Specialisation of Higher-Order Functions for Debugging

Bernard Pope and Lee Naish

The University of Melbourne, Victoria 3010, Australia,
{bjpop,lee}@cs.mu.oz.au,
WWW home page: http://www.cs.mu.oz.au/{~bjpop,~lee}

Abstract. Because functions are abstract values without convenient print representations, implementing debuggers which support higher-order code is a challenge. We present an algorithm for statically specialising higher-order functions and encoding higher-order values to allow printing. We define our algorithm for a small functional language and discuss how it may be extended to support sophisticated features of modern functional programming languages. This research forms part of a project[1] to build a declarative debugger for Haskell, based primarily on source-to-source transformation.

1 Introduction

The predominance of higher-order programming in modern functional programming languages presents a challenge for designers of debuggers, particularly those based on source transformation or instrumentation. Higher-order languages permit functions to be returned as results, passed as arguments and contained in data structures. However, since functions are abstract values they are difficult to convert into a printable representation, which is essential for debugging. Source code identifiers for functions are generally discarded during compilation and new functions can be constructed dynamically using lambda abstractions and function composition. The difficulty of displaying functional values in the context of debugging is further discussed by Chitil *et al* [2].

Our approach employs a family of data structures to encode higher-order values and printable representations of those values. We use a source-to-source transformation (for portability) and avoid encoding first-order values (for efficiency). This has proven to be non-trivial within strongly typed polymorphic languages. Polymorphism allows the same source code to be used with multiple types, some of which are first-order and some of which are not. For example, consider the *map* function:

$$\begin{aligned} map = \lambda\, f\ l\ .\ & \textbf{case}\ l\ \textbf{of} \\ & [\,] \mapsto [\,] \\ & (:)\ x\ xs\ \mapsto (:)\ (f\ x)\ (map\ f\ xs) \end{aligned}$$

[1] Supported by the Australian Research Council.

Applications of *map* can produce a list of integers, as in the expression '*map id* $[1, 2, 3]$', or a list of functions, as in the expression '*map* $(+)$ $[1, 2, 3]$', and so on. The output from a declarative debugger arising from the inspection of these applications respectively might have the form:

```
map id [1,2,3] = [1,2,3], is this correct?
map (+) [1,2,3] = [(+) 1, (+) 2, (+) 3], is this correct?
```

Thus, to build a useful debugging system we must be able to generate meaningful printable representations of functions.

Using type information that is propagated through the call graph of the program, we specialise calls to and definitions of higher-order functions. To maintain the type correctness of the resulting specialised code, we may be required to make clones of some higher-order functions.

The context of this work is a project to build a portable declarative debugger for Haskell, which we describe in the next section. We follow this with a description of a simple functional language and define our transformation over this language. After discussing some extensions and related work we conclude.

2 Declarative debugging

The idea of using the declarative semantics to guide a semi-automated search for the source of bugs in programs is due to Shapiro [11]. The original work focussed on locating logical errors in Prolog programs, however the principles behind the method can be extended to other programming languages and paradigms.[2] Much of the original work in the field was performed under the title of *algorithmic debugging*, however, we adopt the term *declarative debugging* to emphasize the importance of declarative semantics in the technique.

Several declarative debugging techniques for functional languages have been based on the creation and traversal of a tree which describes (at some level of abstraction) the evaluation of a given program. We define an *evaluation dependence tree* (EDT) to represent an instance of the evaluation of a program.[3] The key feature of the EDT is that it reflects the syntactic dependencies of the program definition, rather than the execution dependencies due to evaluation. Each node in an EDT represents a function application that occurred during the execution of a program. At each node we record representations of three things: the result of the application; the function that was applied; and the arguments that the function was applied to. Function applications from the right hand side of the function definition form the children of the EDT node, and are themselves represented as EDTs.

There have been two approaches taken to constructing the EDT in the literature. The first approach, as exemplified in the work of Nilsson *et al* (for example

[2] Westman and Fritzson [12] report that algorithmic debugging has been applied to imperative languages and parallel logic languages.

[3] The term *evaluation dependence tree* is borrowed from Nilsson and Sparud [9], however, the structure of our tree is different from theirs.

[8]), requires a modified language implementation that generates the EDT as a side-effect of executing the program. As the proponents of this technique note, a significant effort is required to implement the necessary language implementation modifications. Such a technique allows for several optimisations to be made in the generation of the EDT, however it is not particularly portable between different language implementations. The second approach uses a source-to-source transformation on the program text. The definition of each function in a program is transformed to return a pair containing the original result of the function and an EDT node representing an application of that function during execution. The transformed program is then executed, resulting in a pair containing the value of the original program and a tree representing the EDT for that execution. This approach has been independently specified by Nilsson and Sparud [9] and Naish and Barbour [7], however there are significant differences in the EDTs that they define. We adopt the proposal of Naish and Barbour. A detailed exposition of the transformation algorithm and EDT is given in Pope [10].

An EDT node represents a call to a function and the result returned in some sub-computation of the program execution. The debugger prints EDT nodes and the user identifies erroneous sub-computations. By traversing part of the tree, noting which EDT nodes are erroneous, a bug can be isolated in a relatively small section of code. The EDT for a higher-order program has higher order values in its nodes and this paper addresses the problem of printing them.

3 The Object Language

We define a small functional language (called Mini-Fp) as the object of our transformation algorithm. It is designed in the spirit of larger languages such as Haskell and ML. The abstract syntax for the language is given in figure 1. The concrete syntax for the language is based on that of Haskell. We assume that various basic data-types, such as integers, characters, lists and tuples are built in to the language. We also assume that commonly available functions over these types (such as addition for integers) are available as primitive functions. We do not define an evaluation semantics for Mini-Fp, however the specialisation algorithm is suitable for languages with either strict or non-strict semantics. We assume a static type system for the language and a type inference algorithm based on the well known Hindley-Milner system [6]. A program consists of one or more variable declarations and zero or more data declarations. Type annotations are permitted, but not required, for top and let bound variables.[4] One of the top-level value declarations must be called *main*, it must have no arguments and be monomorphic. The evaluation of a program proceeds from an initial call to this function.

[4] Type annotations would be required in some circumstances if the language supported polymorphic recursion. This is covered in more detail in section 5.

$x \in$ Variables		
$p \in$ Primitives		
$c \in$ Data Constructors		
$f \in$ Type Constructors		
$v \in$ Type Variables		
$s \in$ Type Schemes	$::= \forall\ \overline{v}\ .\ t$	
$d \in$ Declarations	$::= x = e \quad \mid \quad f\ v_1 \dots v_k = \{c_i\ t_{i1} \dots t_{in_i}\}_{i=1}^{m}$	
$t \in$ Types	$::= v \quad \mid \quad t_1 \rightarrow t_2 \quad \mid \quad f\ t_1 \dots t_k$	
$e \in$ Expressions	$::= \lambda x.e \quad \mid \quad e_1\ e_2 \quad \mid \quad x \quad \mid \quad c \quad \mid \quad \text{let } d_1 \dots d_n \text{ in } e$	
	$\mid \quad \text{case } e \text{ of } a_1 \dots a_n \quad \mid \quad p\ e_1 \dots e_n$	
$a \in$ Alternatives	$::= c\ e_1 \dots e_n \mapsto e \quad \mid \quad x \mapsto e$	

Fig. 1. Abstract syntax of MINI-FP

4 The Specialisation Algorithm

The specialisation algorithm has two stages. The first stage is called *cloning*, and the second stage is called *encoding*. The role of cloning is to expand the static call graph of the program such that calls to functions with higher-order arguments are distinguished based on the shape of the type of each higher-order argument. In the presence of functions which are polymorphic in one or more higher-order arguments, cloning may require multiple copies of the definition of the function to be made. The outcome is a reduction in the polymorphism of the program and expansion of the size of code in the program. The role of encoding is to provide printable representations of functional values. After encoding, all higher-order arguments will be lifted into a special representation that contains both the original value and a printable representation of the value. The encoding stage also provides a means for converting between encoded and un-encoded values. The cloning stage is necessary to ensure that the encoding stage preserves the type correctness properties of the program. The algorithm is an example of a type-directed program transformation and is similar in spirit (but not motivation) to the defunctionalisation algorithm of Bell *et al* [1]. The intention of the algorithm is to map a type correct MINI-FP program to a type correct MINI-FP program, such that when the the two programs are evaluated they return the same value.

4.1 A specialisation example

Consider the following MINI-FP program:

$$
\begin{aligned}
&\mathit{main} &&= \mathit{compose}\ \mathit{plus}\ (\mathit{plus}\ 1)\ 3\ (\mathit{twice}\ \mathit{id}\ 5)\\
&\mathit{compose} &&= \lambda\, f\ g\ x\ .\ f\ (g\ x)\\
&\mathit{twice} &&= \lambda\, x\ .\ \mathit{compose}\ x\ x\\
&\mathit{plus} &&= \lambda\, x\ y\ .\ (+)\ x\ y\\
&\mathit{id} &&= \lambda\, x\ .\ x
\end{aligned}
$$

Each function is called with the following types:

$$
\begin{array}{ll}
main & \Rightarrow Int \\
compose & \Rightarrow (Int \to Int \to Int) \to (Int \to Int) \to Int \to Int \to Int \\
 & \Rightarrow (Int \to Int) \to (Int \to Int) \to Int \to Int \\
twice & \Rightarrow (Int \to Int) \to Int \to Int \\
plus & \Rightarrow Int \to Int \to Int \\
id & \Rightarrow Int \to Int \\
(+) & \Rightarrow Int \to Int \to Int
\end{array}
$$

Notice that *compose*, a higher-order polymorphic function, is called in two different ways. Cloning will distinguish the two usages of *compose* by making a clone (or copy) of the code for *compose* and uniquely naming each clone apart. The algorithm must also rename the two calls to *compose*. At this stage the program is identical to the original except for the increase in size and modification of the static call graph.

$$
\begin{array}{ll}
main & = compose_1\ plus\ (plus\ 1)\ 3\ (twice_1\ id\ 5) \\
compose_1 & = \lambda\, f\ g\ x\ .\ f\ (g\ x) \\
compose_2 & = \lambda\, f\ g\ x\ .\ f\ (g\ x) \\
twice_1 & = \lambda\, x\ .\ compose_2\ x\ x
\end{array}
$$

The encoding stage of the specialisation algorithm will convert each higher-order argument into a first-order value by encapsulating the argument inside a data structure. The data-structure will contain the higher-order function and a representation of that function (for simplicity we use a *String* as the representation). In the code below, the type $(F_1\ a\ b)$ will be used to encapsulate all higher-order values of type $(a \to b)$ such that b is first-order. More generally, $(F_n\ v_1\ \ldots\ v_{n+1})$ encapsulates all higher-order values of type $(t_1 \to \ldots \to t_{n+1})$ such that t_{n+1} is first-order:[5]

$$
\begin{array}{ll}
\mathbf{data}\ F_1\ a\ b & = F_1\ (a \to b)\ String \\
\mathbf{data}\ F_2\ a\ b\ c & = F_2\ (a \to b \to c)\ String \\
apply_1 & = \lambda\, x\ .\ \mathbf{case}\ x\ \mathbf{of}\ (F_1\ f\ s)\ \mapsto\ f \\
apply_2 & = \lambda\, x\ .\ \mathbf{case}\ x\ \mathbf{of}\ (F_2\ f\ s)\ \mapsto\ f \\
main & = compose_1\ (F_2\ plus\ \text{“plus”})\ (F_1\ (plus\ 1)\ \text{“}plus\ 1\text{”})\ 3 \\
 & \quad (twice_1\ (F_1\ id\ \text{“id”})\ 5) \\
compose_1 & :: (F_2\ b\ c\ d) \to (F_1\ a\ b) \to a \to c \to d \\
compose_1 & = \lambda\, f\ g\ x\ .\ apply_2\ f\ (apply_1\ g\ x) \\
compose_2 & :: (F_1\ b\ c) \to (F_1\ a\ b) \to a \to c \\
compose_2 & = \lambda\, f\ g\ x\ .\ apply_1\ f\ (apply_1\ g\ x) \\
twice_1 & :: (F_1\ a\ a) \to a \to a \\
twice_1 & = \lambda\, x\ .\ compose_2\ x\ x
\end{array}
$$

When a variable is bound to an encoded value, and that encoded value is applied in an expression, we must decode the value (to the original function) before it

[5] Type annotations are provided for the clones of *compose* and *twice* for illustrative purposes only.

is applied. For each introduced type $(F_n\ v_1\ \ldots\ v_{n+1})$, we define an associated $apply_n$ function of type $((F_n\ v_1\ \ldots\ v_{n+1}) \rightarrow v_1 \rightarrow \ldots \rightarrow v_{n+1})$ which selects the original function from the encoded representation.

At this point it may be unclear why we distinguish between $(a \rightarrow b)$ and $(a \rightarrow b \rightarrow c)$ when the second type is just an instance of the first (and hence why we cloned the *compose* definition). Indeed we could have performed the above encoding with $(F_1\ a\ b)$ alone. The distinction is dictated by our desire to avoid the encoding of all values in the program: we only wish to encode higher-order values. This partial encoding presents a problem: if an encoded value is applied in an expression, is the result encoded or not? The answer to this question is determined by the number of arguments such an encoded value will require before the result is first-order. In our example, *plus* requires two arguments, (*plus* 1) requires one, and 3 requires none.

Thus we must distinguish between $(a \rightarrow b)$ and $(a \rightarrow b \rightarrow c)$ (and so on) because we need to be sure that the rightmost type variable does not itself expand out to a functional type. It is only by employing a family of encodings F_n, with the requirement that the rightmost argument be first-order, that we can guarantee this behaviour, and ultimately maintain type correctness in the encoded program. The use of multiple encodings means multiple $apply_n$ functions must be used — this is the reason for cloning definitions. In our example, *compose* was cloned because the final versions which support encoding require different $apply_n$ functions (the cloning phase deduces this from the types *compose* is called with).

One final detail that needs clarification is the application of data-constructors to higher-order values. Here we follow the algorithm of Bell *et al* [1]. As with function application, we must encode the higher-order argument(s) of the constructor, however there is no requirement for cloning constructor definitions, as there is with value declarations. This is because the types of all data-constructors have the form $(t_1 \rightarrow \ldots \rightarrow t_n)$ such that t_n is not a type variable. Partial application of data-constructors are encoded in the same way as partial application of functions. One complication arises when higher-order application of constructors is made explicit in the definition of the type (as is the case with the first argument of the constructors F_1 and F_2 above). In such circumstances we replace each functional argument of a constructor with a fresh type variable and generalise the definition of the type with each new variable that is introduced.

4.2 The Cloning Algorithm

Conceptually the cloning algorithm is quite simple, however the implementation details are complicated by the existence of local (let bound) values and the associated scoping of variables in nested expressions. We define the cloning algorithm in an abstract functional style, driven by the recursive function *clone*.

The *clone* function performs a recursive traversal through the call graph of the program and ensures that declarations are cloned at the correct level of nesting in the program, and that the computation terminates in the presence of recursive declarations. It is assumed that the type schemes for all top and let

```
Function clone (Q1, S1, P1)
begin
    if Q1 == ∅ then
        return P1
    else
        (x = e, t) ∈ Q1
        Q2 ⟵ Q1 − {(x = e, t)}
        S2 ⟵ S1 ∪ {(x, t̂)}
        C ⟵ calls (e, t)
        (Q3, P2) ⟵ newCalls (C, S2, P1)
        return clone (Q2 ∪ Q3, S2, P2)
end
```

bound variables are available from type inference. There are three parameters to the function *clone*. Their roles are as follows.

Q is a set of pairs (d, t) such that d is the definition of a let bound or top bound variable and t is a type at which d is called. Q contains a set of unprocessed calls. Initially Q is the singleton set containing the definition and type of *main*. The algorithm terminates when Q is empty. S is a set of pairs $(x, \hat{t}\,)$ such that x is the name of a let bound or top bound variable and $\hat{t}$ is an abstraction of the type at which x is called. S records all the calls that have been processed previously by *clone*. The information in S is used to ensure that unnecessary work is not performed, in particular that the right amount of cloning is achieved and that the algorithm terminates with recursive definitions. S is initially the empty set. P is the syntax tree of the entire program. When a clone of a declaration is made the new (uniquely named) declaration is added to the syntax tree, and the call graph is modified to suit the new name. Initially P is the syntax tree of the original program, upon termination of the algorithm, P is the syntax tree of the cloned program.

Given a call type t_c, there are many ways form an abstraction. A simple method would be to map all nullary type constructors in t_c to a new unique nullary type constructor. Under such a scheme we would consider the types *Int*, *Bool*, *Char* equivalent, but we would consider (*Int*→*Int*) different. The problem with this scheme is that it it too conservative and may lead to unnecessary cloning.

The goal of the abstraction technique is to reduce the amount of cloning required, whilst allowing the encoding algorithm to maintain type correctness. Our choice of abstraction method is directed by one important observation: cloning is only required when the type scheme of a variable contains arrow type arguments that are polymorphic. Since we are interested in the polymorphic properties of a variable we must make use of the information contained in its type scheme as deduced by type inference. Therefore we can narrow our abstraction technique even further. Consider the type scheme of a variable as a tree with arrows and non-nullary type-constructors at the nodes, and type variables and nullary type-constructors at the leaves. We call the set of variables that occur as right children of arrow nodes the *distinguishing type variables* (DTVs). The type at which these variables are called determines how higher-order arguments are encoded, and ultimately how much cloning is needed. The function *dtv* below computes the set of DTVs for a given type scheme.

$$
\begin{array}{ll}
dtv(\forall\, \overline{v}\, .\, t) & = dtv(t) \\
dtv(f\ t_1\ \ldots\ t_k) & = \bigcup_{i=1}^{k}\ dtv(t_i) \\
dtv(v) & = \emptyset \\
dtv(t_1 \rightarrow v) & = dtv(t_1)\ \cup\ \{v\} \\
dtv(t_1 \rightarrow t_2) & = dtv(t_1)\ \cup\ dtv(t_2),\ \textit{such that } t_2 \textit{ is not a type variable}
\end{array}
$$

Some well known type schemes and their corresponding set of DTVs are illustrated in the table below:

type scheme	DTVs
$bottom :: \forall a\ .\ a$	$\emptyset$
$id :: \forall a\ .\ a \rightarrow a$	$\{a\}$
$map :: \forall a, b\ .\ (a \rightarrow b) \rightarrow [a] \rightarrow [b]$	$\{b\}$
$compose :: \forall a, b, c\ .\ (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$	$\{c, b\}$

For a variable x with type scheme s and a call type t_c we would like to know how the DTVs of s are instantiated when s and t_c are unified. In particular we would like to know the number of arrows in the rightmost branch of each instantiating type when represented as a tree. We define an abstract type to be a (possibly empty) set of pairs (v, n) such that v is a type variable and n is a natural number. Given the function *mgu*, which computes the most general unifiers of a type scheme and a type (as a set of substitutions), we calculate the abstracted type for a type scheme s and call type t_c using the function *abstype* below:

$$
\begin{array}{l}
abstype(s,\ t_c) \\
\quad = \{\ (v,\ order(t_v))\ \textit{such that}\ v\ \in\ dtv(s),\ \textit{and}\ (v,\ t_v)\ \in\ mgu(s,\ t_c)\ \}
\end{array}
$$

$$
\begin{array}{ll}
order(v) & = 0 \\
order(f\ t_1\ \ldots\ t_k) & = 0 \\
order(t_1 \rightarrow t_2) & = 1\ +\ order(t_2)
\end{array}
$$

If we consider the type scheme of map, the table below shows the abstracted versions of various call types:

call type	abstracted type
$(Int \rightarrow Int) \rightarrow [Int] \rightarrow [Int]$	$\{(b, 0)\}$
$(Bool \rightarrow [Char]) \rightarrow [Bool] \rightarrow [[Char]]$	$\{(b, 0)\}$
$(Int \rightarrow Int \rightarrow Int) \rightarrow [Int] \rightarrow [Int \rightarrow Int]$	$\{(b, 1)\}$
$(Bool \rightarrow Bool \rightarrow Bool \rightarrow Bool) \rightarrow [Bool] \rightarrow [Bool \rightarrow Bool \rightarrow Bool]$	$\{(b, 2)\}$

For variables whose set of DTVs is empty (such as *bottom*), the cloning algorithm will process calls to them at most once, since all calls to such variables are always considered equivalent. Furthermore, the definitions of such variables will never be cloned.

The definition of *clone* relies on two external functions, *calls* and *newCalls*. The function *calls* takes a pair (e, t_c) as it argument, such that e is the right hand side expression of a declaration for some variable x, and t_c is a type at which x is called. Using the type-schemes deduced from type inference, the type t_e of e is calculated and unified with t_c. The unification instantiates calls to polymorphic functions made in e. The types of those calls, along with the names of the functions that were called are collected together and returned in the set C. The function *newCalls* has three arguments: the set C of (variable, type) pairs (the output from *calls*) describing newly deduced calls; a set S of (variable, abstracted type) pairs describing the calls already processed; and P the current state of the syntax tree. For each call (x, t) in C, *newCalls* inspects S to see if x has been previously processed at an abstracted call $\hat{t}$. The declarations and call types of all unprocessed calls are returned in a set. For calls whose abstracted types are non-empty sets, clones are made and added to the syntax tree, and call sites in the existing syntax tree are updated to coincide with the new name of the appropriate cloned declaration. Care has to be taken to ensure that recursive calls inside cloned declarations are renamed appropriately.

The termination, soundness and completeness properties of the cloning algorithm are discussed in section 5 of this paper.

4.3 The Encoding Algorithm

The encoding algorithm transforms the program so that all higher-order values which are used as arguments to functions or data constructors are encoded (wrapped in F_n constructors). This requires a change to the types throughout (the higher-order part of) the program. Types of top level and let bound variables are changed so arrows in the rightmost branch of the type (those separating the types of arguments and the result) are retained but all nested function types are replaced by the associated encoded function type: $t_1 \to \ldots \to t_{n+1}$ (where t_{n+1} is first-order) is replaced by $F_n\ t_1\ \ldots\ t_{n+1}$. All function types of lambda and pattern bound variables are converted in the same way. Thus the new type scheme for $compose_2$, generated for types of the form $(b \to c \to d) \to (a \to b) \to a \to c \to d$, is $(F_2\ b\ c\ d) \to (F_1\ a\ b) \to a \to c \to d$ and within its definition the type of f is converted from $b \to c \to d$ to $F_2\ b\ c\ d$ and g converted from $a \to b$ to $F_1\ a\ b$ (see section 4.1). Similarly, a map clone map_1 for type $(a \to b \to c) \to [a] \to [b \to c]$ has its type scheme converted to $(F_2\ a\ b\ c) \to [a] \to [F_1\ b\ c]$.

There are two kinds of changes to the program needed. We need to encode some arguments and we need to decode some variables which are being applied. Decoding is required for applications $v\ e$, where v is a lambda or pattern bound variable originally with a function type (after type conversion it has type $F_n\ t_1\ \ldots\ t_{n+1}$). Decoding v is done using $apply_n$. For example, in the definition of $compose_2$ where f is applied, $apply_2$ is used to decode it (and $apply_1$ is used to decode g). If v is a top-level expression in a definition it should also be treated as an application (or definitions could be eta-expanded to make the application explicit).

Encoding is required for applications $e_1\ e_2$ where the original type of e_2 is a function type (the new type is $F_n\ t_1\ \ldots\ t_{n+1}$). Encoding e_2 is done by applying the data constructor F_n and adding the string expression. For let and top level bound variables strings are generated in the obvious way. For applications, lambda, case and let expressions strings are generated by concatenating the different components separated by the appropriate spaces, keywords, *et cetera* (for example, see *main* in Section 4.1). For lambda and pattern bound variables we need a generic way to convert a value (with any higher-order components encoded) into a string. This is not possible to express in MINI-FP but possible in most modern functional languages. Below we assume a function *toString* exists (we use the type class mechanism of Haskell in our implementation).

Expressions of the form $e_1\ (v\ e)$, where v is a lambda or pattern bound variable originally with a function type, and $(v\ e)$ also having a functional type, require both decoding and (re)encoding. The function and string must be extracted from the coded function v and applied to and concatenated with e and its string representation, respectively. We use a family of functions, toF_i of type $(F_{i+1}\ t_1\ \ldots\ t_{i+2}) \to t_1 \to (F_i\ t_2\ \ldots\ t_{i+2})$. If the encoded type of v is $F_{i+1}\ t_1\ \ldots\ t_{i+2}$ then the expression above is converted to $e_1\ (toF_i\ v\ e)$. The toF_i functions are defined as follows:

$$\begin{array}{lcl} toF_1 &=& \lambda\, x\ y\ .\ \mathbf{case}\ x\ \mathbf{of}\ (F_2\ f\ s) \mapsto F_1\ (f\ \ y)\ (append\ \ s\ \ (toString\ y)) \\ toF_2 &=& \lambda\, x\ y\ .\ \mathbf{case}\ x\ \mathbf{of}\ (F_3\ f\ s) \mapsto F_2\ (f\ \ y)\ (append\ \ s\ \ (toString\ y)) \\ && \vdots \end{array}$$

For example, the definition of map_1 discussed above needs decoding and re-encoding where f is applied:

$$\begin{array}{l} map_1 :: (F_2\ \ a\ \ b\ \ c) {\to} [a] {\to} [F_1\ \ b\ \ c] \\ map_1 = \lambda\ f\ \ l\ .\ \mathbf{case}\ l\ \mathbf{of} \\ \qquad [\,]\ \mapsto\ [\,] \\ \qquad (:)\ x\ \ xs\ \ \mapsto\ \ (:)\ (toF_1\ \ f\ \ x)\ (map_1\ \ f\ \ xs) \end{array}$$

5 Discussion and Extensions

Polymorphic recursion occurs when the type of at least one recursive call to a function is not equal to, but is an instance of, the type at which the function was defined. Such recursion is not supported by the Hindley-Milner type inference algorithm. There are various extensions to the Hindley-Milner algorithm that permit polymorphic recursion. For example, the type-checking algorithm of the Haskell language (as described in [4]) allows polymorphic recursion only when sufficient type annotations are provided for all functions which are used in a polymorphically recursive context. If we allow polymorphic recursion using the

Haskell scheme, and encounter the following function, the cloning stage of the specialisation algorithm will fail to terminate:

$$\begin{aligned} f &:: (a \rightarrow b) \rightarrow c \\ f &= \lambda\ g\ .\ f\ (\lambda\ x\ .\ g) \end{aligned}$$

In the expression '$f\ id$' the abstracted call type of f is $\{(b,0),(c,0)\}$. In successive recursive calls to f the abstracted call types follow the pattern $\{(b,1),(c,0)\}$, $\{(b,2),(c,0)\}$ and so on. Each new call is considered different thus requiring an infinite family of cloned versions of f. However, not all forms of polymorphic recursion cause non-termination in the cloning algorithm. For example, the function h in the example below is polymorphically recursive:

$$\begin{aligned} h &:: [a] \rightarrow b \\ h &= \lambda\ x\ .\ h\ [x] \end{aligned}$$

However, its abstracted call type remains constant with each recursive call: $\{(b,0)\}$.

The cloning algorithm will terminate whenever the abstracted types of successive recursive calls to a function are equal (for first order arguments this criteria is trivial). The detection of non-terminating cases is an area of future work. If detection is possible, then calls to such functions will not be specialised. Ultimately this implies that less information about applications of such functions will be available to the declarative debugger. This impacts on the completeness of the debugging algorithm, but not the soundness, and is probably a small price to pay in practice to allow restricted forms of polymorphic recursion.

Most modern functional programming languages provide modules as a means of sub-dividing and encapsulating code that when combined together constitute a program. This complicates the specialisation algorithm when function calls are made across module boundaries. The current specialisation algorithm assumes that the entire program definition will be transformed. In the presence of multi-module programs this would require an entire pass over the program in the dependency order of the module graph, a process that is complicated further when there are mutual dependencies between modules. This issue is also of particular relevance to the debugging transformation, which we would ideally like to be able to apply to a subset of the entire program. Having to transform (and traverse) the entire program is sub-optimal. Furthermore, both the specialisation and the debugging transformations introduce overheads in the execution of the program. In the context of debugging the goal would be to restrict the scope of the transformations to *suspicious* code, leaving the remainder of the code in its original state. This is an important area of future research.

Type classes are a powerful mechanism for allowing overloading of identifiers, and are an integral part of the type system of languages such as Haskell and Mercury. Type classes complicate the cloning algorithm because one identifier may have declarations for multiple instances of a type class. The current cloning algorithm assumes that each identifier has exactly one definition. It may be possible to selectively clone instances of identifiers which are class members based on type of the call that was made. The incorporation of type classes will be an important aspect of future development for the specialisation algorithm.

For the specialisation algorithm to be sound, we require two things. First, that the resulting specialised program is well typed according to our chosen type inference scheme (for now we assume Hindley Milner, with possible extensions to allow polymorphic recursion). Secondly, given an evaluation semantics for our language, the resulting program must have the same result when evaluated as the original program. For the cloning stage, type correctness should be relatively straightforward to prove, and we may get some leverage from the work of Bell *et al.* For the encoding stage, type correctness may be more difficult to prove, however, it is envisaged that the proof will make use of an augmented type inference algorithm with explicit rules for encoding and decoding functional values. Proving evaluation equivalence is yet to be explored, however, we hope to be able to make the proof independent of the strictness of the evaluation strategy. For completeness we require a semantics for the language which provides the EDT. Under such a semantics, the algorithm is considered complete if when applied to a type correct program all the nodes in the corresponding EDT are printable in an unambiguous manner with respect to the source level definition of the program.

There may be additional uses of the algorithm we have presented here other than debugging. For example the design of evaluation schemes for higher-order languages is complicated in the presence of polymorphism. Peyton Jones highlights this in a discussion of the design space for the Spineless Tagless G-machine [5]:

> It is worth noting that, in a polymorphic language, it is not always possible to distinguish between thunks whose value will turn out to be a function from thunks whose value is a data value. For example, consider the function:
>
> ```
> compose f g x = f (g x)
> ```
>
> Is `(g x)` a function or not? It depends, of course, on the type of `g` and, since `compose` is polymorphic, this is not statically determined.

The cloning stage of the specialisation algorithm presents a possible solution to the problem of statically determining whether expressions are functional or not. This, in turn, may allow us to bridge the gap between curried and un-curried language implementations.

6 Related work

An algorithm for type-driven defunctionalisation is given by Bell *et al* [1]. This takes a higher-order polymorphic program, specialises it to remove some polymorphism and encodes all higher-order values. More polymorphism is removed than in our work and higher-order values are encoded using (possibly recursive) first-order data structures. This leads to greater code expansion, more new types and associated decoding functions and likely greater runtime overheads. The architecture of our debugger introduces further code expansion and overheads so it is important for our treatment of higher-order code to be frugal. Defunctionalisation is motivated by the translation of higher-order programs into languages that only support first-order functions, whereas our specialisation algorithm is motivated by the need for printable representations of higher-order values for debugging. The defunctionalisation algorithm completely removes all higher-order values from the program, whereas our specialisation algorithm simply wraps them up inside a data structure. This means that we lose much less polymorphism than the defunctionalisation algorithm. The relaxed constraints on our specialisation algorithm mean that it can support some forms of polymorphic recursion, however, the defunctionalisation algorithm does not support any form of polymorphic recursion.

Hughes [3] presents a method for specialising a program based on partial evaluation of an interpreter for the source language. This provides a very general framework for performing many sorts of program specialisations including *monomorphisation* and *firstification*. The resulting firstification (or defunctionalisation) of this method achieves somewhat similar results to Bell *et al.* It would be interesting to investigate whether this more general framework could be tuned to generate the results that we require for debugging.

7 Conclusion

This paper has presented a novel application of type-directed program specialisation for the purposes of generating printable representations of higher-order values for debugging. In particular, we have shown that the algorithm can be expressed in two parts, *cloning* and *encoding*. Definitions for each of these parts have been provided. A key feature of the algorithm is that a large degree of polymorphism is retained in the specialised program, and first-order values are not encoded. This has two main benefits. First, the overheads introduced by the specialisation are reduced, particularly the expansion in code size and costs associated with encoding and decoding values. Second, the algorithm is applicable to some forms of polymorphic recursion. The output of the specialisation algorithm will be used by future debugging transformations, which will hopefully lead to an effective and portable means for diagnosing errors in strongly-typed higher-order polymorphic functional programming languages.

References

1. J. Bell, F. Bellegarde, and J. Hook. Type-driven defunctionalization. In M. Tofte, editor, *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, pages 25–37, 1997.

2. O. Chitil, C. Runciman, and M. Wallace. Tracing and Debugging of Lazy Functional Programs — A Comparative Evaluation of Three Systems. In M. Mohnen and P. Koopman, editors, *Proceedings of the 12th International Workshop on Implementation of Functional Languages*, number AIB-00-7 in Aachener Informatik Berichte, pages 47–62. RWTH Aachen, 2000.
3. J. Hughes. Type specialisation for the lambda-calculus; or, a new paradigm for partial evaluation based on type inference. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Dagstuhl Workshop on Partial Evaluation*, LNCS. Springer Verlag, February 1996.
4. M. Jones. Typing Haskell in Haskell. Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht, Utrecht, Netherlands, 1999.
5. S. Peyton Jones. Implementing lazy functional languages on stock hardware: The Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.
6. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.
7. L. Naish and T. Barbour. Towards a portable lazy functional declarative debugger. *Australian Computer Science Communications*, 18(1):401–408, 1996.
8. H. Nilsson and P. Fritzson. Lazy algorithmic debugging: Ideas for practical implementation. In P. Fritzson, editor, *Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*, pages 117–134, Linköping, Sweden, 1993.
9. H. Nilsson and J. Sparud. The evaluation dependence tree: an execution record for lazy functional debugging. Technical report, Department of Computer and Information Science, Linköping University, 1996.
10. B. Pope. Buddha: A declarative debugger for Haskell. Technical Report 98/12, The Department of Computer Science and Software Engineering, The University of Melbourne, 1998.
11. E. Shapiro. *Algorithmic Program Debugging*. The MIT Press, 1982.
12. R. Westman and P. Fritzson. Graphical user interfaces for algorithmic debugging. In P. Fritzson, editor, *Lecture Notes in Computer Science*, volume 749, pages 273–286. Springer, May 1993.