Lecture Notes on Negative Types

15-417/817: HOT Compilation
Frank Pfenning

Lecture 7
February 4, 2025

1 Introduction

So far, our language has been entirely "first-order", that is, we could not pass functions as arguments or return them from functions or store them in pairs, etc. As one of the teaching assistants put it: "*How much longer can we call this course **Higher-Order** Typed Compilation without higher-order functions*." In today's lectures we complete the picture of type constructors by introducing negative types for functions $A \to B$ and lazy records $\&\{\ell : A_\ell\}_{\ell \in L}$ (also known as *objects*). This is also a good opportunity to revisit the key components of our development such as the ND and Sax languages, their type systems, their dynamics, and their compilation.

2 Function Types in ND

The most obvious extension towards a higher-order language is to introduce general function types $A \to B$. For the moment, such functions will remains *linear*, so a more explicit notation would be $A \multimap B$. However, within a few lectures we will move on to a language in which linear, affine, and other forms of functions coexist, so we stick with $A \to B$ as a unifying notation.

The obvious way to introduce a function is via λ-abstraction, the obvious way to eliminate is via application.

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.\, e : A \to B}\ {\to}I \qquad \frac{\Gamma \vdash e_1 : A \to B \quad \Delta \vdash e_2 : A}{\Gamma\,;\,\Delta \vdash e_1\, e_2 : B}\ {\to}E$$

In the case for application we need to split the context to account for linearity.

We went from this to an algorithm for type-checking in two steps: in the first step we make it bidirectional; in the second step we generate additive output contexts of variables actually used.

For the bidirectional rules, we need to remember that a hypothesis $x : A$ is, bidirectionally, a shorthand for $x \Longrightarrow A$. Then:

$$\frac{\Gamma, x : A \vdash e \Longleftarrow B}{\Gamma \vdash \lambda x.\, e \Longleftarrow A \to B}\ {\to}I \qquad \frac{\Gamma \vdash e_1 \Longrightarrow A \to B \quad \Delta \vdash e_2 \Longleftarrow A}{\Gamma\,;\,\Delta \vdash e_1\, e_2 \Longrightarrow B}\ {\to}E$$

As expected, the each judgment reverses direction between the introduction and elimination rules. Let's quickly verify the elimination rules, making sure all information is available when it

needs to be. We show everything that is known at each stage during type-checking in green.

$$\frac{\Gamma \vdash e_1 \Longrightarrow A \to B \quad \Delta \vdash e_2 \Longleftarrow A}{\Gamma\ ;\ \Delta \vdash e_1\ e_2 \Longrightarrow B}\ {\to}E$$

For now, we assume we can calculate Γ and Δ from $\Gamma\ ;\ \Delta$, even though this will not actually be realized until we add output contexts. Also, since $e_1\ e_2$ is known, so are e_1 and e_2.

$$\frac{\Gamma \vdash e_1 \Longrightarrow A \to B \quad \Delta \vdash e_2 \Longleftarrow A}{\Gamma\ ;\ \Delta \vdash e_1\ e_2 \Longrightarrow B}\ {\to}E$$

At this point we cannot actually check e_2 against A, because we don't know what A is (yet). But we can synthesize a type for e_1 instead, which (if type-correct) gives us $A \to B$.

$$\frac{\Gamma \vdash e_1 \Longrightarrow A \to B \quad \Delta \vdash e_2 \Longleftarrow A}{\Gamma\ ;\ \Delta \vdash e_1\ e_2 \Longrightarrow B}\ {\to}E$$

From this we can extract A and B, the first which is needed to check e_2, and the second one is what the application $e_1\ e_2$ synthesizes.

$$\frac{\Gamma \vdash e_1 \Longrightarrow A \to B \quad \Delta \vdash e_2 \Longleftarrow A}{\Gamma\ ;\ \Delta \vdash e_1\ e_2 \Longrightarrow B}\ {\to}E$$

Using an input context containing all variables that are lexically in scope we can refine this into

$$\frac{\Gamma, x : A \vdash e \Longleftarrow B\ /\ \Xi}{\Gamma \vdash \lambda x.\, e \Longleftarrow A \to B\ /\ (\Xi \setminus x)}\ {\to}I \qquad \frac{\Gamma \vdash e_1 \Longrightarrow A \to B\ /\ \Xi_1 \quad \Gamma \vdash e_2 \Longleftarrow A\ /\ \Xi_2}{\Gamma \vdash e_1\ e_2 \Longrightarrow B\ /\ (\Xi_1\ ;\ \Xi_2)}\ {\to}E$$

One fundamental difference between values of positive and negative type is that the former can be explicitly observed, while the latter can only be interacted with. For example, functions are *compiled* and their structure may be lost, but they can be applied to obtain an observable result. This is true in all main-stream functional languages, in part because it supports modularity and allows the compiler to generate efficient code without having to maintain a source representation. The λ-calculus [Church and Rosser, 1936] was not developed, however, with such a distinction in mind. It emerged only later, through the study of proof theory and its connection to programming (see, for examples, Levy's call-by-push-value [Levy, 2006]). Levy's analysis also explains why functions here evaluate their arguments, unlike the notion of conversion underlying the original λ-calculus.

In the dynamics, functions appear as a new kind of value, $\lambda x.\, e$ which itself does not contain other values but general expressions. Evaluation then becomes

$$\frac{}{\lambda x.\, e \hookrightarrow \lambda x.\, e} \qquad \frac{e_1 \hookrightarrow \lambda x.\, e(x) \quad e_2 \hookrightarrow V_2 \quad e(V_2) \hookrightarrow V}{e_1\ e_2 \hookrightarrow V}$$

In the next lecture, when we discuss closures, we will have a chance to revisit this rules and its consequences.

3 Lazy Records

Lazy records, which can be used to implement some forms of objects albeit without all the affordances of object-oriented languages, is another new form of type, $\&\{\ell : A_\ell\}_{\ell \in L}$ somehow symmetric to sums. This type is inhabited by *values* $\{\ell \Rightarrow e_\ell\}_{\ell \in L}$ that cannot be directly observed. While we apply functions to arguments, we instead project fields from lazy records writing $e.k$ for a label k. The byword "lazy" is to suggest that it contains arbitrary expressions, not values. As for sums, we require the index set L to be nonempty. This is not a fundamental necessity, but a simplifying restriction.

Leading with the dynamics for a change:

$$\frac{}{\{\ell \Rightarrow e_\ell\}_{\ell \in L} \hookrightarrow \{\ell \Rightarrow e_\ell\}_{\ell \in L}} \qquad \frac{e \hookrightarrow \{\ell \Rightarrow e_\ell\}_{\ell \in L} \quad e_k \hookrightarrow V \quad (k \in L)}{e.k \hookrightarrow V}$$

The typing rules combine aspects of sums and functions.

$$\frac{(\Gamma \vdash e_\ell : A_\ell) \quad (\forall \ell \in L)}{\Gamma \vdash \{\ell \Rightarrow e_\ell\}_{\ell \in L} : \&\{\ell : A_\ell\}_{\ell \in L}}\ \&I \qquad \frac{\Gamma \vdash e : \&\{\ell : A_\ell\}_{\ell \in L} \quad (k \in L)}{\Gamma \vdash e.k : A_k}\ \&E$$

Note that for the introduction rule, all premises must check with the same context Γ. That's because due to linearity, at runtime exactly one branch will be projected out, and whichever branch it is must be well-typed in Γ.

We skip the purely bidirectional intermediate step and go directly to the version with output contexts.

$$\frac{(\Gamma \vdash e_\ell \Longleftarrow A_\ell \mathbin{/} \Xi) \quad (\forall \ell \in L)}{\Gamma \vdash \{\ell \Rightarrow e_\ell\}_{\ell \in L} \Longleftarrow \&\{\ell : A_\ell\}_{\ell \in L} \mathbin{/} \Xi}\ \&I \qquad \frac{\Gamma \vdash e \Longrightarrow \&\{\ell : A_\ell\}_{\ell \in L} \mathbin{/} \Xi \quad (k \in L)}{\Gamma \vdash e.k \Longrightarrow A_k \mathbin{/} \Xi}\ \&E$$

Note that exactly the same variables Ξ must be used in each field of the record. When we move away from the purely linear set-up, this will require some changes.

4 Three Small Examples

First, turning what typically might be defined as a metavariable

```
defn proj1 (x : nat) (y : nat) : nat
```

into a first-class function. One of the differences is that we can partially apply a first-class function, that is, `proj1 ('zero()) : nat -> nat`, which will build a so-called *closure* at runtime. This does not apply to metavariables that are always used with all arguments (that is, a full substitution).

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

defn destroy (x : nat) : 1 = match x with
| 'zero() => ()
| 'succ(y) => destroy y
end

defn proj1 : nat -> nat -> nat = fun x => fun y =>
    match destroy y with | () => x end
```

The second example is a (linear) counter object holding a natural number that can receive increment and decrement messages and responds with `'none()` if the number cannot be decremented (also terminating the object in the process).

```
type ctr = &{'inc : ctr,
             'dec : +{'none : 1, 'some : ctr}}

defn counter (v : nat) = record
| 'inc => counter ('succ v)
| 'dec => match v with
          | 'zero() => 'none()
          | 'succ(w) => 'some(counter w)
          end
end
```

Third, an example that combines features of functions and objects: a store. Even if not expressed in the type, this particular implementation of a store constitutes a stack.

```
type store = &{'ins : nat -> store,
               'del : +{'none : 1, 'some : nat * store}}

defn empty : store = record
| 'ins => fun x => elem x empty
| 'del => 'none()
end

defn elem (x : nat) (s : store) : store = record
| 'ins => fun y => elem y (elem x s)
| 'del => 'some (x, s)
end
```

5 Subtyping

Before moving on from ND to Sax, we consider subtyping because negatives create some new phenomena. Recall what we consider the defining properties of subtyping: we can consider an expression to have a larger type without change, and we can consider a variable to have a smaller type, again without change.

Right subsumption: If $\Gamma \vdash e : A$ and $A \leq B$ then $\Gamma \vdash e : B$

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

As a typical example consider

```
type pos = +{'succ : nat}
```

If we know $e : pos$, we can forget this detailed knowledge and treat the expression as if $e : nat$. Conversely, if $x : nat \vdash e : C$ and then $x : pos \vdash e : C$ should also be okay—e just doesn't take advantage of this more precise information.

Recall that the subtyping rules are interpreted *coinductively*, that is, infinite derivations are allowed. Based on these considerations, we get

$$\frac{B_1 \leq A_1 \quad A_2 \leq B_2}{A_1 \to A_2 \leq B_1 \to B_2}$$

For example, a function $nat \to pos$ is also a function $pos \to nat$. We say function types are *contravariant* in their argument and *covariant* in their result.

For lazy records, if

$$\Gamma \vdash e : \&\{\ell : A_\ell\}_{\ell \in L}$$

then we can project only any field $\ell \in L$. If $L \supseteq K$, then it also holds that

$$\Gamma \vdash e : \&\{k : A_k\}_{k \in K}$$

we just don't know about any hidden fields in $L - K$ that are inaccessible. The upshot is the rule

$$\frac{(L \supseteq K) \quad (A_k \leq B_k) \quad (\forall k \in K)}{\&\{\ell : A_\ell\}_{\ell \in L} \leq \&\{k : B_k\}_{k \in K}}$$

It also means that if we check a record against a type, we should allow additional fields in the record. Otherwise, the fundamental property of subtyping would fail.

$$\frac{(L \supseteq K) \quad (\Gamma \vdash e_k \Longleftarrow A_k \;/\; \Xi) \quad (\forall k \in K)}{\Gamma \vdash \{\ell \Rightarrow e_\ell\}_{\ell \in L} \Longleftarrow \&\{k : A_k\}_{k \in K} \;/\; \Xi}\ \&E$$

In summary, large values for ND are now:

$$\begin{array}{rrcll} \text{Large values} & V & ::= & (V_1, V_2) \mid ( ) \mid k(V) & \text{(positive)} \\ & & \mid & \lambda x.\, e \mid \{\ell \Rightarrow e_\ell\}_{\ell \in L} & \text{(negative)} \end{array}$$

6 Extending Sax

We also have to extend Sax to include the new types. This actually reveals some symmetries that are hidden in ND. Let's remember how this work for pairs. On the logical side, we took the noninvertible rules and turned them into axioms. We did this so we could type cells. From the sequent calculus (which we didn't discuss in depth), we know the right rule is not invertible but the left rule is. A small example demonstrating this is the proof of $B \otimes A \vdash A \otimes B$. We also have to remember that in the process assignment we always read from the variables on left and write to the variable on the right (which is the *destination* of the computation). For these typing rules, we do not take subtyping into account, but that can be incorporated into the system following our prior approach.

$$\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma\,;\,\Delta \vdash A \otimes B}\otimes R \quad \rightsquigarrow \quad \frac{}{A, B \vdash A \otimes B}\otimes X \quad \rightsquigarrow \quad \frac{}{a : A, b : B \vdash \mathbf{write}\ c\ (a, b) :: (c : A \otimes B)}\otimes X$$

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

For negative types, the properties are reversed: the right rule is invertible while the left rule is not. So we start with the parts below, leaving ?? where we have to pause and think.

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\to R \quad \rightsquigarrow \quad \text{(unchanged)} \quad \rightsquigarrow \quad \frac{\Gamma, x : A \vdash P :: (y : B)}{\Gamma \vdash \mathbf{write}\ c\ ?? :: (c : A \to B)}\to R$$

$$\frac{\Gamma \vdash A \quad \Delta, B \vdash \delta}{\Gamma\,;\,\Delta, A \to B \vdash \delta}\to L \quad \rightsquigarrow \quad \frac{}{A, A \to B \vdash B}\to X \quad \rightsquigarrow \quad \frac{}{a : A, c : A \to B \vdash \mathbf{read}\ c\ ?? :: (b : B)}\to X$$

Let's consider $\to R$ first. Looking at the premise, P is a process reading from x and writing to y. They are both fresh variables in the premise, so we have

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

So we write a continuation $(x, y) \Rightarrow P$ to a cell. When reading from such a cell, we need to pass it the function argument (as x) and destination (as y). So:

$$\frac{}{a : A, c : A \to B \vdash \mathbf{read}\ c\ (a, b) :: (b : B)}\ \to X$$

Looking back, we already have a pair of addresses as a small value, and a continuation $(x, y) \Rightarrow P$ in our language, we are just using them in new ways. In particular, we now store this form of continuation in a memory cell, and we pass a pair to a continuation we have read.

So we generalize Sax from small values and continuations to the encompassing concept of a storable. The fact that writing a continuation into a memory cell is unrealistic, which is why we introduce *closure* and *closure conversion* in the next lecture. Even though we haven't discussed lazy records yet, we extrapolate from the other constructors and then illustrates how it comes out. We also mention $\bot$ (which is dual to $\mathbf{1}$), but which requires a judgment for processes with an empty succedent. Such processes cannot write an answer, so while there are circumstances where they may be useful we elide them from the language (at least for now).

Commands	P, Q	$::=$	$\mathbf{write}\ c\ S$	
		$\mid$	$\mathbf{read}\ c\ S$	
		$\mid$	$\mathbf{cut}\ (x : A)\ P\ Q$	
		$\mid$	$\mathbf{id}\ a\ b$	
		$\mid$	$\mathbf{call}\ F\ d\ b_1\ \ldots\ b_n$	
Storables	S	$::=$	$v \mid K$	
Small values	v	$::=$	(a, b)	$(\otimes, \to)$
		$\mid$	$()$	$(\mathbf{1}, [\bot])$
		$\mid$	$k(a)$	$(\oplus, \&)$
Continuations	K	$::=$	$(x, y) \Rightarrow P$	$(\otimes, \to)$
		$\mid$	$() \Rightarrow P$	$(\mathbf{1}, [\bot])$
		$\mid$	$\{\ell(x_\ell) \Rightarrow P_\ell\}_{\ell \in L}$	$(\oplus, with)$

Passing a small value to a continuation is now a more symmetric relationship, but the essence is the same as before.

$$v \bowtie K = K \bowtie v = v \rhd K$$

where

$$\begin{array}{lcllll} (a, b) & \rhd & (x, y) \Rightarrow P(x, y) & = & P(a, b) & \\ () & \rhd & () \Rightarrow P & = & P & \\ k(a) & \rhd & \{\ell(x_\ell) \Rightarrow P_\ell(x_\ell)\}_{\ell \in L} & = & P_k(a) & (k \in L) \end{array}$$

and

$$\begin{array}{lcll} \mathsf{proc}\ (\mathbf{write}\ c\ S) & \longrightarrow & \mathsf{cell}\ c\ S & \\ \mathsf{cell}\ c\ S, \mathsf{proc}\ (\mathbf{read}\ c\ S') & \longrightarrow & \mathsf{proc}\ (S \bowtie S') & \\ \mathsf{proc}\ (\mathbf{cut}\ (x : A)\ P(x)\ Q(x)) & \longrightarrow & \mathsf{proc}\ P(a), \mathsf{proc}\ Q(a) & (a\ \text{fresh}) \\ \mathsf{cell}\ b\ S, \mathbf{proc}\ (\mathbf{id}\ a\ b) & \longrightarrow & \mathsf{cell}\ a\ S & \\ \mathbf{proc}\ (\mathbf{call}\ F\ c\ \overline{b}) & \longrightarrow & \mathsf{proc}\ P(c, \overline{b}) & \text{where } F(x, \overline{y}) = P(x, \overline{y}) \end{array}$$

Now we finally explain lazy records. We realize we cannot just project a record onto a field as in ND with $.k$ because the expression in each fields requires a destination! That is, we have to project instead with $.k(a)$, where a is the destination. But that's like selecting from a sum, except a is a destination. Putting these obervations together (and cross-checking with the intuitions of the dualities conjecture above), we obtain:

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

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

7 Compilation

With all this background on typing and the dynamics, compilation is an afterthought, complicated only later when we need to address closure conversion.

$$\begin{array}{lcl}
[\![\lambda x.\, e]\!]\, d & = & \mathbf{write}\ d\ ((x, y) \Rightarrow [\![e]\!]\, y) \\
[\![e_1\ e_2]\!]\, d & = & \mathbf{cut}\ x_1 \\
 & & \quad [\![e_1]\!]\, x_1 \\
 & & \mathbf{cut}\ x_2 \\
 & & \quad [\![e_2]\!]\, x_2 \\
 & & \mathbf{read}\ x_1\ (x_2, d) \\
 & & \\
[\![\{\ell \Rightarrow e_\ell\}_{\ell \in L}]\!]\, d & = & \mathbf{write}\ d\ \{\ell(x_\ell) \Rightarrow [\![e_\ell]\!]\, x_\ell\}_{\ell \in L} \\
[\![e.k]\!]\, d & = & \mathbf{cut}\ x \\
 & & \quad [\![e]\!]\, x \\
 & & \mathbf{read}\ x\ k(d)
\end{array}$$

References

Alonzo Church and J.B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, May 1936.

Paul Blain Levy. Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation*, 19(4):377–414, 2006.