January 1996

Type-driven defunctionalization

Jeffrey M. Bell
Francoise Bellegarde
James Hook

Follow this and additional works at: http://digitalcommons.ohsu.edu/csetech

Recommended Citation
Bell, Jeffrey M.; Bellegarde, Francoise; and Hook, James, "Type-driven defunctionalization" (1996). CSETech. 82.
http://digitalcommons.ohsu.edu/csetech/82

This Article is brought to you for free and open access by OHSU Digital Commons. It has been accepted for inclusion in CSETech by an authorized administrator of OHSU Digital Commons. For more information, please contact champieu@ohsu.edu.
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, bellegarde, 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 realize 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 source 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.

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-
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:  
\[ \text{ddclos} := \text{datatype} \alpha_1 \ldots \alpha_n \ T = \text{ddclos} \]
\[ \text{ddclos} := \text{ddclos} \times \text{ddclos} \]
\[ \text{ddclos} := \text{C type}_1 \times \ldots \times \text{type}_n \ \ (n \geq 0) \]

Functions:  
\[ \text{fdcl} := \text{fun} \ v_1 \ldots v_n = \text{term} \]

Terms:  
\[ \text{term} := \text{rator} \ \text{term}_1 \ldots \text{term}_n \ \ (n \geq 0) \]
\[ \text{rator} := \text{f} \ \text{v} \ C \]
\[ \text{pat} := \text{C} \ \text{v}_1 \ldots \text{v}_n \ \ (n \geq 0) \]

Types:  
\[ \text{type} := \alpha \rightarrow \text{type}_1 \rightarrow \text{type}_2 \rightarrow \text{type}_3 \ldots \rightarrow \text{type}_n \rightarrow \text{type}_1 \times \text{type}_2 \]

Program:  
\[ \text{program} := \text{ddclos}^* \ \text{fdcl}^* \ \text{term} \]

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 \(T\) captures the arrow type arguments of the arrow type \(\text{type}\) where each arrow type argument of type \(\text{type}\) is encoded by a constructor in the datatype \(T\). Consider the following example of a term: \(\text{map id [1, 2] where map is declared as in Figure 1.} \)
The type of \text{map} is:
\[ \text{map} : (\alpha \rightarrow \beta) \rightarrow \text{list} \alpha \rightarrow \text{list} \beta \]

It is a higher-order function since it has an arrow type argument. The type of \text{id} is \(\alpha \rightarrow \alpha\). But in the context of the application \text{map id [1, 2]} \text{id} is instantiated at type \(\text{int} \rightarrow \text{int}\). Therefore, this occurrence of \text{id} can be encoded by a constructor \(C_{\text{id}}\) in a datatype \(T_{\text{int}}\) which is created to contain the encodings of arrow type arguments of type \(\text{int} \rightarrow \text{int}\). So doing, the type of \text{map} has to become \(T_{\text{int}} \rightarrow \text{list} \rightarrow \text{list} \). However, in another application, the type of \text{map} could be instantiated otherwise. The solution is to create as many different versions, called \text{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 is indicated with braces. Specifically, braces subscripted by a type \(\{\}\) are placed around encoded fragments that are used in their original context. In the dual case we write braces with an inverse \(\{\}^{-1}\). 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 \text{map} as presented in Figure 5 is of type:
\[ \Psi = (\text{int} \rightarrow \text{int}) \rightarrow \text{list} \text{int} \rightarrow \text{list} \text{int}. \]

A clone of \text{f} at a specialized type \(\text{type}\) is given the function symbol \(f\). It is created by taking a copy of the declaration of \text{f} in which arrow type variables are replaced with an application of the function \(\{\}\) to the arrow type variable, as shown in Figure 5. Note that at this point the recursive call to \text{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 \(\{\}\) must be transformed into an application of an \text{apply} function. In the body of \text{map}, the application \(\{\}\) is transformed into \(\text{apply}_{\text{int} \rightarrow \text{int}} \text{int} \rightarrow \text{int} \) as shown in Figure 7. The \text{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 \text{apply}_{\text{int} \rightarrow \text{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 \text{fully-applied}, in the context of a program \(P\). It also updates the set \(\Delta\) of function symbol declarations of \(P\) and the set \(\Theta\) of datatype declarations of \(P\), so that a transformation rule transforms a triple \((\text{term}, \Delta, \Theta)\) into a new triple. The three rules \(\text{FunSpec}, \text{EncodeClosed}, \text{ApplyVar}\) shown in Figures 9, 10 and 11, allow us to defunctionalize the fully-applied application \text{map id [1, 2]}. Definitions of the functions \text{order} and \text{functional} used in these rules can be found in Figure 14.

The rules reference type information calculated by type
Definitions:

\[ \{v\}_t = \text{the unencoded value of } t \]

\[ \{ [v]_t \}_t^{-1} = v \]

\[ \{ f (t_1 \ldots t_n) \}_t^{-1} = f \{ t_1 \}_t^{-1} \ldots \{ t_n \}_t^{-1} \]

\[ \{ \text{cons} (t_1 \ldots t_n) \}_t^{-1} = \text{cons} \{ t_1 \}_t^{-1} \ldots \{ t_n \}_t^{-1} \]

Typing rule:

\[ \Gamma \vdash t : \Pi \]

\[ \Gamma \vdash \{ t \}_t : \Pi \]

Figure 6: Definitions and typing rules for semantic functions

\[
\text{fun } \text{map}_\varphi F \ y \ = \ \text{case } y \ of \\
\quad \text{Nil } \Rightarrow \text{Nil} \\
\quad \text{Cons}(x, x s) \Rightarrow \text{Cons} (\text{apply}_{\text{int}\rightarrow \text{int}} F x, \text{map}_\varphi \{ F \}_{\text{int}\rightarrow \text{int}} x s)
\]

Figure 7: Clone of map where \( F \) is seen as encoded

\[
\text{datatype } \text{int}\rightarrow \text{int} = C_{\text{int}\rightarrow \text{int}} \\
\text{fun } \text{apply}_{\text{int}\rightarrow \text{int}} \ f \ x = \ \text{case } f \ of \\
\quad C_{\text{int}\rightarrow \text{int}} \Rightarrow (id \ x)
\]

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 \( \text{map } \text{id}[1,2] \) into \( \text{map}_\varphi \text{id}[1,2] \) and adds the clone \( \text{map}_\varphi \) from Figure 5 to the function declaration set \( \Delta \). The rule (EncodeClosed) encodes arrow type arguments into constructors of datatypes. For example, it transforms the expression \( \text{map}_\varphi \text{id}[1,2] \) into \( \text{map}_\varphi C_{\text{int}\rightarrow \text{int}}[1,2] \) and adds the encoding-derived declarations in Figure 8 to the declaration set \( \Delta \). Next, in the body of the declaration, the application \( \{ F \}_{\text{int}\rightarrow \text{int}} x \) is transformed into \( \text{apply}_k F x \) by the rule (ApplVar). In the clone \( \text{map}_\varphi \) there remains a fully-applied higher-order application of \( \text{map} \) which comes from the original recursive application of \( \text{map} \). 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 \( \text{map}_\varphi \) clone of \( \text{map} \), the specialized type of \( \text{map} \) in the recursive application: \( \text{map} \{ F \}_{\text{int}\rightarrow \text{int}} x s \) is recognized as the type \( \Pi \) because of the subscript \( \text{int} \rightarrow \text{int} \).

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 \( \text{map} \) into \( \text{map}_\varphi \) in the body of the declaration of the clone \( \text{map}_\varphi \), and change \( \{ F \}_{\text{int}\rightarrow \text{int}} \) into \( F \) via the application of \( \{ \}^{-1} \). No more rules apply; the result of the transformation is the first-order program composed of the term \( \text{map}_\varphi C_{\text{int}\rightarrow \text{int}}[1,2] \) and the declarations in Figure 8 and below:

\[
\text{fun } \text{map}_\varphi F \ y \ = \ \text{case } y \ of \\
\quad \text{Nil } \Rightarrow \text{Nil} \\
\quad \text{Cons}(x, x s) \Rightarrow \text{Cons} (\text{apply}_{\text{int}\rightarrow \text{int}} F x, \text{map}_\varphi \{ F \}_{\text{int}\rightarrow \text{int}} x s)
\]

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 \( \text{id} \) is of
IF
\[ \exists j, j \in 1 \ldots n, \quad \text{functional}(\Pi_j) \]
\& all functional variables in functional arguments are arguments of \{ \} \_ for some \( \Psi \)
\& \( n = \text{order}(\Pi) \& \text{order}(\Omega) = 0 \& f \) is a function symbol

AND
\[ \Gamma \vdash t_i : \Pi_i, \forall i, i \in 1 \ldots n, \]
\[ \Gamma \vdash f t_1 \ldots t_n : \Omega, \]
\[ \sigma \Pi = \Pi_1 \rightarrow \ldots \rightarrow \Pi_n \rightarrow \Omega \]

THEN
\[ f t_1 \ldots t_n, \Delta, \Theta \Rightarrow f_{\Pi} \{ t_1 \}^{-1} \ldots \{ t_n \}^{-1}, \Delta', \Theta \]

WHERE
\[ \Delta' = \begin{cases} \Delta \cup \{ f_{\Pi} x_1 \ldots x_n = M[x_i \leftarrow \{ x_n \}_{\Pi_n} \ldots x_i \leftarrow \{ x_i \}_{\Pi_i}] \} & \text{where } \Delta(f) = f x_1 \ldots x_n = M \text{ and } i, j \in 1 \ldots k, \text{ are the indices of the functional arguments} \\
& \end{cases} \]

Figure 9: (FunSpec) Functional type specialization transformation rule

IF
\[ \exists j, j \in 1 \ldots n \& \text{order}(\Pi_j) > 0 \& t_j \text{ is a closed term} \]
\& \( f \) is a function symbol or a function variable \& \( \text{order}(\Omega) = 0 \)

AND
\[ \Gamma \vdash t_i : \Pi_i, \forall i, i \in 1 \ldots n, \]
\[ \Gamma \vdash f t_1 \ldots t_n : \Omega, \]

THEN
\[ F t_1 \ldots t_j \ldots t_n, \Delta, \Theta \Rightarrow F t_1 \ldots C_{\Pi_j}^{t_j} \ldots t_n, \Delta', \Theta' \]

WHERE
\[ \Delta' = \begin{cases} \text{if } \text{Apply}_{\Pi_j} \text{ has not been declared in } \Delta \text{ then } \\
\Delta \cup \{ \text{Apply}_{\Pi_j} x y_1 \ldots y_{\text{order}(\Pi_j)} = \\
\text{case } x \text{ of } C_{\Pi_j}^{t_j} \Rightarrow t_j y_1 \ldots y_{\text{order}(\Pi_j)} \} & \text{else } \\
\text{add to it the case arm } C_{\Pi_j}^{t_j} \Rightarrow t_j y_1 \ldots y_{\text{order}(\Pi_j)} \} \\
\end{cases} \]
\[ \Theta' = \begin{cases} \text{if } I_{\Pi_j} \text{ has not been declared in } \Theta \text{ then } \\
\Theta \cup \{ \text{datatype } T_{\Pi_j} = C_{\Pi_j}^{t_j} \} & \text{else } \\
\text{add to it the constructor } C_{\Pi_j}^{t_j} \} \end{cases} \]

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

IF
\[ \text{order}(\Omega) = 0 \& F \text{ is a variable} \]

AND
\[ \Gamma \vdash F t_1 \ldots t_n : \Omega \]

THEN
\[ [F]_{\Pi} t_1 \ldots t_n, \Delta, \Theta \Rightarrow \text{Apply}_{\Pi} F t_1 \ldots t_n, \Delta, \Theta \]

Figure 11: (ApplyVar) Higher-order variable application
In this case, the specialization of the function will become monomorphic. The transformation must be able to encode both kinds of arrow type arguments of fully-applied functional applications.
Arrow type argument expressions may contain arrow type variables as well as first-order variables. For example an argument could be \( t = (\{Z\}; \{F_{int \to int}, \text{int} \to \text{int}\}) \), where \( \Omega \) is the type \( \text{int} \to \text{int} \to \text{int} \to \text{int} \). 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 \( f \) of type \( \text{int} \to \text{int} \) must be encoded in a parametric datatype \( \alpha \) specialized to \( \text{int} \to \text{int} \) by a constructor \( C_{\text{int} \to \text{int}} \) of type

\[
(T_G \times T_{\text{int} \to \text{int}} \times \alpha) \to (\text{int} \to \text{int}).
\]

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 \( \text{Encode} \) presented in Appendix A subsumes the rule \( \text{Encod} \) 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 \( \text{Cons} \) has the type \( \text{int} \to \text{int} \to \text{list} \to \text{list} \to \text{list} \) in the application \( \text{Cons}(\text{id}, \{x\}, \{x\}) \). The functional argument \( \text{id} \) of type \( \text{int} \to \text{int} \) has to be encoded into a constructor \( C_{\text{int} \to \text{int}} \) as for an application of a higher-order function. The rule \( \text{Encode} \) 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 \( \Pi \) of a term 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 \( \{\} \) to functional variables in the arm bodies. This is accomplished by the rule \( \text{(UpdateArms)} \) of Figure 15. For example:

\[
\text{case } \{x\}_{\text{int} \to \text{int}} \text{ of }
\begin{align*}
\text{Cons}(x_1, x_2) & \Rightarrow (x_1; y) \\
\end{align*}
\]

By matching the functional type \( \text{list} \to \text{int} \) against the parametric type \( \text{list} \alpha \), the rule \( \text{Encode} \) applies \( \{\} \) to functional variables in 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 \( \text{list} \to \text{int} \) 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 \( \text{Store} \) in the declaration: \( \text{datatype} \) \( \alpha \), \( \beta \) \( \text{store} = \text{Store} \alpha \to \beta \). Unlike a function symbol, a constructor cannot have clones in datatypes corresponding to different type instances of \( \alpha \) and \( \beta \). 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 \( \text{(Generalize Arrows)} \) of Figure 16 the
IF

\[
\text{functional}(T\Phi_1 \ldots \Phi_k) \quad \land \text{ all functional variables in } t \text{ are arguments of } \{\cdot\}_\Psi \text{ for some } \Psi
\]

AND

\[
\Gamma \vdash t : T\Phi_1 \ldots \Phi_k
\]

THEN

\[
\begin{aligned}
\text{case } t \text{ of } & C_1 x_1 \ldots x_{m_1} \Rightarrow t_1 \\
\vdots & \\
C_n x_n \ldots x_{m_n} \Rightarrow t_n,
\end{aligned}
\qquad
\begin{aligned}
\text{case } \{t\}^{-1} \text{ of } & C_1 x_1 \ldots x_{m_1} \Rightarrow t'_1 \\
\vdots & \\
C_n x_n \ldots x_{m_n} \Rightarrow t'_n
\end{aligned}
\]

WHERE

\[
T \text{ is declared } \text{datatype } \alpha_1 \ldots \alpha_k \ T = \\
\forall i, \ i \in 1 \ldots n, \ t'_i = t_i[x_{i_1} \leftarrow \{x_{i_2}\}_\Psi_{i_2}, \ldots, x_{i_p} \leftarrow \{x_{i_p}\}_\Psi_{i_p}]
\]

\[
\left\{ \begin{array}{l}
C_i \text{ appears as } C_i \Psi_1 \ldots \Psi_{m_i} \text{ in the instantiated subtype } [\alpha_1 \leftarrow \Phi_1, \ldots, \alpha_k \leftarrow \Phi_k] \\
j_i, \ i \in 1 \ldots p \text{ are the indices of functional pattern variables.}
\end{array} \right.
\]

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

datatype \textit{store} becomes \textit{datatype } \alpha \textit{ store } = \textit{Store } \alpha \textit{ so that encoding of differently typed functional arguments in different applications of the constructor \textit{Store} is possible.}

Rules (\textit{FunSpec}) and (\textit{ExpandSpec}) introduce applications of \{\cdot\}_\Psi to functional terms. These applications are ultimately removed when the functional term is applied (in (\textit{ApplVar})), when the term is examined (in (\textit{UpdateArms})), or by applying \{\cdot\}^{-1} (in (\textit{FunSpec}) and (\textit{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 \textit{Cons}(\{f\}_\Psi, \textit{Nil}). For this, we use the rule (\textit{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 \textit{T}_\Pi is created for functional arguments of type \Pi,

- 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 \textit{apply} function,

- to know when a clone must be an \eta-extended copy of the original polymorphic declaration, and

- to discriminate arrow type arguments of constructors in different arms of a \textit{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 \zeta in the context \Pi denoted by \Pi[\zeta]. In the previous section, a transformation rule has been written in the following abbreviated form:

\[
\text{if } C \ \zeta, \Delta, \Theta \Longrightarrow \zeta', \Delta', \Theta'
\]

Suppose that a function \zeta extracts the set \Delta of the function declarations from the program and that a function \theta extracts the set \Theta of type declarations from the program. As usual, the notation \(M[N'/N]\) denotes that the occurrence(s) of the subterm \N' in \M is replaced by the subterm \N'' in \M'. A transformation rule on a program \Pi[\zeta] could be expressed as:

\[
\text{if } C \ \Pi[\zeta] \Longrightarrow \Pi[\zeta'][\Delta'/\zeta'(\Pi[\zeta]), \Theta'/\theta(\Pi[\zeta])]
\]

Given a program, the defunctionalization algorithm applies the six rules (\textit{FunSpec}, \textit{ExpandSpec}, \textit{Encode}, \textit{ApplVar}, \textit{UpdateArms}, \textit{Generalize Arrows}) 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 \textit{apply} function. The label of an \textit{apply} function indicates the type of the functional encoding term \Psi. The type of the encoded term is the datatype \textit{T}_\Psi. Therefore:

\[
\vdash \textit{apply}_\Psi : T_\Psi \rightarrow \Psi
\]

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

\[
\vdash f_\Psi : \Psi
\[ \exists j, j \in 1 \ldots n, \text{functional}(\Pi_j) \]
\[ \land \text{all functional variables in functional arguments are arguments of \{\}_\Psi \text{ for some } \Psi } \]
\[ \land c \text{ is a constructor} \]
\[ \Gamma \vdash t_i : \Pi_i, \forall i, i \in 1 \ldots n, \]
\[ THEN \]
\[ c \langle t_1 \ldots t_m \Delta, \Theta \Rightarrow \Rightarrow \rangle c \langle t_1 \rangle \ldots \langle t_m \rangle, \Delta, \Theta \]

**Figure 16:** (GeneralizeArrows) Arrow constructor generalization

<table>
<thead>
<tr>
<th>IF</th>
<th>THEN</th>
</tr>
</thead>
</table>
| \[ \exists j, j \in 1 \ldots n, \land \text{order}(\Phi_j) > 0 \]
| \[ \land \text{all variables of } t_j \text{ are arguments of } \{\}_\Psi \text{ for some } \Psi \] |
| \[ C t_1 \ldots t_m \Delta, \Theta \cup \{\alpha_1 \ldots \alpha_m \} \]
| \[ \Rightarrow \Rightarrow \]
| \[ C t_1 \ldots t_m \Delta, \Theta \cup \{\beta_1 \ldots \beta_k \} \] |

**Figure 17:** (UpdateCon) Removes applications of \{\}_\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 \( \Phi \) and a datatype \( T_\Phi \) for function variables. However, when no rules apply, every term \{t\}_\Psi has been changed into a variable of type \( T_\Phi \) 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 \( e = F t_1 \ldots t_m \) is bound to \( t \) in the environment, then the evaluation of \[ e \]
  reduces to:

  \[ \text{app (eval [ } t ] \text{ evid [ } t_1 \ldots t_n \text{ ]} \]  

The rule transforms \( e \) into \( e' = \text{apply}_{T_\Phi} F t_1 \ldots t_m \).

But here \( F \) is an encoding of the term \( t' \) of the transformation of \( t \). Suppose \( C_{u_1} u_2 \ldots u_m \) is the encoding of \( t' \), and the constructor \( C_{u_1} \) belongs to the datatype \( T_\Phi \). The term \( u \) is the encoded term so \( t' \) is an instance of \( u \). Let \( x_1, \ldots, x_m \) be the variables of \( u \). Then the substitution \( \sigma \) is \( \{ x_1 \leftarrow u_1, \ldots, x_m \leftarrow u_m \} \), and the declaration of \( \text{apply}_{T_\Phi} \) contains the arm:

\[ C_{u_1} u_2 \ldots u_m \Rightarrow t' x_1 \ldots x_n \]

After pattern-matching with the substitution \( \sigma \), \( e' \) reduces to:

\[ \text{eval [ } \sigma(u) \text{ evid [ } t_1 \ldots t_n \text{ ]} \]  

which reduces to

\[ \text{app (eval [ } \sigma(u) \text{ ] evid [ } t_1 \ldots t_n \text{ ]} \]
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 \( \succeq \) which ensures the subterm property, is closed under context and extends the following partial ordering on terms:

\[
x \succ \frac{n}{l}
\]

\[
v_1 t_1 \ldots t_n \succ \text{apply}_{\text{\textit{ps}}} v_1 t_1 \ldots t_n
\]

\[
t \succ C[,]\text{, where } \Gamma \vdash t : \Pi
\]

\[
f t_1 \ldots t_n \succ \text{fix} t_1 \ldots t_n
\]

The associated \( \sim \) is equivalence by \( p \)-extension. This quasi-ordering is well-founded since \( \sim \) 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 \( \{ M_0, M_1, \ldots, M_n \} \Rightarrow \{ M'_0, M'_1, \ldots, M'_n \} \), then \( \{ M_0, M_1, \ldots, M_n \} \Rightarrow \{ M'_0, M'_1, \ldots, M'_n \} \) where \( \Rightarrow \) is the multiset ordering induced by \( \succ \).

- Rules (FunSpec) and (ExpandSpec) transform a subterm \( t \) of a term \( M_i \) into \( M_i[t'[\ell]] \), \( t \succ t' \) by (4), so \( M_i \succ M_i[t'[\ell]] \). If a clone \( M_j[v_1 \ldots v_p] \) where \( p > 0 \) is added to the multiset, \( M_i \succ M_j[v_1 \ldots v_p] \) by (1).

- Rules (UpdateArms) and (ApplyVar) transform a subterm \( t \) of an element \( M_i \) into \( M_i[t'[\ell]] \), \( t \succ t' \) respectively by (1) and (2), so \( M_i \succ M_i[t'[\ell]] \).

- Rule (Encode) transforms a subterm \( t \) of a term \( M_i \) into \( M_i[t'[\ell]] \), \( t \succ t' \) by (3), so \( M_i \succ M_i[t'[\ell]] \) of one \( M_i \) by (3). Moreover, the rule adds an arm body \( t \ y_1 \ldots y_k \) to the apply function, but \( M_i \succ t - t \ y_1 \ldots y_k \).

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 \( e \) 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.

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 \([\text{Wad88}]\) 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 \([\text{CD}]\). None of these methods consider the case of higher-order constructor applications.

Our transformation is based on Reynolds' method \([\text{Rey72}]\) 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 \( \lambda \)-algorithm \([\text{CD}]\), which is used to remove some functional results by \( \eta \)-expansion. Our transformation includes the functionality of the \( \lambda \)-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 \( \lambda \)-algorithm \([\text{CD}]\) 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 \( \text{map} \) in the introductory example in Section 2 is variable-only, as is the first argument of \( \text{map} \) in the example in Section B.1. Therefore in applications of \( \text{map} \) or \( \text{mp} \), the functions \( \text{map} \) and \( \text{mp} \) 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 \( \text{f} \), variable-only functional arguments lead to a clone of \( \text{f} \) 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 \( \text{f} \) 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 recogniz-
able 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 monomorphism 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 symbols 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 [B94, KBB94] 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 functional.

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


A (Encode) rule for encoding functional arguments

\[
\begin{align*}
\text{IF} \quad & \exists j, j \in 1 \ldots n, \ t_j \text{ is not a variable} \land \ \text{order}(\Pi_j) > 0 \\
& \land \text{all functional variables in } t_j \text{ are arguments of } \{\cdot\}_\Psi \text{ for some } \Psi \\
& \land F \text{ is a function symbol, a function variable, or a data constructor} \\
& \land \text{order}(\Omega) = 0 \\
\text{AND} \quad & \forall i, i \in 1 \ldots n, \Gamma \vdash t_i : \Pi_i, \\
& \Gamma \vdash F \text{ } t_1 \ldots t_m : \Omega, \\
& v_1 \ldots v_k \text{ are the variables in } t_j \\
\text{THEN} \quad & F \ t_1 \ldots t_j \ldots t_m, \Delta, \Theta \implies F \ t_1 \ldots (C^{\Pi_j}_{\Pi_j} v_1 \ldots v_k) \ldots t_m, \Delta', \Theta' \\
\text{WHERE} \quad & \Delta' = \left\{ \begin{array}{l}
\text{if Apply}_H \text{ has not been declared in } \Delta \text{ then} \\
\Delta \cup \{\text{Apply}_H \ x_1 \ldots x_{\text{order}(\Pi_j)} = \}
\text{case } y \text{ of } C^{\Pi_j}_{\Pi_j} v_1 \ldots v_k \Rightarrow t_j x_1 \ldots x_{\text{order}(\Pi_j)}
\text{else add to it the arm } C^{\Pi_j}_{\Pi_j} v_1 \ldots v_k \Rightarrow t_j x_1 \ldots x_{\text{order}(\Pi_j)}
\end{array} \right.
\]
\[
\Theta' = \left\{ \begin{array}{l}
\text{if } T_{\Pi_j} \text{ has not been declared in } \Theta \text{ then} \\
\Theta \cup \{\text{datatype } \alpha_1 \ldots \alpha_m \ T_{\Pi_j} = C^{\Pi_j}_{\Pi_j} \Phi_1 \times \ldots \times \Phi_k \}
\text{where } \forall i, i \in 1 \ldots k, \Phi_i = \left\{ \begin{array}{l}
\text{if } v_i \text{ is annotated by } \Psi \\
\text{then } \Psi \\
\text{else } \beta \text{ where } \beta \text{ is a fresh type variable}
\end{array} \right.
\text{else add to it the constructor } C^{\Pi_j}_{\Pi_j} \Phi_1 \times \ldots \times \Phi_k
\end{array} \right.
\]

B Examples

B.1 Second-order argument:

This example is inspired from [BH94b].

fun mp Z F x = case x of
  Nil => Nil
| Cons(x, xs) => 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 -> int)} = C^{inc}_{int -> int} | C^{(Z F)}_{int -> int} \times T_{(int -> int)}

datatype T_{(int -> int)} = C^{db}_{int -> int} \\

fun mp_q Z F x = case x of
  Nil => Nil
| Cons(x, xs) => Cons.apply_{(int -> int)} F x, mp_q Z (C^{(Z F)}_{int -> int}) xs)
fun apply_{(int -> int)} F x = case F of
  C^{inc}_{int -> int} => inc x
| C^{(Z F)}_{int -> int} (Z G) => (apply_{(int -> int)} Z G x)
fun apply_q Z F x = case Z of
  C^{db}_{int -> int} => db_q F x
fun db_q F x = apply_{(int -> int)} F (apply_{(int -> int)} F x)
fun inc x = x + 1

with the terms: mp_q^{db}_{int -> int} C^{inc}_{int -> int} [2, 3, 4]
\Phi = ((int \rightarrow int) \rightarrow int \rightarrow int) \rightarrow (int \rightarrow int) \rightarrow list int \rightarrow list int,
\Psi = (int \rightarrow int) \rightarrow int \rightarrow int

If combined with specialization with respect to the value of the variable-only first argument of mp,
this program becomes:

\[
\text{datatype } T_{\text{int} \rightarrow \text{int}} = C_{\text{int} \rightarrow \text{int}}^{(\mathbb{N})} \mid C_{\text{int} \rightarrow \text{int}}^{(\mathbb{N})} T_{\text{int} \rightarrow \text{int}}
\]

\[
\text{fun } \text{mp}_\Phi\ F\ x = \text{case } x\ \text{of}
\]

\[
\begin{align*}
\text{Nil} & \Rightarrow \text{Nil} \\
\text{Cons}(x, x's) & \Rightarrow \text{Cons}(\text{apply}_{\text{int} \rightarrow \text{int}} F\ x, \text{mp}_\Phi (C_{\text{int} \rightarrow \text{int}}^{(\mathbb{N})} (\text{db}, F))\ x's)
\end{align*}
\]

\[
\text{fun } \text{db}_\Phi\ f\ x = \text{apply}_{\text{int} \rightarrow \text{int}} F\ (\text{apply}_{\text{int} \rightarrow \text{int}} F\ x)
\]

\[
\text{fun } \text{inc}\ x = x + 1
\]

with the term: \(\text{mp}_\Phi\ C_{\text{int} \rightarrow \text{int}}^{[2, 3, 4]}\)

where \(\Phi = ((\text{int} \rightarrow \text{int}) \rightarrow \text{int} \rightarrow \text{int}) \rightarrow \text{list} \rightarrow \text{list} \rightarrow \text{list}\).

B.2 List of functions:

This example is borrowed from \([\text{CD}].\)

\[
\text{fun } \text{maph}\ F s\ y = \text{case } s\ \text{of}
\]

\[
\begin{align*}
\text{Nil} & \Rightarrow \text{Nil} \\
\text{Cons}(F, s's) & \Rightarrow \text{Cons}(F\ y, \text{maph}\ F s y)
\end{align*}
\]

\[
\text{fun } \text{add}\ y = \text{case } y\ \text{of}
\]

\[
\begin{align*}
\text{Nil} & \Rightarrow \text{Nil} \\
\text{Cons}(x, x's) & \Rightarrow \text{Cons}(k\ x, \text{add}\ x s)
\end{align*}
\]

with the term \(\text{maph}\ (\text{add}\ x s)\ y\) of type: \text{list} \rightarrow \text{int}.

becomes:

\[
\text{datatype } \alpha\ T_{\text{int} \rightarrow \text{int}} = C_{\text{int} \rightarrow \text{int}}^{(\mathbb{N})} \alpha
\]

\[
\text{fun } \text{maph}_\Phi\ F s\ y = \text{case } s\ \text{of}
\]

\[
\begin{align*}
\text{Nil} & \Rightarrow \text{Nil} \\
\text{Cons}(F, s's) & \Rightarrow \text{Cons}(\text{apply}_{\text{int} \rightarrow \text{int}} F\ y, \text{maph}_\Phi\ F s y)
\end{align*}
\]

\[
\text{fun } \text{add}\ y = \text{case } y\ \text{of}
\]

\[
\begin{align*}
\text{Nil} & \Rightarrow \text{Nil} \\
\text{Cons}(x, x's) & \Rightarrow \text{Cons}(C_{\text{int} \rightarrow \text{int}}^{(\mathbb{N})} x, \text{add}\ x s)
\end{align*}
\]

\[
\text{fun } \text{apply}_{\text{int} \rightarrow \text{int}} F\ y = \text{case } F\ \text{of}
\]

\[
C_{\text{int} \rightarrow \text{int}}^{(\mathbb{N})} x \Rightarrow k\ x\ y
\]

fun \(k\ x\ z = z + 5\ s\ x\) with the term: \(\text{maph}_\Phi\ (\text{add}\ x s)\ y\)

where \(\Phi = \text{list} (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \rightarrow \text{list} \rightarrow \text{list} \rightarrow \text{list}.\)

B.3 Pair of functions:

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

\[
\text{fun } \text{fmin}\ t = \text{case } t\ \text{of}
\]

\[
\begin{align*}
\text{Leaf} a & \Rightarrow (\text{Leaf}, a) \\
\text{Tree} (t1, t2) & \Rightarrow \\
\text{case } \text{(fmin}\ t1)\ of \\
(F1, m1) & \Rightarrow \text{case } \text{fmin}\ t2\ \text{of} \\
(F2, m2) & \Rightarrow (k\ F1\ F2, \text{min}(m1, m2))
\end{align*}
\]

fun \(k\ F\ G\ x = \text{Tree} (F\ x, G\ x)\)

becomes:

\[
\text{case } \text{(fmin}\ t)\ \text{of}\ (F, m) \Rightarrow (\text{apply}_{\text{tree} \rightarrow \text{int} \rightarrow \text{int}} F\ m)
\]

where \(\Phi = \text{tree} \rightarrow \text{int} \rightarrow \text{tree} \rightarrow \text{int} \times \text{int}\) with declarations:

\[
\text{datatype } T_{\text{tree} \rightarrow \text{int} \rightarrow \text{int}} = C_{\text{tree} \rightarrow \text{int} \rightarrow \text{int}}^{\text{Leaf}} \mid C_{\text{tree} \rightarrow \text{int} \rightarrow \text{int}}^{\text{Tree}} T_{\text{tree} \rightarrow \text{int} \rightarrow \text{int}} \times T_{\text{tree} \rightarrow \text{int} \rightarrow \text{int}}
\]

\[
\text{fun } \text{fmin}_t\ = \text{case } t\ \text{of}
\]

\[
\begin{align*}
\text{Leaf} a & \Rightarrow (C_{\text{tree} \rightarrow \text{int} \rightarrow \text{int}} a) \\
\text{Tree} (t1, t2) & \Rightarrow \\
\text{case } \text{(fmin}\ t1)\ \text{of} \\
(F1, m1) & \Rightarrow \text{case } \text{(fmin}\ t2)\ \text{of} \\
(F2, m2) & \Rightarrow (C_{\text{tree} \rightarrow \text{int} \rightarrow \text{int}} (F1, F2), \text{min}(m1, m2))
\end{align*}
\]

\[
\text{fun } \text{apply}_{\text{tree} \rightarrow \text{int} \rightarrow \text{int}} F\ m = \text{case } F\ \text{of}
\]
\[
C^{\text{Leaf}}_{\text{tree} \rightarrow \text{int} \rightarrow \text{int}} \Rightarrow \{\text{Leaf } m\}
\]

\[
C^{\text{Leaf}}_{\text{tree} \rightarrow \text{int} \rightarrow \text{int}} \Rightarrow \{F1, F2\} \Rightarrow \{k \ F1 \ F2 \ m\}
\]

\[
\text{fun} \ k \ F \ G \ m = \text{Tree} \ (\text{apply}_{\text{Tree} \rightarrow \text{int} \rightarrow \text{int}} \ F \ m, \text{apply}_{\text{Tree} \rightarrow \text{int} \rightarrow \text{int}} \ G \ m)
\]

B.4 Mutually recursive functions:

datatype \(\alpha\ \text{dec} = \text{Dec} \ \alpha \times \text{exp} \ \alpha\)
datatype \(\alpha\ \text{exp} = \text{Var} \ \alpha \mid \text{App} \ \alpha \times \text{exp} \ \alpha\)

\[
\text{fun} \ \text{fold-dec} \ D \ V \ A \ L \ x = \text{case } x \ \text{of}
\]

\[
\text{Dec}(v, x) \Rightarrow D \ v (\text{fold-exp} \ D \ V \ A \ L \ x)
\]

\[
\text{fun} \ \text{fold-exp} \ D \ V \ A \ L \ x = \text{case } x \ \text{of}
\]

\[
\text{Var} \ v \Rightarrow A (\text{fold-exp} \ D \ V \ A \ L \ y) (\text{fold-exp} \ D \ V \ A \ L \ z)
\]

\[
\text{Let} \ (x, z) \Rightarrow \text{Let} (\text{fold-dec} \ D \ V \ A \ L \ x) (\text{fold-exp} \ D \ V \ A \ L \ z)
\]

and the term \(\text{fold-exp} \ \text{proj}^2 \ \text{unit} \ \text{append} \ \text{append} \ (\text{Var} \ 'x')\),

becomes:

\[
\text{fold-exp}_{\Delta} C^{\text{proj}^2}_{\Delta \rightarrow \Sigma \rightarrow \Sigma} C^{\text{unit}}_{\Sigma \rightarrow \Sigma \rightarrow \Sigma} C^{\text{append}}_{\Sigma \rightarrow \Sigma \rightarrow \Sigma} C^{\text{append}}_{\Sigma \rightarrow \Sigma \rightarrow \Sigma} (\text{Var} \ 'x'),
\]

\[
\Pi = (\Delta \rightarrow \Sigma \rightarrow \Sigma) \rightarrow (\text{string} \rightarrow \Sigma) \rightarrow (\Sigma \rightarrow \Sigma \rightarrow \Sigma) \rightarrow (\Sigma \rightarrow \Sigma \rightarrow \Sigma) \rightarrow \text{exp} \rightarrow \Sigma,
\]

\[
\Psi = (\Delta \rightarrow \Sigma \rightarrow \Sigma) \rightarrow (\text{string} \rightarrow \Sigma) \rightarrow (\Sigma \rightarrow \Sigma \rightarrow \Sigma) \rightarrow (\Sigma \rightarrow \Sigma \rightarrow \Sigma) \rightarrow \Delta \rightarrow \Sigma,
\]

\[
\Delta = \text{dec string}, \ \Theta = \text{exp string}, \ \text{and } \Sigma = \text{list string},
\]

and the added declarations:

datatype \(T^{\text{dec exp}}_{\Delta \rightarrow \Sigma \rightarrow \Sigma} = C^{\text{proj}^2}_{\Delta \rightarrow \Sigma \rightarrow \Sigma}
\]
datatype \(T^{\text{string exp}}_{\Sigma \rightarrow \Sigma \rightarrow \Sigma} = C^{\text{unit}}_{\Sigma \rightarrow \Sigma \rightarrow \Sigma}
\]
datatype \(T^{\text{append exp}}_{\Sigma \rightarrow \Sigma \rightarrow \Sigma} = C^{\text{append}}_{\Sigma \rightarrow \Sigma \rightarrow \Sigma}
\]

\[
\text{fun} \ \text{fold-dec} \ D \ V \ A \ L \ x = \text{case } x \ \text{of}
\]

\[
\text{Dec}(v, x) \Rightarrow \text{apply}_{\Delta \rightarrow \Theta \rightarrow \Sigma} (\text{fold-exp}_{\Delta} D \ V \ A \ L \ x)
\]

\[
\text{fun} \ \text{fold-exp} \ D \ V \ A \ L \ x = \text{case } x \ \text{of}
\]

\[
\text{Var} \ v \Rightarrow \text{apply}_{\Theta \rightarrow \Sigma} V \ v
\]

\[
\text{Let} \ (x, z) \Rightarrow \text{apply}_{\Delta \rightarrow \Theta \rightarrow \Sigma} A (\text{fold-exp}_{\Delta} D \ V \ A \ L \ y) (\text{fold-exp}_{\Delta} D \ V \ A \ L \ z)
\]

\[
\text{fun} \ \text{apply}_{\Delta \rightarrow \Theta \rightarrow \Sigma} v \ x \ z = \text{case } v \ \text{of}
\]

\[
\text{proj}^2 \ \text{unit} x
\]

\[
\text{fun} \ \text{apply}_{\Theta \rightarrow \Sigma} v \ x \ y = \text{case } v \ \text{of}
\]

\[
\text{unit} x
\]

\[
\text{fun} \ \text{apply}_{\Sigma \rightarrow \Theta \rightarrow \Sigma} v \ x \ y = \text{case } v \ \text{of}
\]

\[
\text{append} y
\]