Enumerating Well-Typed Terms Generically

Alexey Rodriguez Yakushev[1] Johan Jeuring[2,3]

[1]Vector Fabrics B.V., Paradijslaan 28, 5611 KN Eindhoven, The Netherlands
[2]Department of Information and Computing Sciences, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
[3]School of Computer Science, Open University of the Netherlands, P.O. Box 2960, 6401 DL Heerlen, The Netherlands

alexey.rodriguez@gmail.com johanj@cs.uu.nl

Abstract

We use generic programming techniques to generate well-typed lambda terms. We encode well-typed terms by generalized algebraic datatypes (GADTs) and existential types. The Spine approach (Hinze et al. 2006; Hinze and Löh 2006) to generic programming supports GADTs, but it does not support the definition of generic producers for existentials. We describe how to extend the Spine approach to support existentials and we use the improved Spine to define a generic enumeration function. We show that the enumeration function can be used to generate the terms of simply typed lambda calculus.

1. Introduction

This paper discusses the problem of given a type, generate lambda terms of that type. There exist several algorithms and/or tools for producing lambda terms given a type (Augustsson 2005; Katayama 2005; Koopman and Plasmeijer 2007). The approach discussed in this paper uses generic programming techniques on Generalized Algebraic Datatypes (GADTs) and existentials to enumerate well-typed lambda terms. The enumeration function is much simpler than previous work, and the main problem lies in making generic programming techniques available for GADTs and existentials in functions that produce values of a particular datatype, such as an enumeration function.

Since their introduction to Haskell, Generalized Algebraic Datatypes (GADTs) (Xi et al. 2003; Cheney and Hinze 2003; Peyton Jones et al. 2006) are often used to improve the reliability of programs. GADTs encode datatype invariants by type constraints in constructor signatures. With this information, the compiler rejects values for which such invariants do not hold during type-checking. In particular, GADTs can be used to model sets of well-typed terms such that values representing ill-typed terms cannot be constructed. Other applications of GADTs include well-typed program transformations, implementation of dynamic typing, staged computation, ad-hoc polymorphism and tag-less interpreters.

Given the growing relevance of GADTs, it is important to provide generic programming support for generalized datatype definitions. The generation of datatype values using generic programming is of particular interest. Generic value generation has been used before to produce test data, which can be used to check the validity of program properties (Koopman et al. 2003). In generic value generation, the datatype definition acts as a specification for test data. However, this specification is often imprecise, since it gives rise to either values that do not occur in practice, or, worse, ill-formed values (for example, a program fragment with unbound variables). For this reason, QuickCheck (Claessen and Hughes 2000) allows the definition of custom generators.

GADT definitions may specify types more precisely than normal datatypes. In the case of well-typed terms, the constraints in the datatype definition describe the formation rules of a well-typed value. It follows that a generic producer function on a GADT might produce values that are better suited for testing program properties. For example, it should be possible to use a generic value generation function with a GADT encoding lambda calculus, in order to produce a well-typed lambda term with which a tag-less interpreter can be tested.

The *spine* view (Hinze et al. 2006; Hinze and Löh 2006), which is based on "Scrap Your Boilerplate" (Lämmel and Peyton Jones 2003), is the only approach to generic programming in Haskell that supports GADTs. The main idea behind the spine view is to make the application of a data constructor to its arguments explicit. The spine view represents a datatype value by means of two cases: the representation of a datatype constructor, and the representation of the application to constructor arguments. A generic function can then be defined by case analysis on the spine view. Hinze and Löh (2006) describe how to use the spine view to represent GADT values and define generic functions to consume and produce such values.

Besides GADT definitions, our definition of well-typed terms uses existentially quantified type variables. In particular, the type of expression application is that of the function return type. The argument type is hidden from the application type and is therefore existentially quantified. Under certain conditions, the spine view supports the definition of generic functions that *consume* existentially typed values. Unfortunately, it cannot be used to define a generic function that *produces* them. It follows that the spine view cannot in general be used to define generic enumerators for well-typed terms.

This paper extends the spine view to allow the use of producer functions on existential types. We make the following contributions:

- We show how to support existential types systematically within the spine view. We extend the spine view to encode existentially quantified type variables explicitly. This enables the definition of generic functions that perform case analysis on such types. As a consequence, the extended spine view supports the definition of generic producers that work on existential types. We demonstrate the increased generality by defining generic serialization and deserialization for existential types and GADTs.
- We define a generic enumeration function that can be used with GADTs and existential types. This function can be used to enumerate the well-typed terms represented by a GADT. Consider a GADT that represents terms in the simply typed lambda calculus. The enumeration of terms with type $\mathsf{Expr}\ ((\mathsf{b} \to \mathsf{c}) \to (\mathsf{a} \to \mathsf{b}) \to \mathsf{a} \to \mathsf{c})$ yields the term that corresponds to function

composition. The enumeration function requires explicit support for existential types in producers. For that reason it cannot be defined in approaches such as that of Hinze and Löh (2006).

This paper is organized as follows. Section 2 introduces the spine view and gives several examples illustrating why this view is not suitable to define producers for existential types. Section 3 describes our extensions to the spine view, which enable producer support for existential types. Section 4 uses the extended spine view to define a generic enumeration function. The enumeration function is then used to produce well-typed lambda calculus terms. Section 5 discusses related work, and section 6 concludes.

2. The spine view

The spine view was introduced by Hinze et al. (2006). This view supports the definition of generic functions that consume (such as *show* or *eq*) or transform (such as *map*) datatype values. We introduce the spine view using the generic show function as an example. This function prints the textual representation of a value based on the type structure encoded by the view. To implement this function, we need case analysis on types to implement type-dependent behavior.

2.1 Case analysis on types

The spine view uses GADTs to implement case analysis on types. We define a type representation datatype where each constructor represents a specific type:

```
data Type :: * → * where
  Int    :: Type Int
  Maybe  :: Type a → Type (Maybe a)
  Either :: Type a → Type b → Type (Either a b)
  List   :: Type a → Type [a]
  ( :→ ) :: Type a → Type b → Type (a → b)
```

An overloaded function can be implemented by performing case analysis on types. To perform case analysis on types we pattern match on the type representation values. The GADT pattern matching semantics (Peyton Jones et al. 2006) ensures that the type variable a is refined to the target type of the matched constructor:

```
show :: Type a → a → String
show Int          n        = showInt n
show (Maybe a)    (Just x)  = paren ("Just"  • show a x)
show (Maybe a)    Nothing   = "Nothing"
show (Either a b) (Left x)  = paren ("Left"  • show a x)
show (Either a b) (Right y) = paren ("Right" • show b y)
show (List a)     ((:) x xs) = paren ("(:)"
                               • show a x
                               • show (List a) xs)
show (List a)     []        = "[]"
```

This function prints a textual representation of a datatype value. Note that we choose to print lists in prefix syntax rather than the usual Haskell notation. The operator (•) separates two strings with a white space, and *paren* prints parentheses around a string argument.

2.2 The spine representation of values

Type representations can be used to implement overloaded functions, but such functions are not generic. The user needs to define new *show* cases for every datatype added to the program. To define generic functions, we make use of the spine view.

The spine view represents all datatype values by means of two cases: a constructor and the application of a (partially applied) constructor to an argument. This is embodied in the Spine datatype:

```
data Spine :: * → * where
  Con   :: ConInfo a → Spine a
  (:◇:) :: Spine (a → b) → Typed a → Spine b
infixl 0 :◇:
```

The *Con* case of the spine view stores a value constructor of type a together with additional information including the constructor name, the fixity, and the constructor tag. This additional information is stored in the datatype ConInfo. The application case (:◇:) consists of a functional value Spine that consumes a-values, and the argument a paired with its type representation in the datatype Typed. We show Typed, and a simplified ConInfo containing only the constructor name below:

```
data ConInfo a = ConInfo{conName :: String, conVal :: a}
data Typed a   = (:▷:)     {val :: a, rep :: Type a}
```

To write a generic function, we first convert a value to its Spine representation. We show how to perform this conversion using the type-indexed function *toSpine*:

```
toSpine :: Type a → a → Spine a
toSpine Int          x          = Con (conint x x)
toSpine (Maybe a)    (Just x)   = Con (conjust Just) :◇: x :▷: a
toSpine (Maybe a)    Nothing    = Con (connothing Nothing)
toSpine (Either a b) (Left x)   = Con (conleft Left) :◇: x :▷: a
toSpine (Either a b) (Right y)  = Con (conright Right) :◇: y :▷: b
toSpine (List a)     ((:) x xs) = Con (concons (:))
                                    :◇: x :▷: a :◇: xs :▷: List a
toSpine (List a)     []         = Con (connil [])
```

Because we reuse the constructor information in later sections of the paper, we define ConInfo values separately. We give some examples below:

```
conint :: Int → a → ConInfo a
conint i     = ConInfo (showInt i)

connothing, conjust :: a → ConInfo a
connothing = ConInfo "Nothing"
conjust    = ConInfo "Just"
```

In summary, to enable generic programming using the spine view, we define a GADT for type representations, the Spine datatype, and conversions from datatype values to their spine representations. The conversions for datatypes are written only once, and then the same conversion can be reused for different generic functions. The conversion to the spine representation is regular enough that it can be automatically generated from the syntax trees of datatype declarations[1]

Equipped with the spine representation, we can write a number of generic functions. For example, this is the definition of generic *show*:

```
show :: Type a → a → String
show rep x = paren (gshow (toSpine rep x))

gshow :: Spine a → String
gshow (Con con)      = conName con
gshow (con :◇: arg) = gshow con • show (rep arg) (val arg)
```

This function is a simplified variant of the *show* function defined in the Haskell prelude. All datatype values are printed uniformly: constructors are separated from the arguments by means of the • operator, and parentheses are printed around fully applied constructors.

[1] At the time of writing, Template Haskell cannot handle GADT declarations. Our prototype generates the spine representation for a GADT using a manually constructed declaration syntax tree instead of parsing the GADT declaration and processing it via Template Haskell.

2.3 Transformer functions

The spine view also supports the definition of generic transformer functions. Examples of such functions include incrementing all Int values in a tree, and applying a function to all nodes of a specific type in a tree.

To write such a function, we need to convert the spine representation back to the represented value *after* it has been traversed and transformed. This is achieved by the *fromSpine* function:

```
fromSpine :: Spine a → a
fromSpine (Con con)      = conVal con
fromSpine (con :◇: arg) = fromSpine con (val arg)
```

See Hinze et al. (2006) for examples of transformer functions using the spine view.

2.4 A view for producers

It is impossible to write *read*, the inverse to *show* using the current Spine datatype. We could for example use the following type for *read*:

```
read :: Type a → String → [(a, String)]
```

This function produces all possible parses of type a (paired with unused input) from a representation for the type a and an input string. To write such a generic function, we would need a spine representation to guide the parsing process. Unfortunately, a representation Spine a cannot be used for this purpose. A value of Spine a represents a particular value of type a (for example, a singleton list) rather than the full datatype structure (a description of the cons and nil constructors and their arguments). To enable a generic definition of generic read and other producer generic functions, Hinze and Löh (2006) introduce the type spine view. This view describes all values of a rather than a particular one.

```
type TypeSpine a = [Signature a]
data Signature :: ∗ → ∗ where
  Sig   :: ConInfo a → Signature a
  (:⊕:) :: Signature (a → b) → Type a → Signature b
infixl 0 :⊕:
```

Here we again have two cases, one for encoding a constructor and another for the application of a (partially) applied constructor to an argument. The application case contains only a type representation and no argument value anymore. A value of TypeSpine a is a list of constructor signatures representing all constructors of the represented datatype. The type-indexed function *typeSpine* produces the type spine representations of all datatypes on which generic programming is to be used.

```
typeSpine :: Type a → TypeSpine a
typeSpine Int          = [Sig (conint i i)
                          | i ← [minBound . . maxBound]]
typeSpine (Maybe a)    = [Sig (connothing Nothing)
                          , Sig (conjust Just) :⊕: a]
typeSpine (Either a b) = [Sig (conleft Left)  :⊕: a
                          , Sig (conright Right) :⊕: b]
typeSpine (List a)     = [Sig (connil [ ])
                          , Sig (concons (:))  :⊕: a :⊕: List a]
```

The generic parsing function, *read*, builds a parser that deserializes a value of type a:

```
read :: Type a → Parser a
```

For the purposes of this paper, we assume that Parser is an abstract parser type with a monadic interface, with some standard derived functions:

```
return       :: a → Parser a
(>>=)        :: Parser a → (a → Parser b) → Parser b
ap           :: Parser (a → b) → Parser a → Parser b
(>>)         :: Parser a → Parser b → Parser b
noparse      :: Parser a
alternatives :: [Parser a] → Parser a
readInt      :: Parser Int
lex          :: Parser String
token        :: String → Parser ()
readParen    :: Parser a → Parser a
```

The definition of generic read uses *readInt* to read an integer value. For other datatypes, we make parsers for each of the constructor representations and merge all the alternatives in a single parser.

```
read :: Type a → Parser a
read Int = readInt
read rep = alternatives [readParen (gread conrep)
                        | conrep ← typeSpine rep]
```

The generic parser of a constructor is built by induction on its signature representation. The base case (*Sig*) tries to recognize the constructor name and returns the constructor value. The application case parses the function and argument parts recursively and the results are combined using monadic application:

```
gread :: Signature a → Parser a
gread (Sig c)         = token (conName c) >> return (conVal c)
gread (con :⊕: arg) = gread con ‘ap‘ read arg
```

In the definition of generic read, we could also have used parser combinators based on an applicative interface (McBride and Paterson 2007) instead of a monadic one. For an example of parser combinators with an applicative interface see Swierstra and Duponcheel (1996). In Section 3.1 we show that existentially typed values cannot be parsed using purely applicative parser combinators, because generic read on existentials makes essential use of bind (>>=).

2.5 Generalized algebraic datatypes

Recall that generalized algebraic datatypes are datatypes to which type-level constraints are added. Such constraints can be used to encode invariants that datatype values must satisfy. For example, we can define a well-typed abstract syntax tree by having the syntactic categories of constructs in the target type of constructors:

```
data Expr :: ∗ → ∗ where
  EZero   :: Expr Int
  EFalse  :: Expr Bool
  ESuc    :: Expr Int  → Expr Int
  ENot    :: Expr Bool → Expr Bool
  EIsZero :: Expr Int  → Expr Bool
```

We have constants for integer and boolean values, and operators that act on them.

GADTs can easily be represented in the spine view. For instance, the definition of *toSpine* for this datatype is as follows:

```
toSpine :: Type a → a → Spine a
. . .
toSpine (Expr Int)  EZero       = Con (conezero   EZero)
toSpine (Expr Bool) EFalse      = Con (conefalse  EFalse)
toSpine (Expr Int)  (ESuc e)    = Con (conesuc    ESuc)
                                  :◇: e :▷: Expr Int
toSpine (Expr Bool) (ENot e)    = Con (conenot    ENot)
                                  :◇: e :▷: Expr Bool
toSpine (Expr Bool) (EIsZero e) = Con (coneiszero EIsZero)
                                  :◇: e :▷: Expr Int
```

This definition requires the extension of Type with the representation constructors *Expr* and *Bool*. Generic functions defined on the spine view, such as generic show, can now be used on Expr.

What about generic producer functions? These can be used on Expr too, because it is also possible to construct datatype representations for GADTs in the type spine view:

```
typeSpine :: Type a → TypeSpine a
...
typeSpine (Expr Int)  = [Sig (conezero   EZero)
                        ,Sig (conesuc    ESuc) :⊕: Expr Int]
typeSpine (Expr Bool) = [Sig (conefalse  EFalse)
                        ,Sig (conenot    ENot) :⊕: Expr Bool
                        ,Sig (coneiszero EIsZero) :⊕: Expr Int]
```

To parse boolean expressions, we invoke the generic read function as follows:

```
readBoolExpr :: Parser (Expr Bool)
readBoolExpr = read (Expr Bool)
```

The parser for integer expressions would use a different argument for *Expr*. In this example, we are assuming that the expression to be parsed is always of a fixed type. A more interesting scenario would be to leave the type of the GADT unspecified and let it be dynamically determined from the parsed value. This would be useful if the programmer wants to parse some well-typed expression regardless of the type that the expression has.

A possible solution to parsing a GADT without specifying its type argument would be to existentially quantify over that argument in the result of the parsing function. Next, we discuss how the spine view deals with existential types.

2.6 Existential types and consumer functions

In Haskell, existential types are introduced in constructor declarations. A type variable is existentially quantified if it is mentioned in the argument type declarations but omitted in the target type. For example, consider dynamically typed values:

```
data Dynamic :: ∗ where
  DynVal :: Type a → a → Dynamic
```

The type variable a in the declaration is existentially quantified. It is used to hide the type of the a-argument used when building a Dynamic value. The type a is kept abstract when pattern matching a Dynamic value, but by case analyzing the type representation it is possible to dynamically recover the type a. Thus, statically, Dynamic values all have the same type, but, dynamically, the type distinction can be recovered and acted upon.

To represent dynamic values in Spine, we add type representations for Type itself and Dynamic. Hence, we add the following two constructors to Type:

```
data Type :: ∗ → ∗ where
  ...
  Type    :: Type a → Type (Type a)
  Dynamic :: Type Dynamic
```

Now, Dynamic values may be represented as follows by the spine view:

```
toSpine :: Type a → a → Spine a
...
toSpine Dynamic (DynVal rep val) = Con (condynval DynVal)
                                   :◇: rep :▷: Type rep
                                   :◇: val :▷: rep
```

While Dynamic values may be easily represented, this is not the case for all datatypes having existential types. Recall that in a spine representation, every constructor argument is paired with its type representation in the datatype Typed. In general, in constructors having existential types, it may not be possible to build such a pair because the representation of the existential type may be missing. The constructor *DynVal* is a special case, because it carries the representation type of the existential a. For an example where the representation of a constructor with existential types is not possible, consider adding an application constructor to the expression datatype:

```
data Expr :: ∗ → ∗ where
  ...
  EApp :: Expr (a → b) → Expr a → Expr b
```

and consider the corresponding *toSpine* alternative:

```
toSpine :: Type a → a → Spine a
...
toSpine (Expr b) (EApp fun arg) = Con (coneapp EApp)
                                  :◇: fun :▷: Expr (a :→ b)
                                  :◇: arg :▷: Expr a
```

This code is incorrect due to the unbound variable *a* which stands for the existential representation. The conclusion here is that the spine view can be used on an existential type, as long as the constructor in which it occurs carries a type representation for it.

2.7 Existential types and producer functions

The view for producer functions, the type spine view, cannot represent existential types equally well as the spine view. For instance, consider how to generate such a representation for dynamic values:

```
typeSpine :: Type a → [Signature a]
...
typeSpine Dynamic = [Sig (condynval DynVal) :⊕: Type a :⊕: a]
```

What should the representation *a* be? There are two options, we either fix it to a single type representation or we range over all possible type representations. Choosing one type representation would be too restrictive, because *read* would only parse dynamic values of that type and fail on any other type. We try the second option:

```
typeSpine Dynamic = [Sig (condynval DynVal) :⊕: Type a :⊕: a
                    | a ← types]
```

This code does not yet have the behavior we desire. For *typeSpine* to be type-correct, *types* must return a list of representations all having the same type. Because Type is a singleton type (each type has only one value), *types* returns a single type representation. We would like *types* to generate a list of all possible type representations, but different type representations have different types. Therefore, *types* should return representations whose represented type is existentially quantified. To this end, we define the type of boxed type representations:

```
data BType = ∀a.Boxed (Type a)
applyBType :: (∀a.Type a → c) → BType → c
applyBType f (Boxed a) = f a
```

Now we can define the type spine of dynamic values, for which we assume a list of boxed representations (*types*):

```
types :: [BType]
typeSpine Dynamic = [Sig (condynval DynVal) :⊕: Type a :⊕: a
                    | Boxed a ← types]
```

The boxed representations are used to construct a list of constructor signatures that represent a dynamic value of the corresponding type. There are infinitely many type instances of polymorphic types, therefore there are infinitely many Dynamic constructor representations. An infinite type spine is not a desirable representation to work with. The *read* function would try to parse the input using every Dynamic constructor representation. If there is a correct parse, parsing would eventually succeed with one of the representations. However, if there is no correct parse, parsing would not terminate. Moreover, this representation precludes implementing more efficient variants of parsing.

Infinite type spine representations for datatypes with existentials make the use of generic producers on such datatypes unpractical. Before describing a modified type spine view that solves this problem, we explore a couple of non-generic examples to motivate our design decisions.

We start with the parser definition for Dynamic values. In the code above, we are able to parse any possible dynamic value because there are *DynVal* constructor signatures for all possible types. For each signature, we build a parser that parses the corresponding type representation and a value having that type.

Now, rather than parsing the two arguments of the constructor *DynVal* independently, we introduce a dependency on representations. First, we parse the type representation for the existential. Then, we use it to build a parser of the corresponding type and parse the second argument. In this way, we no longer need to have an infinite representation of types because we obtain the representation of the existential during the parsing process:

```
read :: Type a → Parser a
...
read Dynamic = do
  Boxed a ← readType
  value   ← read a
  return (DynVal a value)
```

To this end, we use a function that parses type representations. Because the result may be of an arbitrary type, *readType* produces a representation that is boxed:

```
readType :: Parser BType
```

We defer the presentation of *readType* to Section 3.4.

The same technique can be used to parse any constructor having an existential type. For example, the definition for parsing expression applications is as follows:

```
read (Expr b) = do
  Boxed a ← readType
  fun     ← read (Expr (a :→ b))
  arg     ← read (Expr a)
  return (EApp fun arg)
```

In this example, the type representation that is parsed is used to build the type representations for the two remaining arguments.

These two examples show that constructors with existential types must be handled differently from other constructors. In such constructors, the constructor argument representations depend on the type representation of the existential type. In our examples, this dependency is witnessed by the dynamic construction of parsers based on the type representation that was previously parsed.

3. An improved Spine view: support for existential types

We start this section by showing how to extend the spine view for producers to represent existential types explicitly. Then, we show why this extension is also necessary for the consumer spine view.

3.1 The existential case for producer functions

We have learned two things from the *read* examples for constructors with existential types. First, we need a way to represent existential variables explicitly, so that generic functions can handle existential type variables specifically. And second, there is a dependency from constructor arguments on the existential variable. For example, we can only parse the function and argument parts of an expression application, if we have already parsed the existential type representation. We modify the type spine view to accommodate these two aspects. We extend the constructor signatures in this view with a constructor to represent existential quantification: *AllEx*. The dependency of type b on an existential type a is made explicit by means of a function from type representations of type a to representations of b.

```
data Signature :: ∗ → ∗ where
  Sig   :: a → Signature a
  (:⊕:) :: Signature (a → b) → Type a → Signature b
  AllEx :: (∀a.Type a → Signature b) → Signature b
```

Interestingly, the type variable a is universally rather than existentially quantified. Why is this the case? The type spine view represents all possible values of a datatype, therefore the existential variable must range over all possible types. This also explains the name of the constructor *AllEx*, which stands for all existential type representations.

There is another modification to the type spine view. The *Sig* constructor no longer carries constructor information. Instead, this information is stored at the top-level of the representation:

```
type TypeSpine a = [ConInfo (Signature a)]
```

This change is not strictly necessary but it is convenient. Suppose that the constructor information is still stored in *Sig*. Now, applications that need to perform a pre-processing pass using constructor information (for example, for more efficient parsing) would be forced to apply the function in *AllEx* only to obtain the constructor information. Having this information at the top-level, rather than at the *Sig* constructor, avoids the trouble of dealing with *AllEx* unnecessarily.

The function *typeSpine* has to be modified to deal with the new representation:

```
typeSpine :: Type a → TypeSpine a
typeSpine Int        = [ conint i (Sig i)
                       | i ← [minBound..maxBound]]
typeSpine (Maybe a)  = [connothing (Sig Nothing)
                       ,conjust    (Sig Just    :⊕: a)]
typeSpine (Either a b) = [conleft  (Sig Left    :⊕: a)
                       ,conright   (Sig Right   :⊕: b)]
typeSpine (List a)   = [connil     (Sig [])
                       ,concons (Sig (:) :⊕: a :⊕: List a)]
typeSpine Dynamic    =
  [condynval (AllEx (λa → Sig (DynVal a) :⊕: a))]
```

Now let us rewrite the *read* function using the new type spine view. First of all, the constructor is parsed in *read*, because the constructor information is now at the top-level:

```
read :: Type a → Parser a
read Int = readInt
read rep = alternatives [ readParen (conParser conrep)
                        | conrep ← typeSpine rep]
  where conParser conrep = token (conName conrep)
                           ≫ gread (conVal conrep)
```

The function that performs generic parsing is not very different for the first two Signature constructors:

```
gread :: Signature a → Parser a
gread (Sig c)         = return c
gread (con :⊕: arg) = gread con ‘ap‘ read arg
```

The existential case is the most interesting one. We first parse the type representation, and then continue with parsing the remaining part of the constructor.

```
gread (AllEx f) = readType ≫= applyBType (gread ∘ f)
```

This example effectively captures the *read* examples for dynamically typed values and for expression applications. The type representation is used to build the parser for the remaining constructor arguments. This dependency is expressed using the bind operation on parsers (≫=). This means that the definition of generic read for existential types must be based on monadic parser combinators, and therefore applicative parser combinators cannot be used.

There is one function that we use to read type representations:

```
readType :: Parser BType
```

Because type representations are somewhat special, we deal with them separately in Section 3.4.

3.2 Choice in the representation of existentials

There a choice in the representation of existential quantification. Consider the representation of *DynVal* given above. The function argument of *AllEx* receives a type representation and uses it to build the partially applied constructor value *Sig* (*DynVal a*). This value requires only one more argument which is represented by *a*.

An alternative way to encode *DynVal* is to make all of the constructor arguments explicit:

```
typeSpine :: Type a → TypeSpine a
...
typeSpine Dynamic =
  [condynval (AllEx (λa → Sig DynVal :⊕: Type a :⊕: a))]
```

The two approaches differ in whether a generic function has access to the type representation in the application case (:⊕:). It would seem that the second representation of *DynVal* is more flexible because it would allow the production of values different than *a* for the first argument. However, Type is a singleton type, so the only value (excluding $\perp$) that inhabits the type represented by *Type a* is *a* itself. It follows that the second representation of *DynVal* is not an improvement over the first. For this reason, we always choose to expose the representation of an existential by means of the existential case only (*AllEx*).

3.3 The existential case for consumer functions

Producer functions need a modified type spine view (TypeSpine) to handle existential types. Do we need to modify the spine view (Spine) for consumers too? After all, we were able to define *toSpine* for Dynamic using the existing view. There is a good reason why we still need to modify the spine view to handle existentials in an appropriate way. Consider the *read* and *show* functions for example. There is a clear dependency on the representation of existential types during parsing. It is not possible (or at least very impractical) to parse a dynamic value without first having the type representation for it. Therefore, existential type representations should appear earlier than the constructor arguments that depend on it in the text input used for parsing. This means that *show* must pretty print the type representation for the existential before the dependent constructor arguments. However, the current spine view makes this difficult because the representation for the existential may appear in any position.

We solve the problem above making the dependence between existential types and constructor arguments explicit. Like the type spine view, the new constructor encodes the dependency on existentials using a function. The type variable is existentially quantified because in this case we are representing a specific constructor value:

```
data Spine :: ∗ → ∗ where
  Con   :: a → Spine a
  (:◇:) :: Spine (a → b) → Typed a → Spine b
  Ex    :: Type a → (Type a → Spine b) → Spine b
```

As in the type spine view, the constructor information is lifted out of the *Con* constructor onto the top-level. The new function *toSpine* is as follows:

```
toSpine :: Type a → a → ConInfo (Spine a)
toSpine Int          x           = conint x      (Con x)
toSpine (Maybe a)    (Just x)    = conjust       (Con Just
                                                   :◇: x :▷: a)
toSpine (Maybe a)    Nothing     = connothing    (Con Nothing)
toSpine (Either a b) (Left x)    = conleft       (Con Left
                                                   :◇: x :▷: a)
toSpine (Either a b) (Right y)   = conright      (Con Right
                                                   :◇: y :▷: b)
toSpine (List a)     (x : xs)    =
  concons (Con (:) :◇: x :▷: a :◇: xs :▷: List a)
toSpine (List a)     []          = connil        (Con [])
toSpine Dynamic      (DynVal a x) = condynval    (Ex a dynSig)
  where dynSig a = Con (DynVal a) :◇: x :▷: a
```

The *show* function is modified as follows to use the constructor information that appears at the top-level:

```
show :: Type a → a → String
show rep x = paren (conName spinecon
                     • gshow (conVal spinecon))
  where spinecon = toSpine rep x
```

Generic show does not change much for the two first spine cases:

```
gshow :: Spine a → String
gshow (Con con)     = ""
gshow (con :◇: arg) = gshow con • show (rep arg) (val arg)
```

For the existential case, generic show prints the type representation first and continues printing the remaining constructor values:

```
gshow (Ex a f) = showType a • gshow (f a)
```

The function for printing type representations is explained next:

```
showType :: Type a → String
```

3.4 Handling type representations

In the example above, we have used the function *readType* to parse a type representation. The function *readType* returns a boxed representation since the represented type is dynamically determined during parsing. Unfortunately, it is not easy to define producers that return boxed representations using generic programming. If special care is not taken, such functions may loop when invoked. In the following we describe the problem in more detail and we propose a solution.

3.4.1 Parsing type representations

The obvious way to parse a type representation is to do it generically by using the *read* function. To this end, we use generic read to parse boxed type representations:

```
readType :: Parser BType
readType = read BType
```

Unfortunately, the function given above is non-terminating. First, remember that BType uses existential quantification, and hence its type spine is:

```
typeSpine BType = [conboxed (AllEx (λa → Sig (Boxed a)))]
```

Since the type spine uses an existential case, *gread* would try to parse a BType-value calling *readType* recursively. Therefore, trying to parse a boxed type representation would lead to parsing an existential type, which leads to parsing a boxed type representation and so on.

How can we solve this problem? A desperate solution would be to give up using generic programming in the definition of *readType*. This approach is undesirable because every generic producer would need to have a type representation case. Worse even, every such case would have to handle all type representation constructors. If there are *n* generic functions and *m* represented types, the programmer would need to write $n \times m$ cases. Despite this significant problem, it is worth exploring a non-generic variant of *readType* and try to generalize it.

```
readType = alternatives (map readParen
                [ do token "Int"
                     return (Boxed Int)
                , do token "Maybe"
                     Boxed arg ← readType
                     return (Boxed (Maybe arg))
                , do token "Either"
                     Boxed left  ← readType
                     Boxed right ← readType
                     return (Boxed (Either left right))
                , do token "List"
                     Boxed arg ← readType
                     return (Boxed (List arg))
                ])
```

This example shows that parsing a type representation is no different than parsing a normal datatype in that the type argument of the GADT plays no role here. This example also illustrates the verbosity of writing such boilerplate without using generic programming.

The code of *readType* suggests that we could forget the "GADT-ness" of type representations during parsing. This is the first step we take towards being able to define generic producers for boxed representations, namely, defining the datatype of type codes, a non-GADT companion to type representations:

```
data TCode :: * where
  CInt      :: TCode
  CMaybe    :: TCode → TCode
  CEither   :: TCode → TCode → TCode
  CList     :: TCode → TCode
  CArrow    :: TCode → TCode → TCode
  CType     :: TCode → TCode
  CDynamic  :: TCode
  CTCode    :: TCode
```

Besides naming and the absence of a type argument, this datatype is identical to type representations. To make the relation between type codes and type representations precise, we introduce two conversion functions. The first function converts a type representation to a type code, erasing the type information in the process:

```
eraseType :: Type a → TCode
eraseType Int          = CInt
eraseType (Maybe a)    = CMaybe  (eraseType a)
eraseType (Either a b) = CEither (eraseType a) (eraseType b)
eraseType (List a)     = CList   (eraseType a)
eraseType (a :→ b)     = CArrow  (eraseType a) (eraseType b)
eraseType (Type a)     = CType   (eraseType a)
eraseType Dynamic      = CDynamic
eraseType TCode        = CTCode
```

Conversely, we want to be able to convert from a type code to a type representation. Note, however, that the resulting type-index depends on the value of the type code and hence the result is a boxed representation:

```
interpretTCode :: TCode → BType
interpretTCode CInt         = Boxed Int
interpretTCode (CMaybe a)   = applyTCode (Boxed ∘ Maybe) a
interpretTCode (CEither a b) =
  applyTCode (λr → applyTCode (Boxed ∘ Either r) b) a
interpretTCode (CList a)    = applyTCode (Boxed ∘ List) a
interpretTCode (CArrow a b) =
  applyTCode (λr → applyTCode (Boxed ∘ (r :→ )) b) a
interpretTCode (CType a)    = applyTCode (Boxed ∘ Type) a
interpretTCode CDynamic     = Boxed Dynamic
interpretTCode CTCode       = Boxed TCode

applyTCode :: ∀c.(∀a.Type a → c) → TCode → c
applyTCode f code = applyBType f (interpretTCode code)
```

Using type codes it is now possible to implement parsing of type representations generically. To implement *readType*, we parse a type code value and then we interpret it to obtain a type representation:

```
readType :: Parser BType
readType = read TCode ≫= return ∘ interpretTCode
```

Here *TCode* is the type representation for type codes, we do not show the spine and type spine views for this datatype as they are no different from that of other datatypes.

Showing a type representation was no problem previously, we could have written *showType* as follows:

```
showType :: Type a → String
showType a = show (Type a) a
```

However, to remain compatible with *read* we use type codes as the means to pretty print type representations:

```
showType :: Type a → String
showType = show TCode ∘ eraseType
```

Summarizing, *readType* is a special function. It cannot be defined by instantiating *read* to boxed representations. Such an instantiation leads to non-termination because parsing a boxed representation uses the existential case of generic parsing, which in turn makes the recursive call to *readType*. To solve this problem, we defined type codes, a non-GADT analogue of type representations. Non-termination is no longer an issue with type codes. To parse a type code we no longer need to parse existential types, which prevents the recursive call to *readType*. This machinery enables the definition of *readType* as a generic program. This machinery can be reused for other generic producers, for example, see the definition of *enumerateType* in Section 4.

3.5 Equality of type representations

In this section, we have introduced machinery to handle type representations generically, namely type codes and conversion functions between type codes and type representations. In Section 4, we show an advanced GADT example that requires a last piece of machinery: equality on type representations.

Below we show a function which compares two type representations, if the two representations are equal, it returns a proof that the two values represent the same type. First, we introduce the type of type equalities:

```
data TEq :: * → * → * where
  Refl :: TEq a a
```

A value of type TEq a b can be used to convince the type checker that two types a and b are the same at compile time. Since two type representations may not be the same, function *teq* returns the result in a monad:

```
teq :: Monad m ⇒ Type a → Type b → m (TEq a b)
teq Int         Int         = return Refl
teq (Maybe a)   (Maybe b)   = liftM  cong1 (teq a b)
teq (List a)    (List b)    = liftM  cong1 (teq a b)
teq (Either a c) (Either b d) = liftM2 cong2 (teq a b) (teq c d)
teq (Lam a c)   (Lam b d)   = liftM2 cong2 (teq a b) (teq c d)
teq (a :→ c)    (b :→ d)    = liftM2 cong2 (teq a b) (teq c d)
teq _           _           = fail "Different reprs"
```

Here, we use *liftM* and *liftM2* to turn congruence functions into functions on monads. Congruence functions are used to lift equality proofs of types to arbitrary type constructors. These are defined as follows:

```
cong1 :: TEq a b → TEq (f a) (f b)
cong1   Refl      = Refl

cong2 :: TEq a b → TEq c d → TEq (f a c) (f b d)
cong2   Refl       Refl      = Refl
```

3.6 Type codes and dependently typed programming

In the literature of generic programming based on dependent types, sets of types having common structure are modelled by universes (Benke et al. 2003). Values known as universe codes describe type structure and an interpretation function makes the relationship between codes and types explicit.

The generic programming approach that this chapter describes would greatly benefit from the use of dependent types. Our approach is slightly redundant due to the necessity of both type representations and type codes. If we were to revise our approach to use dependent types, the generic machinery would be based on type codes only. Previously, the type representation datatype described the relationship between types and the values that represent them. Using dependent types, this relationship would be defined by interpretation on codes and therefore type representations would not be necessary. Furthermore, producers like *readType* would no longer need to generate type representations. It follows that it would not be possible to accidentally define a non-terminating variant of such producers.

3.7 On the partiality of parsing typed syntax trees

Parsing is necessarily a partial operation. A parser for lists will fail to produce a value if the string to parse is not the textual representation of a list. Generic read is also a partial operation: the constructor names to be recognized in the input depend on the type representation argument of *gread*.

Generalized algebraic datatypes make the behavior of *gread* more interesting. When a GADT is used, the set of constructors to recognize in the input will, in general, be a subset of all constructors in the GADT. For example, *gread* (*Expr Int*) parses all constructors with target type Expr Int but it fails to recognize the constructors *EFalse* and *ENot*. Note that this behavior is closely related to type-checking: what would be type-checking errors in a different context are presented here as parsing errors.

A more interesting case is that of expression application. In this case, a representation for the function argument type is parsed and it steers the parsing of the function and the argument expressions. In this case, a type incompatibility between function and argument would be revealed as a parsing error.

4. Application: enumeration applied to simply typed lambda calculus

Generalized algebraic datatypes can encode sophisticated invariants using type-level constraints. We can combine such precise datatypes with generic producer functions, to generate values that have interesting properties. The example of this section combines a datatype representing terms of the simply typed lambda calculus with a generic function that enumerates all the values of a datatype. Using this function we can, for example, generate the terms that have the type of function composition.

4.1 Representing the simply typed lambda calculus

Terms of the simply typed lambda calculus can be represented as follows:

```
data Lam :: ∗ → ∗ → ∗ where
  Vz  :: Lam a (EnvCons a e)
  Vs  :: Lam a e → Lam a (EnvCons b e)
  Abs :: Lam b (EnvCons a e) → Lam (a → b) e
  App :: Type a → Lam (a → b) e → Lam a e → Lam b e
```

The datatype Lam can be read as the typing relation for the simply typed lambda calculus. A value of type Lam a e represents the typing derivation for a term of type a in an environment e. Environments are encoded by list-like type constructors:

```
data EnvCons a e
data EnvNil
```

Each Lam constructor is a rule of the typing relation. The first constructor (*Vz*) represents a variable occurrence of type a, which refers to the first position of the environment (EnvCons a e). We can build a variable occurrence that refers to a deeper environment position by means of the weakening constructor *Vs*. Lambda abstractions are typed by means of the *Abs* constructor. In this case, a b-expression that is typeable in an environment containing a in the first position can be turned into a lambda abstraction of type a → b. The application constructor is almost like application in our previous example, *EApp*, except that *App* includes a representation for the existential type.

The spine representation for this datatype can be defined as follows:

```
toSpine (Lam a e) Vz = convz (Con Vz)
toSpine (Lam a (EnvCons b e)) (Vs tm) =
  convs (Con Vs :◇: tm :▷: Lam a e)
toSpine (Lam (a :→ b) e) (Abs tm) = conabs (Con Abs :◇: body)
  where body = tm :▷: Lam b (EnvCons a e)
toSpine (Lam b e) (App a tm1 tm2) = conapp (Ex a app)
  where app a = Con (App a) :◇: tm1 :▷: Lam (a :→ b) e :◇: tm2
                  :▷: Lam a e
```

The type representations are pattern matched in the *Vs* and *Abs* constructors to build the representation in the right hand side. The *App* constructor has an existential type, therefore we use *Ex* in the spine representation. Using the type representation, we can now print lambda terms.

For producer functions, we define the type spine view on Lam as follows:

```
typeSpine (Lam a e) = concat
  [[convz  (Sig Vz) | EnvCons a' e' ← [e], Refl ← teq a a']
  ,[convs  (Sig Vs :⊕: Lam a e') | EnvCons b e' ← [e]]
  ,[conabs (Sig Abs :⊕: Lam b (EnvCons a' e)) | a' :→ b ← [a]]
  ,[conapp
      (AllEx
          (λb → Sig (App b) :⊕: Lam (b :→ a) e :⊕: Lam b e))]
  ]
```

We test whether a constructor signature has the desired target type by performing pattern matching on type representations. The cases *Vz* and *Vs* are only usable if the environment type argument is not empty. Additionally, the target type of *Vz* requires the equality of the type and the first position in the environment. Therefore, the *Vz* case invokes type equality (*teq*) on the term type (*a*) and the type of the first environment position (*a'*). The abstraction constructor

(*Abs*) requires an arrow type, which is checked by pattern matching against an arrow type representation. The application constructor can always be used, because there is no restriction on the target type of *App*.

This type spine representation is more informative and larger than previous examples. The reason is that the GADT type argument is more complex because of the use of type-level environments. Furthermore, the type constraint in *Vz* requires the use of type equality (*teq*). Fortunately, it is possible to generate the type spine representation by induction on the syntax of the datatype declaration. It would be possible to automate this process using external tools such as DrIFT and Template Haskell if these tools supported GADTs.

4.2 Breadth first search combinators

The generic enumeration function generates all possible values of a datatype in breadth first search (BFS) order. The order used in the search corresponds to the search cost of terms generated. The type BFS is used for the results of a breadth first search procedure:

$$\textbf{type}\ \mathsf{BFS\ a} = [[\mathsf{a}]]$$

The type BFS represents a list of multisets sorted by cost. The first multiset contains terms of cost zero, the second contains terms of cost one and so on. Using this datatype, a consumer can inspect the terms up to a certain cost bound and hence the search does not continue if further terms are not demanded. This is useful because the enumeration function returns a potentially infinite list of multisets.

Multiple BFS values can be zipped together by concatenating multisets having terms of equal cost:

$$\begin{array}{l}
zip_{bfs} :: [\mathsf{BFS\ a}] \to \mathsf{BFS\ a} \\
zip_{bfs}\ [\,]\ \ = [\,] \\
zip_{bfs}\ xss = \textbf{if}\ all\ null\ xss\ \textbf{then}\ [\,]\ \textbf{else} \\
\qquad\qquad concatMap\ head\ xss' : zip_{bfs}\ (map\ tail\ xss') \\
\quad \textbf{where}\ xss' = filter\ (\neg \circ null)\ xss
\end{array}$$

It is more convenient to manipulate BFS results using monadic notation. Therefore, we define return and bind on BFS:

$$\begin{array}{l}
return_{bfs}\ x = [[x]] \\
(\gg\!=_{bfs}) :: \forall \mathsf{a\ b}.\mathsf{BFS\ a} \to (\mathsf{a} \to \mathsf{BFS\ b}) \to \mathsf{BFS\ b} \\
(\gg\!=_{bfs})\ xss\ f = \\
\quad foldr\ (\lambda xs\ xss \to zip_{bfs}\ (map\ f\ xs +\!\!+ [[\,]:xss]))\ [\,]\ xss
\end{array}$$

Return creates a search result that contains a value of cost zero. Bind feeds the terms found in a search *xss* to a search procedure *f*. The cost of the term passed to *f* is added to the costs of that search procedure. Consider, for example, the search results *aSearch* consisting of the terms $\lambda x\ y \to y$ and $\lambda x\ y \to x$ with costs three and four respectively; and a search procedure that produces a term of cost one by adding an abstraction to its argument:

$$\begin{array}{lll}
aSearch & & = [[\,],[\,],[\,],[Abs\ (Abs\ Vz)],[Abs\ (Abs\ (Vs\ Vz))]] \\
f & tm & = [[\,],[Abs\ tm]]
\end{array}$$

Then, the expression $(aSearch \gg\!=_{bfs} f)$ evaluates to the following:

$$[[\,],[\,],[\,],[\,],[Abs\ (Abs\ (Abs\ Vz))],[Abs\ (Abs\ (Abs\ (Vs\ Vz)))]]$$

The two terms in the initial search result now have an additional abstraction argument and have costs of four and five respectively.

The cost addition property of bind can be stated more formally as follows:

$$\begin{array}{l}
propBind :: \mathsf{BFS\ a} \to (\mathsf{a} \to \mathsf{BFS\ b}) \to \mathsf{Bool} \\
propBind\ xss\ f = all\ (all\ costBind)\ (costs\ xss\ f) \\
\quad \textbf{where}\ costBind\ (c,(c_{xss},c_f)) = c \equiv c_{xss} + c_f \\
costs :: \mathsf{BFS\ a} \to (\mathsf{a} \to \mathsf{BFS\ b}) \to \mathsf{BFS\ (Int,(Int,Int))} \\
costs\ xss\ f = cost\ (cost\ xss\ \ \gg\!=_{bfs} \lambda(c_{xss},x) \to \\
\qquad\qquad\qquad\quad cost\ (f\ x) \gg\!=_{bfs} \lambda(c_f,y) \quad \to \\
\qquad\qquad\qquad\quad return_{bfs}\ (c_{xss},c_f))
\end{array}$$

where *cost* annotates each BFS result value with its cost:

$$\begin{array}{l}
cost :: \mathsf{BFS\ a} \to \mathsf{BFS\ (Int,a)} \\
cost = zipWith\ (\lambda sz \to map\ ((,)\ sz))\ [0\,..]
\end{array}$$

Additionally we use a function that increases the cost of the values found in a search procedure:

$$\begin{array}{l}
spend :: \mathsf{Int} \to \mathsf{BFS\ a} \to \mathsf{BFS\ a} \\
spend\ n = (!!n) \circ iterate\ ([\,]:)
\end{array}$$

When using a very expensive search procedure, it is useful to increase the cost of terms exponentially:

$$\begin{array}{l}
raise :: \mathsf{Int} \to \mathsf{BFS\ a} \to \mathsf{BFS\ a} \\
raise\ base\ xss = traverse\ 0\ xss\ \textbf{where} \\
\quad traverse\ \_\quad\ [\,] \qquad\quad = [\,] \\
\quad traverse\ 0\quad (xs:xss) = [\,]:xs:traverse\ 1\ xss \\
\quad traverse\ exp\ (xs:xss) = spend\ (base^{exp} - base^{exp-1} - 1) \\
\qquad\qquad\qquad\qquad\qquad\qquad (xs:traverse\ (exp+1)\ xss)
\end{array}$$

For example, *spend* 2 *aSearch* and *raise* 2 *aSearch* evaluate to:

$$[[\,],[\,],[\,],[\,],[\,],[Abs\ (Abs\ Vz)],[Abs\ (Abs\ (Vs\ Vz))]]$$

and

$$\begin{array}{l}
[[\,],[\,],[\,],[\,],[\,],[\,],[\,],[\,],[Abs\ (Abs\ Vz)], \\
\ [\,],[\,],[\,],[\,],[\,],[\,],[\,],[Abs\ (Abs\ (Vs\ Vz))]]
\end{array}$$

respectively.

4.3 Generic enumeration

The generic enumeration function returns values of a datatype, classified by cost in increasing order. The cost of a term is the number of datatype constructors used therein (constructors used in type representations are an exception and we discuss them last in the definition of *enumerateType*).

$$\begin{array}{l}
enumerate :: \mathsf{Type\ a} \to \mathsf{BFS\ a} \\
enumerate\ a = zip_{bfs}\ [genumerate\ (conVal\ s) \mid s \leftarrow typeSpine\ a]
\end{array}$$

At the top-level, function *genumerate* is invoked on each constructor signature and the resulting search results are zipped together.

The first case of *genumerate* returns the constructor value as the search result assigning it a cost of one. The second case performs search recursively on the function and argument parts and combines the results using BFS monadic application $ap_{bfs} :: \mathsf{BFS\ (a \to b)} \to \mathsf{BFS\ a} \to \mathsf{BFS\ b}$.

$$\begin{array}{l}
genumerate :: \mathsf{Signature\ a} \ \ \to \mathsf{BFS\ a} \\
genumerate\quad (Sig\ c) \qquad\quad = spend\ 1\ (return_{bfs}\ c) \\
genumerate\quad (fun :\oplus: arg) = \\
\quad genumerate\ fun\ `ap_{bfs}`\ enumerate\ arg
\end{array}$$

The third case deals with existential types and hence in our particular application it deals with expression application. This case first enumerates all possible types, and then constructs a constructor signature using *f*, for each type, and enumeration is called recursively:

$$\begin{array}{l}
genumerate\ (AllEx\ f) \quad = \quad enumerateType \\
\qquad\qquad\qquad\qquad\qquad \gg\!=_{bfs} genumerate \circ applyBType\ f
\end{array}$$

As usual with producer functions, *enumerateType* returns a boxed representation. The enumeration of types is performed on type codes, which *interpretTCode* converts to boxed representations.

```
enumerateType :: BFS BType
enumerateType = raise 4 (enumerate TCode)
                    >>=_bfs return_bfs ∘ interpretTCode
```

For the examples in this paper, we are not interested in values that have very complex existential types. Therefore, we keep their size small by assigning an exponential cost to existentials. This also has the effect of reducing the search space, which makes the generation of interesting terms within small cost upperbounds more likely.

4.4 Term enumeration in action

For convenience, we define a wrapper function to perform enumeration of lambda terms:

```
enumerateLam :: Type a → Int → BFS (Lam a EnvNil)
enumerateLam a cost = take (cost + 1)
                           (enumerate (Lam a EnvNil))
```

Our term datatype can perfectly deal with open terms. But the user interface becomes simpler if only closed terms are provided. Therefore, the wrapper function only generates closed lambda terms.

A direct invocation of the enumeration function will result in an attempt to generate an infinite number of terms. For convenience, our wrapper function takes a cost upperbound that limits the cost of terms that are reported. Because of lazy evaluation, the search procedure stops when all terms within the cost bound are reported. The user may choose to increase the cost upperbound in subsequent invocations if the desired term is not found.

The language that the Lam datatype represents is very simple. There are no datatypes, recursion, and arithmetic operations. For example, we cannot expect the enumeration function to generate the successor or predecessor functions for naturals if functions of the type Int → Int are requested. In principle, it is not difficult to extend the language by adding the appropriate constants to Lam. For example, we could add naturals and arithmetic operations on them. We could also add list constructors and elimination functions and even recursion operators such as catamorphisms and paramorphisms.

However, we can keep our language simple and still generate many interesting terms. We focus our attention to parametrically polymorphic functions. Although we do not model parametric polymorphism explicitly in Lam, such functions are naturally generated when the requested type is an instance of the polymorphic type. For instance, a request with type Int → Int generates the identity function.

To make the intent of generating polymorphic functions more explicit, we define a few types that are uninhabited in the Lam language. These types play the role of type variables in polymorphic type signatures:

```
data A
data B
data C
data D
```

Of course, we also introduce the corresponding type representation constructors:

```
data Type :: * → * where
  ...
  A :: Type A
  B :: Type B
  C :: Type C
  D :: Type D
```

In our first example, we generate the code for the identity function. The type of the identity function is $\forall a.a \to a$, which in our notation translates to $A :\to A$. The function we expect to generate is $\lambda x \to x$, which in Lam is written as *Abs Vz*. This term consists of two constructors, therefore a cost upperbound of two should suffice to generate it. The application *enumerateLam* $(A :\to A)$ 2 results in:

```
[[], [], [Abs Vz]]
```

It is instructive to sketch the search procedure as it looks for the identity function. First, *enumerate* is called on the identity type with a closed environment. This function calls *genumerate* on all constructor signatures that match the desired type. The two variable cases *Vz* and *Vs* are not considered, because they cannot be used with an empty environment. Application can always be used but recall that it requires an existential type representation, so the cost is at least 5, which is more expensive than the function that we are looking for. The abstraction case matches the identity type so enumeration is called recursively to generate the abstraction body. Now, a term of type A is requested with a type A in the first position in the environment. The case *Vz* matches perfectly with this request so the term *Abs Vz* is returned with cost two.

There are infinitely many lambda calculus terms of a given type when that type is inhabited. A simple way to way to obtain a new term is by creating a redex that reduces to the term that we currently have. For example, we can obtain a new identity function by adding a redex in the function body: $\lambda x \to (\lambda x \to x)\ x$. Can this term be found by our enumeration function? Yes, provided that we increase the cost upperbound to include that of our new term. The new term is essentially two identity functions plus an application constructor, which makes a cost upperbound of nine. We evaluate *enumerateLam* $(A :\to A)$ 9 which yields:

```
[[], [], [Abs Vz], [], [], [], [], [], [], [Abs (App A (Abs Vz) Vz)]]
```

This example shows that the search space is somewhat redundant. A way to speed up term search would be to avoid the generation of redundant terms by adding constraints to Lam. For example, we could avoid redeces by preventing the generation of abstractions in the left part of applications.

Another interesting example is the generation of the application function. This function has type ∀a b.(a → b) → a → b, which in our notation is written $((A :\to B) :\to A :\to B)$. We evaluate *enumerateLam* $((A :\to B) :\to A :\to B)$ 10 to generate an application function, which results in:

```
[[], [], [Abs Vz], [], [], [], [], [], [], [],
 [Abs (Abs (App A (Vs Vz) Vz))]]
```

These are the encodings for the functions $\lambda x \to x$ and $\lambda x\ y \to x\ y$. The careful reader may wonder why the other identity term $\lambda x \to (\lambda x \to x)\ x$, which has cost 8 in the previous example, is not generated. The answer is that the cost of the term includes that of the type representation used in the application constructor. Since this example has a different type, the type representation would be $A :\to B$ rather than A. It follows that the term $\lambda x \to (\lambda x \to x)\ x$ is not generated because it has a cost of 14.

Our last example is function composition. The type of this function is ∀a b c.(b → c) → (a → b) → a → c. To generate composition, we evaluate

```
last (enumerateLam ((B :→ C) :→ (A :→ B) :→ (A :→ C)) 19)
```

which yields to the encoding of $\lambda x\ y\ z \to x\ (y\ z)$:

```
[Abs (Abs (Abs (App B (Vs (Vs Vz)) (App A (Vs Vz) Vz))))]
```

5. Related work

To the best of our knowledge, only the spine approach (Hinze et al. 2006; Hinze and Löh 2006) enables generic programming on generalized algebraic datatypes in Haskell. This is the approach on which the work in this chapter is based. Because both the spine and the type spine view can encode GADTs, both consumer and producer functions can be defined on such datatypes. Interestingly, to properly support GADTs for producer functions, the approach should also support existential types. For example, when reading a GADT from disk, we may want the GADT type argument to be dynamically determined from the disk contents. Therefore, we would existentially quantify over that argument. However, the spine approach supports existential types for consumers but not for producers.

Generalized algebraic datatypes are inspired by inductive families in the dependent types community. We are aware of two approaches (Benke et al. 2003; Morris 2007) that support definitions by induction on the structure of inductive families. Both approaches make essential use of evaluation on the type level to express the constraints over inductive families. Examples of type families on which generic programming is applied include trees (indexed by their lower and upper size bounds), finite sets, vectors and telescopes. Neither approach gives examples for the support of existential types so it is not clear whether these are supported.

Weirich (2002) proposes a language that provides a construct to perform runtime case analysis on types. In order to support universal and existential quantification, the language includes analyzable type constants for both quantifiers. This approach supports the definition of consumers and producers. Moreover, if the language is extended with polymorphic kinds it supports quantification over arbitrarily kinded types.

Our approach to defining breadth-first search combinators is not novel. Spivey (2000) defines a set of breadth-first search combinators such as monadic join and composition, and proves desirable properties for them. There are many similarities between our work and that of Spivey. It would be interesting to see whether our combinators satisfy the same properties as the combinators proposed by Spivey.

Koopman and Plasmeijer (2007) generate lambda calculus terms by performing systematic enumeration based on a grammar. To reduce the size of the search space, the grammar has syntactic restrictions such as that the applications of certain operands are always saturated, and recursive calls are always guarded by a conditional. The candidate terms are then reported to the user based on whether they satisfy an input-output specification, which is established by evaluation.

Djinn (Augustsson 2005) generates lambda calculus terms based on a user-supplied type. This tool implements the decision procedure for intuitionistic propositional calculus due to Dyckhoff (1992). Similarly, the work of Katayama (2005) makes use of a type inference monad to generate well-typed terms. Later, the candidate terms are evaluated and checked against an input-output specification. As in our approach, Djinn and the approach of Katayama generate only well-typed terms so there is no need for a type checking phase to discard ill-typed terms.

The main difference between the work of Koopman and Plasmeijer (2007) and ours is that our generator is typed-based. It follows that our generator never returns ill-typed terms because the search space is reduced by means of type-level constraints in the GADT. Generating ill-typed terms has advantages. For example, Koopman's approach can generate the Y-combinator. On the other hand, ill-typed terms are usually not desirable, so these have to be discarded through either evaluation (Koopman), which slows down the generation algorithm. In Koopman's work the generation of ill-typed terms is prevented to some extent by the syntactic constraints imposed on the grammar.

A type-based generator, such as Djinn, Katayama's generator and our approach, is able to synthesize polymorphic functions without the need for input-output specifications. Koopman's work, however, cannot generate polymorphic functions based solely on type information.

Djinn supports user-defined dataypes. Katayama and Koopman's generators are able to generate recursive programs. Our approach currently generates programs for a rather spartan language. However, it should be possible to add introduction and elimination constants for (recursive) datatypes, and recursion operators such as catamorphisms and paramorphisms.

Both Djinn and our approach enumerate terms guided by type information. However, the two approaches are very different. Djinn has a carefully crafted algorithm that handles the application of functional values in such a way that it is not necessary to exhaustively enumerate the infinite search space. As a consequence, Djinn is able to detect that a type is uninhabited in finite time. In contrast, our approach produces function applications by means of exhaustive enumeration. First, all the possible types of an argument are enumerated, and, for each of them, function and argument terms are enumerated to construct an application. The good side of an exhaustive approach like ours is that it can generate all possible terms of a given type. For example, it can generate all Church numerals, whereas Djinn only generates those corresponding to zero and one. On the bad side, if unbounded, our approach does not terminate when trying to generate a term for an uninhabited type.

We have not performed a careful performance comparison but we believe that our generator may be the slowest of the approaches considered here. Probably the main culprit for inefficiency is the implementation of the existential case. Currently this case enumerates all possible types, even if no applications for that argument type can be constructed. Ill-typed terms are never generated, but resources are nevertheless consumed when attempting to enumerate terms having possibly uninhabited types. It is difficult to make the algorithm smarter about generating types because, being generic, it does not make assumptions about the particularities of lambda calculus. On the other hand, it is possible to reduce the search space by adjusting the definition of Lam. For example, we could forbid the formation of redeces to avoid redundancy of terms, or even adopt the syntactic restrictions used in Koopman's work.

While our approach may be less efficient, it has the virtue of simplicity: the core of the generation algorithm consists of roughly a dozen lines of code and there is no need for an evaluation or a type checking phase. Furthermore, it has the advantage of an elegant separation between the grammar constraints and the formulation of the enumeration algorithm. This allows us to use the enumeration function to generate other languages, whereas the other generators are specific to lambda calculus.

6. Conclusions

We have presented an extension of the spine approach to generic programming, which supports the definition of generic producers for existential types. This extension allows the definition of, for example, generic read for datatypes that use existential quantification.

Our approach opens the way for a new application of generic programming. By taking the standard enumeration generic function and extending it with a case for existentials, we obtain a function that enumerates well-typed terms. For example, we can instantiate enumeration to the GADT that represents terms of the simply typed lambda calculus and use the resulting function to search for terms that have a given type. Such an application was not previously possible because producers that handle existential types could not be generically defined.

Acknowledgments

We are grateful to Andres Löh for the productive discussions that inspired this paper. The ST reading club at Utrecht University gave detailed feedback on an earlier version of this paper. We thank the anonymous reviewers for the useful suggestions and comments. We also thank Lambert Meertens, the careful reader who motivated us to improve the explanations in Section 4.4.

References

Lennart Augustsson. Announcing Djinn, version 2004-12-11, a coding wizard. Available from `http://permalink.gmane.org/gmane.comp.lang.haskell.general/12747`, 2005.

Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10(4):265–289, 2003.

James Cheney and Ralf Hinze. First-class phantom types. Technical report, Cornell University, 2003.

Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2000*, pages 286–279, 2000.

R Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *The Journal of Symbolic Logic*, 57(3):795–807, 1992.

Ralf Hinze and Andres Löh. "Scrap Your Boilerplate" revolutions. In *Proceedings of the 8th International Conference on Mathematics of Program Construction, MPC'06*, LNCS 4014, pages 180–208, 2006.

Ralf Hinze, Andres Löh, and Bruno C. d. S. Oliveira. "Scrap Your Boilerplate" reloaded. In Philip Wadler and Masimi Hagiya, editors, *FLOPS'06*, LNCS 3945, 2006.

Susumu Katayama. Systematic search for lambda expressions. In M. van Eekelen, editor, *6th Symposium on Trends in Functional Programming, TFP 2005*. Institute of Cybernetics, Tallinn, 2005. ISBN 9985-894-88-X.

Pieter Koopman and Rinus Plasmeijer. Systematic synthesis of λ-terms. In *Essays dedicated to Henk Barendregt on the Occasion of his 60th Birthday*, 2007.

Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. Gast: Generic automated software testing. In R. Peña and T. Arts, editors, *IFL'02*, volume 2670 of *LNCS*, 2003.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 26–37, 2003.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(01):1–13, 2007. doi: 10.1017/S0956796807006326.

Peter Morris. *Constructing Universes for Generic Programming*. PhD thesis, The University of Nottingham, 2007.

S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP'06*, pages 50–61, 2006.

Michael Spivey. Combinators for breadth-first search. *Journal of Functional Programming*, 10(4):397–408, 2000. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S0956796800003749.

S. D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, volume 1129 of *LNCS-Tutorial*, pages 184–207. Springer-Verlag, 1996.

Stephanie Weirich. Higher-order intensional type analysis. In *In Proc. 11th ESOP, LNCS 2305*, pages 98–114. Springer-Verlag, 2002.

Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL'03*, pages 224–235, New Orleans, January 2003.