Lecture Notes on Compilation

15-417/817: HOT Compilation
Frank Pfenning

Lecture 4
January 23, 2025

1 Introduction

In the last lecture we introduced first-order functional programming (if such a thing exists) based on a computational interpretation of natural deduction for positive equirecursive types. For lack of a better name, we'll call this source language ND. In the lecture before we introduced an imperative intermediate language called SAX, based on a computational interpretation of the semi-axiomatic sequent calculus.

In this lecture we show how to compile the former to the latter. This echoes a proof-theoretic translation from natural deduction to the semi-axiomatic sequent calculus. We will also see our very first optimization!

We also address one of the most annoying parts of programming in ND, namely that, so far, we have to deconstruct data one constructor at a time. The solution is a form of matching with nested patterns.

2 Compiling from ND to SAX

Recall the two typing judgments. First, for ND:

$$\underbrace{x_1 : A_1, \ldots, x_n : A_n}_{\text{value variables}} \vdash \underbrace{e : A}_{\text{computation}}$$

where the x_i stands for a large value of type A_i, and e computes a large value of type A. Next, for SAX:

$$\underbrace{x_1 : A_1, \ldots, x_n : A_n}_{\text{source addresses}} \vdash P :: \underbrace{d : A}_{\text{destination address}}$$

where the x_i stand for addresses we may (and in the linear case, must) read and d is the destination command P must write a small value to.

It looks possible to keep variables the same, because the large value represented by x_i may be read off the heap at address x_i. On the other hand, the computation of e *returns* a value, while the command P *writes* a value to destination d. This suggests defining a translation taking an expression and a destination, returning a command

$$[\![e]\!]\, d = P$$

such that

$$\Gamma \vdash e : A \quad \text{implies} \quad \Gamma \vdash [\![e]\!]\, d :: (d : A)$$

We now go through the few constructs we have one-by-one to see what the translation might be. We should keep in mind the typing requirement above, and the intuition:

If e evaluates to large value V then the heap at destination d after executing $[\![e]\!]\, d$ represents V.

Variables. This is easy:

$$[\![x]\!]\, d = \mathbf{id}\ d\ x$$

We can verify that this translation types correctly, and also follows our intuition about computation: if the heap at address x represents a value V, then the heap at d will do so afterwards.

Unit 1. The constructor is again straightforward.

$$[\![(\,)]\!]\, d = \mathbf{write}\ d\ (\,)$$

The destructor for type $\mathbf{1}$ in ND is

$$\mathbf{match}\ e\ \mathbf{with}\ (\,) \Rightarrow e'$$

We can translate this if we allocate a new destination x for the value of e, and then read from x matching against the unit value $(\,)$.

$$\begin{array}{rcl} [\![\mathbf{match}\ e\ (\,) \Rightarrow e']\!]\, d' & = & \mathbf{cut}\ x : \mathbf{1} \\ & & \quad [\![e]\!]\, x \\ & & \mathbf{read}\ x\ (\,) \Rightarrow [\![e']\!]\, d' \quad (x \text{ fresh}) \end{array}$$

Pairs $A \otimes B$. In this case, constructor expressions (e_1, e_2) require us to allocate two new destinations, one for the value of e_1 and one for the value of e_2.

$$\begin{array}{rcl} [\![(e_1, e_2)]\!]\, d & = & \mathbf{cut}\ x_1 : A_1 \\ & & \quad [\![e_1]\!]\, x_1 \\ & & \mathbf{cut}\ x_2 : A_2 \\ & & \quad [\![e_2]\!]\, x_2 \\ & & \mathbf{write}\ d\ (x_1, x_2) \quad (x_1 \text{ and } x_2 \text{ fresh}) \end{array}$$

One problem we sweep under he rug here is how to obtain the types A_1 and A_2 for e_1 and e_2. In order to understand this, let's verify that the translation is indeed type preserving as we claim. What we write below would be one case in proving the preservation of types under the translation, where $[\![\mathcal{D}_1]\!]$ and $[\![\mathcal{D}_2]\!]$ would be obtained by induction hypothesis.

$$\frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Gamma_1 \vdash e_1 \Longleftarrow A_1 & \Gamma_2 \vdash e_2 \Longleftarrow A_2\end{array}}{\Gamma_1, \Gamma_2 \vdash (e_1, e_2) \Longleftarrow A_1 \otimes A_2}\ \otimes I$$

$$\rightsquigarrow$$

$$\dfrac{\begin{array}{cc}[\![\mathcal{D}_1]\!] & \dfrac{\begin{array}{cc}[\![\mathcal{D}_2]\!] & \\ \Gamma_2 \vdash [\![e_2]\!]\, x_2 :: (x_2 : A_2) & x_1 : A_1, x_2 : A_2 \vdash \mathbf{write}\ d\ (x_1, x_2) :: (d : A_1 \otimes A_2)\end{array}}{x_1, \Gamma_2 : A \vdash \_ :: (d : A_1 \otimes A_2)}\ \otimes X \\ \Gamma_1 \vdash [\![e_1]\!]\, x_1 :: (x_1 : A_1) & x_1, \Gamma_2 : A \vdash \_ :: (d : A_1 \otimes A_2)\end{array}}{\Gamma_1, \Gamma_2 \vdash \_ :: (d : A_1 \otimes A_2)}\ \begin{array}{c}\mathsf{cut} \\ \mathsf{cut}\end{array}$$

Here we have elided some of the commands resulting from the translation for the sake of readability. We see that the translation may really be considered a translation of typing derivations instead of terms. If we follow the structure of bidirectional derivations, the types we need for the cuts (namely A_1 and A_2) will be available to us.

The translation of the destructor follows the previous idea: allocate a new destination x, evaluate the match subject to write a pair to x, and then read the result from x.

$$\begin{array}{rcl} [\![\mathbf{match}\ e\ (x_1, x_2) \Rightarrow e']\!]\, d' & = & \mathbf{cut}\ x : A_1 \otimes A_2 \\ & & \quad [\![e]\!]\, x \\ & & \mathbf{read}\ x\ (x_1, x_2) \Rightarrow [\![e']\!]\, d' \quad (x \text{ fresh}) \end{array}$$

Again, we can map this onto a translation between typing derivations.

$$\frac{\mathcal{D} \qquad\qquad \mathcal{E} \\ \Gamma \vdash e \Longrightarrow A_1 \otimes A_2 \quad \Delta, x_1 : A_1, x_2 : A_2 \vdash e' \Longleftarrow A'}{\Gamma, \Delta \vdash \mathbf{match}\ e\ \mathbf{with}\ (x_1, x_2) \Rightarrow e' \Longleftarrow A'}\ \otimes E$$

$$\rightsquigarrow$$

$$\frac{\begin{array}{cc} [\![\mathcal{D}]\!] & \dfrac{[\![\mathcal{E}]\!]}{\Delta, x_1 : A_1, x_2 : A_2 \vdash [\![e']\!]\, d' :: (d' : A')} \\ \Gamma \vdash [\![e]\!]\, x :: (x : A_1 \otimes A_2) & \dfrac{}{\Delta, x : A_1 \otimes A_2 \vdash \mathbf{read}\ x\ (x_1, x_2) \Rightarrow [\![e']\!]\, d' :: (d' : A')}\ \otimes L \end{array}}{\Gamma, \Delta \vdash \_ :: (d' : A')}\ \mathsf{cut}$$

Sums $\oplus\{\ell : A_\ell\}$. There isn't really anything new here, except that each branch of a match expression must be translated. We don't show how to compute the types, but it follows the reasoning for pairs.

$$\begin{array}{rcll} [\![k(e)]\!]\, d & = & \mathbf{cut}\ x : A_k & \\ & & \quad [\![e]\!]\, x & \\ & & \mathbf{write}\ d\ k(x) & (x \text{ fresh}) \\ \\ [\![\mathbf{match}\ e\ \mathbf{with}\ \{\ell(x_\ell) \Rightarrow e_\ell\}_{\ell \in L}]\!]\, d' & = & \mathbf{cut}\ x : \oplus\{\ell : A_\ell\}_{\ell \in L} & \\ & & \quad [\![e]\!]\, x & \\ & & \mathbf{read}\ x\ \{\ell(x_\ell) \Rightarrow [\![e_\ell]\!]\, d'\}_{\ell \in L} & (x \text{ fresh}) \end{array}$$

This already concludes our compiler. Even if we don't give a correctness proof, it doesn't seem like it should be difficult to construct.

3 Some Simple Examples

Say, we have in ND:

$\mathbf{type}\ \mathsf{nat} = \oplus\{\underline{\mathsf{zero}} : \mathbf{1}, \underline{\mathsf{succ}} : \mathsf{nat}\}$
$\mathbf{defn}\ \mathsf{one} : \mathsf{nat} = \underline{\mathsf{succ}}(\underline{\mathsf{zero}}(\,))$

The type will be the same in SAX, and the definition becomes

$\mathbf{proc}\ \mathsf{one}\ (d : \mathsf{nat}) = [\![\underline{\mathsf{succ}}(\underline{\mathsf{zero}}(\,))]\!]\, d$

Its easy to work out the definition, but we recommend you go through it.

$$
\begin{aligned}
[\![\underline{\mathsf{succ}}(\underline{\mathsf{zero}}(\,))]\!]\, d &= \mathbf{cut}\ x_1 : \mathsf{nat} \\
&\qquad [\![\underline{\mathsf{zero}}(\,)]\!]\, x_1 \\
&\quad\ \mathbf{write}\ d\ \underline{\mathsf{succ}}(x_1) \\
&= \mathbf{cut}\ x_1 : \mathsf{nat} \\
&\qquad \mathbf{cut}\ x_2 : \mathsf{nat} \\
&\qquad\quad [\![(\,)]\!]\, x_2 \\
&\qquad \mathbf{write}\ x_1\ \underline{\mathsf{zero}}(x_2) \\
&\quad\ \mathbf{write}\ d\ \underline{\mathsf{succ}}(x_1) \\
&= \mathbf{cut}\ x_1 : \mathsf{nat} \\
&\qquad \mathbf{cut}\ x_2 : \mathsf{nat} \\
&\qquad\quad \mathbf{write}\ x_2\ (\,) \\
&\qquad \mathbf{write}\ x_1\ \underline{\mathsf{zero}}(x_2) \\
&\quad\ \mathbf{write}\ d\ \underline{\mathsf{succ}}(x_1)
\end{aligned}
$$

This looks like a quite plausible implementation: we allocate cells on the heap and fill them with the correct small values.

Let's consider the predecessor as something with an elimination rule.

$$
\begin{aligned}
&\mathbf{defn}\ \mathsf{pred}\ (x : \mathsf{nat}) : \mathsf{nat} = \mathbf{match}\ x\ \mathbf{with} \\
&\quad \mid \underline{\mathsf{zero}}(u) \Rightarrow \underline{\mathsf{zero}}(u) \\
&\quad \mid \underline{\mathsf{succ}}(y) \Rightarrow y
\end{aligned}
$$

This becomes

$$\mathbf{proc}\ \mathsf{pred}\ (d : \mathsf{nat})\ (x : \mathsf{nat}) = [\![\ldots]\!]\, d$$

where ... is the match expression above. Following the translation we obtain

$$
\begin{aligned}
&\mathbf{cut}\ z : \mathsf{nat} \\
&\quad \mathbf{id}\ z\ x \\
&\mathbf{read}\ z\ \{ \\
&\mid \underline{\mathsf{zero}}(u) \Rightarrow \mathbf{cut}\ w : \mathbf{1} \\
&\qquad\qquad\quad \mathbf{id}\ w\ u \\
&\qquad\qquad \mathbf{write}\ d\ \underline{\mathsf{zero}}(w) \\
&\mid \underline{\mathsf{succ}}(y) \Rightarrow \mathbf{id}\ d\ y \\
&\}
\end{aligned}
$$

Perhaps the only thing noteworthy here is that two of the three uses of the identity seems unnecessary. These are cuts followed by identities.

In proof theory, it has been observed that cut and identities are opposites and cancel each other out. A cut allows us to use a succedent as an antecedent and the identity allows us to use an antecedent as a succedent. These cancellation laws are

$$
\frac{\dfrac{}{A \vdash A}\ \mathsf{id} \qquad \begin{matrix}\mathcal{E} \\ \Delta, A \vdash C\end{matrix}}{\Delta, A \vdash C}\ \mathsf{cut}
\quad \rightsquigarrow \quad
\begin{matrix}\mathcal{E} \\ \Delta, A \vdash C\end{matrix}
$$

$$
\frac{\dfrac{\begin{matrix}\mathcal{D} \\ \Gamma \vdash A\end{matrix} \qquad \dfrac{}{A \vdash A}\ \mathsf{id}}{\Gamma \vdash A}\ \mathsf{cut}}{}
\quad \rightsquigarrow \quad
\begin{matrix}\mathcal{D} \\ \Gamma \vdash A\end{matrix}
$$

Writing out the SAX commands that are assigned to these derivations, we see hidden renamings.

$$\textbf{cut}\ (x : A)\ (\mathsf{id}\ x\ y)\ Q(y) \quad \rightsquigarrow \quad Q(x)$$

$$\textbf{cut}\ (x : A)\ P(x)\ (\mathsf{id}\ y\ x) \quad \rightsquigarrow \quad P(y)$$

It is easy to verify that these optimization not only preserve the types, but also the computational behavior.

One can build these optimizations directly into the translation, or one can design a separate optimization pass that eliminates cut/identity pairs of these two forms. From general principles, we recommend the latter. Keep the translations as simple as possible (and therefore most likely to be correct) and introduce optimizations separately. An advantage of this approach is that it is also easier to measure the impact of an optimization. Our experience is that in the cut/identity optimization are significant. In Lab 2, we may be able to back this up with some numbers.

4 Weak Inversion

From proof search in logical systems we are familiar with the concept of *inversion*. Essentially, a rule is invertible if the premises of a rule are valid whenever the conclusion is. In goal-directed (that is, bottom-up) proof search we can always apply the an invertible rule, reducing the goal of proving a sequent to proving the premises of the rule *without having to consider any alternatives*.

Inversion is mostly formulated in the sequent calculus were all rules are read from the conclusion to the premises. There is also a version for natural deduction which turns out to be an excellent basis for deriving pattern matching that is robust in ways we explain later.

For the positive fragment (which is what we have), we can break down as assumption whenever it is made. Since assumptions are made in each elimination rule we get the following reconstruction of inversion.

$$\frac{\Gamma \vdash A \quad \Delta \vdash A \rhd C}{\Gamma, \Delta \vdash C}\ \mathsf{match}$$

We do not add A to Δ directly, but put it on a "stoop" to be broken down before we return to the usual hypothetical judgment. Already the first rule requires us to generalize our viewpoint:

$$\frac{\Delta \vdash A \cdot B \rhd C}{\Delta \vdash A \otimes B \rhd C}\ \otimes L?$$

We should be able to continue to break down both A and B separately, so instead of a single proposition (which for us will be a type), we have a whole sequence of them. We'll call these Ω. Then we get

$$\frac{\Delta \vdash A \cdot B \cdot \Omega \rhd C}{\Delta \vdash A \otimes B \cdot \Omega \rhd C}\ \otimes L$$

The unit just disappears.

$$\frac{\Delta \vdash \Omega \rhd C}{\Delta \vdash \mathbf{1} \cdot \Omega \rhd C}\ \mathbf{1}L$$

For sums, we get as many premises as there are summands.

$$\frac{(\Delta \vdash A_\ell \rhd C) \quad (\forall \ell \in L)}{\Delta \vdash \oplus\{\ell : A_\ell\}_{\ell \in L} \rhd C}\ \oplus L$$

When the list of possibly invertible propositions is empty, we revert back to the usual judgment.

$$\frac{\Delta \vdash C}{\Delta \vdash (\cdot) \rhd C}\ \mathsf{empty}$$

As given, the system does not work for recursive proposition (think recursive type like nat) because we would have to continue ad infinitum. So we can cut off the inversion phase at any point at our discretion, moving the leftmost proposition into the collection of hypotheses.

$$\frac{\Delta, A \vdash \Omega \rhd C}{\Delta \vdash A \cdot \Omega \rhd C}\ \mathsf{stop}$$

So, inversion is allowed, but not forced, which is why we call this *weak inversion*.

5 Weak Inversion and Pattern Matching

It turns out that allowing (weak) inversion corresponds to allowing nested patterns in match expressions. This has been made explicit and investigated in depth by Zeilberger [2009]. We use a somewhat different notation, more suited to our programming language.

The pattern themselves are symmetric to large values, but can stop at variables.

$$\begin{array}{lllll} \text{Patterns} & p, q & ::= & (p_1, p_2) \mid (\,) \mid k(p) \mid x \\ \text{Pattern Sequences} & \overline{p} & ::= & p \cdot \overline{p} \mid (\cdot) \end{array}$$

We need pattern sequences to match the sequences of types Ω (formerly interpreted as propositions). A single branch in a pattern matching expression then has the form

$$p_1 \cdots p_n \Rightarrow e$$

We check a sequence of such branches against a sequence of types, in the judgment

$$\Delta \vdash A_1 \cdots A_n \rhd (p_1 \cdot p_n \Rightarrow e)^* \Longleftarrow C$$

The $()^*$ here indicates that we have a finite sequence of such branches, where each may have different pattern sequences and expressions, but each pattern sequence must have the same length.

Just as the logical inversion judgment distinguishes cases on A_1, we will do the same, projection out the relevant patterns. We write K for a branch, and K^* for a sequence of branches. filter^* applies the operation to each branch and collects the results.

$$\frac{\Delta \vdash A \cdot B \cdot \Omega \rhd (\mathsf{filter}^*\ (\_,\_)\ K^*) \Longleftarrow C}{\Delta \vdash (A \otimes B) \cdot \Omega \rhd K^* \Longleftarrow C}\ \otimes L$$

where

$$\mathsf{filter}\ (\_,\_)\ ((x, y) \cdot \overline{q} \Rightarrow e) \quad = \quad x \cdot y \cdot \overline{q} \Rightarrow e$$

The filter operation for pairs results in an error in all other cases.

$$\frac{\Delta \vdash \Omega \rhd (\mathsf{filter}^*\ (\,)\ K^*) \Longleftarrow C}{\Delta \vdash \mathbf{1} \cdot \Omega \rhd K^* \Longleftarrow C}\ \mathbf{1}L$$

where

$$\mathsf{filter}\ (\,)\ ((\,)\cdot\overline{q}\Rightarrow e) \quad = \quad \overline{q}\Rightarrow e$$

The filter operation for unit results in an error in all other cases.

Finally, the filter for sums.

$$\frac{(\Delta\vdash A_\ell\cdot\Omega\rhd(\mathsf{filter}^*\ (\ell(\_))\ K^*)\Longleftarrow C)\quad(\forall\ell\in L)}{\Delta\vdash\oplus\{\ell:A_\ell\}_{\ell\in L}\cdot\Omega\rhd K^*\Longleftarrow C}\ \oplus L$$

where each application of $\mathsf{filter}^*\ \ell(\_)$ returns either a single branch or none. These are then concatenated.

$$\begin{array}{lcll}\mathsf{filter}\ \ell(\_)\ (k(p)\cdot\overline{q}\Rightarrow e) & = & p\cdot\overline{q}\Rightarrow e & \text{for } \ell=k\\ \mathsf{filter}\ \ell(\_)\ (k(p)\cdot\overline{q}\Rightarrow e) & = & (\cdot) & \text{for } \ell\neq k\end{array}$$

The filter operation for labeled patterns results in an error in all other cases.

Filtering based on sums is quite lenient in the sense that extraneous branches in the pattern match are ignored. Based on the types of the subject of the match, these are unreachable and therefore ignoring them will not lead to a runtime error. On the other hand, it might seem to be a likely error if the programmer has written extraneous branches. A brief further remark on that in the final section of this lecture's notes. When we reach the empty sequence of types, we also must reach the empty sequence of patterns, and revert back to our usual typing rules. Considering we are in the bidirectional system, this is a checking judgment.

$$\frac{\Delta\vdash e\Longleftarrow C}{\Delta\vdash(\cdot)\rhd(\cdot)\Rightarrow e\Longleftarrow C}\ \mathsf{empty}$$

It is an error if the empty sequence of types is not matched by an empty sequence of patterns. Also note that there can only be a single branch in order to avoid nondeterminism (or redundant patterns, however, one wants to look at it). So we are less lenient here as in the case of extranous patterns.

Finally, we come to variable patterns which represent a kind of special case because they are not driven by the structure of the type.

$$\frac{\Delta,x:A\vdash\Omega\rhd\mathsf{filter}^*\ x\ K^*\Longleftarrow C\quad(x\ \mathsf{fresh})}{\Delta\vdash A\cdot\Omega\rhd K^*\Longleftarrow C}\ \mathsf{stop}$$

where

$$\mathsf{filter}\ x\ (y\cdot\overline{q}\Rightarrow e(y)) \quad = \quad \overline{q}\Rightarrow e(x)$$

Note that the bound variable name y could be different from x, but that we substitute the fresh name x for y. There may be multiple such branches (which are disambiguated later based on Ω as it matches $\overline{q}$), but the pattern in each of them must be a variable.

Overall, this form of pattern matching has a distinct left-to-right flavor in the way it analyzes the match. Moreover, it does not allow some catch-all patterns that can often be useful. The justification for this is three-fold. First, this form of nested checking of patterns can be compiled without contortions to the one-level-at-a-time matching provided by **read** commands in SAX. Second, when patterns are matched sequentially (including catch-alls for patterns as yet unmatched), assessing linearity of the resulting code is less clear. Finally, under the parallel semantics permitted by SAX, it matters whether cells are read. For example, the two simple matches

match a **with** $\underline{\mathsf{zero}}(u) \Rightarrow \underline{\mathsf{zero}}(u)$
match a **with** $\underline{\mathsf{zero}}(\,) \Rightarrow \underline{\mathsf{zero}}(\,)$

have different behaviors! The first can proceed as soon as a small value $\underline{\mathsf{zero}}(b)$ is written into the cell with address a, while the second must also wait for the unit value to be written into b.

At some later point in the course we might consider more general forms of pattern matching. Some investigations in this direction have been made by Stock [2020]. For Lab 2, when you implement nested pattern matching, we are lenient as explained above. We do not write out how the definition of projection carries over to compilation. This could be done either as a translation from ND to ND, eliminating nested pattern matches, followed by the translation we gave, or it could be built into a generalized translation from ND to SAX.

6 Subtyping and Intersection Types

We did not cover this in lecture, but one reason one might want to consider allowing extraneous branches is to have nice properties for subtyping. In generally, we would want the following metatheorems to be true:

1. If $\Gamma \vdash e : A$ and $A \leq B$ then $\Gamma \vdash e : B$
2. If $A \leq B$ and $\Gamma, x : B \vdash e : C$ then $\Gamma, x : A \vdash e : C$

The first just transports to expressions the idea that any large value of type A should also have to B if $A \leq B$. The second exploits the same definition, but in reverse because it is used on variables that stand for values. However, the second part will be false if we do not allow extraneous branches. For example, with

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

we have pos $\leq$ nat. The expression

```
match x with (
| 'zero() => 'zero()
| 'succ(y) => y
)
```

checks with x : nat. But unless we allow extraneous branches, it would not check at x : pos, even though pos $\leq$ nat.

More generally, we might want to assign more than one possible type to a given definition. For example,

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

decl pred (x : nat) : nat
decl pred (x : pos) : nat
decl pred (x : zero) : zero
```

```
defn pred x = match x with (
| 'zero() => 'zero()
| 'succ(y) => y
)
```

When checking the second type, we do not consider the zero branch, when checking the third type, we do not consider the succ branch. And yet, it is perfectly valid to assign all three types to the given definition of pred.

Assigning multiple types to the same expression is the domain of *intersection types*. If they are considered only at the top level as here, we do not form any explicit intersections, but some of the issues and solutions remain the same.

References

Benedikt Stock. General pattern matching for session-typed concurrent programs. Bachelor Thesis, Jacobs University, Bremen, Germany, May 2020.

Noam Zeilberger. *The Logical Basis of Evaluation Order and Pattern-Matching*. PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2009. Available as Technical Report CMU-CS-09-122.