Lecture Notes on Closure Conversion

15-417/817: HOT Compilation
Frank Pfenning

Lecture 9
February 11, 2025

1 Introduction

In the previous lecture we introduced closures into the dynamics of ND and Sax. At a lower level, we'd like a closure to consist of an environment and a code pointer. How do we get there? There are two steps: the first step is to lift all closures to the top-level, in the form of metavariables similar to those making up definitions. The second step is to build the tuples making up the environment at runtime and store a pair with the address of the environment and the code pointer to memory. In this lecture we will talk about the first step: lifting closures to the top level.

2 Lifting Negative Expressions

Let's consider a simple definition.

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

defn rev (l : list) (acc : list) : list =
match l with
| 'nil() => acc
| 'cons(hd, tl) => rev tl ('cons (hd, acc))
end
```

This is perfectly fine as a definition, and in ND probably the right one. But we could also just take a list and return a function!

```
defn revfun (l : list) : list -> list =
match l with
| 'nil() => fun acc => acc
| 'cons(hd, tl) => fun acc => revfun tl ('cons (hd, acc))
end
```

We see that `revfun` contains two function constructors, one in each branch. The first one is closed, the second one has free variables `hd` and `tl`. Closure conversion generates a new definition for each. In general, every constructor for values of negative type should be lifted to the top level, creating a new definition.

```
defn rev_ (l : list) : list -> list =
match l with
```

```
  | 'nil() => revfun2_1
  | 'cons(hd, tl) => revfun2_2 tl hd
  end

  defn rev_1 : list -> list =
    fun acc => acc

  defn rev_2 (tl : list) (hd : nat) : list -> list =
    fun acc => rev_ tl ('cons (hd, acc))
```

After this transformation, all constructors for negative types are first in the definition of a metavariable.

A pleasant property of this transformation is that this revised program will execute exactly as the original program using the dynamics defined in the last lecture. So we haven't done much except to lift all λ-abstractions to the top level.

A similar transformation applies to lazy records. Let's consider a simple store interface from Lecture 7.

```
  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
```

The `empty` and `elem` definitions are already of the correct form at the top level, since they start with a **`record`** constructor. Embedded are two functions, so new definitions have to be created for them.

```
  defn empty_ : store =
  record
  | 'ins => empty_1
  | 'del => 'none()
  end

  defn empty_1 : nat -> store =
    fun x => elem_ x empty_

  defn elem_ (x : nat) (s : store) : store =
  record
  | 'ins => elem_1 x s
  | 'del => 'some (x, s)
  end

  defn elem_1 (x : nat) (s : store) : nat -> store =
    fun y => elem_ y (elem_ x s)
```

3 A Type-Directed Transformation

When we lift a function or record constructor to the top level, as a meta-variable, we need to know the free variables in an expression so we can parametrize the new definition by it. But this is exactly what the output context Ξ in

$$\Gamma \vdash e \Longleftrightarrow A \,/\, \Xi$$

tracks: the variables used in e! So we should instrument these two judgments with a transformed expression e' in which negative constructors (`fun`, `record`) have been replaced by calls to new metavariables. It's quite tedious to write all this as an inference system, since besides the transformed expression e we also need to collect the new top-level definitions. So we indicate the latter part informally. The judgment then becomes

$$\Gamma \vdash (e \rightsquigarrow e') \Longleftrightarrow A \,/\, \Xi$$

where e' is the result of applying closure conversion to e.

First, the rules for positives are updated systematically, just rebuilding the expression after translation.

$$\frac{}{\Gamma \vdash (\,) \rightsquigarrow (\,) \Longleftarrow \mathbf{1} \,/\, \cdot}\ \mathbf{1}I$$

$$\frac{\Gamma \vdash s \rightsquigarrow s' \Longrightarrow \mathbf{1} \,/\, \Xi_1 \quad \Gamma \vdash e \rightsquigarrow e' \Longleftarrow C \,/\, \Xi_2}{\Gamma \vdash \mathbf{match}\ s\ \mathbf{with}\ (\,) \Rightarrow e \rightsquigarrow \mathbf{match}\ s'\ \mathbf{with}\ (\,) \Rightarrow e' \Longleftarrow C \,/\, \Xi_1 \,;\, \Xi_2}\ \mathbf{1}E$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' \Longleftarrow A_1 \,/\, \Xi_1 \quad \Gamma \vdash e_2 \rightsquigarrow e_2' \Longleftarrow A_2 \,/\, \Xi_2}{\Gamma \vdash (e_1, e_2) \rightsquigarrow (e_1', e_2') \Longleftarrow A_1 \otimes A_2 \,/\, \Xi_1 \,;\, \Xi_2}\ \otimes I$$

$$\frac{\Gamma \vdash s \rightsquigarrow s' \Longrightarrow A_1 \otimes A_2 \,/\, \Xi_1 \quad \Gamma, x_1 : A_1, x_2 : A_2 \vdash e \rightsquigarrow e' \Longleftarrow C \,/\, \Xi_2}{\Gamma \vdash \mathbf{match}\ s\ \mathbf{with}\ (x_1, x_2) \Rightarrow e \rightsquigarrow \mathbf{match}\ s'\ \mathbf{with}\ (x_1, x_2) \Rightarrow e' \Longleftarrow C \,/\, \Xi_1 \,;\, ((\Xi_2 \setminus x_1) \setminus x_2)}\ \otimes E$$

$$\frac{(k \in L) \quad \Gamma \vdash e \rightsquigarrow e' \Longleftarrow A_k \,/\, \Xi}{\Gamma \vdash k(e) \rightsquigarrow k(e') \Longleftarrow \oplus\{\ell : A_\ell\}_{\ell \in L} \,/\, \Xi}\ \oplus I$$

$$\frac{\begin{array}{c}\Gamma \vdash s \rightsquigarrow s' \Longrightarrow \oplus\{\ell : A_\ell\}_{\ell \in L} \,/\, \Xi_1 \\ (\Gamma, x_\ell : A_\ell \vdash e_\ell \rightsquigarrow e_\ell' \Longleftarrow C \,/\, \Xi_\ell \qquad \Xi_\ell \setminus x_\ell = \Xi_2) \qquad (\forall \ell \in L)\end{array}}{\Gamma \vdash \mathbf{match}\ s\ \mathbf{with}\ \{\ell(x_\ell) \Rightarrow e_\ell\}_{\ell \in L} \rightsquigarrow \mathbf{match}\ s'\ \mathbf{with}\ \{\ell(x_\ell) \Rightarrow e_\ell'\}_{\ell \in L} \Longleftarrow C \,/\, \Xi_1 \,;\, \Xi_2}\ \oplus E$$

$$\frac{}{\Gamma \vdash x \rightsquigarrow x \Longrightarrow A \,/\, x : A}\ \mathsf{var} \qquad \frac{\Gamma \vdash e \rightsquigarrow e' \Longrightarrow A' \,/\, \Xi \quad A' \leq A}{\Gamma \vdash e \rightsquigarrow e' \Longleftarrow A \,/\, \Xi}\ \Rightarrow/\Leftarrow$$

The first interesting rule concerns the function constructor.

$$\frac{\Gamma, x : A \vdash e \Longleftarrow B \,/\, \Xi}{\Gamma \vdash \lambda x.\, e \Longleftarrow A \to B \,/\, (\Xi \setminus x)}\ \to I$$

We have to transform this into a top-level definition which is then used. Here, $F[\Xi] : A \to B = \lambda x.\, e'$ is the new definition added to the top-level signature. The substitution $\mathbf{id}_\Xi$ is the identity

substitution on Ξ, substituting each variable in Ξ for itself.

$$\frac{\Gamma, x : A \vdash e \rightsquigarrow e' \Longleftarrow B \;/\; \Xi' \quad \Xi = \Xi' \setminus x \quad F[\Xi] : A \to B = \lambda x.\, e' \quad F \text{ fresh}}{\Gamma \vdash \lambda x.\, e \rightsquigarrow F[\mathbf{id}_\Xi] \Longleftarrow A \to B \;/\; \Xi} \to I$$

Notice how we exploit the fact that Ξ contains exactly the variables that occur free in $\lambda x.\, e$. This is actually independent from whether the variables are treated linearly or not, which means our approach will easily scale to a nonlinear language.

Function application does not change: in $e_1\, e_2$, e_1 will evaluate to a closure and application proceeds as laid out in the last lecture.

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' \Longrightarrow A \to B \;/\; \Xi_1 \quad \Gamma \vdash e_2 \rightsquigarrow e_2' \Longleftarrow A \;/\; \Xi_2}{\Gamma \vdash e_1\, e_2 \rightsquigarrow e_1'\, e_2' \Longrightarrow B \;/\; (\Xi_1 \,;\, \Xi_2)} \to E$$

Lazy records do not introduce any new ideas. The change is again at the constructor, where a new top-level definition is constructed.

$$\frac{(\Gamma \vdash e_\ell \rightsquigarrow e_\ell' \Longleftarrow A_\ell \;/\; \Xi) \quad (\forall \ell \in L) \quad F[\Xi] : \&\{\ell : A_\ell\}_{\ell \in L} = \{\ell \Rightarrow e_\ell'\}_{\ell \in L} \quad F \text{ fresh}}{\Gamma \vdash \{\ell \Rightarrow e_\ell\}_{\ell \in L} \rightsquigarrow F[\mathbf{id}_\Xi] \Longleftarrow \&\{\ell : A_\ell\}_{\ell \in L} \;/\; \Xi} \,\&I$$

$$\frac{\Gamma \vdash e \rightsquigarrow e' \Longrightarrow \&\{\ell : A_\ell\}_{\ell \in L} \;/\; \Xi \quad (k \in L)}{\Gamma \vdash e.k \rightsquigarrow e'.k \Longrightarrow A_k \;/\; \Xi} \,\&E$$

You should convince yourself that in these rules, neither the type of the expression nor the output context Ξ changes. This could be expressed as the following theorem, which follows by rule induction over the given derivation.

> *If* $\Gamma \vdash e \Longleftrightarrow A \;/\; \Xi$ *then there exists an* e' *such that* $\Gamma \vdash e \rightsquigarrow e' \Longleftrightarrow A \;/\; \Xi$ *and* $\Gamma \vdash e' \Longleftrightarrow A \;/\; \Xi$. *Moreover, modulo the names of new metavariables,* e' *is unique.*

We can also show that e and e' compute the same observable output, typically via a kind of bisimulation argument between the traces.

4 Executing Programs After Closure Conversion

If we take a new definition $F[\Xi] : A \to B = \lambda x.\, e$ or $F[\Xi] : \&\{\ell : A_\ell\}_{\ell \in L}$, then calling this definition with a substitution σ will immediately build a closure. That is

$$\eta \vdash F[\mathbf{id}_\Xi] \hookrightarrow \langle \eta, F[\mathbf{id}_\Xi] \rangle$$

This, however, might be considered a bit wasteful because η may have all variables lexically in scope, while only the variables in $\mathbf{id}_\Xi$ matter. So we could rewrite this as

$$\eta \vdash F[\mathbf{id}_\Xi] \hookrightarrow F\langle \eta|_\Xi \rangle$$

where $\eta|_\Xi$ is the restriction of η to the variables in Ξ.

This use of F on the right-hand side looks to be related to contextual modal possibility [Nanevski et al., 2008, Section 9.1], except that we would need to change application to "unwrap" the contextual modality. This seems in line with the observation by Minamide et al. [1996] that full closure conversion (including application) can be explained via existential types. However, their approach does not a priori enforce that closures really are closed, only that they are existentially typed. Perhaps a combination of the ideas could provide a more complete explanation for closures, but we do not pursue this here.

5 Closures in Sax

We have already seen environments and closures in Sax in the last lecture. The change to Sax is analogous to the change in ND: negative storables (that is, continuations K) are restricted to be the first constructs on right-hand sides of definitions. A further step would be to model computation and storage at a lower level of abstraction, but we leave this to a future lecture or miniproject.

6 Optimizations

The most immediate optimization is to allow a top-level definition to start with a whole sequence of constructors of negative type. This avoids the overhead of building intermediate closures for every single constructor. At a call site for for a closure definition we would than have to select the appropriate instance, expecting and processing multiple arguments. This allows the functional idiom of "partial application", eliminating much of the overhead for a full application.

Beyond that, control flow analysis for whole program optimization seems to be the most important optimization [Cejtin et al., 2000].

References

Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In Gert Smolka, editor, *9th European Symposium on Programming (ESOP 2000)*, pages 56–71, Berlin, Germany, March 2000. Springer LNCS 1782.

Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. Typed closure conversion. In *23rd Symposium on Principles of Programming Languages (POPL 1996)*, pages 271–283. ACM, January 1996.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *Transactions on Computational Logic*, 9(3), 2008.