y is represented by $z.\pi_2$, but this association is not explicit.

To be rigorous, we define a mapping ρ from variables to addresses of the same type that is threaded through the translation. We write $\rho(\Gamma)$ for applying this mapping to the variables in Γ, with all resulting addresses synthesizing.

If $\Gamma \vdash e \Longleftarrow A$ *then* $\rho(\Gamma) \vdash [\![e]\!]^{\rho}\, d :: (d \Leftarrow A)$.
If $\Gamma \vdash s \Longrightarrow A$ *and* $\Delta, a \Rightarrow A \vdash \underline{K}(a) :: \delta$ *then* $\rho(\Gamma)\ ;\ \Delta \vdash [s]^{\rho}\, \underline{K} :: \delta$.

Specifically for pairs we get

$$\begin{array}{lcl}
[\![(e_1, e_2)]\!]^{\rho}\, d & = & \mathbf{snip}^{\Leftarrow}\ d.\pi_1 \\
& & \quad [\![e_1]\!]^{\rho}\, d.\pi_1 \\
& & \mathbf{snip}^{\Leftarrow}\ d.\pi_2 \\
& & \quad [\![e_2]\!]^{\rho}\, d.\pi_2 \\
& & \mathbf{write}\ d\ (_, _) \\
[\![\mathbf{match}\ s\ ((x, y) \Rightarrow e)]\!]^{\rho}\, d & = & [s](\underline{\lambda}c.\, \mathbf{read}\ c\ ((_, _) \Rightarrow [\![e]\!]^{\rho'}\, d)) \\
& & \text{where } \rho' = (\rho, x \mapsto c.\pi_1, y \mapsto c.\pi_2)
\end{array}$$

For the example, we also need to translate variables (which synthesize), and we need to handle the transition from synthesis to checking.

$$\begin{array}{lcl}
[x]^{\rho}\, \underline{K} & = & \underline{K}(\rho(x)) \\
[\![s]\!]^{\rho}\, d & = & [s](\underline{\lambda}a.\, \mathbf{id}\ d\ a)
\end{array}$$

[1] The standard notation K overlaps with our internal notion of continuation K, so we underline it when used at the metalevel.

The last case uses the id⇒⇐ version of the identity.

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

It is also the only place in the translation where the identity is introduced, and therefore the only place where subtyping comes into play. This is satisfying, because in ND the only place for subtyping is the $\Longrightarrow\Longleftarrow$ rule.

As an example, we consider

$$z : A \otimes B \vdash \mathbf{match}\ z\ ((x, y) \Rightarrow (y, x)) \Longleftarrow B \otimes A$$

If we start with $\rho = (z \mapsto c)$ then we can calculate the translation and we end up with the code from last lecture. We highly encourage you to carry out this calculation.

It is also a straightforward exercise to fill in the remaining translation rules for the positive fragment, so we skip them and move on to the negatives.

3 Function Types

In the original paper about data layout using Snax [DeYoung and Pfenning, 2022], we left negative types they were the way in Sax [DeYoung et al., 2020] because we didn't quite know how to handle them from the layout perspective. We still don't know everything, but at least in the linear case matters have somewhat clarified.

Let's recall the rules for function types $A \to B$ in Sax. Because the function type is negative, right rules remain and left rules are turned into axioms.

$$\frac{\Gamma, x : A \vdash P :: (y : B)}{\Gamma \vdash \mathbf{write}\ c\ ((x, y) \Rightarrow P) :: (c : A \to B)}\ {\to}R \qquad \frac{}{a : A, c : A \to B \vdash \mathbf{read}\ c\ (a, b) :: (b : B)}\ {\to}X$$

Annotating this as synthesis or checking, keeping in mind the rules of the game from the previous lecture, we get

$$\frac{\Gamma, x \Rightarrow A \vdash P :: (y \Leftarrow B)}{\Gamma \vdash \mathbf{write}\ c\ ((x, y) \Rightarrow P) :: (c \Leftarrow A \to B)}\ {\to}R \qquad \frac{}{a \Leftarrow A, c \Rightarrow A \to B \vdash \mathbf{read}\ c\ (a, b) :: (b \Rightarrow B)}\ {\to}X$$

Whenever a type is decomposed into its components we should be able to compute the address of the components.

$$\frac{\Gamma, c.\pi_1 \Rightarrow A \vdash P :: (c.\pi_2 \Leftarrow B)}{\Gamma \vdash \mathbf{write}\ c\ ((\_, \_) \Rightarrow P) :: (c \Leftarrow A \to B)}\ {\to}R$$

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

We have done this mechanically, so there are some questions. When we write the continuation $(\_, \_) \Rightarrow P$ we know where the program P can find its argument and its destination. But it is not silent, because $K = ((\_, \_) \Rightarrow P)$ still has to be written. In practice, there is also the question of closure conversion which we put aside for now. Also, $\mathbf{read}\ c\ (\_, \_)$ cannot be silent, because it actually invokes the continuation K stored at c. This program reads from $c.\pi_1$ (the argument of

the function) and writes to $c.\pi_2$ (the destination of the function). So the actual layout might be something like this:

$$(c : A \to B) \mapsto \boxed{(K) \mid A \mid B}$$

where (K) is the address of the (Snax) continuation K. So the address calculations would be slightly different than for positive pairs, because $c.\pi_1 \neq c$, for example.

How do we now compile? We have

$$\begin{array}{lcl} [\![\lambda x.\, e]\!]^\rho\, d & = & \mathbf{write}\ d\ ((\_,\_) \Rightarrow [\![e]\!]^{\rho'}\, d.\pi_2) \\ & & \text{where } \rho' = (\rho, x \mapsto d.\pi_1) \end{array}$$

$$\begin{array}{lcl} [e_1\, e_2]^\rho\, \underline{K} & = & [e_1](\underline{\lambda} a.\, \mathbf{snip}^{\Leftarrow}\ a.\pi_1 \\ & & \qquad [\![e_2]\!]^\rho\, a.\pi_1 \\ & & \quad \mathbf{snip}^{\Rightarrow}\ a.\pi_2 \\ & & \qquad \mathbf{read}\ a\ (\_,\_) \\ & & \quad \underline{K}(a.\pi_2)) \end{array}$$

In order to see the calling conventions hiding here, we can compile $[\![f\ \ x]\!]^\rho\ \ d$ where f and x are variables with $\rho = (f \mapsto c, x \mapsto a)$. Eliding explicit snips by using sequencing, we get

$$\begin{array}{lcll} [\![f\, x]\!]^\rho\, d & = & \mathbf{id}\ c.\pi_1\ a\ ; & \%\ \textit{move argument } x \textit{ at } a \textit{ into place} \\ & & \mathbf{read}\ c\ (\_,\_)\ ; & \%\ \textit{call function } f \\ & & \mathbf{id}\ d\ c.\pi_2 & \%\ \textit{move result to destination } d \end{array}$$

If we take the example of swap from the last lecture and reformulate it as a function $f : A \otimes B \to B \otimes A$ then the layout will be something like

$$c \mapsto \begin{array}{|c|c|c|c|c|} \hline (K) & A & B & B & A \\ & c.\pi_1.\pi_1 & c.\pi_1.\pi_2 & c.\pi_2.\pi_1 & c.\pi_2.\pi_2 \\ \hline \end{array}$$

Here, the first two slots are taken by the arguments of type A and B, and the second two slots are taken up by the results of type B and A. We also have (the address of) the code. In short form, this would be

$$\begin{array}{l} (\_,\_) \Rightarrow \\ \quad \mathbf{id}\ c.\pi_2.\pi_1\ c.\pi_1.\pi_2\ ; \\ \quad \mathbf{id}\ c.\pi_2.\pi_2\ c.\pi_1.\pi_1 \end{array}$$

which is about as efficient as we could hope. In addition, before a call to f the arguments have to be moved into place, and after the return of f the have to be moved to their destination.

4 Lazy Records

Lazy records work analogously, but are a bit simpler. We go directly to the bidirectional form.

$$\frac{(\Gamma \vdash P_\ell :: (x_\ell \Leftarrow A_\ell)) \quad (\forall \ell \in L)}{\Gamma \vdash \mathbf{write}\ c\ \{\ell(x_\ell) \Rightarrow P_\ell\}_{\ell \in L} :: (c \Leftarrow \&\{\ell : A_\ell\}_{\ell \in L})}\ \&R$$

$$\frac{(k \in L)}{c \Rightarrow \&\{\ell : A_\ell\}_{\ell \in L} \vdash \mathbf{read}\ c\ k(a) :: (a \Rightarrow A_k)}\ \&X$$

As may be expected, we reuse the projections $a.\overline{l}$, although the actual address calculations will differ because the continuation needs to be stored as well. This parallels the way function types differ from pairs.

$$\frac{(\Gamma \vdash P_\ell :: (c.\overline{\ell} \Leftarrow A_\ell)) \quad (\forall \ell \in L)}{\Gamma \vdash \mathbf{write}\ c\ \{\ell(\_) \Rightarrow P_\ell\}_{\ell \in L} :: (c \Leftarrow \&\{\ell : A_\ell\}_{\ell \in L})}\ \&R$$

$$\frac{(k \in L)}{c \Rightarrow \&\{\ell : A_\ell\}_{\ell \in L} \vdash \mathbf{read}\ c\ k(\_) :: (c.\overline{k} \Rightarrow A_k)}\ \&X$$

We also have:

$$\begin{array}{lcl} [\![\{\ell \Rightarrow e_\ell\}]\!]^\rho\, d & = & \mathbf{write}\ d\ \{\ell(\_) \Rightarrow [\![e_\ell]\!]^\rho\, d.\overline{\ell}\} \\ [e.k]^\rho\, \underline{K} & = & [e]^\rho(\underline{\lambda} a.\ \mathbf{snip}^{\Rightarrow}\ a.\overline{k} \\ & & \qquad \mathbf{read}\ a\ k(\_) \\ & & \qquad \underline{K}(a.\overline{k})) \end{array}$$

5 Upshifts

Downshifts interrupt the layout patterns by storing an arbitrary address, so we do the same here. The rules may look a little bit degenerate because we are in the purely linear case, but the upshift can still be justified as building a so-called *thunk* as in call-by-push-value [Levy, 2001].

$$\frac{\Gamma \vdash P :: (x \Leftarrow A)}{\Gamma \vdash \mathbf{write}\ c\ (\langle x \rangle \Rightarrow P) :: (c \Leftarrow \uparrow A)}\ \uparrow R$$

$$\frac{}{c \Rightarrow \uparrow A \vdash \mathbf{read}\ c\ \langle a \rangle :: (a \Rightarrow A)}\ \uparrow L$$

These rules remain the same as in Snax because no address calculation is performed. Instead, addresses are stored and retrieved.

$$\begin{array}{lcl} [\![\mathbf{susp}\ e]\!]^\rho\, d & = & \mathbf{write}\ d\ (\langle x \rangle \Rightarrow [\![e]\!]^\rho x) \\ [e.\mathbf{force}]^\rho\, \underline{K} & = & [e]^\rho(\underline{\lambda} a.\ \mathbf{cut}\ x \\ & & \qquad \mathbf{read}\ a\ \langle x \rangle \\ & & \qquad \underline{K}(x)) \end{array}$$

6 Further Questions

In our presentation of Snax, we have presumed a linear type system, which is the best-case scenario. Still, there are several questions we haven't tackled. One is: In ND, we presuppose (and maintain) that all variables in the context are distinct. What about in Snax? Now that the context doesn't just contain variables but addresses (which include projections) we need a counterpart of the distinctness condition. See DeYoung and Pfenning [2022] for such a condition that entails suitable forms of preservation and progress for the dynamics.

The intuition behind synthesis and checking seems clear, but can we elaborate it to a more precisely defined algorithm for type-checking? How does this account for the conditions mentioned in the previous paragraph?

Along similar lines, the calling conventions that arise may be questionable in the case contraction is permitted. The problem is that multiple calls to the same function would interfere with each other of the allocation for the function argument and destination were shared. It appears that for positive types, the presence of contraction is most naturally and efficiently implemented by sharing, but for negative types contraction may actually need to be copying. In a sequential implementation, this copying may be akin to a creating a fresh stack frame.

The considerations for negative types interact with closure conversion. Precisely, what is this interaction and how can it be managed in a lower-level implementation?

Optimizations, such as cut/id reduction or vertical and horizontal reuse will have to be reconsidered. A step in that direction has been taken by Ng [2024], but it differs from the current implementation in the research compiler. How do we formulate such optimizations, and prove their dynamics correct?

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.

Paul Blain Levy. *Call-by-Push-Value*. PhD thesis, University of London, 2001.

Daniel Ng. Memory reuse in linear functional computation. Honors thesis, Carnegie Mellon University, May 2024. URL http://www.cs.cmu.edu/~fp/courses/15417-s25/misc/Ng24.pdf.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740, Boston, Massachusetts, August 1972. ACM Press. Reprinted in *Higher-Order and Symbolic Computation*, 11(4), pp.363–397, 1998.