CLOSURE CONVERSION

Abstract. To this point we've been working with functions that have free variables, like $\lambda x.\ f\ g\ x$. In the theory, instantiation of free variables is implemented by substitution, but hardware does not support this operation. Thus we now define a language, IL-Closure, in which terms explicitly carry environments defining their free variables, "closing" over them. We also address the techniques used to close over free type variables, a topic that traditionally required some pretty advanced type machinery. However, due to a clever trick in our definition of CPS conversion, we are able to elide most of the complexity of closing over type variables.

1. Motivation

Why do we need to closure convert? In a lambda expression

$$\lambda x.\ e$$

e may "capture" free variables defined in the enclosing environment, but this is actually true of any expression in our language. The key is that, because the lambda abstraction suspends the evaluation of e, e may later be evaluated in an environment in which those enclosing variables are no longer in scope. For example, in

$$f = \lambda x.\ \lambda y.\ x$$

if we define

$$g = f\ n$$

then the function g outlives the scope of the variable x to which it refers. If our target platform implemented variable substitution, we would solve this problem by dynamically substituting n for the variable x when f is applied; but the hardware executing our code probably does *not* implement substitution, so we need some other implementation to express this dynamic variable capture. The canonical solution is to create a closure, which extends each lambda abstraction with an additional argument, called the "environment", which explicitly binds each variable captured by the function. Then, the closed function is paired with a mapping from those variables to the values they refer to.

2. Efficient well-typed translations

Consider the term

$$\lambda x : \texttt{int}.\ x + y + z$$

Performing a naive closure translation, we get

$$\begin{aligned}&\langle \lambda env : \texttt{int} \times \texttt{int}.\ \lambda x : \texttt{int}.\\ &\quad \texttt{let}\ y = \pi_0\ env\ \texttt{in}\\ &\quad \texttt{let}\ z = \pi_1\ env\ \texttt{in}\\ &\quad x + y + z\\ &, \langle y, z\rangle\rangle\end{aligned}$$

As a consequence, function application needs to be translated as well. If we have

$$f\ n$$

prior to closure conversion, we need to translate to

$$\begin{aligned}&\texttt{let}\ f = \pi_0\ e\ \texttt{in}\\ &\texttt{let}\ env = \pi_1\ e\ \texttt{in}\\ &f\ env\ n\end{aligned}$$

The strategy outlined above is sufficient if targeting an untyped closure language. However, if we want our target language to have types, we run into problems. In general, this strategy seeks to translate expressions of type $\tau_1 \to \tau_2$ to expressions of type $(\tau_{env} \to \tau_1 \to \tau_2) \times \tau_{env}$. For the sake of efficiency, we should choose τ_{env} such that it includes only those variables that are actually captured by the lambda expression, but doing so fails to produce a well-typed result in general. For example, if we translate

$$\texttt{if}\ b\ \texttt{then}\ (\lambda x : \texttt{int}.\ x + y)\ \texttt{else}\ (\lambda x : \texttt{int}.\ x)$$

as such, we get

$$\begin{aligned}&\texttt{if}\ b\ \texttt{then}\ (\\ &\quad \lambda env : \times\{\texttt{int}\}.\ \lambda x : \texttt{int}.\\ &\quad\quad \texttt{let}\ y\ =\ \pi_0\ env\ \texttt{in}\\ &\quad\quad x + y\\ &)\ \texttt{else}\ (\\ &\quad \lambda env : \times\{\}.\ \lambda x : \texttt{int}.\ x\\ &)\end{aligned}$$

This fails to type check because each branch of the if has a different type. In the worst case, in order to pick a single τ_{env} that works, we need to close over every variable in scope, which is wildly inefficient.

To solve this problem, we instead existentially quantify over the type of the environment. Specifically, we translate expressions of type $\tau_1 \to \tau_2$ to expressions of type $\exists \alpha_{env} : \texttt{type}.\ (\alpha_{env} \to \tau_1 \to \tau_2) \times \alpha_{env}$. Note that in our implementation closures will actually be uncurried, so in fact we will have something more like

$$\overline{\tau_1 \to \tau_2} = \exists \alpha_{env} : \texttt{type}.\ (\tau_1 \times \alpha_{env} \to \tau_2) \times \alpha_{env}$$

3. Closure conversion and free type variables

You may have noticed that we have thus far completely ignored the fact that *type variables*, as well as term variables, may appear free in an expression. In fact, terms of the form

$$\lambda x : \tau.\ e$$
$$\Lambda \alpha : k.\ e$$

can **both** capture **type and term** variables. So we have a couple of questions to consider. First, we have to consider how we will close each of these with respect to type variables. Second, we have to define $\overline{\forall \alpha : k.\ \tau}$ so that we can give a well-typed translation of polymorphic terms. This turns out to be somewhat difficult, and requires a concept called transluscent types. We present a much simpler solution by avoiding the problem completely: during CPS-conversion, we compiled polymorphic types to existential types, and got rid of Λ-abstractions completely. This solves the second problem completely and the first problem partially, but we still need to consider the problem of closing

$$\lambda x : \tau.\ e$$

with respect to type variables. In fact we defer addressing this to our next translation pass, called hoisting. It's worth noting as such that closure conversion as defined here does not actually close terms with respect to type variables, but defers that work until later.

4. Syntax and judgements

The syntax for IL-Closure is the same as the syntax for IL-CPS. The two principal typing judgements for this language are

$$\Delta; \Gamma \vdash e : 0$$
$$\Delta; \Gamma \vdash v : \tau$$

The statics are mostly unchanged from IL-CPS, except we have a different rule for $\neg\tau$ introduction.

$$\text{4A}\ \frac{\Delta \vdash \tau : \mathtt{type} \qquad \Delta; \cdot, x : \tau \vdash e : 0}{\Delta; \Gamma \vdash \lambda x : \tau.\ e : \neg\tau}$$

Observe that e is required to type-check using only x as a term variable; in other words, $\lambda x : \tau.\ e$ may have no free term variables.

5. Translation

Type translation is straightforward, except at $\neg\tau$. Recall that $\neg\tau$ can be thought of as $\tau \to 0$. Then, following the type translation described in section 2, we have

$$\overline{\neg\tau} = \exists \alpha_{env} : \mathtt{type}.\ \neg(\tau \times \alpha_{env}) \times \alpha_{env}$$

All other types are translated directly. The term translation rules of interest are as follows.

5A
$$\frac{\Delta \vdash \tau : \texttt{type} \qquad \Delta; \Gamma, x : \tau \vdash e : 0 \rightsquigarrow \overline{e} \qquad \Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n}{\Delta; \Gamma \vdash \lambda x : \tau.\ e : \neg\tau \rightsquigarrow \begin{array}{l} \texttt{pack}\ [\overline{\tau_1} \times \cdots \times \overline{\tau_n}, \\ \quad \langle (\lambda y : \overline{\tau} \times (\overline{\tau_1} \times \cdots \times \overline{\tau_n}). \\ \quad\quad \texttt{let}\ x = \pi_1\ y\ \texttt{in} \\ \quad\quad \texttt{let}\ env = \pi_2\ y\ \texttt{in} \\ \quad\quad \texttt{let}\ x_1 = \pi_1\ env\ \texttt{in} \\ \quad\quad \cdots \\ \quad\quad \texttt{let}\ x_n = \pi_n\ env\ \texttt{in} \\ \quad\quad \overline{e}), \\ \quad \langle x_1, \cdots, x_n \rangle \rangle] \\ \texttt{as}\ \exists \alpha_{env} : \texttt{type}.\ \neg(\overline{\tau} \times \alpha_{env}) \times \alpha_{env} \end{array}}$$

5B
$$\frac{\Delta; \Gamma \vdash v_1 : \neg\tau \rightsquigarrow \overline{v_1} \qquad \Delta; \Gamma \vdash v_2 : \tau \rightsquigarrow \overline{v_2}}{\Delta; \Gamma \vdash v_1\ v_2 : 0 \rightsquigarrow \begin{array}{l} \texttt{unpack}\ [\alpha_{env}, x] = \overline{v_1}\ \texttt{in} \\ \quad \texttt{let}\ f = \pi_1\ x\ \texttt{in} \\ \quad \texttt{let}\ env = \pi_2\ x\ \texttt{in} \\ \quad f \langle \overline{v_2}, env \rangle \end{array}}$$

6. A brief note on implementation

Closures are expensive, and a robust implementation should avoid creating them whenever possible. So, though this translation pass is needed to take care of higher-order usage of functions, "known" functions defined at the top-level should not be converted into closures.