Generic Functional Programming

Conceptual Design, Implementation and Applications

Artem Alimarine

Generic Functional Programming

Conceptual Design, Implementation and Applications

een wetenschappelijke proeve op het gebied
van de Natuurwetenschappen, Wiskunde en Informatica

Proefschrift

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de Rector Magnificus prof. dr. C.W.P.M. Blom,
volgens besluit van het College van Decanen
in het openbaar te verdedigen op
woensdag 14 September 2005
des namiddags om 3.30 uur precies.

door

Artem Alimarine

geboren op 2 december 1971 te Moskou

©Copyright 2005 Artem Alimarine
ISBN 90-9019574-2
Typeset with LaTeX 2_ε

Acknowledgements

This thesis would not come into existence without help of many people, to which I would like to express my gratitude here.

First of all I would like to thank my promotor Rinus Plasmeijer and co-promotor Sjaak Smetsers. Rinus helped me to find my ways in the computer science research and grow into a scientist. He guided the research in the right direction, always giving freedom to choose my own way. Rinus together with Marko van Eekelen created an excellent environment for productive research, which I had a privilege to work in. Working with Sjaak Smetsers taught me a lot about doing scientific research. His patience and dedication helped me to do the research and served as an excellent example.

The research presented in this thesis is a result of a joint effort with several co-authors: Peter Achten, Pieter Koopman, Jan Tretmans, Sjaak and Rinus. I would therefore like to thank them for fruitful cooperation and discussions.

All the members of the Software Technology group in Nijmegen were always helpful during these years. In particular I would like to thank John van Groningen and Ronny Wichers Schreur for helping me to understand the internals of the Clean Compiler.

I am especially grateful to Marianne Dekker for her constant and active help with the administrational, organizational and bureaucratical issues involved in the preparation of the thesis and the defense.

I would also like to thank Jesse Hughes for organizing a reading group on category theory and for supervising it, helping me to study it.

I fill very grateful to the members of the Doctoral Thesis Commitee: Johan Jeuring, Ralf Hinze and Sven-Bodo Scholz. They took time to read the draft of the thesis very carefully and made useful and detailed comments. Johan spotted some serious errors and many typos giving me a chance to improve the thesis.

Finally, I would like to express my deep gratitude to my family who were there for me supporting me in all possible ways despite the thousands kilometers that separate us.

Contents

Chapter 1

Introduction

This thesis explores *generic* functional programming. Generic programming is important because it relieves the programmer from defining similar functions for various data types. With generic programming such functions can be specified once and for all saving the implementation and maintenance time and costs. Generic programming facilitates code reuse by automatic derivation of code.

A pure lazy functional language CLEAN has been used as the basis for the research. This chapter introduces the concepts used throughout the text.

We start with a short introduction of functional programming highlighting relevant features (section 1.1). The mathematical nature of functional languages facilitates formal reasoning about programs, program derivation and transformation. Most functional languages have a strong static type system based on algebraic types. Generic programming uses the algebraic structure of types in order to derive programs that work with values of these types. Therefore, type systems are a key feature for this research. They are discussed in section 1.2.

Section 1.3 introduces dynamics, a concept that allows incorporation of dynamic typing into a statically typed language. Static type checks are performed at compile-time, whereas in modern programming environments not all program parts are available during compilation. For instance, plug-ins or dynamically linked libraries typically become available only at run-time. Therefore, dynamic typing is necessary to allow for type checking of the code linked to the program during its execution. In this thesis we explore the integration of generic programming into a functional language, and, in particular, its interaction with dynamics.

Section 1.4 introduces generic programming. It outlines the main idea of using the structure of types to define generic functions. The review of work related to generic programming is put separately in chapter 7.

A feature of a programming language is useful only if its performance is adequate. The code generated by the generic compiler is slow and memory consuming, which dictates the need for optimization. Section 1.5 briefly introduces program optimization techniques used in the subsequent chapters to optimize generic programs.

As mentioned above, this study is based on the programming language Clean. All examples in this thesis are given in Clean with the exception of the chapters on optimization (4 and 5), the cases where a simple core functional language is used. Related features of the programming language Clean are introduced in section 1.6. In this thesis we will use Clean code examples.

The last section of this chapter (1.7) describes the goals and the contributions of this research. It also gives an outline of the rest of the thesis.

1.1 Functional Programming Languages

Functional programming languages are based on the mathematical concept of a function. A function is a mapping from the argument set (domain) to the result set (range). Programs in functional programming languages are built from function applications. Like in mathematics the function result depends only on the function arguments. A functional programming language is called *pure* if it has this property, referred to as *referential transparency.* In contrast, in *imperative languages* the result of a function does not depend only on its arguments, but may also depend on some state, such as global variables.

Referential transparency facilitates reasoning about programs. Mathematical methods can be readily applied to prove properties of functional programs. This means, for instance, that it is relatively easy to perform program transformations that provably maintain some properties. It is easier for the programmer to understand a program fragment, since a statement of the program does not depend on global variables textually scattered through the whole program. The meaning of a statement is determined solely by that statement.

Modern software systems are very complex. The usual design technique used to deal with complexity is to *divide and conquer*, i.e. to break a complex task into a composition of smaller subtasks. Referential transparency helps to do that, since the properties of the whole task depend only on the properties of the composed subtasks.

Referential transparency also means that the evaluation order of the program statements can be changed without altering the meaning of the program. Evaluation of the program statements can be carried out in the data driven order with implicit control flow. In other words, only the code needed to obtain the final result is executed. This evaluation order is called *lazy*. Many pure functional languages including CLEAN adopt lazy evaluation. As opposed to lazy evaluation, the *strict* evaluation is based on the explicit control flow given by the order of the statements in the program. Languages that do not have referential transparency typically adopt strict evaluation. For instance, imperative languages use strict evaluation. Strictness is needed to ensure that the side effects of the statements occur in a certain predictable order. Lazy evaluation has a number of properties: only expressions that are really needed to compute the final result are evaluated; computations involving infinite data can be carried out. Lazy languages do not need built-in conditional constructs because they can be implemented as library functions in the language itself. For instance, the conditional construct *if* can be implemented as a function that takes three arguments: the condition, *then* and *else* branches. The condition determines the branch subject to evaluation.

Functional languages support higher-order functions: functions that take other functions as arguments or return functions as results. Higher-order functions enable the *combinatorial* style of programming. Combinators are higher-order functions that combine their arguments in various ways. Programs are built from basic blocks by combining them with help of combinators. Typical examples of combinators are monad, arrow and parsing combinators.

Many advanced features of a functional language are compiled to code that involves higher-order functions and combinatorial composition of the code. For instance, Haskell's *do* notation is compiled into code that uses monadic combinators. Another example is the dictionary translation of Haskell's and CLEAN's overloading (see section 1.2.2). The generic programming approach used in this thesis also uses a combinatorial code composition.

Laziness is an essential concept for the combinatorial style of programming because it allows combinators to evaluate their arguments only when their results are needed. This is especially important in case of recursive combinators, where laziness ensures termination.

1.2 Type Systems

Most operations can only be performed on values of some *type*. For instance, the increment operation is only applicable to numbers. Applying an operation to a value of a wrong type, e.g. trying to increment a string, is considered a *type error*.

Static *type systems* use compile-time checks to guarantee that the input program does not contain type errors. For instance, such a type system rejects a program containing the expression `1 + "abc"`. A type system is called *safe*, if it guarantees that type errors cannot occur at run-time.

From the abstract interpretation viewpoint, the static type system computes a compile-time approximation of the run-time value of an expression. This approximation is called the *static type* of the expression. For instance, for a term 1+2 the type is `Int`.

Type systems are good not only for detecting errors, but also for documenting the program. As we show later, types can also be used as specifications for the purpose of generating program fragments. In essence, generic programming uses types to derive functions that work on values of these types. For instance, the pretty-printer for a type is determined by that type itself. Therefore, types can be used as formal specifications to derive programs.

Static typing has two modes of operation: type *checking* and type *inference*. The type systems with type checking require that the programmer provides function types. The function bodies are then checked for consistency with those types. The type systems with type inference do not require that the programmer provides the types. Types are derived automatically by the type system. In general, type checking admits more programs than type inference. For some programs, if the type is provided by the programmer, the compiler is able to check that the program has that type. However, if the type is not provided for the same program, the compiler is not able to derive it.

1.2.1 Hindley-Milner Type System

Most pure functional languages are statically typed. They use a type system based on the Hindley-Milner (HM) type system [Hin69, Mil78, DM82, Myc84] first introduced by Milner in functional language ML. This type system combines type checking and type inference: the user has a choice of providing the type or having it derived by the type system.

One of the essential aspects of the HM type system is that it supports *parametric polymorphism*: a function can act on values of a family of types rather than of fixed types. For instance, the identity function can be defined for any type.

```
id :: a → a
id x = x
```

This function is polymorphic since it works for any type a.

A new data type can be introduced by means of an *algebraic type definition*. For instance, the list, binary tree and n-ary rose tree data types can be defined as follows[1].

```
:: List a     = Nil | Cons a (List a)
:: Tree a b = Leaf a | Branch (Tree a b) b (Tree a b)
:: Rose a     = Node a (List (Rose a))
```

The list data type has two branches: the list can be empty (Nil) or nonempty (Cons). A non-empty list is a pair of head and tail. The data type is polymorphic: it is parameterized with the list element a. The type is also recursive: the tail of the list is a list. The binary tree data type is also polymorphic. It is parameterized with two arguments: one for the leaf labels and one for the branch labels. The n-ary tree is built with the help of the existing list data type.

As noted before, parametric polymorphism enables functions to work on a family of types. For instance, to determine the length of a list there is no need to know the type of the list elements.

```
length :: (List a) → Int
length Nil = 0
length (Cons x xs) = 1 + length xs
```

This function is polymorphic in the list element type. However, for a list to be summed the elements must be numbers.

The HM type system is very flexible, and it still supports type inference. For instance, the types of id and length do not have to be specified, they can be automatically derived. Many extensions to HM loose this ability to infer types (see for instance Section 1.2.5).

1.2.2 Overloading

Similar operations can often be defined for various types. For instance, the equality can be defined for integers, characters, booleans, and for many user-defined data types. Overloading [WB89] is a mechanism that allows the use

[1] In CLEAN a data type definition starts with ::.

of the same name for similar operations on different types. For example, one can write both `x == 1` and `'a' == y` using the same operator name `==`. The overloading system assures that for the former case the equality operator for integers is used, and for the latter case, that for characters.

The general type of an overloaded operation is given by a *class* declaration. For instance, the following class declares an overloaded equality operator[2]

```
class Eq a where
    (==) infix 2 :: a a → Bool.
```

The implementations for specific data types are given by the *instance* declarations. For example, the instances for integers and characters are defined in the following way:

```
instance Eq Int where
    (==) x y = eqInt x y

instance Eq Char where
    (==) x y = eqChar x y
```

Instance functions have the types obtained by instantiation of the general type provided by the class declaration. In the example above the equality for integers has type `Int Int → Bool` and the equality for characters has type `Char Char → Bool`, which are both instances of the general type `a a → Bool` obtained from it by substituting `Int` and `Char` for `a` respectively.

Overloading enables the programmer to write abstract code independent of specific types, i.e. the code that works on any data type that supports certain operations (e.g. equality). The programmer provides the instances of these operations on the data types in question. For example, the linear lookup in an associative list works with any key that supports equality:

```
lookup :: key value [(key, value)] → value | Eq key
lookup key default []               = default
lookup key default [(k,v): tail ] =
    | k == key  = v
    | otherwise = lookup key default tail
```

The function is polymorphic in `key`. However, it is not *parametrically* polymorphic, since it requires that the equality operation (==) is defined on the `key` type. This kind of polymorphism is often referred to as *ad-hoc* polymorphism. Class requirements on polymorphic arguments are called *context*

[2]In CLEAN the function type reflects the formal arity of a function. The type `a a → Bool` means that the formal arity of the instances must be 2.

restrictions[3].

The HM type system with overloading does not loose the ability to *infer* types. For instance, the type of `lookup` does not have to be specified – it would be inferred by the type system.

Note that a type class defines the *interface* for all its instances. It does not specify how they should be implemented: the implementation is provided by the programmer. However, often for some overloaded operation also the *implementations* are similar. For instance, the implementation of equality on a user-defined type normally performs component-wise comparison of the type constituents. In other words, the equality for those types is determined by the type itself. Instead of writing similar definitions manually, they can be generated automatically if the type structure is known. This is essentially the goal of generic programming.

Dictionary translation of overloading

Here we show how overloaded functions are translated into a core language without overloading. This is needed to understand the relation between overloading and generic programming. We use the dictionary translation of overloading [WB89]. We explain the translation by the example of the overloaded equality operator.

Class declarations are translated into records. Such a record is called the *dictionary* of the class. It contains one field for each member of the class. For instance, the equality class `Eq` is translated into a record (translated code is given in *italic*)

:: Eq a = { dict_eq :: a a → Bool }

To distinguish between the translated code and the original code, we *italize* the translated code. The members of a class are translated into functions that select the corresponding field in the dictionary.

(==) :: (Eq a) a a → Bool
(==) dict x y = dict.dict_eq x y

Note that, unlike the original equality function, the translated function takes three arguments. The first (dictionary) argument is added by the translation. The instance declarations are translated directly as values of the dictionary record types.

eq_{Int} *:: Eq Int*
eq_{Int} *= { dict_eq = eqInt }*

[3]In CLEAN the context restrictions are given in the end of a function type after |.

```
eq_Char :: Eq Char
eq_Char = { dich_eq = eqChar }
```

When the user writes, for instance,

```
Start = 'a' == 'b'
```

the overloading resolution automatically plugs in the right dictionary, resulting in

```
Start = (==) eq_Char 'a' 'b'
```

In the implementation, the functions that use overloading are given an additional *dictionary argument* for each context restriction. For instance, the lookup function has a context restriction on keys, which leads to a dictionary argument.

```
lookup :: (Eq key) key value [(key, value)] → value
lookup dict key default []              = default
lookup dict key default [(k,v): tail]
    | (==) dict k key  = v
    | otherwise        = lookup dict key default tail
```

Again, when an overloaded function is used, the system automatically plugs in the dictionaries. For instance, the code

```
Start = lookup 1 "default" [(1,"one"), (2, "two")]
```

is translated into

```
Start = lookup eq_Int 1 "default" [(1,"one"), (2, "two")].
```

1.2.3 Higher-order types and type kinds

The number of (polymorphic) arguments in a data type definition is called its *arity.* For instance the list and the rose tree (defined on page 15) have arity one; the binary tree has arity two. The compiler checks that data type constructors are applied consistently with respect to arity. To facilitate the checks, the compiler uses a simple type system. This type system assigns types to types. The type of a type is called a *(type) kind.* In a simple case, the type kind is just the arity.

Most modern functional languages allow higher-order types. Consider, for instance the rose tree type. It uses the list type to hold the sub-trees. We can generalize the rose tree type by parameterizing it with the container used to hold children.

```
:: GRose f a = GNode a (f (GRose f a))
```

Here f stands for a container, a type that takes one argument (has arity one). For instance the original rose tree can now be defined as a type synonym[4].

```
:: Rose' a :== GRose List a
```

To check consistency of such types it is not sufficient to use arities as type kinds. Instead, type kinds are represented as special binary trees

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

The usual type constructors, like Char on Int, have kind $\star$. The list type constructor List and the rose tree Rose have kind $\star \rightarrow \star$, the binary tree Tree has kind $\star \rightarrow \star \rightarrow \star$, the generalized rose tree GRose has kind $(\star \rightarrow \star) \rightarrow \star \rightarrow \star$.

1.2.4 Type Constructor Classes

Higher-order types are especially useful in combination with overloading. Consider, for instance, the mapping function for the list type:

```
map :: (a → b) [a] → [b]
map []      = []
map [x:xs]  = [f x : map f xs]
```

Mapping can be defined not only for lists, but for many other type constructors. Overloading on type constructors can be used to give all such mappings the same name:

```
class Functor f where
    mapf :: (a → b) (f a) → f b
```

Here the type variable f ranges over types of kind $\star \rightarrow \star$, such as List or Rose, rather than types of kind $\star$ like Int or (List Int). A class whose class variable ranges over the types of a kind higher than $\star$ is sometimes called a *type constructor class*. The instances are given in the usual way.

```
instance Functor List where
    mapf f Nil           = Nil
    mapf f (Cons x xs)   = Cons (f x) (mapf x xs)

instance Functor Rose where
    mapf f (Rose x xs) = Rose (f x) (mapf (mapf f) xs)
```

The generic approach we introduce in this thesis uses type constructor classes as the basis for generic functions.

[4]In CLEAN the syntax with :== introduces a type synonym.

1.2.5 Higher-rank polymorphism

The HM type system has a limited form of polymorphism, namely rank-1 polymorphism. This means that functions cannot take polymorphic functions as arguments. Rank-1 types can be written with explicit universal quantification of type variables, where the quantifiers can appear only on top. For instance, the type of length can be written with an explicit quantifier on top as ∀a.(List a) → Int.

Many modern functional languages extend polymorphism up to rank-2 [OL96]. With rank-2 polymorphism, a function's arguments can be quantified. For instance, the function below takes a polymorphic function as an argument

```
lengths :: (∀ a. (List a) → Int) (List b, List c) → (Int, Int)
lengths l (xs, ys) = (l xs, l ys)
```

The first argument of this function is applied to values of different types, so it needs to be polymorphic.

A functional programming language based on HM type system can also be extended to rank-n polymorphism For instance, Peyton Jones and Shields [JS04] provide a practical extension for Glasgow Haskell. There is also an extension of HM typing to rank-n polymorphism [JS04].

A rank-n function takes a rank-$(n-1)$ function as an argument. For instance, one can define a rank-3 function that takes the rank-2 lengths as an argument.

The code produced by the generic scheme used in this thesis can only be typed with rank-n typing. In fact, the kind of a type determines the polymorphism rank, needed to type generic instances of that type [Hin00c].

1.3 Dynamics

Some programming languages, e.g. Lisp and Smalltalk, use *dynamic* instead of static typing. Dynamically typed languages rely on run-time type checks to ensure type-correctness of the program execution result. Such a language needs run-time type tagging of expressions to perform the type checks. If a type check fails, the whole program typically terminates with an error or throws an exception.

The major advantage of dynamic typing is its flexibility. Dynamic typing allows for very fast creation of small programs, which is useful for scripting. Dynamic typing enables programs to dispatch on the type of data. By means of a so called *type case* the program can adapt to the type of the input data.

The major disadvantage of dynamic typing is that the typing errors are only detected at run-time. Basically, run-time type checks occur too late, i.e. when the program is running rather than in the process of its development. Another disadvantage is that some program transformations are hard to perform, because the types are not known at compile time.

Static typing facilitates compile-time error detection and contributes to the compiler optimizations. However, there are situations when the type of an expression cannot be determined at compile-time. Consider for example a program that uses plug-ins linked into the program at run-time. Since the type of such a plug-in becomes known to the main program only at run-time, the compile-time type checking cannot be used.

The concept of *dynamics* was introduced to use the benefits of dynamic typing in a statically typed language [ACPP91, ACP^{+}92, Pil98]. The idea is to have a special predefined static type called *Dynamic* that can hold a value of any static type. Values of this type we call *dynamics*. The run-time representation of a dynamic is a pair of an ordinary value and encoding of its static type. For instance, if a value v has type τ (we write v :: τ) then the dynamic for this pair is $\langle$v, $\tau\rangle$ :: Dynamic. This is similar to type tagging used in dynamically typed languages.

In essence, two primitives are needed to work with dynamics: pack and unpack. The first one packs a value and its static type into a dynamic, whereas the second one tries to unpack a dynamic as a value of a certain static type. Unpacking fails if the type stored in the dynamic does not match the required static type being unpacked.

```
pack :: τ → Dynamic | TC τ
unpack :: Dynamic → Maybe τ | TC τ
```

The primitive pack constructs a pair $\langle$v, $\tau\rangle$::Dynamic from a value v :: τ. The primitive unpack takes a value of the form $\langle$v, $\sigma\rangle$::Dynamic. It matches the requested type τ against the actual type σ. If the types match it returns Just v; otherwise it returns Nothing.

The context restriction TC τ is needed to reflect the fact that the functions are not *parametrically* polymorphic: their action depends on the actual instance of τ. Such functions are called type-dependent functions [Pil98]. The name TC stands for *type code*.

Since a dynamic can contain a value of any type, one can think of *dynamic polymorphism*. Functions that work with dynamics are polymorphic in the sense that they work with values of any type.

1.4 Generic Polytypic Programming

The term *generic programming* has two main meanings in the computer science literature. In object-oriented languages the word *generic* denotes a form of parametric or ad hoc polymorphism (e.g. C++ templates). In the world of functional languages we are interested in the word *generic* refers to *polytypic or structural* polymorphism. The term *polytypic programming* is also used in the literature. From now on, unless otherwise noted, the term *generic* refers to such structural polymorphism.

Polytypic programming is a way to specify a generic operation for an arbitrary type by induction on the structure of types. For instance, equality can be specified in a generic way as to work for lists of booleans, trees of integers etc.

1.4.1 Structural Representation of Types

The main idea behind generic programming is to define (generic) functions on the structural representation of a data type. In essence, generic programming exploits the following aspects of the type definitions.

Polymorphism. Types can be parameterized by other types. For instance, the list type is parameterized with a *type variable* that corresponds to the element type.

```
:: List a = Nil | Cons a (List a)
```

Application. Type definitions can refer to other type constructors, i.e. they can be built from other types. In particular, a type can refer to itself directly or indirectly, i.e. be *recursive*. For instance, the rose tree type

```
:: Rose a = Node a (List (Rose a))
```

is built with the help of the list type. Moreover, it is recursive, since it also refers to itself.

Structure. Type definitions have algebraic structure.

Choice, (co-product, sum). Type definitions can have alternatives. For instance, the list type has two alternatives: for the empty list (Nil) and non-empty lists (Cons).

Product. Data constructors can have zero or more arguments, i.e. they can be tuples of types. For instance, the Nil constructor has no arguments, and the Cons constructor has two arguments.

Arrow types. Type definitions can involve arrow types. For instance, the following data type contains an arrow.

```
:: Fun a b = F (a → b).
```

The arrow type must be treated specially, not just as a type constructor of arity two. This is because the arrow is known to be *contra-variant* in the argument and *co-variant* in the result.

In this structural view we set aside specific names of type and data constructors and focus exclusively on the structure of types.

Generic polytypic programming uses this structure to define functions that work for all types. The idea is simple: if we know what a function does on all these basic building blocks of types, we also know what it does on an arbitrary type. Given that a type constitutes a composition of basic building blocks, a function for such type would be a composition of functions for the basic blocks. Thus, generic functions are built inductively from the base cases. For instance, if the definition of equality covers all the base cases it covers all types inductively built from these base cases.

As will be shown in chapter 7, the use of this structure differs from one generic programming approach to another. Not all of them explicitly exploit all the aspects of the structure.

1.4.2 Why Generic Programming Matters

Generic programming provides the following benefits to programmers:

Less coding, code for free. Generic programming enables the programmer to achieve the same result with less coding because some boring routine code is generated automatically.

Less re-coding, simpler maintenance of programs. When a data type changes, programmers are often compelled to change functions involving that data type. However, instances of generic functions automatically adjust to changes in data types. Hence, a generic program is easier to modify and maintain.

Simplicity and beauty. Generic programs are often simpler and more elegant than their non-generic counterparts. At the same time they are more general.

1.5 Optimization by Partial Evaluation

In this section we give a short overview of partial evaluation. See [JGS93] for more details. Partial evaluation is used here as an optimization technique for eliminating overhead introduced by the generic specialization procedure. It is very important to optimize the generic code because it is full of conversions between values of types and their structural representations. The performance penalty is so bad that it can hinder the generic programming applicability in practice. Chapters 4 and 5 provide examples of such programs.

A functional language compiler needs to optimize the input programs to achieve acceptable performance of the resulting code. Most optimization techniques use partial evaluation. In this thesis we explore applicability of general purpose optimization techniques for optimizing generics.

1.5.1 Partial Evaluation

Program execution proceeds as evaluation to normal form *at run-time.* We refer to this as the *standard evaluation.* It proceeds as a sequence of single-step reductions. At each step one reducible expression, *redex*, is reduced. The normal form is reached when there are no more redexes left.

The idea of optimization by *partial evaluation* is to minimize the number of redexes to be evaluated at run-time, by evaluating as much redexes as possible *at compile-time.* The evaluation is called partial, because in general it cannot reach the normal form, since not all values of variables are known at compile-time. Thus, not all variables are bound at the time of the compile-time transformation. Therefore partial evaluation must deal with open terms, i.e. terms that contain unbound variables. A generalization of the standard evaluation to evaluation of open terms is called *symbolic evaluation.* Symbolic evaluation extends standard evaluation with rules for open terms. Consider for instance the function

$$\mathsf{foo}\ x\ xs = \mathsf{head}\ (\mathsf{Cons}\ x\ xs)$$

The following evaluation sequence illustrates symbolic evaluation of the right-hand side, which is an open term.

$$\mathsf{head}\ (\mathsf{Cons}\ x\ xs) \ \rightsquigarrow\ \mathsf{case}\ (\mathsf{Cons}\ x\ xs)\ \mathsf{of}\ \mathsf{Cons}\ y\ ys \rightarrow y \ \rightsquigarrow\ x$$

Here the intermediate constructor Cons is eliminated, not only making the resulting code for function foo faster, but also improving the memory usage.

1.5.2 The Termination

As standard evaluation, partial (symbolic) evaluation is not guaranteed to terminate in the presence of recursive functions. Optimizations are performed at compile time. It is unacceptable that a compiler does not terminate. Therefore, compile-time transformation algorithms perform so called *termination analysis* to ensure that the transformation terminates. Since the halting problem is undecidable, the termination analysis is conservative, i.e. it can reject programs that can be further transformed.

Normally, a transformation algorithm involves transformation on two levels: global and local. On the global level, the transformation is applied to each function in the program. The local level transformation is applied to the right-hand side of a function. It proceeds by induction on the structure of terms. Both iterations are potential sources of non-termination.

On the local level the transformation evaluates terms, which needs *unfolding* of function applications. Unfolding of recursive functions can lead to non-termination. Therefore, the termination analysis must prevent from infinite unfolding of functions. When the algorithm detects potential infinite unfolding of a function, it stops unfolding and reintroduces a call to a potentially recursive function. This step is called *folding*; it can lead to creation of new function definitions.

On the global level the transformation is applied to all function definitions. The folding steps of the local transformation can lead to creation of new function definitions. To achieve a good optimization result the transformation must be applied to these new functions, which can again lead to creation of new functions, to which the transformation must be applied, and so on. Thus, the termination analysis must ensure that only a finite number of functions is generated.

1.5.3 Online Termination Analysis

As said before, local non-termination is caused by repetitive unfolding of recursive functions. The idea of online termination analysis is to remember functions applications that are unfolded. Before unfolding a function application the algorithm checks whether a similar application has occurred before on the same evaluation path. The second occurrence is potentially dangerous because it can lead to the third unfolding and so forth. Therefore, when the algorithm detects such a repetition, it performs a *fold* step.

The algorithms differ in what function applications are considered to be similar. The most simplistic check would refuse all applications of the same function. In practice, however, this check is too restrictive. Not all such applications are dangerous. Only "growing" applications can lead to creation of infinite number of functions. One of the techniques that makes this precise is homeomorphic embedding [Leu98].

1.5.4 Offline Termination Analysis

Algorithms with offline termination analysis perform the local transformation in two stages. The first stage determines the redexes that are safe to reduce with respect to termination. The second stage performs actual reduction of these safe redexes.

One of the well-known techniques with offline analysis is *fusion* [Chi94, CK96, AGS03]. Fusion only considers function-to-function applications as redexes. For instance, fusion considers the application length (map f xs) as a redex. Instead of unfolding and folding directly, fusion combines such a pair of functions into a single function. Such a combination requires unfolding the involved functions. In this example the function length is called a *consumer* and the function map a *producer*. The transformation creates a single function length_map which performs only a single traversal of the list.

The termination analysis is performed off-line. As mentioned before, the local transformation is performed in two stages. The first stage determines which functions are *proper consumers* and *proper producers*. The second stage only fuses proper consumers with proper producers.

1.5.5 Optimization of Generic Functions

Chapters 4 and 5 of the present thesis are devoted to optimization of code generated by the generic specialization scheme. As noted above, our goal was to completely eliminate generic overhead. We identify generic overhead by data constructors belonging to the generic structural representation of types. In this way it is easy to see whether the program is completely optimized: it is, if it does not contain those data constructors.

Chapter 4 describes a simple partial evaluation algorithm with no termination analysis and shows how it can be used to optimize a certain class of generic programs. To prevent non-termination on recursive generic instances, we abstract from the recursion using the fix-point combinator. There it is formally proven that the generic overhead is completely eliminated for that class of generic programs. However, that class does not cover many practically important examples.

Originally we expected that the fusion (off-line) algorithm we had at hand [AGS03] would be able to eliminate generic overhead in many programs. However, the termination analysis of that algorithm is too restrictive even for relatively simple generic examples. Both the producer and the consumer analyses are too restrictive. The consumer analysis is too much syntactic rather than semantic based. The producer analysis does not take into account the context where the producer occurs. Chapter 5 describes the algorithm with both analyses improved in such a way that optimization of generics becomes possible for a large class of generic programs. This improvements are important not only for generic programs, but also for programs written in a combinatorial style, which involves a lot of intermediate data and higher-order functions.

Additionally, we experimented with an optimization algorithm with online termination analysis. However, the online algorithm depends too much on the evaluation order, which makes it hard to reason about, i.e. to predict the result of evaluation. Moreover, the online algorithm was significantly slower than the off-line algorithm. For relatively big examples, it was sometimes hard to distinguish between non-termination and long execution.

1.6 Clean

In the present thesis we use a pure lazy functional programming language Clean. Below we present the relevant features of the language.

The type System. The Clean type system is based on the Hindley-Milner typing system extended with overloading, higher-order types and rank-2 polymorphism. Additionally, Clean has an extension to the type system called *uniqueness typing* [vESP96, BS96]. This extension allows destructive updates without the loss of referential transparency. The idea is that in case of one reference to a node, the node can be updated in place rather than copied and returned. This feature is used, for instance, to implement functional input/output and efficient array operations.

Dynamics. Clean supports a very strong form of dynamic typing [Pil98]. All the Clean types can be stored in a dynamic. Dynamics can be eagerly or lazily written to or read from the disk. Sharing in the values stored in dynamics is preserved. Also unevaluated expressions can be stored in dynamics. When a dynamic containing an unevaluated expression is loaded, the code needed to evaluate this expression may have to be linked into the running program. The dynamic linker links the code at run time. Dynamics have been used as the basis for a functional shell [WP02].

Generics. The author of the present thesis has designed a generic programming extension for the programming language CLEAN and implemented it as part of the CLEAN 2 compiler. The extension is presented in detail in chapter 2. The goal of the implementation was to create a vehicle for the generic programming research used to study interaction of generic programming with other features of a functional programming language and applicability of generic programming in practice.

GENERIC CLEAN is based on the Hinze's ideas of type-indexed values and kind-indexed types. The novel feature of GENERIC CLEAN is that it integrates this generic approach with the overloading mechanism of CLEAN.

Generics are integrated with the module system of the language. Generic functions and instances can be exported from and imported to a module.

Generics are also integrated with the uniqueness typing of Clean. The generic function types can be given uniqueness attributes. Like in the Hinze's kind-indexed approach, the types of the instances are produced from the type of the generic function, with the help of kind-indexing. The kind-indexing procedure has been extended to support uniqueness typing.

1.7 Outline of this Research

1.7.1 Research Questions

Our research was motivated by the following research questions.

Conceptual design. How do we incorporate generic programming into a functional language like CLEAN? More specifically, how do we combine a generic programming feature with the existing features of CLEAN?

Performance. How do we make the generated code adequate in terms of performance?

Applications. How useful is generic programming in practice?

1.7.2 Research Plan

To answer the questions above we needed an implementation of a generic programming extension to experiment with. Since we wanted to experiment with the interaction of generic programming with other features of the language, the generic feature had to be implemented as an extension to a real functional programming language. Therefore, the author has designed and implemented a generic programming feature as part of the CLEAN compiler. The design was based on an existing approach to generic programming, namely, on type-indexed values and kind-indexed types. We have chosen this approach, because it can handle arbitrary types of arbitrary kinds. However, we adopted the design to fit better with the other features of the language. In particular, we have combined generics with overloading (chapter 2).

The implementation gave us the possibility to experiment with the interaction of generics and dynamics (chapter 3). Dynamics can contain values of arbitrary types. Generic functions work on values of arbitrary types. Therefore, it is natural that generic functions work on dynamic values. Generic specialization happens at compile-time, whereas dynamics are a run-time concept. Moreover, generic functions are not first-class citizens of the language; they are schemes used to generate first-class instance functions. We studied how generic functions can work on dynamic values and how they themselves could be stored in dynamics, in other words, how generic functions can be made first-class. This research has been continued in [WSP04].

A programming language feature is only useful in practice if its performance is adequate. The code produced by the generic specialization is extremely slow, because it uses numerous conversions between values and their structural representations. Thus, optimization was required to make the generated code efficient. Our intention was to develop a general purpose optimization algorithm applicable for optimizing generic programs. We chose for a general purpose algorithm, because optimization is needed not only for generics, but also, for example, for dictionaries introduced by overloading or for monadic programs. Compiler developers prefer to have only one optimizer in the compiler. Another requirement we had was complete elimination of generic overhead in a large class of generic programs.

The experiments with optimization were performed not in the CLEAN compiler, but in a separate simple functional language interpreter. In this way it was easier to experiment and modify the transformation algorithm. The intention was, indeed, to implement the algorithm later in the real CLEAN compiler. The research on optimization led to a general purpose optimization algorithm that can completely eliminate generic overhead for many generic programs (chapters 4 and 5). The research on optimization of generics is not limited to CLEAN, but can be used for optimization of code produced by generic compilers based on the approach of type-indexed values.

To show usefulness of generic programming in practice we used generic programming in real-world applications. One of this applications, the automatic test system GAST is presented in chapter 6 of this thesis. Generic programming is used there to generate sequences of test data. This research has been further developed by van Weelden *et al.* [vWFO^{+}04]. They use GAST to test Smart Cards Applets written in Java. GAST is also used for testing software in the industry.

GENERIC CLEAN has been also used to implement Generic Graphical Editor Components [AEP03, AEP04, AEPW04a, AEPW04b]. Generic programming is used there to obtain the displaying and the graphic editing code for arbitrary types.

1.7.3 Outline of the Thesis

This thesis is essentially a collection of co-authored papers published elsewhere (chapters 2-6). The papers are taken with small changes in the layout; the references are merged. In particular, the related work overview and conclusions in each paper are left intact and reflect the situation at the time of the paper publication.

The chapters 2 and 3 are devoted to the design and the implementation of a generic extension to the compiler and inter-operation of generics with other language features: overloading, the module system and dynamics.

Chaper 2 is based on the paper *A Generic Extension for Clean* [AP02] is co-authored with Rinus Plasmeijer. It presents our design of the generic programming extension. This design is based on Hinze's *polytypic kind-indexed* approach [Hin00c, HP01]. Our generic functions combine polykinded generic functions and overloading. We also study inter-operation of generic functions with the module system. The generic extension was implemented by the author as part of the new CLEAN 2.1 compiler. The resulting compiler was used to experiment with generic programming by the author and other members of the ST group in the University of Nijmegen. In particular, the chapters 3 and 6 are based on the design. The syntax of the generic instances in chapter 2 has been converted to the new actual syntax because the paper [AP02] uses the old syntax of the first prototype.

Chapter 3 is based on the paper *First Class Generic Functions* [AAP02] co-authored with Peter Achten and Rinus Plasmeijer. In this chapter we show how generic programming can be combined with dynamics. The chapter focuses on turning generic function into first-class citizens of the language in order to use them with dynamics. Generics and dynamics have something in common: dynamic types can contain a value of any type, a generic function works on values of an arbitrary types. Thus, it should be possible for generic functions to operate on dynamics. Hinze proposed a simple unified approach to generics and dynamics [CH02a]. However, this approach does not use the same generic scheme as we use and it uses more restrictive dynamics. Our goal was to merry the existing powerful dynamics and generics. The research of this chapter has been used as the basis for [WSP04]

Chapters 4 and 5 are devoted to optimization of the code generated by the generic specialization procedure.

Chapter 4 is based on the paper *Optimizing Generic Functions* [AS04c] written in co-authorship with Sjaak Smetsers. This chapter introduces a type-based technique to prove that the result of partial evaluation has a certain shape. This technique is used to prove that partial evaluation completely eliminates the overhead introduced by the generic specialization for a large class of generic programs. It is shown that partial evaluation can be used as a simple practical method for optimizing generics.

Chapter 5 continues the subject of optimization of generic programs. It is based on the paper *Fusing Generic Functions* [AS05] and the technical report [AS04b], written in collaboration with Sjaak Smetsers. This chapter presents a general purpose optimization algorithm based on fusion. This method is capable of optimizing even a larger class of programs than the algorithm in the previous chapter. Most practical generic programs belong to that class. It is shown that this method completely eliminates the generic overhead for these programs.

Chapter 6 is based on the paper GAST*: Generic Automated Software Testing* [KATP02] written in co-authorship with Pieter Koopman, Jan Tretmans and Rinus Plasmeijer. This chapter is devoted to a real application of generic programming: generation of the test data for the automated testing system GAST. This test system has been used to test Smart Cards Applets [vWFO$^+$04] and is currently also used in industrial environments.

Chapter 7 contains an overview of work related to the generic programming approach used in the present thesis. This review is split in two parts: approaches of generic programming and applications of generic programming. We review the applications separately, because many of them can be implemented in more than one generic programming approach.

Chapter 8 summarizes the achievements of the presented research, discusses possible directions of future work in the field of generic programming and concludes.

In the aforementioned papers constituting this study, the author has contributed to the research, design and implementation related to generic programming.

1.7.4 Contributions

Here we enumerate the contributions of the presented research.

Combining generics and overloading. The proposed approach combines generics and overloading in such a way that kind-indexed generic functions are overloaded. This allows to define an overloaded function generically for any kind.

Mixing generic and specific behavior. A user specified instance for a type can refer to the instance which the generic scheme generates for that type. This allows, for instance, to specify specific actions for some data constructors of a data type and fall back to the generated code for the other constructors.

A practical implementation. The generic programming feature has been implemented as an extension to the programming language Clean in the Clean compiler. The implementation has facilitated further research in the field of generic programming and its applications.

- The implementation supports the module system. Generic functions and their instances can be exported from a module.
- The implementation supports uniqueness typing. Although the interaction of the uniqueness typing and generics has not been formalized, the implementation provides a practical solution: it allows to derive generic instances for data types with uniqueness properties that are useful in practice.

Integration with dynamics. We have developed an approach to inter-operation of poly-kinded generics functions and dynamics. Generics inter-operate with dynamics in two ways:

- Generic functions can be used to work on dynamic values. Conceptually, generic functions work on values of any type, whereas dynamics can contain values of any type. It is important for generic functions to work with dynamic values because it allows to handle dynamics generically, for instance, to compare to dynamic values. This is useful in applications like OS shell based on dynamics [WP02].
- Generic functions can be stored in dynamics, i.e. are made first-class citizens.

This research line has been continued by others [WSP04].

Optimization of generics. We have performed the first systematic study of optimization of the output of the generic specialization. The optimization is crucial since without it the generated code would be extremely slow.

- The proposed optimization technique predictably removes generic overhead for a large class of generic programs.
- The proposed optimization technique is applicable for generic approaches based on type-indexed values.
- The proposed optimization algorithm is general purpose: it can be used to optimize not only generic programs. For instance, programs written using monadic or parser combinators can be optimized as well. So, such an algorithm can replace many ad-hoc optimization steps. This is important, because compiler writers do not want to have several optimizers in the compiler.

Applications. Practical utility of generic programming has been shown by using generic programming techniques to develop:

- The design of a generic scheme for systematic and automatic generation of test data for the test system GAST. The test system is a real application used for software testing. The generic programming enables systematic and automatic generation of test data of an arbitrary type, which is crucial for automated testing.
- The generic programming extension has been used by others to implement generic Graphical Editor Components [AEP03, AEP04, AEPW04a, AEPW04b].

Chapter 2

A Generic Programming Extension for Clean

Generic programming enables the programmer to define functions by induction on the structure of types. Once defined, such a generic function can be used to generate a specialized function for any user defined data type. Several ways to support generic programming in functional languages have been proposed, each with its own pros and cons. In this paper we describe a combination of two existing approaches, which has the advantages of both of them. In our approach overloaded functions with class variables of an arbitrary kind can be defined generically. A single generic definition defines a kind-indexed family of overloaded functions, one for each kind. For instance, the generic mapping function generates an overloaded mapping function for each kind.

Additionally, we propose a separate extension that allows to specify a customized instance of a generic function for a type in terms of the generated instance for that type.

2.1 Introduction

The standard library of a programming language normally defines functions like equality, pretty printers and parsers for standard data types. For each new user defined data type the programmer often has to provide similar functions for that data type. This is a monotone, error-prone and boring work that can take lots of time. Moreover, when such a data type is changed, the functions for that data type have to be changed as well. Generic programming enables the user to define a function once and specialize it to the data types he or she needs. The idea of generic programming is to define the functions by induction on the structure of types. This idea is based on the fact that a data type in many functional programming languages, including Clean, can be represented as a sum of products of types.

In this paper we present a design and implementation of a generic extension for Clean. Our work is mainly based on two other designs. The first is the generic extension for Glasgow Haskell, described by Hinze and Peyton Jones in [HP01]. The main idea is to automatically generate methods of a type class, e.g. equality. Thus, the user can define overloaded functions generically. The main limitation of this design is that it only supports type classes, whose class variables range over types of kind $\star$.

The second design described by Hinze in [Hin00c] is the one used in the Generic Haskell Prototype. In this approach generic functions have so-called kind-indexed types. The approach works for any kind but the design does not provide a way to define overloaded functions generically.

The design presented here combines the benefits of the kind-indexed approach with those of overloading. Our contributions are:

- We propose a generic programming extension for Clean that allows for kind-indexed families of overloaded functions defined generically. A generic definition produces overloaded functions with class variables of any kind (though the current implementation is limited to the second-order kind).

- We propose an additional extension, customized instances, that allows to specify a customized instance of a generic function for a type in terms of the generated instance for that type.

The paper is organized as follows. Section 2.2 gives an introduction to generic programming by means of examples. In Section 2.3 our approach is described. We show examples in Generic Clean and their translation to non-generic Clean. In section 2.4 we discuss the implementation in more detail. In section 2.5 we describe customized instances. Finally, we discuss related work and conclude.

2.2 Generic Programming

In this section we give a short and informal introduction to generic programming by example. First we define a couple of functions using type constructor classes. Then we discuss how these examples can be defined generically.

2.2.1 Type Constructor Classes

This subsection demonstrates how the equality function and the mapping function can be defined using overloading. These examples are the base for the rest of the paper. We will define the functions for the following data types:

```
:: List a     = Nil | Cons a (List a)
:: Tree a b = Tip a | Bin b (Tree a b) (Tree a b)
```

The overloaded equality function for these data types can be defined in Clean as follows:

```
class eq t :: t t → Bool
instance eq (List a) | eq a where
    eq Nil Nil                              = True
    eq (Cons x xs) (Cons y ys)              = eq x y & eq xs ys
    eq x y                                  = False
instance eq (Tree a b) | eq a & eq b where
    eq (Tip x) (Tip y)                      = eq x y
    eq (Bin x lxs rxs) (Bin y lys rys)      = eq x y & eq lxs lys & eq rxs rys
    eq x y                                  = False
```

All these instances have one thing in common: they check that the data constructors of both compared objects are the same and that all the arguments of these constructors are equal. Note also that the context restrictions are needed for all the type arguments, because we call the equality functions for these types.

Another example of a type constructor class is the mapping function:

```
class fmap t :: (a → b) (t a) → (t b)
instance fmap List where
    fmap f Nil           = Nil
    fmap f (Cons x xs)   = Cons (f x) (fmap f xs)
```

The class variable of this class ranges over types of kind $\star \to \star$. In contrast, the class variable of equality ranges over types of kind $\star$. The tree type has kind $\star \to \star \to \star$. The mapping for a type of this kind takes two functions: one for each type argument.

```
class bimap t :: (a → b) (c → d) (t a c) → (t b d)
instance bimap Tree where
    bimap fx fy (Tip x)          = Tip (fx x)
    bimap fx fy (Bin y ls rs)    = Bin (fy y) (bimap fx fy ls) (bimap fx fy rs)
```

In general the mapping function for a type of arity n, takes n functions: one for each type argument. In particular, the mapping function for types of kind $\star$ is the identity function. This remark is important for section 2.3 where we define a mapping function for types of all kinds.

2.2.2 Generic Classes

In this subsection we show how to define the equality function generically, i.e. by induction on the structure of types. The user provides the generic definition of equality once. This definition can be used to produce the equality function for any specific data type. The approach described in this subsection assumes only generic definitions for classes, whose class variables range over types of kind $\star$. This is the approach described by Peyton Jones and Hinze in [HP01]. We present it here for didactic reasons. In the next section we will present our approach, based on Hinze's kind-indexed types [Hin00c], which does not have the limitation of kind $\star$.

The structure of a data type can be represented as a sum of products of types. For instance, a Clean data type

$$:: T\ a_1\ ...\ a_n = K_1\ t_{11}\ ...\ t_{1l_1} |\ ...\ | K_m\ t_{m1}\ ...\ t_{ml_m}$$

can be regarded as

$$\mathrm{T}^{\circ}\ a_1\ ...\ a_n = (t_{11} \times \ldots \times t_{1l_1}) + \ldots + (t_{m1} \times \ldots \times t_{ml_m})$$

List and Tree from the previous section can be represented as

```
List° a     = 𝟙 + a × (List a)
Tree° a b   = a + b × (Tree a b) × (Tree a b)
```

Here $\mathbb{1}$ denotes the nullary product. To encode such a representation in Clean we use the following types for binary sums and products.

```
:: UNIT         = UNIT
:: PAIR a b     = PAIR a b
:: EITHER l r   = LEFT l | RIGHT r
```

N-ary sums and products can be represented as nested binary sums and products. The UNIT type is used to represent the product of zero elements, the EITHER type is a binary sum and the PAIR type is a binary product. With these types List° and Tree° can be represented as (in Clean a synonym type is introduced with :==)

```
:: List° a     :== EITHER UNIT (PAIR a (List a))
:: Tree° a b   :== EITHER a (PAIR b (PAIR (Tree a b) (Tree a b)))
```

Note that these types are not recursive. For instance, the right hand side of List° refers to the plain List rather than to List°. So, the encoding affects only the "top-level" of a type definition. The recursive occurrences of List type are converted to List° "lazily". In this way it is easy to handle mutually recursive types (see [HP01]).

We need conversion functions to convert between a data type T and its generic representation T°. For example, the conversion functions for lists are

```
fromList :: (List a) → List° a
fromList Nil                 = LEFT UNIT
fromList (Cons x xs)         = RIGHT (PAIR x xs)
toList :: (List° a) → List a
toList (LEFT UNIT)           = Nil
toList (RIGHT (PAIR x xs))   = Cons x xs
```

Now we are ready to define the equality generically. All the programmer has to do is to specify the instances for unit, sum, product and primitive types.

```
class eq t :: t t → Bool
instance eq Int where
    eq x y                        = eqInt x y
instance eq UNIT where
    eq UNIT UNIT                  = True
instance eq (PAIR a b)      | eq a & eq b where
    eq (PAIR x1 x2) (PAIR y1 y2)  = eq x1 y1 & eq x2 y2
instance eq (EITHER a b)    | eq a & eq b where
    eq (LEFT x) (LEFT y)          = eq x y
    eq (RIGHT x) (RIGHT y)        = eq x y
    eq x y                        = False
```

This definition is enough to produce the equality functions for almost all data types: an object of a data type can be (automatically) converted to the generic representation using the conversion functions and the generic representations can be compared using the instances above. The integers are compared with the predefined function eqInt. We use integers as the only representative of primitive types. Other primitive types can be handled analogously. The UNIT type has only one inhabitant; the equality always return True. Pairs are compared component-wise. Binary sums are equal only when the constructors are equal and their arguments are equal. In general a data types may involve arrows. To handle such data types the user has to provide an instance on the arrow type (→). Since equality cannot be sensibly defined for arrows, we have omitted the instance: comparing types containing arrows will result in a compile time overloading error.

These definitions can be used to produce instances for almost all data types. For instance, when the programmer wants the equality functions to be generated for lists and trees, (s)he specifies the following

```
derive eq (List a)
derive eq (Tree a b)
```

These definitions can be used to generate the following instances:

```
instance eq (List a)     | eq a where
  eq x y     = eq (fromList x) (fromList y)
instance eq (Tree a b)   | eq a & eq b where
  eq x y     = eq (fromTree x) (fromTree y)
```

So, we can implement the equality on arbitrary types using the equality on their generic representations. It is important to note that the way we convert the arguments and the results to and from the generic representation depends on the type of the generic function. The compiler generates these conversions automatically as described in section 2.4.4.

When we try to use the same approach to define fmap generically, we have a problem. The type language has to be extended for lambda abstractions on the type level. See [Hin00b] for details. Another problem is that we need to provide different mapping functions for different kinds: like fmap for kind $\star \to \star$, bimap for kind $\star \to \star \to \star$ and so on. Both of these problems are solved by the approach with kind-indexed types [Hin00c]. In our design, described in the following section, we use this approach in combination with type constructor classes.

2.3 Generic Clean

In this section we show how generic functions can be defined and used in Clean. We use the mapping function as an example. To define the generic mapping function we write

```
generic map a1 a2 :: a1 → a2
map{|Int|} x                          = x
map{|UNIT|} x                         = x
map{|PAIR|} mapl mapr (PAIR l r)      = PAIR (mapl l) (mapr r)
map{|EITHER|} mapl mapr (LEFT l)      = LEFT (mapl l)
map{|EITHER|} mapl mapr (RIGHT r)     = RIGHT (mapr r)
```

The **generic** definition introduces the type of the generic function. The base case instance definitions map{|...|} provide the mapping for the primitive types, UNIT, PAIR and EITHER.

The reader has probably noticed that the instances do not seem to “fit” together: they take a different number arguments. The function for integers takes no additional arguments, only the integer itself. Similarly, the function for UNIT takes only the UNIT argument; mapping for types of kind $\star$ is the identity function. The functions for EITHER and PAIR take two additional arguments; mapping for types of kind $\star \to \star \to \star$ needs two additional arguments: one for each type argument. The generic definition is actually a template that generates an infinite set of mapping classes, one class per kind. So, using the definition above we have defined

```
class map_⋆ t :: t → t
class map_⋆→⋆ t :: (a1 → a2) (t a1) → (t a2)
class map_⋆→⋆→⋆ t :: (a1 → a2) (b1 → b2) (t a1 b1) → (t a2 b2)
...
```

The class for kind $\star$ has the type of the identity function. The other two classes are renamings of the fmap and bimap classes from the previous section. The instances are bound to the classes according to the kind of the instance type.

```
instance map_⋆ Int where
    map_⋆ x                                = x
instance map_⋆ UNIT where
    map_⋆ x                                = x
instance map_⋆→⋆→⋆ PAIR where
    map_⋆→⋆→⋆ mapl mapr (PAIR l r)         = PAIR (mapl l) (mapr r)
instance map_⋆→⋆→⋆ EITHER where
    map_⋆→⋆→⋆ mapl mapr (LEFT l)           = LEFT (mapl l)
    map_⋆→⋆→⋆ mapl mapr (RIGHT r)          = RIGHT (mapr r)
```

The programmer does not have to write the kind indexes, they are assigned automatically by the compiler.

For convenience we introduce a type synonym for the type specified in the **generic** definition of mapping:

$:: \ \mathsf{Map}\ a_1\ a_2 :== a_1 \to a_2$

The type of the generic mapping for a type of any kind can be computed using the following algorithm [Hin00c]:

$$\begin{aligned} &:: \ \mathsf{Map}_{\star}\ t_1\ t_2 && :== \mathsf{Map}\ t_1\ t_2 \\ &:: \ \mathsf{Map}_{k \to l}\ t_1\ t_2 && :== \forall a_1\ a_2.\ (\mathsf{Map}_k\ a_1\ a_2) \to \mathsf{Map}_l\ (t_1\ a_1)\ (t_2\ a_2) \end{aligned}$$

The mapping function for a type t of a kind k has type:

class map_k t :: Map_k t t

The type specified in a generic declaration, like Map, is called the polykinded type [Hin00c] of the generic function. We have to note that, though the type of map has two type arguments, the generated classes have only one class argument. This property holds for all generic functions: the corresponding classes always have one class argument. It remains to be researched how to extend the approach for classes with more than one argument. In this example, we use type Map_k t t with both arguments filled with the same variable t. It means that the consumed argument has the same top level structure as the produced result. We need two type variables to indicate that the structure does not have to be the same at the lower level. In the example of the reduce function at the end of this section we will give an idea about how to find the generic type of a function.

The programmer specifies which instances must be generated by the compiler. For List we write:

```
derive map List
```

The mapping for types of kind $\star \rightarrow \star$, like lists, can be used as usually, but the user now has to explicitly specify which map of the generated family of maps to apply. This is done by giving the kind between {| and |} as in

```
map{|⋆ → ⋆|} inc (Cons 1 (Cons 2 (Cons 3 Nil)))
```

Similarly, we can also get the mapping for type Tree, which is of kind $\star \rightarrow \star \rightarrow \star$.

```
instance map Tree generic
```

It can be used as in

```
map{|⋆ → ⋆ → ⋆|} inc dec (Bin 1 (Tip 2) (Tip 3))
```

In this example the values in the tips of the tree are incremented, the values in the branches of the tree are decremented. For readability reasons we will write kind indexes as subscripts from now on.

Let's go back to the equality example and see how to define generic equality in Clean:

```
generic eq t    :: t t → Bool
eq{|Int|} x y                                   = eqInt x y
eq{|UNIT|} x y                                  = True
eq{|PAIR|} eql eqr (PAIR l1 r1) (PAIR l2 r2)    = eql l1 l2 && eqr r1 r2
eq{|EITHER|} eql eqr (LEFT l1) (LEFT l2)        = eql l1 l2
eq{|EITHER|} eql eqr (RIGHT r1) (RIGHT r2)      = eqr r1 r2
eq{|EITHER|} eql eqr x y                        = False
```

In this definition, like in the definition of map, the instances have additional arguments depending on the kind of the instance type. Again, the programmer specifies the instances to be generated, say:

```
derive eq List
derive eq Tree
```

This will result in two instances: $\mathsf{eq}_{\star\to\star}$ for List and $\mathsf{eq}_{\star\to\star\to\star}$ for Tree. The equality can be used as in

```
eq⋆→⋆ eq⋆ [1,2,3] [1,2,3]
    ⇒ True
eq⋆→⋆→⋆ eq⋆ eq⋆ (Bin 1 (Tip 2) (Tip 3)) (Bin 1 (Tip 2) (Tip 3))
    ⇒ True
eq⋆→⋆ (λx y→eq⋆ (length x) (length y)) [[1,2],[3,4]]  [[1,1],[2,2]]
    ⇒ True
```

In the last line the two lists are equal if they are of the same length and the lengths of the element lists are equal.

One can see that this equality is more general than one defined in Section 2.2: the user can specify how to compare the elements of the structure. However, it is inconvenient to pass the "dictionaries" (such as $eq_\star$) manually every time. For this reason we generate additional instances that turn explicit dictionaries into implicit ones:

```
instance eq⋆ (List a) | eq a where
    eq⋆ x y = eq⋆→⋆ eq⋆ x y
instance eq⋆ (Tree a b) | eq a & eq b where
    eq⋆ x y = eq⋆→⋆→⋆ eq⋆ eq⋆ x y
```

Such instances make it possible to call

```
eq⋆ [1,2,3]  [1,2,3]
eq⋆ (Bin 1 (Tip 2) (Tip 3)) (Bin 1 (Tip 2) (Tip 3))
```

with the same effect as above.

The equality operator defined as a type class in the standard library of Clean can now be defined using the generic equality:

```
(==) infixr 5 :: t t → Bool | eq⋆ t
(==) x y = eq⋆ x y
```

Consider an application of *map*

```
map⋆→⋆ (λx → 0) [[1,2], [3,4]]
```

What would it return: [0,0] or [[0,0], [0,0]]? The overloading will always choose the first. If the second is needed, the user has to write

```
map⋆→⋆ (map⋆→⋆ (λx → 0)) [[1,2], [3,4]]
```

As one more example we show the right reduce function, which is a generalization of foldr on lists. It takes a structure of type t and an "empty" value of type b and collapses the structure into another value of type b. Thus, the type is $t \rightarrow b \rightarrow b$, where t is the structure, i.e. t is a generic variable, and b is a parametrically polymorphic variable.

```
generic rreduce t :: t b → b
rreduce{|Int|}  x e                         = e
rreduce{|UNIT|} x e                         = e
rreduce{|PAIR|} redl redr (PAIR l r) e      = redl l (redr r e)
rreduce{|EITHER|} redl redr (LEFT l) e      = redl l e
rreduce{|EITHER|} redl redr (RIGHT r) e     = redr r e
```

Reducing types of kind $\star$ just returns the "empty" value. The instance for pairs uses the result of the reduction for the second element of the pair as the "empty" argument for the reduction of the first element. To reduce the sum we reduce the arguments.

The function rreduce is an example of a parametrically polymorphic function: b is a non-generic polymorphic type variable. We can define the standard foldr function that is defined on types of kind $\star \rightarrow \star$ using rreduce.

```
foldr :: (a b → b) b (t a) → b | rreduce⋆→⋆ t
foldr op e x = rreduce⋆→⋆ op x e
```

How do we come up with the type for generic reduce knowing the type of reduce for lists (foldr)? The type of the standard foldr definition is:

```
foldr :: (a → b → b) b [a] → b
```

If it is generalized to any type of kind $\star \rightarrow \star$, it becomes

```
foldr :: (a → b → b) b (t a) → b
```

The type (t a) is the structure that we are collapsing. The first argument is the function that we apply to the elements of the structure, i.e. it is folding for type a of kind $\star$. So, we can choose the type (a → b → b) as the generic type. With this generic type we get

```
class rreduce⋆ a :: a b → b
class rreduce⋆→⋆ a :: (a1 → b → b) (a a1) b → b
class rreduce⋆→⋆→⋆ a :: (a1 → b → b) (a2 → b → b) (a a1 a2) b → b
```

The type for kind $\star \rightarrow \star$ is the same as the type of foldr, except that the last two arguments are flipped. This idea of finding out the generic type can be used for other functions that normally make sense for types of kind $\star \rightarrow \star$.

2.4 Implementation

In this section we describe how the generic definitions are translated to classes and instances of non-generic Clean. Suppose we need to specialize a generic function g to a data type T, i.e. generate an instance of g for T. A generic definition in general looks like

generic g a_1 ... a_r :: G a_1 ... a_r p_1 ... p_s

Here G is the polykinded type of the generic function g, a_i are polykinded type variables and p_i are polymorphic type variables (i.e. the function is parametrically polymorphic with respect to them). We will denote p_1 ... p_s as $\vec{p}$.

To generate the instance for a data type T the following has to be specified by the user

derive g T

A data type T has the following form

$:: \mathrm{T}\ \mathrm{a}_1\ ...\ \mathrm{a}_n = \mathrm{K}_1\ \mathrm{t}_{11}\ ...\ \mathrm{t}_{1l_1}\ |\ ...\ |\ \mathrm{K}_m\ \mathrm{t}_{m1}\ ...\ \mathrm{t}_{ml_m}$

As an example in this section we will use the generic equality function defined in the previous section and its specialization to lists

```
:: Eq a :== a → a → Bool
generic eq :: Eq a
:: List a = Nil | Cons a (List a)
derive eq List
```

Here is a short summary of what is done to specialize a generic function g to a data type T. The following subsections give more details.

- Create the class g_κ for the kind κ of the data type T, if not already created. The instance on T becomes an instance of that class (subsection 2.4.1).
- Build the generic representation T° for the type T. Also build the conversion functions between T and T° (subsection 2.4.2)
- Build the specialization of g to the generic representation T°. We have all the ingredients needed to build the specialization because the type T° is defined using sums and products. The instances for sums and products are provided by the user as part of the generic definition (subsection 2.4.3).
- The generic function is now specialized to the generic representation T°, but we need to specialize it to the type T. We generate an adaptor that converts the function for T° into the function for T (subsection 2.4.4).

- Build the specialization to the type T. It uses the specialization to T° and the adaptor. The instance g_κ on T is the specialization of the generic function g to the type T. (subsection 2.4.3).
- For convenience we additionally create shorthand instances for kind $\star$ (subsection 2.4.5).

2.4.1 Kind-indexed Classes

The polykinded type of a generic function is used to compute the type of the function for a kind using the following algorithm [Hin00c]:

$:: \ G_\star \ t_1 \ ... \ t_r \ \vec{p} \quad :== G \ t_1 \ ... \ t_r \ \vec{p}$
$:: \ G_{\kappa \to \kappa'} \ t_1 \ ... \ t_r \ \vec{p} :== \forall \ a_1 \ ... \ a_r. \ (G_\kappa \ a_1 \ ... \ a_r \ \vec{p} \ ) \to G'_\kappa \ (t_1 \ a_1) \ ... \ (t_r \ a_r) \ \vec{p}$

From now on we will use the following shorthands:

$:: \ G' \ \ t \ \vec{p} :== G \ t \ ... \ t \ \vec{p}$
$:: \ G'_\kappa \ t \ \vec{p} :== G_\kappa \ t \ ... \ t \ \vec{p}$

where icode"t" in each right hand side occurs r times.

The generic extension of Clean translates a generic definition into a family of class definitions, one class per kind. The class has one class argument of kind κ and one member. The type of the member is the polykinded type of the generic function specialized to kind κ:

class $g_\kappa \ t :: \ G'_\kappa \ t \ \vec{p}$

Unlike [Hin00c], we use the polykinded types to type the class members rather than functions.

Each instance of a generic function is bound to one of the classes according to the kind of the instance type. For our example we have so far

class $eq_{\star\to\star}$ t :: (Eq a) $\to$ Eq (t a)
instance $eq_{\star\to\star}$ List **where** $eq_{\star\to\star}$ eqa = ...

where the body of the instance is still to be generated.

2.4.2 Generic Type Representation

To specialize a generic function to a concrete data type one needs to build the generic representation of that type. This is rather straightforward. The algorithms for building the generic representation types and the conversion functions are described by Hinze in [Hin99]. The conversion functions are packed into a record defined in the generic prelude:

:: Iso a a°= {iso :: a $\to$ a°, osi :: a° $\to$ a}

Here we just give an example of the generic type representation and the isomorphism for the list type:

```
List° a :== EITHER UNIT (PAIR a (List a))
iso_List :: Iso (List a) (List° a)
iso_List = {iso=isoList,osi=osiList}
where
    isoList  Nil                     = LEFT UNIT
    isoList  (Cons x xs)             = RIGHT (PAIR x xs)
    osiList  (LEFT UNIT)             = Nil
    osiList  (RIGHT (PAIR x xs))     = Cons x xs
```

2.4.3 Specialization

In this subsection we show how to specialize a generic function g to a data type T. It is done by first specializing it to the generic type representation T°. This specialization g_{T° is then used to build the specialization g_T to the data type T. The specialization to the generic type representation T° is:

g_{T° :: G'_k T° $\vec{p}$
g_{T° v_1 ... $v_n = \mathcal{S}(g, \{a_1 := v_1, \ldots, a_n := v_n\}, T^\circ)$

The following algorithm is used to generate the right hand side by induction on the structure of the generic representation:

$\mathcal{S}(g, \mathcal{E}, a)$	$=$	$\mathcal{E}[a]$	type variables
$\mathcal{S}(g, \mathcal{E}, T)$	$=$	g_T	type constructors
$\mathcal{S}(g, \mathcal{E}, t\ s)$	$=$	$\mathcal{S}(g, \mathcal{E}, t)\ \mathcal{S}(g, \mathcal{E}, s)$	type application
$\mathcal{S}(g, \mathcal{E}, t \to s)$	$=$	$g_\to\ \mathcal{S}(g, \mathcal{E}, t)\ \mathcal{S}(g, \mathcal{E}, s)$	arrow type

Type variables are interpreted as value variables bound in the environment $\mathcal{E}$, type constructors T as instances g_T of the generic function g on the data type T, type application as value application and arrow type as application of the instance of g for the arrow type. In [Hin00c] Hinze proves that the functions specialized in this way are well-typed provided that g is well-typed. For the equality on List° the specialization is:

eq_{List° :: (Eq a) → Eq (List° a)
eq_{List° eqa = eq_{EITHER} eq_{UNIT} (eq_{PAIR} eqa (eq_{List} eqa))

The functions eq_{EITHER}, eq_{PAIR} and eq_{UNIT} are instances of the generic equality for the corresponding types. The function eq_{List} is the specialization to lists that we are generating.

The specialization to T is generated using the specialization to T°:

```
$g_T$ :: $G'_k$ T $\vec{p}$
$g_T$ $v_1$ ... $v_n$ = adaptor ($g_{T^\circ}$ $v_1$ ... $v_n$)
where
    adaptor :: (G' ($T^\circ$ $a_1$ ... $a_n$) $\vec{p}$) $\rightarrow$ G' (T $a_1$ ... $a_n$) $\vec{p}$
    adaptor = ...
```

The adaptor converts the function for T° into the function for T. The adaptors are generated using bidirectional mappings [HP01], described in the next subsection. The equality specialized to lists is

```
$eq_{List}$ :: (Eq a) $\rightarrow$ Eq (List a)
$eq_{List}$ eqa = adaptor ($eq_{List^\circ}$ eqa)
where
    adaptor :: (Eq ($List^\circ$ a)) $\rightarrow$ Eq (List a)
    adaptor = ...
```

The mutually recursive definitions of eq_{List} and eq_{List° show why we do not need type recursion in the generic type representation: the function converts lists to the generic representations as needed.

Now it is easy to fill in the instance for the type T. It is just the specialization to the type T.

```
instance $g_\kappa$ T where $g_\kappa$ = $g_T$
```

The instance of the equality for lists is:

```
instance $eq_{\star\rightarrow\star}$ List where $eq_{\star\rightarrow\star}$ = $eq_{List}$
```

2.4.4 Adaptors

Adaptors are more complicated than one would expect. The reason is that generic function types and data types may contain arrows. Since the arrow type is contravariant in the argument position, we need bidirectional mapping functions to map it [HP01]. We define bidirectional mapping by induction on the structure of types as a special generic function predefined in the compiler:

```
generic bmap a b :: Iso a b
```

It is automatically specialized to all data types in the following way

```
instance bmap T where
    bmap $v_1$ ... $v_n$ = {iso=isoT, osi=osiT}
    where
        isoT ($K_1$ $x_{11}$ ... $x_{1m_1}$)        = $K_1$ $x'_{11}$ ... $x'_{1m_1}$
        ...
        isoT ($K_m$ $x_{m1}$ ... $x_{ml_m}$)        = $K_m$ $x'_{m1}$ ... $x'_{ml_m}$
    osiT  (K₁ x₁₁ ... x₁ₘ₁)        = K₁ x″₁₁ ... x″₁ₘ₁
    ...
    osiT  (Kₘ xₘ₁ ... xₘₗₘ)        = Kₘ x″ₘ₁ ... x″ₘₗₘ
```

Here $x_{ij} :: t_{ij}$ is the jth argument of the data constructor K_i. New constructor arguments x'_{ij} and x''_{ij} are given by

$$\begin{aligned} x'_{ij} &= (\mathcal{S}(\text{bmap}, \{a_1 := v_1, \ldots, a_n := v_n\},\ t_{ij})).\text{iso}\ x_{ij} \\ x''_{ij} &= (\mathcal{S}(\text{bmap}, \{a_1 := v_1, \ldots, a_n := v_n\},\ t_{ij})).\text{osi}\ x_{ij} \end{aligned}$$

The environment passed to $\mathcal{S}$ binds the type arguments of the data type T to the corresponding function arguments. For example, the instance for lists is

```
instance bmap List where
    bmap v = {iso=isoList, osi=osiList}
    where
        isoList  Nil            = Nil
        isoList  (Cons x xs)    = Cons (v.iso x)  ((bmap_List v).iso xs)
        osiList  Nil            = Nil
        osiList  (Cons x xs)    = Cons (v.osi x)  ((bmap_List v).osi xs)
```

The instance for the arrow is predefined as

```
instance bmap (→) where
    bmap bmaparg bmapres = {iso=isoArrow, osi=osiArrow}
    where
        isoArrow  f  = bmapres.iso · f · bmaparg.osi
        osiArrow  f  = bmapres.osi · f · bmaparg.iso
```

This instance demonstrates the need for pairing the conversion functions together.

This generic function is used to build bidirectional mapping for a generic function type:

```
bmap_g :: Iso_kind(G) G ... G
bmap_g v₁ ... v_r u₁ ... u_s
        = S(bmap, {a₁ := v₁, ..., a_r := v_r, p₁ := u₁, ..., p_s := u_s}, G a⃗ p⃗)
```

The function lifts the isomorphisms for the arguments to the isomorphism for the function type. In the function type the data type Iso is used as a polykinded type. It is instantiated to the type of the generic function G. The right hand side is defined by induction on the structure of type G. For the generic equality we have

```
bmap_eq :: (Iso a a°) → (Iso (Eq a) (Eq a°))
bmap_eq v = bmap_→ v (bmap_→ v bmap_Bool)
```

Bidirectional mapping for the primitive type Bool is the identity mapping, because it has kind $\star$.

Now we can generate the body of the adaptor:

adaptor = (bmap$_g$ iso$_T$ ... iso$_T$ isoId ... isoId). osi

The a-arguments are filled in with the isomorphism for the data type T and the p-arguments with the identity isomorphism. In the current implementation ps are limited to kind $\star$, so we use the identity to map them. In our example of the equality on lists the adaptor is

adaptor = (bmap$_{\mathrm{eq}}$ iso$_{\mathrm{List}}$).osi

2.4.5 Shorthand Instances for Kind $\star$

For each instance on a type of a kind other than $\star$ a shorthand instance for kind $\star$ is created. Consider the instance of a generic function g for a type T a$_1$ ... a$_n$, $n \geq 1$. The kind κ of the type T is $\kappa = \kappa_1 \rightarrow ... \rightarrow \kappa_n \rightarrow \star$.

```
instance g$_\star$ (T a$_1$ ... a$_n$) | g$_{\kappa_1}$ a$_1$ & ... & g$_{\kappa_n}$ a$_n$
where
    g$_\star$ = g$_\kappa$ g$_{\kappa_1}$ ... g$_{\kappa_n}$
```

For instance, for the equality on lists and trees we have

```
instance eq$_\star$ [a] | eq$_\star$ a where
    eq$_\star$ x y = eq$_{\star\rightarrow\star}$ eq$_\star$ x y
instance eq$_\star$ Tree a b | eq$_\star$ a & eq$_\star$ b where
    eq$_\star$ x y = eq$_{\star\rightarrow\star\rightarrow\star}$ eq$_\star$ eq$_\star$ x y
```

These instances make it is possible to call

eq$_\star$ [1,2,3] [1,2,3]

instead of

eq$_{\star\rightarrow\star}$ eq$_\star$ [1,2,3] [1,2,3]:

they turn explicit arguments into dictionaries of the overloading system.

2.5 Customized Instances

Generic functions can be defined to perform a specific task on objects of a specific data type contained in any data structure. Such generic functions have the big advantage that they are invariant with respect to changes in the data structure.

Let's for example consider terms in a compiler.

```
:: Expr
    = ELambda Var Expr
    | EVar Var
    | EApp Expr Expr
:: Var  =  Var String
```

We can define a generic function to collect free variables in any data structure (e.g. parse tree):

```
generic fvs t :: t → [Var]
fvs{|Int|} x                           = []
fvs{|UNIT|} x                          = []
fvs{|PAIR|} fvsl fvsr (PAIR l r)       = removeDup(fvsl l ++ fvsl r)
fvs{|EITHER|} fvsl fvsr (LEFT l)       = fvsl l
fvs{|EITHER|} fvsl fvsr (RIGHT r)      = fvsr r
fvs{|Var|} x                           = [x]
fvs{|Expr|} (ELambda var expr)         = removeMember var (fvs⋆ expr)
fvs{|Expr|} (EVar var)                 = fvs⋆ var
fvs{|Expr|} (EApp fun arg)             = removeDup(fvs⋆ fun ++ fvs⋆ arg)
```

UNITs and Ints do not contain variables, so the instances return empty lists. For pairs the variables are collected in both components; the concatenated list is returned after removing duplicates. For sums the variables are collected in the arguments. The instance on Var returns the variable as a singleton list. For lambda expressions we collect variables in the lambda body and filter out the lambda variable. For variables we call the instance on variables. For applications we collect the variables in the function and in the argument and return the concatenated list.

Now, if the structure containing expressions (e.g. a parse tree) changes, the same generic function can still be used to collect free variables in it. But if the expression type itself changes we have to modify the last instance of the function accordingly. Let's have a closer look at the last instance. Only the first alternative does something special - it filters out the bound variables. The other two alternatives just collect free variables in the arguments of the data constructors. Thus, except for lambda abstractions, the instance behaves as if it was generated by the generic extension. The generic extension of CLEAN provides a way to deal with this problem. The user can refer to the generic implementation of an instance that (s)he provides. In the example the instance on Expr can be written more compactly:

```
fvs{|Expr|} (ELambda var expr)      = removeMember var (fvs⋆ expr)
fvs{|Expr|} x                       = fvs{|generic|} x
```

The name fvs{|**generic**|} is bound to the generic implementation of the instance in which it occurs. The code generated for the instance on Expr is:

$\mathsf{fvs}^{g}_{\mathrm{Expr}}$ x = (bmap$_{\mathrm{fvs}}$ iso$_{\mathrm{Expr}}$).osi (fvs$_{\mathrm{Expr}^\circ}$ x)
fvs$_{\mathrm{Expr}}$ (ELambda var expr) = removeMember var (fvs$_{\mathrm{Expr}}$ expr)
fvs$_{\mathrm{Expr}}$ x = $\mathsf{fvs}^{g}_{\mathrm{Expr}}$ x

Here $\mathsf{fvs}^{g}_{\mathrm{Expr}}$ denotes the function generated for fvs{|**generic**|}. The function for the generic representation fvs$_{\mathrm{Expr}^\circ}$ is generated as usually.

2.6 Related Work

Generic Haskell is an extension for Haskell based on the approach of kind-indexed types, described in [Hin00c]. Despite pretty different notation, generic definitions in Generic Haskell and Clean are similar. The user provides the polykinded type and cases for sums, products, unit, arrow and primitive types. In Generic Haskell an overloaded function cannot be defined generically. This means that, for instance, the equality operator (==) has to be defined manually. In Clean overloaded functions are supported. For instance, the equality operator in Clean can be defined in terms of the generic function *eq*:

(==) **infixr** 5 :: t t → Bool | eq$_\star$ t
(==) x y = eq$_\star$ x y

Currently Generic Haskell does not support the module system. Clean supports the module system for generics in the same way as it does it for overloaded functions.

Glasgow Haskell supports generic programming as described in [HP01]. In GHC generic definitions are used to define default implementation of class members, giving systematic meaning to the *deriving* construct. Default methods can be derived for type classes whose class argument is of kind $\star$. That means that functions like mapping cannot be defined generically. In Clean a *generic* definition provides default implementation for methods of a kind-indexed family of classes. For instance, it possible in Clean to customize how elements of lists are compared:

eq$_{\star\to\star}$ (λx y→eq$_\star$ (length x) (length y)) [[1,2],[3,4]] [[1,1], [2,2]]
    ⇒ True

This cannot be done in GHC, since the equality class is defined for types of kind $\star$. In Clean one generic definition is enough to generate functions for all (currently up to second-order) kinds. This is especially important for functions like mapping.

In [CA01] Chen and W. Appel describe an approach to implement specialization of generic functions using dictionary passing. Their work is at the intermediate language level; our generic extension is a user level facility. Our implementation is based on type classes that are implemented using dictionaries. In SML/NJ the kind system of the language is extended, which we do not require.

PolyP [JJ97] is a language extension for Haskell. It is a predecessor of Generic Haskell. PolyP supports a special *polytypic* construct, which is similar to our *generic* construct. In PolyP, to specify a generic function one needs to provide two additional cases: for type application and for type recursion. PolyP generic functions are restricted to work on regular types. A significant advantage of PolyP is that recursion schemes like catamorphisms (folds) can be defined. It remains to be seen how to support such recursion schemes in Clean.

In [LVK00] Lämmel, Visser and Kort propose a way to deal with generalized folds on large systems of mutually recursive data types. The idea is that a fold algebra is separated in a basic fold algebra and updates to the basic algebra. The basic algebras model generic behavior, whereas updates to the basic algebras model specific behavior. Existing generic programming extensions, including ours, allow for type indexed functions, whereas their approach needs type-indexed algebras. Our customized instances (see section 2.5) provide a simple solution for dealing with type-preserving (map-like) algebras (see [LVK00]). To support type-unifying (fold-like) algebras we need more flexible encoding of the generic type representation.

2.7 Conclusions and Future Work

In this paper we have presented a generic extension for Clean that allows to define overloaded functions with class variables of any kind generically. A generic definition generates a family of kind-indexed type (constructor) classes, where the class variable of each class ranges over types of the corresponding kind. For instance, a generic definition of map defines overloaded mapping functions for functors, bifunctors etc. Our contribution is in extending the approach of kind-indexed types [Hin00c] with overloading.

Additionally, we have presented an extension that allows for customization of generated instances. A custom instance on a type may refer to the generated function for that type. With this feature a combination of generic and specific behavior can be expressed.

Currently our prototype lacks optimization of the generated code. The overhead introduced by the generic representation, the conversion functions and the adaptors is in most cases unacceptable. But we are convinced that a partial evaluator can optimize out this overhead and yield code comparable with hand-written one. Our group is working on such an optimizer.

Generic Clean currently cannot generate instances on array types and types of a kind higher then order 2. Class contexts in polykinded types are not yet supported. To support pretty printers and parsers the data constructor information has to be stored in the generic type representation. The current prototype has a rudimentary support for uniqueness typing. Uniqueness typing in polykinded types must be formalized and implemented in the compiler. As noted in section 2.6 our design does not support recursion schemes like catamorphisms. We plan to add the support in the future.

Acknowledgements. We are grateful to Sjaak Smetsers for fruitful discussions and comments on this paper. For helpful comments we are also grateful to Peter Achten, Marko van Eekelen and three anonymous referees. We would like to thank Ralf Hinze for the discussion of customized instances.

Chapter 3

When Generic Functions Use Dynamic Values

Dynamic types allow strongly typed programs to link in external code *at run-time* in a type safe way. Generic programming allows programmers to write code schemes that can be specialized *at compile-time* to arguments of arbitrary type. Both techniques have been investigated and incorporated in the pure functional programming language Clean. Because generic functions work on all types and values, they are the perfect tool when manipulating dynamic values. But generics rely on compile-time specialization, whereas dynamics rely on run-time type checking and linking. This seems to be a fundamental contradiction. In this paper we show that the contradiction does not exist. From any generic function we derive a function that works on dynamics, and that can be parameterized with a dynamic type representation. Programs that use this technique combine the best of both worlds: they have concise universal code that can be applied to any dynamic value regardless of its origin. This technique is important for application domains such as type-safe mobile code and plug-in architectures.

3.1 Introduction

In this paper we discuss the interaction between two recent additions to the pure, lazy, functional programming language Clean 2.0(.1) [BvEvL$^+$87, NSvEP91, PE93]:

Dynamic types Dynamic types allow strongly typed programs to link in external code (*dynamics*) *at run-time* in a type safe way. Dynamics can be used anywhere, regardless from the module or even application that created them. Dynamics are important for type-safe applications with *mobile code* and *plug-in* architectures.

Generic programming enables us to write general function schemes that work for any data type. From these schemes the compiler can derive automatically any required instance of a specific type. This is possible because of Clean's strong type system. Generic programs are a compact way to elegantly deal with an important class of algorithms. To name a few, these are *comparison*, *pretty printers*, *parsers*.

In order to apply a generic function to a dynamic value in the current situation, the programmer should do an exhaustive type pattern-match on all possible dynamic types. Apart from the fact that this is impossible, this is at odds with the key idea of generic programming in which functions *do* an exhaustive distinction on types, but on their finite (and small) structure.

One would imagine that it is alright to apply a generic function to any dynamic value. Consider for instance the application of the generic equality function to two dynamic values. Using the built-in dynamic type unification, we can easily check the equality of the *types* of the dynamic values. Now using a generic equality, we want to check the equality of the *values* of these dynamics. In order to do this, we need to know at *compile-time* of which type the instance of the generic equality should be applied. This is not possible, because the type representation of a dynamic is only known at *run-time*.

We present a solution that uses the current implementation of generics and dynamics. The key to the solution is to guide a generic function through a dynamic value using an explicit type representation of the dynamic value's type. This guide function is predefined once. The programmer writes generic functions as usual, and in addition provides the explicit type representation.

The solution can be readily used with the current compiler if we assume that the programmer includes type representations with dynamics. However, this is at odds with the key idea of dynamics because these already store type representations with values. We show that the solution also works for conventional dynamics if we provide a low-level access function that retrieves the type representation of any dynamic.

Contributions of this paper are:

- We show how one can combine generics and dynamics in one single framework in accordance with their current implementation in the compiler.

- We argue that, in principle, the type information available in dynamics is enough, so we do not need to store extra information, and instead work with conventional dynamics.
- Programs that exploit the combined power of generics and dynamics are universally applicable to dynamic values. In particular, the code handles dynamics in a generic way without precompiled knowledge of their types.

In this paper we give introductions to dynamics (Section 3.2) and generics (Section 3.3) with respect to core properties that we rely on. In Section 3.4 we show our solution that allows the application of generic functions to dynamic values. An example of a generic pretty printing tool is given to illustrate the expressive power of the combined system (Section 3.5). We present related work (Section 3.6), our current and future plans (Section 3.7), and conclude (Section 3.8).

3.2 Dynamics in Clean

The Clean system has support for *dynamics* in the style as proposed by Pil [Pil98, Pil04]. Dynamics serve two major purposes:

Interface between static and run-time types: Programs can convert values from the statically typed world to the dynamically typed world and back without loss of type security. Any Clean expression e that has (verifiable or inferable) type t can be formed into a value of type Dynamic by: **dynamic** e :: t or: **dynamic** e[1].

Here are some examples:

```
toDynamic :: [Dynamic]
toDynamic = [e1, e2, e3, dynamic [e1,e2,e3]]
where
    e1 = dynamic 50 :: Int
    e2 = dynamic reverse :: ∀a: [a] → [a]
    e3 = dynamic reverse ['a'..'z'] :: [Char]
```

Any Dynamic value can be matched in function alternatives and case expressions. A 'dynamic pattern match' consists of an expression pattern *e-pat* and a type pattern *t-pat* as follows: (e−pat :: t−pat). Examples are:

[1]Note that :: binds tighter than the application of **dynamic**.

```
dynApply :: Dynamic Dynamic → Dynamic
dynApply (f::a → b) (x::a) = dynamic (f x) :: b
dynApply _ _   = abort "dynApply: arguments of wrong type."

dynSwap :: Dynamic → Dynamic
dynSwap ((x,y) :: (a,b))  = dynamic (y,x) :: (b,a)
```

It is important to note that *unquantified* type pattern variables (a and b in dynApply and dynSwap) do not indicate polymorphism. Instead, they are bound to (unified with) the offered type, and range over the full function alternative. The dynamic pattern match fails if unification fails.

Finally, *type-dependent* functions are a flexible way of *parameterizing* functions with the type to be matched in a dynamic. Type-dependent functions are overloaded in the TC class, which is a built-in class that basically represents all *type codeable* types. The overloaded argument can be used in a dynamic type pattern by postfixing it with ^. Typical examples that are also used in this paper are the packing and unpacking functions:

```
pack :: a → Dynamic | TC a
pack x = dynamic x::a^

unpack :: Dynamic → a | TC a
unpack (x::a^)  = x
unpack _        = abort "unpack: argument of wrong type."
```

Serialization: At least as important as switching between compile-time and run-time types, is that dynamics allow programs to *serialize* and *deserialize* values without loss of type security. Programs can work safely with data and code that do not originate from themselves.

Two library functions store and retrieve dynamic values in named files, given a proper unique environment that supports file I/O:

```
writeDynamic :: String Dynamic *env → (Bool, *env) | FileSystem env
readDynamic  :: String *env → (Bool,Dynamic,*env)} | FileSystem env
```

Making an effective and efficient implementation is hard work and requires careful design and architecture of the compiler and run-time system. It is not our intention to go into any detail of such a project, as these are presented in [VP02]. What needs to be stressed in the context of this paper is that dynamic values, when read in from disk, contain a binary representation of a complete Clean computation graph, a representation of the compile-time type, and references to the related rewrite rules. The programmer has no means of access to these representations other than those explained above.

At this stage, the Clean 2.0.1 system restricts the use of dynamics to *basic*, *algebraic*, *record*, *array*, and *function* types. Very recently, support for polymorphic functions has been added. Overloaded types and overloaded functions have been investigated by Pil [Pil04]. Generics obviously haven't been taken into account, and that is what this paper addresses.

3.3 Generics in Clean

The Clean approach to generics [AP02] combines the polykinded types approach developed by Hinze [Hin00c] and its integration with overloading as developed by Hinze and Peyton Jones [HP01]. A generic function basically represents an infinite set of overloaded classes. Programs define for which types instances of generic functions have to be generated. During program compilation, all generic functions are converted to a finite set of overloaded functions and instances. This part of the compilation process uses the available compile-time type information.

As an example, we show the generic definition of the ubiquitous equality function. It is important to observe that a generic function is defined in terms of *both* the type *and* the value. The signature of equality is:

```
generic gEq a :: a a → Bool
```

This is the type signature that has to be satisfied by an instance for types of kind $\star$ (such as the basic types Boolean, Integer, Real, Character, and String). The generic implementation compares the values of these types, and simply uses the standard overloaded equality operator ==. In the remainder of this paper we only show the Integer case, as the other basic types proceed analogously.

```
gEq{|Int|} x y = x == y
```

Algebraic types are constructed as sums of pairs – or the empty unit pair – of types. It is useful to have information (name, arity, priority) about data constructors. For brevity we omit record types. The data types that represent sums, pairs, units, and data constructors are collected in the module StdGeneric.dcl:

```
:: EITHER a b      = LEFT a | RIGHT b
:: PAIR a b        = PAIR a b
:: UNIT          = UNIT
:: CONS a        = CONS a
```

The built-in function type constructor $\rightarrow$ is reused. The kinds of these cases

$$\begin{array}{lll} \mathsf{EITHER}, \mathsf{PAIR}, (\rightarrow) & : & \star \rightarrow \star \rightarrow \star \\ \mathsf{UNIT} & : & \star \\ \mathsf{CONS} & : & \star \rightarrow \star \end{array}$$

determine the number and type of the higher-order function arguments of the generic function definition. These are used to compare the sub structures of the arguments.

```
gEq{|UNIT|} UNIT UNIT                            = UNIT
gEq{|PAIR|} fx fy  (PAIR x1 y1) (PAIR x2 y2)     = fx x1 x2 && fy y1 y2
gEq{|EITHER|} fx fy (LEFT x1) (LEFT x2)          = fx x1 x2
gEq{|EITHER|} fx fy (RIGHT y1) (RIGHT y2)        = fy y1 y2
gEq{|EITHER|} fx fy _ _                          = False
gEq{|CONS|} f (CONS x) (CONS y)                  = f x y
```

The only case that is missing here is the function type $\rightarrow$, as one cannot define a feasible implementation of function equality.

Programs must ask explicitly for an instance on type T of a generic function g by:

```
derive g T
```

This provides the programmer with a *kind-indexed* family of functions $\mathsf{g}_\star$, $\mathsf{g}_{\star\rightarrow\star}$, $\mathsf{g}_{\star\rightarrow\star\rightarrow\star}$, ... The function g_κ is denoted as: $\mathsf{g}\{|\ \kappa\ |\}$. The programmer can parameterize g_κ for any $\kappa \neq \star$ to customize the behavior of g. As an example, consider the standard binary tree type

```
:: Tree a = Leaf | Node (Tree a) a (Tree a)
```

and let

```
a = Node Leaf 5 (Node Leaf 7 Leaf)
b = Node Leaf 2 (Node Leaf 4 Leaf)
```

The expression (gEq{|∗|} a b) applies integer equality to the elements and hence yields false, but (gEq{|∗→∗|} (λ _ _ → True) a b) applies the binary constant true function, and yields true.

3.4 Generics + Dynamic in Clean

In this section we show how we made it possible for programs to manipulate *dynamics* by making use of *generic* functions. Suppose we want to apply the generic equality function gEq of Section 3.3 to two dynamics, as mentioned in Section 3.1. One would expect the following definition to work:

```
dynEq :: Dynamic Dynamic → Bool // this code is incorrect
dynEq (x::a) (y::a) = gEq{|*|} x y
dynEq _ _           = False
```

However, this is not the case because at compile-time it is impossible to check if the required instance of gEq exists, or to derive it automatically simply because of the absence of the proper compile-time type information.

In our solution, the programmer has to write:

```
dynEq :: Dynamic Dynamic → Bool // this code is correct
dynEq x=:(_::a) y=:(_::a)       = _gEq (dynTypeRep x) x y
dynEq _ _                       = False
```

Two new functions have come into existence: _gEq and dynTypeRep. The first is a function of type Type Dynamic Dynamic → Bool that can be derived automatically from gEq (in Clean, identifiers are not allowed to start with _, so this prevents accidental naming conflicts); the second is a predefined low-level access function of type Dynamic → Type. The type Type is a special dynamic that contains a type representation, and is explained below. The crucial difference with the incorrect program is that _gEq works on the complete dynamic.

We want to stress the point that the programmer only needs to write the generic function gEq as usual *and* the dynEq function. All other code can, in principle, be generated automatically. However, this is not currently incorporated, so for the time being this code (_gEq) needs to be included manually. The remainder of this section is completely devoted to explaining what code needs to generated. Function and type definitions that can be generated automatically are *italicized.*

The function _gEq is a function that *specializes* gEq to the type τ of the content of its dynamic argument. We show that specialization can be done by a single function specialize that is parameterized with a generic function and a type, and that returns the instance of the generic function for the given type, packed in a dynamic. We need to pass types and generic functions to specialize, but neither are available as values. Therefore, we must first make suitable representations of types (Section 3.4.1) and generic functions (Section 3.4.2).

We encode types with a new type (TypeRep τ) and pack it in a Dynamic with synonym definition Type such that all values (t :: TypeRep τ) :: Type satisfy the invariant that t is the type representation of τ. We wrap generic functions into a record of type GenRec that basically contains all of its specialized instances to basic types and the generic constructors *sum*, *pair*, *unit*, and *arrow*. Now specialize :: GenRec Type → Dynamic (Section 3.4.3) yields the function that we want to apply to the content of dynamics, but it is still packed in a dynamic. We show that for each generic function there is a transformer function that applies this encapsulated function to dynamic arguments (Section 3.4.5). For our gEq case, this is _gEq.

In Section 3.4.6 we show that *specialization* is sufficient to handle all generic and non-generic functions on dynamics. However, it forces programmers to work with dynamics that are extended with the proper Type. An elegant solution is obtained with the low-level access function dynTypeRep which retrieves Types from dynamics, and can therefore be used instead (Section 3.4.7).

The remainder of this section fills in the details of the scheme as sketched above. We continue to illustrate every step with the gEq example. When speaking in general terms, we assume that we have a function g that is generic in argument a and has type (G a). So in our example g = gEq, and G = Eq defined as

```
:: Eq a :== a a → Bool
```

We will have a frequent need for conversions from a type a to a type b and vice versa. These are conveniently combined into a record of type Bimap a b (see subsection 3.4.4 for its type definition and the standard bimaps that we use).

3.4.1 Dynamic type representations

Dynamic type representations are dynamics of synonym type Type containing values t :: TypeRep τ such that t represents τ, with TypeRep defined as:

```
:: TypeRep t
   = TRInt | TRUnit | TREither Type Type | TRPair Type Type
   | TRArrow Type Type
   | TRCons String Int Type
   | TRType
       [Type]   // to contain TypeRep a₁, ..., TypeRep aₙ
       Type     // to contain TypeRep (T° a₁ ... aₙ)
       Dynamic  // to contain bimap (T a₁ ... aₙ) (T° a₁ ... aₙ)
```

For each data constructor (TRC $t_1 \ldots t_n$) ($n \leq 0$) we provide a n-ary *constructor* function tr$C$ of type Type ... Type →Type that assembles the corresponding alternative, and establishes the relation between representation and type. For basic types and the cases that correspond with generic representations (*sum*, *pair*, *unit*, and *arrow*), these are straightforward and proceed as follows:

```
trInt  :: Type
trInt  = dynamic TRInt :: TypeRep Int

trEither  :: Type Type → Type
trEither  tra=:(_ :: TypeRep a) trb=:(_ :: TypeRep b)
    = dynamic (TREither tra trb) :: TypeRep (Either a b)

trArrow :: Type Type → Type
trArrow tra=:(_ :: TypeRep a) trb=:(_ :: TypeRep b)
    = dynamic (TRArrow tra trb) :: TypeRep (a → b)
```

These constructors enable us to encode the structure of a type. However, some generic functions, like a pretty printer, need type specific information about the type, such as the name and the arity. Suppose we have a type constructor $T\ a_1 \ldots a_n$ with a data constructor $C\ t_1 \ldots t_m$. The `TRCons` alternative collects the *name* and *arity* of its data constructor. We have omitted the fixity for simplicity. This is the same information a programmer might need when handling the `CONS` case of a generic function (although in the generic equality example we had no need for it).

```
trCons :: String Int Type → Type
trCons name arity tra=:(_ :: TypeRep a)
    = dynamic (TRCons name arity tra) :: TypeRep (CONS a)
```

The last alternative TRType with the constructor function

```
trType :: [Type] Type Dynamic → Type
trType args tg=:(_::TypeRep t°) conv=:(_::Bimap t t°)
    = dynamic (TRType args tg conv) :: TypeRep t
```

is used for custom types. The first argument args stores type representations (TypeRep a_i) for the type arguments a_i. These are needed for generic dynamic function application (Section 3.4.6). The second argument is the type representation for the sum-product type $T^\circ\ a_1\ ...\ a_n$ needed for generic specialization (Section 3.4.3). The last argument conv stores the conversion functions between $T\ a_1\ \ldots\ a_n$ and $T^\circ\ a_1\ \ldots\ a_n$ needed for specialization.

The type representation of a recursive type is a recursive term. For instance, the Clean *list* type constructor is defined internally as

```
:: [] a = _Cons a [a] | _Nil
```

Generically speaking it is a *sum* of: *(a)* the data *cons*tructor (_Cons) of the *pair* of the element type and the list itself, and *(b)* the data *cons*tructor (_Nil) of the *unit*. The sum-product type for list (as in standard static generics) is

```
:: List° a :== EITHER (CONS (PAIR a [a])) (CONS UNIT)
```

Note that List° is not recursive: it refers to [], not List°. Only the top-level of the type is converted into generic representation. This way it is easier to handle mutually recursive data types. The generated type representation, trList° reflects its structure on the term level:

```
trList° :: Type → Type
trList° tra = trEither
    (trCons "_Cons" 2 (trPair tra ( trList tra )))                    // (a)
    (trCons "λ_Nil" 0 trUnit)                                          // (b)
```

The type representation for [] is defined in terms of List°; trList and trList° are mutually recursive:

```
trList :: Type → Type
trList tra=:(_::TypeRep a)
    = trType [tra] ( trList° tra) (dynamic epList :: Bimap [a] ( List° a))
where
    epList = { map_to = map_to, map_from = map_from }
    map_to   [x:xs]                     = LEFT (CONS (PAIR x xs))
    map_to   []                         = RIGHT (CONS UNIT)
    map_from (LEFT (CONS (PAIR x xs)))    = [x:xs]
    map_from (RIGHT (CONS UNIT))          = []
```

As a second example, we show the dynamic type representation for our running example, the equality function which has type Eq a:

```
trEq :: Type → Type
trEq tra=:(_::TypeRep a) = trArrow tra (trArrow tra trBool)
```

3.4.2 First-class generic functions

In this section we show how to turn a generic function g, that really is a compiler scheme, into a first-class value genrec$g$ :: GenRec that can be passed to the specialization function. The key idea is that for the specialization function it is sufficient to know what the generic function would do in case of basic types, the generic cases *sum*, *pair*, *unit*, and *arrow*, and for custom types. For instance, for Integers, we need $g\{|*|\}$ :: G Int, and for *pairs*, this is

```
g{|*→*→*|} :: ∀a b: (G a) (G b) → G (PAIR a b)
```

These instances *are* functions, and hence we can collect them, packed as dynamics, in a record of type GenRec. We make essential use of dynamics, and their ability to hold polymorphic functions. (The compiler will actually *inline* the corresponding right-hand side of g.) The generated code for gEq is:

```
genrecgEq :: GenRec
genrecgEq =
{ genConvert  = dynamic convertEq       // Section 3.4.3
, genType     = trEq                    // Section 3.4.1
, genInt      = dynamic gEq{|*|} :: Eq Int
, genUNIT     = dynamic gEq{|*|} :: Eq UNIT
, genPAIR     = dynamic gEq{|*→*→*|} :: ∀a b: (Eq a) (Eq b) → Eq (PAIR a b)
, genEITHER   = dynamic gEq{|*→*→*|} :: ∀a b: (Eq a) (Eq b) → Eq (EITHER a b)
, genARROW    = dynamic gEq{|*→*→*|} :: ∀a b: (Eq a) (Eq b) → Eq (a → b)
, genCONS     = λn a → dynamic gEq{|*→*|} :: ∀a: (Eq a) → Eq (CONS a)
}
```

3.4.3 Specialization of first-class generics

In Section 3.4.1 we have shown how to construct a representation t of any type τ, packed in the dynamic (t::TypeRep τ)::Type. We have also shown in Section 3.4.2 how to turn any generic function g into a record genrec$g$ :: GenRec that can be passed to functions. This puts us in the position to provide a function, called `specialize`, that takes such a generic function representation and a dynamic type representation, and that yields $g :: G\ \tau$, packed in a conventional dynamic. This function has type GenRec Type → Dynamic. Its definition is a case distinction based on the dynamic type representation. The basic types and the generic *unit* case are easy:

```
specialize genrec (TRInt :: TypeRep Int) = genrec.genInt
specialize genrec (TRUnit :: TypeRep UNIT) = genrec.genUNIT
```

The generic case for sums contains a function of the type (G a) → (G b) → G (EITHER a b). When specializing to EITHER a b (i.e. the type representation passed to specialize is TREither tra trb), we have to get a function of type G (EITHER a b) from functions of types G a and G b obtained by applying specialize to the type representations of a and b. Note that for recursive types the specialization process will be called recursively.

```
specialize genrec ((TREither tra trb) :: TypeRep (EITHER a b))
    = applyGenCase2
        (genrec.genType tra) (genrec.genType trb)
        genrec.genEither
        ( specialize genrec tra) ( specialize genrec trb)
```

```
applyGenCase2 :: Type Type Type Dynamic Dynamic → Dynamic
applyGenCase2 (trga :: TypeRep ga)
              (trgb :: TypeRep gb)
              (gtab :: ga gb → gtab)
              dga dgb
    = dynamic gtab (unwrapTR trga dga) (unwrapTR trgb dgb) :: gtab

unwrapTR :: (TypeRep a) Dynamic → a | TC a
unwrapTR _ (x :: a^) = x
```

The first two arguments of `applyGenCase2` are type representations for G a and G b. The following argument is, in this case, the generic case for EITHER of type (G a) → (G b)→ G (EITHER a b). The last two arguments are the specializations of the generic function to types a and b. Note, that applyGenCase2 may not be strict in the last two arguments, otherwise it would lead to non-termination on recursive types, forcing recursive calls to specialize . In principle it is possible to extract the type representations (the first two arguments) from the last two arguments. However, in this case the last two arguments would become strict due to dynamic pattern match needed to extract the type information and, therefore, cause nontermination. Cases for products, arrows and constructors are handled analogously.

The case for TRType handles specialization to custom data types, e.g. [Int]. Arguments of such types have to be converted to their generic representations; results have to be converted back from the generic representation. This is done by means of bidirectional mappings. The bimap ep between a and a° needs to be lifted to the bimap between (G a) and (G a°). This conversion is done by convertG below, and is also included in the generic representation of g in the genConvert field (Section 3.4.2). dynApply2 is the 2-ary version of dynApply.

```
specialize genrec ((TRType args tra° ep) :: TypeRep a)
    = dynApply2 genrec.genConvert ep (specialize genrec tra°)
```

The definition of convertG has a standard form, namely:

```
convertG :: (Bimap a b) (G a) → (G b)
convertG ep = (bimapG ep).map_from
```

The function body of (bimapG a) is derived from the structure of the type term G a :

$$\text{bimap}G\ a = \langle G\ a \rangle$$

defined as:

$$\begin{array}{lcll}
\langle x \rangle & = & x & (\textit{type variables, including } a) \\
\langle t_1 \rightarrow t_2 \rangle & = & \langle t_1 \rangle \longrightarrow \langle t_2 \rangle & \\
\langle c\ t_1 \ldots t_n : \kappa \rangle & = & \text{bimapId} & \textbf{if } a \notin \bigcup \mathit{Var}(t_i) (n \geq 0) \\
 & = & \text{bimapId}\{|\kappa|\}\ \langle t_1 \rangle \ldots \langle t_n \rangle & \textbf{otherwise}
\end{array}$$

Var yields the variables of a type term. The generated code for convertEq and bimapEq is:

```
convertEq :: (Bimap a b) → (Eq b) → (Eq a)
convertEq ep = (bimapEq ep).map_from

bimapEq :: (Bimap a b) → Bimap (a → a → Bool) (b → b → Bool)
bimapEq ep = ep ⟶ ep ⟶ bimapId
```

where the code for (⟶) and bimapId is given below.

3.4.4 Bimap combinators

A (Bimap a b) is a pair of two conversion functions of type a → b and b → a. The trivial *Bimap*s bimapId and bimapDynamic are predefined:

```
:: Bimap a b = { map_to :: a → b, map_from :: b → a }

bimapId :: Bimap a a
bimapId = { map_to = id, map_from = id }

bimapDynamic :: Bimap a Dynamic | TC a
bimapDynamic = { map_to = pack, map_from = unpack } // (Section 3.2)
```

The bimap combinator inv swaps the conversion functions of a bimap, oo forms the sequential composition of two bimaps, and ⟶ obtains a functional bimap from a domain and range bimap.

```
inv  :: (Bimap a b) → Bimap b a
inv {map_to, map_from} = {map_to = map_from, map_from = map_to}

(oo) infixr 9 :: (Bimap b c) (Bimap a b)  → Bimap a c
(oo) f g =
    { map_to        = f.map_to    ∘ g.map_to
    , map_from      = g.map_from  ∘ f.map_from
    }

(⟶) infixr 0 :: (Bimap a b) (Bimap c d) → Bimap (a → c) (b → d)
(⟶) x y =
    { map_to        = λf → y.map_to   ∘ f ∘ x.map_from}
    , map_from       = λf → y.map_from ∘ f ∘ x.map_to}
    }
```

3.4.5 Generic dynamic functions

In the previous section we have shown how the specialize function uses a dynamic type representation as a 'switch' to construct the required generic function *g*, packed in a dynamic. We now transform such a function into the function _g :: Type → (G Dynamic), that can be used by the programmer. This function takes the same dynamic type representation argument as specialize . Its body invariably takes the following form (bimapDynamic and inv are included in Appendix 3.4.4):

```
_g :: Type → G Dynamic
_g tr = case specialize genrecg tr of
    (f :: G a) → convertG (inv bimapDynamic) f
```

As discussed in the previous section, convertG transforms a (Bimap a b) to a conversion function of type (G b) → (G a). When applied to

(inv bimapDynamic) :: (Bimap Dynamic a),

it results in a conversion function of type (G a) → (G Dynamic). This is applied to the packed generic function f :: G a, so the result function has the desired type (G Dynamic).

When applied to our running example, we obtain:

```
_gEq :: Type → Eq Dynamic
_gEq tr = case specialize genrecgEq tr of
    (f :: Eq a) → convertEq (inv bimapDynamic) f
```

3.4.6 Applying generic dynamic functions

The previous section shows how to obtain a function _g from a generic function *g* of type (G a) that basically applies *g* to dynamic arguments, assuming that these arguments internally have the same type a. In this section we show that with this function we can handle all generic and non-generic functions on dynamics. In order to do so, we require the programmer to work with *extended* dynamics, defined as:

```
:: DynamicExt = DynExt Dynamic Type
```

An extended dynamic value (DynExt $(v{::}\tau)$ $(t{::}\mathsf{TypeRep}\ \tau)$) basically is a pair of a *conventional* dynamic $(v{::}\tau)$ *and* its dynamic type representation $(t{::}\mathsf{TypeRep}\ \tau)$. Note that we make effective use of the built-in unification of dynamics to enforce that the dynamic type representation really is the same as the type of the conventional dynamic.

For the running example gEq we can now write an equality function on extended dynamics, making use of the generated function _gEq:

```
dynEq :: DynamicExt DynamicExt → Bool
dynEq (DynExt x=:(_::a) tx) (DynExt y=:(_::a) _)        = _gEq tx x y
dynEq _ _                                                = False
```

It is the task of the programmer to handle the cases in which the (extended) dynamics do not contain values of the proper type. This is an artefact of dynamic programming, as we can never make assumptions about the content of dynamics.

Finally, we show how to handle *non-generic* dynamic functions, such as the dynApply and dynSwap in Section 3.2. These examples illustrate that it is possible to maintain the invariant that extended dynamics always have a dynamic type representation of the type of the value in the corresponding conventional dynamic. It should be observed that these non-generic functions are basically *monomorphic* dynamic functions due to the fact that unquantified type pattern variables are implicitly existentially quantified. The function wrapDynamicExt is a predefined function that conveniently packs a conventional dynamic and the corresponding dynamic type representation into an extended dynamic.

```
dynApply :: DynamicExt DynamicExt → DynamicExt
dynApply    (DynExt (f :: a → b) ((TRArrow tra tra) :: TypeRep (a → b)))
            ((DynExt (x :: a) _)
    = wrapDynamicExt (f x) trb

dynSwap :: DynamicExt → DynamicExt
dynSwap (DynExt ((x, y) :: (a, b)) ((TRType [tra, trb] _ _) :: TypeRep (a, b)))
    = wrapDynamicExt (y, x) (trTuple2 trb tra)

wrapDynamicExt :: a Type → DynamicExt | TC a
wrapDynamicExt x tr=:(_ :: TypeRep a^)
    = DynExt (dynamic x :: a^) tr
```

3.4.7 Elimination of extended dynamics

In the previous section we have shown how we can apply generic functions to conventional dynamics if the program manages *extended* dynamics. We emphasized in Section 3.2 that every conventional dynamic stores the representation of all compile-time types that are related to the type of the dynamic value [VP02]. This enables us to write a low-level function dynTypeRep that computes the dynamic type representation as given in the previous section from any dynamic value. Informally, we can have:

```
dynTypeRep :: Dynamic → Type
dynTypeRep (x::t) = dynamic ⟨code of t⟩ :: TypeRep t
```

If we assume that we have this function (future work[2]), we do not need the extended dynamics anymore. The dynEq function can now be written as:

```
dynEq :: Dynamic Dynamic → Bool
dynEq x=:(_::a) y=:(_::a)    = _gEq (dynTypeRep x) x y
dynEq _ _                    = False
```

The signature of this function suggests that we might be able to derive dynamic versions of generic functions automatically as just another instance. Indeed, for type schemes (G a) in which a appears at an argument position, there is always a dynamic argument from which a dynamic type representation can be constructed. However, such an automatically derived function is necessarily a *partial* function when a appears at more than one argument position, because one cannot decide what the function should do in case the dynamic arguments have non-matching contents. In addition, if a appears only at the result position, then the type scheme is not an instance of (G Dynamic), but rather Type → G Dynamic.

3.5 Example: A Pretty Printer

Pretty printers belong to the classic examples of generic programming. In this section we deviate a little from this well-trodden path by developing a program that sends a graphical version of any dynamic value to a user-selected printer. The generic function gPretty that we will develop below is given a value to display. It computes the bounding box (Box) and a function that draws the value if provided with the location of the image (Point2 Picture → Picture). Graphical metrics information (such as text width and height) depends on the resolution properties of the output environment (the abstract and unique type *Picture). Therefore gPretty is a state transformer on Pictures, with the synonym type

```
:: St s a :== s → (a, s)
```

Picture is predefined in the Clean Object I/O library [AW00], and so are Point2 and Box.

[2]This is done in [WSP04].

```
generic gPretty t :: t → St Picture (Box, Point2 Picture  → Picture)
:: Point2 = { x     :: Int , y     :: Int }
:: Box    = { box_w :: Int , box_h :: Int }
```

The key issue of this example is how gPretty handles dynamics. If we assume that _gPretty is the derived code of gPretty as presented in Section 3.4 (that is either generated by the compiler or manually included by the programmer) then this code does the job:

```
dynPretty :: Dynamic → St Picture (Box, Point2 Picture → Picture)
dynPretty dx = _gPretty (dynTypeRep dx) dx
```

It is important to observe that the program contains no *derived* instances of the generic gPretty function. Still, it can display every possible dynamic value.

We first implement the gPretty function and then embed it in a simple GUI. In order to obtain compact code we use a monadic programming style [Wad90]. Clean has no special syntax for monads, but the standard combinators

```
return :: a → St s a
(>>=)  :: (St s a) (a → St s b) → St s b
```

are easily defined.

The generic instances for basic types simply refer to the string instances that do the real work. It draws the text and the enclosing rectangle (∘ is function composition, we assume that the get−Metrics−Info function returns the width and height of the argument string, proportional margins, and base line offset of the font):

```
gPretty{|Int|} x
   = gPretty{|⋆|} (toString x)
gPretty{|String|} s
   = getMetricsInfo s >>= λ(width, height, hMargin, vMargin, fontBase) →
      let bound = { box_w = 2 ∗ hMargin + width, box_h=2∗vMargin + height }
      in return
         ( bound
         , λ {x, y} → drawAt {x = x + hMargin, y + vMargin +fontBase } s
            ∘ drawAt { x = x + 1, y = y + 1 }
               { box_w=bound.box_w−2,box_h=bound.box_h−2 }
         )
```

The other cases only place the recursive parts at the proper positions and compute the corresponding bounding boxes. The most trivial ones are UNIT, which draws nothing, and EITHER, which continues recursively (poly)typically:

```
gPretty{|Unit|} _ = return (zero, const id)
gPretty{|EITHER|} pl pr (LEFT x) = pl x
gPretty{|EITHER|} pl pr (RIGHT x) = pr x
```

PAIRs are drawn in juxtaposition with top edges aligned. A CONS draws the recursive component below the constructor name and centers the bounding boxes.

```
gPretty{|PAIR|} px py (PAIR x y)
    =   px x ≫= λ({box_w = wx, box_h = hx}, fx) →
        py y ≫= λ({box_w = wy, box_h = hy}, fy) →
        let bound = { box_w = wx + wy, box_h = max hx hy }
        in return (bound, λpos → fy { pos & x = pos.x+wx } o fx pos
gPretty{|CONS of {gcd_name}|} px (CONS x)
    =   gPretty{|∗|} gcd_name ≫= λ({box_w = wc, box_h = hc}, fc) →
        px x ≫= λ({box_w = wx, box_h = hx}, fx) →
        let bound { box_w = max wc wx, box_h = hc + hx }
        in return (bound, λpos → fx (pos + {x=(bound.box_w−wx)/2, y=hc})
                                ◦ fc (pos + {x=(bound.box_w−wc)/2, y=0}))
```

The construct CONS **of** d binds the variable d to the constructor descriptor of the matching constructor. In the example above the constructor name field gcd_name is selected.

This completes the generic pretty printing function. We will now embed it in a GUI program. The `Start` function creates a GUI framework on which the user can drop files. The program response is defined by the `ProcessOpenFiles` attribute function which applies `showDynamic` to each dropped file path name.

```
Start :: ∗World → ∗World
Start world = startIO SDI Void id
                [ processClose closeProcess
                , processOpenFiles (λfs → pSt → foldr showDynamic pSt fs)
                ] world
```

The function showDynamic checks if the file contains a dynamic, and if so, sends it to the printer. This job is taken care of by the print function, which takes as third argument a Picture state transformer that produces the list of pages. For reasons of simplicity we assume that the image fits on one page.

```
showDynamic :: String (PSt Void) → PSt Void
showDynamic fileName pSt
    = case readDynamic fileName pSt of
        (True, dx, pSt) →
            ( snd
            ◦ uncurry ( print True False (pages dx))
```

```
              ∘ defaultPrintSetup
              ) pSt
          (_, _, pSt) → pSt
where
    pages :: Dynamic PrintInfo → St Picture [IdFun Picture]
    pages dx _ = dynPretty dx ≫= λ(_, draw_dx) → return [draw_dx zero]
```

3.6 Related Work

The idea of combining generic functions with dynamic values was first expressed in [AH02], but no concrete implementation details were presented. The work reported here is about the implementation of such a combination.

Cheney and Hinze [CH02a] present an approach that unifies dynamics and generics in a single framework. Their approach is based on explicit type representations for every type, which allows for *poor man's dynamics* to be defined explicitly by pairing a value with its type representation. In this way, a generic function is just a function defined by induction on type representations. An advantage of their approach is that it reconciles generic and dynamic programming right from start, which results in an elegant representation of types that can be used both for generic and dynamic programming. Dynamics in Clean have been designed and implemented to offer a *rich man's dynamics* (Section 3.2). Generics in Clean are schemes used to generate functions based on types available at compile-time. For this reason we have developed a first-class mechanism to be able to specialize generics at run-time. Our dynamic type representation is inspired by Cheney and Hinze, but is less verbose since we can rely on built-in dynamic type unification.

Altenkirch and McBride [AM03] implement generic programming support as a library in the dependently typed language OLEG. They present the generic specialization algorithm due to Hinze [Hin00a] as a function `fold`. For a generic function (given by the set of base cases) and an argument type, `fold` returns the generic function specialized to the type. Our `specialize` is similar to their `fold`; it also specializes a generic to a type.

3.7 Current and Future Work

The low-level function dynTypeRep (Section 3.4.7) has to be implemented. We expect that this function gives some opportunity to simplify the TypeRep data type. Polymorphic functions are a recent addition to dynamics, and we will want to handle them by generic functions as well. The solution as presented in this paper works for generic functions of kind $\star$. We want to extend the scheme so that higher order kinds can be handled as well. In addition, the approach has to be extended to handle generic functions with several generic arguments. The scheme has to be incorporated in the compiler, and we need to decide how the derived code should be made available to the programmer.

3.8 Summary and Conclusions

In this paper we have shown how generic functions can be applied to dynamic values. The technique makes essential use of dynamics to obtain first-class representations of generic functions and dynamic type representations. The scheme works for all generic functions. Applications built in this way combine the best of two worlds: they have compact definitions and they work for any dynamic value even if these originate from different sources and even if these dynamics rely on alien types and functions. Such a powerful technology is crucial for type-safe mobile code, flexible communication, and plug-in architectures. A concrete application domain that has opportunities for this technique is the functional operating system *Famke* [WP02] (parsers, pretty printers, tool specialization).

Acknowledgements

The authors would like to thank the anonymous referees for suggestions to improve the presentation of this paper.

Chapter 4

Optimizing Generic Functions

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.

4.1 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 [AP02, CHJ$^+$01, 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$^+$01] and Generic Clean [AP02]. 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 4.2 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 4.5, we extend the semantics of our language to evaluation of open expressions, and establish some properties of this so-called symbolic evaluation. In section 4.6 we discuss termination issues of symbolic evaluation of the generated code. Section 4.7 discusses related work. Section 4.8 reiterates our conclusions.

4.2 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]:

```
data 𝟙       = Unit
data α × β   = (α, β)
data α + β   = Inl α | Inr β
```

For instance, the data types

```
data List α   =  Nil | Cons α (List α)
data Tree α β =  Tip α | Bin β (Tree α β) (Tree α β)
data Rose α   =  Node α (List (Rose α))
```

are represented as synonym types

```
type List° α   = 𝟙 + α × (List α)
type Tree° α β = α + β × Tree α β × Tree α β
type Rose° α   = α × List (Rose α)
```

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:

```
to_List :: List α → List° α
to_List = λl . case l of
    Nil → Inl Unit
    Cons x xs → Inr (x, xs)

from_List :: List° α → List α
from_List = λl . case l of
    Inl u → case u of Unit → Unit
    Inr p → case p of (x, xs) → Cons x xs
```

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]

```
data α ⇄ β = EP (α → β) (β → α)
```

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

```
to :: (α ⇄ β) → (α → β)
to = λx . case x of EP t f → t
```

```
from :: (α ⇄ β) → (β → α)
from = λx . case x of EP t f → f

inv :: (α ⇄ β) → (β ⇄ α)
inv = λx → EP (from x) (to x)

(•) infixl 9 :: (β ⇄ γ) → (α ⇄ β) → (α ⇄ γ)
(•) = λa . λb . EP (to a ∘ to b) (from b ∘ from a)
```

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

```
conv_List :: List α ⇄ List° α
conv_List =  EP to_List from_List
```

To define a generic (*polytypic*) function the programmer provides the basic *poly-kinded type* [Hin00c] and the instances on the base types. For example, the generic mapping is given by the type

```
type Map α β = α → β
```

and the base cases

```
map_𝟙 :: 𝟙 → 𝟙
map_𝟙 = λx . case x of Unit → Unit

map_× :: ∀ α₁ α₂ β₁ β₂ . (α₁ → β₁) → (α₂ → β₂) → (α₁ × α₂ → β₁ × β₂)
map_× = λf . λg . λe . case e of
    (x, y) → (f x, g y)

map_+ :: ∀ α₁ α₂ β₁ β₂ . (α₁ → β₁) → (α₂ → β₂) → (α₁ + α₂ → β₁ + β₂)
map_+ =λf . λg . λe . case e of
    Inl x → Inl (f x)
    Inr x → Inr (g x)
```

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

```
map_List° :: (α → β) → List° α → List° β
map_List° = λf . map_+ map_𝟙 (map_× f (map_List f))
```

Note that the structure of $\mathsf{map}_{\mathrm{List}^\circ}$ 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

$\mathsf{ep}_{\mathbb{1}} :: \mathbb{1} \rightleftarrows \mathbb{1}$
$\mathsf{ep}_{\mathbb{1}} = \mathsf{EP}\ \mathsf{map}_{\mathbb{1}}\ \mathsf{map}_{\mathbb{1}}$

$\mathsf{ep}_{+} :: \forall\ \alpha_1\ \alpha_2\ \beta_1\ \beta_2\ .\ (\alpha_1 \rightleftarrows \beta_1) \to (\alpha_2 \rightleftarrows \beta_2) \to (\alpha_1 + \alpha_2 \rightleftarrows \beta_1 + \beta_2)$
$\mathsf{ep}_{+} = \lambda\mathsf{a}\ .\ \lambda\mathsf{b}\ .\ \mathsf{EP}\ (\mathsf{map}_{+}\ (\mathsf{to\ a})\ (\mathsf{to\ b}))\ (\mathsf{map}_{+}\ (\mathsf{from\ a})\ (\mathsf{from\ b}))$

$\mathsf{ep}_{\times} :: \forall\ \alpha_1\ \alpha_2\ \beta_1\ \beta_2\ .\ (\alpha_1 \rightleftarrows \beta_1) \to (\alpha_2 \rightleftarrows \beta_2) \to (\alpha_1 \times \alpha_2 \rightleftarrows \beta_1 \times \beta_2)$
$\mathsf{ep}_{\times} = \lambda\mathsf{a}\ .\ \lambda\mathsf{b}\ .\ \mathsf{EP}\ (\mathsf{map}_{\times}\ (\mathsf{to\ a})\ (\mathsf{to\ b}))\ (\mathsf{map}_{\times}\ (\mathsf{from\ a})\ (\mathsf{from\ b}))$

$\mathsf{ep}_{\to} :: \forall\ \alpha_1\ \alpha_2\ \beta_1\ \beta_2\ .\ (\alpha_1 \rightleftarrows \beta_1) \to (\alpha_2 \rightleftarrows \beta_2) \to ((\alpha_1 \to \alpha_2) \rightleftarrows (\beta_1 \to \beta_2))$
$\mathsf{ep}_{\to} = \lambda\mathsf{a}\ .\ \lambda\mathsf{b}\ .\ \mathsf{EP}\ (\lambda\mathsf{f}\ .\ \mathsf{to\ b} \circ \mathsf{f} \circ \mathsf{from\ a})\ (\lambda\mathsf{f}\ .\ \mathsf{from\ b} \circ \mathsf{f} \circ \mathsf{to\ a})$

$\mathsf{ep}_{\rightleftarrows} :: \forall\ \alpha_1\ \alpha_2\ \beta_1\ \beta_2\ .\ (\alpha_1 \rightleftarrows \beta_1) \to (\alpha_2 \rightleftarrows \beta_2) \to ((\alpha_1 \rightleftarrows \alpha_2) \rightleftarrows (\beta_1 \rightleftarrows \beta_2))$
$\mathsf{ep}_{\rightleftarrows} = \lambda\mathsf{a}\ .\ \lambda\mathsf{b}\ .\ \mathsf{EP}\ (\lambda\mathsf{e}\ .\ \mathsf{b} \bullet \mathsf{e} \bullet \mathsf{inv\ a})\ (\lambda\mathsf{e}\ .\ \mathsf{inv\ b} \bullet \mathsf{e} \bullet \mathsf{a})$

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 Map α β we have:

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

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

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

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

$\mathsf{map}_{\mathrm{List}} = \lambda\mathsf{f}\ .\ \lambda\mathsf{l}\ .\ \textbf{case}\ \mathsf{l}\ \textbf{of}$
$\quad \mathsf{Nil} \to \mathsf{Nil}$
$\quad \mathsf{Cons\ x\ xs} \to \mathsf{Cons}\ (\mathsf{f\ x})\ (\mathsf{map}_{\mathrm{List}}\ \mathsf{f\ xs})$

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.

4.3 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.

4.3.1 Syntax

Definition 4.1 (Expressions and Functions)

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

$$\begin{array}{lcl} E & ::= & x \mid C\vec{E} \mid F \mid \lambda x.E \mid E\ E' \mid \textsf{case}\ E\ \textsf{of}\ P_1 \rightarrow E_1 \cdots P_n \rightarrow E_n \\ P & ::= & C\vec{x} \end{array}$$

2. *A* function definition *is an expression of the form $F = E_F$ with* $\mathrm{FV}(E_F) = \emptyset$*. 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).

4.3.2 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 4.5, we extend this standard evaluation to so-called *symbolic evaluation*: evaluation of expressions containing free variables.

Definition 4.2 (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.

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

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

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.

4.4 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 4.3 (Types)
The set of types *is given by the following syntax. Below, α ranges over* type variables, *and* T *over type constructors.*

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

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 4.4 (Type environments)

1. *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*

$$\mathrm{T}\ \vec{\alpha} = \cdots \mid C_i\ \vec{\sigma_i} \mid \cdots$$

be the specification of T *in $\mathcal{A}$. Then we write*

$$\mathcal{A} \vdash C_i : \forall \vec{\alpha}.\vec{\sigma_i} \to \mathrm{T}\ \vec{\alpha}.$$

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

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

Type derivation

Definition 4.5 (Type Derivation)

1. *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.

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

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

$$\frac{B \vdash E : \tau \quad B \vdash C_i : \vec{\rho_i} \to \tau \qquad B, \vec{x_i} : \vec{\rho_i} \vdash E_i : \sigma}{B \vdash \mathsf{case}\ E\ \mathsf{of}\ \cdots C_i\vec{x_i} \mid E_i \cdots : \sigma}\ (\sigma\text{-case})$$

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

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

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

4.5 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 4.2).

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 4.6 (Symbolic Evaluation) *We adjust definition 4.2 of evaluation by replacing the E-λ rule, and by adding rules for dealing with new* *combinations of expressions.*

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

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

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

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

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

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

Note that the rules (E-case) and (E-app) from definition 4.2 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 4.7 (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 4.2.

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

The following definition characterizes the results of symbolic evaluation.

Definition 4.8 (Symbolic Normal Forms) *The set of* symbolic normal forms *(indicated by N_s) is defined by the following syntax.*

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

Proposition 4.9 (Correctness of Symbolic Normal Form)

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

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

4.5.1 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 4.10 (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 4.11 (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.*

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

  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.*

$$\begin{array}{lcll}
C_T(\alpha) & = & \emptyset & \\
C_T(\mathrm{T}) & = & \cup_i[\{C_i\} \cup C_T(\vec{\sigma_i})], & \text{where } \mathrm{T} = \cdots \mid C_i\vec{\sigma_i} \mid \cdots \\
C_T(\tau \to \sigma) & = & C_T(\tau) \cup C_T(\sigma) & \\
C_T(\tau\ \sigma) & = & C_T(\tau) \cup C_T(\sigma) & \\
C_T(\forall\alpha.\sigma) & = & C_T(\sigma) &
\end{array}$$

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

Example 4.12 The rose tree Rose from section 4.2 is built with help of the List type. For these types we have

$C_T(\textsf{List}) = \{\textsf{Nil}, \textsf{Cons}\}$
$C_T(\textsf{Rose}) = \{\textsf{Node}, \textsf{Nil}, \textsf{Cons}\}$.

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 4.13 *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. $\Box$

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 4.14 (Deforestation Property)

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

Proof: By proposition 4.9, 4.13, and 4.10. $\Box$

4.5.2 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 4.15 *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 [AS04a]. $\Box$

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

Proposition 4.16 *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 constructor C one has*

$$C \in C_N(N) \;\Rightarrow\; C \in C_T(\sigma, \mathrm{T})$$

Proof: By proposition 4.14, 4.10, and 4.15. □

Recall from section 4.2 that the intermediate data introduced by the generic specializer are built from the structural representation base types $\{\times, +, \mathsf{Unit}, \rightleftarrows\}$. 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.

4.6 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 4.16.

Corollary 4.17 (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 $\mathsf{map}_{\mathrm{List}^\circ}$ which, in turn, calls recursively $\mathsf{map}_{\mathrm{List}}$. 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 4.18 (Non-recursive specialization) The specialization of map to List without recursion:

```
map'_List = λm .
    from (ep_→ conv_List conv_List
        ∘ (λf . map_+ map_𝟙 (map_× f (m f)))
```

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

```
map'_List = λm . λf . λx . case x of
    Nil         → Nil
    Cons y ys   → Cons (f y) (m f ys)
```

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

type Mapl α $\beta = \alpha \rightarrow$ List β

and the base cases

```
mapl_𝟙 : 𝟙 → List 𝟙
mapl_𝟙 = return Unit

mapl_× : ∀ α1 α2 β1 β2 . (α1 → List β1) (α2 → List β2) → α1 × α2 → β1 × β2
mapl_× = λf . λg . λp . case p of
    (x, y) → f x ≫= λx' . g y ≫= λy' . return (x', y')

mapl_+ : : ∀ α1 α2 β1 β2 . (α1 → List β1) (α2 → List β2) → α1 + α2 → β1 + β2
mapl_× = λf . λg . λe . case e of
    Inl x = f x ≫= λx' → return (Inl x')
    Inr y = g y ≫= λy' → return (Inr y')
```

where

```
return = λx . Cons x Nil
(≫=) = λl . λf → flatten (map f l)
```

are the monadic return and (infix) bind for the list monad. The specialization of mapl to any data type, e.g. Rose, uses the embedding-projection specialized to Mapl (see section 4.2).

```
mapl_Rose : (α → List β) → Rose α → List (Rose β)
mapl_Rose = from (ep_Mapl conv_Rose conv_Rose) ∘ mapl_Rose°
```

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

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

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

$\mathsf{ep}_{\mathrm{List}} : (\alpha \rightleftarrows \beta) \to (\mathsf{List}\ \alpha \rightleftarrows \mathsf{List}\ \beta)$
$\mathsf{ep}_{\mathrm{List}} = \mathsf{from}\ (\mathsf{ep}_{\rightleftarrows}\ \mathsf{conv}_{\mathrm{List}}\ \mathsf{conv}_{\mathrm{List}}) \circ \mathsf{ep}_{\mathrm{List}^\circ}$

$\mathsf{ep}_{\mathrm{List}^\circ} : (\alpha \rightleftarrows \beta) \to (\mathsf{List}^\circ\ \alpha \rightleftarrows \mathsf{List}^\circ\ \beta)$
$\mathsf{ep}_{\mathrm{List}^\circ} = \lambda \mathsf{f}\ .\ \mathsf{ep}_{+}\ \mathsf{ep}_{\mathbb{1}}\ (\mathsf{ep}_{\times}\ \mathsf{f}\ (\mathsf{ep}_{\mathrm{List}}\ \mathsf{f}))$

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

4.7 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+01] and Generic Clean [AP02] 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.

4.8 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.

Chapter 5

Fusing Generic Functions

Generic programming is accepted by the functional programming community as a valuable tool for program development. Several functional languages have adopted the generic scheme of type-indexed values. This scheme works by specialization of a generic function to a concrete type. However, the generated code is extremely inefficient compared to its hand-written counterpart. The performance penalty is so big that the practical usefulness of generic programming is compromised. In this paper we present an optimization algorithm that is able to completely eliminate the overhead introduced by the specialization scheme for a large class of generic functions. The presented technique is based on consumer–producer elimination as exploited by fusion, a standard general purpose optimization method. We show that our algorithm is able to optimize many practical examples of generic functions.

5.1 Introduction

Generic programming is recognized as an important tool for minimizing boilerplate code that results from defining the same operation on different types. One of the most wide-spread generic programming techniques is the approach of type-indexed values [Hin00a]. In this approach, a generic operation is defined once for all data types. For each concrete data type an instance of this operation is generated. This instance is an ordinary function that implements the operation on the data type. We say that the generic operation is *specialized* to the data type.

The generic specialization scheme uses a structural view on a data type. In essence, an algebraic type is represented as a sum of products of types.

The structural representation uses *binary* sums and products. Generic operations are defined on these structural representations. Before applying a generic operation the arguments are converted to the structural representation, then the operation is applied to the converted arguments and then the result of the operation is converted back to its original form.

A programming language's feature is only useful in practice, if its performance is adequate. Directly following the generic scheme leads to very inefficient code, involving numerous conversions between values and their structural representations. The generated code additionally uses many higher-order functions (representing dictionaries corresponding to the type arguments). The inefficiency of generated code severely compromises the utility of generic programming.

In the previous work [AS04c] we used a partial evaluation technique to eliminate generic overhead introduced by the generic specialization scheme. We proved that the described technique completely removes the generic overhead. However, the proposed optimization technique lacks termination analysis, and therefore works only for non-recursive functions. To make the technique work for instances on recursive types we abstracted the recursion with a Y-combinator and optimized the non-recursive part. This technique is limited to generic functions that do not contain recursion in their types, though the instance types can be recursive. Another disadvantage of the proposed technique is that it is tailored specifically to optimize generics, because it performs the recursion abstraction of generic instances.

The present paper describes a general purpose optimization technique that is able to optimize a significantly larger class of generic instances. In fact, the proposed technique eliminates the generic overhead in nearly all practical generic examples. When it is not able to remove the overhead completely, it still improves the code considerably. The presented optimization algorithm is based on fusion [AGS03, Chi94]. In its turn, fusion is based on the consumer-producer model: a producer produces data which are consumed by the consumer. Intermediate data are eliminated by combining (*fusing*) consumer-producer pairs.

The contributions of the present paper are:

- The original fusion algorithm is improved by refining both consumer and producer analyses. Our main goal is to achieve good fusion results for generics, but the improvements also appear to pay off for non-generic examples.

- We describe the class of generic programs for which the generic overhead is completely removed. This class includes nearly all practical generic programs.

In the next section we introduce the code generated by the generic specialization. This code is subject to the optimization described further in the paper. The generated code is represented in a simple functional language defined in section 5.3. Section 5.4 defines the semantics of fusion with no termination analysis. Basic properties of this fusion algorithm are discussed in section 5.5. Standard fusion with termination analysis [AGS03] is described in section 5.6. Sections 5.7 and 5.8 introduce our extensions to the consumer and the producer analyses. Fusion of generic programs is described in 5.9. The performance results for generic programs are presented in section 5.10. Section 5.11 discusses related work. Conclusions are presented and future work is discussed in section 5.12.

5.2 Generics

In this section we give a brief overview of the generic specialization scheme which is based on the approach by Hinze [Hin00a]. Generic functions exploit the fact that any data type can be represented in terms of sums, pairs and unit, called the *base types*. These base types can be specified by the following Haskell-like data type definitions.

$$\begin{array}{lcl}\textbf{data}\ \mathbb{1} & = & \mathsf{Unit}\\ \textbf{data}\ a\ \times\ b & = & \mathsf{Pair}\ a\ b\\ \textbf{data}\ a\ +\ b & = & \mathsf{Inl}\ a \mid \mathsf{Inr}\ b\end{array}$$

A generic (*type-indexed*) function g is specified by means of instances for these base types. The structural representation of a concrete data type, say T, is used to generate an instance of g for T. The idea is to convert an object of type T first to its structural representation, apply the generic operation g to it, and convert the resulting object back from its structural to its original representation.

Suppose that the generic function g has generic (*kind-indexed*) type G. Then the instance g_T of g for the concrete type T has the following form.

$$g_T \vec{f} = \mathsf{adapt}_{\langle G,T\rangle}(g_{T^\circ} \vec{f})$$

where T° denotes the structural representation of T, g_{T° represents the instance of g on T°, and the adapter $\mathsf{adapt}_{\langle G,T\rangle}$ takes care of conversion between T and T°. We will illustrate this generic specialization scheme with a few examples, starting with the structural representations of some familiar data types:

$$\begin{aligned}
&\mathbf{data}\ \mathsf{List}\ a &&= \mathsf{Nil} \mid \mathsf{Cons}\ a\ (\mathsf{List}\ a)\\
&\mathbf{data}\ \mathsf{Tree}\ a &&= \mathsf{Leaf}\ a \mid \mathsf{Branch}\ (\mathsf{Tree}\ a)\ (\mathsf{Tree}\ a)\\
&\mathbf{data}\ \mathsf{Rose}\ a &&= \mathsf{Node}\ a\ (\mathsf{List}\ (\mathsf{Rose}\ a))
\end{aligned}$$

These types are represented as

$$\begin{aligned}
&\mathbf{type}\ \mathsf{List}^{\circ}\ a &&= \mathbb{1} + a \times \mathsf{List}\ a\\
&\mathbf{type}\ \mathsf{Tree}^{\circ}\ a &&= a + \mathsf{Tree}\ a \times \mathsf{Tree}\ a\\
&\mathbf{type}\ \mathsf{Rose}^{\circ}\ a &&= a \times \mathsf{List}\ (\mathsf{Rose}\ a)
\end{aligned}$$

Observe that only the top-level of the data definitions is converted to the structural form.

A type and its structural representation are isomorphic. The isomorphism is witnessed by a pair of conversion functions. For instance, for lists these functions are

```
convTo_List :: List a → List° a
convTo_List l      =  case  l of
                              Nil              →  Inl Unit
                              Cons x xs        →  Inr (Pair x xs)
convFrom_List :: List° a → List a
convFrom_List l    =  case  l of
                              Inl Unit         →  Nil
                              Inr (Pair x xs)  →  Cons x xs
```

To define a generic function g the programmer has to provide the generic type G, and the instances on the base types. For example, the generic mapping is given by the type

$$\mathbf{type}\ \mathsf{Map}\ a\ b = a \to b$$

and the base cases

```
map_𝟙          =  case u of  Unit      →  Unit
map_× l r p    =  case p of  Pair x y  →  Pair (l x) (r y)
map_+ l r e    =  case e of  Inl x     →  Inl (l x)
                             Inr y     →  Inr (r y)
```

This is all that is needed for the generic specializer to build an instance of map for any concrete data type T. As said before, such an instance is generated by interpreting the structural representation T° of T, and by creating an appropriate adapter. For instance, the generated mapping for List° is

$$\begin{aligned}&\mathsf{map}_{\mathsf{List}^\circ} :: \mathsf{Map}\ a\ b \rightarrow \mathsf{Map}\ (\mathsf{List}^\circ\ a)\ (\mathsf{List}^\circ\ b)\\ &\mathsf{map}_{\mathsf{List}^\circ}\ f = \mathsf{map}_{+}\ \mathsf{map}_{\mathbb{1}}\ (\mathsf{map}_{\times}\ f\ (\mathsf{map}_{\mathsf{List}}\ f))\end{aligned}$$

Note how the structure of $\mathsf{map}_{\mathsf{List}^\circ}$ directly reflects the structure of List°. The adaptor converts the instance on the structural representation into an instance on the concrete type itself. E.g., the adapter converting $\mathsf{map}_{\mathsf{List}^\circ}$ into $\mathsf{map}_{\mathsf{List}}$ (i.e. the mapping function for List), has type

$$\mathsf{adapt}_{\langle\mathsf{Map},\mathsf{List}\rangle} :: \mathsf{Map}\ (\mathsf{List}^\circ\ a)\ (\mathsf{List}^\circ\ b) \rightarrow \mathsf{Map}\ (\mathsf{List}\ a)\ (\mathsf{List}\ b)$$

The code for this adapter function is described below. We can now easily combine $\mathsf{adapt}_{\langle\mathsf{Map},\mathsf{List}\rangle}$ with $\mathsf{map}_{\mathsf{List}^\circ}$ to obtain a mapping function for the original List type.

$$\begin{aligned}&\mathsf{map}_{\mathsf{List}} :: \mathsf{Map}\ a\ b \rightarrow \mathsf{Map}\ (\mathsf{List}\ a)\ (\mathsf{List}\ b)\\ &\mathsf{map}_{\mathsf{List}}\ f = \mathsf{adapt}_{\langle\mathsf{Map},\mathsf{List}\rangle}\ (\mathsf{map}_{\mathsf{List}^\circ}\ f)\end{aligned}$$

The way the adaptor works depends on the type of the generic function as well as on the concrete data type for which an instance is created. So called *embedding projections* are used to devise the automatic conversion. In essence such an embedding projection distributes the original conversion functions (the isomorphism between the type and its structural representation) over the type of the generic function. In general, the type of a generic function can contain arbitrary type constructors, including arrows. These arrows may also appear in the definition of the type for which an instance is derived. To handle such types in a uniform way, conversion functions are packed into *embedding-projection pairs*, EPs (e.g. see [HP01]), which are defined as follows.

$$\mathbf{data}\ a \rightleftarrows b = \mathsf{EP}\ (a \rightarrow b)\ (b \rightarrow a)$$

For instance, packing the List conversion functions into an EP leads to:

$$\begin{aligned}&\mathsf{conv}_{\mathsf{List}} :: \mathsf{List}\ a \rightleftarrows \mathsf{List}^\circ\ a\\ &\mathsf{conv}_{\mathsf{List}} = \mathsf{EP}\ \mathsf{convTo}_{\mathsf{List}}\ \mathsf{convFrom}_{\mathsf{List}}\end{aligned}$$

Now the adapter for G and T can be specified in terms of embedding projections using the EP that corresponds to the isomorphism between T and T° as a basis. Actually, embedding projections are represented as a generic function themselves. This has the advantage that we can use the same specialization scheme for embedding projections that is used for other generic functions. More concretely, an embedding projection is a generic function ep with the generic type $a \rightleftarrows b$, and the base cases:

$$\begin{aligned}
\mathsf{ep}_{\mathbb{1}} \quad &= \quad \mathsf{EP}\ \mathsf{map}_{\mathbb{1}}\ \mathsf{map}_{\mathbb{1}} \\
\mathsf{ep}_{\times}\ f\ g \quad &= \quad \mathsf{EP}\ (\mathsf{map}_{\times}\ (\mathsf{to}\ f)\ (\mathsf{to}\ g))\ (\mathsf{map}_{\times}\ (\mathsf{from}\ f)\ (\mathsf{from}\ g)) \\
\mathsf{ep}_{+}\ f\ g \quad &= \quad \mathsf{EP}\ (\mathsf{map}_{+}\ (\mathsf{to}\ f)\ (\mathsf{to}\ g))\ (\mathsf{map}_{+}\ (\mathsf{from}\ f)\ (\mathsf{from}\ g)) \\
\mathsf{ep}_{\rightarrow}\ f\ g \quad &= \quad \mathsf{EP}\ (\mathsf{mapAR}\ (\mathsf{from}\ f)\ (\mathsf{to}\ g))\ (\mathsf{mapAR}\ (\mathsf{to}\ f)\ (\mathsf{from}\ g)) \\
\mathsf{ep}_{\rightleftarrows}\ f\ g \quad &= \quad \mathsf{EP}\ (\mathsf{mapEP}\ (\mathsf{to}\ f)\ (\mathsf{from}\ f)(\mathsf{to}\ g)\ (\mathsf{from}\ g)) \\
&\qquad\quad (\mathsf{mapEP}\ (\mathsf{from}\ f)\ (\mathsf{to}\ f)(\mathsf{from}\ g)\ (\mathsf{to}\ g))
\end{aligned}$$

where

$$\begin{aligned}
\mathsf{to}\ e \quad &= \quad \mathsf{case}\ e\ \mathsf{of}\ \mathsf{EP}\ t\ f \rightarrow t \\
\mathsf{from}\ e \quad &= \quad \mathsf{case}\ e\ \mathsf{of}\ \mathsf{EP}\ t\ f \rightarrow f \\
\mathsf{mapAR}\ a\ r\ f \quad &= \quad r \circ f \circ a \\
\mathsf{mapEP}\ t_a\ f_a\ t_r\ f_r\ e \quad &= \quad EP\ (t_r \circ \mathsf{to}\ e \circ f_a)\ (t_a \circ \mathsf{from}\ e \circ f_r)
\end{aligned}$$

These instances are based on the basic instances of the previously defined function map.

Apart from the usual instances for sum, pair and unit, we have included the instances on $\rightarrow$ and $\rightleftarrows$. In particular the latter might look somewhat mysterious. The reason for specifying this instance is rather technical: it appears in the adapter of the specialized version of ep for a concrete type T, e.g. see section 5.9.1

The generic specializer generates the instance of ep specific to a generic function, again by interpreting its generic type. E.g. for mapping (with the generic type $\mathsf{Map}\ a\ b$) we get:

$$\begin{aligned}
&\mathsf{ep}_{\mathsf{Map}} :: (a_1 \rightleftarrows a_2) \rightarrow (b_1 \rightleftarrows b_2) \rightarrow (\mathsf{Map}\ a_1\ b_1 \rightleftarrows \mathsf{Map}\ a_2\ b_2) \\
&\mathsf{ep}_{\mathsf{Map}}\ a\ b = \mathsf{ep}_{\rightarrow}\ a\ b
\end{aligned}$$

Now the adaptor $\mathsf{adapt}_{\langle\mathsf{Map},\mathsf{List}\rangle}$ is the from-component of this embedding projection applied to $\mathsf{conv}_{\mathsf{List}}$ twice.

$$\mathsf{adapt}_{\langle\mathsf{Map},\mathsf{List}\rangle} = \mathsf{from}\ (\mathsf{ep}_{\mathsf{Map}}\ \mathsf{conv}_{\mathsf{List}}\ \mathsf{conv}_{\mathsf{List}})$$

To compare the generated version of map with its handwritten counterpart, e.g.

$$\begin{aligned}
\mathsf{map}\ f\ l = \mathsf{case}\ l\ \mathsf{of}\quad &\mathsf{Nil} &&\rightarrow\ \mathsf{Nil} \\
&\mathsf{Cons}\ x\ xs &&\rightarrow\ \mathsf{Cons}\ (f\ x)\ (\mathsf{map}\ f\ xs)
\end{aligned}$$

we have inlined the adapter and the instance for the structural representation in the definition of $\mathsf{map}_{\mathsf{List}}$ resulting in

$$\begin{aligned}\mathsf{map}_{\mathsf{List}}\ f \quad = \quad & \mathsf{from}\ (\mathsf{ep}_{\mathsf{Map}}\ \mathsf{conv}_{\mathsf{List}}\ \mathsf{conv}_{\mathsf{List}}) \\ & \quad (\mathsf{map}_{+}\ \mathsf{map}_{\mathbb{1}}\ (\mathsf{map}_{\times}\ f\ (\mathsf{map}_{\mathsf{List}}\ f)))\end{aligned}$$

Clearly, the generated version is much more complicated than the handwritten one, not only in terms of readability but also in terms of efficiency. The latter is the main concern of this paper. The reasons for inefficiency are the intermediate data structures for the structural representation and the extensive usage of higher-order functions. In the rest of the paper we present an optimization technique for generic functions which is based on fusion, and show that this technique is capable of removing all generic overhead, for a large class of generic functions.

5.3 Language

In this section we present the syntax of a simple core functional language that supports essential aspects of functional programming such as pattern matching and higher-order functions. The fusion semantics of this core language is described in the next section. First we introduce some more or less common terminology and notation.

Notatation 5.1 (Vectors)

- *We will use the* vector $\vec{V}$ *for* $(V_1, \ldots, V_n)$*. The length of a vector* V *is indicated by* $|\vec{V}|$

- *If* V *is a vector then* V_i *denotes the* i^{th} *element of* V*, and* $V_{i..j}$ *the (sub)vector* $(V_i, \ldots, V_j)$*. If* $i > j$ *then* $V_{i..j} = ()$*.*

- *Let* V, W *be vectors.* $V \star W$ *denotes the concatenation of* V *and* W*.*

We define the syntax in two steps: expressions and functions.

Definition 5.2 (Expressions)

- *The set of* expressions *is defined as*

$$E ::= x \mid C\,\vec{E} \mid F\,\vec{E} \mid x\ \vec{E}.$$

Here x *ranges over variables,* C *over data constructors and* F *over function symbols.*

- *By $E \subseteq E'$ we denote that E is a subexpression of E', and by $E \subset E'$ we indicate* proper *subexpressions (i.e. $E \neq E'$).*
- *Each (function or constructor) symbol has an* arity*: a natural number that indicates the maximum number of arguments to which the symbol can be applied. An expression E is* well-formed *if the actual arity of applications occurring in E never exceeds the formal arity of the applied symbols. From now on we will only consider well-formed expressions.*

Pattern matching is allowed only at the top level of a function definition. Moreover, only one pattern match per function is permitted and the patterns themselves have to be simple (free of nesting).

Definition 5.3 (Functions)

- *The set of* function bodies *is defined as follows.*

$$\begin{array}{lll} B & ::= & E \mid \textsf{case}\ x\ \textsf{of}\ P_1 \rightarrow E_1 \cdots P_n \rightarrow E_n \\ P & ::= & C\,\vec{x} \end{array}$$

  Variables in a pattern P are called pattern variables

- *The set of free variables in B is indicated by* $\mathrm{FV}(B)$.
- *A* function definition *has the form $F\,\vec{x} = B_F$ with* $\mathrm{FV}(B_F) \subseteq \vec{x}$*. The* arity *of F is $|\vec{x}|$.*
- *F is called a* case function *if it starts with a pattern match $F\,\vec{x} = \textsf{case}\ x_i\ \textsf{of}\ \ldots$. We also say that F is a* case function in *i to indicate that the pattern match occurs on the i^{th} parameter.*
- *A* component *is a set of mutually dependent functions. Let F be a function. By $\widehat{F}$ we denote the component to which F belongs.*

Data constructors are introduced via an *algebraic type definition.* Such a type definition not only specifies the *type* of each data constructor but also its *arity.* For readability reasons in this paper we will use a Haskell-like syntax in the examples.

5.4 Semantics

Most program transformation methods use the so-called *unfold/fold* mechanism to convert expressions and functions. During an *unfold step*, a call to a function is replaced by the corresponding function body in which appropriate parameter substitutions have been performed. During a *fold step*, an expression is replaced by a call to a function of which the body matches that expression.

In the present paper we will use a slightly different way of both unfolding and folding. First of all, we do not unfold all possible function applications but restrict ourselves to so called *consumer–producer* pairs. In a function application $F(\ldots, S(\ldots), \ldots)$ the function F is called a *consumer* and the function or constructor S a *producer*. The intuition behind this terminology is that F consumes the result produced by S. Suppose we have localized a consumer–producer pair in an expression E. More precisely, E contains a subexpression $F\,\vec{E}$, with $E_i = S\,\vec{D}$. Say $k = |\vec{E}|$, and $l = |\vec{D}|$. The idea of *fusion* is to replace this consumer–producer pair consisting of two calls by a single call to the combined function F_iS^l resulting in the application $F_iS^l\,E_{1..(i-1)} \star \vec{D} \star E_{(i+1)..k}$. Moreover, if this combined function is used the first time, a new function definition is generated that has the body of the consumer F in which $S\,(y_1, \ldots, y_l)$ is substituted for the ith argument. Note that this fusion mechanism does not require any explicit folding steps anymore. As an example consider the following definition of app, and the auxiliary function foo.

Example 5.4

```
app l t    =  case l of  Nil        →  t
                         Cons x xs  →  Cons x (app xs t)
foo x y z  =  app (app x y) z
```

The first fusion step leads to the creation of a new function, say app_app, and replaces the nested applications of app by a single application of this new function. The result is shown below.

```
foo x y z      =  app_app x y z
app_app x y z  =  case x of  Nil        →  app y z
                             Cons x xs  →  Cons x (app (app xs y) z)
```

The description of how the body of the new function app_app is created is given at the end of this section. foo itself does not contain consumer-producer pairs anymore; the only pair appears in the body of app_app, namely app (app xs y) z. Again these nested calls are replaced by app_app, and since app_app has already been created, no further steps are necessary.

$$\begin{array}{lllll}\mathsf{app\_app}\ x\ y\ z & = & \mathsf{case}\ x\ \mathsf{of} & \mathsf{Nil} & \rightarrow\ \mathsf{app}\ y\ z \\ & & & \mathsf{Cons}\ x\ xs & \rightarrow\ \mathsf{Cons}\ x\ (\mathsf{app\_app}\ xs\ y\ z)\end{array}$$

This example shows that we need to determine whether a function - app_app in the example - has already been generated. To facilitate this, with each newly created function we associate a special unique name, a so called *symbol tree*. These symbol trees contain all the necessary information to determine whether a new function is equal to an existing one.

Definition 5.5 (Symbol Trees)

- *The set of* symbol trees *is defined by the following syntax. In this definition, S ranges over function and constructor symbols. The special symbol $\Box$ is used to denote anonymous variables.*

$$\begin{array}{lcl} T & ::= & S\,\vec{T'} \\ T' & ::= & \Box \mid T \end{array}$$

- *The* root *of a tree $T = S\vec{T'}$ (denoted by $\ulcorner T \urcorner$) is the symbol S. The* arity *of a tree T (indicated by $\mathsf{ar}(T)$) is the number of $\Box$ symbols in T.*

- *By $T[i{\leftarrow}V]$ we denote the term that is obtained from T by substituting V for the i^{th} occurrence of $\Box$, in a depth-first, left-to-right numbering of $\Box$ symbols.*

$$\begin{array}{rcl} \Box[1{\leftarrow}V] & = & V \\ S\,\vec{T}[i{\leftarrow}V] & = & S\,(\ldots, T_j[i - a_1^{j-1}{\leftarrow}V], \ldots), \\ & & \quad \textit{for } j \textit{ such that } 0 < i - a_1^{j-1} \leq a_j \\ & & \quad \textit{where } a_i = \mathsf{ar}(T_i) \textit{ and } a_1^{j-1} = \sum_{l=1}^{j-1} a_l \end{array}$$

- *By $\Box^k$ we denote the vector. $(\underbrace{\Box, \ldots, \Box}_{k})$.*

- *We have two auxiliary operations on trees for respectively increasing and decreasing the arity:*

  1. $(S\,\vec{T}) \boxplus k = S\,(\vec{T} \star \Box^k)$

2. *$(S\,\vec{T}) \boxminus k$ is the tree obtained by removing the last k occurrences of $\square$(using the same numbering as above). Formally:*

$$\begin{array}{lcll}
(S\,\vec{T}) \boxminus 0 & = & S\,\vec{T} & \\
(S\,\vec{T}) \boxminus (k+1) & = & (S\,\vec{T}^{\boxminus}) \boxminus k & \\
\textit{where} & & & \\
(T_1,\ldots,T_l)^{\boxminus} & = & (T_1,\ldots,T_{l-1}), & \textit{if } T_l = \square \\
 & = & (T_1,\ldots,T_{l-1}, S'\,(\vec{V}^{\boxminus})), & \textit{if } T_l = S'\,\vec{V} \textit{ and } \mathsf{ar}(T_l) > 0 \\
 & = & (T_1,\ldots,T_{l-1})^{\boxminus} \star (T_l), & \textit{otherwise}
\end{array}$$

- *Let S be a symbol, and T a tree. T is called* cyclic *in S if T contains a path on which S occurs more than once, i.e. if for some tree T' one has $S\,(\ldots,T',\ldots) \subseteq T$ and $S\,(\ldots) \subseteq T'$.*

The following definition shows how a symbol tree can be converted to a function that corresponds to the original expression from which this tree has been created.

Definition 5.6 (Converting Symbol Trees) *Let T be a symbol tree of arity k. The operation $[\![T]\!]$ yields a function $F_T(x_1,\ldots,x_k) = E$, of which the body E results from T after substituting x_i for the i^{th} occurrence of $\square$, for all $i \le k$. Substitutions are performed in parallel.*

As said before, the evaluation of a function application possibly leads to the creation of new functions. The name (symbol tree) of that new function is created from the symbol tree corresponding to the consumer and the symbol tree or data constructor of the producer.

Definition 5.7 (Building Symbol Trees)

- ***Basic trees:*** *With each initial function F, say with arity n, we associate the tree $F\square^n$. Initial functions are all functions present in the original program subject to fusion.*

- ***New trees:*** *Let F be a function with symbol tree T_F, and arity n.*

  There are two ways to introduce new functions in which F is involved, and hence to introduce new symbol trees: (1) when F appears as a consumer in a consumer-producer pair (definition 5.12) or (2) when the arity of F is increased (definition 5.8).

  1. *Let S be a function or data constructor, say with arity m. Suppose we are using S with k actual arguments, $k \le m$. The result*

of combining F with S at argument position i, $i \leq n$, is a new symbol tree $F_i S^k$ defined as follows

– *S is a function: Let T_S be the symbol tree of S. Then*

$$F_i S^k = T_F[i \leftarrow T_S \boxminus (m-k)].$$

– *S is a constructor:*

$$F_i S^k = T_F[i \leftarrow S\,\square^k].$$

2. *The result of increasing the arity of T_F by k is a symbol tree*

$$F^{\oplus k} = T_F \boxplus k.$$

Here $F^{\oplus k}$ denotes a function obtained from F by raising the arity by k (see definition 5.8).

The above construction of symbol trees is order independent. For instance, there are two ways to evaluate an application $F(G(H(\ldots)))$, namely one can start with the G–H pair and combine the result with F, or one can start with F–G and combine the result with H. Both ways, however, will lead to the same tree. The same holds for an application like $F(G(\ldots), H(\ldots))$.

Unfolding (which stands in our system for the creation of new function bodies) is based on a notion of substitution for expressions. However, we cannot use a straightforward definition of substitution because a higher-order expression should only start with a variable (see definition 5.2). Suppose we try to substitute an expression $D = F\,\vec{D'}$ for x in $x\;\vec{E}$. This becomes problematic if $\mathsf{arity}(F) < |\vec{D'}| + |\vec{E}|$, because in the resulting application $F\,\vec{D'} \star \vec{E}$ is not well-formed. To solve this problem we introduce a new function built from the definition of F by supplying it with additional arguments that *increase* its formal arity. This is made precise below.

Definition 5.8 (Raised Functions) *The operations $\mathcal{R}[\![\cdot]\!]\cdot$ and $F^{\oplus\cdot}$ are defined by simultaneous induction:*

- *Let B be a function body, $\vec{E}$ a list of expressions. The result of applying B to $\vec{E}$, denoted as $\mathcal{R}[\![B]\!]\vec{E}$ is given by*

$$
\begin{array}{lcl}
\mathcal{R}[\![x]\!]\vec{E} & = & x\,\vec{E} \\
\mathcal{R}[\![C\,\vec{D}]\!]\vec{E} & = & C\,\vec{D} \star \vec{E} \\
\mathcal{R}[\![G\,\vec{D}]\!]\vec{E} & = & G\,\vec{D} \star \vec{E},\ \textit{if}\ \delta \leq 0 \\
 & = & G^{\oplus\delta}\vec{D} \star \vec{E}\,,\ \textit{otherwise} \\
 & & \textit{where}\ \delta = |\vec{D}| + |\vec{E}| - \mathsf{arity}(G) \\
\mathcal{R}[\![x\ \vec{D}]\!]\vec{E} & = & x\ \vec{D} \star \vec{E} \\
\mathcal{R}[\![\mathsf{case}\ x\ \mathsf{of}\ \ldots P_i \to D_i \ldots]\!]\vec{E} & = & \mathsf{case}\ x\ \mathsf{of}\ \cdots P_i \to \mathcal{R}[\![D_i]\!]\vec{E} \cdots
\end{array}
$$

- *Let F be a function $F\,\vec{x} = B_F$, with arity n. The result of* raising the arity of F with k *is a function given by:*

$$F^{\oplus k}\,\vec{x} \star (y_1, \ldots, y_k) = \mathcal{R}[\![B_F]\!](y_1, \ldots, y_k)$$

Remark 5.9 Observe that this operation can trigger the creation of more raised functions if the new arguments $\vec{y}$ are added to an application that requires fewer than k arguments to become fully applied.

Now, the definition of substitution becomes straightforward.

Definition 5.10 (Substitution) *A substitution ρ is a function that assigns expressions to variables. This induces the following operation on expressions*

$$
\begin{array}{lcl}
\Sigma[\![x]\!]\rho & = & \rho(x) \\
\Sigma[\![F\,\vec{E}]\!]\rho & = & F\,\Sigma[\![\vec{E}]\!]\rho \\
\Sigma[\![C\,\vec{E}]\!]\rho & = & C\,\Sigma[\![\vec{E}]\!]\rho \\
\Sigma[\![x\ \vec{E}]\!]\rho & = & \mathcal{R}[\![\rho(x)]\!](\Sigma[\![\vec{E}]\!]\rho)
\end{array}
$$

If a substitution is applied to a function body B we have to be careful when B starts with a pattern match. For, the result of such a substitution does not lead to a valid expression if it substitutes a non-variable expression for the selector. We solve this problem by combining consumers and producers in a more sophisticated way. This has been done below. But first we introduce an auxiliary operation to perform pattern matching.

Definition 5.11 (Pattern Matching) *Let $F\,\vec{x} = E_F$ be a case function in i with arity k, and B be a function body. The result of substituting B for x_i in F, denoted as $\mathcal{M}_i[\![B]\!]\langle F\,\vec{x} = E_F\rangle$, is defined by induction on B in the following way.*

$$\begin{array}{lcl}
\mathcal{M}_i[\![y]\!]\langle F\,\vec{x} = \textsf{case}\ x_i\ \textsf{of}\ \ldots\rangle & = & \textsf{case}\ y\ \textsf{of}\ \ldots \\
\mathcal{M}_i[\![G\,\vec{E}]\!]\langle F\,\vec{x} = \ldots\rangle & = & F\,x_{1..i-1} \star (G\,\vec{E}) \star x_{i+1..k} \\
\mathcal{M}_i[\![y\ \vec{E}]\!]\langle F\,\vec{x} = \ldots\rangle & = & F\,x_{1..i-1} \star (y\ \vec{E}) \star x_{i+1..k} \\
\mathcal{M}_i[\![C_j\,\vec{E}]\!]\langle F\,\vec{x} = \textsf{case}\ x_i\ \textsf{of}\ \ldots C_j\,\vec{y_j} \rightarrow A_j \ldots\rangle & = & \Sigma[\![A_j]\!][\vec{E}/\vec{y_j}] \\
\mathcal{M}_i[\![\textsf{case}\ y_k\ \textsf{of}\ \ldots P_j \rightarrow E_j \ldots]\!]\langle F\,\vec{x} = E_F\rangle & & \\
\qquad = \textsf{case}\ y_k\ \textsf{of}\ \ldots P_j \rightarrow \mathcal{M}_i[\![E_j]\!]\langle F\,\vec{x} = E_F\rangle \ldots & &
\end{array}$$

Here $[E/x]$ denotes the substitution of E for x.

Definition 5.12 (Fused Functions) *Let F be a function, and let S be a function or constructor symbol. Assume that the arities of F, S are m, n respectively, and that S is applied to k arguments, $k \leq n$. The result of* fusing *F with such a k-ary version of S at argument position i, $i \leq m$ is a function F_iS^k defined as follows.*

1. *$F\,\vec{x} = \textsf{case}\ x_i\ \textsf{of}\ \cdots$. Then we distinguish the following two cases.*

   a) *S is a function, say with definition $S\,\vec{z} = B_S$. Then*

   $$F_iS^k\,x_{1..(i-1)} \star \vec{z} \star x_{(i+1)..m} = \mathcal{M}_i[\![B_S]\!]\langle F\,\vec{x} = \textsf{case}\ x_i\ \textsf{of}\ \cdots\rangle$$

   b) *S is a constructor. Then*

   $$F_iS^k\,x_{1..(i-1)}\star\vec{y}\star x_{(i+1)..m} = \mathcal{M}_i[\![S\,\vec{y}]\!]\langle F\,\vec{x} = \textsf{case}\ x_i\ \textsf{of}\ \cdots\rangle\ \ (|\vec{y}| = k)$$

2. *$F\,\vec{x} = E$. In that case*

   $$F_iS^k\,x_{1..(i-1)} \star \vec{y} \star x_{(i+1)..m} = \Sigma[\![E]\!][S\,\vec{y}/x_i]\ \ (|\vec{y}| = k)$$

Observe that the body of F in the latter case might start with a pattern match. But this pattern match is not on the variable x_i for which S is substituted, and hence this substitution will not produce an illegal function body.

Example 5.13 The body of the function app_app of example 5.4 results from applying rule 1a of definition 5.12:

$$\mathcal{M}_1[\![\textsf{case}\ l\ \textsf{of}\ \cdots]\!]\langle\textsf{app}(l,t) = \textsf{case}\ l\ \textsf{of}\ \cdots\rangle$$

As a result, the case of the consumer is pushed into the alternatives of the producer where it is eliminated, leading to:

$$\begin{array}{lclcl}
\textsf{app\_app}\ x\ y\ z & = & \textsf{case}\ x\ \textsf{of}\ \ \textsf{Nil} & \rightarrow & \textsf{app}\ y\ z \\
 & & \qquad\qquad \textsf{Cons}\ x\ xs & \rightarrow & \textsf{Cons}\ x\ (\textsf{app}\ (\textsf{app}\ xs\ y)\ z)
\end{array}$$

During fusion (parts of) the user defined functions are examined for consumer-producer pairs that can be fused. If such a fusion introduces a new function this new function itself also becomes a source for new consumer-producer pairs. This leads to the following algorithm.

Definition 5.14 (Fusion) *Let $\mathcal{F}$ be a set of functions. Evaluation of $\mathcal{F}$ consists of repeatedly performing the following three steps until no more consumer-producer pairs can be found (step 1 is no longer successful).*

1. *Look for a function body B in $\mathcal{F}$ that contains a* consumer-producer pair
$$R = F\, E_{1..(i-1)} \star (S\,\vec{D}) \star E_{(i+1)..|\vec{E}|}$$
2. *Let F_iS^k be the symbol tree that corresponds to that consumer-producer pair. Replace R by the expression*
$$F_iS^k\, E_{1..(i-1)} \star \vec{D} \star E_{(i+1)..|\vec{E}|}$$
3. *Set $\mathcal{F} = \mathcal{F} \cup \{F_iS^k\,\vec{z} = B'\}$.*
Here B' is given by the definitions 5.12. To determine whether F_iS^k is already present in $\mathcal{F}$ we use the equality on symbol trees; not on expressions, i.e. we do not compare function bodies.

Remark 5.15 We do not impose any evaluation order, since this order is irrelevant for the outcome. (See also property 5.18.)

Example 5.16 (Fusion) Consider the following functions.

$$\begin{array}{lcl}
\mathsf{plusOrMin}\ s\ m\ n & = & s\ (\mathsf{Pair\ Plus\ Min})\ m\ n \\
\mathsf{First}\ p & = & \mathsf{case}\ p\ \mathsf{of\ Pair}\ x\ y \rightarrow x \\
\mathsf{foo}\ m\ n & = & \mathsf{plusOrMin\ First}\ m\ n
\end{array}$$

The only consumer-producer pair occurs in foo. It will lead to the creation of a new function, $\mathsf{plusOrMin}_1\mathsf{First}^0$. In the new body of this new function the application of First will have 3 arguments, so the arity of First has to be increased by 2 in order to obtain a well-formed expression. Hence, the following two functions are generated.

$$\begin{array}{lcl}
\mathsf{plusOrMin}_1\mathsf{First}^0\ m\ n & = & \mathsf{First}^{\oplus 2}(\mathsf{Pair\ Plus\ Min})\ m\ n \\
\mathsf{First}^{\oplus 2}\ p\ m\ n & = & \mathsf{case}\ p\ \mathsf{of\ Pair}\ x\ y \rightarrow x\ m\ n
\end{array}$$

During the next step $\mathsf{First}^{\oplus 2}$ is fused with Pair resulting in a new function $\mathsf{First}_1^{\oplus 2}\mathsf{Pair}^2$ (in which the pattern match has been eliminated), and a replacement of the original pair in $\mathsf{plusOrMin}_1\mathsf{First}^0$.

$$\begin{array}{lcl} \mathsf{plusOrMin}_1\mathsf{First}^0\ m\ n & = & \mathsf{First}_1^{\oplus 2}\mathsf{Pair}^2\ \mathsf{Plus}\ \mathsf{Min}\ m\ n \\ \mathsf{First}_1^{\oplus 2}\mathsf{Pair}^2\ x\ y\ m\ n & = & x\ m\ n \end{array}$$

During the last two steps $\mathsf{First}_1^{\oplus 2}\mathsf{Pair}^2$ first consumes Plus followed by Min. The Plus will be applied to m, n whereas Min will disappear.

5.5 Basic properties of fusion

In this section we will briefly discuss some basic properties of fusion.

Soundness of fusion can be proved by first defining a semantics for our language, and then by showing that a fusion step of an expression leads to an expression that is semantically equivalent to its original.

As an example we will use a so called *natural operational* (or *big step*) semantics, specifying the result of a computation by means of syntax-driven derivation system. (See also [NN92, AJ01])

Definition 5.17 (Equivalence) *Let E, V be expressions, and let $E \Downarrow V$ denote that E* evaluates *to V (according to the underlying semantics). We say that two functions F, F' are* semantically equivalent *(notation $F \sim F'$) if for all expressions $\vec{E}$*

$$F\,\vec{E} \Downarrow V \Leftrightarrow F'\,\vec{E} \Downarrow V$$

The following property shows that fusion preserves semantics. It can be used, e.g. for proving that fusion is confluent (the order in which expressions are combined is not relevant).

Property 5.18 *Let $R = F\,(\ldots, S\,\vec{(\ldots)}, \ldots)$ be a consumer-producer pair, where S is used with arity k at argument position i of F. Let F_iS^k be the function obtained when F and S are fused. Then*

$$[\![F_iS^k]\!] \sim F_iS^k,$$

where the symbol tree and the function definition of F_iS^k are given by definition 5.5 and 5.12 respectively.

Evaluation by fusion leads to a subset of expressions, so called expressions in *fusion normal form*, or briefly *fusion normal forms.* Also functions bodies are subject to fusion, leading to more or less the same kind of results. These results are characterized by the following syntax.

Definition 5.19 (Fusion Normal Form)

- *The set of expressions in* fusion normal form (*FNF*) *is defined as follows.*
$$\begin{array}{lcl} N & ::= & N' \mid F\,\vec{N'} \mid C\,\vec{N} \\ N' & ::= & v \mid v\,\vec{N} \end{array}$$

- *Function bodies in FNF have the following shape.*
$$\begin{array}{lcl} N_B & ::= & N \mid \textsf{case}\ x\ \textsf{of}\ P_1 \rightarrow N_1 \cdots P_k \rightarrow N_k \\ P & ::= & C\,\vec{x} \end{array}$$

- *A function is in FNF if its body is, and a collection of functions is in FNF if all functions are.*

Remark 5.20 Observe that, in this form, functions are only applied to variables and higher-order applications, and never to constructors or functions.

In chapter 4 a relation is established between the typing of an expression and the data constructors it contains after symbolic evaluation: an expression in symbolic normal form does not contain any data constructors that are not included in a typing for that expression (see the deforestation property 4.14). In case of fusion normal forms (*FNF*s), we can derive a similar property, although *FNF*s may still contain function applications. More specifically, let $C_N(N)$ denote the collection of data constructors of the (body) expression N, and $C_T(\sigma)$ denote the data constructors belonging to the type σ. (For a precise definition of $C_N(\cdot), C_T(\cdot)$, see 4.11). Then we have the following property.

Property 5.21 (Typing FNF)

- *Let N be an expression in FNF. Suppose N is typable, i.e. for some basis B and type σ we have $B \vdash N : \sigma$. Then*
$$C_N(N) \subseteq C_T(B) \cup C_T(\sigma).$$

- *Let $\mathcal{F}$ be a collection of functions in FNF. Suppose $F \in \mathcal{F}$ has type $\vec{\sigma} \rightarrow \tau$. Then*
$$C_N(F) \subseteq C_T(\vec{\sigma}) \cup C_T(\tau).$$

We can use this property in the following way. Let $\mathcal{F}$ be a set of functions. The first step is to apply fusion to the body of each function $F\,\vec{x} = E_F \in \mathcal{F}$.

Then (standard) evaluation of any application of F will only involve objects using data constructors that are contained in the typing for F. More specifically, if F is an instance of a generic function on a user defined data type, then fusion will remove all data constructors of the base types $\{\rightleftarrows, \mathbb{1}, \times, +\}$, provided that neither the generic type of the function nor the instance type itself contains any of these base types.

5.6 Guaranteeing termination

Without any precautions the process of repeatedly eliminating consumer producer pairs might not terminate, or in our setting, will generate an infinite number of new functions.

Standard fusion

To avoid non-termination we will not reduce all possible pairs but restrict reduction to pairs in which only *proper* consumers and producers are involved. In [AGS03] a separate analysis phase is used to determine proper consumers and producers. The following definitions are more or less directly taken from [AGS03]

Definition 5.22 (Active Parameter) *The notions of* active occurrence *and* active parameter *are defined by simultaneous induction.*

- *We say that a variable x occurs actively in a (body) expression B if there exists a subexpression $E \subseteq B$ such that*
  - $E = \mathsf{case}\ x\ \mathsf{of}\ \ldots$, *or*
  - $E = x\ \ldots$, *or*
  - $E = F\,\vec{D}$, *such that $D_i = x$ and act(F)$_i$.*

  By $\mathrm{AV}(E)$ *we denote the set of active variables occurring in E.*

- *Let $F\,\vec{x} = B_F$ be a function. F is active in x_i (notation act(F)$_i$) if $x_i \in \mathrm{AV}(B_F)$.*

The notion of *accumulating parameter* is used to detect potentially growing recursion.

Definition 5.23 (Accumulating Parameter) *Let* $F_1 = B_1, \ldots, F_n = B_n$ *be a set of mutually recursive functions. The function* $F = F_j$ *is* accumulating *in its* i^{th} *parameter (notation* $acc(F)_i$*) if either*

- *there exists a right-hand side* B_k *such that* $F\vec{D} \subseteq B_k$*,and* D_i *is open but not just a variable (i.e.* $z \subset D_i$ *variable* z*), or*
- *there exists a subexpression* $F_k\vec{D} \subseteq B_j$ *such that* F_k *is accumulating in* l*, and* $D_l = x_i$*.*

Observe that the active as well as the accumulating predicate are defined recursively. This will amount to solving a least fixed point equation with respect to the ordering 'false' $\leq$ 'true'.

Definition 5.24 (Proper Consumer) *A function* F *is a* proper consumer *in its* i^{th} *parameter (notation* $con(F)_i$*) if* $act(F)_i$ *and* $\neg acc(F)_i$*.*

Definition 5.25 (Proper Producer) *Let* $F_1, \ldots, F_n$ *be a set of mutually recursive functions with respective right-hand sides* $B_1, \ldots B_n$*.*

- *A body* B_k *is called* unsafe *if it contains a subexpression* $G\vec{E}$*, such that* $con(G)_i$ *and* $E_i = F_j(\cdots)$*, for some* G, j*. In words:* B_k *contains a call to* F_j *on a consuming position.*
- *All functions* F_k *are* proper producers *if none of their right-hand sides is unsafe. Hence, if one of the bodies is unsafe, the complete set becomes improper.*

Remark 5.26 It is important to note that non-recursive functions are *always* proper producers.

Example 5.27 The well-know function for reversing the elements of a list can be defined in two different ways. In the first definition an auxiliary function rev2 is used.

```
rev l      =  rev2 l Nil
rev2 l a   =  case l of  Nil        →  a
                         Cons x xs  →  rev2 xs (Cons x a)
```

Both rev and rev2 are proper producers. The second definition uses app.

$$\begin{array}{lclcl} \mathsf{rev}\ l & = & \mathsf{case}\ l\ \mathsf{of} & \mathsf{Nil} & \rightarrow & \mathsf{Nil} \\ & & & \mathsf{Cons}\ x\ xs & \rightarrow & \mathsf{app}\ (\mathsf{rev}\ xs)\ (\mathsf{Cons}\ x\ \mathsf{Nil}) \end{array}$$

Now rev is no longer a proper producer: the recursive call to rev appears on a consuming position, since app is consuming in its first argument. Consequently a function like foo l = len (rev l) with

$$\begin{array}{lclcl} \mathsf{len}\ l & = & \mathsf{case}\ l\ \mathsf{of} & \mathsf{Nil} & \rightarrow & 0 \\ & & & \mathsf{Cons}\ x\ xs & \rightarrow & 1 + \mathsf{len}\ xs \end{array}$$

will only be transformed if rev is defined in the first way. By the way, the effect of the transformation w.r.t. the gain in efficiency is almost negligible.

5.7 Improved Consumer Analysis

If functions are not too complex, standard fusion will produce good results. In particular, this also holds for many generic functions. However, in some cases the fusion algorithm fails due to both consumer and producer limitations. We will first examine what can go wrong with the current consumer analysis. For this reason we have adjusted the definition of app slightly.

Example 5.28

$$\begin{array}{lclcl} \mathsf{app}\ l\ t & = & \mathsf{case}\ l\ \mathsf{of} & \mathsf{Nil} & \rightarrow & t \\ & & & \mathsf{Cons}\ x\ xs & \rightarrow & \mathsf{app2}\ (\mathsf{Pair}\ x\ xs)\ t \\ \mathsf{app2}\ p\ t & = & \multicolumn{3}{l}{\mathsf{case}\ p\ \mathsf{of}\ \mathsf{Pair}\ x\ xs \rightarrow \mathsf{Cons}\ x\ (\mathsf{app}\ xs\ t)} \end{array}$$

Due to the intermediate Pair constructor the function app is no longer a proper consumer. (The (indirect) recursive call has this active pair as an argument and the non-accumulating requirement prohibits this.)

It is hard to imagine that a normal programmer will write such a function directly. However, keep in mind that the optimization algorithm, when applied to a generic function, introduces many intermediate functions that communicate with each other via basic sum and product constructors. For exactly this reason many relatively simple generic functions cannot be optimized fully.

One might think that a simple inlining mechanism should be capable of removing the Pair constructor. In general, such 'append-like' functions will appear as an intermediate result of the fusion process. Hence, this inlining should be combined with fusion itself which makes it much more problematic. Experiments with very simple inlining show that it is practically impossible to avoid non-termination for the combined algorithm.

To solve the problem illustrated above, we extend fusion with *depth analysis*. Depth analysis is a refinement of the accumulation check (definition 5.23). The original accumulation check is based on a purely syntactic criterion. The improved accumulation check takes into account how the size of the result of a function application increases or decreases with respect to each argument. The idea is to count how many times constructors and destructors (pattern matches) are applied to each argument of a function. If this does not lead to an 'infinite' depth (an infinite depth is obtained if a recursive call extends the argument with one or more constructors) accumulation is still harmless.

Definition 5.29 (Depth) *The functions occ and dep are specified below by simultaneous induction.*

$$
\begin{array}{lcll}
occ(v,x) & = & 0, & \textit{if } v = x \\
 & = & \bot, & \textit{otherwise} \\
occ(v, C\,\vec{E}) & = & \max_i(1 + occ(v, E_i)) & \\
occ(v, F\,\vec{E}) & = & \max_i(dep(F)_i + occ(v, E_i)) & \\
occ(v, x\;\vec{E}) & = & \max(occ(v,x), \max_i(occ(v, E_i))) & \\
\multicolumn{4}{l}{occ(v, \mathsf{case}\ x\ \mathsf{of}\ \ldots C_i\vec{y} \to E_i \ldots)} \\
 & = & \max(-\infty, \max_i(\max(occ(v, E_i)), & \\
 & & \qquad \max_k(occ(y_k, E_i)) - 1))), & \textit{if } v = x \\
 & = & \max_i(occ(v, E_i)), & \textit{otherwise}
\end{array}
$$

Moreover, for each function $F\,\vec{x} = B_F$

$$dep(F)_i = occ(x_i, B_F)$$

using $\bot + x = \bot$, $\max() = \bot$, *and* $(-\infty) + (+\infty) = +\infty$.

Remark 5.30 These two functions are defined as a fixed point equation on $\bot \cup \mathbb{Z} \cup \{+\infty, -\infty\}$, with $\bot \leq -\infty \leq z \leq +\infty$ for all $z \in \mathbb{Z}$. An implementation of this fixed point construction has to limit the domain to a finite subset of $\mathbb{Z}$, extended with $\bot$. The boundaries of this subset can be determined on basis of the structure of the function bodies.

Example 5.31 The depths of the two functions appearing in example 5.28 are $dep(\mathsf{app}) = dep(\mathsf{app2}) = (0, +\infty)$.

The following definition gives an improved version of the accumulation property (definition 5.23).

Definition 5.32 (Accumulating With Depth Analysis)
Let $F_1 = B_1, \ldots, F_n = B_n$ be a set of mutually recursive functions. The function $F = F_j$ is accumulating *in its i^{th} parameter (notation $acc(F)_i$) if either*

1. *$dep(F)_i = +\infty$, or*
2. *for some k there exists an expression $F\vec{D} \subseteq B_k$ such that $\mathrm{AV}(D_i) \neq \emptyset$, and D_i is not just a variable, or*
3. *there exists a subexpression $F_k\vec{D} \subseteq B_j$ such that F_k is accumulating in l, and $D_l = x_i$.*

5.8 Improved Producer Analysis

In some cases not the consumer but the producer classification (definition 5.25) is responsible for not getting optimal transformation results. The problem occurs, for instance, when the type of a generic function contains recursive type constructors. Take, for example, the monadic mapping function for the list monad mapl. The base type of mapl is

$$\textbf{type}\ \mathrm{MapL}\ a\ b = a \rightarrow \mathsf{List}\ b$$

Recall that the specialization of mapl to any data type, e.g. Tree, will use the embedding-projection specialized to MapL (see section 5.2). This embedding projection is based on $\mathsf{ep}_{\mathsf{List}}$: the generic embedding projection specialized to lists. Since List is recursive, $\mathsf{ep}_{\mathsf{List}}$ is recursive as well. Moreover, one can easily show the recursive call to $\mathsf{ep}_{\mathsf{List}}$ appears on a consuming position, and hence $\mathsf{ep}_{\mathsf{List}}$ is not a proper producer. As a consequence, the transformation of a specialized version of mapl gets stuck when it hits on $\mathsf{ep}_{\mathsf{List}}$ appearing as a producer. We illustrate the essence of the problem with a much simpler example based on the data type:

$$\textbf{data}\ \mathsf{Id}\ a = \mathsf{Id}\ a$$

Example 5.33 (Improper producer) Consider the following set of functions.

```
unId i  =  case i of Id x → x
foo     =  Id (unId foo)
bar     =  unId foo
```

Obviously, the function unId is consuming in its argument. Since the recursive call to foo appears as an argument of unId, this function foo is an improper producer. Consequently, the right-hand side of bar cannot be optimized. On the other hand, it seems to be harmless to ignore the producer requirement in this case and to perform a fusion step. As long as we do not evaluate too far termination is not a problem. But how do we prevent of getting into a non-terminating reduction sequence, in case we are dealing with a situation that is less clear?

The solution to this problem is simple: allow improper producers to be unfolded once. But how do we detect whether we have already performed such an unfold step? Actually, this is not as easy as it seems. The transformation algorithm could be parameterized with some kind of *evaluation history* of the improper producers that were unfolded in order to obtain the current expression. However, such a history will make the outcome of the transformation sensible to the evaluation order, which makes reasoning about the transformation much more difficult.

In our transformation algorithm, however, we can use our special tree representation of new function symbols as a substitute for the evaluation history. Remember that a symbol tree contains the information of how the corresponding function was created in terms of the initial set functions and data constructors. Suppose we have a fusion pair consisting of a function F consuming in its i^{th} argument and an improper producer G, say with arity k. The idea is to detect possible non-termination by examining the symbol tree F_iG^k. If this tree contains a cyclic occurrence of some improper producer, we don't fuse; otherwise a fusion step is performed. This leads to the following improved fusion algorithm.

Definition 5.34 (Improved producer analysis) *Let $\mathcal{F}$ be a set of functions.*

- *Let T be a symbol tree. Such a tree is called* unsafe *if there exists an improper producer, say with symbol tree G, such that T is cyclic in $\ulcorner G \urcorner$. Otherwise the tree is called* safe.
- *Let $F\,\vec{x} = B_F \in \mathcal{F}$. A safe* consumer-producer pair *in F is an expression $R \subseteq B_F$ of the form*

$$R = G\,E_{1..(i-1)} \star (S\,\vec{D}) \star E_{(i+1)..|\vec{E}|}$$

such that $con(G)_i$, and for S one of the following properties holds:

- *S is a constructor, or*
- *S is partially applied function (i.e.* $\mathsf{arity}(S) < |\vec{D}|$*), or*
- *S is a proper producer, or*
- *$S \notin \widehat{F}$ and $F_i S^{|\vec{D}|}$ is safe.*

- *Only safe consumer-producer pairs are fused.*

Remark 5.35 We can further improve fusion by replacing unused arguments of functions with $\bot$. More precisely, let $G\,\vec{E}$ be an expression such that E_i is not just a variable. If $dep(G)_i = \bot$ it is safe to replace this expression by

$$G\,E_{1..(i-1)} \star (\bot) \star E_{(i+1)..|\vec{E}|}$$

Remark 5.36 The last requirement in the definition of redex is that the application of an improper producer S occurs outside the component to which S belongs. ($S \notin \widehat{F}$) This requirement is not essential for guaranteeing termination, but it leads to better results for fusing generics.

To illustrate the effect of our refinement we go back to example 5.33. Now, the application in the body of bar is a redex. It will be replaced by $\mathsf{unId}_1\mathsf{foo}^0$, and a new function for this symbol is generated. Following the rules for the introduction of new functions (definition 5.12) the initial body of this function is unId foo, indeed, identical to the expression from which it descended. Again the expression will be recognized as a redex and replaced by $\mathsf{unId}_1\mathsf{foo}^0$, finishing the fusion process.

Properties

Since we no longer fuse *all* consumer-producers pairs but restrict ourselves to *proper* consumers and producers we cannot expect that the result of a fused expression will always be in *FNF*(as defined in definition 5.19). Consequently, such a result might still contain data constructors that we were trying to eliminate. Assume that initially all functions are consuming in all their arguments, and that all functions appearing on a consuming position are proper producers. Even then it is still not guaranteed that fusion leads to *FNF*. During fusion new functions are introduced which do not necessarily fulfill these requirements or the properties of existing functions might change. Take for instance the function foo from the previous example (5.33). An alternative (and equivalent) definition for this function using the Y-combinator

$$Y\ f = f\ (Y\ f)$$

is

$$\mathsf{foo} = Y\ (\mathsf{Id} \circ \mathsf{unId}).$$

Now, the example does not contain any improper producers anymore. However, fusing the body of foo will introduce an auxiliary function identical to the original version of foo. And, as we have seen, this function is not a proper producer. Observe that the body of an improper producer is not in *FNF*, even after is has been optimized. Hence, the characterization of fusion normal forms is no longer correct.

We solve this problem by first giving a more liberal classification of the fusion results. Remember that our main concern is not to eliminate all consumer–producer pairs, but only those communicating intermediate objects caused by the structural representation of data types. The new notion of fusion normal forms is based on the types of functions and data constructors.

Definition 5.37 (T-Free Forms) *Let T be a type constructor.*

- *Let S be a function or data constructor, say with arity n, and type $\vec{\sigma} \to \tau$, where $|\vec{\sigma}| = n$. We say that a k-ary version of S* excludes *T, $k \leq n$, (notation $S \not\supseteq_k T$) if*

  $$C_T(\sigma_{k+1}, \ldots, \sigma_n, \tau) \cap C_T(T) = \emptyset.$$

  We abbreviate $S \not\supseteq_n T$ to $S \not\supseteq T$.

- *The set N_T of expressions in T-free form is defined as:*

  $$\begin{array}{lcl} N_T & ::= & N'_T \mid F\,\vec{N'_T} \mid C\,\vec{N_T} \\ N'_T & ::= & v \mid v\ \vec{N_T} \mid S\,\vec{N'_T} \end{array}$$

  with the additional restriction that for each application of $S\,\vec{N'_T}$ it holds that $S \not\supseteq_{|\vec{N'_T}|} T$.

- *A function is in T-FF if its body is.*

In the next section we show why this new notion of T-*FF* is sufficient to obtain the desired result in case of generic functions. This notion enables us to reason about fusion in a more abstract way. For instance, we can now investigate how an improper producer is combined with its surrounding context, and that this combination again will be in the required form.

For functions in T-*FF* we have a property comparable to property 5.21 of functions in *FNF*.

Property 5.38 *Let T be type constructor, and $\mathcal{F}$ be a collection of functions in T-FF. Then, for any $F \in \mathcal{F}$ we have*

$$F \not\supseteq T \;\Rightarrow\; C_N(F) \cap C_T(T) = \emptyset$$

5.9 Fusion of Generic Instances

In this section we deal with the optimization of instances generated by the generic specializer. An instance is considered to be optimized if the resulting code does not contain constructors belonging to the basic types $\{\rightleftarrows, \mathbb{1}, \times, +\}$. Our goal is to show that under some conditions on the generic base cases, the generic function types, and the instance types the presented fusion algorithm completely removes generic overhead.

Let g be a generic function of type G, and let T be a type constructor. Consider the specialization of g to T. As mentioned in section 5.2, a generated instance consists of an adaptor and the code for the structural representation and has the shape

$$g_T \vec{f} = \mathsf{adapt}_{\langle G,T\rangle}\ (g_{T^\circ}\ \vec{f})$$

The generic constructors that we want to eliminate are $\mathsf{EP}, \mathsf{Pair}, \mathsf{Inl}, \mathsf{Inr}$ and Unit. EP can be found in the adaptor only, whereas the other constructors appear in both adaptor and g_{T°.

In practice, optimizing g_T can only be successful if there are no EPs left in the adaptor $\mathsf{adapt}_{\langle G,T\rangle}$. Therefore, we start with examining how the adaptor is optimized.

5.9.1 Fusing adaptors

We can split the adaptor in two parts corresponding to G and T respectively. The adaptor has the general shape

$$\mathsf{adapt}_{\langle G,T\rangle} \quad = \quad \mathsf{adapt}_{\langle G\rangle}\ \mathsf{adapt}_{\langle T\rangle}\ \ldots\ \mathsf{adapt}_{\langle T\rangle}$$

where *adaptT* is repeated for each (generic) argument of G.

$$\begin{array}{lcl}
\mathsf{adapt}_{\langle G\rangle} & :: & (a \rightleftarrows b) \to \ldots \to (a \rightleftarrows b) \to (G\ a \ldots a) \to (G\ b \ldots b) \\
\mathsf{adapt}_{\langle G\rangle}\ \vec{x} & = & \mathsf{from}\ (\mathsf{ep}_G\ \vec{x}) \\
\mathsf{adapt}_{\langle T\rangle} & :: & T\ \vec{a} \rightleftarrows T^\circ\ \vec{a} \\
\mathsf{adapt}_{\langle T\rangle} & = & \mathsf{conv}_T
\end{array}$$

Here ep_G is the specialization of ep to G, see section 5.2. Note that there are no basic types appearing in the result type $(G\ a \ldots a) \to (G\ b \ldots b)$ of $\mathsf{adapt}_{\langle G\rangle}$. If we are able to show that $\mathsf{adapt}_{\langle G\rangle}$ is fused to $\{\rightleftarrows, \mathbb{1}, \times, +\}$-*FF*, we have eliminated all basic constructors (by property 5.38). Further in this subsection, we limit ourselves to elimination of EPs ($\rightleftarrows$-*FF*).

The instance ep_G is built from the base cases for the generic function ep, and instances of the form ep_P, where P is a type constructor appearing in G. We will first focus on the structure of ep_P Note that if the type P is recursive, the generic instance ep_P will be recursive as well. Since ep_P is a generic instance, it can be written as

$$\mathsf{ep}_P\ \vec{f} = \mathsf{adapt}_{\langle\rightleftarrows,P\rangle}\ (\mathsf{ep}_{P^\circ}\ \vec{f})$$

where the adaptor has the form

$$\begin{array}{lcl} \mathsf{adapt}_{\langle\rightleftarrows,P\rangle} & = & \mathsf{adapt}_{\langle\rightleftarrows\rangle}\ \mathsf{conv}_P\ \mathsf{conv}_P \\ \mathsf{adapt}_{\langle\rightleftarrows\rangle}\ a\ b & = & \mathsf{from}\ (\mathsf{ep}_{\rightleftarrows}\ a\ b) \end{array}$$

It is easy to show that the function $\mathsf{adapt}_{\langle\rightleftarrows,P\rangle}$ can be written as

$$\mathsf{adapt}_{\langle\rightleftarrows,P\rangle} \quad = \quad \mathsf{EP}\ (\mathsf{epto}_{\rightleftarrows,P})\ (\mathsf{epfrom}_{\rightleftarrows,P})$$

where

$$\begin{array}{lcl} \mathsf{epto}_{\rightleftarrows,P}\ e & = & \mathsf{mapAR}\ \mathsf{convTo}_P\ \mathsf{convFrom}_P\ (\mathsf{to}\ e) \\ \mathsf{epfrom}_{\rightleftarrows,P}\ e & = & \mathsf{mapAR}\ \mathsf{convTo}_P\ \mathsf{convFrom}_P\ (\mathsf{from}\ e) \end{array}$$

Fusion of the original $\mathsf{adapt}_{\langle\rightleftarrows,P\rangle}$ leads to a more or less similar result.

In this subsection our goal is to show that the resulting code for adaptors is EP free, i.e in $\rightleftarrows$-*FF*. We illustrate how fusion eliminates intermediate EPs by means of examples. The general case can be treated similarly, but is omitted because it does not help the explanation. The adaptor is built from the combination of EP projections (to and from) and EP instances (e.g. $\mathsf{ep}_{\mathsf{List}}$). The first example shows how the to projection of EP is fused with a recursive instance. The second example shows two recursive instances of EP are fused together. And the third example shows how to is fused with the combinations of two recursive instances. This should convince the reader, that the transformation leads to the adaptors that are free from EPs.

Example 5.39 (Fusing a projection with an instance)
We assume that the instance on lists $\mathsf{ep}_{\mathsf{List}}$ is already fused and is in $\{+, \times, \mathbb{1}\}$-*FF*.

$$
\begin{array}{lcl}
\mathsf{ep}_{\mathsf{List}}\ f & = & \mathsf{EP}\ (\mathsf{epto}_{\mathsf{List}}\ f\ (\mathsf{ep}_{\mathsf{List}}\ f))\ (\mathsf{epfrom}_{\mathsf{List}}\ f\ (\mathsf{ep}_{\mathsf{List}}\ f)) \\
\mathsf{epto}_{\mathsf{List}}\ f\ r\ l & = & \mathsf{case}\ l\ \mathsf{of}\ \begin{array}[t]{lcl} \mathsf{Nil} & \rightarrow & \mathsf{Nil} \\ \mathsf{Cons}\ h\ t & \rightarrow & \mathsf{Cons}\ (\mathsf{to}\ f\ h)\ (\mathsf{to}\ r\ t) \end{array} \\
\mathsf{epfrom}_{\mathsf{List}}\ f\ r\ l & = & \mathsf{case}\ l\ \mathsf{of}\ \begin{array}[t]{lcl} \mathsf{Nil} & \rightarrow & \mathsf{Nil} \\ \mathsf{Cons}\ h\ t & \rightarrow & \mathsf{Cons}\ (\mathsf{from}\ f\ h)\ (\mathsf{from}\ r\ t) \end{array}
\end{array}
$$

Consider the application $\mathsf{to}\ (\mathsf{ep}_{\mathsf{List}}\ f)$. Fusion will introduce a function $\underline{\mathsf{to}\ \mathsf{ep}_{\mathsf{List}}}$ (we indicate new symbols by underlining the corresponding consumer and producer, and also leave out the argument number and the actual arity of the producer). The body of this function is optimized as follows:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\underline{\mathsf{to}\ \mathsf{ep}_{\mathsf{List}}}\ f\ l} \\
\rightsquigarrow & \{\textit{unfolding}\ \mathsf{ep}_{\mathsf{List}}\} \\
 & \mathsf{to}\ (\mathsf{EP}\ (\mathsf{epto}_{\mathsf{List}}\ f\ (\mathsf{ep}_{\mathsf{List}}\ f))\ (\mathsf{epfrom}_{\mathsf{List}}\ f\ (\mathsf{ep}_{\mathsf{List}}\ f)))\ l \\
\rightsquigarrow & \{\textit{unfolding}\ \mathsf{to}\} \\
 & \mathsf{epto}_{\mathsf{List}}\ f\ (\mathsf{ep}_{\mathsf{List}}\ f)\ l \\
\rightsquigarrow & \{\textit{unfolding}\ \mathsf{epto}_{\mathsf{List}}\} \\
 & \mathsf{case}\ l\ \mathsf{of}\ \begin{array}[t]{lcl} \mathsf{Nil} & \rightarrow & \mathsf{Nil} \\ \mathsf{Cons}\ h\ t & \rightarrow & \mathsf{Cons}\ (\mathsf{to}\ f\ h)\ (\mathsf{to}\ (\mathsf{ep}_{\mathsf{List}}\ f)\ t) \end{array} \\
\rightsquigarrow & \{\textit{folding}\ \mathsf{to}, \mathsf{ep}_{\mathsf{List}}\} \\
 & \mathsf{case}\ l\ \mathsf{of}\ \begin{array}[t]{lcl} \mathsf{Nil} & \rightarrow & \mathsf{Nil} \\ \mathsf{Cons}\ h\ t & \rightarrow & \mathsf{Cons}\ (\mathsf{to}\ f\ h)\ (\underline{\mathsf{to}\ \mathsf{ep}_{\mathsf{List}}}\ f\ t) \end{array}
\end{array}
$$

Obviously, the resulting code is in $\rightleftarrows$-*FF*, and due to property 5.38 this function will not generate any EP-constructor.

Example 5.40 (Fusing two instances)
We assume fusion of the instance for the list and tree types. The instance for tree after fusion is

$$
\begin{array}{lcl}
\mathsf{ep}_{\mathsf{Tree}}\ f & = & \mathsf{EP}\ (\mathsf{epto}_{\mathsf{Tree}}\ f\ (\mathsf{ep}_{\mathsf{Tree}}\ f))\ (\mathsf{epfrom}_{\mathsf{Tree}}\ f\ (\mathsf{ep}_{\mathsf{Tree}}\ f)) \\
\mathsf{epto}_{\mathsf{Tree}}\ f\ r\ t & = & \mathsf{case}\ t\ \mathsf{of} \\
 & & \quad \begin{array}{lcl} \mathsf{Leaf}\ x & \rightarrow & \mathsf{Leaf}\ (\mathsf{to}\ f\ x) \\ \mathsf{Branch}\ x\ y & \rightarrow & \mathsf{Branch}\ (\mathsf{to}\ r\ x)\ (\mathsf{to}\ r\ y) \end{array} \\
\mathsf{epfrom}_{\mathsf{Tree}}\ f\ r\ t & = & \mathsf{case}\ t\ \mathsf{of} \\
 & & \quad \begin{array}{lcl} \mathsf{Leaf}\ x & \rightarrow & \mathsf{Leaf}\ (\mathsf{from}\ f\ x) \\ \mathsf{Branch}\ x\ y & \rightarrow & \mathsf{Branch}\ (\mathsf{from}\ r\ x)\ (\mathsf{from}\ r\ y) \end{array}
\end{array}
$$

Fusion of $\mathsf{ep_{Tree}}\ (\mathsf{ep_{List}}\ f)$ proceeds as follows.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\underline{\mathsf{ep_{Tree}}\ \mathsf{ep_{List}}}\ f} \\
\rightsquigarrow & \{\textit{unfolding}\ \mathsf{ep_{Tree}}\} \\
 & \mathsf{EP}\ (\mathsf{epto_{Tree}}\ (\mathsf{ep_{List}}\ f)\ (\mathsf{ep_{Tree}}\ (\mathsf{ep_{List}}\ f)))\ (\ldots) \\
\rightsquigarrow & \{\textit{folding}\ \mathsf{ep_{Tree}}, \mathsf{ep_{List}}\} \\
 & \mathsf{EP}\ (\mathsf{epto_{Tree}}\ (\mathsf{ep_{List}}\ f)\ (\underline{\mathsf{ep_{Tree}}\ \mathsf{ep_{List}}}\ f))\ (\ldots) \\
\rightsquigarrow & \{\textit{fusing}\ \mathsf{epto_{Tree}}, \mathsf{ep_{List}}\} \\
 & \mathsf{EP}\ (\underline{\mathsf{epto_{Tree}}\ \mathsf{ep_{List}}}\ f\ (\underline{\mathsf{ep_{Tree}}\ \mathsf{ep_{List}}}\ f))\ (\ldots)
\end{array}
$$

where fusion of $\mathsf{epto_{Tree}}\ (\mathsf{ep_{List}}\ f)$ proceeds as

$$
\begin{array}{lllll}
\multicolumn{5}{l}{\underline{\mathsf{epto_{Tree}}\ \mathsf{ep_{List}}}\ f\ r\ l} \\
\rightsquigarrow & \multicolumn{4}{l}{\{\textit{unfolding}\ \mathsf{epto_{Tree}}\}} \\
 & \mathsf{case}\ t\ \mathsf{of} & \mathsf{Leaf}\ x & \rightarrow & \mathsf{Leaf}\ (\mathsf{to}\ (\mathsf{ep_{Tree}}\ f)\ x) \\
 & & \mathsf{Branch}\ x\ y & \rightarrow & \mathsf{Branch}\ (\mathsf{to}\ r\ x)\ (\mathsf{to}\ r\ y) \\
\rightsquigarrow & \multicolumn{4}{l}{\{\textit{folding}\ \mathsf{to}, \mathsf{ep_{List}}\}} \\
 & \mathsf{case}\ t\ \mathsf{of} & \mathsf{Leaf}\ x & \rightarrow & \mathsf{Leaf}\ (\underline{\mathsf{to}\ \mathsf{ep_{List}}}\ f\ x) \\
 & & \mathsf{Branch}\ x\ y & \rightarrow & \mathsf{Branch}\ (\mathsf{to}\ r\ x)\ (\mathsf{to}\ r\ y)
\end{array}
$$

Fusion has eliminated an intermediate EP produced by $\mathsf{ep_{List}}$ and consumed by $\mathsf{ep_{Tree}}$ from the original expression $\mathsf{ep_{Tree}}\ (\mathsf{ep_{List}}\ f)$.

Example 5.41 (Projection of fused instances)
Fusion of to with $\underline{\mathsf{ep_{Tree}}\ \mathsf{ep_{List}}}$ is similar to the first example. It yields

$$
\begin{array}{lllll}
\underline{\mathsf{to}\ \underline{\mathsf{ep_{Tree}}\ \mathsf{ep_{List}}}}\ f\ t & = & \multicolumn{3}{l}{\mathsf{case}\ t\ \mathsf{of}} \\
 & & \mathsf{Leaf}\ x & \rightarrow & \mathsf{Leaf}\ (\underline{\mathsf{to}\ \mathsf{ep_{List}}}\ f\ x) \\
 & & \mathsf{Branch}\ l\ r & \rightarrow & \mathsf{Branch}\ (\underline{\mathsf{to}\ \underline{\mathsf{ep_{Tree}}\ \mathsf{ep_{List}}}}\ f\ l) \\
 & & & & \qquad\qquad (\underline{\mathsf{to}\ \underline{\mathsf{ep_{Tree}}\ \mathsf{ep_{List}}}}\ f\ r)
\end{array}
$$

These examples illustrate that fusion of adaptors leads to $\rightleftarrows$-*FF*. Provided that the generic and the instance types do not involve EPs, the adaptor is the only part of a generated instance that originally contains EPs. Therefore, fusion transforms instances into $\rightleftarrows$-*FF*.

5.9.2 Requirements

As said before, not all adaptors can be fused to $\rightleftarrows$-*FF*. In fact, when certain type constructors are involved in a generic type, that generic type results in an adaptor that cannot be optimized to $\rightleftarrows$-*FF*. Namely, (1) *nested* and (2) *contra-variantly recursive* types lead to such adaptors. We explain both kinds with an example.

- *Nested types* are types like

  $$\textbf{data}\ \mathsf{Nest}\ a = \mathsf{NNil} \mid \mathsf{NCons}\ a\ (\mathsf{Nest}\ (a, a))$$

  i.e. recursive types in which the arguments of the recursive occurrence(s) are not just variables. The ep that is generated for Nest is

  $$\begin{aligned}\mathsf{ep}_{\mathsf{Nest}}\ a \quad = \quad & \mathsf{from}\ (\mathsf{ep}_{\mathsf{EP}}\ \mathsf{conv}_{\mathsf{Nest}}\ \mathsf{conv}_{\mathsf{Nest}}) \\ & \qquad (\mathsf{ep}_{+}\ \mathsf{ep}_{\mathbb{1}}\ (\mathsf{ep}_{\times}\ a\ (\mathsf{ep}_{\mathsf{Nest}}(\mathsf{ep}_{(,)}\ a\ a))))\end{aligned}$$

  This function is accumulating, and hence not a proper consumer. Fusion will therefore not be able to eliminate all EPs in a term like $\mathsf{ep}_{\mathsf{Nest}}\ \mathsf{conv}_T$.

- *Contra-variantly recursive types* are types like

  $$\textbf{data}\ \ \mathsf{Contra} = \mathsf{Contra}\ (\mathsf{Contra} \rightarrow \mathrm{Int})$$

  i.e. recursive types in which one or more of the recursive occurrence(s) appear on a contra-variant position (the first argument of the →-constructor). We will not go into further details to explain why instance of ep for these types are not of the right form.

It is important that these requirements are on type constructors that occur in the generic type, but not on the instance types. We believe that all the above requirements on type constructors are not very restrictive in practice.

Apart from these type constructor requirements, we have the additional restriction that the type G of the generic function itself if free of *self-application.* Self application of types means applying a type constructor to itself, e.g. List (List a). Self application of types will lead to self-application of functions, in particular of embedding projections. The problem is that most eps are improper producers, and hence a nested application of such functions will immediately create a cyclic symbol tree of the corresponding consumer-producer pair. It will therefore not be accepted for fusion. Consider, for instance, a generic non-deterministic parser. This parser could have the following type.

$$\textbf{type}\ \ \mathsf{Parser}\ a = (\mathsf{List}\ \mathrm{Char}) \rightarrow \mathsf{List}\ (a, \mathsf{List}\ \mathrm{Char})$$

This leads to the following embedding projection

$$\mathsf{ep}_{\mathsf{Parser}}\ a = \mathsf{ep}_{\rightarrow}\ (\mathsf{ep}_{\mathsf{List}}\ \mathsf{ep}_{\mathrm{Char}})\ (\mathsf{ep}_{\mathsf{List}}\ (\mathsf{ep}_{(,)}\ a\ (\mathsf{ep}_{\mathsf{List}}\ \mathsf{ep}_{\mathrm{Char}})))$$

After a few steps this function will lead to a self application of $\mathsf{ep}_{\mathsf{List}}$, which obstructs further fusion. This problem can be avoided by choosing different types for different purposes. For instance, the parser's type can be changed into

$$\textbf{type}\ \ \mathsf{Parser'}\ a = (\mathsf{List}\ \mathrm{Char}) \to \mathsf{List'}\ (a, \mathsf{List}\ \mathrm{Char})$$

where List' is just another list

$$\textbf{data}\ \ \mathsf{List'}\ a = \mathsf{Nil'}\ |\ \mathsf{Cons'}\ a\ (\mathsf{List'}\ a)$$

Another useful trick to overcome this problem is to automatically replace closed EP terms like $(\mathsf{ep}_{\mathsf{List}}\ \mathsf{ep}_{\mathrm{Char}})$ with the identity $\mathsf{epid} = \mathsf{EP}\ \mathsf{id}\ \mathsf{id}$. This is possible because mapping for types of kind $\star$ is identity. Then the instance becomes

$$\mathsf{ep}_{\mathsf{Parser}}\ a = \mathsf{ep}_{\to}\ \mathsf{epid}\ (\mathsf{ep}_{\mathsf{List}}\ (\mathsf{ep}_{(,)}\ a\ \mathsf{epid}))$$

5.9.3 Fusing generic instances

So far we have shown that the adaptor is free from EPs. Adaptor is the only part of a generated instance that contains EPs. Now our goal is to show that a generated instance is free from sums, products and units. A generated instance can be written in the following form

$$g_T \vec{f} = \mathsf{adapt}_{\langle G,T\rangle}\ (g_{\Sigma\Pi}\ \ldots (g_{\mathcal{T}}\ \vec{f})\ \ldots)$$

where $g_{\Sigma\Pi}$ is a combination of the base cases and $g_{\mathcal{T}}$-s are free from the base cases and the base types. For instance, consider mapping for the rose trees.

$$\mathsf{map}_{Rose}\ f = \mathsf{adapt}_{\langle Map,Rose\rangle}\ (\mathsf{map}_{\times}\ f\ (\mathsf{map}_{List}\ (\mathsf{map}_{Rose}\ f)))$$

Here $g_{\Sigma\Pi}$ is $\mathsf{map}_{\times}$ and $g_{\mathcal{T}}$-s are f and $\mathsf{map}_{List}\ (\mathsf{map}_{Rose}\ f))$. Sums, products and units appear only in the adaptor and in the $g_{\Sigma\Pi}$ part. They do not appear in the $g_{\mathcal{T}}$ part. The type of the expression $\mathsf{adapt}_{\langle G,T\rangle}\ (g_{\Sigma\Pi}\ \vec{x})$ does not contain the base types. For instance,

$$\begin{aligned}&\lambda x.\lambda y.\mathsf{adapt}_{\langle Map,Rose\rangle}(\mathsf{map}_{\times}\ x\ y)\\ &\quad :: (a \to b) \to (\mathsf{List}\ (\mathsf{Rose}\ a) \to \mathsf{List}\ (\mathsf{Rose}\ b)) \to (\mathsf{Rose}\ a \to \mathsf{Rose}\ b)\end{aligned}$$

Therefore, it is enough to show that under some conditions fusion of the adaptor with the $g_{\Sigma\Pi}$ part will lead to elimination of the basic constructors, i.e. to $\{+, \times, \mathbb{1}\}$-*FF*. The idea is that the functions used in the base cases should not prevent the basic constructors coming close to the corresponding destructors. Consider, for example, the base case for products of the monadic mapping for lists (section **??**).

$$\begin{array}{lcl}\mathsf{mapl}_\times\ l\ r\ p & = & \mathsf{case}\ p\ \mathsf{of} \\ & & \mathsf{Pair}\ x\ y \rightarrow l\ x \gg\!= \lambda x'.r\ y \gg\!= \lambda y'.\mathsf{return}\ (\mathsf{Pair}\ x'\ y')\end{array}$$

The Pair produced in this instance is consumed in the adaptor. Fusion brings the constructor and the destructor close together, so that they are eliminated. The monadic operations (for lists) that surround the pair constructor do not constitute a problem, because they are "sufficiently consuming and producing". So far, we have not found a way to precisely state what "sufficiently consuming and producing" is. However, all the examples we tried had the base cases amenable for optimization. In practice the restriction is not severe.

Accumulation in the base cases can prevent elimination of the basic constructors. The base cases are essentially accumulating only when the generic function's type refers to a nested type, such as Nest above. However, we have already excluded nested type constructor from generic types in the previous subsection.

5.10 Performance Evaluation

We have implemented the improved fusion algorithm as a source-to-source translator for the core language presented in section 5.3[1]. The input language has been extended with syntactical constructs for specifying generic functions. Apart from the usual checks for statical semantics, the translator is also able to infer types. We used the Clean compiler [PvE01] to evaluate the performance of the optimized and unoptimized code.

Of course, we have investigated many example programs, but in this section we will only present the result of two examples that are realistic and/or illustrative: simple mapping with the type Map (section 5.2) and non-deterministic parser with the type Parser' (section 5.9).

The generic map function was used to apply the increment function to a list of $2.7\ 10^8$ integers. We have computed the overhead due to the creation of the list and the evaluation of the applied function and subtracted

[1]for people who want to experiment with the presented optimization technique, the sources of this prototype compiler are available and can be obtained by sending an email to one of the authors.

this from the measured execution times. The language used for the non-deterministic parser was extremely ambiguous leading to more than 200.000 different parses for an input consisting of a list of only 12 characters separated by spaces.

program	*unoptimized (sec)*	*optimized (sec)*	*speedup (times)*
map	66.78	8.42	7.9
parser	45.65	0.51	89.5

The parser example shows a gain in efficiency by a factor of 90. Not mentioned in table is the fact that the optimized version also uses considerably less memory: we had to increase the heap size of the unoptimized version to 128 MB, whereas the optimized version could easily run within in a few MB. The execution time of 45.65 sec can be split up into real execution time (12.0 sec) and garbage collection time (33.65 sec). These figures might appear too optimistic, but other experiments with a complete XML-parser defined generically confirm that these results are certainly not exaggerated.

5.11 Related work

The present work is based on the earlier work [AS04c] that used partial evaluation to optimize generic programs. To avoid non-termination we used fix-point abstraction of recursion in generic instances. This algorithm was, therefore, specifically tailored for optimization of generic programs. The algorithm presented here has also been designed with optimization of generic programs in mind. However it is a general-purpose algorithm that can improve other programs. The present algorithm completely removes generic overhead from a considerably larger class of generic programs than [AS04c].

The present optimization algorithm is an improvement of fusion algorithm [AGS03], which is in turn based on Chin's fusion [Chi94] and Wadler's deforestation [Wad88]. We have improved both consumer and producer analyses to be more semantically than syntactically based.

Chin and Khoo [CK96] improve the consumer analysis using the *depth* of a variable in a term. In their algorithm, *depth* is only defined for constructor terms, i.e. terms that are only built from variables and constructor applications. This approach is limited to first order functions. Moreover, the functions must be represented in a special *constructor-based* form. In contrast, our *depth* is defined for arbitrary terms of our language. Our algorithm does not require functions in to be represented in a special form, and it can handle higher order functions.

The present paper uses a generic scheme based on type-indexed values [Hin00a]. However, we believe that our algorithm will also improve code generated by other generic schemes, e.g PolyP [JJ97].

5.12 Conclusions and Future Work

In this paper we have presented an improved fusion algorithm, in which both producer and consumer analyses have been refined. We have shown how this algorithm *completely* eliminates generic overhead for a large class of programs. This class is described; it covers many practical examples. Presented performance figures show that the optimization leads to a huge improvement in both speed and memory usage.

In this paper we have ignored the aspect of data sharing. Generic specialization does not generate code that involves sharing, although sharing can occur in the base cases provided by the programmer. A general purpose optimization algorithm should take sharing into account to avoid duplication of work and code bloat. In the future we would like to extend the algorithm to take care of sharing. We believe that it will not affect the results for optimization of generic programs.

Additionally, we want to investigate other applications of this algorithm than generic programs. For instance, many programs are written in a combinatorial style using monadic or arrow combinators. Such combinators normally store functions in simple data types, i.e. wrap functions. To actually apply a function they need to unwrap it. It is worth looking at elimination of the overhead of wrapping-unwrapping.

Chapter 6

Gast: Generic Automated Software Testing

Software testing is a labor-intensive, and hence expensive, yet heavily used technique to control quality. In this paper we introduce Gast, a fully automatic test tool. Properties about functions and datatypes can be expressed in first order logic. Gast automatically and systematically generates appropriate test data, evaluates the properties for these values, and analyzes the test results. This makes it easier and cheaper to test software components. The distinguishing property of our system is that the test data are generated in a systematic and generic way using generic programming techniques. This implies that there is no need for the user to indicate how data should be generated. Moreover, duplicated tests are avoided, and for finite domains Gast is able to prove a property by testing it for all possible values. As an important side-effect, it also encourages stating formal properties of the software.

6.1 Introduction

Testing is an important and heavily used technique to measure and ensure software quality. It is part of almost any software project. The testing phase of typical projects takes up to 50% of the total project effort, and hence contributes significantly to the project costs. Any change in the software can potentially influence the result of a test. For this reason tests have to be repeated often. This is error-prone, boring, time consuming, and expensive.

In this paper we introduce a tool for automatic software testing. Automatic testing significantly reduces the effort of individual tests. This implies that performing the same test becomes cheaper, or one can do more tests within the same budget. In this paper we restrict ourselves to *functional testing*, i.e. examination whether the software obeys the given specification.

In this context we distinguish four steps in the process of functional testing: 1) *formulation of a property* to be obeyed: what has to be tested; 2) *generation of test data*: the decision for which input values the property should be examined, 3) *test execution*: running the program with the generated test data, and 4) *test result analysis*: making a verdict based on the results of the test execution.

The introduced Generic Automatic Software Test system, Gast, performs the last three steps fully automatically. Gast generates test data based on the types used in the properties, it executes the test for the generated test values, and gives an analysis of these test results. The system either produces a message that the property is proven, or the property has successfully passed the specified number of tests, or Gast shows a counterexample.

Gast makes testing easier and cheaper. As an important side-effect it encourages the writing of properties that should hold. This contribute to the documentation of the system. Moreover, there is empirical evidence that writing specifications on its own contributes to the quality of the system [TWC01].

Gast is implemented in the functional programming language Clean [PE02]. The primary goal is to test software written in Clean. However, it is not restricted to software written in Clean. Functions written in other languages can be called through the foreign function interface, or programs can be invoked.

The properties to be tested are expressed as functions in Clean, they have the power of first order predicate logic. The specifications can state properties about individual functions and datatypes as well as larger pieces of software, or even about complete programs. The definition of properties and their semantics are introduced in Section 3.

Existing automatic test systems, such as QuickCheck [CH00, CH02b], use random generation of test data. When the test involves user-defined datatypes, the tester has to indicate how elements of that type should be generated. Our test system, Gast, improves both points. Using systematic generation of test data, duplicated tests involving user-defined types do not occur. This makes even proofs possible. By using a generic generator the tester does not have to define how elements of a user-defined type have to be generated. Although Gast has many similarities with QuickCheck, it differs in the language to specify properties (possibilities and semantics), the generation of test data and execution of tests (by using generics), and the analysis of test results (proofs). Hence, we present GAST as a self–contained tool. We will point out similarities and differences between the tools whenever appropriate.

Generic programming deals with the universal representation of a type instead of concrete types. This is explained in Section 6.2. Automatic data generation is treated in Section 6.4. If the tester wants to control the generation of data explicitly, he is able to do so (Section 6.7).

After these preparations, the test execution is straightforward. The property is tested for the generated test data. GAST uses the code generated by the CLEAN compiler to compute the result of applying a property to test data. This has two important advantages. First, there cannot exist semantic differences between the ordinary CLEAN code and the interpretation of properties. Secondly, it keeps GAST simple. In this way we are able to construct a light-weight test system. This is treated in Section 6.5. Next, test result analysis is illustrated by some examples. In Section 6.7 we introduce some additional tools to improve the test result analysis. Finally, we discuss related work and open issues and we conclude.

6.2 Generic Programming

Generic programming [HP01, Hin00c, AP02, CHJ$^+$01] is based on a universal tree representation of datatypes. Whenever required, elements of any datatype can be transformed to and from that universal tree representation. The generic algorithm is defined on this tree representation. By applying the appropriate transformations, this generic algorithm can be applied to any type.

Generic programming is essential for the implementation of GAST. However, users do not have to know anything about generic programming. The reader who wants to get an impression of GAST might skip this Section on first reading.

Generic extensions are currently developed for Haskell [JH02] and CLEAN [PE02]. In this paper we will use CLEAN without any loss of generality.

6.2.1 Generic Types

The universal type is constructed using the following type definitions [AP02].[1]

```
:: UNIT          = UNIT                  // leaf of the type tree
```

[1]CLEAN uses additional constructs for information on constructors and record fields.

```
:: PAIR   a b      = PAIR a b              // branch in the tree
:: EITHER a b      = LEFT a | RIGHT b      // choice between a and b
```

As an example, we give two algebraic datatypes, Color and List, and their generic representation, Color° and List°. The symbol :== in the generic version of the definition indicates that it are just type synonyms, they do not define new types.

```
:: Color    = Red | Yellow | Blue // ordinary algebraic type definition
:: Color°   :== EITHER (EITHER UNIT UNIT) UNIT // generic representation

:: List a   = Nil | Cons a (List a)
:: List°    :== EITHER UNIT (PAIR a (List a))
```

The transformation from the user-defined type to its generic counterpart is done by automatically generated functions like[2]:

```
ColorToGeneric :: Color → EITHER (EITHER UNIT UNIT) UNIT
ColorToGeneric Red    = LEFT (LEFT UNIT)
ColorToGeneric Yellow = LEFT (RIGHT UNIT)
ColorToGeneric Blue   = RIGHT UNIT

ListToGeneric :: (List a) → EITHER UNIT (PAIR a (List a))
ListToGeneric Nil           = LEFT UNIT
ListToGeneric (Cons x xs) = RIGHT (PAIR x xs)
```

The generic system automatically generates these functions and their inverses.

6.2.2 Generic Functions

Based on this representation of types one can define generic functions. As example we will show the generic definition of equality[3].

```
generic gEq a  :: a a → Bool
gEq{|UNIT|}          _          _           = True
gEq{|PAIR|}   fa fx  (PAIR a x) (PAIR b y)  = fa a b && fx x y
gEq{|EITHER|} fl fr  (LEFT x)   (LEFT y)    = fl x y
gEq{|EITHER|} fl fr  (RIGHT x)  (RIGHT y)   = fr x y
gEq{|EITHER|} _  _   _          _           = False
gEq{|Int|}           x          y           = x == y
```

[2]We use the direct generic representation of result types instead of the type synonyms Color° and List° since it shows the structure of result more clearly.

[3]We only consider the basic type Int here. Other basic types are handled similarly.

The generic system provides additional arguments to the instances for PAIR and EITHER to compare instances of the type arguments (a and b in the definition).

In order to use this equality for Color an instance of gEq for Color must be derived by: **derive** gEq Color. The system generates code equivalent to

```
gEq{|Color|} x y = gEq{|EITHER|} (gEq{|EITHER|} gEq{|UNIT|} gEq{|UNIT|})
                   gEq{|UNIT|} (ColorToGeneric x) (ColorToGeneric y)
```

The additional arguments needed by gEq{|EITHER|} in gEq{|Color|} are determined by the generic representation of the type Color: Color°.

If this version of equality is not what you want, you can always define your own instance of gEq for Color, instead of deriving the default.

The infix version of this generic equality is defined as:

```
(===) infix 4 :: !a !a → Bool | gEq{|∗|} a
(===) x y = gEq{|∗|} x y
```

The addition | C a to a type is a class restriction: the type a should be in class C. Here it implies that the operator === can only be applied to type a, if there exists a defined or derived instance of gEq for a.

This enables us to write expressions like Red === Blue. The necessary type conversions form Color to Color° need not be specified, they are generated and applied at the appropriate places by the generic system.

It is important to note that the user of types like Color and List need not be aware of the generic representation of types. Types can be used and introduced as normally; the static type system also checks the consistent use of types as usual.

6.3 Specification of Properties

The first step in the testing process is the formulation of properties in a formalism that can be handled by GAST. In order to handle properties from first order predicate logic in GAST we represent them as functions in CLEAN. These functions can be used to specify properties of single functions or operations in CLEAN, as well as properties of large combinations of functions, or even of entire programs.

Each property is expressed by a function yielding a Boolean value. An expression with value True indicates a successful test, False indicates a counter example. This solves the famous *oracle problem*: how do we decide whether the result of a test is correct.

The arguments of such a property represent the universal variables of the logical expression. Properties can have any number of arguments, each of these arguments can be of any type.

In this paper we will only consider well-defined and finite values as test data. Due to this restriction we are able to use the *and*-operator (**&&**) and *or*-operator (||) of Clean to represent the logical operators *and* ($\wedge$) and *or* ($\vee$) respectively.

Our first example involves the implementation of the logical or-function using only a two-input nand-function as basic building element.

```
or :: Bool Bool → Bool
or x y = nand (not x) (not y) where not x = nand x x
```

The desired property is that the value of this function is always equal to the value of the ordinary or-operator, ||, from Clean. That is, the ||-operator serves as specification for the new implementation, or. In logic, this is:

$$\forall x \in Bool\,.\,\forall y \in Bool\,.\,x||y = or\,x\,y$$

This property can be represented by the following function in Clean. By convention we will prefix property names by prop.

```
propOr :: Bool Bool → Bool
propOr x y = x||y == or x y
```

The user invokes the testing of this property by the main function:

```
Start = test propOr
```

Gast yields `Proof: success for all arguments after 4 tests` for this property. Since there are only finite types involved the property can be proven by testing.

For our second example we consider the classical implementation of stacks:

```
:: Stack a :== [a]

pop :: (Stack a) → Stack a
pop [_:r] = r

top :: (Stack a) → a
top [a:_] = a

push :: a (Stack a) → Stack a
push a s = [a:s]
```

A desirable property for stacks is that after pushing some element onto the stack, that element is on top of the stack. Popping an element just pushed on the stack yields the original stack. The combination of these properties is expressed as:

```
propStack :: a (Stack a) → Bool | gEq{|*|} a
propStack e s = top (push e s) === e && pop (push e s) === s
```

This property should hold for any type of stack-element. Hence we used polymorphic functions and the generic equality, ===, here. However, GAST can only generate test data for some concrete type. Hence, we have to specify which type GAST should use for the type argument a. For instance by:

```
propStackInt :: (Int (Stack Int) → Bool)
propStackInt = propStack
```

In contrast to properties that use overloaded types, it actually does not matter much which concrete type we choose. A polymorphic property will hold for elements of any type if it holds for elements of type Int. The test is executed by Start = test propStackInt. GAST yields: Passed after 1000 tests. This property involves the very large type integer and the infinite type stack, so only testing for a finite number of cases, here 1000, is possible.

In propOr we used a reference implementation (||) to state a property about a function (or). In propStack the desired property is expressed directly as a relation between functions on a datatype. Other kind of properties state relations between the input and output of functions, or use model checking based properties. For instance, we have tested a system for safe communication over unreliable channels by an alternating bit protocol with the requirement that the sequence of received messages should be equal to the input sequence of messages.

The implication operator, $\Rightarrow$, is often added to predicate logic. For instance $\forall x.x \geq 0 \Rightarrow (\sqrt{x}\,)^2 = x$. We can use the law $p \Rightarrow q = \neg p \vee q$ to implement it:

```
(===>) infix 1 :: Bool Bool → Bool
(===>) p q = ~p || q
```

In Section 6.7.1 we will return to the semantics and implementation of $p \Rightarrow q$

In first order predicate logic one also has the existential quantifier, $\exists$. If this is used to introduce values in a constructive way it can be directly transformed to local definitions in a functional programming language, for instance as: $\forall x.x \geq 0 \Rightarrow \exists y.y = \sqrt{x} \wedge y^2 = x$ can directly be expressed using local definitions.

```
propSqrt :: Real → Bool
propSqrt x = x ≥0 ===> let y = sqrt x in y*y == x
```

In general it is not possible to construct an existentially quantified value. For instance, for a type *Day* and a function *tomorrow* we require that each day can be reached: $\forall day. \exists d.\, tomorrow\ d = day$. In Gast this is expressed as:

```
propSurjection :: Day → Property
propSurjection day = Exists λd = tomorrow d === day
```

The success of the Exists operator depends on the types used. The property propSurjection will be proven by Gast. Also for recursive types it will typically generate many successful test cases, due to the systematic generation of data. However, for infinite types it is impossible to determine that there does not exists an appropriate value (although only completely undefined tests are a strong indication of an error).

The only task of the tester is to write properties, like propOr, and to invoke the testing by Start = test propOr. Based on the type of arguments needed by the property, the test system will generate test data, execute the test for these values, and analyze the results of the tests. In the following three sections we will explain how Gast works. The tester does not have to know this.

6.3.1 Semantics of Properties

For Gast we extend the standard operational semantics of Clean. The standard reduction to weak head normal form is denoted as $whnf\,[\![\ e\ ]\!]$. The additional rules are applied after this ordinary reduction. The implementation will follow these semantics rules directly. The possible results of the evaluation of a property are the values **Suc** for success, and **CE** for counterexample. In these rules $\lambda\, x.p$ represents any function (i.e. a partially parameterized function, or a lambda-expression). The evaluation of a property, $Eval\,[\![\ p\ ]\!]$, yields a list of results:

$$\begin{aligned}
Eval\,[\![\ \lambda\, x.p\ ]\!] &= [r | v \leftarrow genAll; r \leftarrow Eval\,[\![\ (\lambda\, x.p)\, v\ ]\!]\,] && (6.1)\\
Eval\,[\![\ \text{True}\ ]\!] &= [\,\mathbf{Suc}\,] && (6.2)\\
Eval\,[\![\ \text{False}\ ]\!] &= [\,\mathbf{CE}\,] && (6.3)\\
Eval\,[\![\ e\ ]\!] &= Eval\,[\![\ whnf\,[\![\ e\ ]\!]\ ]\!] && (6.4)
\end{aligned}$$

To test property p we evaluate $Test\,[\![\ p\ ]\!]$. The rule $An\,[\![\ l\ ]\!]\,n$ analysis a list of test results In rule 6.5, N is the maximum number of tests. There are three possible test results: **Proof** indicates that the property holds for all well-defined values of the argument types, **Passed** indicated that the property passed N tests without finding a counterexample, **Fail** indicates that a counterexample is found.

$$\begin{aligned}
&\mathit{Test}\,[\![\, p \,]\!] &&= \mathit{An}\,[\![\, \mathit{Eval}\,[\![\, \mathit{whnf}\,[\![\, p \,]\!]\;]\!]\;]\!]\,N && (6.5)\\
&\mathit{An}\,[\![\, [\,] \,]\!]\,n &&= \mathbf{Proof} && (6.6)\\
&\mathit{An}\,[\![\, l \,]\!]\,0 &&= \mathbf{Passed} && (6.7)\\
&\mathit{An}\,[\![\, [\,\mathbf{CE} : \mathit{rest}\,] \,]\!]\,n &&= \mathbf{Fail} && (6.8)\\
&\mathit{An}\,[\![\, [\,r : \mathit{rest}\,] \,]\!]\,n &&= \mathit{An}\,[\![\, \mathit{rest} \,]\!]\,(n-1), \mathrm{if}\, r \neq \mathbf{CE} && (6.9)
\end{aligned}$$

The most important properties of this semantics are:

$$\begin{aligned}
&\mathit{Test}\,[\![\, \lambda x.p \,]\!] = \mathbf{Proof} &&\Rightarrow \forall v.(\lambda x.p)v && (6.10)\\
&\mathit{Test}\,[\![\, \lambda x.p \,]\!] = \mathbf{Fail} &&\Rightarrow \exists v.\neg(\lambda x.p)v && (6.11)\\
&\mathit{Test}\,[\![\, p \,]\!] = \mathbf{Passed} &&\Leftrightarrow \forall r \in (\mathit{take}\,N\,\mathit{Eval}\,[\![\, p \,]\!]).r \neq \mathbf{CE} && (6.12)
\end{aligned}$$

Property 6.10 state that GAST only produces **Proof** if the property is universal valid. According to 6.11, the system yields only **Fail** if a counter example exists. Finally, the systems yields **Passed** if the first N tests do not contain a counterexample. These properties can be proven by induction and case distinction from the rules 1 to 6.9 given above. Below we will introduce some additional rules for $\mathit{Eval}\,[\![\, p \,]\!]$, in such a way that these properties are preserved.

The semantics of the Exists-operator is:

$$\mathit{Eval}\,[\![\, \mathbf{Exists}\;\lambda\, x.p \,]\!] = \mathit{One}\,[\![\, [r|v \leftarrow \mathit{genAll}; r \leftarrow \mathit{Eval}\,[\![\, (\lambda\, x.p)\, v \,]\!]\,]\,]\!]\,M \qquad (6.13)$$

where

$$\begin{aligned}
&\mathit{One}\,[\![\, [\,] \,]\!]\,m &&= [\,\mathbf{CE}\,] && (6.14)\\
&\mathit{One}\,[\![\, l \,]\!]\,0 &&= [\,\mathsf{undef}\,] && (6.15)\\
&\mathit{One}\,[\![\, [\,\mathbf{Suc} : \mathit{rest}\,] \,]\!]\,m &&= [\,\mathbf{Suc}\,] && (6.16)\\
&\mathit{One}\,[\![\, [\,r : \mathit{rest}\,] \,]\!]\,m &&= \mathit{One}\,[\![\, \mathit{rest} \,]\!]\,(m-1), \mathrm{if}\, r \neq \mathbf{Suc} && (6.17)
\end{aligned}$$

The rule $\mathit{One}\,[\![\, l \,]\!]$ scans a list of semantic results, it yields success if the list of results contains at least one success within the first M results. As soon as one or more results are rejected the property cannot be proven anymore. It can, however, still successfully test the property for N values. To ensure termination also the number of rejected test is limited by an additional counter. These changes for $\mathit{An}\,[\![\, l \,]\!]$ are implemented by analyse in Section 6.6.

6.4 Generating Test Data

To test a property, step 2) in the test process, we need a list of values of the argument type. GAST will evaluate the property for the values in this list.

Since we are testing in the context of a referentially transparent language, we are only dealing with pure functions: the result of a function is completely determined by its arguments. This implies that repeating the test for the same arguments is useless: referential transparency guarantees that the results will be identical. GAST should prevent the generation of duplicated test data.

For finite types like Bool or non-recursive algebraic datatypes we can generate all elements of the type as test data. For basic types like Real and Int, generating all elements is not feasible. There are far too many elements, e.g. there are 2^{32} integers on a typical computer. For these types, we want GAST to generate some common border values, like 0 and 1, as well as random values of the type. Here, preventing duplicates is usually more work (large administration) than repeating the test. Hence, we do not require that GAST prevents duplicates here.

For recursive types, like list, there are infinitely many instances. GAST is only able to test properties involving these types for a finite number of these values. Recursive types are usually handled by recursive functions. Such a function typically contains special cases for small elements of the type, and recursive cases to handle other elements. In order to test these functions we need values for the special cases as well as some values for the general cases. We achieve this by generating a list of values of increasing size. Preventing duplicates is important here as well.

The standard implementation technique in functional languages would probably make use of classes to generate, compare and print elements of each datatype involved in the tests [CH00]. Instances for standard datatypes can be provided by a test system. User-defined types however, would require user-defined instances for all types, for all classes. Defining such instances is error prone, time consuming and boring. Hence, a class based approach would hinder the ease of use of the test system. Special about GAST is that we use generic programming techniques such that one general solution can be provided once and for all.

To generate test data, GAST builds a list of generic representations of the desired type. The generic system transforms these generic values to the type needed. Obviously, not any generic tree can be transformed to instances of a given type. For the type Color only the trees LEFT (LEFT UNIT), LEFT (RIGHT UNIT), and RIGHT UNIT represent valid values. The additional type–dependent argument inserted by the generic system (see the gEq example shown above) provides exactly the necessary information to guide the generation of values.

To prevent duplicates we record the tree representation of the generated values in the datatype Trace.

```
:: Trace = Unit | Pair [(Trace,Trace)] [(Trace,Trace)]
         | Either Bool Trace Trace | Int [Int] | Done | Empty
```

A single type Trace is used to record visited parts of the generic tree (rather than the actual values or their generic representation), to avoid type incompatibilities.

The type Trace looks quite different from the ordinary generic tree since we record *all* generated values in a single tree. An ordinary generic tree just represents *one* single value.

New parts of the trace are constructed by the generic function generate. The function nextTrace prepares the trace for the generation of the next element from the list of test data.

The function genAll uses generate to produce the list of all values of the desired type. It generates values until the next trace indicates that we are done.

```
genAll :: RandomStream → [a] | generate{|*|} a
genAll rnd = g Empty rnd
where g Done rnd = []
      g t    rnd = let (x, t2, rnd2) = generate{|*|} t rnd
                       (t3, rnd3)    = nextTrace t2 rnd2
                   in [x: g t3 rnd3]
```

For recursive types, the generic tree can grow infinitely. Without detailed knowledge about the type, one cannot determine where infinite branches occur. This implies that any systematic depth-first strategy to traverse the tree of possible values can fail to terminate. Moreover, small values appear close to the root of the generic tree, and have to be generated first. Any depth–first traversal will encounter these values too late. A left-to-right strategy (breath–first) will favor values in the left branches and vice versa. Such a bias in any direction is undesirable.

In order to meet all these requirements, nextTrace uses a random choice at each Either in the tree. The RandomStream, a list of pseudo random values, is used to choose. If the chosen branch appears to be exhausted, the other branch is explored. If both branches cannot be extended, all values in this subtree are generated and the result is Done. The generic representation of a type is a balanced tree, this guarantees an equal distribution of the constructors if multiple instances of the type occur (e.g. [Color] can contain many colors).

The use of the Tree prevents duplicates, and the random choice prevents a left–to–right bias. Since small values are represented by small trees the will occur very likely soon in the list of generated values.

An element of the desired type is produced by genElem using the random stream. Left and Right are just sensible names for the Boolean values.

```
nextTrace (Either _ tl tr) rnd
   = let (b, rnd2) = genElem rnd in
     if b (let (tl‘ , rnd3) = nextTrace tl rnd2 in
             case tl‘ of
               Done    = let (tr‘ , rnd4) = nextTrace tr rnd3 in
                           case tr‘ of
                             Done = (Done, rnd4)
                             _    = (Either Right tl tr‘ , rnd4)
               _       = (Either Left tl‘ tr , rnd3))
           (let (tr‘ , rnd3) = nextTrace tr rnd2 in
             case tr‘ of
               Done    = let (tl‘ , rnd4) = nextTrace tl rnd3 in
                           case tl‘ of
                             Done = (Done, rnd4)
                             _    = (Either Left tl‘ tr , rnd4)
               _       = (Either Right tl tr‘ , rnd3))
```

The corresponding instance of generate follows the direction indicated in the trace. When the trace is empty, it takes a boolean from the random stream and creates the desired value as well as the initial extension of the trace.

```
generic generate a :: Trace RandomStream → (a, Trace, RandomStream)
generate{|EITHER|} fl fr Empty rnd
  = let (f,rnd2) = genElem rnd in
    if f (let (l,tl,rnd3) = fl Empty rnd2
          in (LEFT l, Either Left tl Empty, rnd3))
         (let (r,tr,rnd3) = fr Empty rnd2
          in (RIGHT r, Either Right Empty tr, rnd3))
generate{|EITHER|} fl fr (Either left tl tr) rnd
  | left = let (l,tl2,rnd2) = fl tl rnd
           in (LEFT l, Either left tl2 tr, rnd2)
         = let (r,tr2,rnd2) = fr tr rnd
           in (RIGHT r, Either left tl tr2, rnd2)
```

For Pair the function nextTrace uses a breath–first traversal of the tree implemented by a queue. Two lists of tuples are used to implement an efficient queue. The tuple containing the current left branch the next right branch, as well as the tuple containing the next left branch and an empty right branch are queued.

6.4.1 Generic generation of Functions as Test Data

Since CLEAN is a higher order language it is perfectly legal to use a function as an argument or result of a function. Also in properties, the use of higher order functions can be very useful. A well-known property of the function map is:

```
propMap :: (a→b) (b→c) [a] → Bool | gEq{|∗|} c
propMap f g xs = map g (map f xs) === map (g o f) xs
```

In order to test such a property we must choose a concrete type for the polymorphic arguments. Choosing Int for all type variables yields:

```
propMapInt :: ((Int→Int) (Int→Int) [Int] → Bool)
propMapInt = propMap
```

This leaves us with the problem of generating functions automatically. Functions are not datatypes and hence cannot be generated by the default generic generator. Fortunately, the generic framework provides a way to create functions. We generate functions of type a→b by an instance for generate{|(→)|}. First, a list of values of type b is generated. The argument of type a is transformed in a generic way to an index in this list. For instance, a function of type Int → Color could look like λa = [Red,Yellow,Blue] !! (abs a % 3). Like all test data, genAll generates a list of these functions. Currently GAST does not keep track of generated functions in order to prevent duplicates, or to stop after generating all possible functions. Due to space limitations we omit details.

6.5 Test Execution

Step 3) in the test process is the test execution. The implementation of an individual test is a direct translation of the given semantic rules introduced above. The type class Testable contains the function evaluate which directly implements the rules for $Eval\,[\![\, p \,]\!]$.

```
class Testable a
where
    evaluate :: a RandomStream Admin → [Admin]
```

In order to be able to show the arguments used in a specific test, we administrate the arguments represented as strings as well as the result of the test in a record called Admin. There are three possible results of a test: undefined (UnDef), success (Suc), and counter example found (CE).

```
:: Admin = {res::Result, args :: [String]}
:: Result = UnDef | Suc | CE
```

Instances of TestArg can be argument of a property. The system should be able to generate elements of such a type (generate) and to transform them to string (genShow) in order to add them to the administration.

```
class TestArg a | genShow{|*|}, generate{|*|} a
```

The semantic equations 6.2 and 6.3 are implemented by the instance of evaluate for the type Bool. The fields res and arg in the record ad (for administration) are updated.

```
instance Testable Bool
where
    evaluate b rs ad = [{ad & res=if b Suc CE, args=reverse ad.args}]
```

The rule for function application, semantic equation 6.1, is complicated slightly by administrating function arguments.

```
instance Testable (a→b) | Testable b & TestArg a
where
    evaluate f rs admin
        = let (rs, rs2) = split rs in forAll f (gen rs) rs2 admin

forAll f list rs ad
    = diagonal [ evaluate (f a) (genRandInt s) {ad&args=[show a:ad.args]}
               \\ a←list & s←rs]
```

The function diagonal takes care of a *fair* order of tests. For a 2-argument function f, the system generates two sequences of arguments, call them $[a, b, c, ..]$ and $[u, v, w, ..]$ respectively. The order of tests is $f\,a\,u, f\,a\,v, f\,b\,u, f\,a\,w, f\,b\,v, f\,c\,u, ..$ rather than $f\,a\,u, f\,a\,v, f\,a\,w, .., f\,b\,u, f\,b\,v, f\,b\,w, ...$

6.6 Test Result Evaluation

The final step, step 4), in the test process is the evaluation of results. The system just scans the generated list of test results as indicated by $An\,[\![\,l\,]\!]$. The only extension is the showing of the number and arguments of the current test before the test result is evaluated. In this way the tester of Gast is able to identify the data causing a runtime error or taking a lot of time. A somewhat simplified version of the function test is:

```
test  :: p → [String] | Testable p
test p = analyse (evaluate p RandomStream newAdmin) maxTests MaxArgs
where analyse ::  [Admin] Int Int → [String]
      analyse [] n m = ["λnProof: success for all arguments"]
      analyse l  0 m = ["λnPassed ",toString maxTests," tests"]
      analyse l  n 0 = ["λnPassed ",toString maxArgs," arguments"]
      analyse [res: rest] n m
        = [blank, toString (maxTests−n+1),":":showArgs res.args
           case res.res of
             UnDef  = analyse rest n (m−1)
             Suc    = analyse rest (n−1) (m−1)
             CE     = ["λnCounterexample: ": showArgs res.args []]]
```

6.7 Additional Features

In order to improve the power of the test tool, we introduce some additional features. These possibilities are realized by combinators (functions) that manipulate the administration. We consider the following groups of combinators: 1) an improved implication, $p \Rightarrow q$, that discards the test if p does not hold; 2) combinators to collect information about the actual test data used; 3) combinators to apply user-defined test data instead of generated test data.

QuickCheck provides a similar implication combinator. Our collection of test data relies on generic programming rather than a build–in show function. QuickCheck does not provide a similar generation of user–defined test data.

6.7.1 Implication

Although the implication operator ===> works correctly, it has an operational drawback: if p does not hold, the property $p \Rightarrow q$ holds and is counted as a successful test. This operator is often used to put a restriction on arguments to be considered, as in $\forall x. x \geq 0 \Rightarrow (\sqrt{x})^2 = x$. Here we only want to consider tests where $x \geq 0$ holds, in other situations the test should not be taken into account. This is represented by the result *undefined*. We introduce the operator ==> for this purpose.

$$Eval\,[\![\, \text{True} ==> p \,]\!] \quad = \quad Eval\,[\![\, p \,]\!] \tag{6.18}$$
$$Eval\,[\![\, \text{False} ==> p \,]\!] \quad = \quad [\,\mathbf{Rej}\,] \tag{6.19}$$

If the predicate holds the property p is evaluated, otherwise we explicitly yield an undefined result. The implementation is:

```
(==>) infixr 1 :: Bool p → Property | Testable p
(==>) c p
   | c = Prop (evaluate p)
       = Prop (λrs ad = [{ad & res = Undef}])
```

Since ==> needs to update the administration, the property on the right-hand side is a datatype holding an update–function instead of a Boolean.

```
:: Property = Prop (RandomStream Admin → [Admin])
instance Testable Property
where
   evaluate (Prop p) rs admin = p rs admin
```

The operator ==> can be used as ===> in propSqrt above. The result of executing test propSqrt is Counter−example found after 2 tests: 3.07787e−09. The failure is caused by the finite precision of reals.

6.7.2 Information about Test Data used

For properties like propStack, it is impossible to test all possible arguments. The tester might be curious to known more about the actual test data used in a test. In order to collect labels we extend the administration Admin with a field labels of type [String]. The system provides two combinators to store labels:

```
label     ::      l p → Property | Testable p & genShow{|*|} l
classify  :: Bool l p → Property | Testable p & genShow{|*|} l
```

The function label always adds the given label; classify only adds the label when the condition holds. The function analyse is extended to collect these strings, orders them alphabetically, counts them and computes the fraction of tests that contain this label. The label can be an expression of any type, it is converted to a string in a generic way (by genShow{|*|}).

These functions do not change the semantics of the specification, their only effect is the additional information in the report to the tester.

$$\begin{aligned} Eval[\![\, \textbf{label}\ l\ p \,]\!] &= Eval[\![\, p \,]\!] \\ &\quad \text{adds } l \text{ to the administration} \quad (6.20) \\ Eval[\![\, \textbf{classify}\ \text{True}\ l\ p \,]\!] &= Eval[\![\, \textbf{label}\ l\ p \,]\!] \quad (6.21) \\ Eval[\![\, \textbf{classify}\ \text{False}\ l\ p \,]\!] &= Eval[\![\, p \,]\!] \quad (6.22) \end{aligned}$$

We will illustrate the use of these functions. It is possible to view the exact test data used for testing the property of stacks by

```
propStackL :: Int (Stack Int) → Property
propStackL e s = label (e,s) (top (push e s)===e && pop (push e s)===s)
```

A possible result of testing propStackL for only 4 combinations of arguments is:

```
Passed 4 tests
 (0,[0,1]): 1 (25%)
 (0,[0]): 1 (25%)
 (0,[]): 1 (25%)
 (1,[]): 1 (25%)
```

The function classify can, for instance, be used to count the number of empty stacks occurring in the test data.

```
propStackC e s = classify (isEmpty s) s (propStack e s)
```

A typical result for 200 tests is:

```
Passed 200 tests
 []: 18 (9%)
```

6.7.3 User-defined Test Data

GAST generates sensible test data based on the type of the arguments. Sometimes the tester is not satisfied with this behavior. This occurs for instance if very few generated elements obey the condition of an implication, cause enormous calculations, or overflow.

The property propFib states that the value of the efficient version of the Fibonacci function, fibLin, should be equal to the value of the well-known naive definition, Fib, for non-negative arguments.

```
propFib n = n≥0 ==> fib n == fibLin n

fib 0 = 1
fib 1 = 1
fib n = fib (n−1) + fib (n−2)

fibLin n = f n 1 1
where
    f 0 a b = a
    f n a b = f (n−1) b (a+b)
```

One can prevent long computations and overflow by limiting the size of the argument by an implication. For instance:

```
propFib n = n≥0 && n≤15 ==> fib n == fibLin n
```

This is a rather unsatisfactory solution. The success rate of tests in the generated list of test values will be low, due to the condition many test results will be undefined (since the condition of the implication is false). In those situations it is more efficient if the user specifies the test values, instead of letting the Gast generate it. For this purpose the combinator For is defined. It can be used to test the equivalence of the Fibonacci functions for all arguments from 0 to 15:

```
propFibR = propFib For [0..15]
```

Testing yields `Proof: success for all arguments after 16 tests`.

The semantics of the For combinator is:

$$\mathit{Eval}\,[\![\ \lambda\, x.\, p\ \mathbf{For}\ \mathit{list}\ ]\!] \quad = \quad [r|v \leftarrow \mathit{list}; r \leftarrow \mathit{Eval}\,[\![\ (\lambda\, x.p)\, v\ ]\!]\,] \tag{6.23}$$

The implementation is very simple using the machinery developed above:

```
(For) infixl 0 :: (x→p) [x] → Property | Testable p & TestArg x
(For) p list = Prop (forAll p list )
```

Apart from replacing or combining the automatically generated test date by his own tests, the user can control the generation of data by adding an instance for his type to generate, or explicitly transform generated data-types (e.g. lists to balanced trees).

6.8 Related Work

Testing is labor-intensive, boring and error-prone. Moreover, it has to be done often by software engineers. Not surprisingly, a large number of tools has been developed to automate testing. Although some of these tools are well engineered, none of them gives automatic support like Gast does for *all* steps of the testing process. Only a few tools are able to generate test data for arbitrary types based on the types used in properties [BGM91].

In the functional programming world there are some related tools. The tool QuickCheck [CH00, CH02b] has similar ambitions as our tool. Distinguishing features of our tool are: the generic generation of test data for arbitrary types (instead of based on a user-defined instance of a class), and the systematic generation of test data (instead of random). As a consequence of the systematic generation of test data, our system is able to detect that all possible values are tested and hence the property is proven. Moreover, Gast offers a complete implementation of first order predicate logic.

Auburn [MR01] is a tool for automatic benchmarking of functional datatypes. It is also able to generate test data, but not in a systematic and generic way. Runtime errors and counterexamples of a stated property can be detected.

HUnit [hun] is the Haskell variant of JUnit [jun] for Java. JUnit defines how to structure your test cases and provides the tools to run them. It executes the test defined by the user. Tests are implemented in a subclass of TestCase.

An important area of automatic test generation is testing of reactive systems, or control-intensive systems. In these systems the interaction with the environment in terms of stimuli and responses is important. Typical examples are communication protocols, embedded systems, and control systems. Such systems are usually modelled and specified using some kind of automaton or state machine. There are two main approaches for automatic test generation from such specifications. The first is based on Finite State Machines (FSM), and uses the theory of checking experiments for Mealy-machines [LY96]. Several academic tools exist with which tests can be derived from FSM specifications, e.g., PHACT/THE CONFORMANCE KIT [FMMW98]. Although GAST is able to test the input/output relation of an FSM (see Section 6.3), checking the actual state transitions requires additional research.

The second approach is based on labelled transition systems and emanates from the theory of concurrency and testing equivalences [bri01]. Tools for this approach are, e.g., TGV [FJTJ96], TESTCOMPOSER [KJG99], TESTGEN [HT99], and TORX [BFV^{+}99, TB02]. State-based tools concentrate on the control flow, and cannot usually cope with complicated data structures. As shown above GAST is able to cope with these data structures.

6.9 Discussion

In this paper we introduce GAST, a generic tool to test software. The complete code, about 600 lines, can be downloaded from `http://www.cs.kun.nl/~pieter`. The tests are based on properties of the software, stated as functions based on first order predicate logic. Based on the types used in these properties the system automatically generates test data in a systematic way, checks the property for these generated values, and analyzes the results of these tests.

One can define various kind of properties. The functions used to describe properties are slightly more powerful than first order predicate logic (thanks to the combination of combinators and higher order functions) [vEdM]. In our system we are able to express properties known under names as black-box tests, algebraic properties, and model based, pre- and post-conditional. Using the ability to specify the test data, also user-guided white-box tests are possible.

Based on our experience we indicate four kinds of errors spotted by Gast. The system cannot distinguish these errors. The tester has to analyze them.

1. Errors in the implementation; the kind of mistakes you expect to find.

2. Errors in the specification; in this situation the tested software also does not obey the given property. Analysis of the indicated counter example shows that the specification is wrong instead of the software. Testing improves the confidence in the accuracy of the properties as well as the implementation.

3. Errors caused by the finite precision of the computer used; especially for properties involving reals, e.g. propSqrt, this is a frequent problem. In general we have to specify that the difference between the obtained answer and the required solution is smaller than some allowed error range.

4. Non-termination or run-time errors; although the system does not explicitly handle these errors, the tester notices that the error occurs. Since Gast lists the arguments before executing the test, the values causing the error are known. This appears to detect partially defined functions effectively.

The efficiency of Gast is mainly determined by the evaluation of the property, not by the generation of data. For instance, on a standard PC the system generates up to 100,000 integers or up to 2000 lists of integers per second. In our experience errors pop up rather soon, if they exist. Usually 100 to 1000 tests are sufficient to be reasonably sure about the validity of a property.

In contrast to proof-systems like Sparkle [dMvEP02], Gast is restricted to well-defined and finite arguments. In proof-systems one also investigates the property for non-terminating arguments, usually denoted as $\perp$, and infinite arguments (for instance a list with infinite length). Although it is possible to generate undefined and infinite arguments, it is impossible to stop the evaluation of the property when such an argument is used. This is a direct consequence of our decision to use ordinary compiled code for the evaluation of properties.

Restrictions of our current system are that the types should be known to the system (it is not possible to handle abstract types by generics); if there are restrictions on the types used they should be enforced explicitly;

and world access is not supported. In general it is very undesirable when the world (e.g. the file system on disk) is effected by random tests.

Currently the tester has to indicate that a property has to be tested by writing an appropriate Start function. In the near future we want to construct a tool that extracts the specified properties from CLEAN modules and tests these properties fully automatically.

GAST is not restricted to testing software written in its implementation language, CLEAN. It is possible to call a function written in some other language through the foreign function interface, or to invoke another program. This requires an appropriate notion of types in CLEAN and the foreign languages and a mapping between these types.

Acknowledgements

We thank Marko van Eekelen, Maarten de Mol, Peter Achten, Susan Evens and the anonymous referees for their contribution to GAST and this paper.

Chapter 7

Related Work on Generic Programming

In this chapter we review work related to generic programming approach used in the present thesis. We split the review in two sections. The first one focuses on the approaches to generic programming, whereas the second one focuses on the applications. We review the applications separately, because many of them can be implemented in more than one generic programming approach.

7.1 Approaches to Generic Programming

The structural view of a type has been used in multiple ways. Here we present an overview of the most important of the currently available approaches to generic programming and their implementations.

PolyP. PolyP [JJ97] is the first generic (polytypic) language extension for Haskell [JH02]. It is based on the fixed-point algebraic representation of a data type. The structure of types is made explicit by introducing special type constructors for type parameters, recursion, application, product and sum. However, the method allows for representation of the so called *regular* data types alone. Data type is called *regular* if it has one argument, it is not mutually recursive and not *nested*. A data type is called *nested* if its recursive occurrence on the right-hand side differs from that on the left-hand side. The restriction on regular data types limits practical applicability of the generic extension. Norell and Jansson [NJ03] show how to alleviate the restriction on nested data types.

The structure of a regular type in PolyP is represented by the type's *pattern functor*. A pattern functor is built from base structural types: binary sum and product types, applications, recursion and argument markers:

$$\begin{array}{lcl}
\textbf{type}\ \hat{\mathbb{1}}\ a\ b & = & \mathbb{1} \\
\textbf{type}\ (f\ \hat{\times}\ g)\ a\ b & = & f\ a\ b \times g\ a\ b \\
\textbf{type}\ (f\ \hat{+}\ g)\ a\ b & = & f\ a\ b + g\ a\ b \\
\textbf{type}\ \mathrm{Par}\ a\ b & = & a \\
\textbf{type}\ \mathrm{Rec}\ a\ b & = & b
\end{array}$$

Here $\hat{\mathbb{1}}$, $\hat{\times}$, and $\hat{+}$ are the lifted versions of $\mathbb{1}$, $\times$, and $+$ respectively.

For instance, for the list type the pattern functor (in a variable free form) is:

$$\mathrm{List}^{\circ} = \hat{\mathbb{1}}\ \hat{+}\ \mathrm{Par}\ \hat{\times}\ \mathrm{Rec}$$

The original type is isomorphic to the fix point of its pattern functor. For lists it is

$$\mathrm{List}\ a \cong \mu l.\mathrm{List}^{\circ}\ a\ l$$

The compiler can generate the instance of a generic function for an arbitrary regular data type. This procedure is called generic *specialization*. The generic specialization uses the structure of a type to generate the appropriate instance.

It is important that this approach is fully static. The generic specialization occurs at compile time and delivers typeable code. The generated code is, therefore, known at compile-time and amenable for front-end optimizations.

The structural representation used in PolyP reflects all aspects of the structure of types including the parameter, the application, and the recursion marking. Specifically, explicit recursion treatment enables implementation of such generic functions as *catamorphisms* (generic folds). However, such explicit handling of recursion cannot be used to handle mutually recursive or nested types, which limits PolyP to regular data types. Parameter marking imposes the restriction of unary functors.

Type-Indexed Values. The approach of *type-indexed values* [Hin00c] uses a simpler structural representation than that of PolyP. The structural representation consists only of products and sums of other types, thanks to which the restriction to regular types can be removed. This scheme can specialize generic functions to arbitrary types.

In this generic extension the list type is represented as

$$\text{List}^{\circ}\ a = 1 + a\ \times\ \text{List}\ a$$

Here the representation List° is not recursive - it refers to the original list. Data are converted to and from the representation as the data are processed. Only the top-level of the data structure is converted at each step. This property is crucial for handling mutually recursive and nested types.

A type is isomorphic to its structural representation. For lists:

$$\text{List}\ a \cong \text{List}^{\circ}\ a$$

Unlike in PolyP, the structural representation does not have explicit parameter, application and recursion marks. The absence of this essential information about the data types limits the number of generic functions that can be expressed with this approach. For instance, this approach prevents generic implementation of catamorphisms because recursion needs to be handled explicitly. However, as mentioned above, this approach allows for generic functions to be instantiated for an arbitrary type.

Due to its generality, the approach of type-indexed values is used as the basis for a number of generics implementations: Derivable Type Classes and Generic Haskell, described in the following paragraphs. It is also used as the basis for Generic Clean presented in chapter 2 of this thesis.

Derivable Type Classes. Derivable Type Classes of Glasgow Haskell [HP01] are based on the idea of type-indexed values. Type indexed functions are implemented as overloaded functions. Basically, the generic extension can derive instances automatically. The programmer provides only the base instances on the products and sums. The generic extension of Derivable Type Classes gives the semantics to the Haskell's *deriving* construct. It also enables the programmer to define his or her own classes whose instances can be derived.

This system does not support higher-order kinds: only instances of classes with class variables of kind $\star$ can be derived. This, for instance, precludes derivation of the mapping functions for functors of kind $\star \to \star$.

The approach of Derivable Type Classes combines overloading and generic programming. As noted in Section 1.2.2, a type class provides the interface of its instances. Generic programming complements it by allowing derivation of the instance implementations.

Generic Clean is based on Derivable Type Classes in the sense that it also combines generics with overloading. However, Generic Clean lifts the restriction of kind $\star$.

Kind-Indexed Types. Many generic functions can be defined similarly for types of different kinds. For instance,defined not only for kind $\star \rightarrow \star$, but also for kind $\star \rightarrow \star \rightarrow \star$. It turns out that mapping can be defined for types of arbitrary kinds. However, mapping has types of different shape for types of different kinds. Mapping for unary types (functors) takes one function as an argument, mapping for binary types (bi-functors) takes two functions and so on - the shape of the mapping function for a type depends on the kind of that type.

Hinze introduced the approach of *kind-indexed types* [Hin00c]. This approach makes it possible to define a generic function for types of arbitrary kinds by a single definition. Kind-indexing extends the approach of type-indexed (polytypic) values as indicated by the title of the Hinze's paper: *Polytypic values possess poly-kinded types* [Hin00c].

Kind-indexed types make it possible to derive specializations of generic functions to types of arbitrary kinds. A type indexed by a kind of order n yields a rank-n polymorphic type. To type-check the generated code for arbitrary kinds the type checker should support rank-n types.

In his thesis [Hin00a] Hinze describes a theoretical foundation for this approach used as the basis for Generic Haskell [CHJ+01] and GENERIC CLEAN. In this approach base cases of generic functions are defined in implicitly inductive way, as opposed to explicitly inductive way used in the original approach of type-indexed values. For instance, equality on product with explicit structural induction is defined as follows.

```
eq⟨a × b⟩ (Pair x1 y1) (Pair x2 y2) = eq⟨a⟩ x1 x2 && eq⟨b⟩ y1 y2
```

Here the equality on the components of a pair is named explicitly by eq⟨a⟩ and eq⟨b⟩. The implicitly inductive equality on products takes two additional arguments

```
eq⟨×⟩ eqa eqb (Pair x1 y1) (Pair x2 y2) = eqa x1 x2 && eqb y1 y2
```

Here the equality on components can be passed in these arguments. Clearly, the first definition is more intuitive than the second. However, the second definition is more flexible as it allows passing various functions that can be used to compare the components of the pair.

Generic Haskell. Generic Haskell is a source-to-source translator for Haskell. In his thesis [LÖ4] Andres Löh gives an excellent description of the ideas and algorithms of Generic Haskell. Like GENERIC CLEAN, Generic Haskell is based on the idea of type-indexed values and kind-indexed types.

Originally, Generic Haskell was directly based on the Hinze's idea of type-indexed values possessing kind-indexed types. The drawback of his approach is that generic functions are inductive implicitly rather than explicitly. It is harder for the programmer to write implicitly inductive generic functions because (s)he needs to think about additional arguments that would represent the induction on the substructure. The challenge becomes even higher when generic functions depend on other generic functions because the base cases would take an argument for each generic function involved.

In order to solve these problems of readability and dependency, Löh *et al* developed dependency style Generic Haskell [LCJ03]. With this approach the programmer can write generic functions with explicit induction and still have an option to alter the behavior for substructures. Actually, the compiler translates explicitly inductive definitions into implicitly inductive. Dependencies between generic functions are also handled by the compiler. Instead of dealing with special arguments the programmer uses the generic function names directly. With a special syntax, the user can redefine the behavior on the substructures. In his thesis [LÖ4] Löh uses dependency style to define the semantics of Generic Haskell.

As noted above, generic functions have a common ground with the Haskell- or CLEAN-like type class system. The class system is designed to give the same name for similar functions on different types. In the class system a type class can have several members, which allows for interdependence of the overloaded functions. A type class can also depend on super-classes. Generic Haskell duplicates this functionality without reusing the type-class system of Haskell.

Type-indexed types [HJL01] are another novel feature of Generic Haskell. The idea of indexing types by types is similar to that of indexing values by types. Type-indexed types appear to be useful in many practical examples. For instance, a type of finite mapping depends on the type of the key. Examples of applications where type-indexed types are used will be given in the next subsection.

The information about a data type required by a generic function depends on that generic function. For instance, mapping only requires structural information. However, generic *show* and *read* functions, i.e. pretty-printers and parsers require some name information as well. Some functions also require explicit information about recursion in the type, in the style of POLYP. One can say that different generic functions have different *views on data types.*

POLYP's view on data types is richer than that of type-indexed types in the sense that it keeps more information about the type. Therefore, more generic functions can be defined on it. However, it limits the set of instance types that can be represented in that view. It would be nice if the programmer could define the view used by a particular generic function. The programmer has a choice between a more general view that admits more types or a more specific view that allows for more specific treatment of some set of instance data types. Generic Haskell has several predefined views on data types [LÖ4]. However, currently the programmer cannot define his own new views.

Generic Haskell allows for mixing generic and specific behavior at the level of data constructors [CL03]. The programmer can override the generic behavior with the desired specific behavior per data constructor. This is achieved by associating a type with each constructor, so called *constructor case*. The desired behavior can then be specified by providing the instance for that constructor case. This feature of Generic Haskell is related to the original feature of GENERIC CLEAN (section 2.5), which does not require special types for constructor cases. There is a special syntax construct that allows a user defined instance to call the corresponding generated instance.

Generic Programming within Dependently Typed Programming. Normally, in a functional language the universe of types is separated from the universe of values. In particular, in the HM type system types do not depend on values. They can only be parameterized by other types (polymorphism).

However, some functional programming languages have type systems based on *dependent types* (e.g. Cayenne [Aug98], Epigram [MM04]). In such a type system a type can depend on a value. Dependent typing was originally used in proof assistants, from where it was adopted by some programming languages. Dependent typing is more precise than the usual typing: the type reflects the value properties more precisely.

Languages meant for programs development normally do not use dependent typing because of several software engineering drawbacks. In general, dependent typing is undecidable at compile-time, since types can depend on values known only at run-time. This contradicts to the primary purpose of a static type system - to detect errors at compile time. Another problem with dependent typing is that the compiler cannot *infer* function types any more: the types must be specified. Moreover, programming in such a dependently-typed language is more difficult compared to a conventional language, because the programmer needs to account for dependencies of types on values.

The generic specialization is usually implemented as part of the compiler.

However, Altenkirch and McBride [AM03] show that in a dependently typed language the generic specialization procedure can be directly encoded in the language itself. This means that the whole generic feature can be implemented as a library in a dependently typed language.

Generic Programming with meta-programming. Template meta-programming [SP02] allows the implementation of some programming language features as meta-programs, i.e. programs that are executed at compile time to yield parts of the program, which is then compiled in the usual way.

Norell and Janson [NJ04] have implemented a generic programming extension as a Template Haskell [SP02] meta-program. The current implementation of the Template Haskell lacks some features needed for full implementation of a generic programming scheme, but the prototypical implementation is possible. This meta-program is very compact and easy to modify and experiment with. However, it has a number of drawbacks in practice. First, it lacks syntactic sugar for convenient use of the generic feature. Second, its error messages refer to the generated code instead of the original meta-code, which makes it hard to find the errors.

The Constructor and The Pattern Calculi. The constructor calculus [Jay01] introduces an extended form of a pattern match. The idea is to allow patterns of different types to be used in a single chain of pattern matches. Generic functions are then ordinary first-class functions that perform a pattern match on constructors of the product and the co-product types. For instance, equality has the following shape

```
eq :: a a → Bool
eq Unit Unit                      = True
eq (x1 × x2) (y1 × y2)            = eq x1 y1 && eq x2 y2
eq (Inl x) (Inl y)                = x == y
eq (Inr x) (Inr y)                = x == y
eq x y                            = False
```

Here the pattern matches for product and co-product types are used together in a single function definition.

Some generic functions, like mapping, can be defined for types of arbitrary kinds. The approach of the constructor calculus allows such functions to be defined by a single first-class definition.

Indeed, these features require a sophisticated type system. The constructor calculus uses a *functorial type system*, a type system based on the notion of a functor. Generic functions are polymorphic in the data structure. In this type system a data type is represented as a functor of a tuple of types. The functor represents the data structure and the tuple represents the data. Polymorphism in the structure of data is captured by quantification over functors. The fact that a single generic function needs to work for types of different kinds requires a polymorphic kind system.

Not all generic functions can have a meaningful definition for all types. For instance, mapping or equality cannot be meaningfully defined for the arrow type. For this reason the constructor calculus makes a distinction between the data and non-data types.

The pattern calculus [Jay03] is the successor of the constructor calculus, and makes the pattern matching construct even more expressive because it allows matching on applicative terms. Variables are also allowed at the head of a pattern.

The pattern calculus has a *combinatorial type system*, a type system based on type combinators. In this type system types are represented as a composition of combinators based on SKI-combinators. This allows to lift some restrictions of the functorial type system.

With this approach it is easy to define generic functions that take generic arguments: such a generic function performs pattern match on that argument to accomplish its task. See for instance, equality above: it pattern matches on two such arguments. However, it is harder to define a function that has a generic result, e.g. generic *read* because there is nothing to perform a pattern match on.

The constructor and the pattern calculi are used as the semantic basis for the programming language FISh2.

Intensional Type Analysis. Intensional type analysis of Harper and Morrisett [HM95] allows for type-safe case distinction on types. A language with such case distinctions can be used as an intermediate language for compiling polymorphism. Specializations of polymorphic functions to specific types can be expressed with type case branches.

Weirich [Wei02] extends intensional type analysis for type constructors of arbitrary kind. Type cases can be used to compile polytypic functions. In this way, a generic function in the style of kind-indexed values can be compiled as a single first-class function.

Strategic Programming. *Strategic programming* [LVV02] is a form of generic programming that enables generic traversal of arbitrary heterogeneous data structures by means of so called traversal strategies. A particular traversal is comprises two parts: the action to be performed at each node of a data structure and a strategy that applies that operation recursively on the nodes. Separation of these two aspects is needed to achieve full control over the traversal. A particular traversal is built in a combinatorial style from strategy combinators and basic actions.

Strategic programming facilitates a mixture of generic and specific behavior. Generic behavior is given by the traversal strategies, whereas the specific behavior comes from the operations to be applied at each node.

Strafunski[LV02, LV03] is a Haskell based implementation of strategic programming. Strategies are implemented as library functions. They are first-class citizens of the language. The implementation is relatively simple, though it uses rank-2 polymorphism. It additionally uses a special type class Term that allows to convert any data type into a universal type TermRep and back. This type is used to perform the traversal of sub-structures. The DrIFT [WM03] preprocessor is used to generated instances of this class automatically.

The primary goal of strategic programming is to handle program transformation, re-factoring, re- and reverse engineering. In these applications, strategies are meant to run on parse or syntax trees. For this purpose Strafunski is equipped with a set of parsers to for Java, Haskell, XML and others.

Scrap your boilerplate. In "Scrap your boilerplate" papers [LP03, LP04] Lämmel and Peyton Jones further develop the idea of strategic programming to a general purpose generic programming approach. This approach enables implementation of generic functions such as read, show, comparison as well as functional strategies of Strafunski. However, functions that work for type constructors of arbitrary kind, are impossible to define.

The approach uses a type safe cast to selectively apply functions to values of the desired types. The implementation of type-safe casts is based on the Glasgow Haskell's Typeable class closely related to Clean's dynamic type and TC class (see section 1.3).

This approach does not use a special structural type representation to traverse the structure. Instead, a small set of traversals are directly encoded. The approach essentially uses rank-2 polymorphism (section 1.2.5) to apply the generic functions to substructures of different types. The basic set of traversals is implemented as a type class, whose instances can be easily generated. The specific behavior is applied to types that match by *name* and not by the structure. In this sense the approach is rather nominal than structural.

Lightweight generics and dynamics. The approach of *Lightweight Generics and Dynamics* [CH02a] combines generics and dynamics in a single framework. In this approach types have first-class representatives at the value level. Generic functions in this framework are first-class functions that interpret type arguments. For instance, the generic equality is a function of type

```
eq :: (TypeRep a) a a → Bool
```

The type representation TypeRep a reflects the structure of the type a. The system must maintain the invariant that TypeRep a is indeed the type representation of a.

Dynamics are pairs of values and their type representations with the real type hidden by the existential quantification:

```
:: Dynamic = ∃a: Dyn a (TypeRep a)
```

The disadvantage of this approach is that the programmer can break this invariant. Another disadvantage is that this approach is interpretative, hence, slow. Since the type representation arguments are mostly known at compile time, this last problem can probably be alleviated by a partial evaluation technique similar to those described in chapters 4 and 5 of this thesis.

7.2 Applications of Generic Programming

When generic programming just appeared, the domain of its applications was small. It included only generic versions of some library functions, like equality, mapping, pretty-printing. There were frequent complaints that generic programming was good only for implementing a dozen of generic functions. Indeed, generic programming could be used to define the generalized versions of many library functions. However, recent research has shown that applicability of generic programming stretches far beyond several simple functions. Generic programming has been successfully used in a number of non-trivial examples and software systems. Here we describe just some of them that use PolyP or the type-indexed generic scheme. We have chosen to enumerate the applications of generic programs separately from the generic languages used for their original implementations because many of the programs are actually or potentially implemented in more then one generic language.

Polytypic operations on terms. Terms need operations like pattern matching, rewriting, substitution, unification. Implementations of these are similar for different term types. Jansson and Jeuring give polytypic PolyP algorithms for these operations on generalized terms [JJ98, JJ00].

Generic tries. A trie is a data structure that allows for a fast search of information by a key, a finite mapping from keys to values. The original tries use string keys. Hinze and Jeuring [HJ03] generalize tries for an arbitrary key type. The lookup on a generic trie is a generic function defined defined by induction on the structure of keys. The Generic Haskell implementation uses type-indexed types to express the dependency of the trie type on the key type.

Generic XML tools. An XML document is a structured document. A valid XML document has the structure defined by its Document Type Definition (DTD). The DTD is the type of the document. The generic technique can be used to define generic operations by induction on the structure of DTD. XML tools include XML parser, validators, editors, compressors, encrypters etc. Hinze and Jeuring [HJ03] argue that many XML tools can be defined as *DTD-indexed programs*, similarly to type-indexed functions. They provide an example of such a tool, an XML compressor XComprez. XComprez is implemented in Generic Haskell.

Generic zipper. The *zipper* is a data structure for navigating in a tree. The data structure consists of a tree and a context that allows navigating to the left, right, up, and down in the tree. The function for navigation down shifts the focus to the left-most child of the node, the function for navigation to the left shifts the focus to the left sibling etc. Hinze and Jeuring [HJ03] present a generic version of the zipper. The generic zipper allows for traversal of any data structure rather than a particular tree type alone. The generic zipper navigation functions are defined by induction on the generalized tree structure. The Generic Haskell implementation uses dependencies and type indexed types. Dependencies are needed because the generic functions use each other. Type indexed types are used to express dependency of the navigation focus type from the tree type being navigated.

Data conversion functions. Jansson and Jeuring [JJ99, JJ02] study pairs of generic data conversion functions: generic traversals, generic splitting and merging data structures into shape and data, generic compressor and uncompressor, generic pretty printer and parser. It is proven that pairs of such conversion functions have certain inverse properties. For instance, the generic traversals are inverses of each other, or the uncompressor is the inverse of the compressor.

Generic transpose. Backhouse and Hoogendijk [BH02] generalize the transposition of matrixes to the transposition of arbitrary functors. A matrix can be represented as a list of lists, where the inner lists all have the same *length*. The transposition is a function of the type

$$\mathsf{transpose} :: \mathsf{List}\ (\mathsf{List}\ a) \rightarrow \mathsf{List}\ (\mathsf{List}\ a).$$

This function is generalized to work on a composition $F\ (G\ a)$ of arbitrary functors F and G; where the inner substructures $(G\ a)$ all have the same *shape*. The generalized transposition functions then has the type

$$\mathsf{transpose} :: F\ (G\ a) \rightarrow G\ (F\ a).$$

Norell and Jansson [NJ03] give a PolyP implementation of the generic transpose.

The automatic test system Gast. The test system Gast is a tool for automatic software functionality testing. Such a system ensures that the subject code behaves according to the specification. The subject code is invoked with various inputs. The results are then checked against the specification. Gast uses generic programming to generate test data in a systematic way, i.e. some coverage of input is achieved and no unnecessary duplication of test cases is performed. The input data type is used to guide the data generation. Gast is implemented in Clean and is described in Chapter 6 in detail. Van Weelden *et al.* [vWFO$^+$04] use it to test Smart Cards Applets written in Java. Gast is also used for testing software in the industry.

Graphical editor components. Achten *et al.* [AEP03, AEP04, AEPW04a, AEPW04b] use generics to automatically generate graphic user interfaces. The type determines how the data of that type are displayed and edited. The graphical editor library is implemented in Clean using the GUI library ObjectIO [AW00].

Chapter 8

Conclusions

The present thesis answers the research questions stated in the introduction:

Conceptual design. How do we incorporate generic programming into a functional language like CLEAN? More specifically, how do we combine a generic programming feature with the existing features of CLEAN?

Performance. How do we make the generated code adequate in terms of performance?

Applications. How useful is the generic programming in practice?

The research on the conceptual design has developed in two main directions: integration of type-indexed values with overloading and with dynamics. The research on optimization of generic programs has lead to first systematic study of the subject. Practical usefulness of generic programming has been studies on the example of a generic test system GAST. The following sections present the achievements and possible future work for each of these areas. In the end we give general remarks.

8.1 Integration with overloading

In the present thesis we show how type-indexed values can be integrated with overloading. The approach of GENERIC CLEAN, however, does not support mutual generic functions and dependencies between generic functions, whereas overloading allows for mutually recursive generic functions and dependencies in the form of class inheritance. Currently GENERIC CLEAN allows for only one member per kind-indexed class and kind indexed classes are not allowed to inherit from other classes.

Dependency Style Generic Haskell implements mutual recursion and dependencies without use of overloading, i.e. it duplicates many of the features already provided by overloading. Additionally Generic Haskell allows for explicit style of specifying generic recursion, which is simpler for the programmer than the implicit style used in Clean and earlier versions of Generic Haskell.

We see a tighter integration of generics and overloading as a perspective direction of future work. This integration should allow for mutually recursive generic functions and generic functions that depend on other generic functions. Additionally, for the same reasons as in Generic Haskell we would need to introduce the explicit style of generic recursion.

8.2 Integration with dynamics

Generic functions work on arbitrary types; dynamics can contain values of arbitrary types. It is natural if these two concepts are inter-operable, if generic functions can work on values of the Dynamic type. However, dynamics exist at run-time, whereas generics are translated at compile-time. Moreover, generic functions are not first-class citizens of the language. Instead, they are schemes that are used to generate instance functions. Therefore, generic functions themselves cannot be stored in dynamics. Only their instances can be stored.

Chapter 3 shows how how to make generic functions to inter-operate with dynamics. In particular, generic functions can be turned into first class functions. Additionally, we show how to work with dynamics in a generic way.

However, we have not given a practical solution for the problem of representing the contents of dynamics in a generic structural way. Wichers Schreur and Plasmijer solved it in [WSP04]. This allows for programming in a style close to the one from the "boilerplate" programming (see section 7.1 or [LP03, LP04]), which allows to traverse data structures of arbitrary types and apply specific operations at nodes of a certain type.

8.3 Optimization

In this thesis we have introduced two program transformation algorithms that are able to optimize generic programs. The first one (chapter 4) based on pure partial evaluation with no termination analysis. The termination problem for recursive generic instances is solved by abstracting from recursion by means of the fix-point combinator. In this sense this technique is specific for optimization of generics. The fix-point abstraction trick only works for generic instances that have top-level recursion. This restriction determines the class of generic programs, for which the algorithm *completely* removes the generic overhead: kind-indexed types (i.e. generic functions's types) should not refer to recursive types as they lead to non-toplevel recursion. However, The instance types may be recursive. This class is large, but it does not cover many practically important examples. We have proven that for this class the generic overhead is removed. Another disadvantage is that the transformation algorithm is tailored specifically for generic programs.

The second algorithm (chapter 5) is a general purpose optimizer that is applicable not only for generic programs. It is based on fusion; it uses off-line termination analysis. This algorithm is able to completely eliminate the generic overhead in a much larger class of generic programs. This class covers many practically important examples. It is a future work to proven formally that this is the case.

This algorithm currently ignores the aspects of sharing. We believe that adding support for sharing will not disturb optimization of generic functions. However, since the algorithm is general-purpose, it is essential to account for sharing. Sharing can be added in a way similar to that of the previous fusion algorithm. This is a direction of future work.

Another direction of future work is to study how this algorithm can be used to optimize programs that involve different kinds of combinators: for instance monadic, arrow and parser combinators. Combinatorial style of programming involves a lot of higher-order functions and intermediate data structures. It is interesting to study how they can be eliminated.

An alternative approach to optimization of generic programs would be a transformation techniques crafted specifically for optimization of generated code. Due to its intimate knowledge of the generic specialization process, this specific technique could lead to optimization of a larger class of generic programs than the general purpose algorithm.

Combining generic specialization and optimization into a single algorithm could lead to a generic scheme that directly yields optimal code that does not have generic constructors. A compiler built in this way would be more efficient, since it does not first create inefficient code and than optimize it.

8.4 The application: Gast.

The test system GAST is a real application used to test real software systems, e.g. [vWFO$^+$04]. GAST is written in GENERIC CLEAN: it makes an essential use of generic programming to automatically and systematically generate test input data.

8.5 Other remarks

It is important in practice to be able to mix generic and specific behavior. There are several methods proposed in the literature.

Generic approach presented here uses a structural representation of essentially only three type constructors UNIT, PAIR and EITHER. Some generic functions (*pretty-printer*) make use of additional type constructor CONS that to the information about constructors, such as name and arity. This generic extension can handle almost all imaginable types.

POLYP uses a structural representation with more constructors. Generic functions in POLYP can only be specialized to so called *regular types*. However, more generic functions can be defined for regular types in POLYP then in CLEAN (e.g. generic unification).

It looks that the structural representation should differ depending on the generic function. So, we need to parameterize generic functions by the generic representation it uses. This leads us to the concept of views of types. A view on a type is an isomorphic representation of that type. The author believes that views will increase expressive power and, hence, applicability of generic programming.

Views are partially implemented in Generic Haskell. However, Generic Haskell has only choice between predefined views. It would be nice to let the programmer to specify his new views on types.

Bibliography

[AAP02] Peter Achten, Artem Alimarine, and Rinus Plasmeijer. When generic functions use dynamic values. In R. Peña, editor, *The 14th International workshop on the Implementation of Functional Languages, IFL'02, Selected Papers*, volume 2670 of *LNCS*, pages 17–33. Madrid, Spain, Springer, September 2002.

[ACP$^+$92] M. Abadi, L. Cardelli, B. Pierce, G. Plotkin, and D. Rèmy. Dynamic typing in polymorphic languages. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, San Francisco, June 1992.

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[AEP03] Peter Achten, Marko van Eekelen, and Rinus Plasmijer. Generic graphical user interfaces. In *Proceedings 15th International Workshop on the Implementation of Functional Languages, IFL 2003, Selected Papers*, Edinburgh, Scotland, September 2003. Springer.

[AEP04] Peter Achten, Marko van Eekelen, and Rinus Plasmijer. Compositional model-views with generic graphical user interfaces. In *PADL 2004: Practical Aspects of Declarative Programming*, Dallas, Texas, USA, June 2004. Springer.

[AEPW04a] Peter Achten, Marko van Eekelen, Rinus Plasmijer, and Arjen van Weelden. Automatic generation of editors for higher-order data structures. In *Proceedings of the Second ASIAN Symposium on Programming Languages and Systems (APLAS 2004)*, Taipei, Taiwan, November 2004.

[AEPW04b] Peter Achten, Marko van Eekelen, Rinus Plasmijer, and Arjen van Weelden. Gec: a toolkit for generic rapid prototyping of type safe interactive applications. In *Summer School on Advanced Functional Programming*, University of Tartu, August 2004. Springer.

[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.

[AH02] Peter Achten and Ralf Hinze. Combining generics and dynamics. Technical report NIII-R0206, Nijmegen Institute for Computing and Information Sciences, Faculty of Science,University of Nijmegen, Nijmegen, The Netherlands, July 2002.

[AJ01] Klaus Aehlig and Felix Joachimski. Operational aspects of normalization by evaluation. Submitted to MSCS, 2001.

[AM03] Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In *Proceedings of the IFIP TC2/WG2.1 Working Conference on Generic Programming*, pages 1–20. Kluwer, B.V., 2003.

[AP02] Artem Alimarine and Rinus Plasmijer. A Generic programming extension for Clean. In Thomas Arts and Markus Mohnen, editors, *The 13th International workshop on the Implementation of Functional Languages, IFL'01, Selected Papers*, volume 2312 of *LNCS*, pages 168–186. Älvsjö, Sweden, Springer, September 2002.

[AS04a] Artem Alimarine and Sjaak Smetsers. Efficient generic functional programming. Technical Report NIII-R0425, Nijmegen Institute for Computing and Information Sciences, University of Nijmegen, The Netherlands, Jun 2004.

[AS04b] Artem Alimarine and Sjaak Smetsers. Fusing generic functions. Technical Report NIII-R0434, Nijmegen Institute for Computing and Information Sciences, Faculty of Science,University of Nijmegen, Nijmegen, The Netherlands, August 2004.

[AS04c] Artem Alimarine and Sjaak Smetsers. Optimizing generic functions. In Dexter Kozen, editor, *The 7th International Conference, Mathematics of Program Construction*, number 3125 in LNCS, pages 16 – 31. Stirling, Scotland, UK, Springer, July 2004.

[AS05] Artem Alimarine and Sjaak Smetsers. Fusing generic functions. In *To appear. The 7th International Symposium on Practical Aspects of Declarative Languages, PADL 2005*, Long Beach, California, January 2005.

[Aug98] Lennart Augustsson. Cayenne - a language with dependent types. In *International Conference on Functional Programming*, pages 239–250, 1998.

[AW00] Peter Achten and Martin Wierich. A tutorial to the clean object I/O library - version 1.2. Technical report, University of Nijmegen, Nijmegen, The Netherlands, February 2000.

[BFV$^+$99] A. Belinfante, J. Feenstra, R. Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment. In *Int. Workshop on Testing of Communicating Systems 12*, pages 179–196, 1999.

[BGM91] Gilles Bernot, Marie Claude Gaudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *Softw. Eng. J.*, 6(6):387–405, 1991.

[BH02] Roland Backhouse and Paul Hoogendijk. Generic properties of datatypes, 2002.

[bri01] Testing transition systems: An annotated bibliography. 2067:186–195, 2001.

[BS96] Erik Barendsen and Sjaak Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6(6):579–612, 1996.

[BvEvL$^+$87] T. Brus, M.C.J.D. van Eekelen, M.O. van Leer, , and M.J. Plasmeijer. Clean: A language for functional graph rewriting. In *In Kahn. G. ed. Proceedings of the Third International Conference on Functional Programming Languages and Computer Architecture*, pages 364–384, Portland, Oregon, USA, LNCS 274, 1987. Springer-Verlag.

[CA01] Juan Chen and Andrew Appel W. Dictionary passing for polytypic polymorphism. Technical report tr-635-01, Princeton University Computer Science, March 2001.

[CH00] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. *SIGPLAN Not.*, 35(9):268–279, 2000.

[CH02a] James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 90–104. ACM Press, 2002.

[CH02b] Koen Claessen and John Hughes. Testing monadic code with quickcheck. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 65–77. ACM Press, 2002.

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

[CHJ$^+$01] D. Clarke, R. Hinze, J. Jeuring, A. Löh, and J. de Wit. The generic haskell user's guide, 2001.

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

[CL03] Dave Clarke and Andres Löh. Generic haskell, specifically. In *Proceedings of the IFIP TC2/WG2.1 Working Conference on Generic Programming*, pages 21–47. Kluwer, B.V., 2003.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM Press, 1982.

[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.

[dMvEP02] M. de Mol, M. van Eekelen, and R. Plasmeijer. Theorem proving for functional programmers, SPARKLE: A functional theorem prover. In Thomas Arts and Markus Mohnen, editors, *The 13th International workshop on the Implementation* *of Functional Languages, IFL'01, Selected Papers*, volume 2312 of *LNCS*, page 55. Älvsjö, Sweden, Springer, September 2002.

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

[FJTJ96] J. Fernandez, C. Jard, and C. Viho T. Jéron. Using on-the-fly verification techniques for the generation of test suites. 1102, 1996.

[FMMW98] Loe M. G. Feijs, F. A. C. Meijs, J. R. Moonen, and J. J. van Wamel. Conformance testing of a multimedia system using PHACT. In *Proceedings of the IFIP TC6 11th International Workshop on Testing Communicating Systems*, pages 193–210. Kluwer, B.V., 1998.

[Hin69] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[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. A new approach to generic functional programming. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT.* Symposium on Principles of Programming Languages,Boston, Massachusetts, January 2000.

[Hin00c] 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 of *LNCS*, pages 2–27. Springer, July 2000.

[HJ03] Ralf Hinze and Johan Jeuring. Generic haskell: Applications. In Roland Backhouse and Jeremy Gibbons, editors, *Generic*

Programming, volume 2793 of *LNCS*, pages 57 – 97. Springer-Verlag, 2003. Technical report ICS Utrecht University, UU-CS-2003-016.

[HJL01] R. Hinze, J. Jeuring, and A. Löh. Type-indexed datatypes, 2001.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, California, 1995.

[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.

[HT99] J. He and K. Turner. Protocol-inspired hardware testing. In *Int. Workshop on Testing of Communicating Systems 12*, pages 179–196, 1999.

[hun] HUnit home page. http://hunit.sourceforge.net.

[Jay01] C. Barry Jay. Distinguishing data structures and functions: the constructor calculus and functorial types. In S. Abramsky, editor, *Typed Lambda Calculi and Applications: 5th International Conference TLCA 2001, Kraków, Poland, May 2001 Proceedings*, volume 2044 of *Lecture Notes in Computer Science*, pages 217–239. Springer, 2001.

[Jay03] C. Barry Jay. The pattern calculus, 2003. accepted for publication by ACM Trans. on Progr. Lang. and Sys.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation.* Prentice-Hall, Inc., 1993.

[JH02] S. Peyton Jones and J. Hughes. Report on the programming language haskell 98 – a non-strict, purely functional language, 2002.

[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.

[JJ98] Patrik Jansson and Johan Jeuring. Functional pearl: Polytypic unification. *Journal of Functional Programming*, 8(5):527–536, September 1998.

[JJ99] Patrik Jansson and Johan Jeuring. Polytypic compact printing and parsing. In S. Doaitse Swierstra, editor, *Proceedings 8th European Symposium on Programming, ESOP'99, Amsterdam, The Netherlands, 22–28 March 1999*, volume 1576, pages 273–287. Springer-Verlag, Berlin, 1999.

[JJ00] Patrik Jansson and Johan Jeuring. A framework for polytypic programming on terms, with an application to rewriting. Utrecht University, 2000. UU-CS-2000-19.

[JJ02] Patrik Jansson and Johan Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002.

[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.

[JS04] Simon Peyton Jones and Mark Shields. Practical type inference for arbitrary-rank types, April 2004.

[jun] JUnit home page. http://junit.sourceforge.net.

[KATP02] Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. Gast: Generic automated software testing. In R. Peña, editor, *The 14th International workshop on the Implementation of Functional Languages, IFL'02, Selected Papers*, volume 2670 of *LNCS*, pages 84–100. Madrid, Spain, Springer, September 2002.

[KJG99] A. Kerbrat, T. Jéron, and R. Groz. Automated test generation from sdl specifications. In *The Next Millennium–Proceedings of the* 9^{th} *SDL Forum*, pages 135–152, 1999.

[LÖ4] Andres Löh. *Expolring Generic Haskell*. Phd thesis, Institute Voor Programmatuurkunde en Algoritmiek, 2004. in preparation.

[LCJ03] Andres Löh, Dave Clarke, and Johan Jeuring. Dependency-style generic haskell. *SIGPLAN Not.*, 38(9):141–152, 2003.

[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.

[LP03] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003. Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[LP04] Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings; International Conference on Functional Programming (ICFP 2004)*. ACM Press, September 2004. 12 pages; To appear.

[LV02] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *Proc. Practical Aspects of Declarative Programming PADL 2002*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, January 2002.

[LV03] R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P. Wadler, editors, *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of *LNCS*, pages 357–375. Springer-Verlag, January 2003.

[LVK00] R. Lämmel, J. Visser, and J. Kort. Dealing with Large Bananas. In J. Jeuring, editor, *Proceedings of WGP'2000, Technical Report, Universiteit Utrecht*, pages 46–59, July 2000.

[LVV02] Ralf Lämmel, Eelco Visser, and Joost Visser. The Essence of Strategic Programming. 18 p.; Draft, October15 2002.

[LY96] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines – a survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1123, August 1996.

[Mil78] Robin Milner. A theory of type polymorphism in programming languages. 17:348–375, 1978.

[MM04] C McBride and J McKinna. The view from the left. *Journal of Functional Programing*, 14(1):69–111, 2004.

[MR01] Graeme E. Moss and Colin Runciman. Inductive benchmarking for purely functional data structures. *Journal of Functional Programming*, 11:525–556, 2001.

[Myc84] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *Proceedings of the 6th Colloquium on International Symposium on Programming*, pages 217–228. Springer-Verlag, 1984.

[NJ03] Ulf Norell and Patrik Jansson. Polytypic programming in haskell. In Greg Michaelson and Phil Trinder, editors, *The 14th International workshop on the Implementation of Functional Languages, IFL'03*, LNCS. Pollock Halls, Edinburgh, Scotland, Springer, September 2003. To appear.

[NJ04] Ulf Norell and Patrik Jansson. Prototyping generic programming using Template Haskell. In Dexter Kozen, editor, *The 7th International Conference, Mathematics of Program Construction*, number 3125 in LNCS, pages 314 – 333. Stirling, Scotland, UK, Springer, July 2004.

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

[NSvEP91] Eric Nocker, Sjaak Smetsers, Marko van Eekelen, and Rinus Plasmeijer. Concurrent clean. In Leeuwen Aarts and Rem, editors, *Proc. of Parallel Architectures and Languages Europe (PARLE '91)*, volume 505, pages 202–219. Springer-Verlag, 1991.

[OL96] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Proc. 23rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 65–67, January 1996.

[PE93] Rinus Plasmeijer and Marko Van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley Longman Publishing Co., Inc., 1993.

[PE02] Rinus Plasmeijer and Marko van Eekelen. Concurrent clean language report (version 2.0), 2002.

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

[Pil98] Marco Pil. Dynamic types and type dependent functions. In *Implementation of Functional Languages*, pages 169–185, 1998.

[Pil04] Marco Pil. First class file I/O. phd thesis in preparation, 2004.

[PvE01] M.J. Plasmeijer and M. van Eekelen. *Clean Language Report Version 2.0.* University of Nijmegen, The Netherlands, 2001. draft.

[SP02] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.

[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*, volume 31 of *SIGPLAN Notices*, pages 206–214. ACM Press, May 1996.

[TB02] J. Tretmans and E. Brinksma. Côte de resyste – automated model based testing. In *Progress 2002 – 3^{rd} Workshop on Embedded Systems*, pages 246–255, 2002.

[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.

[TWC01] J. Tretmans, K. Wijbrans, and M. Chaudron. Software engineering with formal methods: The development of a storm surge barrier control system – revisiting seven myths of formal methods. *Formal Methods in System Design*, 19(2):195–215, 2001.

[vEdM] M. van Eekelen and M. de Mol. Reasoning about explicit strictness in a lazy language using mixed lazy/strict semantics.

[vESP96] Marko C. J. D. van Eekelen, Sjaak Smetsers, and Marinus J. Plasmeijer. Graph rewriting semantics for functional programming languages. In *CSL*, pages 106–128, 1996.

[VP02] Martijn Vervoort and Rinus Plasmeijer. Lazy dynamic input/output in the lazy functional language Clean. In R. Peña, editor, *Proceedings 14th International Workshop on the Implementation of Functional Languages (IFL 2002)*, volume 2670 of *LNCS*, Madrid, Spain, September 2002. Springer Verlag.

[vWFO+04] Arjen van Weelden, Lars Frantzen, Martijn Oostdijk, Pieter Koopman, and Jan Tretmans. On-the-fly formal testing of a smart card applet. Technical Report R0428, Nijmegen Institute for Computing and Information Sciences (NIII), 2004.

[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.

[Wad90] Philip Wadler. Comprehending monads. In *Conference on Lisp and Functional programming*, pages 61–78, June 90.

[WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.

[Wei02] Stephanie Weirich. Higher-order intensional type analysis. In *European Symposium on Programming*, pages 98–114, 2002.

[WM03] Noel Winstaley and John Meacham. Drift user's guide. http://repetae.net/john/computer/haskell/DrIFT/drift.ps, December 2003.

[WP02] Arjen van Weelden and Rinus Plasmeijer. Towards a strongly typed functional operating system. In R. Peña, editor, *Proceedings 14th International Workshop on the Implementation of Functional Languages (IFL 2002)*, volume 2670 of *LNCS*, Madrid, Spain, September 2002. Springer Verlag.

[WSP04] Ronny Wichers Schreur and Rinus Plasmijer. Dynamic construction of generic functions. Christian-Albrechts-

Universitaet zu Kiel, pp. 482, Sep 2004. Proceedings Implementation and Application of Functional Languages, 16th International Workshop, IFL'04,,Luebeck, Germany,.

Summary

As the time passes, software systems rapidly become still more complex. The tools used to develop such software systems can hardly keep up with this complexity. In particular, one of the most important tools for software development is a programming language. Functional programming languages are designed to be high-level languages that allow the programmer to focus on the problem solving rather than on the implementation details. Due to the mathematical nature of functional programming languages, their compilers can use formal reasoning methods to check programs for consistency and to translate high-level programs into efficient executable code.

Generic functional programming enables programming on even a higher level. Instead of writing similar functions for different data types, the programmer can specify one function that works for arbitrary data types. Writing these similar functions is typically uninteresting and boring, which often leads to programming errors. Generic programming helps to decrease the time and costs of software development and maintenance. Typical examples of generic functions are the equality operator, *show* (pretty-printer) and *read* (parser), which can be defined for many types.

This thesis explores the field of generic functional programming in three main directions: language design, optimization and applications.

Chapter 2 is devoted to the generic programming language design. We show how the support for generic programming can be built into a lazy functional programming language such as Clean. Our design is based on Hinze's *kind-indexed* types. With the approach of kind-indexed types, instances of generic functions can be derived for arbitrary types of arbitrary kinds. The novelty of our approach is that it uses type classes and overloading as the basis for generic functions. Overloading allows to give the same names to similar functions for different types. Generic programming allows to derive similar implementations of these functions. Therefore, overloading and generics complement each other. For instance, this allows any instance of the overloaded equality operator to be derived automatically.

Chapter 3 focuses on interaction of the generic programming support with another important feature of modern functional languages: *dynamics*. Dynamics allow for a statically typed language to use the advantages of dynamic typing. A dynamic is a pair of a value and its type packed together. This is similar to type tagging used in the implementation of dynamically typed languages.

Dynamics allow strongly typed programs to link in external code *at run-time* in a type safe way. Generic programming allows programmers to write code schemes that can be specialized *at compile-time* to arguments of arbitrary type. Both generics and dynamics have been investigated and incorporated in CLEAN.

Because generic functions work on all types and values, they are a perfect tool when manipulating dynamic values. The interaction of generics and dynamics is important for the following reason. Values of dynamic type can be normally handled by type case distinction on the type stored in the dynamic. Code that uses such a type case can do only a trivial handling of types that are not explicitly matched by the type case. Generic functions allow to treat dynamics containing values of arbitrary, also yet unknown, types. For instance, one can write an equality function that works for dynamics holding values of arbitrary types.

Generics rely on compile-time specialization, whereas dynamics rely on run-time type checking and linking. This seems to be a fundamental contradiction. We show that the contradiction does not exist. ¿From any generic function we derive a function that works on dynamics. This function takes a dynamic type representation as an argument. Programs that use this technique combine the best of both worlds: they have concise universal code that can be applied to any dynamic value regardless of its origin. This technique is important for application domains such as type-safe mobile code and plug-in architectures.

Chapters 4 and 5 are devoted to optimization of generic programs. In the proposed generic approach, the compiler derives instances of generic functions for the requested instance types. The derivation of instances is performed by inductive interpretation of the structure of types. The definition of a generic function provides the bases cases for this induction. These base cases determine the inductive interpretation.

Due to this interpretative nature of generic code, it is extremely slow. The interpretation involves a lot of conversions between real data types and their structural representations. Additionally, it involves a lot of higher-order functions. The performance of such generated code is far from its hand-written counterpart. In fact, the performance is so poor that it compromises usefulness of generics in practice.

We propose a program transformation technique that is able to optimize generic programs. For a large class of such programs the optimization yields code that is close to the hand-written code written specifically for the concerned data types. This optimization is based on partial evaluation of programs at compile-time. More precisely, we propose an extended fusion algorithm. This is the first systematic study of optimization of generic programs. However, these fusion algorithm improvements are not specific for generic programs only. The proposed optimization algorithm is a general purpose algorithm, which can be used to improve performance of functional programs in general.

Generic programming is not limited to just a small set of standard examples, such as equality, show and read functions. Usefulness of generic programming was recognized by the functional language community leading to many interesting applications, which include generic graphical editors, generic XML tools, term processing functions.

In chapter 6 we present one more example of a real program that makes an essential use of the generic programming techniques: an automatic testing system GAST. Properties of functions and data types can be expressed in first-order logic. GAST automatically and systematically generates appropriate test data, evaluates properties of these values, and analyzes the test results. The distinguishing property of GAST is that the test data are generated in a systematic and generic way using generic programming techniques. This implies that there is no need for the user to indicate how data should be generated. Moreover, duplicated tests are avoided, and for finite domains GAST is able to prove a property by testing it for all possible values.

For generic programming to become useful in practice it has to integrate into the overall language design, its performance must be adequate, and its utility has to be shown by a number of real-life examples. The present study makes some contributions to all of these points.

Samenvatting

Software wordt in de loop der tijd steeds maar complexer. De hulpmiddelen die men gebruikt om software te ontwikkelen kunnen deze toenemende complexiteit nauwelijks bijbenen. Een van de belangrijkste hulpmiddelen voor het ontwikkelen van software is een programmeertaal. Functionele programmeertalen zijn ontworpen om de programmeur in staat te stellen een probleem op een hoog niveau van abstractie op te lossen, waardoor de programmeur zich kan concentreren op de essentie van het probleem in plaats van zich te moeten bekommeren om diverse implementatiedetails.

Door de wiskundige grondslag van functionele programmeertalen, kan men formele, wiskundige redeneermethodes gebruiken om de consistentie van programma's (semi-)automatisch te controleren. Bovendien is het mogelijk de hoog niveau programma's te vertalen naar efficiënt uitvoerbare machinecode.

Een *generische* functionele programmeerstijl maakt het mogelijk programma's op een nog hoger niveau van abstractie te ontwikkelen. De programmeur hoeft niet langer meerdere functies te schrijven voor gelijksoortige problemen die enkel verschillen in de gegevenstypes van de gebruikte objecten. In de plaats daarvan hoeft de programmeur slechts één functie te specificiëren die toepasbaar is op ieder denkbaar gegevenstype. Hiermee worden programmeerfouten voorkomen die het schrijven van verschillende gelijkaardige functies, iets wat uitermate oninteressant en saai is, met zich meebrengt.

Generische programmeren zorgt ervoor dat software-ontwikkeling en onderhoud minder tijd en kosten zullen vergen. Typische voorbeelden van generische functies zijn de gelijkheidsoperator, een *show* (pretty-printer) en *read* een (syntactische parser), die voor vele types kunnen worden gedefinieerd. Dit proefschrift onderzoekt het generisch functioneel programmeren in drie belangrijke richtingen: taalontwerp, optimalisering en toepassingen.

Hoofdstuk 2 is gewijd aan het ontwerp van een generische programmeertaal. Wij laten zien hoe men ondersteuning voor generisch programmeren kan inbouwen in een luie (lazy) functionele programmeertaal zoals CLEAN. Het ontwerp is gebaseerd op de *kind-gendexeerde* typen van Hinze.

Met deze aanpak van *kind*-geïndexeerde types, kunnen instanties van generische functies worden afgeleid voor willekeurige types van willekeurige *kinds*. Nieuw aan deze aanpak is dat typeklassen en overloading gebruikt worden als basis voor generische functies. Overloading maakt het mogelijk om dezelfde naam te gebruiken voor gelijkaardige functies van verschillend type. Generisch programmeren maakt het mogelijk gelijkaardige implementaties van deze functies af te leiden. Daarom vullen overloading en generische functies elkaar aan. Men kan nu, bijvoorbeeld, iedere denkbare instantie van de overloaded gelijkheidsoperator automatisch afleiden.

In hoofdstuk 3 onderzoeken we de combinatie van generisch programmeren en een andere belangrijke uitbreiding van moderne functionele talen: *dynamics*. Dynamics maken het mogelijk in een statisch getypeerde taal dynamische typering te gebruiken. In een dynamic wordt een waarde samen met zijn type opgeslagen. Dit lijkt op *type-labels* zoals die gebruikt worden in de implementatie van dynamisch getypeerde talen (ook wel *type tagging* genoemd).

Dynamics maken het mogelijk dat sterk getypeerde programma's tijdens executie op een type veilige manier kunnen worden uitgebreid met externe code (*plug-ins*). Generisch programmeren stelt de programmeur in staat generieke code te schrijven die *compile-time* gespecialiseerd wordt voor argumenten van het concreet aangeboden type.

Zowel generics als dynamics zijn onderzocht en opgenomen in CLEAN.

Omdat generische functies toegepast kunnen worden op willekeurige types en waarden, vormen zij een perfect hulpmiddel voor het manipuleren van dynamische waarden. De koppeling van generics met dynamics is belangrijk om de volgende reden. Men kan de waarden van een dynamisch type inspecteren via gevalsonderscheiding op basis van het type dat in de dynamic is opgeslagen. Als het type echter niet past in een van de onderscheiden gevallen kan men enkel nog triviale bewerkingen op de dynamische waarden verrichten. Generische functies maken het mogelijk dynamics te bewerken die waarden van willekeurige of zelfs onbekende types bevatten. Men kan, bijvoorbeeld, een gelijkheidsfunctie schrijven die werkt op dynamics met waarden van een willekeurig, voor het programma onbekend type.

Generics worden tijdens vertaling gespecialiseerd, terwijl bij dynamics juist tijdens de uitvoering van het programma de aangeboden types worden geverifieerd en ontbrekende code wordt ingevoegd. Dit lijkt een fundamentele tegenspraak te zijn. Wij tonen echter aan dit niet het geval is.

Voor een willekeurige generische functie kan men een functie afleiden die op een dynamic werkt. Deze functie neemt een representatie van een dynamisch type als argument. Programma's die deze techniek gebruiken combineren het beste uit beide werelden: zij bevatten beknopte universele code die kan worden toegepast op een willekeurige dynamische waarde, ongeacht zijn oorsprong. Deze technologie is van belang voor toepassingen zoals typeveilige mobiele code en plug-in architectuur.

De hoofdstukken 4 en 5 gaan over het optimaliseren van generische programma's. In de voorgestelde generische aanpak, leidt de vertaler automatisch instanties af van de generische functies voor de aangeboden concrete types. Dit wordt bereikt door een inductieve interpretatie van de structuur van types. De definitie van een generische functie bevat de basisgevallen voor deze inductie. Uit deze basisgevallen kan de inductieve interpretatie worden afgeleid.

De code die uit deze interpretatie voortkomt is echter uiterst langzaam. Men moet namelijk iedere concrete waarde converteren naar zijn structurele representatie en weer terug. Voor deze omzetting worden diverse hogere-orde functies gebruikt die inefficiënte implementaties hebben. De kwaliteit van de gegenereerde code is dan ook veel slechter dan die van handmatig geschreven code. Het is zelfs zo slecht dat daardoor generisch programmeren praktisch niet haalbaar leek.

In het proefschrift introduceren we een automatische transformatietechniek, een zogenaamde fusietechniek, waarmee generische programma's geoptimaliseerd kunnen worden. Voor een grote klasse van generische programma's kan men daarmee code genereren die ook qua efficiëntie vergelijkbaar is met hand geschreven code die op maat voor het concrete data type is gemaakt. De optimalisatie is gebaseerd op partiële evaluatie van programma's door de vertaler.

Verder introduceren wij een uitbreiding van het fusie-algoritme. Het is de eerste keer dat optimalisatie technieken voor generische programma's systematisch bestudeerd zijn. Het ontwikkelde fusiealgoritme is niet alleen geschikt voor generische programma's, maar kan tevens gebruikt worden om de efficiëntie te verbeteren van functionele programma's die geen gebruik maken van generische functies.

Generisch programmeren is veel breder toepasbaar dan de bekende reeks standaardvoorbeelden, zoals gelijkheid, "show"- en "read"-functies. Het nut van generisch programmeren is inmiddels algemeen erkend door de functionele talen-gemeenschap en heeft geleid tot vele interessante toepassingen zoals generische grafische editors en generische hulpmiddelen voor XML-toepassingen.

In hoofdstuk 6 presenteren we een ander voorbeeld van een toepassing die gebruik maakt van generische technieken, te weten een automatisch test systeem genaamd GAST. In GAST kunnen eigenschappen van functies en data worden uitgedrukt in eerste-orde logica. GAST genereert automatisch en systematisch geschikte test-data, test vervolgens of deze gegenereerde waarden de opgegeven eigenschappen inderdaad bezitten en analyseert de testresultaten. Het kenmerkende van GAST is dat de testgegevens op een systematische en generieke manier gegenereerd worden, uiteraard gebruikmakend van generische programmeertechnieken. De gebruiker hoeft niet meer expliciet aan te geven hoe en welke data moet worden gegenereerd. Het aanleveren van identieke testdata wordt vermeden. Voor eindige domeinen kan GAST zelfs een eigenschap bewijzen doordat het alle mogelijke waarden systematisch test.

Generisch programmeren zal pas op grote schaal praktisch bruikbaar zijn wanneer het een integraal onderdeel vormt van programmeertalen, wanneer de resulterende code snel genoeg is, en wanneer de bruikbaarheid is aangetoond voor concrete "real-life" toepassingsgebieden. Dit proefschrift levert een bijdrage aan ieder van de deze onderdelen.

Curriculum Vitae

Artem Alimarine was born on 2nd December 1971, in Moscow, Russia. In 1989 he started to study physics at the Physics Department of the Moscow State University. For the Masters degree he conducted research in the field of computer simulation of magnetic materials using the cell automata method. He finished the University in 1995 with the Masters degree in Physics. In 1995-1998 he worked as a software engineer at the Moscow Software Center of Digital Equipment Corporation, where he gained practical experience in software engineering. In 1998 he moved to The Netherlands. In 1998-2000 he worked as a software engineer at Philips Medical Systems.

In 2000 he started his Ph.D. at the Software Technology Group University of Nijmegen. In 2000-2004 he conducted research in the field of generic functional programming under supervision of prof. M. J. Plasmeijer.