Optimizing Generic Functions

Artem Alimarine and Sjaak Smetsers

Computing Science Institute
University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
alimarin@cs.kun.nl, sjakie@cs.kun.nl

Abstract. Generic functions are defined by induction on the structural representation of types. As a consequence, by defining just a single generic operation, one acquires this operation over any particular type. An instance on a specific type is generated by interpretation of the type's structure. A direct translation leads to extremely inefficient code that involves many conversions between types and their structural representations. In this paper we present an optimization technique based on compile-time symbolic evaluation. We prove that the optimization removes the overhead of the generated code for a considerable class of generic functions. The proof uses typing to identify intermediate data structures that should be eliminated. In essence, the output after optimization is similar to hand-written code.

1 Introduction

The role of generic programming in the development of functional programs is steadily becoming more important. The key point is that a single definition of a generic function is used to automatically generate instances of that function for arbitrarily many types. These generic functions are defined by induction on a structural representation of types. Adding or changing a type does not require modifications in a generic function; the appropriate code will be generated automatically. This eradicates the burden of writing similar instances of one particular function for numerous different data types, significantly facilitating the task of programming. Typical examples include generic equality, mapping, pretty-printing, and parsing.

Current implementations of generic programming [AP01,CHJ$^+$02,HP01], generate code which is strikingly slow because generic functions work with structural representations rather than directly with data types. The resulting code requires numerous conversions between representations and data types. Without optimization automatically generated generic code runs nearly 10 times slower than its hand-written counterpart.

In this paper we prove that compile-time (*symbolic*) evaluation is capable of reducing the overhead introduced by generic specialization. The proof uses typing to predict the structure of the result of a symbolic computation. More specifically, we show that if an expression has a certain type, say σ, then its symbolic normal form will contain no other data-constructors than those belonging to σ.

It appears that general program transformation techniques used in current implementations of functional languages are not able to remove the generic overhead. It is even difficult to predict what the result of applying such transformations on generic functions will be, not to mention a formal proof of completeness of these techniques.

In the present paper we are looking at generic programming based on the approach of kind-indexed types of Hinze [Hin00a], used as a basis for the implementation of generic classes of Glasgow Haskell Compiler (GHC) [HP01], Generic Haskell [CHJ+02] and Generic Clean [AP01]. The main sources of inefficiency in the generated code are due to heavy use of higher-order functions, and conversions between data structures and their structural representation. For a large class of generic functions, our optimization removes both of them, resulting in code containing neither parts of the structural representation (binary sums and products) nor higher-order functions introduced by the generic specialization algorithm.

The rest of the paper is organized as follows. In section 2 we give motivation for our work by presenting the code produced by the generic specialization procedure. The next two sections are preliminary; they introduce a simple functional language and the typing rules. In section 5, we extend the semantics of our language to evaluation of open expressions, and establish some properties of this so-called symbolic evaluation. In section 6 we discuss termination issues of symbolic evaluation of the generated code. Section 7 discusses related work. Section 8 reiterates our conclusions.

2 Generics

In this section we informally present the generated code using as an example the generic mapping specialized to lists. The structural representation of types is made up of just the unit type, the binary product type and the binary sum type [Hin99]:

$$\begin{aligned}
&\textbf{data}\ \mathbb{1} &&= \mathbb{1}\\
&\textbf{data}\ \alpha \times \beta &&= (\alpha, \beta)\\
&\textbf{data}\ \alpha + \beta &&= \mathsf{Inl}\ \alpha \mid \mathsf{Inr}\ \beta
\end{aligned}$$

For instance, the data types

$$\begin{aligned}
&\textbf{data}\ \mathsf{List}\ \alpha &&= \mathsf{Nil} \mid \mathsf{Cons}\ \alpha\ (\mathsf{List}\ \alpha)\\
&\textbf{data}\ \mathsf{Tree}\ \alpha\ \beta &&= \mathsf{Tip}\ \alpha \mid \mathsf{Bin}\ \beta\ (\mathsf{Tree}\ \alpha\ \beta)\ (\mathsf{Tree}\ \alpha\ \beta)
\end{aligned}$$

are represented as

$$\begin{aligned}
&\textbf{type}\ \mathsf{List}^{\circ}\ \alpha &&= \mathbb{1} + \alpha \times (\mathsf{List}\ \alpha)\\
&\textbf{type}\ \mathsf{Tree}^{\circ}\ \alpha\ \beta &&= \alpha + \beta \times \mathsf{Tree}\ \alpha\ \beta \times \mathsf{Tree}\ \alpha\ \beta
\end{aligned}$$

Note that the representation of a recursive type is not recursive.

The structural representation of a data type is isomorphic to that data type. The conversion functions establish the isomorphism:

$$
\begin{array}{ll}
\mathsf{to}_{\mathsf{List}} & : \mathsf{List}\ \alpha \to \mathsf{List}^{\circ}\ \alpha \\
\mathsf{to}_{\mathsf{List}} & = \lambda l.\mathsf{case}\ l\ \mathsf{of} \\
& \qquad \mathsf{Nil}\ \to\ \mathsf{Inl}\ \mathbb{1} \\
& \qquad \mathsf{Cons}\ x\ xs\ \to\ \mathsf{Inr}\ (x, xs) \\
\mathsf{from}_{\mathsf{List}} & : \mathsf{List}^{\circ}\ \alpha \to \mathsf{List}\ \alpha \\
\mathsf{from}_{\mathsf{List}} & = \lambda l.\mathsf{case}\ l\ \mathsf{of} \\
& \qquad \mathsf{Inl}\ u\ \to\ \mathsf{case}\ u\ \mathsf{of}\ \mathbb{1}\ \to\ \mathsf{Nil} \\
& \qquad \mathsf{Inr}\ p\ \to\ \mathsf{case}\ p\ \mathsf{of}\ (x, xs)\ \to\ \mathsf{Cons}\ x\ xs
\end{array}
$$

The generic specializer automatically generates the type synonyms for structural representations and the conversion functions.

Data types may contain the arrow type. To handle such types the conversion functions are packed into *embedding-projection pairs* [HP01]

$$\mathbf{data}\ \alpha \rightleftarrows \beta = \mathsf{EP}\ (\alpha \to \beta)\ (\beta \to \alpha)$$

The projections, the inversion and the (infix) composition of embedding-projections are defined as follows:

$$
\begin{array}{ll}
\mathsf{to} & : (\alpha \rightleftarrows \beta) \to (\alpha \to \beta) \\
\mathsf{to} & = \lambda x.\mathsf{case}\ x\ \mathsf{of}\ \mathsf{EP}\ t\ f\ \to\ t \\
\mathsf{from} & : (\alpha \rightleftarrows \beta) \to (\beta \to \alpha) \\
\mathsf{from} & = \lambda x.\mathsf{case}\ x\ \mathsf{of}\ \mathsf{EP}\ t\ f\ \to\ f \\
\mathsf{inv} & : (\alpha \rightleftarrows \beta) \to (\beta \rightleftarrows \alpha) \\
\mathsf{inv} & = \lambda x.\mathsf{EP}\ (\mathsf{from}\ x)\ (\mathsf{to}\ x) \\
\bullet & : (\beta \rightleftarrows \gamma) \to (\alpha \rightleftarrows \beta) \to (\alpha \rightleftarrows \gamma) \\
\bullet & = \lambda a.\lambda b.\mathsf{EP}\ (\mathsf{to}\ a\ \circ\ \mathsf{to}\ b)\ (\mathsf{from}\ b\ \circ\ \mathsf{from}\ a)
\end{array}
$$

For instance, the generic specializer generates the following embedding-projection pair for lists:

$$
\begin{array}{ll}
\mathsf{conv}_{\mathsf{List}} & : \mathsf{List}\ \alpha \rightleftarrows \mathsf{List}^{\circ}\ \alpha \\
\mathsf{conv}_{\mathsf{List}} & = \mathsf{EP}\ \mathsf{to}_{\mathsf{List}}\ \mathsf{from}_{\mathsf{List}}
\end{array}
$$

To define a generic (*polytypic*) function the programmer provides the basic *poly-kinded type* [Hin00b] and the instances on the base types. For example, the generic mapping is given by the type

$$\mathbf{type}\ \ \mathsf{Map}\ \alpha\ \beta = \alpha \to \beta$$

and the base cases

$$
\begin{array}{ll}
\mathsf{map}_{\mathbb{1}} & : \mathbb{1} \to \mathbb{1} \\
\mathsf{map}_{\mathbb{1}} & = \lambda x.\mathsf{case}\ x\ \mathsf{of}\ \mathbb{1}\ \to\ \mathbb{1} \\
\mathsf{map}_{\times} & : \forall \alpha_1 \alpha_2 \beta_1 \beta_2.(\alpha_1 \to \beta_1) \to (\alpha_2 \to \beta_2) \to (\alpha_1 \times \alpha_2 \to \beta_1 \times \beta_2) \\
\mathsf{map}_{\times} & = \lambda f.\lambda g.\lambda p.\mathsf{case}\ p\ \mathsf{of}\ (x, y)\ \to\ (f\ x, g\ y) \\
\mathsf{map}_{+} & : \forall \alpha_1 \alpha_2 \beta_1 \beta_2.(\alpha_1 \to \beta_1) \to (\alpha_2 \to \beta_2) \to (\alpha_1 + \alpha_2 \to \beta_1 + \beta_2) \\
\mathsf{map}_{+} & = \lambda f.\lambda g.\lambda e.\mathsf{case}\ e\ \mathsf{of} \\
& \qquad \mathsf{Inl}\ x\ \to\ \mathsf{Inl}\ (f\ x) \\
& \qquad \mathsf{Inr}\ y\ \to\ \mathsf{Inr}\ (g\ y)
\end{array}
$$

The generic specializer generates the code for the structural representation T° of a data type T by interpreting the structure of T°. For instance,

$$\begin{array}{l}\mathsf{map}_{\mathsf{List}^\circ} \ : \ (\alpha \to \beta) \to \mathsf{List}^\circ\ \alpha \to \mathsf{List}^\circ\ \beta \\ \mathsf{map}_{\mathsf{List}^\circ} = \lambda f.\mathsf{map}_+\ \mathsf{map}_{\mathbb{1}}\ (\mathsf{map}_\times\ f\ (\mathsf{map}_{\mathsf{List}}\ f))\end{array}$$

Note that the structure of $\mathsf{map}_{\mathsf{List}^\circ}$ reflects the structure of List°.

The way the arguments and the result of a generic function are converted from and to the structural representation depends on the base type of the generic function. Embedding-projections are used to devise the automatic conversion. Actually, embedding-projections form a predefined generic function that is used for conversions in all other generic functions (e.g. map) [Hin00a]. The type of this generic function is $\alpha \rightleftarrows \beta$ and the base cases are

$$\begin{array}{ll}
\mathsf{ep}_{\mathbb{1}} & : \mathbb{1} \rightleftarrows \mathbb{1} \\
\mathsf{ep}_{\mathbb{1}} & = \mathsf{EP}\ \mathsf{map}_{\mathbb{1}}\ \mathsf{map}_{\mathbb{1}} \\
\mathsf{ep}_+ & : (\alpha_1 \rightleftarrows \alpha_2) \to (\beta_1 \rightleftarrows \beta_2) \to (\alpha_1 + \beta_1 \rightleftarrows \alpha_2 + \beta_2) \\
\mathsf{ep}_+ & = \lambda a.\lambda b.\mathsf{EP}\ (\mathsf{map}_+\ (\mathsf{to}\ a)\ (\mathsf{to}\ b))\ (\mathsf{map}_+\ (\mathsf{from}\ a)\ (\mathsf{from}\ b)) \\
\mathsf{ep}_\times & : (\alpha_1 \rightleftarrows \alpha_2) \to (\beta_1 \rightleftarrows \beta_2) \to (\alpha_1 \times \beta_1 \rightleftarrows \alpha_2 \times \beta_2) \\
\mathsf{ep}_\times & = \lambda a.\lambda b.\mathsf{EP}\ (\mathsf{map}_\times\ (\mathsf{to}\ a)\ (\mathsf{to}\ b))\ (\mathsf{map}_\times\ (\mathsf{from}\ a)\ (\mathsf{from}\ b)) \\
\mathsf{ep}_\to & : (\alpha_1 \rightleftarrows \alpha_2) \to (\beta_1 \rightleftarrows \beta_2) \to ((\alpha_1 \to \beta_1) \rightleftarrows (\alpha_2 \to \beta_2)) \\
\mathsf{ep}_\to & = \lambda a.\lambda b.\mathsf{EP}\ (\lambda f.\mathsf{to}\ b\ \circ\ f\ \circ\ \mathsf{from}\ a)\ (\lambda f.\mathsf{from}\ b\ \circ\ f\ \circ\ \mathsf{to}\ a) \\
\mathsf{ep}_\rightleftarrows & : (\alpha_1 \rightleftarrows \alpha_2) \to (\beta_1 \rightleftarrows \beta_2) \to ((\alpha_1 \rightleftarrows \beta_1) \rightleftarrows (\alpha_2 \rightleftarrows \beta_2)) \\
\mathsf{ep}_\rightleftarrows & = \lambda a.\lambda b.\mathsf{EP}\ (\lambda e.b\ \bullet\ e\ \bullet\ \mathsf{inv}\ a)\ (\lambda e.\mathsf{inv}\ b\ \bullet\ e\ \bullet\ a)
\end{array}$$

The generic specializer generates the instance of ep specific to a generic function. The generation is performed by interpreting the base (kind-indexed) type of the function. For mapping (with the base type $\mathsf{Map}\ \alpha\ \beta$) we have:

$$\begin{array}{l}\mathsf{ep}_{\mathsf{Map}} \ : \ (\alpha_1 \rightleftarrows \alpha_2) \to (\beta_1 \rightleftarrows \beta_2) \to ((\alpha_1 \to \beta_1) \rightleftarrows (\alpha_2 \to \beta_2)) \\ \mathsf{ep}_{\mathsf{Map}} = \lambda a.\lambda b.\mathsf{ep}_\to\ a\ b\end{array}$$

Now there are all the necessary components to generate the code for a generic function specialized to any data type. In particular, for mapping on lists the generic specializer generates

$$\begin{array}{l}\mathsf{map}_{\mathsf{List}} \ : \ (\alpha \to \beta) \to \mathsf{List}\ \alpha \to \mathsf{List}\ \beta \\ \mathsf{map}_{\mathsf{List}} = \mathsf{from}\ (\mathsf{ep}_{\mathsf{Map}}\ \mathsf{conv}_{\mathsf{List}}\ \mathsf{conv}_{\mathsf{List}}) \circ \mathsf{map}_{\mathsf{List}^\circ}\end{array}$$

This function is much more complicated than its hand-coded counterpart

$$\begin{array}{l}\mathsf{map}_{\mathsf{List}} = \lambda f.\lambda l.\mathsf{case}\ l\ \mathsf{of} \\ \qquad\qquad\qquad \mathsf{Nil}\ \to\ \mathsf{Nil} \\ \qquad\qquad\qquad \mathsf{Cons}\ x\ xs\ \to\ \mathsf{Cons}\ (f\ x)\ (\mathsf{map}_{\mathsf{List}}\ f\ xs)\end{array}$$

The reasons for inefficiency are the intermediate data structures for the structural representation and extensive usage of higher-order functions. In the rest of the paper we show that symbolic evaluation guarantees that the intermediate data structures are not created by the resulting code. The resulting code is comparable to the hand-written code.

3 Language

In the following section we present the syntax and operational semantics of a core functional language. Our language supports essential aspects of functional programming such as pattern matching and higher-order functions.

3.1 Syntax

Definition 1 (Expressions and Functions)

a) *The set of* expressions *is defined by the following syntax. In the definition, x ranges over variables,* $\mathbf{C}$ *over constructors and* $\mathbf{F}$ *over function symbols. Below the notation $\vec{a}$ stands for $(a_1, \ldots, a_k)$.*

$$\begin{aligned} E &::= x \mid \mathbf{C}\vec{E} \mid \mathbf{F} \mid \lambda x.E \mid E\ E' \mid \mathsf{case}\ E\ \mathsf{of}\ P_1 \to E_1 \cdots P_n \to E_n \\ P &::= \mathbf{C}\vec{x} \end{aligned}$$

b) *A* function definition *is an expression of the form $\mathbf{F} = E_{\mathbf{F}}$ with $\mathrm{FV}(E_{\mathbf{F}}) = \emptyset$. With $\mathrm{FV}(E)$ we denote the set of free variables occurring in E.*

The distinction between *applications* (expressions) and *specifications* (functions) is reflected by our language definition. *Expressions* are composed from applications of function symbols and constructors. Constructors have a fixed arity, indicating the number of arguments to which they are applied. Partially applied constructors can be expressed by λ-expressions. A function expression is applied to an argument expression by an (invisible, binary) application operator. Finally, there is a case-construction to indicate pattern matching. *Functions* are simply named expressions (with no free variables).

3.2 Semantics

We will describe the evaluation of expressions in the style of *natural operational semantics*, e.g. see [NN92]. The underlying idea is to specify the result of a computation in a compositional, syntax-driven manner.

In this section we focus on evaluation to *normal form* (i.e. expressions being built up from constructors and λ-expressions only). In section 5, we extend this standard evaluation to so-called *symbolic evaluation*: evaluation of expressions containing free variables.

Definition 2 (Standard Evaluation)
Let E, N be expressions. Then E is said to evaluate to *N (notation $E \Downarrow N$) if $E \Downarrow N$ can be produced in the following derivation system.*

$$\lambda x.E \Downarrow \lambda x.E \quad (\textit{E-}\lambda) \qquad \frac{\vec{E} \Downarrow \vec{N}}{\mathbf{C}\vec{E} \Downarrow \mathbf{C}\vec{N}}\ (\textit{E-cons}) \qquad \frac{\mathbf{F} = E_{\mathbf{F}} \qquad E_{\mathbf{F}} \Downarrow N}{\mathbf{F} \Downarrow N}\ (\textit{E-fun})$$

$$\frac{E \Downarrow \mathbf{C}_i\vec{E} \qquad D_i[\vec{x} := \vec{E}] \Downarrow N}{\mathsf{case}\ E\ \mathsf{of}\ \ldots \mathbf{C}_i\vec{x} \to D_i \ldots \Downarrow N}\ (\textit{E-case}) \qquad \frac{E \Downarrow \lambda x.E'' \qquad E''[x := E'] \Downarrow N}{E\ E' \Downarrow N}\ (\textit{E-app})$$

Here $E[x := E']$ denotes the term that is obtained when x in E is substituted by E'.

Observe that our evaluation does not lead to standard normal forms (expressions without redexes): if such an expression contains λs, there may still be redexes below these λs.

4 Typing

Typing systems in functional languages are used to ensure consistency of function applications: the type of each function argument should match some specific input type. In generic programming types also serve as a basis for specialization. Additionally, we will use typing to predict the constructors that appear in the result of a symbolic computation.

Syntax of types

Types are defined as usual. We use $\forall$-types to express polymorphism.

Definition 3 (Types)
The set of types *is given by the following syntax. Below, α ranges over* type variables*, and* T *over type constructors.*

$$\sigma, \tau ::= \alpha \mid \mathrm{T} \mid \sigma{\to}\tau \mid \sigma\ \tau \mid \forall\alpha.\sigma$$

We will sometimes use $\vec{\sigma}{\to}\tau$ as a shorthand for $\sigma_1{\to}\ldots{\to}\sigma_k{\to}\tau$. The set of free type variables of σ is denoted by $\mathrm{FV}(\sigma)$.

The main mechanism for defining new data types in functional languages is via algebraic types.

Definition 4 (Type environments)

a) *Let $\mathcal{A}$ be an* algebraic type system*, i.e. a collection of* algebraic type definitions*. The type specifications in $\mathcal{A}$ give the types of the algebraic data constructors. Let*
$$\mathrm{T}\ \vec{\alpha} = \cdots \mid \mathbf{C}_i\ \vec{\sigma_i} \mid \cdots$$
be the specification of T *in $\mathcal{A}$. Then we write*
$$\mathcal{A} \vdash \mathbf{C}_i : \forall\vec{\alpha}.\vec{\sigma_i}{\to}\mathrm{T}\ \vec{\alpha}.$$

b) *The function symbols are supplied with a type by a* function type environment *$\mathcal{F}$, containing declarations of the form $\mathbf{F} : \sigma$.*

For the sequel, fix a function type environment $\mathcal{F}$, and an algebraic type system $\mathcal{A}$.

Type derivation

Definition 5 (Type Derivation)
a) The type system deals with typing statements *of the form*

$$B \vdash E : \sigma,$$

where B is a type basis (i.e a finite set of declarations of the form $x : \tau$). Such a statement is valid if it can be produced using the following derivation rules.

$$B, x : \sigma \vdash x : \sigma \quad (\sigma\text{-var}) \qquad \frac{\mathbf{F} : \sigma \in \mathcal{F}}{B \vdash \mathbf{F} : \sigma} \, (\sigma\text{-}\mathcal{F}) \qquad \frac{\mathcal{A} \vdash \mathbf{C} : \sigma}{B \vdash \mathbf{C} : \sigma} \, (\sigma\text{-}\mathcal{A})$$

$$\frac{B \vdash \mathbf{C} : \vec{\tau} \to \sigma \quad B \vdash \vec{E} : \vec{\tau}}{B \vdash \mathbf{C}\vec{E} : \sigma} \, (\sigma\text{-cons})$$

$$\frac{B \vdash E : \tau \quad B \vdash \mathbf{C}_i : \vec{\rho_i} \to \tau \qquad B, \vec{x_i} : \vec{\rho_i} \vdash E_i : \sigma}{B \vdash \textsf{case}\ E\ \textsf{of}\ \cdots \mathbf{C}_i \vec{x_i} \to E_i \cdots : \sigma} \, (\sigma\text{-case})$$

$$\frac{B \vdash E : \tau \to \sigma \quad B \vdash E' : \tau}{B \vdash E\ E' : \sigma} \, (\sigma\text{-app}) \qquad \frac{B, x : \tau \vdash E : \sigma}{B \vdash \lambda x.E : \tau \to \sigma} \, (\sigma\text{-}\lambda)$$

$$\frac{B \vdash E : \sigma \quad \alpha \notin \mathrm{FV}(B)}{B \vdash E : \forall \alpha.\sigma} \, (\sigma\text{-}\forall\text{-intro}) \qquad \frac{B \vdash E : \forall \alpha.\sigma}{B \vdash E : \sigma[\alpha := \tau]} \, (\sigma\text{-}\forall\text{-elim})$$

b) The function type environment $\mathcal{F}$ is type correct *if each function definition is type correct, i.e. for $\mathbf{F}$ with type σ and definition $\mathbf{F} = E_{\mathbf{F}}$ one has $\emptyset \vdash E_{\mathbf{F}} : \sigma$.*

5 Symbolic evaluation

The purpose of symbolic evaluation is to reduce expressions at compile-time, for instance to simplify the generated mapping function for lists (see section 2).

If we want to evaluate expressions containing free variables, evaluation cannot proceed if the value of such a variable is needed. This happens, for instance, if a pattern match on such a free variable takes place. In that case the corresponding case-expression cannot be evaluated fully. The most we can do is to evaluate all alternatives of such a case-expression. Since none of the pattern variables will be bound, the evaluation of these alternatives is likely to get stuck on the occurrences of variables again.

Symbolic evaluation gives rise to a new (extended) notion of normal form, where in addition to constructors and λ-expressions, also variables, cases and higher-order applications can occur. This explains the large number of rules required to define the semantics.

Definition 6 (Symbolic Evaluation) *We adjust definition 2 of evaluation by replacing the E-λ rule, and by adding rules for dealing with new combinations of expressions.*

$$x \Downarrow x \quad (E\text{-}var) \qquad \frac{E \Downarrow N}{\lambda x.E \Downarrow \lambda x.N}\ (E\text{-}\lambda)$$

$$\frac{E \Downarrow \mathsf{case}\ D\ \mathsf{of}\ \cdots P_i \to D_i \cdots \qquad \mathsf{case}\ D_i\ \mathsf{of}\ \cdots Q_j \to E_j \cdots \Downarrow N_i}{\mathsf{case}\ E\ \mathsf{of}\ \cdots Q_j \to E_j \cdots \Downarrow \mathsf{case}\ D\ \mathsf{of}\ \cdots P_i \to N_i}\ (E\text{-}case\text{-}case)$$

$$\frac{E \Downarrow x \qquad E_i \Downarrow N_i}{\mathsf{case}\ E\ \mathsf{of}\ \cdots P_i \to E_i \cdots \Downarrow \mathsf{case}\ x\ \mathsf{of}\ \cdots P_i \to N_i \cdots}\ (E\text{-}case\text{-}var)$$

$$\frac{E \Downarrow E'\ E'' \qquad E_i \Downarrow N_i}{\mathsf{case}\ E\ \mathsf{of}\ \cdots P_i \to E_i \cdots \Downarrow \mathsf{case}\ E'\ E''\ \mathsf{of}\ \cdots P_i \to N_i \cdots}\ (E\text{-}case\text{-}app)$$

$$\frac{E \Downarrow \mathsf{case}\ D\ \mathsf{of}\ \cdots P_i \to D_i \cdots \qquad D_i\ E' \Downarrow N_i}{E\ E' \Downarrow \mathsf{case}\ D\ \mathsf{of}\ \cdots P_i \to N_i \cdots}\ (E\text{-}app\text{-}case)$$

$$\frac{E \Downarrow x \qquad E' \Downarrow N}{E\ E' \Downarrow x\ N}\ (E\text{-}app\text{-}var) \qquad \frac{E \Downarrow D\ D' \qquad E' \Downarrow N}{E\ E' \Downarrow D\ D'\ N}\ (E\text{-}app\text{-}app)$$

Note that the rules (E-case) and (E-app) from definition 2 are responsible for removing constructor-destructor pairs and applications of the lambda-terms. These two correspond to the two sources of inefficiency in the generated programs: intermediate data structures and higher-order functions. The rules (E-case-case) and (E-app-case) above are called code-motion rules [DMP96]: their purpose is to move code to facilitate further transformations. For instance, the (E-case-case) rule pushes the outer case in the alternatives of the inner case in hope that an alternative is a constructor. If so, the (E-case) rule is applicable and the intermediate data are removed. Similarly, (E-app-case) pushes the application arguments in the case alternatives hoping that an alternative is a lambda-term. In this case (E-app) becomes applicable.

Example 7 (Symbolic Evaluation) Part of the derivation tree for the evaluation of the expression $\mathsf{map}_\times\ f_1\ g_1\ (\mathsf{map}_\times\ f_2\ g_2\ p)$ is given below. The function $\mathsf{map}_\times$ is defined in section 2.

$$\dfrac{\begin{array}{l}\mathsf{map}_\times \Downarrow \\ \quad \lambda f.\lambda g.\lambda p.\mathsf{case}\ p\ \mathsf{of} \\ \qquad (x,y) \ \to\ (f\ x, g\ y)\end{array} \qquad \dfrac{\begin{array}{l}\mathsf{map}_\times\ f_2\ g_2\ p \Downarrow \\ \quad \mathsf{case}\ p\ \mathsf{of} \\ \qquad (x',y') \ \to\ (f_2\ x', g_2\ y')\end{array} \qquad \begin{array}{l}\mathsf{case}\ (f_2\ x', g_2\ y')\ \mathsf{of} \\ \quad (x,y) \to (f_1\ x, g_1\ y) \Downarrow \\ \qquad (f_1\ (f_2\ x'), g_1\ (g_2\ y'))\end{array}}{\begin{array}{l}\mathsf{case}\ \mathsf{map}_\times\ f_2\ g_2\ p\ \mathsf{of} \\ \quad (x,y) \to (f_1\ x, g_1\ y) \Downarrow \\ \qquad \mathsf{case}\ p\ \mathsf{of}\ (x',y') \ \to\ (f_1\ (f_2\ x'), g_1\ (g_2\ y'))\end{array}}}{\mathsf{map}_\times\ f_1\ g_1\ (\mathsf{map}_\times\ f_2\ g_2\ p) \Downarrow \mathsf{case}\ p\ \mathsf{of}\ (x',y') \ \to\ (f_1\ (f_2\ x'), g_1\ (g_2\ y'))}$$

The following definition characterizes the results of symbolic evaluation.

Definition 8 (Symbolic Normal Forms) *The set of* symbolic normal forms *(indicated by N_s) is defined by the following syntax.*

$$N_s ::= \mathbf{C}\vec{N_s} \mid \lambda x.N_s \mid N_h \mid \mathsf{case}\ N_h\ \mathsf{of}\ \cdots P_i \to N_s \cdots$$
$$N_h ::= x \mid N_h\ N_s$$

Proposition 9 (Correctness of Symbolic Normal Form)

$$E \Downarrow N \;\Rightarrow\; N \in N_s$$

Proof: By induction on the derivation of $E \Downarrow N$. □

5.1 Symbolic evaluation and typing

In this subsection we will show that the type of an expression (or the type of a function) can be used to determine the constructors that appear (or will appear after reduction) in the symbolic normal form of that expression. Note that this is not trivial because an expression in symbolic normal form might still contain potential redexes that can only be determined and reduced during actual evaluation. Recall that one of the reasons for introducing symbolic evaluation is the elimination of auxiliary data structures introduced by the generic specialization procedure.

The connection between evaluation and typing is usually given by the so-called *subject reduction property* indicating that typing is preserved during reduction.

Proposition 10 (Subject Reduction Property)

$$B \vdash E : \sigma, E \Downarrow N \;\Rightarrow\; B \vdash N : \sigma$$

Proof: By induction on the derivation of $E \Downarrow N$. □

There are two ways to determine constructors that can be created during the evaluation of an expression, namely, (1, directly) by analyzing the expression itself or (2, indirectly) by examining the type of that expression.

In the remainder of this section we will show that (2) includes all the constructors of (1), provided that (1) is determined after the expression is evaluated symbolically. The following definition makes the distinction between the different ways of indicating constructors precise.

Definition 11 (Constructors of normal forms and types)

– *Let N be an expression in symbolic normal form. The set of constructors appearing in N (denoted as $C_N(N)$) is inductively defined as follows.*

$$\begin{array}{ll}
C_N(\mathbf{C}\vec{N}) & = \{\mathbf{C}\} \cup C_N(\vec{N}) \\
C_N(\lambda x.N) & = C_N(N) \\
C_N(x) & = \emptyset \\
C_N(N\ N') & = C_N(N) \cup C_N(N') \\
C_N(\mathsf{case}\ N\ \mathsf{of}\ \cdots P_i \to N_i \cdots) & = C_N(N) \cup (\cup_i C_N(N_i))
\end{array}$$

Here $C_N(\vec{N})$ should be read as $\cup_i C_N(N_i)$.

– *Let σ be a type. The set of constructors in σ (denoted as $C_T(\sigma)$) is inductively defined as follows.*

$$\begin{array}{ll} C_T(\alpha) &= \emptyset \\ C_T(\mathrm{T}) &= \cup_i[\{\mathbf{C}_i\} \cup C_T(\vec{\sigma_i})], \qquad \text{where } \mathrm{T} = \cdots \mid \mathbf{C}_i\vec{\sigma_i} \mid \cdots \\ C_T(\tau{\rightarrow}\sigma) &= C_T(\tau) \cup C_T(\sigma) \\ C_T(\tau\ \sigma) &= C_T(\tau) \cup C_T(\sigma) \\ C_T(\forall\alpha.\sigma) &= C_T(\sigma) \end{array}$$

– *Let B be a basis. By $C_T(B)$ we denote the set $\cup C_T(\sigma)$ for each $x : \sigma \in B$.*

Example 12 For the List type from section 2 and for the Rose tree

$$\textbf{data}\ \mathsf{Rose}\ \alpha = \mathsf{Node}\ \alpha\ (\mathsf{List}\ (\mathsf{Rose}\ \alpha))$$

we have $C_T(\mathsf{List}) = \{\mathsf{Nil}, \mathsf{Cons}\}$ and $C_T(\mathsf{Rose}) = \{\mathsf{Node}, \mathsf{Nil}, \mathsf{Cons}\}$.

As a first step towards a proof of the main result of this section we concentrate on expressions that are already in symbolic normal form. Then their typings give a safe approximation of the constructors that are possibly generated by those expressions. This is stated by the following property. In fact, this result is an extension of the *Canonical Normal Forms Lemma*, e.g. see [Pie02].

Proposition 13 *Let $N \in N_s$. Then*

$$B \vdash N : \sigma\ \Rightarrow\ C_N(N) \subseteq C_T(B) \cup C_T(\sigma).$$

Proof: By induction on the structure of N_s. $\square$

The main result of this section shows that symbolic evaluation is adequate to remove constructors that are not contained in the typing statement of an expression. For traditional reasons we call this the *deforestation property.*

Proposition 14 (Deforestation Property)

$$B \vdash E : \sigma, E \Downarrow N\ \Rightarrow\ C_N(N) \subseteq C_T(B) \cup C_T(\sigma)$$

Proof: By proposition 9, 13, and 10. $\square$

5.2 Optimising Generics

Here we show that, by using symbolic evaluation, one can implement a compiler that for a generic operation yields code as efficient as a dedicated hand coded version of this operation.

The code generated by the generic specialization procedure is type correct [Hin00a]. We use this fact to establish the link between the base type of the generic function and the type of a specialized instance of that generic function.

Proposition 15 *Let g be a generic function of type σ, T a data-type, and let g_{T} be the instance of g on T. Then g_{T} is typeable. Moreover, there are no other type constructors in the type of g_{T} than T itself or those appearing in σ.*

Proof: See [AS03]. $\square$

Now we combine typing of generic functions with the deforestation property leading to the following.

Proposition 16 *Let g be a generic function of type σ, T a data-type, and let g_{T} be the instance of g on T. Suppose $g_{\mathrm{T}} \Downarrow N$. Then for any data type S one has*

$$\mathrm{S} \notin \sigma, \mathrm{T} \;\Rightarrow\; C_T(\mathrm{S}) \cap C_N(N) = \emptyset.$$

Proof: By proposition 14, 10, and 15. $\square$

Recall from section 2 that the intermediate data introduced by the generic specializer are built from the structural representation base types $\{\times, +, \mathbb{1}, \rightleftarrows\}$. It immediately follows from the proposition above that, if neither σ nor T contains a structural representation base type S, then the constructors of S are not a part of the evaluated right-hand side of the instance g_T.

6 Implementation aspects: termination of symbolic evaluation

Until now we have avoided the termination problem of the symbolic evaluation. In general, this termination problem is undecidable, so precautions have to be taken if we want to use the symbolic evaluator at compile-time. It should be clear that non-termination can only occur if some of the involved functions are recursive. In this case such a function might be unfolded infinitely many times (by applying the rule (E-fun)). The property below follows directly form proposition 16.

Corollary 17 (Efficiency of generics) *Non-recursive generic functions can be implemented efficiently. More precisely, symbolic evaluation removes intermediate data structures and functions concerning the structural representation base types.*

The problem arises when we deal with generic instances on recursive data types. Specialization of a generic function to such a type will lead to a recursive function. For instance, the specialization of map to List contains a call to $\mathsf{map}_{\mathsf{List}^\circ}$ which, in turn, calls recursively $\mathsf{map}_{\mathsf{List}}$. We can circumvent this problem by breaking up the definition into a non-recursive part and to reintroduce recursion via the standard fixed point combinator $Y = \lambda f.f(Y f)$. Then we can apply symbolic evaluation to the non-recursive part to obtain an optimized version of our generic function. The standard way to remove recursion is to add an extra parameter to a recursive function, and to replace the call to the function itself by a call to that parameter.

Example 18 (Non-recursive specialization) The specialization of map to List without recursion:

$$\begin{aligned}
\mathsf{map}'_{\mathsf{List}} &= \lambda m.\mathsf{from}\ (\mathsf{ep}_{\rightarrow}\ \mathsf{conv}_{\mathsf{List}}\ \mathsf{conv}_{\mathsf{List}}) \circ \\
&\qquad (\lambda f.\mathsf{map}_{+}\ \mathsf{map}_{\mathbb{1}}\ (\mathsf{map}_{\times}\ f\ (m\ f))) \\
\mathsf{map}_{\mathsf{List}} &= Y \mathsf{map}'_{\mathsf{List}}
\end{aligned}$$

After evaluating $\mathsf{map}'_{\mathsf{List}}$ symbolically we get

$$\begin{aligned}
\mathsf{map}'_{\mathsf{List}} = \lambda m.\lambda f.\lambda x.\ & \mathsf{case}\ x\ \mathsf{of} \\
& \mathsf{Nil} \quad\quad\quad \rightarrow \mathsf{Nil} \\
& \mathsf{Cons}\ y\ ys\ \ \rightarrow \mathsf{Cons}\ (f\ y)\ (m\ f\ ys)
\end{aligned}$$

showing that all intermediate data structures are eliminated.

Suppose the generic instance has type τ. Then the non-recursive variant (with the extra recursion parameter) will have type $\tau \rightarrow \tau$, which obviously has the same set of type constructors as τ.

However, this way of handling recursion will not work for generic functions whose base type contains a recursive data type. Consider for example the monadic mapping function for the list monad mapl with the base type

$$\mathbf{type}\ \ \mathsf{Mapl}\ \alpha\ \beta = \alpha \rightarrow \mathsf{List}\ \beta$$

and the base cases

$$\begin{aligned}
\mathsf{mapl}_{\mathbb{1}}\ &:\ \mathbb{1} \rightarrow \mathsf{List}\ \mathbb{1} \\
\mathsf{mapl}_{\mathbb{1}}\ &= \mathsf{return}\ \mathbb{1} \\
\mathsf{mapl}_{\times}\ &:\ \forall \alpha_1 \alpha_2 \beta_1 \beta_2.(\alpha_1 \rightarrow \mathsf{List}\ \beta_1) \rightarrow (\alpha_2 \rightarrow \mathsf{List}\ \beta_2) \rightarrow \alpha_1 \times \alpha_2 \\
&\qquad\quad \rightarrow \mathsf{List}\ (\beta_1 \times \beta_2) \\
\mathsf{mapl}_{\times}\ &= \lambda f.\lambda g.\lambda p.\mathsf{case}\ p\ \mathsf{of}\ (x, y)\ \ \rightarrow\ \ f\ x \gg= \lambda x'.g\ y \gg= \lambda y'.\mathsf{return}\ (x', y') \\
\mathsf{mapl}_{+}\ &:\ \forall \alpha_1 \alpha_2 \beta_1 \beta_2.(\alpha_1 \rightarrow \mathsf{List}\ \beta_1) \rightarrow (\alpha_2 \rightarrow \mathsf{List}\ \beta_2) \rightarrow \alpha_1 + \alpha_2 \\
&\qquad\quad \rightarrow \mathsf{List}\ (\beta_1 + \beta_2) \\
\mathsf{mapl}_{+}\ &= \lambda f.\lambda g.\lambda e.\mathsf{case}\ e\ \mathsf{of} \\
&\qquad\quad \mathsf{Inl}\ x\ \ \rightarrow\ \ f\ x \gg= \lambda x'.\mathsf{return}\ (\mathsf{Inl}\ x') \\
&\qquad\quad \mathsf{Inr}\ y\ \ \rightarrow\ \ g\ y \gg= \lambda y'.\mathsf{return}\ (\mathsf{Inr}\ y')
\end{aligned}$$

where

$$\begin{aligned}
\mathsf{return} &= \lambda x.\mathsf{Cons}\ x\ \mathsf{Nil} \\
(\gg=) &= \lambda l.\lambda f.\mathsf{flatten}\ (\mathsf{map}\ f\ l)
\end{aligned}$$

are the monadic return and (infix) bind for the list monad. The specialization of mapl to any data type, e.g. Tree, uses the embedding-projection specialized to Mapl (see section 2).

$$\begin{aligned}
\mathsf{mapl}_{\mathsf{Tree}}\ &:\ (\alpha \rightarrow \mathsf{List}\ \beta) \rightarrow \mathsf{Tree}\ \alpha \rightarrow \mathsf{List}\ (\mathsf{Tree}\ \beta) \\
\mathsf{mapl}_{\mathsf{Tree}}\ &= \mathsf{from}\ (\mathsf{ep}_{\mathsf{Mapl}}\ \mathsf{conv}_{\mathsf{Tree}}\ \mathsf{conv}_{\mathsf{Tree}}) \circ \mathsf{mapl}_{\mathsf{Tree}^{\circ}}
\end{aligned}$$

The embedding-projection $\mathsf{ep}_{\mathsf{Mapl}}$

$$\begin{aligned}\mathsf{ep}_{\mathsf{Mapl}} &: (\alpha_1 \rightleftarrows \alpha_2) \to (\beta_1 \rightleftarrows \beta_2) \to ((\alpha_1 \to \mathsf{List}\ \beta_1) \rightleftarrows (\alpha_2 \to \mathsf{List}\ \beta_2))\\ \mathsf{ep}_{\mathsf{Mapl}} &= \lambda a.\lambda b.\mathsf{ep}_{\to}\ a\ (\mathsf{ep}_{\mathsf{List}}\ b)\end{aligned}$$

contains a call to the (recursive) embedding-projection for lists $\mathsf{ep}_{\mathsf{List}}$

$$\begin{aligned}\mathsf{ep}_{\mathsf{List}} &: (\alpha \rightleftarrows \beta) \to (\mathsf{List}\ \alpha \rightleftarrows \mathsf{List}\ \beta)\\ \mathsf{ep}_{\mathsf{List}} &= \mathsf{from}\ (\mathsf{ep}_{\rightleftarrows}\ \mathsf{conv}_{\mathsf{List}}\ \mathsf{conv}_{\mathsf{List}}) \circ \mathsf{ep}_{\mathsf{List}^\circ}\\ \mathsf{ep}_{\mathsf{List}^\circ} &: (\alpha \rightleftarrows \beta) \to (\mathsf{List}^\circ\ \alpha \rightleftarrows \mathsf{List}^\circ\ \beta)\\ \mathsf{ep}_{\mathsf{List}^\circ} &= \lambda f.\mathsf{ep}_{+}\ \mathsf{ep}_{\mathbb{1}}\ (\mathsf{ep}_{\times}\ f\ (\mathsf{ep}_{\mathsf{List}}\ f))\end{aligned}$$

We cannot get rid of this recursion (using the Y-combinator) because it is not possible to replace the call to $\mathsf{ep}_{\mathsf{List}}$ in $\mathsf{ep}_{\mathsf{mapl}}$ by a call to a non-recursive variant of $\mathsf{ep}_{\mathsf{List}}$ and to reintroduce recursion afterwards.

Online non-termination detection

A way to solve the problem of non-termination is to extend symbolic evaluation with a mechanism for so-called *online non-termination detection*. A promising method is based on the notion of *homeomorphic embedding* (*HE*) [Leu98]: a (partial) ordering on expressions used to identify 'infinitely growing expressions' leading to non-terminating evaluation sequences. Clearly, in order to be safe, this technique will sometimes indicate unjustly expressions as dangerous. We have done some experiments with a prototype implementation of a symbolic evaluator extended with termination detection based on HEs. It appeared that in many cases we get the best possible results. However, guaranteeing success when transforming arbitrary generics seems to be difficult. The technique requires careful fine-tuning in order not to pass the border between termination and non-termination. This will be a subject to further research.

In practice, our approach will handle many generic functions as most of them do not contain recursive types in their base type specifications, and hence, do not require recursive embedding-projections. For instance, all generic functions in the generic Clean library (except the monadic mapping) fulfill this requirement.

7 Related Work

The generic programming scheme that we use in the present paper is based on the approach by Hinze[Hin00a]. Derivable type classes of GHC [HP01], Generic Haskell [CHJ+02] and Generic Clean [AP01] are based on this specialization scheme. We believe symbolic evaluation can also be used to improve the code generated by PolyP [JJ97]. The authors of [HP01] show by example that inlining and standard transformation techniques can get rid of the overhead of conversions between the types and their representations. The example presented does not involve embedding-projections and only treats non-recursive conversions from a data type to its generic representation. In contrast, our paper gives a formal treatment of optimization of generics. Moreover, we have run GHC 6.0.1 with the maximum level of optimization (-O2) on derived instances of the generic equality function: the result code was by far not free from the structural representation overhead.

Initially, we have tried to optimize generics by using *deforestation* [Wad88] and *fusion* [Chi94,AGS03]. Deforestation is not very successful because of its demand that functions have to be in *treeless form*. Too many generic functions do not meet this requirement. But even with a more liberal classification of functions we did not reach an optimal result. We have extended the original fusion algorithm with so-called *depth analysis* [CK96], but this does not work because of the *producer classification*: recursive embedding-projections are no proper producers. We also have experimented with alternative producer classifications but without success. Moreover, from a theoretical point of view, the adequacy of these methods is hard to prove. [Wad88] shows that with deforestation a composition of functions can be transformed to a single function without loss of efficiency. But the result we are aiming at is much stronger, namely, all overhead due to the generic conversion should be eliminated.

Our approach based on symbolic evaluation resembles the work that has been done on the field of compiler generation by partial evaluation. E.g., both [ST96] and [Jø92] start with an interpreter for a functional language and use partial evaluation to transform this interpreter into a more or less efficient compiler or optimizer. This appears to be a much more general goal. In our case, we are very specific about the kind of results we want to achieve.

Partial evaluation in combination with typing is used in [DMP96,Fil99,AJ01]. They use a two-level grammar to distinguish static terms from dynamic terms. Static terms are evaluated at compile time, whereas evaluation of dynamic terms is postponed to run time. Simple type systems are used to guide the optimization by classifying terms into static and dynamic. In contrast, in the present work we do not make explicit distinction between static and dynamic terms. Our semantics and type system are more elaborate: they support arbitrary algebraic data types. The type system is used to reason about the result of the optimization rather than to guide the optimization.

8 Conclusions and future work

The main contributions of the present paper are the following:

- We have introduced a symbolic evaluation algorithm and proved that the result of the symbolic evaluation of an expression will not contain data constructors not belonging to the type of that expression.
- We have shown that for a large class of generic functions symbolic evaluation can be used to remove the overhead of generic specialization. This class includes generic functions that do not contain recursive types in their base type.

Problems arise when involved generic function types contain recursive type constructors. These type constructors give rise to recursive embedding projections which can lead to non-termination of symbolic evaluation. We could use fusion to deal with this situation but then we have to be satisfied with a method that sometimes produces less optimal code. It seems to be more promising to extend symbolic evaluation with online termination analysis, most likely based on the homeomorphic embedding [Leu98]. We already did some research in this area but this has not yet led to the desired results.

We plan to study other optimization techniques in application to generic programming, such as program transformation in computational form [TM95]. Generic specialization has to be adopted to generate code in computational form, i.e. it has to yield *hylomorphisms* for recursive types.

Generics are implemented in Clean 2.0. Currently, the fusion algorithm of the Clean compiler is used to optimize the generated instances. As stated above, for many generic functions this algorithm does not yield efficient code. For this reason we plan to use the described technique extended with termination analysis to improve performance of generics.

References

[AGS03] Diederik van Arkel, John van Groningen, and Sjaak Smetsers. Fusion in practice. In Ricardo Peña and Thomas Arts, editors, *The 14th International Workshop on the Implementation of Functional Languages, IFL'02, Selected Papers*, volume 2670 of *LNCS*, pages 51–67. Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Springer, 2003.

[AJ01] Klaus Aehlig and Felix Joachimski. Operational aspects of normalization by evaluation. Submitted to MSCS. Available from http://www.mathematik.uni-muenchen.de/~joachski, 2001.

[AP01] Artem Alimarine and Rinus Plasmijer. A generic programming extension for Clean. In Thomas Arts and Markus Mohnen, editors, *Proceedings of the 13th International Workshop on Implementation of Functional Languages, IFL 2001*, pages 257–278, Stockholm, Sweden, September 2001. Ericsson Computer Science Laboratory.

[AS03] Artem Alimarine and Sjaak Smetsers. Efficient generic functional programming. Technical report, Nijmegen Institute for Computing and Information Sciences, University of Nijmegen, The Netherlands, 2003. to appear.

[Chi94] Wei-Ngan Chin. Safe fusion of functional expressions II: further improvements. *Journal of Functional Programming*, **4**(4):515–555, October 1994.

[CHJ+02] Dave Clarke, Ralf Hinze, Johan Jeuring, Andres Löh, and Jan de Wit. The generic haskell user's guide. Technical report, uu-cs-2002-047, Utrecht University, 2002.

[CK96] Wei-Ngan Chin and Siau-Cheng Khoo. Better consumers for program specializations. *Journal of Functional and Logic Programming*, 1996(4), November 1996.

[DMP96] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does the trick. *ACM Transactions on Programming Languages and Systems*, 18(6):730–751, 1996.

[Fil99] Andrzej Filinski. A semantic account of type-directed partial evaluation. In *Principles and Practice of Declarative Programming*, pages 378–395, 1999.

[Hin99] Ralf Hinze. A generic programming extension for Haskell. In Erik Meijer, editor, *Proceedings of the 3rd Haskell Workshop*. Paris, France, September 1999. The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-1999-28.

[Hin00a] Ralf Hinze. Generic programs and proofs. Habilitationsschrift, Universität Bonn, October 2000.

[Hin00b] Ralf Hinze. Polytypic values possess polykinded types. In Roland Backhouse and J.N. Oliveira, editors, *Proceedings of the Fifth International Conference on Mathematics of Program Construction (MPC 2000)*, volume 1837, pages 2–27, July 2000.

[HP01] Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41.1 of Electronic Notes in Theoretical Computer Science. Elsevier Science, August 2001. The preliminary proceedings appeared as a University of Nottingham technical report.

[JJ97] P. Jansson and J. Jeuring. Polyp - a polytypic programming language extension. In *The 24th ACM Symposium on Principles of Programming Languages, POPL '97*, pages 470–482. ACM Press, 1997.

[Jø92] Jesper Jørgensen. Generating a compiler for a lazy language by partial evaluation, popl '92. In *The 19th ACM Symposium on Principles of Programming Languages*, pages 258–268. Albuquerque, New Mexico, ACM Press, January 1992.

[Leu98] Michael Leuschel. Homeomorphic embedding for online termination. Technical Report DSSE-TR-98-11, Department of Electronics and Computer Science, University of Southampton, UK, October 1998.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley Professional Computing, 1992. ISBN 0 471 92980 8.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002. ISBN 0–262–16209–1.

[ST96] Michael Sperber and Peter Thiemann. Realistic compilation by partial evaluation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 206–214, May 1996.

[TM95] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *Conf. Record 7th ACM SIGPLAN/SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95*, pages 306–313, New York, June 1995. La Jolla, San Diego, CA, USA, ACM Press.

[Wad88] Phil Wadler. Deforestation: transforming programs to eliminate trees. In *Proceedings of the European Symposium on Programming*, number 300 in LNCS, pages 344–358, Berlin, Germany, March 1988. Springer-Verlag.