Type-driven Defunctionalization

Jeffrey M. Bell & Françoise Bellegarde*& James Hook†
Pacific Software Research Center
Oregon Graduate Institute of Science & Technology
PO Box 91000
Portland, Oregon 97291-1000
USA
{bell,bellegar,hook}@cse.ogi.edu

Abstract

In 1972, Reynolds outlined a general method for eliminating functional arguments known as *defunctionalization*. The idea underlying defunctionalization is encoding functional values as first-order data, and then to realized the applications of the encoded function via an *apply* function. Although this process is simple enough, problems arise when defunctionalization is used in a polymorphic language. In such a language, a functional argument of a higher-order function can take different type instances in different applications. As a consequence, its associated *apply* function can be untypable in the soucre language. In the paper we present a defunctionalization transformation which preserves typability. Moreover, the transformation imposes no restriction on functional arguments of recursive functions, and it handles functions as results as well as functions encapsulated in constructors or tuples. The key to this success is the use of type information in the defunctionalization transformation. Run-time characteristics are preserved by defunctionalization; hence, there is no performance improvement coming from the transformation itself. However closures need not be implemented to compile the transformed program. Since the defunctionalization is driven by type information, it can also easily perform a specialization of higher-order functions with respect to the values of their functional arguments, hence gaining a real run-time improvement of the transformed program.

1 Introduction

Defunctionalization is the transformation of a program that uses higher-order functions into a semantically equivalent first-order program. This paper presents defunctionalization as a source-to-source translation in a Hindley-Milner typable functional language. Defunctionalization is very closely related to "closure conversion" in functional compilers. We are motivated to investigate it as a separate transformation because we have developed tools for functional language compilation that are based on typed source-to-source transformation and that generate typed first-order programs in conventional languages. In addition, we apply first-order program transformation techniques to the defunctionalized representations of programs. The explicit study of typed defunctionalization illuminates issues related to type specialization. The algorithm presented performs necessary type specialization but does not generate a strictly monomorphic representation.

$$
\begin{array}{l}
\textbf{fun}\ \ map\ F\ y\ =\ \textbf{case}\ y\ \textbf{of} \\
\qquad Nil \Rightarrow Nil \\
\quad\ \mid\ \ Cons(x, xs) \Rightarrow Cons(F\ x, map\ F\ xs) \\
\textbf{fun}\ addone\ l\ =\ map\ incr\ l \\
\textbf{fun}\ subone\ l\ =\ map\ decr\ l
\end{array}
$$

Figure 1: Example higher-order program

$$
\begin{array}{l}
\textbf{fun}\ apply\_map\ encoding\ arg\ = \\
\qquad \textbf{case}\ encoding\ \textbf{of} \\
\qquad\qquad \text{“}incr\text{”}\ \Rightarrow\ incr\ arg \\
\qquad\quad\ \mid\ \ \text{“}decr\text{”}\ \Rightarrow\ decr\ arg \\
\textbf{fun}\ \ map'\ F\ y\ =\ \textbf{case}\ y\ \textbf{of} \\
\qquad Nil \Rightarrow Nil \\
\quad\ \mid\ \ Cons(x, xs) \Rightarrow Cons(apply\_map\ F\ x, map'\ F\ xs) \\
\textbf{fun}\ addone\ l\ =\ map'\ \text{“}incr\text{”}\ l \\
\textbf{fun}\ subone\ l\ =\ map'\ \ \text{“}decr\text{”}\ l
\end{array}
$$

Figure 2: Defunctionalized program

Reynolds outlined a general method for *defunctionalization* [Rey72]. The idea underlying defunctionalization is encoding functional values as first-order data. Since a first-order value cannot be applied as a function, applications of the encoded functionals need to be modified, by introducing a call to an *apply* function. The *apply* function is called wherever the functional argument was applied in the original higher-order function. The *apply* function takes as arguments the encoded functional and all the arguments to the functional. The *apply* function dispatches based on the encoding, and applies the appropriate function to the remaining arguments. For example, if the program in Figure 1 is defunctionalized using strings containing the function name as the representation of function values, the program in Figure 2 is the result.

Reynolds' method defunctionalizes functions that have functional arguments, but not functions that return func-

*Currently at the *Université de Franche-Comté (France)*

†The authors are supported by a contract with Air Force Material Command (F19628-93-C-0069).

$$
\begin{array}{l}
\textbf{fun}\ \mathit{apply}_{int \to int}\ \mathit{encoding}\ \mathit{arg}\ = \\
\quad \textbf{case}\ \mathit{encoding}\ \textbf{of} \\
\qquad \text{“}\mathit{incr}\text{”} \Rightarrow \mathit{incr}\ \mathit{arg} \\
\textbf{fun}\ \mathit{apply}_{string \to int}\ \mathit{encoding}\ \mathit{arg}\ = \\
\quad \textbf{case}\ \mathit{encoding}\ \textbf{of} \\
\qquad \text{“}\mathit{str2int}\text{”} \Rightarrow \mathit{str2int}\ \mathit{arg} \\
\textbf{fun}\ \mathit{map}_{int \to int}\ F\ y\ =\ \textbf{case}\ y\ \textbf{of} \\
\quad \mathit{Nil} \Rightarrow \mathit{Nil} \\
\mid\ \mathit{Cons}(x, xs) \Rightarrow \mathit{Cons}(\ \mathit{apply\_map}_{int \to int}\ F\ x, \\
\qquad\qquad \mathit{map}_{int \to int}\ F\ xs) \\
\textbf{fun}\ \mathit{map}_{string \to int}\ F\ y\ =\ \textbf{case}\ y\ \textbf{of} \\
\quad \mathit{Nil} \Rightarrow \mathit{Nil} \\
\mid\ \mathit{Cons}(x, xs) \Rightarrow \mathit{Cons}(\mathit{apply\_map}_{string \to int}\ F\ x, \\
\qquad\qquad \mathit{map}_{string \to int}\ F\ xs) \\
\textbf{fun}\ \mathit{addone}\ l\ =\ \mathit{map}_{int \to int}\ \text{“}\mathit{incr}\text{”}\ l \\
\textbf{fun}\ \mathit{subone}\ l\ =\ \mathit{map}_{string \to int}\ \text{“}\mathit{str2int}\text{”}\ l
\end{array}
$$

Figure 3: Defunctionalized example with specialization

tions. Chin and Darlington address this in their $\mathcal{A}$-algorithm [CD], which removes some functional results by η-expansion. Our transformation includes the capabilities of the $\mathcal{A}$-algorithm.

Problems arise when defunctionalization is used in a polymorphic language [Bel94, BH94a]. In the above example, *map* is called twice, each time passing a function of type $int \to int$. Suppose that the second call to *map* used a function with a different type, e.g. $str2int : string \to int$. Then the *apply_map* function would be:

$$
\begin{array}{l}
\textbf{fun}\ \mathit{apply\_map}\ \mathit{encoding}\ \mathit{arg}\ = \\
\quad \textbf{case}\ \mathit{encoding}\ \textbf{of} \\
\qquad \text{“}\mathit{incr}\text{”} \Rightarrow \mathit{incr}\ \mathit{arg} \\
\quad\mid\ \text{“}\mathit{str2int}\text{”} \Rightarrow \mathit{str2int}\ \mathit{arg}
\end{array}
$$

This function is ill typed in a Hindley-Milner language (although the resulting program will have no "wrong" behaviors if the orignal program was type correct). Our solution is to specialize *apply* functions by the type of the functionals whose encodings are interpreted in the *apply* function. Under this scheme, there would be one version of *apply_map* that applies $int \to int$ functions, and another version that applies $string \to int$ functions. Since the *apply* functions are specialized by type, so must be the higher-order functions that rely on the various *apply* functions. Again, this specialization is according to the type of the encoded functions. A type-correct defunctionalized version of the modified example is given in Figure 3.

Another complication encountered by the defunctionalization algorithm is illustrated in Figure 4, showing an implementation of the *sum* function using continuation-passing style. Here, *sum* is called with a series of different function values for the argument F. This requires an encoding for F that accounts for all possible values of F, which is accomplished by encoding F in a recursive datatype, as described in Section 2.5.

Our desire for a defunctionalization algorithm was motivated by work on software component generators that perform type-faithful translations of higher-order functional programs to first-order imperative languages (e.g. Ada) [B+94, KBB+94]. Although specialization-based techniques for defunctionalization exist [CD], they do not defunctionalize functions in data constructors (e.g. lists of functions). In addition, there are higher-order functions on which specialization fails because an infinite family of specializations would be generated, such as the example in Figure 4.

$$
\begin{array}{l}
\textbf{fun}\ \mathit{sum}\ F\ y\ =\ \textbf{case}\ y\ \textbf{of} \\
\quad \mathit{Nil} \Rightarrow F\ 0 \\
\mid\ \mathit{Cons}(x, xs) \Rightarrow \mathit{sum}\ (\lambda n.1 + F\ n)\ xs
\end{array}
$$

Figure 4: Higher-order function in CPS

Terms	t, M, e (terms are underlined in transformation rules)
Variables	v, w, x, y, z, F, G, H
Function symbols	f, g, h
Type variables	α, β, δ
Type terms	Π, Φ, Ω, Ψ
Data constructors	C labelled with the type and the term it encodes
Type constructors	T labelled with the type it encodes as an index

Table 2: Naming conventions

Our type-driven transformation is presented as a set of transformation rules. Two rules decrease higher-orderness and three rules lessen polymorphism. The rules make clear the amount of monomorphization which is necessary for defunctionalization.

The remainder of the paper is organized as follows. Section 2 describes how the type-driven defunctionalization transformation is applied to higher-order programs. Section 3 summarizes the results related to soundness, termination, and effectiveness of the transformation (full proofs are given in a technical report [BBH96]). Section 4 presents our conclusions and future work. The appendices include several illustrative examples.

2 Presentation of the transformation

The defunctionalization transformation applies to a restricted form of higher-order polymorphic strongly-typed functional language. A grammar for the language is presented in Table 1. This is a simple polymorphic language without local let or lambda bindings. Only function symbols, function variables, and constructors can appear in function application position. A program consists of datatype declarations followed by function declarations followed by a (top-level) term. This language form can be calculated from, say, a core ML program by the standard lambda-lifting transformation [Joh85]. These restrictions simplify the exposition; the language can be extended without fundamental changes. The naming conventions used in this and following Sections are given in Table 2.

Datatypes:	*ddecl*	$::=$ `datatype` $\alpha_1 \ldots \alpha_n$ `T` $=$ *cdecs*	
	cdecs	$::=$ *cdec* $\mid$ *cdec* $\mid$ *cdecs*	
	cdec	$::=$ `C` $type_1 \times \ldots \times type_n$	$(n \geq 0)$
Functions:	*fdecl*	$::=$ `fun f v1...vn` $=$ *term*	
Terms:	*term*	$::=$ *rator* $term_1 \ldots term_n$	$(n \geq 0)$
		$\mid$ `case` *term* `of` pat_1 `=>` $term_1 \mid \ldots \mid pat_n$ `=>` $term_n$	
	rator	$::=$ `f` $\mid$ `v` $\mid$ `C`	
	pat	$::=$ `C v1 ...vn`	$(n \geq 0)$
		$\mid$ `(v1, v2)`	
Types:	*type*	$::= \alpha \mid type_1 \rightarrow type_2 \mid$ `T` $type_1 \ldots type_n \mid type_1 \times type_2$	
Program:	*program*	$::= ddecl^* \; fdecl^* \; term$	

Table 1: The grammar of the polymorphic higher-order language

2.1 Functional type specialization and arrow type parameter encoding

The transformation relies on analyses of types. The basic idea is to replace arrow type arguments (of type order 0) by appropriate elements of a datatype. A datatype T_Π captures the arrow type arguments of the arrow type Π where each arrow type argument of type Π is encoded by a constructor in the datatype T_Π. Consider the following example of a term: *map id* $[1,2]$ where *map* is declared as in Figure 1. The type of *map* is:

$$map : (\alpha \rightarrow \beta) \rightarrow list\ \alpha \rightarrow list\ \beta$$

It is a higher-order function since it has an arrow type argument. The type of *id* is $\alpha \rightarrow \alpha$. But in the context of the application *map id* $[1,2]$, *id* is instantiated at type $int \rightarrow int$. Therefore, this occurrence of *id* can be encoded by a constructor $C^{id}_{int\rightarrow int}$ in a datatype $T_{int\rightarrow int}$ which is created to contain the encodings of arrow type arguments of type $int \rightarrow\ int$. So doing, the type of *map* has to become $T_{int\rightarrow int} \rightarrow list\ int \rightarrow list\ int$. However, in another application, the type of *map* could be instantiated otherwise. The solution is to create as many different versions, called *clones*, of the higher-order function as needed. The transformation requires a different clone of the higher-order function for each type at which the higher-order function is applied in all of its applications. Cloning is necessary because each clone of the higher-order function will use a specialized encoding of function values.

When creating and manipulating clones, it is necessary to keep track of which expressions are in the encoded representation and which are not. This information in indicated with braces. Specifically, braces subscripted by a type ($\{\cdot\}_\Pi$) are placed around encoded fragments that are used in their original context. In the dual case we write braces with an inverse ($\{\cdot\}^{-1}$). The definition and typing rules are given in Figure 6.

The creation of a clone of a higher-order function is not a simple task. An arrow type parameter in the higher-order function becomes a 0-order type parameter in its clones, so it can no longer be applied as a function. The first step is to create a clone in which the arrow type argument is still a function but of a specialized type. For example, the clone of *map* as presented in Figure 5 is of type:

$$\Psi = (int \rightarrow\ int) \rightarrow\ list\ int \rightarrow\ list\ int.$$

$$
\begin{array}{l}
\textbf{fun}\ \ map_\Psi\ F\ y\ =\ \textbf{case}\ y\ \textbf{of} \\
\qquad Nil \Rightarrow Nil \\
\quad \mid\ Cons(x, xs) \Rightarrow Cons(\ \{F\}_{int\rightarrow int}\ x, \\
\qquad\qquad\qquad\qquad\qquad map\ \{F\}_{int\rightarrow int}\ xs)
\end{array}
$$

Figure 5: Higher-order clone of *map* of type Ψ

A clone of f at a specialized type Π is given the function symbol f_Π. It is created by taking a copy of the declaration of f in which arrow type variables are replaced with an application of the function $\{\cdot\}_\Pi$ to the arrow type variable, as shown in Figure 5. Note that at this point the recursive call to *map* is not encoded. This will be addressed when this function call is defunctionalized.

The next step is to encode arrow type arguments in datatypes. Then, in the body of the clone, an application of $\{F\}_\Pi$ must be transformed into an application of an *apply* function. In the body of map_Ψ, the application $\{F\}_{int\rightarrow int}\ x$ is transformed into $apply_{int\rightarrow int}\ F\ x$ as shown in Figure 7. The *apply* function depends on the datatype. It establishes a correspondence between the encoding and the encoded terms. For the example, the transformation adds the declaration of an $apply_{int\rightarrow int}$ function shown in Figure 8.

2.2 Transformation rules

The transformation informally described in the previous section is guided by a set of transformation rules. A rule transforms an expression of type order 0, which we call *fully-applied*, in the context of a program P. It also updates the set Δ of function symbol declarations of P and the set Θ of datatype declarations of P, so that a transformation rule transforms a triple (*term*, Δ, Θ) into a new triple. The three rules $(FunSpec)$, $(EncodeClosed)$, $(ApplyVar)$ shown in Figures 9, 10 and 11, allow us to defunctionalize the fully-applied application *map id* $[1,2]$. Definitions of the functions *order* and *functional* used in these rules can be found in Figure 14.

The rules reference type information calculated by type

Definitions:

$\{t\}_{\Pi}$ = the unencoded value of t

$\{\{\mathtt{v}\}_{\Pi}\}^{-1} = \mathtt{v}$
$\{\mathtt{f}\ t_1 \ldots t_n\}^{-1} = \mathtt{f}\ \{t_1\}^{-1} \ldots \{t_n\}^{-1}$
$\{\mathtt{C}\ t_1 \ldots t_n\}^{-1} = \mathtt{C}\ \{t_1\}^{-1} \ldots \{t_n\}^{-1}$
$\{\mathtt{v}\ t_1 \ldots t_n\}^{-1} = \mathtt{v}\ \{t_1\}^{-1} \ldots \{t_n\}^{-1}$
$\{\mathtt{case}\ t\ \mathtt{of}\ p_1 \Rightarrow t_1 \mid \ldots \mid p_n \Rightarrow t_n\}^{-1} =$
$\quad \mathtt{case}\ \{t\}^{-1}\ \mathtt{of}\ p_1 \Rightarrow \{t_1\}^{-1} \mid \ldots \mid p_n \Rightarrow \{t_n\}^{-1}$

Typing rule:

$$\frac{\Gamma \vdash t : \Pi}{\Gamma \vdash \{t\}_{\Pi} : \Pi}$$

Figure 6: Definitions and typing rules for semantic functions

fun $map_{\Psi}\ F\ y\ =$ **case** y **of**
$\quad Nil \Rightarrow Nil$
$\mid\ Cons(x, xs) \Rightarrow Cons(\ apply_{int\to int}\ F\ x,$
$\qquad map\ \{F\}_{int\to int}\ xs)$

Figure 7: Clone of *map* where F is seen as encoded

datatype $T_{int\to int}\ =\ C^{id}_{int\to int}$
fun $apply_{int\to int}\ f\ x\ =$ **case** f *of*
$\quad C^{id}_{int\to int} \Rightarrow (id\ x)$

Figure 8: Encoding-derived declarations

inference on the original, untransformed terms. As the transformation progresses, this information is propagated unchanged. Since the transformation proceeds nondeterministically through untyped intermediate representations, it is important to note that sometimes the type does not apply to the term being manipulated, but the input term of which it is a residual.

The rule *(FunSpec)* specializes a fully-applied higher-order application of a function symbol according to the type of its arrow type arguments. For instance, it transforms $map\ id\ [1,2]$ into $map_{\Psi}\ id\ [1,2]$ and adds the clone map_{Ψ} from in Figure 5 to the function declaration set Δ. The rule *(EncodeClosed)* encodes arrow type arguments into constructors of datatypes. For example, it transforms the expression $map_{\Psi}\ id\ [1,2]$ into $map_{\Psi}\ C^{id}_{int\to int}\ [1,2]$ and adds the encoding-derived declarations in Figure 8 to the declaration set Δ. Next, in the body of the declaration, the application $\{F\}_{int\to int}\ x$ is transformed into $apply_{\Psi}\ F\ x$ by the rule *(ApplVar)*. In the clone map_{Ψ} there remains a fully-applied higher-order application of *map* which comes from the original recursive application of *map*. No more transformation rules are needed to cope with recursive calls in a set of mutually recursive clone declarations, as explained in the following section.

2.3 Higher-order recursive functions

In a clone, types can be inferred with Hindley-Milner type inference [Mil78] augmented with the rules of Figure 6 and from the type label of the clone function symbol. For example, in the body of the map_{Ψ} clone of *map*, the specialized type of *map* in the recursive application: $map\ \{F\}_{int\to int}\ xs$ is recognized as the type Ψ because of the subscript $int \to int$.

Since we suppose Hindley-Milner typability, recursive calls in a set of mutually-recursive declarations are of a consistent inferred type. Therefore, in a set of mutually-recursive specialized clones, the types of recursive calls are of consistent specialized types. This serves the useful purpose of allowing the specialization rule *(FunSpec)* to *fold* the specialized recursive call anywhere it occurs in the set of mutually-recursive clone declarations without the need for further analysis. For example, the rule *(FunSpec)* is used again to change the occurrence of *map* into map_{Ψ} in the body of the declaration of the clone map_{Ψ}, and change $\{F\}_{int\to int}$ into F via the application of $\{\cdot\}^{-1}$. No more rules apply; the result of the transformation is the first-order program composed of the term $map_{\Psi}\ C^{id}_{int\to int}\ [1,2]$ and the declarations in Figure 8 and below:

fun $map_{\Psi}\ F\ y\ =$ **case** y **of**
$\quad Nil \Rightarrow Nil$
$\mid\ Cons(x, xs) \Rightarrow Cons(apply_{int\to int}\ F\ x, map_{\Psi}\ F\ xs)$

An example of mutually-recursive functions is presented in Appendix B.4.

2.4 Polymorphic higher-order application

Cloning a polymorphic function is not as simple when it is specialized in such a way that it becomes a function that returns a function. In such a case, a polymorphic function symbol f with arity a may be applied to a number of arguments n where $n > a$. For example, although *id* is of

IF

$$\exists\, j,\ j \in 1 \ldots n,\ \mathit{functional}(\Pi_j)$$
$$\wedge \text{ all } \mathit{functional} \text{ variables in } \mathit{functional} \text{ arguments are arguments of } \{\cdot\}_\Psi \text{ for some } \Psi$$
$$\wedge\ n = \mathrm{order}(\Pi) \wedge\ \mathrm{order}(\Omega) = 0 \wedge\ f \text{ is a function symbol}$$

AND

$$\begin{array}{l} \Gamma \vdash t_i\ :\ \Pi_i,\ \forall i,\ i \in 1 \ldots n, \\ \Gamma \vdash \underline{f\ t_1\ \ldots\ t_n}\ :\ \Omega, \\ \Gamma \vdash \underline{f}\ :\ \Pi, \\ \sigma\Pi = \Pi_1 \rightarrow \ldots \rightarrow \Pi_n \rightarrow \Omega \end{array}$$

THEN

$$\underline{f\ t_1\ \ldots\ t_n}, \Delta, \Theta \Longrightarrow \underline{f_{\sigma\Pi}\ \{t_1\}^{-1}\ \ldots\ \{t_n\}^{-1}}, \Delta', \Theta$$

WHERE

$$\Delta' = \begin{cases} \Delta \cup \{\underline{f_{\sigma\Pi}\ x_1 \ldots x_n = M}[x_{i_1} \leftarrow \underline{\{x_{i_1}\}_{\Pi_{i_1}}} \ldots x_{i_k} \leftarrow \underline{\{x_{i_k}\}_{\Pi_{i_k}}}]\} \\ \mathit{where}\ \Delta(f) = \underline{f\ x_1\ \ldots\ x_n = M}\ \mathit{and}\ i_j,\ j \in\ 1 \ldots k, \\ \text{are the indices of the } \mathit{functional} \text{ arguments} \end{cases}$$

Figure 9: (FunSpec) Functional type specialization transformation rule

IF

$$\exists\, j,\ j \in 1 \ldots n \wedge\ \mathrm{order}(\Pi_j) > 0 \wedge\ t_j \text{ is a closed term}$$
$$\wedge\ \boldsymbol{F} \text{ is a function symbol or a function variable} \wedge\ \mathrm{order}(\Omega) = 0$$

AND

$$\begin{array}{l} \Gamma \vdash t_i\ :\ \Pi_i, \\ \forall i,\ i \in 1 \ldots n, \\ \Gamma \vdash \underline{\boldsymbol{F}\ t_1\ \ldots\ t_n}\ :\ \Omega, \end{array}$$

THEN

$$\underline{\boldsymbol{F}\ t_1\ \ldots\ t_j \ldots\ t_n}, \Delta, \Theta \Longrightarrow \underline{\boldsymbol{F}\ t_1\ \ldots\ C^{t_j}_{\Pi_j}\ \ldots\ t_n}, \Delta', \Theta'$$

WHERE

$$\Delta' = \begin{cases} \mathit{if}\ \mathit{Apply}_{\Pi_j} \text{ has not been declared in } \Delta\ \mathit{then} \\ \Delta \cup \{\underline{\mathit{Apply}_{\Pi_j}\ x\ y_1\ \ldots\ y_{\mathrm{order}(\Pi_j)} =} \\ \quad \underline{\mathit{case}\ x\ \mathit{of}\ C^{t_j}_{\Pi_j}\ \Rightarrow\ t_j\ y_1 \ldots\ y_{\mathrm{order}(\Pi_j)}}\}\ \mathit{else} \\ \text{add to it the } \mathit{case} \text{ arm } \underline{C^{t_j}_{\Pi_j}\ \Rightarrow\ t_j\ y_1 \ldots\ y_{\mathrm{order}(\Pi_j)}} \end{cases}$$

$$\Theta' = \begin{cases} \mathit{if}\ T_{\Pi_j} \text{ has not been declared in } \Theta \\ \mathit{then}\ \Theta \cup \{\underline{\textbf{datatype}\ T_{\Pi_j} = C^{t_j}_{\Pi_j}}\} \\ \mathit{else} \text{ add to it the constructor } C^{t_j}_{\Pi_j} \end{cases}$$

Figure 10: (EncodeClosed) Closed arrow type parameter encoding transformation rule

IF

$$\mathrm{order}(\Omega) = 0 \wedge\ F \text{ is a variable}$$

AND

$$\Gamma \vdash \underline{F\ t_1 \ldots t_n} : \Omega$$

THEN

$$\underline{\{F\}_\Pi\ t_1\ \ldots\ t_n}, \Delta, \Theta \Longrightarrow \underline{\mathit{Apply}_\Pi\ F\ t_1\ \ldots\ t_n}, \Delta, \Theta$$

Figure 11: (ApplVar) Higher-order variable application

$$
\boxed{\begin{array}{l}
(case\ t\ of\ p_1\ \Rightarrow\ t_1\ ||\ ...\ ||\ p_n\ \Rightarrow\ t_n)\ t' \\
\to \\
case\ t\ of\ p_1\ \Rightarrow\ t_1\ t'\ ||\ ...\ ||\ p_n\ \Rightarrow\ t_n\ t'
\end{array}}
$$

Figure 13: Rewrite rule for *case expression* normalization

- $order : type \to int \ = \lambda \mathtt{t}.\mathbf{case}\ \mathtt{t}\ \mathbf{of}$
  $\alpha \ \Rightarrow\ 0$
  $\mid \alpha \to \beta \ \Rightarrow\ 1 + order(\beta)$
  $\mid \mathtt{T}\ \Pi_1 \ldots \Pi_n \Rightarrow\ 0$
  $\mid \Pi_1 \times \Pi_2 \Rightarrow\ 0$

- $functional : type \to bool \ =$
  $\lambda \mathtt{t}.\mathbf{case}\ \mathtt{t}\ \mathbf{of}$
  $\alpha \ \Rightarrow\ false$
  $\mid \alpha \to \beta \ \Rightarrow\ true$
  $\mid \ \mathtt{T}\ \Pi_1 \ldots \Pi_n \ \Rightarrow\ functional(\Pi_1) \vee \ldots \vee functional(\Pi_n)$
  $\vee\ \exists$ a constructor $\mathtt{C} : \Gamma_1 \times \ldots \times \Gamma_m \to T\ \Pi_1 \ldots \Pi_n$ such that
  $functional(\Gamma_1) \vee \ldots \vee functional(\Gamma_m)$
  $\mid \Pi_1 \times \Pi_2 \Rightarrow\ false$

Figure 14: Difference between *order* and *functional*

arity 1, in *id id* (*id* 7) the first occurrence of *id* has two arguments. In this case, the specialized clone of the function definition must not be simply a copy, but an η-extension annotated with the types of the functional arguments. The function *id* in the expression *id id* (*id* 7) is instantiated at two types: the first-order type $int \to int$ and the second-order type $\Phi = (int \to int) \to (int \to int)$. In addition, the second *id* appears as a parameter to a higher-order function and the third as a first-order function. The defunctionalization must distinguish on both order and these distinct roles. Since the outermost application has a functional argument *id*, an η-extended clone **fun** $id_\Phi\ x\ y\ = \{x\}_{int\to int}\ y$ is produced, along with a transformation of the application into: $id_\Phi\ id\ (id\ 7)$. This is accomplished by the rule *(ExpandSpec)* shown in Figure 12.

This rule expands the body M of the function it specializes with fresh variables so that it becomes fully-applied. This works fine if M is a function symbol or a variable, but M can also be a *case* expression (see the grammar in Table 1). In this case, *normalize* uses the rewrite rule shown in Figure 13, as much as needed, to push the variables inside case arms so that case expressions do not occur as operators in function applications.

Both rules *(FunSpec)* and *(ExpandSpec)* help to decrease the occurrences of polymorphic applications since the functional polymorphic arguments of fully-applied applications become monomorphic.

The higher-orderness of programs is addressed by the rules *(EncodeClosed)* and *(ApplVar)*. In the above example, the result of the transformation is the term $id_\Phi\ C^{id}_{int\to int}\ (id\ 7)$, with the declarations:

$$
\begin{array}{l}
\mathbf{datatype}\ T_{int\to int}\ =\ C^{id}_{int\to int} \\
\mathbf{fun}\ id\ x = x \\
\mathbf{fun}\ \ apply_{int\to int}\ x\ y\ =\ \mathbf{case}\ x\ of \\
\qquad C^{id}_{int\to int} \Rightarrow (id\ y) \\
\mathbf{fun}\ id_\Phi\ x\ y\ = apply_{int\to int}\ x\ y
\end{array}
$$

This example can be extended to require the encoding at an arbitrarily high order. Thus a defunctionalization based on function declarations instead of function applications cannot work. It is also clear from this example that the algorithm must be sensitive to the set of monomorphic types at which every function symbol in the program occurs. It is necessary to start with an expression with a monomorphic type and recursively perform type specialization on all function symbols occurring in the monomorphic expression.

The transformation rules require fully-applied applications, thereby intertwining the monomorphization of the higher-order components and their encoding counterparts during the transformation. Consider for example the application *id* (*id id*) (*id* 7). Only the higher-order argument of the application differs from the example above. It is (*id id*) which is a higher-order application itself. However in this context it is a higher-order argument and as such, it has to be encoded by a constructor $C^{(id\ id)}_{int\to int}$. Proceeding as above, we get first:

$$
\begin{array}{l}
\mathbf{datatype}\ T_{int\to int}\ =\ C^{(id\ id)}_{int\to int} \\
\mathbf{fun}\ \ apply_{int\to int}\ x\ y\ =\ \mathbf{case}\ x\ of \\
\qquad C^{(id\ id)}_{int\to int} \Rightarrow id\ id\ y \\
\mathbf{fun}\ id_\Phi\ x\ y\ = apply_{int\to int}\ x\ y
\end{array}
$$

with the term $id_\Phi\ C^{(id\ id)}_{int\to int}\ (id\ 7)$. The application *id id* is fully-applied as *id id y* in the body of the generated $apply_{int\to int}$ function. Because the rules work solely from fully-applied applications, it is only at this point that the defunctionalization of *id id y* is done. This provides the first-order program composed of the term $id_\Phi\ C^{(id\ id)}_{int\to int}\ (id\ 7)$ with the declarations below:

$$
\begin{array}{l}
\mathbf{datatype}\ T_{int\to int}\ =\ C^{(id\ id)}_{int\to int} \mid C^{id}_{int\to int} \\
\mathbf{fun}\ id\ x = x \\
\mathbf{fun}\ \ apply_{int\to int}\ x\ y\ =\ \mathbf{case}\ x\ of \\
\qquad C^{(id\ id)}_{int\to int} \Rightarrow id_\Phi\ C^{id}_{int\to int}\ y \\
\qquad C^{id}_{int\to int} \Rightarrow id\ y \\
\mathbf{fun}\ id_\Phi\ x\ y\ = apply_{int\to int}\ x\ y
\end{array}
$$

Notice that polymorphism can induce arrow type arguments that are syntactically equal but are not of the same type just as in the application *id id id* 7. The two arguments *id* of types $\Psi = (int \to int) \to (int \to int)$ and $int \to int$ are encoded by two different constructors: C^{id}_Ψ and $C^{id}_{int\to int}$ since the constructor symbols are built on both the type and the term they encode. The interested reader can find more realistic examples in Appendix B.

2.5 Encoding nonclosed arrow type arguments

Closed arrow type arguments are always encodable into constant constructors but arrow type arguments may contain variables. The transformation must be able to encode both kinds of arrow type arguments of fully-applied functional applications.

IF

$$\exists\ i,\ i \in 1 \ldots n,\ \mathit{functional}(\Pi_i)$$
$$\wedge \text{ all } \mathit{functional} \text{ variables in } \mathit{functional} \text{ arguments are arguments of } \{\cdot\}_\Psi \text{ for some } \Psi$$
$$\wedge\ n > \mathrm{order}(\Pi) \wedge\ \mathrm{order}(\Omega) = 0 \wedge\ f \text{ is a function symbol}$$

AND

$$\begin{array}{l} \Gamma \vdash t_i\ :\ \Pi_i,\ \forall i,\ i \in 1 \ldots n, \\ \Gamma \vdash \underline{f\ t_1\ \ldots\ t_n}\ :\ \Omega, \\ \Gamma \vdash \underline{f\ :\ \Pi}, \\ \sigma\Pi = \Pi_1 \rightarrow \ldots \rightarrow \Pi_n \rightarrow \Omega \end{array}$$

THEN

$$\underline{f\ t_1\ \ldots\ t_n}, \Delta, \Theta \Longrightarrow \underline{f_{\sigma\Pi}\ \{t_1\}^{-1}\ \ldots\ \{t_n\}^{-1}}, \Delta', \Theta$$

WHERE

$$\Delta' = \begin{cases} \Delta \cup \{\underline{f_{\sigma\Pi}\ x_1\ \ldots\ x_{\mathrm{order}(\Pi)}\ v_1\ \ldots\ v_p =} \\ \mathrm{normalize}(\underline{Mv_1\ \ldots\ v_p})[x_{i_1} \leftarrow \underline{\{x_{i_1}\}_{\Pi_{i_1}}} \ldots, x_{i_k} \leftarrow \underline{\{x_{i_k}\}_{\Pi_{i_k}}}, \\ v_{l_1} \leftarrow \underline{\{v_{l_1}\}_{\Pi_{l_1}}}, \ldots, v_{l_m} \leftarrow \underline{\{v_{l_m}\}_{\Pi_{l_m}}}] \\ \mathit{where}\ \Delta(f) = \underline{f\ x_1\ \ldots\ x_{\mathrm{order}(\Pi)} = M}, \\ i_j,\ j \in\ 1 \ldots k,\ \text{and } l_j,\ j \in\ 1 \ldots m \text{ are the indices of} \\ \text{the } \mathit{functional} \text{ arguments},\quad p = n - \mathrm{order}(\Pi) + 1. \end{cases}$$

Figure 12: (ExpandSpec) Functional η-extension and specialization

Arrow type argument expressions may contain arrow type variables as well as first-order variables. For example an argument could be $t = (\{Z\}_\Omega\ (\{F\}_{int \rightarrow (int \rightarrow int)}\ x))$, where Ω is the type $(int \rightarrow int) \rightarrow int \rightarrow int$. As above, such an arrow type argument must be encoded in a datatype that corresponds to its type. This can be done by encoding the argument as a function constructor rather than as a first-order constructor.

First-order variables are not encoded by the transformation. The types of the values of the first-order variables thus remain variable types and parametric datatypes are generated to encode arrow type terms which contain first-order variables. Thus we are doing as much monomorphization as needed, and no more.

Since functional variable values are encoded, their types are those of the encoding datatypes. In the above example, since t contains a first-order variable, the functional argument t of type $int \rightarrow int$ must be encoded in a parametric datatype $\alpha\ T_{int \rightarrow int}$ by a constructor $C^t_{int \rightarrow int}$ of type

$$(T_\Omega \times T_{int \rightarrow (int \rightarrow int)} \times \alpha) \rightarrow (int \rightarrow int).$$

By encoding functional variable values, the datatypes that are created for the encodings can be recursive (See Section B.1 for an example of such a recursive datatype). The rule *Encode* presented in Appendix A subsumes the rule *EncodeClosed* presented in Figure 10.

2.6 Higher-order constructors

A special case of higher-order application is higher-order constructor application.

A higher-order constructor can be an instance of a polymorphic constructor. For example the list constructor *Cons* has the type $(int \rightarrow int) \times list(int \rightarrow int) \rightarrow list(int \rightarrow int)$ in the application $Cons(id, \{xs\}_{list(int \rightarrow int)})$. The functional argument id of type $int \rightarrow int$ has to be encoded into a constructor $C^{id}_{int \rightarrow int}$ as for an application of a higher-order function. The rule *(Encode)* in Appendix A allows encodings of functional arguments of higher-order functions as well as functional arguments of constructors.

It is by matching the type Π of a term t against the datatype of the patterns in a case expression that we know the functional types of function variables in a pattern. These types are used to apply $\{\cdot\}_\Pi$ to functional variables in the arm bodies. This is accomplished by the rule *(UpdateArms)* of Figure 15. For example:

$$\begin{array}{ll} \textbf{case} & \{x\}_{list\ int \rightarrow int}\ \textbf{of} \\ & Cons(x_1, x_2) \Rightarrow (x_1\ y) \end{array}$$

$$\Longrightarrow$$

$$\begin{array}{ll} \textbf{case} & x\ \textbf{of} \\ & Cons(x_1, x_2) \Rightarrow (\{x_1\}_{int \rightarrow int}\ y) \end{array}$$

By matching the functional type $list(int \rightarrow int)$ of x against the parametric type $list\ \alpha$ with the substitution $\sigma = \{\alpha \leftarrow (int \rightarrow int)\}$. The type domain of the data constructor *Cons* is then specialized into $(int \rightarrow int) \times list(int \rightarrow int)$, allowing the rule *(UpdateArms)* to apply $\{\cdot\}_{int \rightarrow int}$ to the arrow type variable x_1. Tuple patterns are treated in the same way. The interested reader can consider the examples B.2 and B.3.

Notice that the type $list(int \rightarrow int)$ is considered as *functional* (see Figure 14) though it is of type order 0. Only term arguments of type order greater than 0 need to be encoded but any term of a functional type may be an argument of a polymorphic function which, in this case, has to be type specialized.

There is a minor complication when a datatype declares a functional constructor explicitly like the constructor *Store* in the declaration: **datatype** $(\alpha, \beta)\ store = Store\ \alpha \rightarrow \beta$. Unlike a function symbol, a constructor cannot have clones in datatypes corresponding to different type instances of α and β. A way around this is to generalize such a datatype. Generalization is safe for type inference. Moreover since the programs are type correct, it is useless to typecheck the arrow. By the rule *(GeneralizeArrows)* of Figure 16 the

IF

$$functional(T\Phi_1 \ldots \Phi_k)$$
$$\wedge \text{ all } functional \text{ variables in } t \text{ are arguments of } \{\cdot\}_\Psi \text{ for some } \Psi$$

AND

$$\Gamma \vdash\ t : T\Phi_1 \ldots \Phi_k$$

THEN

$$\begin{array}{l} \underline{case\ t\ of} \\ \quad \underline{C_1 x_{1_1} \ldots x_{1_{m_1}} \Rightarrow t_1} \\ \quad \vdots \\ \quad \underline{C_n x_{n_1} \ldots x_{n_{m_n}} \Rightarrow t_n,} \end{array} , \Delta, \Theta \Longrightarrow \begin{array}{l} \underline{case\ \{t\}^{-1}\ of} \\ \quad \underline{C_1 x_{1_1} \ldots x_{1_{m_1}} \Rightarrow t'_1} \\ \quad \vdots \\ \quad \underline{C_n x_{n_1} \ldots x_{n_{m_n}} \Rightarrow t'_n} \end{array} , \Delta, \Theta$$

WHERE

$$T \text{ is declared } \underline{\textbf{datatype}\ \alpha_1 \ldots \alpha_k\ T = \blacksquare}$$

$$\forall\ i,\ i \in 1 \ldots n,\ t'_i = t_i[x_{i_{j_1}} \leftarrow \underline{\{x_{i_{j_1}}\}_{\Psi_{j_1}}}, \ldots, x_{i_{j_p}} \leftarrow \underline{\{x_{i_{j_p}}\}_{\Psi_{j_p}}}]$$
$$\begin{cases} C_i \text{ appears as } C_i \Psi_1 \ldots \Psi_{m_i} \text{ in the instantiated sumtype } \blacksquare[\alpha_1 \leftarrow \Phi_1, \ldots, \alpha_k \leftarrow \Phi_k] \\ j_l,\ l \in 1 \ldots p \text{ are the indices of functional pattern variables.} \end{cases}$$

Figure 15: (UpdateArms) rule for updating case arm variables

datatype *store* becomes **datatype** α *store* = *Store* α so that encoding of differently typed functional arguments in different applications of the constructor *Store* is possible.

Rules *(FunSpec)* and *(ExpandSpec)* introduce applications of $\{\cdot\}_\Psi$ to functional terms. These applications are ultimately removed when the functional term is applied (in *(ApplVar)*), when the term is examined (in *(UpdateArms)*), or by applying $\{\cdot\}^{-1}$ (in *(FunSpec)* and *(ExpandSpec)*). However, applications of $\{\cdot\}_\Psi$ are not removed when an (unapplied) functional term is used as an argument to a higher-order constructor, as in $Cons(\{f\}_\Psi, Nil)$. For this, we use the rule *(UpdateCon)*, as shown in Figure 17.

2.7 The use of types

In summary, types are used by the defunctionalization algorithm

- to replace arrow type arguments of higher-order function applications by appropriate elements of a datatype: A datatype T_Π is created for functional arguments of type Π,
- to create clones of polymorphic higher-order functions specialized by the types of their functional arguments; the clone names are simply labelled by their types,
- to recover the appropriate clone name in a recursive call that occurs in (mutually) recursive clones of higher-order function declarations,
- to recognize the datatype in which is encoded the value of a arrow type variable so that its application can be replaced by an application of an *apply* function,
- to know when a clone must be an η-extended copy of the original polymorphic declaration, and
- to discriminate arrow type arguments of constructors in different arms of a **case** expression by analysing the type of the matched expression.

The encoding of arrow type arguments into datatypes, together with a type analysis originated by fully-applied applications, accommodates the transformation of higher-order programs into a first-order equivalent program.

3 Study of the transformation

A transformation rule transforms a program which contains an expression $\underline{e}$ in the context P denoted by $P[\underline{e}]$. In the previous section, a transformation rule has been written in the following abbreviated form:

$$if\ C \quad \underline{e}, \Delta, \Theta \Longrightarrow \underline{e'}, \Delta', \Theta'$$

Suppose that a function ζ extracts the set Δ of the function declarations from the program and that a function ϑ extracts the set Θ of type declarations from the program. As usual, the notation $M[N'/N]$ denotes that the occurrence(s) of the subterm N are replaced by the subterm N' in M. A transformation rule on a program $P[\underline{e}]$ could be expressed as:

$$if\ C \quad P[\underline{e}] \Longrightarrow P[\underline{e'}][\Delta'/\zeta(P[\underline{e}]), \Theta'/\vartheta(P[\underline{e}])]$$

Given a program, the defunctionalization algorithm applies the six rules {*FunSpec, ExpandSpec, Encode, ApplVar, UpdateArms, GeneralizeArrows*} in any order until none of them is applicable. It relies upon a type inference system which infers the type of an expression according to the Hindley-Milner algorithm while taking into account type information introduced by the transformation. For that, the type inference system is given the inference rule of Figure 6 and the following two additional rules: The first rule is for the type of an *apply* function. The label of an *apply* function indicates the type of the functional encoding term Ψ. The type of the encoded term is the datatype T_Ψ. Therefore:

$$\vdash\ apply_\Psi\ :\ T_\Psi \rightarrow \Psi$$

The second rule is for the type of a clone function symbol:

$$\vdash\ f_\Psi\ :\ \Psi$$

IF

$$\exists\ j,\ j \in 1 \ldots n, \wedge\ \text{order}(\Phi_j) > 0$$
$$\wedge \text{ all variables of } t_j \text{ are arguments of } \{\cdot\}_\Psi \text{ for some } \Psi$$

THEN

$$\underline{C\ t_1\ \ldots\ t_n}, \Delta, \Theta \cup \{\alpha_1 \ldots \alpha_m\ T = \ldots \mid C\ \Phi_1 \times \ldots \times \Phi_j \times \ldots \times \Phi_n \mid \ldots\}$$
$$\Longrightarrow$$
$$\underline{C\ t_1\ \ldots\ t_n}, \Delta, \Theta \cup \{\beta_1 \ldots \beta_k\ T = \ldots \mid C\ \Phi_1 \times \ldots \times \beta \times \ldots \times \Phi_n \mid \ldots\}$$

WHERE

$\alpha_1 \ldots \alpha_m$ are the type variables in $\{\ldots \mid C\ \Phi_1 \times \ldots \times \Phi_j \times \ldots \times \Phi_n \mid \ldots\}$
$\beta_1 \ldots \beta_k$ are the type variables in $\{\ldots \mid C\ \Phi_1 \times \ldots \times \beta \times \ldots \times \Phi_n \mid \ldots\}$
β is a fresh type variable

Figure 16: (GeneralizeArrows) Arrow constructor generalization

IF

$\exists\ j,\ j \in 1 \ldots n,\ \mathit{functional}(\Pi_j)$
$\wedge$ all *functional* variables in *functional* arguments are arguments of $\{\cdot\}_\Psi$ for some Ψ
$\wedge$ c is a constructor

AND

$$\Gamma \vdash t_i\ :\ \Pi_i,\ \forall i,\ i \in 1 \ldots n,$$

THEN

$$\underline{c\ t_1\ \ldots\ t_n}, \Delta, \Theta \Longrightarrow \underline{c\ \{t_1\}^{-1}\ \ldots\ \{t_n\}^{-1}}, \Delta, \Theta$$

Figure 17: (UpdateCon) Removes applications of $\{\cdot\}_\Psi$ from higher-order constructor arguments

The following paragraphs address the issues of soundness, termination and effectiveness of the transformation.

3.1 Soundness

Theorem 1 *If a program is well typed according to the Hindley-Milner algorithm, then the transformation results in a well-typed equivalent program.*

Here, by *equivalent*, we mean *have the same result when evaluated.*

Rules *(FunSpec)*, *(ExpandSpec)* and *(UpdateArms)* preserve type. Rule *(Encode)* introduces a confusion between function type Φ and a datatype T_Φ for function variables. However, when no rules apply, every term $\{t\}_\Phi$ has been changed into a variable of type T_Φ by application of the rule *(ApplVar)*.

For proof of the preservation of the equivalence, we compare the reductions of e by an evaluator **eval**. Depending on the chosen evaluation order, the function **evid** means either **eval** or the identity. In this way, the proof is independent of a particular semantics. The function **app** is an evaluator which applies an evaluated function to a set of evaluated arguments. We prove that transformation rules preserve evaluation by induction on the structure of an application.

Proof:

- **Rules** *(FunSpec)*, *(ExpandSpec)* **and** *(UpdateArms)* **address only typing issues so they have no impact on the evaluation.**

- **The encoding made by** *(Encode)* **does not change the evaluation assuming that the evaluation of application of encoding term and application simulated by the** *apply* **function to the encoded term are equivalent. This last assumption corresponds to the transformation made by the rule** *(ApplyVar)*. **Suppose that variable F in the application $e = F\ t_1\ \ldots\ t_n$ is bound to t in the environment, then the evaluation of $[\![e]\!]$ reduces to:**

$$\textbf{app}\ (\textbf{eval}\ [\![\ t\ ]\!])\ \textbf{evid}\ [\![\ t_1\ ]\!]\ \ldots\ \textbf{evid}\ [\![\ t_n\ ]\!]$$

The rule transforms e into $e' = apply_{T_\Pi}\ F\ t_1\ \ldots\ t_n$. But here F is an encoding of the term t' result of the transformation of t. Suppose $C^u_\Pi\ u_1\ \ldots\ u_m$ is the encoding of t', and the constructor C^u_Π belongs to the datatype T_Π. The term u is the encoded term so t' is an instance of u: $t' = \sigma(u)$. Let $x_1, \ldots, x_m$ be the variables of u. Then the substitution σ is $\{x_1 \leftarrow u_1, \ldots, x_m \leftarrow u_m\}$, and the declaration of $apply_{T_\Pi}$ contains the arm:

$$C_u\ u_1\ \ldots\ u_m \Rightarrow t'\ x_1\ \ldots\ x_n$$

After pattern-matching with the substitution σ, e' reduces to:

$$\textbf{eval}\ [\![\ \sigma(u)\ \textbf{evid}\ [\![\ t_1\ ]\!]\ \ldots\ \textbf{evid}\ [\![\ t_n\ ]\!]\ ]\!].$$

which reduces to

$$\textbf{app}\ (\textbf{eval}\ [\![\ \sigma(u)\ ]\!])\ \textbf{evid}\ [\![\ t_1\ ]\!]\ \ldots\ \textbf{evid}\ [\![\ t_n\ ]\!]$$

Since $\sigma(u) = t'$, and since by induction t and t' have the same results e and e' have the same results.

□

Notice that, if efficiency is counted as a number of reduction steps then the transformed first-order program is slightly less efficient than the source higher-order program since there are supplementary reduction steps for pattern matching the case expressions in *apply* functions.

3.2 Termination

Theorem 2 *The transformation always terminates.*

Proof: We consider the least partial quasi-ordering on term $\succsim$ which enjoys the subterm property, is closed under context and extends the following partial ordering on terms:

$$x \succ \underline{x_{:\Pi}} \tag{1}$$

$$\underline{v_{:\Pi}}\ t_1 \ldots t_n \succ \underline{apply_{\Pi}\ v\ t_1 \ldots t_n} \tag{2}$$

$$t \succ C^t_{\Pi},\ where\ \Gamma \vdash t : \Pi \tag{3}$$

$$\underline{f\ t_1 \ldots t_n} \succ \underline{f_{\Pi}\ t_1 \ldots t_n} \tag{4}$$

The associated $\sim$ is equivalence by η-extension. This quasi-ordering is well-founded since $\succ$ is well-founded.

Consider the multiset $\{M_0, M_1, \ldots, M_n\}$ of a term M_0 and the term bodies of its function declarations, we prove that if $\{M_0, M_1, \ldots, M_n\} \Longrightarrow \{M'_0, M'_1, \ldots, M'_n\}$, then $\{M_0, M_1, \ldots, M_n\} \gg \{M'_0, M'_1, \ldots, M'_n\}$ where $\gg$ is the multiset ordering induced by $\succ$.

- Rules *(FunSpec)* and *(ExpandSpec)* transform a subterm t of an element M_i into $M_i[t'/t]$, $t \succ t'$ by (4), so $M_i \succ M_i[t'/t]$. If a clone $M_{j_\Pi}\ v_1 \ldots v_p$ ($p \geq 0$) is added to the multiset, $M_j \succ M_{j_\Pi} \sim M_{j_\Pi}\ v_1 \ldots v_p$ by (1).
- Rules *(UpdateArms)* and *(ApplVar)* transform a subterm t of an element M_i into $M_i[t'/t]$, $t \succ t'$ respectively by (1) and by (2), so $M_i \succ M_i[t'/t]$.
- Rule *(Encode)* transforms a subterm t of an element M_i into $M_i[t'/t]$, $t \succ t'$ by (3), so $M_i \succ M_i[t/t']$ of one M_i by (3). Moreover, the rule adds an arm body $t\ y_1 \ldots y_k$ to the *apply* function, but $M_i \succ t \sim t\ y_1 \ldots y_k$

□

3.3 Effectiveness:

Theorem 3 *The transformation of a closed program results in a first-order program.*

A closed program is composed of a fully-applied closed term e[1] together with its declarations D. Suppose no transformation rules apply. Applications in e and in declaration bodies cannot have any arrow type arguments since *(Encode)* does not apply. Therefore no variables in declaration bodies can be of an arrow type so that no function symbols denote higher-order functions.

[1]This theorem remains valid if e has free first-order variables

4 Conclusion and future work

The defunctionalization transformation presented in this paper is a complete algorithm for transforming a closed higher-order well-typed functional program, comprising an expression e together with its declarations, into an equivalent first-order program. As far as we know, a complete algorithm such as this has not been presented before. The method that replaces functional applications by macros [Wad88] is elegant but macros cannot be recursive. Although recursion can be recovered by way of recursive local functions, the macro method supports only functional arguments which remain identical in recursive calls. The method that specializes functional applications with respect to the values of arrow type arguments is limited to so called *variable-only* arrow type arguments [CD]. None of these methods consider the case of higher-order constructor applications.

Our transformation is based on Reynolds's method [Rey72] of encoding functional arguments. Our main contribution is to bring together this idea and the idea of using functional application types to drive the defunctionalization transformation. This is crucial for handling polymorphic higher-order functions as has been noted by Chin and Darlington in their $\mathcal{A}$-algorithm [CD], which is used to remove some functional results by *eta*-expansion. Our transformation includes the functionality of the $\mathcal{A}$-algorithm.

While it always produces a first-order program, this transformation has little effect on execution efficiency since the reduction steps of the first-order program are similar to the reduction steps of the original higher-order program. The only gains in performance come from removing the penalties incurred by the implementation of higher-order functions. In contrast, Chin and Darlington's $\mathcal{R}$ algorithm [CD] relies on specialization with respect to the values of functional arguments and returns, when it is applicable, an improved first-order program.

The ideal solution is to add to our set of rules a transformation rule to specialize *variable-only* arrow type arguments with respect to their value to get the best of both worlds. For example, the first argument of *map* in the introductory example in Section 2 is variable-only, as is the first argument of *mp* in the example in Section B.1. Therefore in applications of *map* or *mp*, the functions *map* and *mp* can be specialized with respect to the value of their actual functional parameters rather than encoding them and consequently creating an *apply* function that corresponds to this encoding. At a functional application of f, *variable-only* functional arguments lead to a clone of f specialized with respect to their values. In a combined transformation, the values of the *variable-only* parameters of the application would be substituted in the clone body whereas other functional arguments would lead to a clone of f specialized with respect to their types, their values being encoded into a constructor term of a datatype. In the combined transformation, since a clone is tied to its source application type, the folding of a recursive clone application either coming from type specialization or from value specialization is always recognizable by its type. So, the type annotations and the variable-only analysis of the version body together enable the algorithm to fold the recursive calls in recursive as well as in mutually-recursive versions. We suggest performing the *variable-only* analysis beforehand and to carry on a *variable-only* annotation to the functional arguments of functional versions. The result of applying such a combined transformation can be seen in the example in Section B.1.

Note that the defunctionalization transformation performs a *monomorphization* of functions with respect to their functional arguments and functional results. Full monomorphization of the program can be obtained by specializing also first order function symbol with respect to the type of their applications and annotating first-order variables as well as functional variables.

The defunctionalization transformation, we present in this paper, is a step in a pipe-line of transformations designed to automatically derive a program generator [B+94, KBB+94] from the semantics of a *domain-specific design language*. The purpose of the transformation is to obtain satisfactory performance and to tailor the implementation to a specific platform and software environment. Defunctionalization accommodates software environments which penalize or prohibit functionals. It is also used to translate functional programs into term-rewriting systems in the transformation system Astre [Bel95b, Bel95a] which uses term-rewriting techniques to perform algebraic manipulation on functional programs.

References

[B+94] J. Bell et al. Software Design for Reliability and Reuse: A proof-of-concept demonstration. In *TRI-Ada '94 Proceedings*, pages 396–404. ACM, November 1994.

[BBH96] Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. Technical report, OGICSE, 1996.

[Bel94] J. M. Bell. An Implementation of Reynold's Defunctionalization Method for a Modern Functional Language. Master's thesis, Department of Computer Science and Engineering, Oregon Graduate Institute, January 1994.

[Bel95a] F. Bellegarde. Astre: Towards a fully automated program transformation system. In *Proceedings of the sixth conference on Rewriting Techniques and Applications*, volume 914 of *LNCS*. Springer-Verlag, 1995.

[Bel95b] F. Bellegarde. Automatic Synthesis by Completion. In *Journées Francophones sur les Langages Applicatifs*, INRIA, collection didactiques, 1995.

[BH94a] Jeffrey M. Bell and James Hook. Defunctionalization of typed programs. Technical Report 94-025, Department of Computer Science and Engineering, Oregon Graduate Institute, February 1994.

[BH94b] Françoise Bellegarde and James Hook. Substitution: A formal methods case study using monads and transformations. *Science of Computer Programming*, 23(2–3):287–311, 1994.

[CD] W. Chin and J. Darlington. Higher-Order Removal: A modular approach. To appear in Lisp and Symbolic Computation.

[Joh85] Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In J-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 190–203. Springer-Verlag, 1985.

[KBB+94] Richard B. Kieburtz, Françoise Bellegarde, Jef Bell, James Hook, Jeffrey Lewis, Dino Oliva, Tim Sheard, Lisa Walton, and Tong Zhou. Calculating software generators from solution specifications. Technical Report OGI-CSE-94-032B, Department of Computer Science and Engineering, Oregon Graduate Institute, October 1994.

[Mil78] R. Milner. A theory of type polymorphism. *Journal of Computer and System Science*, pages 348–375, 1978.

[PS87] A. Pettorossi and A. Skowron. Higher order generalization in program derivation. In Springer Verlag, editor, *Proceedings of TAPSOFT '87: Theory and Practice of Software Development*, volume 250 of *LNCS*, 1987.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM National Conference*, pages 717–740. ACM, 1972.

[Wad88] P. Wadler. Deforestation: Transforming Programs to eliminate trees. In *Proceedings of the second European Symposium on Programming ESOP'88*, volume 300 of *LNCS*. Springer-Verlag, 1988.

A (Encode) rule for encoding functional arguments

IF

$$\exists\ j,\ j \in 1 \ldots n,\ t_j \text{ is not a variable} \wedge\ order(\Pi_j) > 0$$
$$\wedge \text{ all } \mathit{functional} \text{ variables in } t_j \text{ are arguments of } \{\cdot\}_\Psi \text{ for some } \Psi$$
$$\wedge\ F \text{ is a function symbol, a function variable, or a data constructor}$$
$$\wedge\ \mathbf{order}(\Omega) = 0$$

AND

$$\forall i,\ i \in 1 \ldots n, \Gamma \vdash t_i\ :\ \Pi_i,$$
$$\Gamma \vdash \underline{F\ t_1\ \ldots,\ t_n}\ :\ \Omega,$$
$$v_1\ \ldots\ v_k \text{ are the variables in } t_j$$

THEN

$$\underline{F\ t_1\ \ldots\ t_j \ldots\ t_n}, \Delta, \Theta \Longrightarrow \underline{F\ t_1\ \ldots\ (C^{t_j}_{\Pi_j}\ v_1\ \ldots\ v_k)\ \ldots\ t_n}, \Delta', \Theta'$$

WHERE

$$\Delta' = \begin{cases} \mathit{if}\ \mathit{Apply}_{\Pi_j} \text{has not been declared in } \Delta\ \mathit{then} \\ \Delta \cup \{\underline{\mathit{Apply}_{\Pi_j}\ y\ x_1\ \ldots\ x_{\mathbf{order}(\Pi_j)}} = \\ \quad \underline{\mathit{case}\ y\ \mathit{of}\ C^{t_j}_{\Pi_j}\ v_1 \ldots v_k\ \Rightarrow\ t_j\ x_1 \ldots\ x_{\mathbf{order}(\Pi_j)}}\} \\ \mathit{else} \text{ add to it the arm } \underline{C^{t_j}_{\Pi_j}\ v_1 \ldots v_k\ \Rightarrow\ t_j\ x_1 \ldots\ x_{\mathbf{order}(\Pi_j)}} \end{cases}$$

$$\Theta' = \begin{cases} \mathit{if}\ T_{\Pi_j} \text{has not been declared in } \Theta\ \mathit{then} \\ \Theta \cup \{\underline{\mathbf{datatype}\ \alpha_1 \ldots \alpha_m\ T_{\Pi_j} = C^{t_j}_{\Pi_j}\ \Phi_1 \times \ldots \times \Phi_k}\} \\ \mathit{where} \begin{cases} \forall\ i, i \in 1 \ldots k, \Phi_i = \begin{cases} \mathit{if}\ v_i \text{ is annotated by } \Psi \\ \mathit{then}\ \Psi \\ \mathit{else}\ \beta\ \mathit{where}\ \beta \text{ is a fresh type variable} \end{cases} \\ \alpha_1 \ldots \alpha_m \text{ are the type variables in } \Phi_1 \ldots \Phi_k \end{cases} \\ \mathit{else} \text{ add to it the constructor } C^{t_j}_{\Pi_j}\ \Phi_1 \times \ldots \times \Phi_k \end{cases}$$

B Examples

B.1 Second-order argument:

This example is inspired from [BH94b].
fun $mp\ Z\ F\ x =$ **case** x **of**
$\quad Nil \Rightarrow Nil$
$\quad |\ \ Cons(x, xs) \Rightarrow Cons(F\ x, mp\ Z\ (Z\ F)\ xs)$
fun $db\ F\ x = F\ (F\ x)$
fun $inc\ x = x + 1$ **with the term** $mp\ db\ inc\ [2,3,4]$ **of type:** $list\ int$**,**
becomes
datatype $T_{int \to int} = C^{inc}_{int \to int}\ |\ C^{(Z\ F)}_{int \to int}\ T_{(int \to int) \to (int \to int)} \times T_{int \to int}$
datatype $T_{(int \to int) \to (int \to int)} = C^{db}_{(int \to int) \to (int \to int)}$

fun $mp_\Phi\ Z\ F\ x =$ **case** x **of**
$\quad Nil \Rightarrow Nil$
$\quad |\ \ Cons(x, xs) \Rightarrow Cons(apply_{T_{int \to int}}\ F\ x, mp_\Phi\ Z\ (C^{(Z\ F)}_{int \to int}\ (Z, F))\ xs)$
fun $apply_{T_{int \to int}}\ F\ x =$ **case** F **of**
$\quad C^{inc}_{int \to int} \Rightarrow inc\ x$
$\quad |\ \ C^{(Z\ F)}_{int \to int}(Z, G) \Rightarrow\ (apply_{T_{(int \to int) \to (int \to int)}}\ Z\ G\ x)$
fun $apply_{T_\Psi}\ Z\ F\ x =$ **case** Z **of**
$\quad C^{db}_{(int \to int) \to (int \to int)} \Rightarrow db_\Psi\ F\ x$
fun $db_\Psi\ F\ x = apply_{T_{int \to int}}\ F\ (apply_{T_{int \to int}}\ F\ x)$
fun $inc\ x = x + 1$
with the term: $mp_\Phi\ C^{db}_{(int \to int) \to (int \to int)}\ C^{inc}_{int \to int}\ [2,3,4]$
$\Phi = ((int \to int) \to int \to int) \to (int \to int) \to list\ int \to list\ int$**,**
$\Psi = (int \to int) \to int \to int$
If combined with specialization with respect to the value of the *variable-only* **first argument of** mp**,**

this program becomes:

datatype $T_{int\to int} = C^{inc}_{int\to int} \mid C^{(db\ F)}_{int\to int}\ T_{int\to int}$
fun $mp_\Phi\ F\ x =$ case x of
$Nil \Rightarrow Nil$
$\mid\ Cons(x, xs) \Rightarrow Cons(apply_{T_{int\to int}}\ F\ x, mp_\Phi\ (C^{(db\ F)}_{int\to int}\ (db, F))\ xs)$
fun $apply_{T_{int\to int}}\ F\ x =$ case F of
$C^{inc}_{int\to int} \Rightarrow inc\ x$
$\mid\ (C^{(db\ F)}_{int\to int}\ G) \Rightarrow\ (db_\Psi\ G\ x)$
fun $db_\Psi\ f\ x = apply_{T_{int\to int}}\ F\ (apply_{T_{int\to int}}\ F\ x)$
fun $inc\ x = x + 1$
with the term: $mp_\Phi\ C^{inc}_{int\to int}\ [2,3,4]$
where $\Phi = ((int \to int) \to int \to int) \to (int \to int) \to list\ int \to list\ int$.

B.2 List of functions:

This example is borrowed from [CD].
fun $maph\ Fs\ y\ =$ case Fs of
$Nil \Rightarrow Nil$
$\mid\ Cons(F, Fs) \Rightarrow Cons(F\ y, maph\ Fs\ y)$
fun $add5\ y\ =$ case y of
$Nil \Rightarrow Nil$
$\mid\ Cons(x, xs) \Rightarrow Cons(k\ x, add5\ xs)$
fun $k\ x\ z = z\ +\ 5 * x$
with the term $maph\ (add5\ xs)\ y$ of type: $list\ int$,

becomes:
datatype $\alpha\ T_{int\to int} = C^{(k\ x)}_{int\to int}\ \alpha$
fun $maph_\Phi\ Fs\ y\ =$ case Fs of
$Nil \Rightarrow Nil$
$\mid\ Cons(F, Fs) \Rightarrow Cons(apply_{T_{int\to int}}\ F\ y, maph_\Phi\ Fs\ y)$
fun $add5\ y\ =$ case y of
$Nil \Rightarrow Nil$
$\mid\ Cons(x, xs) \Rightarrow Cons(C^{(k\ x)}_{int\to int}\ x, add5\ xs)$
fun $apply_{T_{int\to int}}\ F\ y\ =$ case F of
$C^{(k\ x)}_{int\to int}\ x \Rightarrow k\ x\ y$
fun $k\ x\ z = z\ +\ 5 * x$ with the term: $maph_\Phi\ (add5\ xs)\ y$
where $\Phi = list\ (int \to int) \to int\ list \to list\ int$.

B.3 Pair of functions:

This example is borrowed from [PS87]. The term case $(fmin\ t)$ of $(F, m) \Rightarrow (F\ m)$ with the declarations:

fun $fmin\ t =$ case t of
$Leaf\ a \Rightarrow (Leaf, a)$
$\mid\ Tree\ \ (t1, t2) \Rightarrow$
case $(fmin\ t1)$ of
$(F1, m1) \Rightarrow$ case $fmin\ t2$ of
$(F2, m2) \Rightarrow (k\ F1\ F2, min(m1, m2))$
fun $k\ F\ G\ x = Tree\ (F\ x, G\ x)$

becomes:
case $(fmin_\Phi\ t)$ of$(F, m) \Rightarrow (apply_{T_{tree\ int\to int}}\ F\ m)$
where $\Phi = tree\ int \to (int \to tree\ int) \times int$ with declarations:
datatype $T_{tree\ int\to int} = C^{Leaf}_{tree\ int\to int} \mid C^{k}_{tree\ int\to int}\ T_{tree\ int\to int}\ \times\ T_{tree\ int\to int}$
fun $fmin_\Phi\ t =$ case t of
$Leaf\ a \Rightarrow (C^{Leaf}_{tree\ int\to int}\ a)$
$\mid\ Tree\ \ (t1, t2) \Rightarrow$
case $(fmin_\Phi\ t1)$ of
$(F1, m1) \Rightarrow$ case $(fmin_\Phi\ t2)$of
$(F2, m2) \Rightarrow (C^{k}_{tree\ int\to int}\ (F1, F2), min(m1, m2))$
fun $apply_{T_{tree\ int\to int}}\ F\ m =$ case F of

$C^{Leaf}_{tree\ int \to int} \Rightarrow (Leaf\ m)$
$\mid\ C^{k}_{tree\ int \to int}\ (F1, F2) \Rightarrow (k\ \ F1\ \ F2\ \ m)$
fun $k\ F\ G\ m = Tree\ (apply_{Ttree\ int \to int}\ \ F\ \ m, apply_{Ttree\ int \to int}\ \ G\ \ m)$

B.4 Mutually recursive functions:

datatype $\alpha\ dec = Dec\ \alpha \times exp\ \alpha$
datatype $\alpha\ exp = Var\ \alpha \mid App\ exp\ \alpha \times\ exp\ \alpha \mid Let\ dec\ \alpha \times exp\ \alpha$
fun $fold\text{-}dec\ D\ \ V\ \ A\ \ L\ x =$ **case** x **of**
$Dec(v, x) \Rightarrow \mathrm{D}\ \ v\ \ (fold\text{-}exp\ \ D\ \ V\ \ A\ \ L\ \ x)$

fun $fold\text{-}exp\ D\ \ V\ \ A\ \ L\ x =$ **case** x **of**
$Var\ v \Rightarrow V\ v$
$|App\ (y, z) \Rightarrow A\ (fold\text{-}exp\ D\ \ V\ \ A\ \ L\ \ y)\ (fold\text{-}exp\ D\ \ V\ \ A\ \ L\ \ z)$
$|Let\ (x, z) \Rightarrow \mathrm{Let}\ (fold\text{-}dec\ D\ \ V\ \ A\ \ L\ \ x)\ (fold\text{-}exp\ D\ \ V\ \ A\ \ L\ \ z)$
and the term $fold\text{-}exp\ proj2\ unit\ append\ append\ (Var\ 'x')$,

becomes:

$fold\text{-}exp_{\Pi}\ \ C^{proj2}_{\Delta \to \Sigma \to \Sigma}\ \ C^{unit}_{string \to \Sigma}\ \ C^{append}_{\Sigma \to \Sigma \to \Sigma}\ \ C^{append}_{\Sigma \to \Sigma \to \Sigma}\ \ (Var\ 'x')$,
$\Pi = (\Delta \to \Sigma \to \Sigma) \to (string \to \Sigma) \to (\Sigma \to \Sigma \to \Sigma) \to (\Sigma \to \Sigma \to \Sigma) \to \Theta\ \ \to \Sigma$,
$\Psi = (\Delta \to \Sigma \to \Sigma) \to (string \to \Sigma) \to (\Sigma \to \Sigma \to \Sigma) \to (\Sigma \to \Sigma \to \Sigma) \to \Delta \to \Sigma$,
$\Delta = dec\ string$, $\Theta = exp\ string$, **and** $\Sigma = list\ string$,
and the added declarations:
datatype $T_{\Delta \to \Sigma \to \Sigma} = C^{proj2}_{\Delta \to \Sigma \to \Sigma}$
datatype $T_{string \to \Sigma} = C^{unit}_{string \to \Sigma}$
datatype $T_{\Sigma \to \Sigma \to \Sigma} = C^{append}_{\Sigma \to \Sigma \to \Sigma}$
fun $fold\text{-}dec_{\Psi}\ \ D\ \ V\ \ A\ \ L\ \ x =$ **case** x **of**
$Dec(v, x) \Rightarrow apply_{\Delta \to \Theta \to \Sigma}\ \ D\ v\ \ (fold\text{-}exp_{\Pi}\ \ D\ \ V\ \ A\ \ L\ \ x)$

fun $fold\text{-}exp_{\Pi}\ \ D\ \ V\ \ A\ \ L\ \ x =$ **case** x **of**
$Var\ v \Rightarrow apply_{string \to \Sigma}\ \ V\ v$
$|App\ (y, z) \Rightarrow apply_{\Sigma \to \Sigma \to \Sigma}\ \ A\ \ (fold\text{-}exp_{\Pi}\ \ D\ \ V\ \ A\ \ L\ \ y)\ \ (fold\text{-}exp_{\Pi}\ \ D\ \ V\ \ A\ \ L\ \ z)$
$|Let\ (x, z) \Rightarrow apply_{\Sigma \to \Sigma \to \Sigma}\ \ \mathrm{L}\ \ (fold\text{-}dec|\Psi\ \ D\ \ V\ \ A\ \ L\ \ x)\ \ (fold\text{-}exp_{\Pi}\ \ D\ \ V\ \ A\ \ L\ \ z)$

fun $apply_{\Delta \to \Sigma \to \Sigma}\ \ v\ \ x\ \ z =$ **case** v **of**
$C^{id}_{\Delta \to \Sigma \to \Sigma} \Rightarrow proj2\ \ x\ \ z$

fun $apply_{string \to \Sigma}\ \ v\ \ x =$ **case** v **of**
$C^{unit}_{string \to \Sigma} \Rightarrow unit\ \ x$

fun $apply_{\Sigma \to \Sigma \to \Sigma}\ \ v\ \ x\ \ y =$ **case** v **of**
$C^{append}_{\Sigma \to \Sigma \to \Sigma} \Rightarrow append\ \ x\ \ y$