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.

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

$$\begin{aligned} [\![\lambda x.e]\!] &= (\lambda(\{\bar{x}\}, x).[\![e]\!], \{\bar{x}\}) && \text{where } \bar{x} \text{ is } \mathrm{fv}(\lambda x.e) \\ [\![e_1\, e_2]\!] &= \mathsf{let}\,(\mathit{code}, \mathit{env}) = [\![e_1]\!]\ \mathsf{in}\ \mathit{code}\,(\mathit{env}, [\![e_2]\!]) \end{aligned}$$

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

$$[\![\tau_1 \rightarrow \tau_2]\!] = \exists\alpha.((\alpha \times [\![\tau_1]\!] \rightarrow [\![\tau_2]\!]) \times \alpha)$$

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:

$$[\![\lambda^m x.e]\!] \quad = \quad m\,\{\bar{x}\} \quad \text{where } \bar{x} \text{ is } \mathrm{fv}(\lambda x.e)$$

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

$$[\![e_1\, e_2]\!] \quad = \quad \mathit{apply}\, [\![e_1]\!]\, [\![e_2]\!]$$

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

$$\begin{array}{l} \mathsf{letrec}\ \mathit{apply} = \lambda f.\lambda \mathit{arg}.\mathsf{case}\, f\ \mathsf{of} \\ \quad \mid m\,\{\bar{x}\} \mapsto \mathsf{let}\, x = \mathit{arg}\ \mathsf{in}\ [\![e]\!] \qquad \text{(* one such clause for every tag } m \text{ *)} \end{array}$$

Does defunctionalization preserve types?

Imagine the source program contains the functions $\lambda^{succ}x.x+1$ and $\lambda^{not}x.\mathsf{not}\,x$, whose types are $int \rightarrow int$ and $bool \rightarrow bool$. Then, the body of *apply* contains the following clauses:

$$\begin{array}{lcl} \mid succ & \mapsto & \mathsf{let}\; x = arg\; \mathsf{in}\; x+1 \\ \mid not & \mapsto & \mathsf{let}\; x = arg\; \mathsf{in}\; \mathsf{not}\; x \end{array}$$

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

Prior art: specializing *apply*

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

$$\begin{array}{l} \text{letrec } apply_{int \rightarrow int} = \lambda f.\lambda arg.\text{case } f \text{ of} \\ \quad \mid succ \mapsto \text{let } x = arg \text{ in } x + 1 \\ \text{and } apply_{bool \rightarrow bool} = \lambda f.\lambda arg.\text{case } f \text{ of} \\ \quad \mid not \mapsto \text{let } x = arg \text{ in not } x \end{array}$$

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

$$\begin{array}{rcl} succ & : & Arrow_{int \rightarrow int} \\ not & : & Arrow_{bool \rightarrow bool} \end{array}$$

where $Arrow_{int \rightarrow int}$ and $Arrow_{bool \rightarrow bool}$ are distinct algebraic data types.

Shortcoming: no polymorphism

In this approach, we have

$$
\begin{aligned}
[\![e_1\, e_2]\!] &= apply_{\tau_1 \to \tau_2}\, [\![e_1]\!]\, [\![e_2]\!] \quad \text{where } e_1 \text{ has type } \tau_1 \to \tau_2 \\
[\![\tau_1 \to \tau_2]\!] &= Arrow_{\tau_1 \to \tau_2}
\end{aligned}
$$

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

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

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 τ_1 and τ_2, so this one *apply* function is in fact suitable for translating arbitrary applications.

A uniform type encoding

Let *Arrow* be a binary algebraic data type constructor, and let

$$\begin{aligned} [\![\alpha]\!] &= \alpha \\ [\![\tau_1 \to \tau_2]\!] &= \mathit{Arrow}\, [\![\tau_1]\!]\, [\![\tau_2]\!] \end{aligned}$$

This yields a uniform type encoding.

Since $\lambda^{succ} x.x + 1$ and $\lambda^{not} x.\mathsf{not}\, x$ have types $int \to int$ and $bool \to bool$, their encodings must have types $\mathit{Arrow}\; int\; int$ and $\mathit{Arrow}\; bool\; bool$, respectively. So, we declare:

$$\begin{aligned} succ &: \mathit{Arrow}\; int\; int \\ not &: \mathit{Arrow}\; bool\; bool \end{aligned}$$

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:

$$\text{letrec } apply : \forall\alpha_1\alpha_2.Arrow\,\alpha_1\,\alpha_2 \to \alpha_1 \to \alpha_2 =$$

$$\Lambda\alpha_1.\Lambda\alpha_2.\lambda f : Arrow\,\alpha_1\,\alpha_2.\lambda arg : \alpha_1.$$

$$\text{case } f \text{ of}$$

$$\mid succ \mapsto (*\ f \text{ is } succ, \text{ so } Arrow\,\alpha_1\,\alpha_2 = Arrow\ int\ int \text{ holds } *)$$

$$\text{let } x = arg \text{ in } x + 1 : \alpha_2$$

$$\mid not \mapsto (*\ f \text{ is } not, \text{ so } Arrow\,\alpha_1\,\alpha_2 = Arrow\ bool\ bool \text{ holds } *)$$

$$\text{let } x = arg \text{ in not } x : \alpha_2$$

Case analysis over a guarded algebraic data type yields extra type information.

Defunctionalization is now type-preserving.

Specialization

One may define versions of *apply* that are specialized with respect to the types of the parameter and of the result:

$$apply_{\tau_1 \rightarrow \tau_2} : \forall \bar{\alpha}. [\![\tau_1 \rightarrow \tau_2]\!] \rightarrow [\![\tau_1]\!] \rightarrow [\![\tau_2]\!] \qquad \text{where } \bar{\alpha} \text{ is } \mathrm{ftv}(\tau_1 \rightarrow \tau_2),$$

or with respect to the 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 pruned—for instance, $apply_{int \rightarrow int}$ need not check for the tag *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!)