Efficient Generic Functional Programming

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

June 2, 2004

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 data 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 strutures that should be eliminated. In essence, the output after optimization is similar to hand-written code.
AMS classification (2000): 68N18, 68Q55, 03B15.
CR classification (1998): F.3.2, D.3.1.
Keywords and phrases: symbolic evaluation, partial evaluation, generic functions, polytypic functions, functional programming languages, program transformation, typing, operational semantics.

1 Introduction

The role of generic programming in the development of functional programs is steadily becoming more important. Key point is that a single definition of a generic function is used to automatically generate instances of that function for arbitrarily many different types. These generic functions are defined by induction on a structural representation of types. Typical examples include generic equality, mapping, pretty-printing, and parsing. 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.

Current implementations of generic programming ([AP01, CHJ+02]), generate code which is strikingly slow because the 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 present a compile-time (*symbolic*) evaluation system, and prove that it is capable of reducing the overhead introduced by generic specialization. The emphasis lies on the proof, which uses typing to predict the structure of the result of a symbolic computation: 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. To the best of our knowledge, there is currently no detailed study in the literature of improving performance of generics, in particular of proving completeness of these improvements.

Our approach with respect to generic programming is based on the notions of *type-indexed values* and *kind-indexed types* [Hin00b] as is used in both Generic Clean [AP01] and Generic Haskell [CHJ+02]. 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, which are typically introduced by the generic specialization algorithm.

The rest of the paper is organized as follows. In section 2 we introduce a simple language. Section 3 considers typing aspects for that language. Generics are introduced in section 4. These first three sections are more or less preliminary. Section 5 shows that generic functions are type correct. In section 6, we extend the semantics of our language to evaluation of open expressions, and establish some properties of this so-called symbolic evaluation. Section 7 comprises the main result of the paper: it treats the optimization algorithm of generics, in particular the termination property. Section 8 discusses related work. Section 9 reiterates our conclusions.

2 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. It does not provide sharing, since the implementation of generics does not make use of it.

2.1 Syntax

Definition 2.1 (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. $\vec{E}$ denotes $(E_1, \ldots, E_k)$.*

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

b) A function definition *is an expression of the form*

$$\mathbf{F} = E_{\mathbf{F}}$$

with $\mathrm{FV}(E_{\mathbf{F}}) = \emptyset$. Here, $\mathrm{FV}(E)$ denotes the set of free variables occurring in E.

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 by a (binary) application operator. Finally, there is a case-construction to indicate pattern matching. *Functions* are simply named expressions (with no free variables).

2.2 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 6, we extend this standard evaluation to so-called *symbolic evaluation*: evaluation of expressions containing free variables.

Standard evaluation

Definition 2.2 (Evaluation)

a) The set of normal forms *(NF) is defined by the following syntax.*

$$N \quad ::= \quad \mathbf{C}\vec{N} \mid \lambda x.E$$

b) Let E, V be expressions. Then E is said to evaluate to *V (notation $E \Downarrow V$) if $E \Downarrow V$ can be produced in the following derivation system.*

$$\frac{\vec{E} \Downarrow \vec{V}}{\mathbf{C}\vec{E} \Downarrow \mathbf{C}\vec{V}}\ (E\text{-}cons)$$

$$\lambda x.E \Downarrow \lambda x.E \qquad (E\text{-}\lambda)$$

$$\frac{\mathbf{F} = E_{\mathbf{F}} \qquad E_{\mathbf{F}} \Downarrow V}{\mathbf{F} \Downarrow V}\ (E\text{-}fun)$$

$$\frac{E \Downarrow \mathbf{C}_i\vec{E} \qquad D_i[\vec{x} := \vec{E}] \Downarrow V}{\mathsf{case}\ E\ \mathsf{of}\ \ldots \mathbf{C}_i\vec{x} \to D_i \ldots \Downarrow V}\ (E\text{-}case)$$

$$\frac{E \Downarrow \lambda x.E'' \qquad E''[x := E'] \Downarrow V}{E\ E' \Downarrow V}\ (E\text{-}app)$$

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

Observe that our notion of normal form does not imply that expressions in norm form contain no redexes: if such an expression contains λs, there may still be redexes below these λs. Furthermore, it easy to show that if $E \Downarrow V$ then the expression V is in normal form.

3 Typing

Typing systems in functional languages are used to ensure consistency of 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 kinds and types

The types we use are not restricted to kind $\star$ (E.g. see [Bar92]). A *kind* can be seen as a type of a type. Kinds are built up from the constant $\star$ (indicating the kind of ordinary, first-order types), and the binary constructor $\rightarrow$ (assigned to higher-order types).

Definition 3.1 (Kinds) *The set of* kinds *is given by the following syntax.*

$$\kappa \ ::= \ \star \mid \kappa \rightarrow \kappa'$$

Types are defined as usual. We use $\forall$-types to express polymorphism and a special Λ-construct to allow type variable abstraction. The Λ-types are not essential: they are not derived in our typing system (defined later on); their purpose is merely to facilitate the translation of generic specifications into functions.

Definition 3.2 (Types)

a) The set of types *is given by the following syntax. Below, α ranges over* type variables*, T over type constructors, and κ over kinds.*

$$\sigma, \tau \quad ::= \quad \alpha \mid \mathrm{T} \mid \sigma \rightarrow \tau \mid \sigma \ \tau \mid \forall \alpha : \kappa . \sigma \mid \Lambda \alpha : \kappa . \sigma$$

b) We will sometimes use $\vec{\sigma} \rightarrow \tau$ as a shorthand for $\sigma_1 \rightarrow \ldots \rightarrow \sigma_k \rightarrow \tau$. As usual, $\rightarrow$ associates to the right.

c) 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. In our system, type variables are not restricted to kind $\star$. The kind of a type constructor will depend on the kinds of its arguments, which, in turn, are based on the use of the variables in the type specification of the constructors.

Definition 3.3 (Algebraic Types) *Let $\mathcal{A}$ be an* algebraic type system*, i.e. a collection of* algebraic type definitions*. The type specifications in $\mathcal{A}$ give both the kinds of the type constructors and the types of the algebraic 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*

$$\begin{array}{lll} \mathcal{A} & \vdash & \mathrm{T} : \vec{\kappa} \rightarrow \star \\ \mathcal{A} & \vdash & \mathbf{C}_i : \forall (\vec{\alpha} : \vec{\kappa}) . \vec{\sigma_i} \rightarrow \mathrm{T} \ \vec{\alpha}. \end{array}$$

Kind and type derivation

Before treating type correctness, we introduce the notion of *kind correctness*. This notion is defined by the following derivation system.

Definition 3.4 (Kind Derivation)
Let $\mathcal{A}$ be an algebraic environment.

a) *A* kind basis *is a finite set of declarations of the form $\alpha : \kappa$ concerning distinct type variables.*

b) *The kind derivation system deals with* statements *of the form*

$$K \vdash \sigma : \kappa.$$

Such a statement is valid if it can be produced using the following derivation rules.

$$K, \alpha : \kappa \vdash \alpha : \kappa \qquad (\kappa\text{-var})$$

$$\frac{\mathcal{A} \vdash \mathrm{T} : \kappa}{K \vdash \mathrm{T} : \kappa}\ (\kappa\text{-cons})$$

$$\frac{K \vdash \sigma : \kappa' \to \kappa \quad K \vdash \tau : \kappa'}{K \vdash \sigma\ \tau : \kappa}\ (\kappa\text{-app})$$

$$\frac{K \vdash \sigma : \star \quad K \vdash \tau : \star}{K \vdash \sigma \to \tau : \star}\ (\kappa\text{-}\to)$$

$$\frac{K, \alpha : \kappa' \vdash \sigma : \star}{K \vdash (\forall \alpha : \kappa'.\sigma) : \star}\ (\kappa\text{-}\forall)$$

$$\frac{K, \alpha : \kappa' \vdash \sigma : \kappa}{K \vdash (\Lambda \alpha : \kappa'.\sigma) : \kappa' \to \kappa}\ (\kappa\text{-}\Lambda)$$

c) *$\mathcal{A}$ is* kind correct *if each algebraic type is kind correct, i.e. for each* T *in $\mathcal{A}$ with kind $\vec{\kappa} \to \star$ and definition*

$$\mathrm{T}\ \vec{\alpha} = \cdots \mid \mathbf{C}_i\ \vec{\sigma_i} \mid \cdots$$

one has

$$\vec{\alpha} : \vec{\kappa} \vdash \sigma_{i_j} : \star.$$

Definition 3.5 (Function Type Environment)

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

b) $\mathcal{F}$ *is* kind correct *if all function types are kind correct, i.e. for each* $\mathbf{F} : \sigma$ *in* $\mathcal{F}$ *the type* σ *is kind correct.*

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

Definition 3.6 (Type Derivation)

a) A type basis *is a finite set of declarations of the form* $x : \tau$ *concerning distinct variables.*

b) The type system deals with typing statements *of the form*

$$B \vdash_K E : \sigma,$$

where B *is a type basis, and* K *a kind basis. Such a statement is valid if it can be produced using the following derivation rules (for readability reasons, we have omitted the* K*-subscript of the* $\vdash$*-symbol). Below,* $\mathbf{S}$ *is either a function or a constructor symbol.*

$$B, x : \sigma \vdash x : \sigma \quad (\sigma\text{-var})$$

$$\frac{\mathcal{F}, \mathcal{A} \vdash \mathbf{S} : \sigma}{B \vdash \mathbf{S} : \sigma}\ (\sigma\text{-env}) \qquad \frac{B \vdash \mathbf{C} : \vec{\tau} \rightarrow \sigma \quad B \vdash \vec{E} : \vec{\tau}}{B \vdash \mathbf{C}\vec{E} : \sigma}\ (\sigma\text{-cons})$$

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

$$\frac{B \vdash E : \tau \rightarrow \sigma \quad B \vdash E' : \tau}{B \vdash E\ E' : \sigma}\ (\sigma\text{-app})$$

$$\frac{B, x : \tau \vdash E : \sigma}{B \vdash \lambda x.E : \tau \rightarrow \sigma}\ (\sigma\text{-}\lambda)$$

$$\frac{B \vdash E : \sigma \quad \alpha \notin \mathrm{FV}(B) \quad K \vdash \alpha : \kappa}{B \vdash E : \forall \alpha : \kappa . \sigma}\ (\sigma\text{-}\forall\text{-intro})$$

$$\frac{B \vdash E : \forall \alpha : \kappa . \sigma \quad K \vdash \tau : \kappa}{B \vdash E : \sigma[\alpha := \tau]}\ (\sigma\text{-}\forall\text{-elim})$$

c) *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* $\vdash E_{\mathbf{F}} : \sigma$.

4 Generics

In this section we define generics using the approach of kind-indexed types of Hinze [Hin00b]. First we define how data types are represented (Subsection 4.1). Then we define kind-indexed types and kind-indexing (Subsection 4.2). Then generic functions (Subsection 4.3) and specialization (Subsection 4.4) are treated.

4.1 Structural representation of data types

Algebraic data types are data types built out of sums of products of types. The idea is to represent an algebraic data type using a binary sum type (co-product) and a binary product type [Hin99]. N-ary sums and products are then represented using the binary ones. The following two definitions define the product and the co-product, which are used for structural representation of data types.

Definition 4.1 (Product)

a) *A binary product is a data type*

$$\alpha \times \beta = \alpha \times \beta$$

with projection functions and mapping

$$\begin{array}{lcl} \mathit{Outl} & = & \lambda x.\mathsf{case}\ x\ \mathsf{of}\ l \times r\ \rightarrow\ l \\ \mathit{Outr} & = & \lambda x.\mathsf{case}\ x\ \mathsf{of}\ l \times r\ \rightarrow\ r \\ f\ \otimes\ g & = & \lambda x.\mathsf{case}\ x\ \mathsf{of}\ l \times r\ \rightarrow\ f\ l \times g\ r \end{array}$$

b) *A nullary product (unit) is a data type* $\mathbb{1} = \mathbb{1}$.

c) *An n-ary product is a type*

$$\Pi(\alpha_1, \ldots, \alpha_n) = \begin{cases} \mathbb{1}, & n = 0 \\ \alpha_1, & n = 1 \\ \Pi(\alpha_1, \ldots, \alpha_{n-1}) \times \alpha_n & n > 1 \end{cases}$$

with a constructor

$$\Pi(x_1, \ldots, x_n) = \begin{cases} \mathbb{1}, & n = 0 \\ x_1, & n = 1 \\ \Pi(x_1, \ldots, x_{n-1}) \times x_n & n > 1 \end{cases}$$

and projections

$$\pi_i^n(x) = \begin{cases} x, & n = 1 \\ \mathsf{Outr}\ x, & i = n \\ \mathsf{Outl}\ \pi_i^{n-1}(x) & i < n \end{cases}$$

Definition 4.2 (Co-product)

a) A binary co-product is a data type

$$\alpha + \beta = \mathsf{Inl}\ \alpha \mid \mathsf{Inr}\ \beta$$

with a destructor

$$\begin{array}{lll} l\ \nabla\ r = \lambda x. & \mathsf{case}\ x\ \mathsf{of} & \\ & \mathsf{Inl}\ x & \rightarrow\ l\ x \\ & \mathsf{Inr}\ x & \rightarrow\ r\ x \end{array}$$

and a mapping

$$f\ \oplus\ g = \mathsf{Inl}\ \circ\ f\ \nabla\ \mathsf{Inr}\ \circ\ g$$

b) An n-ary co-product is a type

$$\Sigma(\alpha_1, \ldots, \alpha_n) = \begin{cases} \alpha_1, & n = 1 \\ \Sigma(\alpha_1, \ldots, \alpha_{n-1}) + \alpha_n & n > 1 \end{cases}$$

with injections

$$\iota_i^n(x) = \begin{cases} x, & n = 1 \\ \mathsf{Inr}\ x, & i = n \\ \mathsf{Inl}\ \iota_i^{n-1}(x) & i < n \end{cases}$$

and a destructor

$$\triangledown(f_1, \ldots, f_n) = \begin{cases} f_1, & n = 1 \\ \triangledown(f_1, \ldots, f_{n-1})\ \nabla\ f_n & n > 1 \end{cases}$$

We do not need a nullary co-product (void type) as a data type has at least one alternative.

Definition 4.3 (Structural representation of data types) *Let*

$$\mathrm{T}\ \vec{\alpha} = \cdots \mid \mathbf{C}_i\ \vec{\sigma}_i \mid \cdots$$

be a data type. The structural representation T° *is defined as a synonym type*

$$\mathrm{T}^\circ = \Lambda\vec{\alpha}.\Sigma(\ldots, \Pi(\vec{\sigma}_i), \ldots)$$

Example 4.1 (Structural representation of data types) The types

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

are represented as

$$\begin{array}{lcl} \mathsf{List}^\circ & = & \Lambda\alpha.\ \mathbb{1} + \alpha \times (\mathsf{List}\ \alpha) \\ \mathsf{Tree}^\circ & = & \Lambda\alpha\beta.\ \alpha + \beta \times \mathsf{Tree}\ \alpha\ \beta \times \mathsf{Tree}\ \alpha\ \beta \end{array}$$

As usually, $\times$ and $+$ bind to the left. Note that the representation of a recursive type is not recursive.

To express the isomorphism between two types τ and σ we use embedding-projection pairs that store conversion functions $\tau \to \sigma$ and $\sigma \to \tau$ [HP01].

Definition 4.4 (Embedding-projection pairs)

a) An embedding-projection pair is a pair of functions

$$\alpha \rightleftarrows \beta = \mathit{EP}\ (\alpha \to \beta)\ (\beta \to \alpha)$$

b) Projections:

$$\begin{array}{lcl} \mathit{to} & = & \lambda x.\mathit{case}\ x\ \mathit{of}\ \mathit{EP}\ t\ f\ \to\ t \\ \mathit{from} & = & \lambda x.\mathit{case}\ x\ \mathit{of}\ \mathit{EP}\ t\ f\ \to\ f \end{array}$$

c) Composition:

$$a\ \bullet\ b = \mathit{EP}(\mathit{to}\ a\ \circ\ \mathit{to}\ b)(\mathit{from}\ b\ \circ\ \mathit{from}\ a)$$

d) Inversion:

$$\mathit{inv} = \lambda x.EP\ (\mathit{from}\ x)\ (\mathit{to}\ x)$$

The isomorphism between a data type $\mathrm{T}\ \vec{\alpha}$ and its representation $\mathrm{T}^\circ\ \vec{\alpha}$ is also expressed as a pair of conversion functions $\mathrm{T}\ \vec{\alpha} \rightleftarrows \mathrm{T}^\circ\ \vec{\alpha}$.

Definition 4.5 (Conversion) *Let* $\mathrm{T}\ \vec{\alpha} = \cdots \mid \mathbf{C}_i\ \vec{\sigma}_i \mid \cdots$ *be a data type and* T° *its structural representation. The conversion between the two is an embedding-projection pair* $\mathit{conv}_\mathrm{T} = \mathit{EP}\ \mathit{to}_\mathrm{T}\ \mathit{from}_\mathrm{T}$ *where*

$$\begin{array}{lcl} \mathit{to}_\mathrm{T} & = & \lambda x.\mathit{case}\ x\ \mathit{of}\ \cdots C_i\ \vec{x}\ \to\ \iota_i^k(\Pi(\vec{x}))\cdots \\ \mathit{from}_\mathrm{T} & = & \lambda x.\bigtriangledown(\cdots, \lambda x.C_i\ \pi_1^{m_i}(x)\ \ldots\ \pi_{m_i}^{m_i}(x), \cdots) \end{array}$$

Here k *is the number of constructors in* T, $1 \le i \le k$, *and* $m_i = |\vec{\sigma}_i|$.

Example 4.2 (Conversion) The conversion for lists is

$$
\begin{array}{lcl}
\mathsf{conv}_{\mathsf{List}} & : & \mathsf{List}\ \alpha \rightleftarrows \mathsf{List}^{\circ}\ \alpha \\
\mathsf{conv}_{\mathsf{List}} & = & \mathsf{EP}\ \mathsf{to}_{\mathsf{List}}\ \mathsf{from}_{\mathsf{List}} \\
\text{where} & & \\
\quad \mathsf{to}_{\mathsf{List}} & = & \lambda l.\mathsf{case}\ l\ \mathsf{of} \\
 & & \quad \mathsf{Nil} \ \rightarrow\ \mathsf{Inl}\ \mathbb{1} \\
 & & \quad \mathsf{Cons}\ x\ xs \ \rightarrow\ \mathsf{Inr}\ (x \times xs) \\
\quad \mathsf{from}_{\mathsf{List}} & = & \lambda l.\mathsf{case}\ l\ \mathsf{of} \\
 & & \quad \mathsf{Inl}\ u \ \rightarrow\ \mathsf{case}\ u\ \mathsf{of}\ \mathbb{1} \ \rightarrow\ \mathsf{Nil} \\
 & & \quad \mathsf{Inr}\ p \ \rightarrow\ \mathsf{case}\ p\ \mathsf{of}\ (x \times xs) \ \rightarrow\ \mathsf{Cons}\ x\ xs
\end{array}
$$

4.2 Kind-indexed types

Many functions can be defined not only for various types of the same kind but also for types of various kinds. For instance, there is a mapping function for functors, bi-functors and functors of other kinds. These mapping functions have different types depending on the kinds of the functors. Our goal is to handle all such cases by a single generic definition. Kind-indexed types facilitate this goal. They are schemes that compute a type from a kind. [Hin00b].

Definition 4.6 (Kind-indexed type) *A kind indexed type is generated by the following grammar*

$$\gamma = \sigma \mid \Gamma\vec{\alpha}.\gamma$$

Variables bound by Γ *must be of kind* $\star$*. These variables are called kind-indexed variables.*

Definition 4.7 (Kind-indexing) *Let* $\gamma = \Gamma\alpha_1\ \ldots\ \alpha_n.\sigma$ *be a kind-indexed type. Kind indexing is the following operation*

$$
\begin{array}{lcl}
\gamma\langle\star\rangle & = & \Lambda\alpha_1 : \star\ \ldots\ \alpha_n : \star.\sigma \\
\gamma\langle\kappa \rightarrow \kappa'\rangle & = & \Lambda\alpha_1 : \kappa \rightarrow \kappa'\ \ldots\ \alpha_n : \kappa \rightarrow \kappa'.\forall\beta_1 : \kappa\ \ldots\ \beta_n : \kappa. \\
 & & \quad \gamma\langle\kappa\rangle\ \beta_1\ \ldots\ \beta_n\ \rightarrow\ \gamma\langle\kappa'\rangle\ (\alpha_1\ \beta_1)\ \ldots\ (\alpha_n\ \beta_n)
\end{array}
$$

Example 4.3 (Kind-indexing) Consider the following kind-indexed type for the mapping function $\mathsf{Map} = \Gamma\alpha\beta.\alpha \rightarrow \beta$. It can be used to generate the type of mapping for a functor of any kind.

$$
\begin{array}{lcl}
\mathsf{Map}\langle\star\rangle & = & \Lambda\alpha\beta.\alpha \to \beta \\
\mathsf{Map}\langle\star \to \star\rangle & = & \Lambda\alpha\beta.\forall\alpha_1\beta_1.(\alpha_1 \to \beta_1) \\
 & & \to (\alpha\ \alpha_1 \to \beta\ \beta_1) \\
\mathsf{Map}\langle\star \to \star \to \star\rangle & = & \Lambda\alpha\beta.\forall\alpha_1\beta_1.(\alpha_1 \to \beta_1) \\
 & & \to (\forall\alpha_2\beta_2.(\alpha_2 \to \beta_2) \\
 & & \quad \to (\alpha\ \alpha_1\ \alpha_2 \to \beta\ \beta_1\ \beta_2)) \\
 & = & \Lambda\alpha\beta.\forall\alpha_1\beta_1\alpha_2\beta_2. \\
 & & (\alpha_1 \to \beta_1) \to (\alpha_2 \to \beta_2) \\
 & & \to (\alpha\ \alpha_1\ \alpha_2 \to \beta\ \beta_1\ \beta_2)) \\
\mathsf{Map}\langle(\star \to \star) \to \star\rangle & = & \Lambda\alpha\beta.\forall\alpha_1\beta_1. \\
 & & (\forall\alpha_{11}\beta_{11}.(\alpha_{11} \to \beta_{11}) \\
 & & \quad \to (\alpha_1\ \alpha_{11} \to \beta_1\ \beta_{11})) \\
 & & \to (\alpha\ \alpha_1 \to \beta\ \beta_1)
\end{array}
$$

4.3 Generic functions

Informally, a generic function is a template used to generate functions that operate on concrete data types. This template is provided by the programmer and consists of a kind-indexed type (see subsection 4.1) and a set of base case definitions on basic types and the generic structure of types (Subsection 4.2). The kind-indexed type is used to generate types of all generated functions.

Definition 4.8 (Generic function)

a) The instance environment $\mathcal{I}$ is a mapping from type constructors $\mathrm{T} \in \mathcal{T}$ to function symbols $\mathbf{F}$. We treat the arrow type $\to$ as a type constructor of kind $\star \to \star \to \star$, i.e. $(\to) \in \mathcal{T}$.

b) A generic function $\mathcal{G}$ is a triple $\langle\gamma, \mathcal{I}, \mathcal{C}\rangle$ where γ is a kind-indexed type, $\mathcal{I}$ is an instance environment and $\mathcal{C} \subset \mathcal{T}$ is a set of base cases, i.e type constructors, for which the instances are provided by the user.

Example 4.4 (Generic mapping) Generic mapping

$$\mathsf{map} = \langle \mathsf{Map}, \mathcal{I}_{\mathsf{map}}, \{\mathbb{1}, +, \times\}\rangle$$

is a generic function with the following base cases in $\mathcal{I}_{\mathsf{map}}$:

$$\begin{array}{lcl}
\mathsf{map}_{\mathbb{1}} & : & \mathbb{1} \to \mathbb{1} \\
\mathsf{map}_{\mathbb{1}} & = & \mathsf{id} \\
\mathsf{map}_{\times} & : & \forall\alpha_1\alpha_2\beta_1\beta_2.(\alpha_1 \to \beta_1) \to (\alpha_2 \to \beta_2) \\
 & & \to (\alpha_1 \times \alpha_2 \to \beta_1 \times \beta_2) \\
\mathsf{map}_{\times} & = & \lambda fg.f \otimes g \\
\mathsf{map}_{+} & : & \forall\alpha_1\alpha_2\beta_1\beta_2.(\alpha_1 \to \beta_1) \to (\alpha_2 \to \beta_2) \\
 & & \to (\alpha_1 + \alpha_2 \to \beta_1 + \beta_2) \\
\mathsf{map}_{+} & = & \lambda fg.f \oplus g
\end{array}$$

4.4 Specialization of generic functions

This subsection defines how a generic function is specialized to a concrete function for a given type. Specialization is defined in two steps: first we introduce specialization by induction on the structure of types, and then use this notion to define specialization to algebraic data types.

The following operation is used to take a fresh instance of a type where each free variable is substituted by a variable with the same name and added index.

Definition 4.9 (Fresh indexed type) *Let τ be a type. The operation of fresh indexed type $(\tau)_i$ is defined as follows*

$$(\tau)_i = \tau[\alpha := \alpha_i], \forall\alpha \in \mathrm{FV}(\tau)$$

Example 4.5 (Fresh indexed type) $(\alpha)_i = \alpha_i$ and $(\Lambda\alpha.\alpha\ \beta)_i = \Lambda\alpha.\alpha\ \beta_i$

Specialization of a generic operation to a type yields an expression that performs the operation on the value of the type.

Definition 4.10 (Specialization)

a) *The specialization environment $\mathcal{E}$ is a finite set of declarations of the form $\alpha : \kappa := x$ concerning distinct type variables α and distinct term variables x.*

b) *The specialization $\mathcal{G}\langle\mathcal{E}, \tau\rangle$ of a generic function $\mathcal{G}$ to a monomorphic type τ (i.e. a type without $\forall$s) is defined inductively on the structure of types:*

$$\begin{array}{lcl}
\mathcal{G}\langle\mathcal{E}, \alpha\rangle & = & \mathcal{E}(\alpha) \\
\mathcal{G}\langle\mathcal{E}, \mathrm{T}\rangle & = & \mathcal{I}(\mathrm{T}) \\
\mathcal{G}\langle\mathcal{E}, \sigma\ \rho\rangle & = & \mathcal{G}\langle\mathcal{E}, \sigma\rangle\ \mathcal{G}\langle\mathcal{E}, \rho\rangle \\
\mathcal{G}\langle\mathcal{E}, \sigma \to \rho\rangle & = & \mathcal{I}(\to)\ \mathcal{G}\langle\mathcal{E}, \sigma\rangle\ \mathcal{G}\langle\mathcal{E}, \rho\rangle \\
\mathcal{G}\langle\mathcal{E}, \Lambda\alpha.\sigma\rangle & = & \lambda x.\mathcal{G}\langle\mathcal{E}[\alpha := x], \sigma\rangle
\end{array}$$

c) We write $\mathcal{G}\langle\sigma\rangle$ as an abbreviation of $\mathcal{G}\langle\emptyset,\sigma\rangle$.

Example 4.6 (Specialization) Specialization of map to $\mathsf{List}^\circ = \Lambda\alpha.\ \mathbb{1} + \alpha \times (\mathsf{List}\ \alpha)$:

$$\mathsf{map}_{\mathsf{List}^\circ} = \lambda x.\mathsf{map}_+\ \mathsf{map}_{\mathbb{1}}\ (\mathsf{map}_\times\ x\ (\mathsf{map}_{\mathsf{List}}\ x))$$

So far we defined how to specialize a generic function for the representation type T°. However, we have to generate an instance for the real type T. Before looking at it we introduce a generic function for embedding-projection pairs, which is used to convert generated *functions* on structural types $\mathrm{T}^\circ\ \vec{\alpha}$ into the *functions* on the types $\mathrm{T}\ \vec{\alpha}$. This function is similar to generic map introduced in the examples above except that it operates on pairs of arrows instead of a single arrow.

Definition 4.11 (Generic embedding-projections) *The generic function*

$$\mathcal{EP} = \langle\Gamma\alpha\beta.\alpha \rightleftarrows \beta, \mathcal{I}_{ep}, \{\mathbb{1}, +, \times, \rightarrow, \rightleftarrows\}\rangle$$

where the instance environment $\mathcal{I}_{ep}$ contains the following base cases:

$$\begin{array}{lcll}
ep_{\mathbb{1}} & = & EP & \mathbf{id}\ \mathbf{id} \\
ep_+ & = & \lambda a.\lambda b.EP & (to\ a\ \oplus\ to\ b)\ (from\ a\ \oplus\ from\ b) \\
ep_\times & = & \lambda a.\lambda b.EP & (to\ a\ \otimes\ to\ b)\ (from\ a\ \otimes\ from\ b) \\
ep_\rightarrow & = & \lambda a.\lambda b.EP & (\lambda f.to\ b\ \circ\ f\ \circ\ from\ a) \\
 & & & (\lambda f.from\ b\ \circ\ f\ \circ\ to\ a) \\
ep_\rightleftarrows & = & \lambda a.\lambda b.EP & (\lambda e.b\ \bullet\ e\ \bullet\ inv\ a) \\
 & & & (\lambda e.inv\ b\ \bullet\ e\ \bullet\ a)
\end{array}$$

Unlike in map, here we have additional base cases on arrows $\mathsf{ep}_\rightarrow$ and embedding-projection pairs $\mathsf{ep}_\rightleftarrows$. The case for arrows $\mathsf{ep}_\rightarrow$ is needed to convert *functions* on $\mathrm{T}^\circ\ \vec{\alpha}$ into *functions* on $\mathrm{T}\ \vec{\alpha}$. The base case for embedding-projection pairs $\mathsf{ep}_\rightleftarrows$ is needed to prevent non-termination of the specialization of $\mathcal{EP}$, which requires specialization of $\mathcal{EP}$ to $\rightleftarrows$.

Finally, we define how to generate a function for a data type T using specialization for T°.

Definition 4.12 (Specialization to a data type) *Let $\mathcal{G} = \langle\Gamma\vec{\alpha}.\sigma, \mathcal{I}, \mathcal{C}\rangle$ be a generic function and $\mathrm{T}\ \vec{\beta}$ be a data type such that $\mathrm{T} \notin \mathcal{C} \cup (\rightarrow)$. Let $n = |\vec{\alpha}|$ and $m = |\vec{\beta}|$. The instance $\mathbf{G}_T = \mathcal{I}(\mathrm{T})$ is generated as follows:*

$$\mathbf{G}_T = from\ (\mathcal{EP}\langle\Lambda\vec{\alpha}.\sigma\rangle\ \underbrace{conv_{\mathrm{T}}\ \ldots\ conv_{\mathrm{T}}}_{n\ \text{times}})\ \circ_m\ \mathcal{G}\langle\mathrm{T}^\circ\rangle$$

where $f \circ_k g = \lambda x_1 \ldots \lambda x_k.f\ (g\ x_1\ \ldots\ x_k)$

Example 4.7 (Specialization to a data type) Instance of mapping for List:

$$\mathsf{map}_{\mathsf{List}} = \mathsf{from}\ (\mathsf{ep}_{\rightarrow}\ \mathsf{conv}_{\mathsf{List}}\ \mathsf{conv}_{\mathsf{List}}) \circ \mathsf{map}_{\mathsf{List}}{}^{\circ}$$

This generated mapping function is equivalent to the standard mapping function for lists.

5 Typing of Generics

In this section we show that the code generated by the generic specialization is well-typed. The proofs are based on Hinze [Hin00a].

Lemma 5.1 (Typing conversion) $conv_{\mathrm{T}} : \mathrm{T}\ \vec{\alpha} \rightleftarrows \mathrm{T}^{\circ}\ \vec{\alpha}$

Definition 5.1 (Type correct generic function)

a) The instance environment $\mathcal{I}$ is type correct iff $\forall t \in \mathcal{T}.\mathcal{I}(t) : \gamma\langle\kappa\rangle\ t\ \ldots\ t$ where κ is the kind of t.

b) The generic function is type correct iff its instance environment is type correct.

Definition 5.2 (Type correct specialization environment)

a) The specialization environment $\mathcal{E}$ for the generic function $\mathcal{G} = \langle\gamma, \mathcal{I}, \mathcal{C}\rangle$ is type correct iff $\forall\alpha : \kappa \in \mathcal{E}.\mathcal{E}(\alpha) : \gamma\langle\kappa\rangle\ \alpha_1\ \ldots\ \alpha_n$.

b) The specialization basis $|\mathcal{E}|$ is a set of entries of the form $x : \gamma\langle\kappa\rangle\ \alpha_1\ \ldots\ \alpha_n$.

The following key property states that the specialization process yields a well-typed expression of the desired type.

Proposition 5.2 (Typing specialization) *Let*

a) $\mathcal{G}$ be a type-correct generic function of type γ,

b) $\tau : \kappa$ be a monomorphic type,

c) $\mathcal{E}$ be a type-correct environment for $\mathcal{G}$ such that $\mathrm{FV}(\tau) \subseteq \mathcal{E}$.

Then $|\mathcal{E}| \vdash \mathcal{G}\langle\mathcal{E}, \tau\rangle : \gamma\langle\kappa\rangle\ (\tau)_1\ \ldots\ (\tau)_n$.

Proof: By induction on the structure of monomorphic types.

- Case $\tau \equiv \alpha : k$. By type-correctness of $\mathcal{E}$.
- Case $\tau \equiv \mathrm{T} : k$. By type-correctness of the generic function.
- Case $\tau \equiv \sigma\ \rho$. Let $\sigma : \kappa \to \kappa'$ and $\rho : \kappa$. By IH for σ:

$$|\mathcal{E}| \vdash \mathcal{G}\langle \mathcal{E}, \sigma \rangle : \gamma\langle \kappa \to \kappa' \rangle\ (\sigma)_1\ \ldots\ (\sigma)_n$$

  By the definition of kind-indexing:

$$\begin{aligned} &\gamma\langle \kappa \to \kappa' \rangle\ (\sigma)_1\ \ldots\ (\sigma)_n \\ &\quad = \forall \beta_1 \ldots \beta_n . \gamma\langle \kappa \rangle\ \beta_1\ \ldots\ \beta_n \to \gamma\langle \kappa' \rangle\ (\sigma\ \beta)_1\ \ldots\ (\sigma\ \beta)_n \\ &\quad = \forall \beta_1 \ldots \beta_n . \gamma\langle \kappa \rangle\ \beta_1\ \ldots\ \beta_n \to \gamma\langle \kappa' \rangle\ ((\sigma)_1\ \beta_1)\ \ldots\ ((\sigma)_n\ \beta_n) \end{aligned}$$

  By IH for ρ:

$$|\mathcal{E}| \vdash \mathcal{G}\langle \mathcal{E}, \rho \rangle : \gamma\langle \kappa \rangle\ (\rho)_1\ \ldots\ (\rho)_n$$

  By (σ-$\forall$-elim) and (σ-app):

$$\begin{aligned} |\mathcal{E}| \vdash \mathcal{G}\langle \mathcal{E}, \sigma \rangle\ \mathcal{G}\langle \mathcal{E}, \rho \rangle &\ :\ \gamma\langle \kappa' \rangle\ (\sigma)_1\ (\rho)_1\ \ldots\ (\sigma)_n\ (\rho)_n \\ |\mathcal{E}| \vdash \mathcal{G}\langle \mathcal{E}, \sigma \rangle\ \mathcal{G}\langle \mathcal{E}, \rho \rangle &\ :\ \gamma\langle \kappa' \rangle\ (\sigma\ \rho)_1\ \ldots\ (\sigma\ \rho)_n \end{aligned}$$

- Case $\tau \equiv \sigma \to \rho$. Similar to the previous case.
- Case $\tau \equiv \Lambda\alpha.\sigma$. Let $\alpha : \kappa$ and $\sigma : \kappa'$. Let $\mathcal{E}' = \mathcal{E}, \alpha := x$ where $x : \gamma\langle \kappa \rangle\ \alpha_1\ \ldots\ \alpha_n$. Let $E = \mathcal{G}\langle \mathcal{E}', \sigma \rangle$
  By IH:

$$|\mathcal{E}'| \vdash E : \gamma\langle \kappa' \rangle\ (\sigma)_1\ \ldots\ (\sigma)_n$$

  By (σ-Λ-intro) and (σ-app):

$$|\mathcal{E}'| \vdash E : \gamma\langle \kappa' \rangle\ ((\Lambda\alpha.(\sigma)_1)\ \alpha)\ \ldots\ ((\Lambda\alpha.(\sigma)_n)\ \alpha)$$

  By the definition of type indexing

$$|\mathcal{E}'| \vdash E : \gamma\langle \kappa' \rangle\ ((\Lambda\alpha.\sigma)\ \alpha)_1\ \ldots\ ((\Lambda\alpha.\sigma)\ \alpha)_n$$

  By (σ-Λ-intro):

$$\begin{aligned} |\mathcal{E}| \vdash \lambda x.E\ &:\ \gamma\langle \kappa \rangle\ \alpha_1\ \ldots\ \alpha_n \\ &\quad \to \gamma\langle \kappa' \rangle\ ((\Lambda\alpha.\sigma)\ \alpha)_1\ \ldots\ ((\Lambda\alpha.\sigma)\ \alpha)_n \end{aligned}$$

By (σ-$\forall$-intro):

$$\begin{array}{rcl} |\mathcal{E}| \vdash \lambda x.E & : & \forall\alpha_1\ \ldots\ \alpha_n.\gamma\langle\kappa\rangle\ \alpha_1\ \ldots\ \alpha_n \\ & & \rightarrow \gamma\langle\kappa'\rangle\ ((\Lambda\alpha.\sigma)\ \alpha)_1\ \ldots\ ((\Lambda\alpha.\sigma)\ \alpha)_n \end{array}$$

By kind-indexing 4.7:

$$|\mathcal{E}| \vdash \lambda x.E : \gamma\langle\kappa \rightarrow \kappa'\rangle\ (\Lambda\alpha.\sigma)_1\ \ldots\ (\Lambda\alpha.\sigma)_n$$

The following property states that a function generated for a data type is well-typed.

Proposition 5.3 (Typing generated instances) *Let $\mathcal{G} = \langle\gamma, \mathcal{I}, \mathcal{C}\rangle$ be a generic function and $\mathrm{T} : \kappa$ be a data type. Then $\mathbf{G}_T = \mathcal{I}(\mathrm{T}) : \gamma\langle\kappa\rangle\ \mathrm{T}\ \ldots\ \mathrm{T}$.*

Proof: Let $\gamma = \Gamma\vec{\alpha}.\sigma$ and $\gamma_\star = \gamma\langle\star\rangle = \Lambda\vec{\alpha}.\sigma$. By the definition of kind indexed types all $\alpha : \star$. Thus, $\gamma_\star : \kappa'$ where $\underbrace{\star \rightarrow \ldots \rightarrow \star}_{n \text{ times}} \rightarrow \star$ and $n = |\vec{\alpha}|$.

By typing specialization 5.2

$$\mathcal{EP}\langle\gamma_\star\rangle : (\Gamma\beta\beta'.\beta \rightleftarrows \beta')\langle\kappa'\rangle\ (\gamma_\star)_1\ (\gamma_\star)_2$$

Since $\gamma_\star$ is a closed type

$$\mathcal{EP}\langle\gamma_\star\rangle : (\Gamma\beta\beta'.\beta \rightleftarrows \beta')\langle\kappa'\rangle\ \gamma_\star\ \gamma_\star$$

By the definition of kind-indexing 4.7

$$\begin{array}{rcl} \mathcal{EP}\langle\gamma_\star\rangle & : & \forall\beta_1\beta'_1 \ldots \beta_n\beta'_n. \\ & & (\beta_1 \rightleftarrows \beta'_1) \rightarrow \ldots \rightarrow (\beta_n \rightleftarrows \beta'_n) \\ & & \rightarrow (\gamma_\star\ \beta_1\ \ldots\ \beta_n \rightleftarrows \gamma_\star\ \beta'_1\ \ldots\ \beta'_n) \end{array}$$

By definition $\mathsf{conv}_\mathrm{T} : \mathrm{T}\ \alpha_1\ \ldots\ \alpha_m \rightleftarrows \mathrm{T}^\circ\ \alpha_1\ \ldots\ \alpha_m$ where m is the arity of type T. Thus

$$\begin{array}{l} \mathcal{EP}\langle\gamma_\star\rangle\ \mathsf{conv}_\mathrm{T}\ \ldots\ \mathsf{conv}_\mathrm{T} : \\ \quad \gamma_\star\ (T\ \alpha_{11}\ \ldots\ \alpha_{1m})\ \ldots\ (T\ \alpha_{n1}\ \ldots\ \alpha_{nm}) \\ \qquad \rightleftarrows \gamma_\star\ (T^\circ\ \alpha_{11}\ \ldots\ \alpha_{1m})\ \ldots\ (T^\circ\ \alpha_{n1}\ \ldots\ \alpha_{nm}) \end{array}$$

Then

$$\begin{array}{l} \mathsf{from}\ (\mathcal{EP}\langle\gamma_\star\rangle\ \mathsf{conv}_\mathrm{T}\ \ldots\ \mathsf{conv}_\mathrm{T}) : \\ \quad \gamma_\star\ (\mathrm{T}^\circ\ \vec{\alpha})_1\ \ldots\ (\mathrm{T}^\circ\ \vec{\alpha})_n \rightarrow \gamma_\star\ (\mathrm{T}\ \vec{\alpha})_1\ \ldots\ (\mathrm{T}\ \vec{\alpha})_n \end{array}$$

By typing specialization 5.2

$$\mathcal{G}\langle\mathrm{T}^\circ\rangle : \gamma\langle\kappa\rangle\ \mathrm{T}^\circ\ \ldots\ \mathrm{T}^\circ$$

Since $\mathrm{T}^\circ = \Lambda\alpha_1 \ldots \alpha_m.\tau$, $k = k_1 \to \ldots \to k_m \to \star$. By kind indexing 4.7

$$\begin{array}{l}\gamma\langle\kappa\rangle\ \mathrm{T}^\circ\ \ldots\ \mathrm{T}^\circ = \forall\alpha_{11} \ldots \alpha_{nm}.\\ \quad \gamma\langle\kappa_1\rangle\ \alpha_{11}\ \ldots\ \alpha_{n1} \to \ldots \to \gamma\langle\kappa_m\rangle\ \alpha_{1m}\ \ldots\ \alpha_{nm}\\ \quad\quad \to \gamma\langle\star\rangle\ (\mathrm{T}^\circ\ \alpha_{11}\ \ldots\ \alpha_{1m})\ \ldots\ (\mathrm{T}^\circ\ \alpha_{n1}\ \ldots\ \alpha_{nm})\end{array}$$

By typing for composition the type of the right-hand-side of the generated function is

$$\begin{array}{l}\mathbf{G}_T : \forall\alpha_{11} \ldots \alpha_{nm}.\\ \quad \gamma\langle\kappa_1\rangle\ \alpha_{11}\ \ldots\ \alpha_{n1} \to \ldots \to \gamma\langle\kappa_m\rangle\ \alpha_{1m}\ \ldots\ \alpha_{nm}\\ \quad\quad \to \gamma\langle\star\rangle\ (\mathrm{T}\ \alpha_{11}\ \ldots\ \alpha_{1m})\ \ldots\ (\mathrm{T}\ \alpha_{n1}\ \ldots\ \alpha_{nm}),\end{array}$$

which by kind indexing is $\gamma\langle\kappa\rangle\ \mathrm{T}\ \ldots\ \mathrm{T}$

6 Symbolic evaluation

The purpose of symbolic evaluation is to reduce expressions at compile-time. For instance, we want to simplify the generated mapping function for lists (see example 4.7).

If we want to evaluate expressions containing free variables, evaluation cannot proceed if the result 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 an 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. Further in this paper we are only interested in symbolic evaluation, so there is no need to introduce a different notation to distinguish it from standard evaluation.

Definition 6.1 (Symbolic Evaluation) *We adjust definition 2.2 of evaluation by replacing the E-λ rule, and by adding rules for dealing with new*

combinations of expressions.

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

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

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

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

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

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

The following definition characterizes the results of symbolic evaluation.

Definition 6.2 (Symbolic Normal Forms) *The set of* symbolic normal forms *(indicated by N_s) is defined by the following syntax.*

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

Next, we prove that our characterization is correct.

Proposition 6.1

$$E \Downarrow V \ \Rightarrow\ V \in N_s$$

Proof: By induction on the derivation of $E \Downarrow V$. We show a characteristic example case.

- Rule (E-case-case). Suppose

  $$\mathsf{case}\ E\ \mathsf{of}\ \cdots Q_j \to E_j \cdots \Downarrow \mathsf{case}\ D\ \mathsf{of}\ \cdots P_i \to V_i \cdots$$

  since

  $$E \Downarrow \mathsf{case}\ D\ \mathsf{of}\ \cdots P_i \to D_i \cdots$$

and

$$\mathsf{case}\ D_i\ \mathsf{of}\ \cdots Q_j \to E_j \cdots \Downarrow V_i.$$

By IH both $\mathsf{case}\ D\ \mathsf{of}\ \cdots P_i \to D_i \cdots \in N_s$ and $V_i \in N_s$. So

$$\mathsf{case}\ D\ \mathsf{of}\ \cdots P_i \to V_i \cdots \in N_s. \qquad \square$$

6.1 Symbolic evaluation and typing

In the remainder of this section 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 might still contain potential redexes that can only be determined and reduced during actual evaluation. Recall that the motivation for introducing symbolic evaluation is to use it as a kind of partial evaluator in order to get rid of auxiliary data structures that are introduced if one uses a straightforward implementation of generic functions. Of course, such a partial evaluator will in general not be able to reduce an expression fully.

The connection between evaluation and typing is usually given by the so-called *subject reduction property* indicating that typing is preserved during reduction. In order to prove this property we need two technical lemmas relating typing and substitution to each other.

Lemma 6.2 (Type Substitution Lemma)

$$B \vdash E : \sigma \ \Rightarrow\ B[\alpha := \rho] \vdash E : \sigma[\alpha := \rho]$$

Lemma 6.3 (Expression Substitution Lemma)

a)

$$\left.\begin{array}{c} B, x : \tau \vdash E : \sigma \\ B \vdash E' : \tau \end{array}\right\} \ \Rightarrow\ B \vdash E[x := E'] : \sigma$$

b)

$$\left.\begin{array}{c} B \vdash E[x := E'] : \sigma \\ B \vdash E' : \tau \end{array}\right\} \ \Rightarrow\ B, x : \tau \vdash E : \sigma$$

Proposition 6.4 (Subject Reduction Property)

$$\left.\begin{array}{c} B \vdash E : \sigma \\ E \Downarrow N \end{array}\right\} \ \Rightarrow\ B \vdash N : \sigma$$

Proof: By induction on the derivation of $E \Downarrow V$. Again, we will not treat all cases, but pick out some interesting ones.

- Rule (E-app). Suppose $EE' \Downarrow V$, since (1) $E \Downarrow \lambda x.E''$ and (2) $E''[x := E'] \Downarrow V$. Suppose $B \vdash EE' : \sigma$, since $B \vdash E : \tau \to \sigma$, and $B \vdash E' : \tau$. By IH for (1) one has $B \vdash \lambda x.E'' : \tau \to \sigma$. From typing rule σ-λ it follows that $B, x : \tau \vdash E : \sigma$. Now, by substitution lemma 6.3 (a) one has $B \vdash E''[x := E'] : \sigma$. Using IH for (2) one can conclude that $B \vdash V : \sigma$.

- Rule (E-case-case). Suppose

$$\mathsf{case}\ E\ \mathsf{of}\ \cdots \mathbf{C}_j \vec{y_j} \to E_j \cdots \Downarrow \mathsf{case}\ D\ \mathsf{of}\ \cdots \mathbf{A}_i \vec{z_i} \to V_i \cdots$$

  since (1) $E \Downarrow \mathsf{case}\ D\ \mathsf{of}\ \cdots \mathbf{A}_i \vec{z_i} \to D_i \cdots$ and (2) $\mathsf{case}\ D_i\ \mathsf{of}\ \cdots \mathbf{C}_j \vec{y_j} \to E_j \cdots \Downarrow V_i$. Suppose

$$B \vdash \mathsf{case}\ E\ \mathsf{of}\ \cdots \mathbf{C}_j \vec{y_j} \to E_j \cdots : \sigma$$

  because $B \vdash E : \tau$, $B \vdash \mathbf{C}_j : \vec{\rho} \Rightarrow \tau$, and $B, \vec{y_j} : \vec{\rho} \vdash E_j : \sigma$. By IH for (1), $B \vdash \mathsf{case}\ D\ \mathsf{of}\ \cdots \mathbf{A}_i \vec{z_i} \to D_i \cdots : \tau$. Hence $B \vdash D : \eta$, $B \vdash \mathbf{A}_i : \vec{\theta} \Rightarrow \eta$, and $B, \vec{z_i} : \vec{\theta} \vdash D_i : \tau$. By lemma 6.3 $B, \vec{y_j} : \vec{\rho} \vdash \mathsf{case}\ D_i\ \mathsf{of}\ \cdots \mathbf{C}_j \vec{y_j} \to E_j \cdots : \sigma$. Now we can apply IH for (2), so $B, \vec{y_j} : \vec{\rho} \vdash V_i : \sigma$. Finally, the typing rule $\sigma - \mathsf{case}$ can be applied, yielding $B \vdash \mathsf{case}\ D\ \mathsf{of}\ \cdots \mathbf{A}_i \vec{z_i} \to V_i \cdots : \sigma$. □

There are two ways to determine constructors that are 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 two definitions make the distinction between the different ways of indicating constructors precise.

Definition 6.3 (Constructors of normal forms) *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.*

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

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

Definition 6.4 (Constructors of types)

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

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

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

Example 6.1 For the types

$$\begin{array}{lcl} \mathsf{List}\ \alpha & = & \mathsf{Nil} \mid \mathsf{Cons}\ \alpha\ (\mathsf{List}\ \alpha) \\ \mathsf{Tree}\ \alpha & = & \mathsf{Node}\ \alpha(\mathsf{List}\ (\mathsf{Tree}\ \alpha)) \end{array}$$

we have

$$\begin{array}{lcl} C_T(\mathsf{List}) & = & \{\mathsf{Nil}, \mathsf{Cons}\} \\ C_T(\mathsf{Tree}) & = & \{\mathsf{Node}, \mathsf{Nil}, \mathsf{Cons}\} \end{array}$$

As a first step towards a proof of the main result of this section we concentrate on expressions that are already in symbol normal form. To prepare for this result we need two auxiliary lemmas.

Lemma 6.5

a) Let σ, τ be types such that $\mathrm{FV}(\sigma) \subseteq \mathrm{FV}(\tau)$. *Then*

$$C_T(\sigma) \subseteq C_T(\tau) \ \Rightarrow\ C_T(\sigma[\alpha := \rho]) \subseteq C_T(\tau[\alpha := \rho]).$$

b) For any basis B:

$$B \vdash \mathbf{C} : \vec{\tau}{\Rightarrow}\sigma \ \Rightarrow\ \{\mathbf{C}\} \cup C_T(\vec{\tau}) \subseteq C_T(\sigma)$$

Lemma 6.6 *Let $\vec{N} \in N_s$. Then*

$$B \vdash x\ \vec{N} : \sigma \ \Rightarrow\ \sigma \in B.$$

Proof: This property follows directly from the rules (σ-var) and (σ-app). □

If an expression is already in symbolic normal form, then its typing gives a safe approximation of the constructors that are possibly generated by that expression. This is stated by the following proposition.

Proposition 6.7 *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.

- Case $N \equiv x$. Trivial, since $C_N(x) = \emptyset$.
- Case $N \equiv \mathbf{C}\vec{N}$. Then $C_N(N) = \{\mathbf{C}\} \cup C_N(\vec{N})$. Suppose $B \vdash \mathbf{C}\vec{N} : \sigma$, since $B \vdash \mathbf{C} : \vec{\tau} \Rightarrow \sigma$ and $B \vdash \vec{N} : \vec{\tau}$. By IH $C_N(\vec{N}) \subseteq C_T(B) \cup C_T(\vec{\tau})$. By lemma 6.5 (a and b) $\{\mathbf{C}\} \cup C_T(\vec{\tau}) \subseteq C_T(\sigma)$, so $C_N(N) \subseteq C_T(B) \cup C_T(\sigma)$.
- Case $N \equiv \mathsf{case}\ N'\ \mathsf{of}\ \cdots \mathbf{C}_i\vec{y_i} \to N_i \cdots$. Then $C_N(N) = C_N(N') \cup_i C_N N_i$. Suppose $B \vdash \mathsf{case}\ N\ \mathsf{of}\ \cdots \mathbf{C}_i\vec{y_i} \to N_i \cdots : \sigma$, since $B \vdash N' : \tau$, $B \vdash \mathbf{C}_i : \vec{\rho} \Rightarrow \tau$, and $B, \vec{y_i} : \vec{\rho} \vdash N_i : \sigma$. By IH $C_N(N') \subseteq C_T(B) \cup C_T(\tau)$. By lemma 6.6 $\tau \in B$, and hence $C_T(\tau) \subseteq C_T(B)$. Again by IH $C_N(N_i) \subseteq C_T(B) \cup C_T(\vec{\rho}) \cup C_T(\sigma)$. Now by lemma 6.5 (a and b) one has that $C_T(\vec{\rho}) \subseteq C_T(\tau)$, so $C_N(N_i) \subseteq C_T(B) \cup C_T(\sigma)$.

All other cases are treated in the same way. □

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 6.8 (Deforestation Property)

$$\left.\begin{array}{c} B \vdash E : \sigma \\ E \Downarrow N \end{array}\right\} \;\Rightarrow\; C_N(N) \subseteq C_T(B) \cup C_T(\sigma)$$

Proof: By proposition 6.1, 6.7, and 6.4. □

7 Optimising Generics

Before we can apply a generic operation to a concrete object we have to convert this object into its generic representation. This conversion as well as the presence of many higher-order functions introduces a great amount of overhead during real execution. In fact, this makes the translation scheme for generics presented in the previous section unsuited as a basis for a real implementation. In this section we will 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 following lemma prepares for the main result of this section.

Lemma 7.1 *Let $\mathcal{G}$ be a generic function of type γ, T a data-type, and let $\mathbf{G}_T$ be the instance of $\mathcal{G}$ on T. Suppose $\mathbf{G}_T \Downarrow N_T$. Then for any data type S one has*

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

Proof: By proposition 6.8, 6.4, and 5.2

Proposition 7.2 (Efficiency of generics) *Non-recursive generics can be implemented efficiently.*

Proof: First we create a generic instance according to definition 4.12. Then, the symbolic normal form of this instance is computed via symbolic evaluation. From the previous lemma it follows that none of the (internal) basic data types $\{\mathbb{1}, +, \times, \rightarrow, \rightleftarrows\}$ will occur in this symbolic normal form (provided that these data types are indeed internal, i.e. are not part of the generic type or of the instance type). Hence if such an optimized generic operation is used inside a program, the evaluation of this program will not lead to the creation of any of these internal constructors.

7.1 On termination of symbolic evaluation

Until now we have avoided the problem that occurs when dealing with programs that do not terminate when evaluated symbolically. 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 problem arises when we deal with generic instances on recursive data types. A 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 $Y = \lambda f.f(Yf)$. 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 7.1 (Non-recursive specialization) The specialization of Map to List without recursion:

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

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

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

showing that all intermediate data structures are eliminated.

Suppose the generic instance has type σ. Then the non-recursive variant (with the extra recursion parameter) will have type $\sigma \rightarrow \sigma$ which means that proposition 7.2 is still valid.

However, this manner of handling recursion will not work if the kind-indexed type γ contains recursive data types. Consider for example the monadic mapping function for the list monad $\mathsf{Mmap} = \Gamma\alpha\beta.\alpha \rightarrow \mathsf{List}\ \beta$. The specialization of Mmap to any data type will use the embedding-projection specialized to $\Lambda\alpha\beta.\alpha \rightarrow \mathsf{List}\ \beta$ (see definition 4.11). This embedding-projection will contain a call to the (recursive) embedding-projection for the List-type. Now we cannot get rid of recursion (using the Y-combinator) because it is not possible to replace the call to $\mathcal{EP}\langle\gamma\rangle$ in the specialization of Mmap by a call to a non-recursive variant of $\mathcal{EP}\langle\gamma\rangle$ and to reintroduce recursion afterwards.

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

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 subject to further research.

8 Related Work

The generic programming scheme that we use in the present paper is based on the approach by Hinze[Hin00a]. In particular, our proof of type correctness of the specialized instances is adopted from this thesis.

In [HP01] Peyton Jones and Hinze 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 generic optimization.

Initially, we have tried to optimize generics by using *deforestation* [Wad88] and *fusion* [Chi94, AGS02]. 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 in the kind of results we want to achieve. To the best of our knowledge, nobody else uses typing to approximate the results of a computation and relates this to an operational semantics.

9 Conclusions and future work

In the present paper we defined a symbolic evaluation algorithm and proved that it is able to optimize code generated by the generic specialization procedure. The optimized code is close to what the hand-written code would be. Problems arise when generic function types are involved that contain recursive type constructors. These type constructors give rise to recursive embedding projections which can lead no 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 on online non-termination detection, most likely based on the homeomorphic projections [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 to improve performance of generics.

References

[AGS02] Diederik van Arkel, John van Groningen, and Sjaak Smetsers. Fusion in practice. In Ricardo Peña and Thomas Arts, editors,

Proceedings of the 14th International Workshop on the Implementation of Functional Languages, IFL'02, volume 2670 of *LNCS*. Departamento de Sistemas Informá ticos y Programación, Universidad Complutense de Madrid, Springer, September 2002.

[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 the Implementation of Functional Languages, IFL'01*, pages 168–186. Älvsjö, Sweden, September 2001.

[Bar92] Henk Barendregt. Lambda calculi with types. In S Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Volumes 1 (Background: Mathematical Structures) and 2 (Background: Computational Structures)*, volume II. 1992.

[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.

[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.

[Jø92] Jesper Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Nineteenth 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.

[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.