Polymorphic Typed Defunctionalization
François Pottier and Nadji Gauthier, INRIA

Outline

1. Closure conversion and defunctionalization
2. Prior art
3. Our approach
4. Closing remarks
Closure conversion

Closure conversion turns a program that makes use of arbitrary functions into a program where only *closed* functions (code pointers) are allowed.

\(\lambda\)-abstractions in the source program are encoded as pairs of a code pointer and an environment (closures).

\[
\begin{align*}
\lfloor \lambda x.e \rfloor &= (\lambda(\{\bar{x}\}, x).\lfloor e \rfloor, \{\bar{x}\}) \quad \text{where } \bar{x} \text{ is } \text{fv}(\lambda x.e) \\
\lfloor e_1 e_2 \rfloor &= \text{let } (\text{code}, \text{env}) = \lfloor e_1 \rfloor \text{ in code } (\text{env}, \lfloor e_2 \rfloor)
\end{align*}
\]

It is known that closure conversion preserves types, provided function types are suitably encoded [Minamide, Morrisett, and Harper, POPL’96]:

\[
\lfloor \tau_1 \rightarrow \tau_2 \rfloor = \exists \alpha.((\alpha \times \lfloor \tau_1 \rfloor \rightarrow \lfloor \tau_2 \rfloor) \times \alpha)
\]
A close cousin: defunctionalization

Defunctionalization [Reynolds, 1972] encodes λ-abstractions as pairs of a tag and an environment, that is, as applications of a data constructor to an environment:

$$\llbracket \lambda^m x.e \rrbracket = m \{\bar{x}\} \quad \text{where } \bar{x} \text{ is } \text{fv}(\lambda x.e)$$

Function application is encoded as a call to a globally defined function apply...

$$\llbracket e_1 e_2 \rrbracket = \text{apply} \llbracket e_1 \rrbracket \llbracket e_2 \rrbracket$$

... which performs case analysis over $m$ and branches to the appropriate code:

\[
\text{letrec } \text{apply} = \lambda f. \lambda \text{arg}. \text{case } f \text{ of} \\
\mid m \{\bar{x}\} \mapsto \text{let } x = \text{arg} \text{ in } \llbracket e \rrbracket \quad (* \text{one such clause for every tag } m \аст) \]

Closure conversion and defunctionalization
Does defunctionalization preserve types?

Imagine the source program contains the functions \( \lambda \text{succ}\, x. x + 1 \) and \( \lambda \text{not}\, x. \text{not}\, x \), whose types are \( \text{int} \to \text{int} \) and \( \text{bool} \to \text{bool} \). Then, the body of \( \text{apply} \) contains the following clauses:

\[
\begin{align*}
\text{succ} & \mapsto \text{let } x = \text{arg} \text{ in } x + 1 \\
\text{not} & \mapsto \text{let } x = \text{arg} \text{ in } \text{not } x
\end{align*}
\]

In (say) \text{System F}, these clauses make incompatible assumptions about \( \text{arg} \), and produce results of incompatible types: thus, \( \text{apply} \) is ill-typed.
Prior art: specializing *apply*

One solution is to split *apply* into a family of functions, indexed by types:

\[
\text{letrec } \text{apply}_{\text{int} \to \text{int}} = \lambda f. \lambda \text{arg}. \text{case } f \text{ of} \\
| \text{succ} \mapsto \text{let } x = \text{arg} \text{ in } x + 1 \\
\text{and } \text{apply}_{\text{bool} \to \text{bool}} = \lambda f. \lambda \text{arg}. \text{case } f \text{ of} \\
| \text{not} \mapsto \text{let } x = \text{arg} \text{ in not } x
\]

Here, the data constructors *succ* and *not* may be declared as follows:

\[
\text{succ} : \text{Arrow}_{\text{int} \to \text{int}} \\
\text{not} : \text{Arrow}_{\text{bool} \to \text{bool}}
\]

where \text{Arrow}_{\text{int} \to \text{int}} and \text{Arrow}_{\text{bool} \to \text{bool}} are distinct algebraic data types.
Shortcoming: no polymorphism

In this approach, we have

$$\displaystyle [e_1 \ e_2] = apply_{\tau_1 \rightarrow \tau_2} \ [e_1] \ [e_2] \ \text{where } e_1 \text{ has type } \tau_1 \rightarrow \tau_2$$

$$\displaystyle [\tau_1 \rightarrow \tau_2] = Arrow_{\tau_1 \rightarrow \tau_2}$$

The trouble is, these definitions only make sense when $\tau_1 \rightarrow \tau_2$ has no free type variables. There is no sensible way of translating

$$\Lambda \alpha_1. \Lambda \alpha_2. \lambda f : \alpha_1 \rightarrow \alpha_2. \lambda x : \alpha_1. (f \ x).$$

As a result, this approach is applicable in a simply-typed setting only (and, via monomorphization, in the setting of ML).
Our approach

In order to translate \((f \; x)\) where \(f\) has type \(\alpha_1 \to \alpha_2\), we must have

\[
apply : \forall \alpha_1 \alpha_2. [\alpha_1 \to \alpha_2] \to [\alpha_1] \to [\alpha_2].
\]

If, furthermore, the type encoding is **uniform**, then the above implies

\[
apply \; [\tau_1] \; [\tau_2] : [\tau_1 \to \tau_2] \to [\tau_1] \to [\tau_2]
\]

for all \(\tau_1\) and \(\tau_2\), so this one \(apply\) function is in fact suitable for translating arbitrary applications.
A uniform type encoding

Let \( \text{Arrow} \) be a binary algebraic data type constructor, and let

\[
\begin{align*}
[\alpha] &= \alpha \\
[\tau_1 \rightarrow \tau_2] &= \text{Arrow} [\tau_1] [\tau_2]
\end{align*}
\]

This yields a uniform type encoding.

Since \( \lambda \text{succ}.x.x + 1 \) and \( \lambda \text{not}.x.\text{not} \ x \) have types \( \text{int} \rightarrow \text{int} \) and \( \text{bool} \rightarrow \text{bool} \), their encodings must have types \( \text{Arrow int int} \) and \( \text{Arrow bool bool} \), respectively. So, we declare:

\[
\begin{align*}
\text{succ} &: \text{ Arrow int int} \\
\text{not} &: \text{ Arrow bool bool}
\end{align*}
\]

\( \text{Arrow} \) is a guarded algebraic data type [Xi, Chen, and Chen, POPL’03].
Does defunctionalization preserve types? (reconsidered)

The body of apply, enriched with type annotations, is now:

\[
\text{letrec } apply : \forall \alpha_1 \alpha_2. \text{Arrow } \alpha_1 \alpha_2 \to \alpha_1 \to \alpha_2 = \\
\Lambda \alpha_1. \Lambda \alpha_2. \lambda f : \text{Arrow } \alpha_1 \alpha_2. \lambda \text{arg : } \alpha_1.
\text{case } f \text{ of } \\
| \text{succ } \mapsto (* f \text{ is succ, so } \text{Arrow } \alpha_1 \alpha_2 = \text{Arrow int int holds *) } \\
| \text{let } x = \text{arg in } x + 1 : \alpha_2 \\
| \text{not } \mapsto (* f \text{ is not, so } \text{Arrow } \alpha_1 \alpha_2 = \text{Arrow bool bool holds *) } \\
| \text{let } x = \text{arg in not } x : \alpha_2 \\
\]

Case analysis over a guarded algebraic data type yields extra type information. Defunctionalization is now type-preserving.
Specialization

One may define versions of \textit{apply} that are \textit{specialized} with respect to the \textit{types} of the parameter and of the result:

\[
apply_{\tau_1 \rightarrow \tau_2} : \forall \bar{\alpha}.[\tau_1 \rightarrow \tau_2] \rightarrow [\tau_1] \rightarrow [\tau_2]
\]

where \(\bar{\alpha}\) is \(\text{ftv}(\tau_1 \rightarrow \tau_2)\), or with respect to the \textit{number} of arguments that are simultaneously available:

\[
apply_n : \forall \alpha_1 \ldots \alpha_n \alpha_{n+1}.[\alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow \alpha_{n+1}] \rightarrow \alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow \alpha_{n+1},
\]

or both.

Branches that lead to an inconsistent typing assumption may be \textit{pruned}—for instance, \(apply_{\text{int} \rightarrow \text{int}}\) need not check for the tag \textit{not}. This allows dispatch to be made more efficient based on type information available at the call site.
Closing remarks

- When viewed as a transformation from System F, extended with guarded algebraic data types, into itself, defunctionalization is type-preserving.

- Defunctionalization per se is not type-directed, so its correctness may be established using a generic (untyped) simulation argument.

- Interesting type-directed optimizations are possible.

- This illustrates the usefulness of guarded algebraic data types as a programming language feature. (Defunctionalization turns Danvy’s [1998] clever `sprintf` encoding back to direct style!)