A Program Transformation for Declarative 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. We present a declarative debugger for lazy functional programs, based primarily on program transformation. The debugger is designed to assist the detection and location of errors in programs which produce incorrect results for some or all of their inputs. We define the transformation over a core functional language, and pay close attention to the treatment of curried function applications. We consider the space complexity of the debugger, and sketch a method for improvement. We use Haskell as the target of the transformation, and consider extending the transformation to support the entire language.

1 Introduction

The lack of debugging tools for lazy functional languages is considered detrimental to their popularity (Wadler [22]). Referential transparency, static type systems and implicit memory management tend to help programmers write correct programs. However, these features do not always prevent programs from behaving in ways that are not intended.

We divide the task of debugging into two parts: performance and correctness. Performance debugging concerns programs that produce correct results, but are either too slow or consume too much space. Determining the reasons for such bugs is traditionally the realm of profiling, and not discussed further in this paper. Correctness debugging concerns programs that produce incorrect results for some, or all, of their inputs. Declarative debugging has been considered a good candidate for locating errors in lazy functional languages because it allows the user to reason at a high level of abstraction. More information and advocacy for declarative debugging can be found in [8, 19, 14].

In this paper we present a means for debugging lazy functional programs based primarily on program transformation. The idea of producing a debugger in this way is not new, and has been studied elsewhere [7, 14, 19, 1]. Our goal is to solve two important outstanding issues in the field: portable support for higher-order programming and space efficiency. This work has culminated in a prototype debugger for a subset of Haskell 98.[1] The primary focus of this paper is the program transformation, which we cover in great detail. After presenting

[1] Hereafter we refer to Haskell 98 simply as Haskell.

the transformation, we consider the space complexity of the debugger and sketch a means for improvement. We consider extending the debugger to full Haskell, discuss related work and finally conclude. Due to constraints on the length of the paper, we presume of the reader considerable familiarity with lazy functional programming languages.

2 Declarative debugging

The idea of using declarative semantics to guide a search for bugs in programs is due to Shapiro [17], with a focus on locating logical errors in Prolog programs. Several declarative debugging techniques for functional languages have been based on the creation and traversal of a tree which gives a semantics for the execution of a given program. We call this entity an *Evaluation Dependence Tree* (EDT).[2] The key feature of the EDT is that it reflects the syntactic dependencies of the program definition, rather than the dependencies due to evaluation. Each node in an EDT represents a function application that occurred during program execution. Function applications from the right hand side of the function definition form the children of the node. 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. Different approaches to debugging lazy functional languages are discussed in section 9.

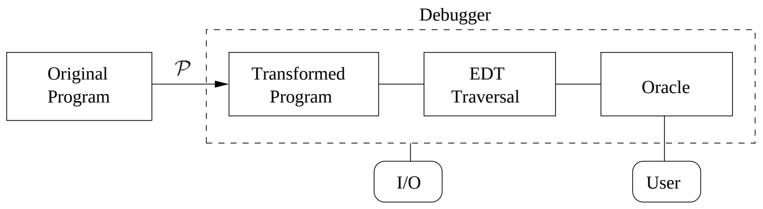

Fig. 1. Architecture of the debugger

The architecture of the debugging system is illustrated in figure 1. The original program source is transformed into a new program by $\mathcal{P}$, described in detail in section 6. The EDT produced by the transformed program is demanded by the EDT traversal module which constitutes a search for erroneous computations. Each node in the EDT describes a (possibly partial) function application. For saturated applications, a node contains (amongst other things) a representation of the function that was applied, the arguments to the function and the result of the application. Empty nodes are created for partial applications, and subsequently ignored by the debugger. The arguments and result are presented only

[2] *Evaluation Dependence Tree* is borrowed from Nilsson and Sparud [11].

to the extent that they were demanded in execution of the original program, respecting the semantics of lazy evaluation.

A node is *buggy* if it corresponds to an erroneous application, but none of its children are buggy (or it has no children) [7]. The EDT traversal is a search for such buggy nodes. The correctness of a node is determined by an oracle, which knows the intended semantics of each function in the program. Typically the oracle will query a human user of the debugger, however, the oracle may be automated in ways to reduce the burden on the user, for example by caching correctness results for certain nodes. If the top node in the EDT is erroneous, the debugger will eventually find a buggy node, such a node is called a *topmost buggy node*. Each node in the EDT also stores a reference to the defining equation for the function at the node. When a topmost buggy node is discovered the debugger reports the offending equation to the user of the debugger. Refinements of this basic idea can associate the source of errors to sub-parts of an equation, for example to a particular branch of a case statement.

We define a small functional language for the purposes of presenting the transformation algorithm. The abstract syntax of the language is given in figure 2,[3] and is a subset of the richer syntax of Haskell [12]. We consider extending the transformation to full Haskell in section 8. Before transformation, we assume that our object program has been type checked using the typing algorithm of Haskell [5]. For convenience of presentation we make Haskell the target of the transformation, and present supporting code in Haskell also.

$$
\begin{array}{lll}
f \in \text{Type Constructors} & & \\
v \in \text{Type Variables} & & \\
t \in \text{Types} & ::= & v \mid t_1 \to t_2 \mid f\ t_1 \ldots t_z \\
s \in \text{Type Schemes} & ::= & \forall\ \overline{v}\ .\ t \\
x \in \text{Variables} & & \\
c \in \text{Data Constructors} & & \\
d \in \text{Declarations} & ::= & x :: s \mid x =: e \mid f\ v_1 \ldots v_z =: \{c_i\ t_1 \ldots t_{y_i}\}_{i=1}^{w} \\
e \in \text{Expressions} & ::= & \lambda x.e \mid e_1\ e_2 \mid x \mid c \mid \text{let}\ d_1\ \ldots\ d_w\ \text{in}\ e \\
 & & \mid\ \text{case}\ e\ \text{of}\ a_1 \ldots a_w \\
a \in \text{Alternatives} & ::= & c\ x_1 \ldots x_z \to e \mid x \to e \\
p \in \text{Programs} & ::= & d_1\ \ldots\ d_n
\end{array}
$$

Fig. 2. Abstract syntax of a small functional language

3 A meta representation of values

Nodes of the EDT store representations of the arguments and results of function applications. In order to store representations of values of many different types

[3] We use ‘ =:’ for declarations to avoid ambiguity in the program transformation.

in the same tree it is necessary to implement a meta-representation of all values in the one type. That is the role of the type *Meta* below.

$$data\ Meta\ =\ MApp\ String\ [Meta]\mid MInt\ Int\mid\ \ldots\ \mid MFun\ String\mid MUneval$$

Constructor applications are encoded by *MApp* using a string to represent the constructor and a list of *Meta* values to encode the arguments. It would be possible to represent the built-in types *Int*, *Integer*, *Char*, *Float* and *Double* as nullary constructors. However, for efficiency reasons such values are encoded as themselves (lifted into the *Meta* type). Functional values are represented as strings, based on identifiers from the program text. The text for lambda abstractions is based on the syntax of Haskell. Partial applications are represented as strings with a space in between the function and each argument. Values that were not evaluated to Weak Head Normal Form are represented by the constructor *MUneval*.

Sharing within a data value is not preserved in the meta-representation. This does not effect the correctness of the debugger, however, the meta-representation may be much less space efficient than its underlying value.[4]

The debugger must convert arbitrary values from a computation into values of type *Meta*. We call such a conversion *reification*. Currently this is implemented by an impure primitive: $reify\ ::\ a \rightarrow Meta$. Such a function is impure because its result depends on the extent to which its argument has been computed, which may differ during the execution of a program. This impurity is *safe* because the meta-representation of values are only demanded after the computation of the original program is finished (therefore the underlying value is in its final state of evaluation).

It has been proposed by Sparud [19] (and discussed at further length in Pope and Naish [15]) to use Haskell's type classes to implement an overloaded form of reification. The idea is to create a class which implements a *reify* method as above. All types that are used in a program are automatically made instances of this class by induction over the definition of the type. There are (at least) two problems that limit this approach. First, it is not always possible to statically disambiguate overloaded expressions in Haskell without the use of type annotations. Such *unresolved overloading* is detected by the process of *context reduction* in the Haskell type system [13, 5]. It may be possible to avoid ambiguity for the special case of reification by modifying the rules of context reduction, although this remains a topic of research. Second, types with arguments that have higher-order kinds pose a difficulty for deriving class instances, for example: $data\ F\ t\ =\ C\ (t\ (F\ t))$. The problem, and a possible solution are discussed in detail in Hinze and Peyton-Jones [4], it is unclear whether future definitions of Haskell will allow class derivation for such higher-order kinded types.

For the reasons mentioned above we require *reify* to be implemented as a language primitive. In our current prototype we make use of primitives pro-

[4] In the worst case the size of the meta-representation may be exponential in the amount of memory used by the data represented.

vided by the Hugs[5] interpreter for peeking at the internal representation of values. The portability of our debugging system is dependent on the provision of implementation-specific versions of *reify*. The difficulty of this is not known, but certainly exacerbated by implementations that discard the names of data constructors after compilation.

4 The EDT

The EDT is represented with the following type:

$$\mathit{data}\ \mathit{EDT}\ =\ \mathit{Redex}\ \mathit{String}\ [\mathit{Meta}]\ \mathit{Meta}\ [\mathit{EDT}]\ |\ \mathit{Void}\ |\ \mathit{Local}\ [\mathit{EDT}]$$

The evaluation of saturated applications are represented by the *Redex* constructor. Each such node contains (in order): the function that was applied, representations of the arguments that the function was applied to, a representation of the result of the application, and links to further EDT nodes for applications that were evaluated as a result of this particular reduction. Partial applications are represented in the EDT by the constructor *Void*. Partial applications are not reducible and therefore contain no useful information for the debugger, thus the *Void* constructor is nullary. Why then do we bother to represent partial applications when they are just empty nodes? In a higher-order polymorphic language it is impossible to always tell whether an application is full or partial, however, the correctness of the program transformation relies on a consistent treatment of applications. The end result is that *all* applications must return a value and an EDT node. The empty nodes resulting from partial applications are ignored by the EDT traversal component of the debugger, and thus do not alter the outcome of the debugging session. The idea of using empty nodes to represent partial applications is due to Caballero and Rodríguez-Artalejo [1]. The *Local* constructor is added to simplify the treatment nested scopes introduced by case and let expressions.

5 Encoding functions

We introduce the type F to encode functional arguments of applications.

$$\begin{aligned} &\mathit{data}\ F\ a\ b\ =\ F\ (a \to (b, \mathit{EDT}))\ \mathit{String} \\ &\mathit{apply}\ (F\ f\ \_\ )\ =\ f \end{aligned}$$

The F encoding performs two roles. First, it transforms the function into a representation where all partial applications return a value and an EDT. When we consider the transformation of function definitions, we will see that saturated applications also return a value and an EDT, which means that all function applications be they partial or full can be treated in the same way. The main

[5] www.haskell.org/hugs

benefit being preservation of type-correctness in the transformed program. Second, it encodes all partial applications of a function with a string. The string serves as the printable representation of a partial application and will be displayed whenever the value of such a higher order argument is demanded by the debugger. The function can be selected from the F encoding using *apply*.

F encodings are created using one of the $partial_k$ functions below, where k corresponds to the number of arguments the application requires until it is saturated. The maximum arity of all functions in the program forms the upper bound for k, typically this number is small.

$$
\begin{array}{l}
partial_1 :: (a \to (b, EDT)) \to String \to F\ a\ b \\
partial_1\ f\ s\ =\ F\ f\ s \\
partial_2 :: (a \to b \to (c, EDT)) \to String \to F\ a\ (F\ b\ c) \\
partial_2\ f\ s\ =\ F\ (\lambda\, v\, .\, (partial_1\ (f\ v)\ (mkString\ s\ v),\ Void))\ s \\
\vdots \\
partial_m :: (t_1 \to \ldots \to t_m \to (t_{m+1}, EDT)) \to String \to F\ t_1\ \ldots\ (F\ t_m\ t_{m+1})\ \ldots \\
partial_m\ f\ s\ =\ F\ (\lambda\, v\, .\, (partial_{m-1}\ (f\ v)\ (mkString\ s\ v),\ Void))\ s
\end{array}
$$

String representations of partial applications are generated by *mkString*. A string naming a function (or partial application of a function) and a string representation of an arbitrary value as argument to that function are concatenated together, separated by a space. The argument values are converted to string by *toString*, which converts its argument to type *Meta* using *reify* and then converts that into a string, thus $toString = metaToString\ .\ reify$ (assuming a suitable definition of *metaToString*).

$$
\begin{array}{l}
mkString :: String \to a \to String \\
mkString\ x\ y\ =\ x\ +\!\!+\ \text{“ ”}\ +\!\!+\ toString\ y
\end{array}
$$

The string encoding of partial applications is used when we wish to reify a higher-order argument. The definition of *reify* in such a case is simple:

$$reify\ (F\ \_\ s)\ =\ MFun\ s$$

It is instructive to see the use of this encoding in an example. Consider the functions $max :: Int \to Int \to Int$ which computes the maximum of two integers and $map :: (a \to b) \to [a] \to [b]$. Under the debugging transformation described in section 6, the types of these functions become $max :: Int \to Int \to (Int, EDT)$ and $map :: (F\ a\ b) \to [a] \to ([b], EDT)$ respectively. Notice carefully how the type of the higher-order argument to map is altered. Now consider the expression: *map max*. Since *max* is a partial application which expects two arguments we must encode it with $partial_2$: $map\ (partial_2\ max\ \text{“max”})$. If we expand this expression using the definition of $partial_2$ we get:

$$map\ (F\ (\lambda\, v\, .\, (partial_1\ (max\ v)\ (mkString\ \text{“max”}\ v),\ Void))\ \text{“max”})$$

Expanding again using the definition of $partial_1$ gives:

$$map\ (F\ (\lambda\ v\ .\ (F\ (max\ v)\ (mkString\ \text{“max”}\ v),\ Void))\ \text{“max”})$$

The argument of *map* now has type $F\ Int\ (F\ Int\ Int)$, which is the encoded form of $Int \rightarrow Int \rightarrow Int$. Consider what happens when the call to the transformed *map* is evaluated. Let us presume that the higher-order argument of map is applied to the value 5. To apply an encoded function we use *apply* to select the function from its encoding, in this case we will get:

$$apply\ (F\ (\lambda\ v\ .\ (F\ (max\ v)\ (mkString\ \text{“max”}\ v),\ Void))\ \text{“max”})\ 5$$

which simplifies to:

$$(F\ (max\ 5)\ \text{“max 5”},\ Void)$$

Which has type $(F\ Int\ Int,\ EDT)$. The first component of this tuple will be returned as an element in the list produced by the call to map.

6 The debugging transformation

6.1 Naive form

We now define a program transformation that maps an object program into one suitable for debugging. The transformation is mostly syntax directed, however type information is needed for the treatment of function definitions and applications. We use double square brackets $[\![-]\!]$ to indicate parameters which range over syntactic expressions and `typewriter` font to indicate tokens which are to be interpreted verbatim. Upper case greek letters Γ, Δ, Σ and Ω range over sets of declarations. For simplicity, we allow ourselves to bind patterns to expressions, for example $(a, b) = x$. Lower case greek letters θ, κ, and ϕ range over sets of variables. Sets are denoted as elements within curly braces $\{-\}$, and are joined using the union operator $\bigcup$. The empty set is $\emptyset$. Angular brackets $\langle - \rangle$ are used to indicate products. A variable with a hat (as in $\hat{var}$) names a fresh variable, not defined previously in its scope. The function *text* produces a string literal from its syntactic argument, *setToList* converts a set of values into a literal list, *arity* computes the number of arguments for an expression based on its type, and *unwords* concatenates a list of strings with a single space in between each. A variable is considered *lambda bound* if it is introduced by a lambda abstraction or by the left-hand-side of a case alternative. A variable is considered *let bound* if it is introduced by a variable declaration, either at the top level or within a let expression.

1. Programs

$$\mathcal{P}\ [\![d_1,\ \ldots,\ d_n]\!]\ =\ \mathcal{D}\ [\![d_1]\!],\ \ldots,\ \mathcal{D}\ [\![d_n]\!]$$

2. Type Schemes and Types

$$\begin{aligned}
\mathcal{S}\ [\![\forall\overline{v}\ .\ t]\!] &= \forall\overline{v}\ .\ \mathcal{T}\ [\![t]\!]\\
\mathcal{T}\ [\![v]\!] &= (v,\ \texttt{EDT})\\
\mathcal{T}\ [\![t_1 \to t_2]\!] &= (\mathcal{U}\ [\![t_1]\!]) \to (\mathcal{T}\ [\![t_2]\!])\\
\mathcal{T}\ [\![f\ t_1\ \ldots\ t_2]\!] &= (f\ (\mathcal{U}\ [\![t_1]\!])\ \ldots\ (\mathcal{U}\ [\![t_2]\!]),\ \texttt{EDT})\\
\mathcal{U}\ [\![v]\!] &= v\\
\mathcal{U}\ [\![t_1 \to t_2]\!] &= \texttt{F}\ (\mathcal{U}\ [\![t_1]\!])\ (\mathcal{U}\ [\![t_2]\!])\\
\mathcal{U}\ [\![f\ t_1\ \ldots\ t_2]\!] &= f\ (\mathcal{U}\ [\![t_1]\!])\ \ldots\ (\mathcal{U}\ [\![t_2]\!])
\end{aligned}$$

3. Data Declarations

$$\begin{aligned}
&\mathcal{D}\ [\![f\ v_1\ \ldots\ v_z\ =:\ \{c_i\ t_1\ \ldots\ t_{y_i}\}_{i=1}^{w}]\!]\\
&\quad = f\ v_1\ \ldots\ v_z\ =:\ \{c_i\ (\mathcal{U}\ [\![t_1]\!])\ \ldots\ (\mathcal{U}\ [\![t_{y_i}]\!])\}_{i=1}^{w}
\end{aligned}$$

4. Type Annotations and Variable Declarations

$$\begin{aligned}
&\mathcal{D}\ [\![x\ ::\ s]\!]\ =\ x\ ::\ \mathcal{S}\ [\![s]\!]\\
&\mathcal{D}\ [\![x\ =:\ e]\!]\ =\ x\ =:\ \mathcal{L}\ [\![e]\!]\ (\mathit{text}\ x)
\end{aligned}$$

5. EDT Construction

$$\begin{aligned}
&\mathcal{L}\ [\![\lambda\, a_1\ \ldots\ \lambda\, a_n\ .\ e]\!]\ \mathit{funrep}\\
&\quad = \lambda\, a_1\ \ldots\ \lambda\, a_n\ .\,\text{let}\\
&\qquad\qquad \Delta \bigcup \{\ \hat{edt}\ =:\ \texttt{Redex}\ \mathit{funrep}\\
&\qquad\qquad\qquad\qquad [\texttt{reify}\ a_1,\ \ldots,\ \texttt{reify}\ a_n]\\
&\qquad\qquad\qquad\qquad (\texttt{reify}\ e')\\
&\qquad\qquad\qquad\qquad (\mathit{setToList}\ \kappa)\ \}\\
&\qquad\quad \text{in}\\
&\qquad\quad (e',\ \hat{edt})\\
&\qquad \text{where :}\\
&\qquad n\ \geq\ 0 \text{ and } e \text{ is not a lambda expression}\\
&\qquad \langle e',\ \Delta,\ \kappa\rangle\ =\ \mathcal{E}\ [\![e]\!]\ \emptyset\ \emptyset
\end{aligned}$$

6. Lambda Abstractions

$$\begin{aligned}
&\mathcal{E}\ [\![\lambda\, x\ .\ e]\!]\ \Delta\ \kappa\ =\ \langle\hat{w},\ \Delta \bigcup \Gamma,\ \kappa\rangle\\
&\qquad \text{where :}\\
&\qquad \Gamma\ =\ \{\ \hat{\mathit{lamrep}}\ =:\ \mathit{text}\ (\lambda\, x\ .\ e)\\
&\qquad\qquad \hat{w}\ =:\ \texttt{partial}_n\ (\mathcal{L}\ [\![\lambda\, x\ .\ e]\!]\ \hat{\mathit{lamrep}})\ \hat{\mathit{lamrep}}\ \}\\
&\qquad n\ =\ \mathit{arity}\ (\lambda\, x\ .\ e)
\end{aligned}$$

7. Expression Applications

$$\begin{aligned}
&\mathcal{E}\ [\![e_1\ \ e_2]\!]\ \Delta\ \kappa\ =\ \langle\hat{v},\ \Sigma \bigcup \Omega,\ \phi \bigcup \{\hat{t}\}\rangle\\
&\qquad \text{where :}\\
&\qquad \langle e_1',\ \Gamma,\ \theta\rangle\ =\ \mathcal{E}\ [\![e_1]\!]\ \Delta\ \kappa\\
&\qquad \langle e_2',\ \Sigma,\ \phi\rangle\ =\ \mathcal{E}\ [\![e_2]\!]\ \Gamma\ \theta\\
&\qquad \Omega\ =\ \{(\hat{v},\ \hat{t})\ =:\ \texttt{apply}\ e_1'\ e_2'\}
\end{aligned}$$

8. Variables

$$\mathcal{E}\,[\![x]\!]\,\Delta\,\kappa \;=\; \begin{cases} \langle x,\ \Delta,\ \kappa\rangle, \text{ if } x \text{ is lambda bound} \\ \langle \hat{v},\ \Delta \bigcup \Gamma,\ \kappa \bigcup \{\hat{t}\}\rangle, \text{ if } (\mathit{arity}\ x) \;=\; 0 \\ \langle \hat{w},\ \Delta \bigcup \Sigma,\ \kappa\rangle, \text{ otherwise} \end{cases}$$

where :
$\Gamma \;=\; \{(\hat{v},\ \hat{t}\,) \;=:\; x\}$
$\Sigma \;=\; \{\hat{w} \;=:\; \texttt{partial}_n\ x\ (\mathit{text}\ x)\}$
$n \;=\; \mathit{arity}\ x$

9. Data Constructors

$$\mathcal{E}\,[\![c]\!]\,\Delta\,\kappa \;=\; \begin{cases} \langle c,\ \Delta,\ \kappa\rangle, \text{ if } (\mathit{arity}\ c) \;=\; 0 \\ \langle \hat{w},\ \Delta \bigcup \Gamma,\ \kappa\rangle, \text{ otherwise} \end{cases}$$

where :
$\Gamma \;=\; \{\hat{w} \;=:\; \texttt{partial}_n\ (\lambda\ v_1\ \ldots\ v_n\ .\ (c\ v_1\ \ldots\ v_n,\ \texttt{Void}))\ (\mathit{text}\ c)\}$
$n \;=\; \mathit{arity}\ c$

10. Let Expressions

$$\mathcal{E}\,[\![\text{let } d_1\ \ldots\ d_n \text{ in } e]\!]\,\Delta\,\kappa \;=\; \langle \hat{v},\ \Delta \bigcup \Gamma,\ \kappa \bigcup \{\hat{t}\}\rangle$$

where :
$\langle e',\ \Sigma,\ \theta\rangle \;=\; \mathcal{E}\,[\![e]\!]\,\emptyset\,\emptyset$
$\Gamma \;=\; \{\ (\hat{v},\ \hat{t}\,) \;=:\; \text{let}$
$\qquad\qquad \{\ \mathcal{D}\,[\![d_1]\!]\ \ldots\ \mathcal{D}\,[\![d_n]\!]\ \} \bigcup \Sigma$
$\qquad\quad \text{in}$
$\qquad\quad (e',\ \texttt{Local}\ (\mathit{setToList}\ \theta))\ \}$

11. Case Expressions

$$\mathcal{E}\,[\![\text{case } e \text{ of } a_1\ \ldots\ a_w]\!]\,\Delta\,\kappa \;=\; \langle \hat{v},\ \Gamma \bigcup \Sigma,\ \theta \bigcup \{\hat{t}\}\rangle$$

where :
$\langle e',\ \Gamma,\ \theta\rangle \;=\; \mathcal{E}\,[\![e]\!]\,\Delta\,\kappa$
$\Sigma \;=\; \{\ (\hat{v},\ \hat{t}) \;=:\; \text{case } e' \text{ of } \mathcal{A}\,[\![a_1]\!]\ \ldots\ \mathcal{A}\,[\![a_w]\!]\ \}$

12. Case Alternatives

$$\mathcal{A}\,[\![c\ x_1\ \ldots\ x_z\ \mapsto\ e]\!]\,\Delta\,\kappa \;=\; \langle c\ x_1\ \ldots\ x_z\ \mapsto\ e',\ \Delta,\ \kappa\rangle$$

where :
$\langle e'',\ \Gamma,\ \theta\rangle \;=\; \mathcal{E}\,[\![e]\!]\,\emptyset\,\emptyset$
$e' \;=\; \text{let } \Gamma \text{ in } (e'',\ \texttt{Local}\ (\mathit{setToList}\ \theta))$

Programs are transformed by $\mathcal{P}$ (rule 1) and type schemes are transformed by $\mathcal{S}$ (rule 2), both are straightforward. Types are transformed by $\mathcal{T}$ and $\mathcal{U}$ (rule 2), the first of which ensures that the result type is paired with an EDT, the second of which ensures that higher-order argument types are F encoded. For example:

$$\begin{aligned}
&\mathcal{T}\ (a{\to}b){\to}a{\to}b\\
&\Rightarrow\ (\mathcal{U}\ a{\to}b)\ \to\ (\mathcal{T}\ a{\to}b)\\
&\Rightarrow\ (\mathtt{F}\ (\mathcal{U}\ a)\ (\mathcal{U}\ b))\ \to\ (\mathcal{U}\ a)\ \to\ (\mathcal{T}\ b)\\
&\Rightarrow\ \mathtt{F}\ a\ b\ \to\ a\ \to\ (b,\ \mathtt{EDT})
\end{aligned}$$

Declarations are transformed by $\mathcal{D}$. We ensure that higher order types are F encoded in the definition of data constructors (rule 3). Type annotations are straightforward (rule 4).

Lambda abstractions are the principal way of introducing new functions in a program. When a lambda abstraction is (let) bound to a variable, we take that variable to be the name of the function. When a lambda abstraction is unbound, we take the text of the whole expression to be the name of the function. In both cases we need to create an EDT for the function. Construction of EDTs is handled by $\mathcal{L}$ which takes an expression and a string as arguments (rule 5). If the expression is a lambda abstraction then it denotes a function, in such a case we presume that the expression is eta-expanded so that if its arity is n there will be exactly n unique variables introduced at the beginning of the abstraction. We require all arguments to functions to be named so that each may be reified and represented in the EDT, eta-expansion is performed as a pre-process phase before transformation. In the case of a non-functional expression the EDT node will contain an empty list of reified arguments. The string argument (*funrep*) to $\mathcal{L}$ provides the name for the EDT node (compare the use of $\mathcal{L}$ in rule 4 and rule 6). Applications within the expression are let bound (denoted by Δ) and the EDT nodes generated by each application are named (denoted by κ). The set of variables in κ form the children of the EDT node. The transformed expression is denoted by e' and is paired with the EDT to form the new result of the entire expression. For example (ellipses are filled in by the transformation $\mathcal{E}\ (f\ g\ j)\ \emptyset\ \emptyset$):

$$\begin{aligned}
&\mathcal{L}\ (\lambda\, j\ .\ f\ g\ j)\ \text{“foo”}\\
&\Rightarrow\ \lambda\, j\ .\ \text{let}\\
&\qquad\qquad (v_1,\ t_1)\ =\ \ldots;\ \ldots\ (v_n,\ t_n)\ =\ \ldots;\\
&\qquad\qquad \mathit{edt}\ =\ \mathtt{Redex}\ \text{“foo”}\ [\mathtt{reify}\, j]\ (\mathtt{reify}\ \ldots)\ [t_1,\ \ldots,\ t_n]\\
&\qquad\quad \text{in}\\
&\qquad\quad (\ldots,\ \mathit{edt})
\end{aligned}$$

Expressions are transformed by $\mathcal{E}$ (rules 6 - 11). It takes three arguments: an expression, a set of bindings, and a set of variables. Both sets are initially empty. It returns a transformed expression (this will be either a variable or a nullary constructor), a new set of bindings which name the results of each function application, and a new set of variables which name the EDT nodes produced by each application. All applications are let bound and all partial applications are appropriately encoded. The purpose of unfolding the expression is to name intermediate EDT nodes and results produced by each function application. The intermediate nodes form the children of the EDT for the function that contains the expression.

Lambda abstractions, functional let bound variables, and non-nullary constructors require F encoding, and are each preceded by a call to $partial_n$, where n is the arity of the entity (see rules 6, 8 and 9). Functional lambda bound variables do not require F encoding, since at runtime they will be bound to a value that is encoded. Non functional let bound variables need special treatment because their definition will be modified to return a value plus an EDT. Nullary constructors remain unchanged. Since all functional values are encoded, all function applications must be preceded by a call to *apply*, so that the function may be *de*coded before it is applied to an argument (rule 7). Transforming the expression *filter* x (*reverse* z), where *filter* has arity 2 and *reverse* has arity 1 and x and z are lambda bound, gives the following bindings:

$$\begin{aligned}
w_1 &= \mathit{partial}_1\ \mathit{reverse}\ \text{“reverse”} \\
w_2 &= \mathit{partial}_2\ \mathit{filter}\ \text{“filter”} \\
(v_1,\ t_1) &= \mathit{apply}\ w_1\ z \\
(v_2,\ t_2) &= \mathit{apply}\ w_2\ x \\
(v_3,\ t_3) &= \mathit{apply}\ v_2\ v_1
\end{aligned}$$

The treatment of let and case expressions is relatively straightforward (rules 10 - 12). The *Local* EDT constructor collects many EDT nodes into one, simplifying the construction of the parent EDT. Care is taken to preserve the lexical scopes introduced by let and case expressions: the new declarations produced by unfolding nested expressions are not be propagated upwards.

6.2 Improved form

There is much redundancy in the code resulting from the naive transformation, due to the assumption that all applications are partial. In practice only some applications are partial, the rest being saturated, or over-saturated. The effect of the redundancy is a loss of efficiency in the transformed program, due to the unnecessary construction of *Void* EDT nodes and the encoding of functional values which are then immediately decoded. Consider the example expression *filter* x (*reverse* z), as discussed previously. Both applications of *filter* and *reverse* are saturated, so we can optimise the bindings that are produced by the transformation to:

$$\begin{aligned}
(v_1,\ t_1) &= \mathit{reverse}\ z \\
(v_2,\ t_2) &= \mathit{filter}\ x\ v_1
\end{aligned}$$

To avoid this redundancy we modify the algorithm in the following ways. For an application of a let bound variable f with arity k:

$$f\ e_1\ \ldots\ e_n$$

the following bindings are produced:

$$
\begin{array}{l}
\text{if } k > n : \\
\qquad \hat{w} =: \texttt{partial}_{k-n}\,(f\ e_1\ \ldots\ e_n) \\
\qquad\qquad\qquad\qquad (\texttt{unwords}\,[\mathit{text}\,f,\ \texttt{toString}\,e_1,\ \ldots,\ \texttt{toString}\,e_n]) \\
\text{if } k = n : \\
\qquad (\hat{v},\ \hat{t}) =: f\ e_1\ \ldots\ e_n \\
\text{if } k < n : \\
\qquad (\hat{v}_1,\ \hat{t}_1) =: f\ e_1\ \ldots\ e_k \\
\qquad (\hat{v}_2,\ \hat{t}_2) =: \texttt{apply}\ \hat{v}_1\ e_{k+1} \\
\qquad \vdots \\
\qquad (\hat{v}_{n-k},\ \hat{t}_{n-k}) =: \texttt{apply}\ \hat{v}_{(n-k)-1}\ e_n
\end{array}
$$

The first case corresponds to a true partial application, in which we apply the appropriate $\mathit{partial}_k$ function based on the number of arguments that are missing. A representation of the partial application is constructed from the name of the function and string representations of the arguments. The second case corresponds to a saturated application, in which case we perform the application directly and bind the result to a pair naming the value and the EDT node produced. The third case corresponds to an over-saturation, where additional arguments must be applied one at a time. For an application of a non-nullary constructor c with arity k:

$$c\ e_1\ \ldots\ e_n$$

the following bindings are produced:

$$
\begin{array}{l}
\text{if } k > n : \\
\qquad \hat{w} =: \texttt{partial}_{k-n}\,(\lambda\, v_1\ \ldots\ \lambda\, v_{k-n}\,.\,(c\ e_1\ \ldots\ e_n\ v_1\ \ldots\ v_{k-n},\ \texttt{Void})) \\
\qquad\qquad\qquad\qquad (\texttt{unwords}\,[\mathit{text}\ c,\ \texttt{toString}\,e_1,\ \ldots,\ \texttt{toString}\,e_n]) \\
\text{if } k = n : \\
\qquad c\ e_1\ \ldots\ e_n
\end{array}
$$

The first case is a true partial application of the constructor, the second case is a full application, which is unchanged by the transformation. It would be possible to also apply the same kind of improvement to applications where the function is another type of expression, for example a lambda abstraction, as in $(\lambda\ x\ .\ x)\ e$. In practice we believe such applications are less common, and in the interest of keeping the transformation as simple as possible we do not make such improvements. We illustrate example program transformations using the improved method in the appendix.

Naming lambda abstractions by their text representation (as in rule 6) seems *ad hoc* and less than satisfactory. The goal of the debugger is to present values in a manner which is most meaningful to the user. For lambda abstractions it is not entirely clear what the most meaningful representation is. The use of the program text for the expression is simple to implement, however, the occurrence of free variables in the expression may complicate the user's interpretation of what is displayed. A possible solution is to represent free variables by the value to which they are bound dynamically, rather than by their name. However, this may not be such a good idea if the representation of the value is very large. Our intuition is that it may be best to provide both the variable name, and also the value to which it is bound, allowing the user of the debugger to toggle the display as they see fit. This is not yet implemented.

7 Space Usage

Perhaps the largest drawback of the debugger that we have defined thus far is its prohibitive memory use. The problem is simple: the full construction of the EDT (even if done lazily) prohibits garbage collection. This means that any intermediate data values that could be collected during the execution of the original program are retained during the execution of the debugged program. Thus the memory consumed by the execution of the debugged program is proportional to the length in time of execution. This limits the use of the debugger to toy programs, and is particularly unsatisfactory.

To solve the problem we must first understand why it occurs. This is best illustrated by an example. Consider the naive reverse function, which reverses its argument list in a rather inefficient way (presuming a suitable definition for *append*):

```
nreverse :: [a]→[a]
nreverse = λ xs . case xs of
                     [ ] → [ ]
                     (h : t) → append (nreverse t) [h]
```

An intermediate list is constructed by the recursive call to *nreverse* which is then consumed by *append* and left as garbage, which can be collected. Consider now the transformed version of this function:

```
nreverse :: [a]→([a], EDT)
nreverse = λ xs . let
                     edt = Redex "nreverse" [reify xs] (reify v1) [t1]
                     (v1, t1) = case xs of
                                   [ ] → ([ ], Local [ ])
                                   (h : t) → let
                                                (v3, t3) = append v2 [h]
                                                (v2, t2) = nreverse t
                                              in
                                                (v3, Local [t2, t3])
                   in
                   (v1, edt)
```

The EDT node returned by a call to *nreverse*, contains a reference to the EDT node returned by the recursive call (t_2, via t_1). Consider then a debugging session. All functions in the program are transformed according to the rules set out earlier. Presume we have a function $mainBug :: (Int, EDT)$, which defines the top level of the computation that we wish to debug. The debugger begins by forcing the first component of the result, then it initiates a traversal of the EDT, which is of course constructed lazily. Forcing the first component of the result corresponds to an evaluation in the original program. Values which were previously garbage can no longer be collected because the top node of the EDT transitively refers to all intermediate values of the computation.

We propose to solve this problem by building the EDT one (or a few) nodes at a time, as a form of iterative deepening. In order to do this we need a means for reconstructing children nodes on demand. Reconsider the recursive call in the transformed definition of *nreverse*:

$$(v_2,\ t_2)\ =\ nreverse\ t$$

The problem is that we make an explicit reference (t_2) to the EDT that is produced from the call. That reference prohibits the collection of all the intermediate lists produced by that call. To avoid the problem we need to separate the computation of values (as in v_2) from the construction of EDTs (as in t_2). To do this we bind the recursive call to a function:

$$exp\ (\ )\ =\ nreverse\ t$$

We compute the original value of the recursive call in this way:

$$v_2\ =\ fst\ (exp\ (\ ))$$

Now the EDT node is not referenced and garbage can be collected as usual. But how to we recover the EDT node if we need it at a later stage? For this we need a means of recomputation. We want to be able to reconstruct the tree by calling the recursive call again. In other words we want to be able to recompute '*nreverse t*' *only* to the extent that it was evaluated originally. In a lazy language, recomputation of sub-expressions is difficult because the extent that the expression is evaluated depends on its dynamic context. Storing the entire dynamic context of the expression would in general be very expensive. The trick is to make use of reification. If we compute the *Meta* representation of the computed value (as in $reify\ v_2$), we have sufficient information to recompute the expression at a later time, since the *Meta* representation tells us how much of the final result was needed. The details of re-evaluation are outside the scope of this paper, however, experiments with our prototype debugger augmented with a re-evaluation scheme have been encouraging.

8 Future work

How difficult is it to extend the debugger to support full Haskell? There are four areas that need consideration. Firstly, the debugger must be able to support the full syntax of the language. This may be achieved by extending the transformation rules to the additional syntactic constructs, or de-sugaring those constructs into a simpler language (or some happy medium of both). To ensure that the debugging session is meaningful to the user, it is important to retain certain properties of the original program such as identifier names and code structure. Although it is tedious to define the transformation over a richer syntax, we do not believe there are any fundamental difficulties in this respect. Secondly, the debugger must support programs that perform I/O. There must be a clear delineation between the I/O performed by the debugger and the I/O performed by the program being debugged. If the debugger uses a re-evaluation scheme as outlined in section 7, it is essential that no attempt is made to re-evaluate computations that perform I/O. Only purely functional code can safely be re-evaluated without changing the semantics of the program execution. Obviously the debugger should avoid attempting to determine the correctness of the primitive I/O operations. Thirdly, the current transformation is not defined over a language with type classes. Although there does not appear to be any intuitive reason for difficulty, the transformation does modify types in the program, and we must be careful that such modification does not interfere with the semantics of type classes. Fourthly, Haskell programs are typically defined over a number of modules. We presume for the present that all modules will undergo transformation, although for efficiency we would like to refine this to a subset of modules where possible.

Outside of the program transformation there is much work to be done on the usability of the debugger. Two key aspects that need attention are the efficient traversal of the EDT and the presentation of data values.

The number of nodes in the EDT corresponds roughly to the duration of the computation. In order to locate program bugs in a reasonable amount of time, the debugger should try to minimise the number of nodes that must be visited. A simple depth-first search of the EDT will be impractical for many programs, and so certain search heuristics are required.

One commonly cited problem with declarative debugging tools is that they require the user to reason about the correctness of function applications with arguments or results that can be large and or complex. The oracle can assist the determination of correctness by reducing the amount of information that the user must initially consume. Certainly for large structured data values a browsing facility is required. Further improvements are possible if the oracle is aware of the type schemes for functions in the program. For example, consider the function *length* :: $[a] \rightarrow Int$, which computes the length of a list. The traditional approach to presenting an application of list is something like:

```
length ["Fat", "rat", "sat", "flat"] => 4, Is this correct?
```

and

```
length [[1,2,3], [3,2,1], [], [12]] => 4, Is this correct?
```

Given that *length* is polymorphic in the elements of the list, it should be possible to determine its correctness irrespective of the actual values in the list, so the oracle could simplify both of these questions to the one:

```
    length [e1, e2, e3, e4] => 4, Is this correct?
```

Generalising questions in this way may also help to reduce the total number of questions asked, since many instances of a question fold into one, and the oracle may remember some of the questions it has asked previously, and thus avoid repeating them again. In situations where the generalised form of the question is difficult to answer the user may request that the oracle provide one with the values further instantiated.

9 Related work

This paper stems from earlier research by Naish and Barbour [7] (see also Naish [6]). In recent work [14], a prototype declarative debugger for Haskell was implemented. Only a subset of the language was supported, the most notable omission being arbitrary higher-order code. An attempt to support higher-order code and curried function definitions based on a type-directed program specialisation was proposed in [16]. Unfortunately the specialisation interacts badly with polymorphic recursion and separate compilation. The prohibitive space consumption of the EDT was considered in [7] and also [14], and the basic idea for solving the problem was also formulated in those papers. However, it is not until now that an implementation has been achieved.

A transformational approach to debugging wrong answers in Functional Logic Programs is described by Caballero and Rodríguez-Artalejo [1]. A family of intermediate functions are introduced for each potential partial application of every function defined in the program. This solves a long standing problem in transformational approaches to declarative debugging of languages that allow curried functions, as outlined in [7]. Our present treatment of partial applications is inspired by this technique. We extend their work in two ways. First, they require $n - 1$ intermediate functions for every function of arity n and $m - 2$ intermediate functions for every constructor of arity m. We only require k intermediate functions where k is the maximum arity of all functions in the program. In general, our approach will require significantly less intermediate functions and thus reduce the size of the transformed code. Second, our transformation encodes the representation of partial applications with strings, whereas they require the names of functions to be provided by the runtime environment. Their transformation is described for a very simple functional language and their debugging technique is based on the traversal of a tree akin to an EDT. They do not mention the problem of space consumed by the full construction of this tree.

Another declarative debugging scheme based on program transformation is described by Sparud [18, 19]. The transformation supports limited forms of higher-order programming and curried applications. Type classes are used for the reification of values, although the difficulties with this approach that we outlined in section 3 still apply.

Rather than generating the EDT by program transformation, Nilsson [8, 10] uses an instrumented runtime environment that constructs the tree as a side-effect of computation. The advantage of this approach is that it allows for greater control over, and access to, the runtime representation of values. A technique called *piecemeal tracing* [9] is employed to constrain the memory consumed by the EDT. The idea is to place an upper limit on the size of memory occupied by the EDT. Initially, only a partial tree is created. Debugging can proceed on the partial tree and may result in the identification of a bug. However, if the fringe of the tree is encountered and the maximum amount of memory is used, the program is re-evaluated from the beginning and a new EDT is created such that some nodes are discarded (or not created) to provide space to expand the tree. The disadvantage of this approach is the complexity of implementation. A whole new compiler for a large subset of Haskell (called Freya) was created for the purposes of providing the necessary instrumented runtime environment. Our motivation for employing program transformation was to simplify the implementation of the debugger and to take advantage of existing compiler technology.

A general framework for tracing, debugging and observing lazy functional computations based on reduction histories (or *Redex Trails*) has been proposed by Runciman *et al* [21, 20]. The trails record a rich amount of information about a computation and various post-processing tools have been developed to view the information in different ways, including declarative debugging [23]. The production of trails is based on a program transformation. The current implementation works with the nhc98 Haskell compiler[6], and supports almost all of Haskell 98. The main cost of recording Redex Trails is the space required to store the trail, the size of which being proportionate to the duration of the computation. To cope with the large space requirement, the trail is serialised and written to file rather than being maintained in main memory.

A means for displaying the evaluation of expressions in a running Haskell program is provided by HOOD[7] [3]. A type class is used to implement an overloaded function called *observe*. This function has the type of the identity function, and so it can freely be inserted into any expression without changing the type of the program. Calls to observe cause the reduction steps for an expression to be logged as a side-effect. The log faithfully reflects the partial evaluation of data. It can capture the evaluation of functions, which are represented extensionally as sets of input and output values. One advantage of HOOD is that it is easily combined into an existing program and it only relies on a few commonly implemented extensions to Haskell.

Detailed comparisons of HOOD, Freya and HAT are documented by Chitil *et al* [2]. The experiments that were used to evaluate the three systems avoided extensive use of higher-order functions, such as continuation passing style, higher-order combinators and monads. Easing the difficulty of debugging higher-order programs is an important area of future research.

[6] www.cs.york.ac.uk/fp/nhc98

[7] Haskell Object Observation Debugger

10 Conclusion

We have presented a program transformation for the declarative debugging of lazy functional programming languages, using a subset of Haskell as an example. The main improvements of our debugger over previous work are portable support for higher-order functions and conservative use of memory. Not all of the debugger can be implemented in the source language. We require the implementation of a primitive to generate meta-representations of values from a computation that respects lazy evaluation. The portability of the debugger is dependent on the availability of this primitive across language implementations. We also require a further primitive for re-evaluation of sub-expressions, however, thorough exposition of this is left for future work. Much of the complexity in the transformation is due to the presence of higher order values and polymorphism, which together make it difficult determine whether applications are partial or saturated. Partial applications (higher order values) are encoded, and allocated a special empty node in the tree which gives a semantics for the program. We believe that debugging is an important area of research for the lazy functional programming community and that declarative debugging is an attractive technique. Currently we have a prototype of our system which works with the Hugs Haskell interpreter. Scaling this debugging technology to real programs is an interesting challenge.

References

1. R. Caballero and M. Rodríguez-Artalejo. A declarative debugger of wrong answers for lazy functional logic programs. In M. Hanus, editor, *Proceedings of the International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001)*, pages 73–86, 2001.
2. O. Chitil, C. Runciman, and M. Wallace. Freja, Hat and Hood - a comparative evaluation of three systems for tracing and debugging lazy functional programs. In M. Mohnen and P. Koopman, editors, *Proceedings of the 12th International Workshop on Implementation of Functional Languages*, volume LNCS 2011, pages 176–193, 2001.
3. A. Gill. Debugging Haskell by observing intermediate data structures. Technical report, University of Nottingham, 2000. In Proceedings of the 4th Haskell Workshop, 2000.
4. R. Hinze and S. Peyton Jones. Derivable type classes, 2000. Submitted to the Haskell Workshop 2000.
5. M. Jones. Typing Haskell in Haskell. Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht, Utrecht, Netherlands, 1999.
6. L. Naish. Declarative debugging of lazy functional programs. *Australian Computer Science Communications*, 15(1):287–294, 1993.
7. L. Naish and T. Barbour. Towards a portable lazy functional declarative debugger. *Australian Computer Science Communications*, 18(1):401–408, 1996.
8. Henrik Nilsson. *Declarative Debugging for Lazy Functional Languages.* PhD thesis, Department of Computer and Information Science, Linköpings universitet, S-581 83, Linköping, Sweden, 1998.

9. Henrik Nilsson. Tracing piece by piece: affordable debugging for lazy functional languages. In *Proceedings of the 1999 ACM SIGPLAN international conference on Functional programming*, pages 36–47, Paris, France, 1999. ACM Press.
10. Henrik Nilsson. How to look busy while being as lazy as ever: The implementation of a lazy functional debugger. *Journal of Functional Programming*, August 2000. To appear.
11. Henrik Nilsson and Jan Sparud. The evaluation dependence tree as a basis for lazy functional debugging. *Automated Software Engineering*, 4(2):121–150, 1997.
12. J. Peterson, K. Hammond, L. Augustsson, B. Boutel W. Burton, J. Fasel, A. Gordon, J. Hughes, P. Hudak, T. Johnsson, M. Jones, E. Meijer, S. Peyton Jones, A. Reid, and P. Wadler. Haskell 98 Language Report. http://www.haskell.org/onlinereport/.
13. J. Peterson and M. Jones. Implementing type classes. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 227–236, 1993.
14. 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.
15. B. Pope and L. Naish. Reification for debugging. In T. Arts, editor, *Draft Proceedings of the International Workshop on the Implementation of Functional Programming Languages (IFL 2001)*, 2001.
16. B. Pope and L. Naish. Specialisation of higher-order functions for debugging. In M. Hanus, editor, *Proceedings of the International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001)*, pages 87–100, 2001.
17. E. Shapiro. *Algorithmic Program Debugging.* The MIT Press, 1982.
18. J. Sparud. A transformational approach to debugging lazy functional programs. Master's thesis, Chalmers University of Technology, Göteborg, Sweden, 1996.
19. J. Sparud. *Tracing and Debugging Lazy Functional Computations.* PhD thesis, Chalmers University of Technology, Sweden, 1999.
20. J. Sparud and C. Runciman. Complete and partial redex trails of functional computations. In T. Davie C. Clack, K. Hammond, editor, *Selected papers from 9th International Workshop on the Implementation of Functional Languages*, volume LNCS 1467, pages 160–177, 1997.
21. J. Sparud and C. Runciman. Tracing lazy functional computations using redex trails. In *PLILP*, pages 291–308, 1997.
22. P. Wadler. Why no one uses functional languages. *SIGPLAN Notices*, 33(8):23–27, 1998.
23. M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-view tracing for Haskell: a new hat. In *Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, pages 151–170, 2001.

A Transformation of map

The original definition:

```
$map$ :: $(a \to b) \to [a] \to [b]$
$map$ = $\lambda f\ l$ . case $l$ of
                 $[\,] \to [\,]$
                 $(h : t) \to (f\ h) : map\ f\ t$
```

The transformed definition:

```
$map$ :: $F\ a\ b \to [a] \to ([b], EDT)$
$map$
 = $\lambda f\ l$ . let
          $edt = EDT$ “map” $[reify\ f,\ reify\ l]\ (reify\ v_1)\ [t_1]$
          $(v_1,\ t_1)$
            = case $l$ of
                $[\,] \to ([\,],\ Local\ [\,])$
                $(h : t) \to$ let
                              $(v_2, t_2) = map\ f\ t$
                              $(v_1, t_1) = apply\ f\ h$
                            in
                            $((v_1 : v_2),\ Local\ [t_1,\ t_2])$
        in
        $(v_1,\ edt)$
```