The Representation of Constraints, Annotations and First Class Patterns over Arbitrary Data Types in Haskell

Michael D. Adams
mdadams@ittc.ku.edu

May 6, 2004

Abstract

This report describes a system that was originally written to describe static semantic constraints over an abstract syntax tree. In the process several modules were developed that are of independent interest. These include the ability to attach annotations to a pre-existing data type, and writing Haskell style patterns as a first class object.

This report begins by describing the framework for annotations and the use of a modified version of the *Generics* library to manipulate those annotations. A system for pre-calculating the symbol table for every node in an abstract syntax tree is developed. How annotations can be used to describe first class patterns and performing matching between patterns and the symbol table annotations is shown. Finally, all these are used together to develop a domain specific embedded language for describing static semantic constraints.

Contents

Chapter 1

Introduction

In many programming languages there are restrictions on what makes a well formed program. Some of these restrictions fall under the category of type checking but others do not. For example a common check in hardware description languages is to forbid connecting together two output ports. In object-oriented languages the inheritance hierarchy should never contain a class that directly or indirectly inherits from itself. These constraints can be checked statically at compile time and thus are called static semantic constraints. This paper describes a mechanism for describing these constraints by making use of first class patterns and by attaching annotations to Haskell data types.

Often static semantic constraints are conceptually simple assertions that should hold true everywhere in a program. So, in principle, describing the static semantic constraints of a language should not be a hard task. Compilers generally generate an abstract syntax tree during the parsing a source file. Therefore, one could simply write a predicate that operates on a single node of the abstract syntax tree and that determines whether the node satisfies the constraint. Then pass that predicate to a tree walker that traverses the abstract syntax tree and tests the predicate wherever it is applicable.

This approach works well if the predicate depends only on the node which it is directly operating on, but most languages have some concept of scope and thus require a symbol table. A slight modification of the tree walker sufficiently handle this. Namely, the tree walker must update the symbol table as it traverses the tree and pass the current symbol table to the predicate as an additional argument. This task is simplified by the Boilerplate paper [1] that introduces the *Generics* library. It provides a tree walker that implements the aforementioned functionality and thereby eliminates the need to write the "boilerplate" code that typically makes up such a tree walker.

This solution works if the symbol table does not need to change once the predicate begins evaluation. However that is not always possible. Suppose we wish to check for cyclic inheritance in a language with top-level namespaces. A superclass may be in a different namespace than the current class. Thus the symbol table will need to be updated during the evaluation of the predicate.

Two possible solutions are apparent. The first is to require the predicate to have knowledge of how to update the symbol table and when necessary do so manually. The second would have the predicate call the tree walker and have the tree walker update the symbol table before returning control back to the predicate for further processing. Neither of these is completely satisfactory. The former violates separation of concerns. The latter introduces a complex interdependency between the predicate and the tree walker and needs a method for the predicate to tell the tree walker to retrieve the symbol table for a specific portion of the tree.

Fortunately, there is a third option. Each node could be annotated with a symbol table in advance. Then the predicate can extract that symbol table from a node when it needs it. Enriching the value space in this manner is a common practice in the Haskell community and is accomplished with the use of monads [8] that wrap around the type and store the extra information. However, in this case monads are not enough. A monad would only wrap around the top of a tree. We require something that wraps around each node of an abstract syntax tree.

The *Annotated* type described in this paper provides the ability to wrap not just the top level of a node but also its descendants. The basic idea is that when a data value is built the constructor is not applied to its arguments as it normally would. Instead the constructor and its arguments are stored in an *Annotated* along with the extra information to be stored at that node. Later the stored constructor may be applied to its arguments. Performing that application will project to a value that is no longer an *Annotated* so it can be used with any function that was written to operate on the original type. Of course, because that result is no longer an *Annotated* the extra information is not available anymore. So projecting to the original type should be done only when the extra information is no longer needed or has already been extracted.

Since the user needs to keep the *Annotated* as an *Annotated* for as long as possible, some means for manipulating an *Annotated* must be provided. It is unreasonable to require direct manipulation as the user is typically interested in the value that the *Annotated* represents and not in the way the *Annotated* represents the value. *Generics* provides a model of manipulation that is very close to what is needed. It cannot be used directly, however, because the functions it defines do not have the right type signatures. Those functions would manipulate the actual structure of the *Annotated* instead of the virtual, represented structure. A slight modification of the *Generics* library solves this problem. The modified version developed in this paper, *MetaGenerics*, operates on the virtual structure of some type a that is represented by the actual structure, $t\ a$. By using *Annotated* m for the type t parameter, *MetaGenerics* allows the manipulation of an *Annotated* without worrying about the details of its representation.

Unfortunately pattern matching, one of the more powerful features of Haskell, is not useable with an *Annotated*. This occurs for the same reason that *Generics* was not usable with *Annotated*. Pattern matching would be operating on the actual *Annotated* structure and not the represented structure. Furthermore patterns are not first class objects in Haskell. They can not be transformed or modified programmatically. Thus it is not possible to write a function that transforms a Haskell pattern into a pattern for the *Annotated* version of a type.

The absence of pattern matching poses a problem because pattern matching significantly aids writing constraint predicates. Fortunately, the *Annotated* type solves the very problem it is causing. It is possible to represent patterns as first class objects using *Annotated*. The key observation is that *Annotated* could wrap each node inside a *Maybe*. The representation of a pattern then simply uses *Nothing* for a wild-card and *Just* for literal a portion. Pattern matching can then be accomplished through *MetaGenerics* and the use of constructor comparison. Constructors are functions and cannot be directly compared. By passing $\perp$ to a constructor for each of its argument and then calling *toConstr*, a *Constr* may be obtained, and it is possible to compare one *Constr* to another. *Annotated* makes feasible the representation of patterns as first class objects and pattern matching over those patterns.

The use of *Annotated* to store symbol table information and first class patterns are combined in a domain specific embedded language for describing constraints. As an added benefit, first class patterns enable this embedded language is not limited to using patterns in the syntactic locations that limit Haskell patterns. This turns out to be quite helpful when writing constraints that test only one constructor or constructor form of a data type. Using Haskell patterns would require an extra pattern clause to handle the cases that are not being tested by the constraint. By using first class patterns the embedded language can automatically handle those cases.

Chapter 2

Annotations

```
{-# OPTIONS -fallow-undecidable-instances #-}
module FirstClassPatterns.Annotated (
        Annotated,
        annotatedExpand,
        annotatedCollapse,
        annotatedUndefineds,
        annotatedLift,
        annotatedCtor
) where

import Control.Monad
import Data.Generics
import Data.Dynamic
import FirstClassPatterns.MetaGenerics
```

2.1 Attaching Annotations to Data

The type *Annotated m b* represents the abstract notion of wrapping every constructor in an algebraic data type, *b*, inside a monad, *m*. It uses *Ctor* to represent constructors and :$: to represent constructor application.

```
data Annotated m b
        = Ctor (m b)
        | forall a ∘ (Data a) ⇒
                Annotated m (a → b) :$: Annotated m a
```

An example will help clarify things. Suppose *foo* is of type *Maybe* [*Int*] and has a value *Just* [1, 2]. Equivalently *foo* may be represented as *Just* ((:) 1 ((:) 2 [])). If we make the constructor applications explicit we get *Just* $ ((:) $ 1 $ ((:) $ 2 $ [])). Injecting *foo* into an *Annotated m* (*Maybe* [*Int*]) can be done by replacing $ with :$: and wrapping every constructor with *Ctor*.*return*. This yields:

```
annotatedFoo :: (Monad m) ⇒ Annotated m (Maybe [Int])
annotatedFoo =
        (Ctor (return Just):$:
                (Ctor (return (:)):$:
                        (Ctor (return 1)):$:
                        ((Ctor (return (:))):$:
                                (Ctor (return 2)):$:
                                (Ctor (return [])))))
```

Notice how declaring :\$: over an existential type ensures everything is typed correctly. Consider just the part corresponding to $(:)\ 2\ [\,]$. We want it to have a type of *Annotated m* $[Int]$. The type of *Ctor* (*return* $(:)$) is *Annotated m* $(Int \to [Int] \to [Int])$ and the type of *Ctor* (*return* 2) is *Annotated m Int*. The combination of these two using :\$: has type *Annotated m* $([Int] \to [Int])$. The next constructor is *Ctor* (*return* $[\,]$) and has type *Annotated m* $[Int]$. Using :\$: to combine this with the previous result, makes a value with a final type of *Annotated m* $[Int]$ which is what was desired. Following a similar line of reasoning the final type of *annotatedFoo* can be shown to be *Annotated m* (*Maybe* $[Int]$).

For the moment it may seem like we have added needless complexity by changing the simple expression *Just* $[1, 2]$ into an obfuscated monstrosity. However, the value space has been enriched due to the presence of the monad around each constructor. Consider what happens if for m we use the type:

```
data Weight b = Weight Int b

instance Monad Weight where
        return = Weight 0
        (Weight w1 x) >>= k =
                case k x of
                        Weight w2 y -> Weight (w1 + w2) y
```

Each constructor now stores extra information about the weight given to that node. This may be done with any monad. For example each node could be tagged with a name or by using the *Maybe* monad entire portions of a value can be marked as not specified.

The case using the *Maybe* monad is of further interest. Suppose we have a variable *partialPair* of type *Annotated Maybe* (*Int*, *Int*). That variable can have the value:

```
partialPairFull :: Annotated Maybe (Int, Int)
partialPairFull = Ctor (Just (,)) :$: (Ctor (Just 1)) :$: (Ctor (Just 2))
```

There is also another possibility. Because *Nothing* has type *Maybe a* for any a. The expression following expression may be used.

```
partialPairEmpty :: Annotated Maybe (Int, Int)
partialPairEmpty = Ctor (Nothing)
```

In addition either or both of the children may be left unspecified, as they are in these expressions.

```
partialPairNoLeft :: Annotated Maybe (Int, Int)
partialPairNoLeft = Ctor (Just (,)) :$: (Ctor Nothing) :$: (Ctor (Just 2))

partialPairNoRight :: Annotated Maybe (Int, Int)
partialPairNoRight = Ctor (Just (,)) :$: (Ctor (Just 2)) :$: (Ctor Nothing)

partialPairNoChildren :: Annotated Maybe (Int, Int)
partialPairNoChildren = Ctor (Just (,)) :$: (Ctor Nothing) :$: (Ctor Nothing)
```

This functionality of being able to omit portions of the tree forms the basis of expressing first class patterns, which will becoverd in more detail when discussing the *Pattern* module.

It should be noted that using an *Annotated* in this way is similar to adding a functor to a fixed-point type. For instance, one may write the type:

```
data TreeF a = Branch a a | Leaf Int
```

This type can be converted to a simple tree type by applying the *Rec* type to take the fixed-point [4, 5].

```
newtype Rec f = In (f (Rec f))
type Tree = Rec TreeF
```

The type of a tree with possibly unspecified children can be expressed by composing *Maybe* with the *TreeF* functor before the fixed-point is taken.

$$\begin{aligned}
&\textbf{newtype}\ \mathit{Comp}\ f\ g\ a = \mathit{Comp}\ (f\ (g\ a))\\
&\textbf{type}\ \mathit{MaybeTreeF} = \mathit{Comp}\ \mathit{Maybe}\ \mathit{TreeF}\\
&\textbf{type}\ \mathit{MaybeTree} = \mathit{Rec}\ \mathit{MaybeTreeF}
\end{aligned}$$

The effect of the above construction is analogous to the earlier *partialPair* example; however there are differences which exist. First, using an *Annotated* does not require that the type be first written as a functor. Any type that is an instance of *Data* can be placed inside an *Annotated*. In GHC, most data types can easily be declared an instance of *Data* by deriving them from *Data* and and the class it depends on, *Typeable* [2].

Secondly, fixed-points are typically only declared with one argument. Therefore all children must be of the same type as that of their parent. Some work has been done with difunctors that operate on two type parameters [5], but over a complex data structure with many different type variables this quickly becomes impractical. Use of *Annotated* does not suffer from this limitation because it can be applied to any type that in an instance of *Data*.

Lastly, in the declaration of *TreeF*, the *Leaf* constructor does not refer to the type parameter a. Instead, it directly refers to an *Int*. Thus, the effects of composing *Maybe* with *TreeF* do not extend into the contents of the *Leaf*. When using an *Annotated*, however, the effects extend into all children. For reasons of symmetry with the fixed-point type, it would be preferable to place the monad around the children of an *Annotated*, in other words the right hand of the :\$: operator. However, such placement would require significant changes to function signatures in the *MetaGeneric* class which in turn would break the symmetry with the *Data* class. As a result, translating functions from the *Generics* library to the *MetaGenerics* library would not be as simple. Instead of this, the the monad is placed in the *Ctor* of an *Annotated* that does not have these same problemrst to get a motivation then we need to switch to MetaGenerics so we can explain mgfoldl, then come back to Annotated to describe the following functions.)

2.2 Higher Level Operations on Annotations

Manipulating an *Annotated* type directly would be quite cumbersome. As the example with *annotatedFoo* showed, even a simple data item becomes quite large when injected to an *Annotated*. Furthermore the existential type inside it would require extreme care when manipulating an *Annotated* to prevent the existential type from escaping [7]. Both of these problems can resolved by expressing manipulations in terms of generic traversals like those found in the *Generics* library [1]. However, the plain *Generics* library will not work because it would attempt to traverse over all parts of the structure in an *Annotated* m a including every *Ctor* and :\$:. Instead of something to traverse the actual structure of *Annotated* m a, we want something to traverse the virtual structure that models the structure of a. The *MetaGenerics* library does precisely this.

$$\begin{aligned}
&\textbf{instance}\ \mathit{MetaGeneric}\ (\mathit{Annotated}\ m)\ \textbf{where}\\
&\qquad \mathit{mgfoldl}\ k\ z\ (f\ {:}\${:}\ \mathit{ma}) =\\
&\qquad\qquad (\mathit{mgfoldl}\ k\ z\ f)\ \text{`}k\text{`}\ \mathit{ma}\\
&\qquad \mathit{mgfoldl}\ _\ z\ x = z\ x\\
&\qquad \mathit{mapply} = ({:}\${:})
\end{aligned}$$

The function *annotatedExpand* injects a data object of type b into *Annotated* m b. This is the most common method to construct an *Annotated* from an existing data object.

$$\begin{aligned}
&\mathit{annotatedExpand} :: (\mathit{Data}\ b, \mathit{Monad}\ m)\\
&\qquad \Rightarrow b \rightarrow \mathit{Annotated}\ m\ b\\
&\mathit{annotatedExpand} = \mathit{gfoldl}\ k\ (\mathit{annotatedCtor} \circ \mathit{return})\\
&\qquad \textbf{where}\\
&\qquad\qquad k\ c\ x = c\ {:}\${:}\ (\mathit{annotatedExpand}\ x)
\end{aligned}$$

Sometimes we don't want to build an *Annotated m b* from a *b* that already exists. Indeed sometimes we can't, as in the case where the monad used is *Maybe* and some of the children don't exist. So, we want to construct an *Annotated m b* a piece at a time. This requires two things: 1) it requires that constructors can be make into an *Annotated*, 2) it requires that constructors can be applied to children.

Although the second requirement is fulfilled by *mapply*, *annotatedExpand* can not fulfill the first requirement because it only operates on instances of *Data*. Constructors belong to the type of functions and do not have a *Data* instance that would behave properly. Consequently we must introduce a new function, *annotatedCtor*, that is intended to only be applied to constructors.

$$
\begin{array}{l}
\mathit{annotatedCtor} :: m\ b \rightarrow \mathit{Annotated}\ m\ b \\
\mathit{annotatedCtor} = \mathit{Ctor}
\end{array}
$$

Using the interface provided with *mapply* and *annotatedCtor*, the previous example of *annotatedFoo* can be rewritten.

$$
\begin{array}{l}
\mathit{annotatedFoo}' :: (\mathit{Monad}\ m) \Rightarrow \mathit{Annotated}\ m\ (\mathit{Maybe}\ [\mathit{Int}]) \\
\mathit{annotatedFoo}' = \\
\qquad (\mathit{annotatedCtor}\ (\mathit{return}\ \mathit{Just})`\mathit{mapply}` \\
\qquad\qquad (\mathit{annotatedCtor}\ (\mathit{return}\ (:))`\mathit{mapply}` \\
\qquad\qquad\qquad (\mathit{annotatedCtor}\ (\mathit{return}\ 1))`\mathit{mapply}` \\
\qquad\qquad\qquad ((\mathit{annotatedCtor}\ (\mathit{return}\ (:)))`\mathit{mapply}` \\
\qquad\qquad\qquad\qquad (\mathit{annotatedCtor}\ (\mathit{return}\ 2))`\mathit{mapply}` \\
\qquad\qquad\qquad\qquad (\mathit{annotatedCtor}\ (\mathit{return}\ [\,])))))
\end{array}
$$

2.3 Additional Operations on Annotations

The functions in *MetaGenerics* plus *annotatedExpand* and *annotatedCtor* provide a solid basis from which to start manipulating *Annotated* objects. However we require a few more functions before we can have a complete set. Each of these functions is greatly simplified by the use of *mgfoldl* from *MetaGenerics*.

The *annotatedCollapse* function projects from *Annotated m b* to *m b* by monadically applying each constructor to its arguments.

$$
\begin{array}{l}
\mathbf{newtype}\ \mathit{COLLAPSE}\ m\ (t :: * \rightarrow *)\ a = \mathit{COLLAPSE}\{\mathit{unCOLLAPSE} :: m\ a\} \\
\mathit{annotatedCollapse} :: (\mathit{Monad}\ m) \Rightarrow \mathit{Annotated}\ m\ b \rightarrow m\ b \\
\mathit{annotatedCollapse}\ x = \mathit{unCOLLAPSE}\ \$\ \mathit{mgfoldl}\ k\ z\ x \\
\qquad \mathbf{where} \\
\qquad\qquad k\ (\mathit{COLLAPSE}\ \mathit{m1})\ \mathit{m2} = \mathit{COLLAPSE}\ \$\ \mathit{m1}\ `\mathit{ap}`\ (\mathit{annotatedCollapse}\ \mathit{m2}) \\
\qquad\qquad z\ (\mathit{Ctor}\ x) = \mathit{COLLAPSE}\ \$\ x
\end{array}
$$

Often times it may be necessary to perform an operation that does not depend on the children of a data object and instead only depends on the constructor. Using *annotatedCollapse* can lead to problems in those cases. Suppose that one wishs to test whether the constructors of two *Annotated Maybe* (*Either Int Char*) are the same and the the data objects happen to be the following.

$$
\begin{array}{l}
\mathit{left} :: \mathit{Annotated}\ \mathit{Maybe}\ (\mathit{Either}\ \mathit{Int}\ \mathit{Char}) \\
\mathit{left} = \mathit{Ctor}\ (\mathit{Just}\ \mathit{Left})\ {:}\$ {:}\ \mathit{Ctor}\ \mathit{Nothing} \\
\\
\mathit{right} :: \mathit{Annotated}\ \mathit{Maybe}\ (\mathit{Either}\ \mathit{Int}\ \mathit{Char}) \\
\mathit{right} = \mathit{Ctor}\ (\mathit{Just}\ \mathit{Right})\ {:}\$ {:}\ \mathit{Ctor}\ \mathit{Nothing}
\end{array}
$$

Calling *annotatedCollapse left* will yield *Nothing*, and so will *annotatedCollapse right*. This gives no information about what value the constructor actually has. Getting around this problem requires the introduction of a new function, *annotatedUndefineds*, that applies the constructor to $\perp$.

$$
\begin{aligned}
&\textbf{newtype}\ \mathit{UNDEF}\ m\ (t :: * \rightarrow *)\ a = \mathit{UNDEF}\{\mathit{unUNDEF} :: m\ a\}\\
&\mathit{annotatedUndefineds} :: (\mathit{Monad}\ m) \Rightarrow \mathit{Annotated}\ m\ b \rightarrow m\ b\\
&\mathit{annotatedUndefineds}\ x = \mathit{unUNDEF}\ \$\ \mathit{mgfoldl}\ k\ z\ x\\
&\qquad\quad \textbf{where}\\
&\qquad\qquad\qquad k\ (\mathit{UNDEF}\ f)\ \_ = \mathit{UNDEF}\ \$\ f\ \text{`}\mathit{ap}\text{`}\ \mathit{return}\ \bot\\
&\qquad\qquad\qquad z\ (\mathit{Ctor}\ x) = \mathit{UNDEF}\ \$\ x
\end{aligned}
$$

The result is safe to use with any function that only depends on the constructors, but it is not safe to access any of its children. This constraint may seem to nullify the usefulness of *annotatedUndefineds*, but in practice this is often the very the function needed. For example, pattern matching depends critically on testing constructor equality without necessarily needing to evaluate the children.

One danger is that when the *Annotated* was originally declared the "constructor" was not a real constructor but a function of the same type and that depended on the value of its arguments such as the following.

$$
\begin{aligned}
&\mathit{fakeCons} :: \mathit{Int} \rightarrow [\mathit{Int}] \rightarrow [\mathit{Int}]\\
&\mathit{fakeCons}\ \_\ (x : \mathit{xs}) = \mathit{xs}\\
&\mathit{fakeCons}\ \_\ [\,] = [\,]\\
\\
&\mathit{fakeAnnotated} :: (\mathit{Monad}\ m) \Rightarrow \mathit{Annotated}\ m\ [\mathit{Int}]\\
&\mathit{fakeAnnotated} = (\mathit{annotatedCtor}\ (\mathit{return}\ \mathit{fakeCons}))\\
&\qquad\quad \text{`}\mathit{mapply}\text{`}(\mathit{annotatedCtor}\ (\mathit{return}\ 1))\\
&\qquad\quad \text{`}\mathit{mapply}\text{`}(\mathit{annotatedCtor}\ (\mathit{return}\ [\,]))
\end{aligned}
$$

The expression *annotatedUndefineds fakeAnnotated* would evaluate to $\bot$ instead of what is expected, namely a constructor with children that evaluate to $\bot$.

An alternative that would avoid this problem and the use of $\bot$ would be to use *Constr* from the *Generics* library. The *annotatedUndefineds* function would return a *Constr*, and *annotatedCtor* would take a *Constr*. On the other hand this would cause a further problem because *Constr* does not have the type information about its arguments that is needed by *Annotated*. The crux of this problem is that there needs to be a way to distinguish between functions and constructors, but also know the type of the constructor's arguments. The former is doable using *Constr*. The latter is doable using construtor functions. Until a way is found to meet both these ends, *Annotated* must remain using a contructor function and *annotatedUndefineds* be used with caution.

The final function defined for manipulating an *Annotated m b* is *annotatedLift*. It allows transformation of the monad that is applied to the constructor.

$$
\begin{aligned}
&\textbf{newtype}\ \mathit{LIFT}\ t\ a = \mathit{LIFT}\{\mathit{unLIFT} :: t\ a\}\\
&\mathit{annotatedLift} ::\\
&\qquad\quad (\mathit{forall}\ a \circ m\ a \rightarrow m\ a)\\
&\qquad\quad \rightarrow \mathit{Annotated}\ m\ b \rightarrow \mathit{Annotated}\ m\ b\\
&\mathit{annotatedLift}\ f\ x = \mathit{unLIFT}\ \$\ \mathit{mgfoldl}\ k\ z\ x\\
&\qquad\quad \textbf{where}\\
&\qquad\qquad\qquad k\ (\mathit{LIFT}\ x)\ y = \mathit{LIFT}\ \$\ \mathit{mapply}\ x\ y\\
&\qquad\qquad\qquad z\ (\mathit{Ctor}\ x) = \mathit{LIFT}\ \$\ \mathit{annotatedCtor}\ (f\ x)
\end{aligned}
$$

2.4 Class Instances for Annotations

The use of a type constructor as a parameter of *Annotated*, namely m, means that *Typeable* can not be derived so the instance is manually declared.

$$
\begin{aligned}
&\textbf{instance}\ (\mathit{Typeable}\ (m\ b))\\
&\qquad\quad \Rightarrow \mathit{Typeable}\ (\mathit{Annotated}\ m\ b)\ \textbf{where}
\end{aligned}
$$

$$
\begin{array}{l}
\mathit{typeOf}\ (\_ :: \mathit{Annotated}\ m\ b) = \mathit{mkAppTy}\ (\mathit{mkTyCon}\ \texttt{"Annotated.Annotated"}) \\
\qquad [\mathit{typeOf}\ (\bot :: m\ b)]
\end{array}
$$

The use of an existential type in *Annotated* means that *Show* can not be derived either, so the instance is also manually declared.

$$
\begin{array}{l}
\mathbf{instance}\ (\mathit{Monad}\ m, \mathit{Show}\ (m\ \mathit{String}), \mathit{Data}\ b) \\
\qquad \Rightarrow \mathit{Show}\ (\mathit{Annotated}\ m\ b)\ \mathbf{where} \\
\qquad \mathit{show} = \mathit{mgeverything}\ k\ z \\
\qquad\qquad \mathbf{where} \\
\qquad\qquad\qquad z\ x = \texttt{"Ctor ("} +\!\!+ \\
\qquad\qquad\qquad\qquad \mathit{show}\ (\mathit{liftM}\ (\mathit{show} \circ \mathit{toConstr})\ (\mathit{annotatedUndefineds}\ x)) +\!\!+ \\
\qquad\qquad\qquad\qquad \texttt{")"} \\
\qquad\qquad\qquad k\ x\ y = x +\!\!+ \texttt{" :\$: ("} +\!\!+ y +\!\!+ \texttt{")"}
\end{array}
$$

Chapter 3

Generalizing Generics to MetaGenerics

```
module FirstClassPatterns.MetaGenerics (
        MetaGeneric (
                mgfoldl,
                mapply
        ),
        MetaGenericT,
        MetaGenericQ,
        MetaGenericM,
        MetaGenericC,
        mgmapT,
        mgmapQ,
        mgmapQl,
        mgmapQr,
        mgmapM,
        mgeverywhere,
        mgeverything,
        mgchildren,
        mtfoldl,
        mtmapQl
) where

import Data.Generics
```

The *Generics* library provides a mechanism for traversing data structures in an automated fashion and dealing with variables of a type that is unknown until runtime [1]. The *MetaGenerics* library is an extension of this concept. The *Generics* library allows one to traverse the structure of a in some type a. *MetaGenerics*, on the other hand, allows one to traverse the structure of a in some type $t\ a$. For example consider *Annotated*, which was the the original motivation behind writing *MetaGenerics*. When manipulating something of type *Annotated* $m\ a$ one is usually interested in the parameter a and not the internals of how *Annotated* represents an a.

```
class MetaGeneric t where
        mgfoldl::
                (forall a b ∘ (Data a)
                        ⇒ c t (a → b)
```

$$\rightarrow t\ a \rightarrow c\ t\ b)$$
$$\rightarrow (forall\ g \circ t\ g \rightarrow c\ t\ g)$$
$$\rightarrow t\ b \rightarrow c\ t\ b$$
$$mapply :: (Data\ a)$$
$$\Rightarrow t\ (a \rightarrow b) \rightarrow t\ a \rightarrow t\ b$$

Because *MetaGeneric* is intended to be an extension of *Data* it would be natural to assume that there would be some straightforward correspondence between the two, and indeed there is. Compare the declaration of *gfoldl* from the *Generics* library to the declaration of *mgfoldl*.

class $Typeable\ a \Rightarrow Data\ a$ **where**
$$gfoldl :: (forall\ a\ b \circ Data\ a \Rightarrow c\ (a \rightarrow b) \rightarrow a \rightarrow c\ b)$$
$$\rightarrow (forall\ g \circ g \rightarrow c\ g)$$
$$\rightarrow a \rightarrow c\ a$$
...

The type signature of *mgfoldl* simply replaces every $c\ a$ with $c\ t\ a$ and every a with $t\ a$. Additionally *MetaGeneric* requires an *mapply* function for applying $t\ (a \rightarrow b)$ to $t\ a$. *Generics* doesn't require this because application of $a \rightarrow b$ to a is a simple function application.

The Boilerplate II paper [2] builds on the work from Boilerplate I and generalized *Typeable*, part of the *Generics* library. Where as *Typeable* only operates on types of kind $*$, *Typeable1* operates on type constructors of kind $* \rightarrow *$. This extention makes *Generics* applicable in more scenarios than previously. In much the same way, *MetaGeneric* extends *Data*, the other major portion of the *Generics* library. *Data* only works for types of kind $*$, but the *MetaGeneric* class does the same job for type constructors of kind $* \rightarrow *$. An argument could be made that *MetaGeneric* would be more appropriately named *Data1* in that it extends *Data* the same way *Typeable1* extends *Typeable*. However this point is debatable as *MetaGeneric* facilitates manipulations over the parameter to the type constructor not the type constructor itself.

The functions in the *MetaGenerics* library are almost direct copies of their equivalents from *Generics*. The modifications consist of changing the type signature, just like when going from *gfoldl* to *mgfoldl*, and using *mapply* instead of function application. This translation process is very mechanical and simple to perform. One more modification is sometimes required, the use of a temporary wrapper type to fulfill the role of c in the signatures of the functions in the *MetaGeneric* class.

3.1 Aliases

type $MetaGenericT\ t = forall\ a \circ Data\ a \Rightarrow t\ a \rightarrow t\ a$
type $MetaGenericQ\ t\ r = forall\ a \circ Data\ a \Rightarrow t\ a \rightarrow r$
type $MetaGenericM\ t\ m = forall\ a \circ Data\ a \Rightarrow t\ a \rightarrow m\ (t\ a)$
type $MetaGenericC\ t\ c = forall\ a \circ Data\ a \Rightarrow t\ a \rightarrow c\ t\ a$
data $MetaGenericC'\ t\ c = MetaGenericC'\{unMetaGenericC' :: MetaGenericC\ t\ c\}$

3.2 Basics

Wrapper types for the following functions. Note the use of explicitly kinded quantification.

newtype $ID\ (t :: * \rightarrow *)\ a = ID\{unID :: t\ a\}$
newtype $CONST\ c\ (t :: * \rightarrow *)\ a = CONST\{unCONST :: c\}$
newtype $COMPOSE\ c\ (t :: * \rightarrow *)\ a = COMPOSE\{unCOMPOSE :: c\ (t\ a)\}$

$$mgmapT :: (MetaGeneric\ t)$$
$$\Rightarrow MetaGenericT\ t \rightarrow MetaGenericT\ t$$
$$mgmapT\ f\ x = unID\ (mgfoldl\ k\ ID\ x)$$

$$
\begin{aligned}
&\quad\textbf{where}\\
&\qquad k\ (ID\ c)\ x = ID\ (mapply\ c\ (f\ x))
\end{aligned}
$$

$$
\begin{aligned}
&mgmapQl :: (MetaGeneric\ t)\\
&\qquad \Rightarrow (r \to r' \to r) \to r\\
&\qquad \to MetaGenericQ\ t\ r' \to MetaGenericQ\ t\ r\\
&mgmapQl\ o\ r\ f = unCONST \circ mgfoldl\ k\ z\\
&\qquad \textbf{where}\\
&\qquad\qquad k\ c\ x = CONST\ \$\ (unCONST\ c)\ `o`\ f\ x\\
&\qquad\qquad z\ \_ = CONST\ r
\end{aligned}
$$

$$
\begin{aligned}
&mgmapQr :: (MetaGeneric\ t)\\
&\qquad \Rightarrow (r' \to r \to r) \to r\\
&\qquad \to MetaGenericQ\ t\ r' \to MetaGenericQ\ t\ r\\
&mgmapQr\ o\ r\ f\ x = unCONST\ (mgfoldl\ k\ (const\ (CONST\ id))\ x)\ r\\
&\qquad \textbf{where}\\
&\qquad\qquad k\ (CONST\ c)\ x = CONST\ (\lambda r \to c\ (f\ x\ `o`\ r))
\end{aligned}
$$

$$
\begin{aligned}
&mgmapQ :: (MetaGeneric\ t)\\
&\qquad \Rightarrow MetaGenericQ\ t\ u \to MetaGenericQ\ t\ [u]\\
&mgmapQ\ f = mgmapQr\ (:)\ [\,]\ f
\end{aligned}
$$

$$
\begin{aligned}
&mgmapM :: (MetaGeneric\ t, Monad\ m)\\
&\qquad \Rightarrow MetaGenericM\ t\ m \to MetaGenericM\ t\ m\\
&mgmapM\ f\ x = unCOMPOSE\ \$\ mgfoldl\ k\ (COMPOSE \circ return)\ x\\
&\qquad \textbf{where}\\
&\qquad\qquad k\ c\ x = COMPOSE\ \$\ \textbf{do}\\
&\qquad\qquad\qquad c' \leftarrow unCOMPOSE\ c\\
&\qquad\qquad\qquad x' \leftarrow f\ x\\
&\qquad\qquad\qquad return\ (mapply\ c'\ x')
\end{aligned}
$$

3.3 Schemes

$$
\begin{aligned}
&mgeverywhere :: (MetaGeneric\ t)\\
&\qquad \Rightarrow MetaGenericT\ t \to MetaGenericT\ t\\
&mgeverywhere\ f = f \circ mgmapT\ (mgeverywhere\ f)
\end{aligned}
$$

$$
\begin{aligned}
&mgeverything :: (MetaGeneric\ t)\\
&\qquad \Rightarrow (r \to r \to r)\\
&\qquad \to MetaGenericQ\ t\ r \to MetaGenericQ\ t\ r\\
&mgeverything\ k\ f\ x\\
&\qquad = foldl\ k\ (f\ x)\ (mgmapQ\ (mgeverything\ k\ f)\ x)
\end{aligned}
$$

3.4 Twins

The *MetaGenerics* library uses the form of twin traversal via *tfoldl* used in GHC 6.2 because *MetaGenerics* was origonally writen before the work from the Boilerplate II paper [2] was available. Boilerplate II introduces a form of twin traversal based on a *gzipWithQ* function. Converting *MetaGenerics* to use *gzipWithQ* style twin traversal has not yet been attempted; though in theory it should not pose too much difficulty.

$$\textbf{data}\ TWIN\ c\ t\ a = TWIN\ [MetaGenericC'\ t\ c]\ (c\ t\ a)$$

$$
\begin{array}{l}
\mathit{mtfoldl} :: (\mathit{MetaGeneric}\ t, \mathit{MetaGeneric}\ s) \\
\qquad \Rightarrow (\mathit{forall}\ a\ b \circ (\mathit{Data}\ a) \\
\qquad\qquad \Rightarrow c\ t\ (a \to b) \\
\qquad\qquad \to c\ t\ a \to c\ t\ b) \\
\qquad \to (\mathit{forall}\ g \circ t\ g \to c\ t\ g) \\
\qquad \to \mathit{MetaGenericQ}\ s\ (\mathit{MetaGenericC}\ t\ c) \\
\qquad \to \mathit{MetaGenericQ}\ s\ (\mathit{MetaGenericC}\ t\ c) \\
\mathit{mtfoldl}\ k\ z\ t\ \mathit{xs}\ \mathit{ys} = \mathbf{case}\ \mathit{mgfoldl}\ k'\ z'\ \mathit{ys}\ \mathbf{of}\ \{\mathit{TWIN}\ \_\ c \to c\} \\
\qquad \mathbf{where} \\
\qquad\qquad l = \mathit{mgmapQ}\ (\lambda x \to \mathit{MetaGenericC}'\ (t\ x))\ \mathit{xs} \\
\qquad\qquad k'\ (\mathit{TWIN}\ (r : \mathit{rs})\ c)\ y = \mathit{TWIN}\ \mathit{rs}\ (k\ c\ (\mathit{unMetaGenericC}'\ r\ y)) \\
\qquad\qquad k'\ \_\ \_ = \mathit{error}\ (\texttt{"MetaGenerics.mtfoldl: "} \\
\qquad\qquad\qquad +\!\!+\ \texttt{"fewer children in fourth than fifth argument"}) \\
\qquad\qquad z'\ f = \mathit{TWIN}\ l\ (z\ f)
\end{array}
$$

Note that *mtfoldl* will fail if the number of children of *xs* is less than the number of children of *ys*. The *tfoldl* function provided by GHC 6.2 and *gzipWithQ* from Boilerplate II share this problem. Boilerplate II concludes that the failing case is one that simply must be avoided by the user of *gzipWithQ*. When dealing with first class pattern matching there are several cases where this limitation becomes cumbersome.

While it is not possible to construct an *mtfoldl* that is perfectly safe, it is possible to construct an equivalent of *tmapQl* that is safe. Its construction starts by writting an implementation that works only when the number of children of x is less than the number of children of y.

$$
\begin{array}{l}
\mathit{mtmapQl}' :: (\mathit{MetaGeneric}\ t, \mathit{MetaGeneric}\ s) \\
\qquad \Rightarrow (r \to r \to r) \to r \\
\qquad \to \mathit{MetaGenericQ}\ s\ (\mathit{MetaGenericQ}\ t\ r) \\
\qquad \to \mathit{MetaGenericQ}\ s\ (\mathit{MetaGenericQ}\ t\ r) \\
\mathit{mtmapQl}'\ o\ r\ f\ x\ y = \mathit{unCONST}\ \$\ \mathit{mtfoldl}\ k\ z\ f'\ x\ y \\
\qquad \mathbf{where} \\
\qquad\qquad f'\ x\ y = \mathit{CONST}\ \$\ f\ x\ y \\
\qquad\qquad k\ (\mathit{CONST}\ c)\ (\mathit{CONST}\ x) = \mathit{CONST}\ (c\ `o`\ x) \\
\qquad\qquad z\ \_ = \mathit{CONST}\ r
\end{array}
$$

Next define a function that counts the number of children a value has.

$$
\begin{array}{l}
\mathit{mgchildren} :: (\mathit{Num}\ c, \mathit{MetaGeneric}\ t) \Rightarrow t\ b \to c \\
\mathit{mgchildren}\ x = \mathit{unCONST}\ \$\ \mathit{mgfoldl} \\
\qquad (\lambda(\mathit{CONST}\ x)\ \_ \to \mathit{CONST}\ \$\ x + 1) \\
\qquad (\lambda\_ \to \mathit{CONST}\ 0)\ x
\end{array}
$$

The final implementation simply swaps the order of its arguments if the number of children in x is less than the number of children in y.

$$
\begin{array}{l}
\mathit{mtmapQl} :: (\mathit{MetaGeneric}\ t, \mathit{MetaGeneric}\ s) \\
\qquad \Rightarrow (r \to r \to r) \\
\qquad \to r \\
\qquad \to \mathit{MetaGenericQ}\ s\ (\mathit{MetaGenericQ}\ t\ r) \\
\qquad \to \mathit{MetaGenericQ}\ s\ (\mathit{MetaGenericQ}\ t\ r) \\
\mathit{mtmapQl}\ o\ r\ f\ x\ y = \\
\qquad \mathbf{if}\ (\mathit{mgchildren}\ x) > (\mathit{mgchildren}\ y) \\
\qquad\qquad \mathbf{then}\ \mathit{mtmapQl}'\ o\ r\ f\ x\ y \\
\qquad\qquad \mathbf{else}\ \mathit{mtmapQl}'\ o\ r\ (\mathit{flip}\ f)\ y\ x
\end{array}
$$

The resulting *mtmapQl* function is safe to use regardless of the number of children in x or y.

Chapter 4

Simple Annotations

```
module FirstClassPatterns.SimpleAnnotated (
        SimpleAnnotated,
        getSimpleAnnotation,
        setSimpleAnnotation,
        simpleCollapse
) where

import Control.Monad
import Data.Monoid
import Data.Generics

import FirstClassPatterns.Annotated
```

SimpleAnnotated is a specialization of *Annotated* that only handles the case where each node of a value is tagged with a piece of information but the structure of the value is not changed by injection to *Annotated*.

```
type SimpleAnnotated s b = Annotated (SimpleAnnotatedInfo s) b
```

The *SimpleAnnotatedInfo* type stores some information, s, and is wrapped around a constructor, b.

```
data SimpleAnnotatedInfo s b
        = SimpleAnnotatedInfo s b deriving (Show, Typeable)
```

In order to be used with *Annotated* the *SimpleAnnotatedInfo* must be declared an instance of *Monad* this requires that the information stored, s, be an instance of *Monoid* in order to satisfy the *Monad* laws [8].

```
instance (Monoid s) ⇒ Monad (SimpleAnnotatedInfo s) where
        return = SimpleAnnotatedInfo mempty
        (SimpleAnnotatedInfo s1 b1) >>= k
                | SimpleAnnotatedInfo s2 b2 ← k b1
                = SimpleAnnotatedInfo (s1 `mappend` s2) b2
```

These functions allow manipulation of the information stored in a *SimpleAnnotated*.

```
getSimpleAnnotation::
        (Monad (SimpleAnnotatedInfo s))
        ⇒ SimpleAnnotated s b
        → s
getSimpleAnnotation x
        | (SimpleAnnotatedInfo s _) ← annotatedUndefineds x
```

$= s$

$$
\begin{aligned}
&\mathit{setSimpleAnnotation} ::\\
&\qquad s\\
&\qquad \rightarrow \mathit{SimpleAnnotated}\ s\ b\\
&\qquad \rightarrow \mathit{SimpleAnnotated}\ s\ b\\
&\mathit{setSimpleAnnotation}\ s = \mathit{annotatedLift}\\
&\qquad (\lambda(\mathit{SimpleAnnotatedInfo}\ \_\ b) \rightarrow \mathit{SimpleAnnotatedInfo}\ s\ b)
\end{aligned}
$$

Since *annotatedCollapse* applied to a *SimpleAnnotated* s b yields a *SimpleAnnotatedInfo* s b and it is always possible to extract the b part of a *SimpleAnnotatedInfo* s b, it is possible to defined a function that projects from a *SimpleAnnotated* s b directly to a b.

$$
\begin{aligned}
&\mathit{simpleCollapse} :: (\mathit{Monad}\ (\mathit{SimpleAnnotatedInfo}\ s))\\
&\qquad \Rightarrow \mathit{SimpleAnnotated}\ s\ b\\
&\qquad \rightarrow b\\
&\mathit{simpleCollapse}\ x\\
&\qquad \mid (\mathit{SimpleAnnotatedInfo}\ \_\ b) \leftarrow \mathit{annotatedCollapse}\ x\\
&\qquad = b
\end{aligned}
$$

Chapter 5

Symbol Tables as Annotations

```
module FirstClassPatterns.Symbolized (
        SymbolName,
        SymbolTarget,
        SymbolTable,
        Symbolized,
        emptySymbolTable,
        combineSymbolTables,
        addToSymbolTable,
        getSymbolTable,
        setSymbolTable,
        getSymbolTarget,
        castSymbolTarget,
        getCastedSymbol,
        symbolizedCollapse,
        Symbolizer,
        makeSymbolized
) where

import Control.Monad
import Data.Monoid
import Data.Generics
import Data.FiniteMap

import FirstClassPatterns.MetaGenerics
import FirstClassPatterns.SimpleAnnotated
import FirstClassPatterns.Annotated
```

5.1 Attaching Symbol Tables to Nodes

Symbolized a represents an *a* but uses a *SimpleAnnotated* to attach a *SymbolTable* to each node.

A *SymbolTarget* is a wrapper around an existentially quantified *Symbolized a*. It is existentially quantified to allow for symbol tables that may have targets of different types.

```
type SymbolName = String
type Symbolized a = SimpleAnnotated SymbolTable a
newtype SymbolTable = SymbolTable (FiniteMap SymbolName SymbolTarget)
        deriving (Typeable)
```

$\textbf{data}\ \mathit{SymbolTarget} = \mathit{forall}\ a \circ (\mathit{Data}\ a) \Rightarrow \mathit{SymbolTarget}\ (\mathit{Symbolized}\ a)$

In order to be used with *SimpleAnnotated* the *SymbolTable* must be an instance of *Monoid*. If the two arguments to *mappend* have entries with the same *SymbolName* as a key, then the entry from the second *SymbolTable* will override the one from the first *SymbolTable*.

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monoid}\ \mathit{SymbolTable}\ \textbf{where} \\
\qquad \mathit{mempty} = \mathit{SymbolTable}\ \mathit{emptyFM} \\
\qquad \mathit{mappend}\ (\mathit{SymbolTable}\ x)\ (\mathit{SymbolTable}\ y) \\
\qquad\qquad = \mathit{SymbolTable}\ (x\ `\mathit{plusFM}`\ y)
\end{array}
$$

5.2 SymbolTable Functions

The *SymbolTable* with no entries is *emptySymbolTable*.

$$
\begin{array}{l}
\mathit{emptySymbolTable} :: \mathit{SymbolTable} \\
\mathit{emptySymbolTable} = \mathit{SymbolTable}\ \mathit{emptyFM}
\end{array}
$$

One may combine one *SymbolTable* with another using *combineSymbolTables*. In the case that they both have an entry using the same *SymbolName* as a key, the entry from the second *SymbolTable* will be used in the new *SymbolTable*.

$$
\begin{array}{l}
\mathit{combineSymbolTables} :: \mathit{SymbolTable} \rightarrow \mathit{SymbolTable} \rightarrow \mathit{SymbolTable} \\
\mathit{combineSymbolTables} \\
\qquad (\mathit{SymbolTable}\ x) \\
\qquad (\mathit{SymbolTable}\ y) \\
\qquad = (\mathit{SymbolTable}\ (x\ `\mathit{plusFM}`\ y))
\end{array}
$$

The *addToSymbolTable* function adds an entry to a *SymbolTable* that maps a *SymbolName* to a *SymbolTarget*. Any previous mapping is overwritten.

$$
\begin{array}{l}
\mathit{addToSymbolTable} :: \\
\qquad \mathit{SymbolName} \rightarrow \mathit{SymbolTarget} \\
\qquad \rightarrow \mathit{SymbolTable} \rightarrow \mathit{SymbolTable} \\
\mathit{addToSymbolTable}\ n\ t\ (\mathit{SymbolTable}\ st) \\
\qquad = \mathit{SymbolTable}\ (\mathit{addToFM}\ st\ n\ t)
\end{array}
$$

The *getSymbolTable* and *setSymbolTable* functions manipulate the *SymbolTable* associated with a given *Symbolized a*.

$$
\begin{array}{l}
\mathit{getSymbolTable} :: \mathit{Symbolized}\ a \rightarrow \mathit{SymbolTable} \\
\mathit{getSymbolTable} = \mathit{getSimpleAnnotation} \\
\\
\mathit{setSymbolTable} :: \\
\qquad \mathit{SymbolTable} \\
\qquad \rightarrow \mathit{Symbolized}\ a \rightarrow \mathit{Symbolized}\ a \\
\mathit{setSymbolTable} = \mathit{setSimpleAnnotation}
\end{array}
$$

The *getSymbolTarget* function returns *Just* of the *SymbolTarget* that is associated with a given *SymbolName* from a *SymbolTable* if one exists. Otherwise, *Nothing* is returned.

$$
\begin{array}{l}
\mathit{getSymbolTarget} :: \\
\qquad \mathit{SymbolTable} \rightarrow \mathit{SymbolName} \\
\qquad \rightarrow \mathit{Maybe}\ \mathit{SymbolTarget} \\
\mathit{getSymbolTarget}\ (\mathit{SymbolTable}\ st)\ \mathit{key} = \mathit{lookupFM}\ st\ \mathit{key}
\end{array}
$$

The *castSymbolTarget* function attempts to project a *SymbolTarget* to a specific *Symbolized a* using the *cast* operation from the *Generics* library. If the cast is valid then *Just* of the projected value is returned. Otherwise, *Nothing* is returned.

$$
\begin{aligned}
&\mathit{castSymbolTarget} :: (\mathit{Typeable}\ a) \\
&\qquad\qquad \Rightarrow \mathit{SymbolTarget} \\
&\qquad\qquad \rightarrow \mathit{Maybe}\ (\mathit{Symbolized}\ a) \\
&\mathit{castSymbolTarget}\ (\mathit{SymbolTarget}\ x) = \mathit{cast}\ x
\end{aligned}
$$

When looking up a symbol there are two ways failure can happen: 1) when there is no symbol associated with a particular name, 2) when the symbol associated with a particular name is not of the expected type. By testing the results of *getSymbolTarget* and *castSymbolTarget* each of these respective cases can be detected. If the user does not care how the lookup failed, then the *getCastedSymbol* function can be used which combines *getSymbolTarget* and *castSymbolTarget*.

$$
\begin{aligned}
&\mathit{getCastedSymbol} :: (\mathit{Typeable}\ a) \\
&\qquad\qquad \Rightarrow \mathit{SymbolTable} \\
&\qquad\qquad \rightarrow \mathit{SymbolName} \\
&\qquad\qquad \rightarrow \mathit{Maybe}\ (\mathit{Symbolized}\ a) \\
&\mathit{getCastedSymbol}\ \mathit{st}\ \mathit{key} = \mathit{getSymbolTarget}\ \mathit{st}\ \mathit{key} \ggg\!= \mathit{castSymbolTarget}
\end{aligned}
$$

5.3 Mapping to and from Symbolized

Construction of a *Symbolized a* manually can be rather complicated because of the possibility of language constructs that can refer to themselves. The *makeSymbolized* function acts to simplify this for the user. It takes a *Symbolizer* function that describes when and what symbols to add, an initial *SymbolTable*, and an *a* which it injects into a *Symbolized a*.

The *Symbolizer* must take a function that turns an *a* into *SymbolTarget* and the *SymbolTable* to add to along with the current *a*. It should modify the *SymbolTable* to include those elements that are in scope starting with the current node, *a*. This means that all child declarations that are part of the scope of *a* should be added.

When adding to the *SymbolTable* the *Symbolizer* should use the provided function to inject from *a* to *SymbolTarget* in order to construct any *SymbolTarget* that is added to the *SymbolTable*. The provided injection function ensures that the *SymbolTable* currently being constructed will be properly associated with the new *Symbolized a* that is being placed inside the *SymbolTarget*.

$$
\begin{aligned}
&\mathbf{type}\ \mathit{Symbolizer}\ a = \\
&\qquad\quad (\mathit{forall}\ a \circ (\mathit{Data}\ a) \Rightarrow a \rightarrow \mathit{SymbolTarget}) \\
&\qquad\quad \rightarrow \mathit{SymbolTable} \\
&\qquad\quad \rightarrow a \rightarrow \mathit{SymbolTable} \\
\\
&\mathit{makeSymbolized} :: (\mathit{Data}\ a) \\
&\qquad\quad \Rightarrow (\mathit{forall}\ a \circ (\mathit{Data}\ a) \Rightarrow \mathit{Symbolizer}\ a) \\
&\qquad\quad \rightarrow \mathit{SymbolTable} \\
&\qquad\quad \rightarrow a \rightarrow \mathit{Symbolized}\ a \\
&\mathit{makeSymbolized}\ g\ \mathit{st}\ x = \mathit{gfoldl}\ k\ z\ x \\
&\qquad\quad \mathbf{where} \\
&\qquad\qquad\qquad \mathit{st}' = g\ (\mathit{SymbolTarget} \circ \mathit{makeSymbolized}\ g\ \mathit{st}')\ \mathit{st}\ x \\
&\qquad\qquad\qquad k\ c\ x = c\ \text{`}\mathit{mapply}\text{`}\ (\mathit{makeSymbolized}\ g\ \mathit{st}'\ x) \\
&\qquad\qquad\qquad z\ x = \mathit{setSymbolTable}\ \mathit{st}'\ (\mathit{annotatedCtor}\ (\mathit{return}\ x))
\end{aligned}
$$

As a convenience to the user the *simpleCollapse* function is exported under another name so the user does not need to import *SimpleAnnotated* in addition to *Symbolized*.

$symbolizedCollapse :: Symbolized\ a \rightarrow a$
$symbolizedCollapse = simpleCollapse$

5.4 Class Instances for SymbolTable and SymbolTarget

SymbolTable is based on *FiniteMap*, but there is no instance of *Show* declared for *FiniteMap*. Thus *SymbolTable* can not automatically derive *Show*, and an instance must manually be declared. The construction function for *Symbolized* allows a self recursive *SymbolTable* to allow language constructs to refer to themselves. Thus *show* must be written to avoid this cycle and getting into an infinite loop. To get around this problem only the key names are output by *show*.

```
instance Show (SymbolTable) where
    show (SymbolTable fm) = "SymbolTable " ++ (show $ keysFM fm)
```

The use of an existential quantifier in *PatternTarget* means that *Show* can not be automatically derived, so an instance is manually declared.

```
instance Show (SymbolTarget) where
    show (SymbolTarget s) = "SymbolTarget " ++ (show s)
```

This *show* function is not recursive. The call to *show s* refers to the instance of *Show* declared for *Annotated*.

Chapter 6

First Class Patterns

```
{-# OPTIONS -fallow-undecidable-instances #-}
module FirstClassPatterns.Pattern (
        Pattern,
        PatternName,
        getPatternInfoMonad,
        getPatternNames,
        setPatternNames
) where

import Control.Monad
import Data.Generics

import FirstClassPatterns.MetaGenerics
import FirstClassPatterns.Annotated
```

A *Pattern a* represents an *a* where any node can have a name associated with it and any node can be left unspecified. Associating names with nodes is done by having a simple list of names as part of the monad used by *Annotated*. A child may be left out similar to what was discussed in the section on using *Maybe* inside an *Annotated*.

```
type Pattern a = Annotated (PatternInfo PatternName Maybe) a
type PatternName = String
data PatternInfo n m b
        = PatternInfo [n] (m b) deriving (Show)
```

In order to be used with *Annotated* the *PatternInfo* must be declared as an instance of *Monad*. For binding two *PatternInfo*'s where neither of them is empty, the lists of names for each of *PatternInfo* are combined.

```
instance Monad (PatternInfo n Maybe) where
        return = PatternInfo [] ∘ Just
        (PatternInfo ns1 Nothing) >>= _ = PatternInfo ns1 Nothing
        (PatternInfo ns1 (Just x)) >>= k
                | PatternInfo ns2 y ← k x
                = PatternInfo (ns1 ++ ns2) y
```

Since a pattern can represent an unspecified element it is reasonable to declare *PatternInfo* to be an instance of *MonadPlus* where *mzero* represents such an unspecified element.

```
instance (Monad (PatternInfo n m), MonadPlus m)
```

$$
\begin{aligned}
&\Rightarrow \mathit{MonadPlus}\ (\mathit{PatternInfo}\ n\ m)\ \textbf{where}\\
&\mathit{mzero} = \mathit{PatternInfo}\ [\,]\ \mathit{mzero}\\
&\mathit{mplus}\ (\mathit{PatternInfo}\ \mathit{ns1}\ x)\ (\mathit{PatternInfo}\ \mathit{ns2}\ y)\\
&\qquad = \mathit{PatternInfo}\ (\mathit{ns1} +\!\!+\ \mathit{ns2})\ (\mathit{mplus}\ x\ y)
\end{aligned}
$$

The use of a type constructor as a parameter, namely m, means that *Pattern* can't derive *Typeable* so we we declare an instance of it here.

$$
\begin{aligned}
&\textbf{instance}\ (\mathit{Typeable}\ n, \mathit{Typeable}\ (m\ b))\\
&\qquad \Rightarrow \mathit{Typeable}\ (\mathit{PatternInfo}\ n\ m\ b)\ \textbf{where}\\
&\qquad \mathit{typeOf}\ (\_ :: \mathit{PatternInfo}\ n\ m\ b)\\
&\qquad\qquad = \mathit{mkAppTy}\ (\mathit{mkTyCon}\ \texttt{"Pattern.PatternInfo"})\\
&\qquad\qquad\qquad [\mathit{typeOf}\ (\bot :: n),\\
&\qquad\qquad\qquad \mathit{typeOf}\ (\bot :: m\ b)]
\end{aligned}
$$

For reporting the monad inside a *PatternInfo* the observer function *setPatternInfoMonad* is provided. It will typically be used either with *annotatedLift* or after an *annotatedCollapse*.

$$
\begin{aligned}
&\mathit{getPatternInfoMonad} :: \mathit{PatternInfo}\ n\ m\ b \rightarrow m\ b\\
&\mathit{getPatternInfoMonad}\ (\mathit{PatternInfo}\ \_\ \mathit{mb}) = \mathit{mb}
\end{aligned}
$$

The *getPatternNames* and *setPatternNames* functions manipulate the names associated with a pattern. They will usually be used with *annotatedLift*.

$$
\begin{aligned}
&\mathit{setPatternNames} ::\\
&\qquad [\mathit{PatternName}]\\
&\qquad \rightarrow \mathit{Pattern}\ a \rightarrow \mathit{Pattern}\ a\\
&\mathit{setPatternNames}\ \mathit{ns} = \mathit{annotatedLift}\\
&\qquad (\lambda(\mathit{PatternInfo}\ \_\ b) \rightarrow \mathit{PatternInfo}\ \mathit{ns}\ b)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{getPatternNames} :: \mathit{Pattern}\ a \rightarrow [\mathit{PatternName}]\\
&\mathit{getPatternNames}\ x\\
&\qquad \mid (\mathit{PatternInfo}\ \mathit{ns}\ \_) \leftarrow \mathit{annotatedUndefineds}\ x\\
&\qquad = \mathit{ns}
\end{aligned}
$$

Chapter 7

Basic Operations on Patterns

module *FirstClassPatterns.PatternOps* (
    ($$), (%%), (@@), _, (??),
    literalPattern
) **where**

import *Data.Generics*
import *Monad*

import *FirstClassPatterns.MetaGenerics*
import *FirstClassPatterns.Annotated*
import *FirstClassPatterns.Pattern*

infixl 9$$, %%, ??
infixr 9@@

Instead of requiring the user to construct a *Pattern* from the individual elements of an *Annotated*, several combinators are provided for constructing a *Pattern*.

When constructing a *Pattern* either argument may or may not already be a *Pattern*. Instead of requiring the user to remember operators for each of the four possible combinations, the operators are overloaded by declaring a class. Unfortunately it is not possible to declare a single class that handles all four cases but two seperate classes can be declared which together cover all cases.

The *PatternApply* class handles the case where the second argument is not a *Pattern*.

class *PatternApply* $f\ a\ b \mid f \to b$ **where**
    patternApply $:: f \to a \to$ *Pattern* b

instance (*Data* a)
    $\Rightarrow$ *PatternApply* (*Pattern* $(a \to b)$) $a\ b$ **where**
    patternApply $f\ a =$ *mapply* f (*annotatedExpand* a)

instance (*Data* a)
    $\Rightarrow$ *PatternApply* $(a \to b)\ a\ b$ **where**
    patternApply $f\ a =$ *mapply* (*annotatedCtor* $ *return* f) (*annotatedExpand* a)

The *PatternChild* class handles the case where the second argument is already a *Pattern*.

class *PatternChild* $f\ a\ b \mid f \to a, f \to b$ **where**
    patternChild $:: f \to$ *Pattern* $a \to$ *Pattern* b

$$\textbf{instance}\ (Data\ a) \Rightarrow PatternChild\ (Pattern\ (a \to b))\ a\ b\ \textbf{where}$$
$$\quad patternChild\ f\ a = mapply\ f\ a$$

$$\textbf{instance}\ (Data\ a) \Rightarrow PatternChild\ (a \to b)\ a\ b\ \textbf{where}$$
$$\quad patternChild\ f\ a = mapply\ (annotatedCtor\ \$\ return\ f)\ a$$

Operators are aliased to each of the class functions.

$$(\$\$) :: (PatternApply\ f\ a\ b) \Rightarrow f \to a \to Pattern\ b$$
$$(\$\$) = patternApply$$

$$(\%\%) :: (PatternChild\ f\ a\ b) \Rightarrow f \to Pattern\ a \to Pattern\ b$$
$$(\%\%) = patternChild$$

Using these operators one could, for example, write the following expression.

$$pattern1 :: Pattern\ [Maybe\ Char]$$
$$pattern1 = (:)\ \$\$\ (Nothing :: Maybe\ Char)\ \%\%((:)\ \%\%\ (Just\ \$\$\ \texttt{'c'})\ \$\$\ ([\,] :: [Maybe\ Char]))$$

The %% is used when the right hand is a subpattern. Elsewhere $$ is used. However because $$ make a call to *annotatedExpand* it can handle a right hand side that is a complex literal expression. So one could rewrite *pattern1* the following.

$$pattern2 :: Pattern\ [Maybe\ Char]$$
$$pattern2 = (:)\ \$\$\ (Nothing :: Maybe\ Char)\ \$\$\ [Just\ \texttt{'c'}]$$

In both cases the types of expressions had to be given in certain places to assist the type checker. This problem arises when there is a type, such as *Maybe*, that takes a type parameter and the particular constructor, such as *Nothing*, leaves the type parameter unspecified. Another example of this is the list type, $[a]$, and the empty list constructor, $[\,]$. Types that do not take a parameter, do not require these type annotations.

There is the additional possibility of attaching a name to a *Pattern*. The @@ operator takes a left hand that is the name and adds it to the *Pattern* on the right.

$$(@@) :: String \to Pattern\ a \to Pattern\ a$$
$$(@@)\ n\ x = setPatternNames\ (n : (getPatternNames\ x))\ x$$

Another variation is the pattern that matches anything. This is represented by __.

$$\_\_ :: Pattern\ b$$
$$\_\_ = annotatedCtor\ mzero$$

The choice of $$, @@, and __ are meant to serve as mnemonics. The function application operator, $ is mirrored by the pattern application operator, $$. Haskell as-patterns and wild-cards which use @ and _ respectively are mirrored by @@ and __.

With these operators more complicated patterns can be expressed such as *pattern3*.

$$pattern3 :: Pattern\ [Either\ (Maybe\ Int)\ Char]$$
$$pattern3 = (:)\ \$\$\ (Left\ (Just\ 1) :: Either\ (Maybe\ Int)\ Char)\ \%\%\ (\texttt{"cdr"}\ @@\ \_\_)$$

Especially note the use of automatic expansion of literals provided by $$ and the naming of the tail of the list to be "cdr".

The sequence *left* %% (*name* @@ __) is so common that a shortcut, ?? is provided. It is the equivalent of variables in Haskell patterns.

$(??) :: (\mathit{PatternChild}\ f\ a\ b) \Rightarrow f \rightarrow \mathit{String} \rightarrow \mathit{Pattern}\ b$
$(??)\ \mathit{left}\ \mathit{name} = \mathit{left}\ \%\%\ (\mathit{name}\ @@\ \_\_)$

So the equivalent of the Haskell pattern (:) *car* (*Right* _) is *pattern4*.

$\mathit{pattern4} :: \mathit{Pattern}\ [\mathit{Either}\ (\mathit{Maybe}\ \mathit{Int})\ \mathit{Char}]$
$\mathit{pattern4} = (:)\ ??\ \texttt{"car"}\ \%\%\ ((:)\ \%\%\ (\mathit{Right}\ \%\%\ \_\_)$
$\quad \$\$([\,] :: [\mathit{Either}\ (\mathit{Maybe}\ \mathit{Int})\ \mathit{Char}]))$

The one case that remains is a pattern made of a single constructor such as *True* or *Nothing*. The function *literalPattern* is provided for these.

$\mathit{literalPattern} :: (\mathit{Data}\ b) \Rightarrow b \rightarrow \mathit{Pattern}\ b$
$\mathit{literalPattern} = \mathit{annotatedExpand}$

$\mathit{pattern5} :: \mathit{Pattern}\ \mathit{Bool}$
$\mathit{pattern5} = \mathit{literalPattern}\ \mathit{True}$

In addition *literalPattern* works to express more complex literal patterns that don't require __, @@ or ?? like *pattern6*.

$\mathit{pattern6} :: \mathit{Pattern}\ (\mathit{Either}\ (\mathit{Maybe}\ [\mathit{Int}])\ \mathit{Char})$
$\mathit{pattern6} = \mathit{literalPattern}\ (\mathit{Left}\ (\mathit{Just}\ [1, 2, 3]))$

Chapter 8

Pattern Matching

module *FirstClassPatterns.SymbolizedMatch* (
PatternBindings,
PatternTarget,
getPatternTarget,
castPatternTarget,
getCastedPattern,
combinePatternBindings,
emptyPatternBindings,
match
) **where**

import *Control.Monad*
import *Data.Generics*
import *Data.FiniteMap*

import *FirstClassPatterns.MetaGenerics*
import *FirstClassPatterns.Annotated*
import *FirstClassPatterns.SimpleAnnotated*
import *FirstClassPatterns.Pattern*
import *FirstClassPatterns.Symbolized*

8.1 Types

The *SymbolizedMatch* module performs pattern matching between a *Symbolized a* and a *Pattern a*. The matching operation returns a *Maybe PatternBindings*. Whether the match was successful can be determined by checking whether the result was a *Nothing* or a *Just*. In a successful match the parts that the named portions of a pattern were matched against are obtained by examining the contents of the *Just*.

The *PatternBindings* type is a set of mappings from a *PatternName* to a *PatternTarget*. In turn, *PatternTarget* wraps an existentially quantified *Symbolized a*. It is existentially quantified because a named portion of a pattern can occur anywhere so we must be able to place any *Symbolized a* inside a *PatternBindings*.

newtype *PatternBindings* = *PatternBindings* (*FiniteMap PatternName PatternTarget*)
data *PatternTarget* = *forall a* ∘ (*Data a*) ⇒ *PatternTarget* (*Symbolized a*)

We can't derive *Show* for PatternTarget because of the existential quantifier inside it. So we declare an instance of it here.

```
instance Show (PatternTarget) where
        show (PatternTarget a) = "PatternTarget (" ++ show a ++ ")"
```

Note that the call to *show* used by the *show* declared here refers to the instance of *Show* declared for *Annotated*.

8.2 PatternBindings and PatternTarget Functions

The *getPatternTarget* function returns *Just* of the *PatternTarget* that is associated with a given *PatternName* from a set of *PatternBindings* if one exits. Otherwise, *Nothing* is returned.

```
getPatternTarget::
        PatternBindings
        → PatternName
        → Maybe PatternTarget
getPatternTarget (PatternBindings pb) key = lookupFM pb key
```

The *castPatternTarget* function attempts to project a *PatternTarget* to a specific *Symbolized a* using the *cast* operation from the *Generics* library. If the cast is valid then *Just* of the projected value is returned. Otherwise *Nothing* is returned.

```
castPatternTarget :: (Typeable a)
        ⇒ PatternTarget
        → Maybe (Symbolized a)
castPatternTarget (PatternTarget x) = cast x
```

When looking up a pattern binding there are two ways failure can happen. The first is when there is no binding associated with a particular name. The second when the binding associated with a particular name is not of the expected type. By testing the results of *getPatternTarget* and *castPatternTarget* each of these respective cases can be detected. If the user does not care about how the lookup failed, then the *getCastedPattern* function can be used which combines *getPatternTarget* and *castPatternTarget*.

```
getCastedPattern :: (Typeable a)
        ⇒ PatternBindings
        → PatternName
        → Maybe (Symbolized a)
getCastedPattern pb key = getPatternTarget pb key >>= castPatternTarget
```

Two sets of *PatternBindings* may be combined using *combinePatternBindings*. In the case that the two *PatternBindings* both have an entry using the same *PatternName* as a key, the entry from the second set of *PatternBindings* will be used.

```
combinePatternBindings::
        PatternBindings
        → PatternBindings
        → PatternBindings
combinePatternBindings
        (PatternBindings x)
        (PatternBindings y)
        = (PatternBindings (x `plusFM` y))
```

The set of *PatternBindings* with no entries is *emptyPatternBindings*

```
emptyPatternBindings = PatternBindings emptyFM
```

8.3 Performing Pattern Matching

The most important function of *SymbolizedMatch*, *match*, performs the matching of a *Pattern a* against a *Symbolized a*. If they are compatible the result is *Just* of the *PatternBindings* that occur when matching them together. If they are not compatible *match* returns *Nothing*. The matching operation is performed by first attempting to match the top level constructors. This is done by *matchCtor* then combining that with the results of recursively matching each of the children. By using *mtmapQl* from the *MetaGenerics* library the *match* function does not have to worry about the internals of how *Pattern* or *Symbolized* are represented.

```
match :: (Data a)
        ⇒ Symbolized a
        → Pattern a
        → Maybe PatternBindings
match s p =
        liftM2 combinePatternBindings
                (matchCtor s p)
                (mtmapQl
                        (liftM2 combinePatternBindings)
                        (return emptyPatternBindings)
                        (λs p → case castss p of
                                Just p′ → match s p′
                                Nothing → Nothing)
                        s p)
```

The *matchCtor* function performs matching on the constructor of the top most node. It does this by performing *ctorEq* to test constructor equality between *s* and *p*. In that test it must take into account that the pattern, *p*, may use *Nothing* instead of a constructor. In that case the match should succeed because a portion of a pattern that is left unspecified by using *Nothing* matches against anything. A call to *mplus* is used to handle this behavior.

If the match succeeds then the bindings needed to perform the match are returned by making a call to *mkBindings*. Otherwise, *Nothing* is returned to indicate match failure.

```
matchCtor :: (Data b)
        ⇒ Symbolized b
        → Pattern b
        → Maybe PatternBindings
matchCtor s p =
        do
        isMatch ←
                getPatternInfoMonad$
                        (liftM2 ctorEq
                                (return $ simpleCollapse s)
                                (annotatedUndefineds p))
                        ‘mplus‘
                        (return True)
        if isMatch
                then return $ (mkBinding s p)
                else Nothing
```

Once a particular part of a pattern match has been determined to succeed, *mkBinding* is used to make the pattern bindings that that are associated with that part of the pattern match.

```
mkBinding :: (Data a)
        ⇒ Symbolized a
```

$$
\begin{aligned}
&\qquad\quad \rightarrow \mathit{Pattern}\ a\\
&\qquad\quad \rightarrow \mathit{PatternBindings}\\
&\mathit{mkBinding}\ s\ p =\\
&\qquad\quad \mathit{PatternBindings}\ \$\ \mathit{foldl}\\
&\qquad\qquad\qquad (\lambda \mathit{fm}\ n \rightarrow \mathit{addToFM}\ \mathit{fm}\ n\ (\mathit{PatternTarget}\ s))\\
&\qquad\qquad\qquad \mathit{emptyFM}\ (\mathit{getPatternNames}\ p)
\end{aligned}
$$

As a pattern is being matched, each stage only needs to know whether the constructors of two data elements are the same without regard to whether the children are equal. The *ctorEq* function performs this test by finding the constructors using *toConstr*, part of the *Generics* library, and testing equality over them.

$$
\begin{aligned}
&\mathit{ctorEq} :: (\mathit{Data}\ a) \Rightarrow a \rightarrow a \rightarrow \mathit{Bool}\\
&\mathit{ctorEq}\ x\ y = (\mathit{toConstr}\ x) \equiv (\mathit{toConstr}\ y)
\end{aligned}
$$

Note that pattern matches are type safe if the success or failure of a pattern match is all that is required. However, in the current implementation the ability to attach names to patterns makes type safety problematic because the part that each named pattern matched against is stored using a wrapper around the existential type *PatternTarget* which is then placed inside a *FiniteMap*. If pattern variables were positional instead of named, it would be possible to encode the type of the variable bindings in the type of a *Pattern*, which would then allow a pattern match to be preformed with complete type safety while still being type safe. Further research is nessisary to determine whether such a modification to the present system is possible.

Chapter 9

A Language for Constraints

```
module FirstClassPatterns.Constraint (
        ConstraintM,
        runConstraintM,
        ErrorType (FatalError, RuleFailures),
        Env,
        emptyEnv,
        Report,
        Rule,
        passes,
        fails,
        fatalFailure,
        ($∼),
        (=∼),
        matchSuccess,
        extMatch,
        (~~⊣), (⊢~~),
        (====),
        collapse,
        reportBoth,
        reportAll,
        deepRule,
        envPatternBindings,
        envSymbolTable,
        derefSymbolTarget,
        derefPatternTarget,
        derefSymbol,
        derefPattern,
        derefSymbolFromPattern,
        forallInList,
        forallInList',
        collapseList
) where

import Control.Monad
import Control.Monad.Error
import Control.Monad.Reader
import Data.Generics
```

```
import Data.FiniteMap
import FirstClassPatterns.Annotated
import FirstClassPatterns.MetaGenerics
import FirstClassPatterns.Pattern
import FirstClassPatterns.PatternOps
import FirstClassPatterns.SymbolizedMatch
import FirstClassPatterns.Symbolized
```

9.1 Types

The functions in the constraint language use a monad that demonstrates the properties of *MonadError* and *MonadReader*. It is declared here using monad transformers [3].

```
type ConstraintM a = (ErrorT ErrorType (Reader Env)) a

runConstraintM m =
        runReader (runErrorT m)
        emptyEnv
```

The type of errors that the monad may deal with are either a *FatalError* that prevents constraint checking from continuing or a set of *RuleFailures* detailing the constraints that were violated.

```
data ErrorType = FatalError String | RuleFailures [String] deriving (Show)

instance Error ErrorType where
        noMsg = FatalError noMsg
        strMsg = FatalError ∘ strMsg
```

The environment for the reader monad includes the *SymbolTable* and the *PatternBindings* of the last pattern match.

```
data Env = Env SymbolTable PatternBindings
emptyEnv = Env emptySymbolTable emptyPatternBindings
```

A *Rule a* represents a constraint that should hold true for a given *Symbolized a*. The result of applying a *Rule a* to an *a* is a *Report* indicating whether the constraints were satisfied. Note, however, that all the information carried by a *Report* is contained in the monad so the parameter to the monad is the unit type, ().

```
type Report = ConstraintM ()
type Rule a = Symbolized a → Report
```

If a *Rule* is satisfied then *passes* should be returned. If it is not satisfied then *fails* should be passed a string indicating how the *Rule* was not satisfied and returned. The *fails* function in turn throws *RuleFailures*. If there is some unexpected error that indicates there is an error in the *Rule* itself and that processing should halt, then a *fatalFailure* should be passed a string indicating the problem and returned.

```
passes :: Report
passes = return ()

fails :: (MonadError ErrorType m) ⇒ String → m a
fails = throwError ∘ RuleFailures ∘ (:[])

fatalFailure :: (MonadError ErrorType m) ⇒ String → m a
fatalFailure = throwError ∘ FatalError
```

9.2 Pattern and Rule Operations

Since projecting a *Symbolized a* to an *a* is a very common operation, *symbolizedCollapse* is aliased to a shorter form.

$$\begin{array}{l} \mathit{collapse} :: \mathit{Symbolized}\ a \rightarrow a \\ \mathit{collapse} = \mathit{symbolizedCollapse} \end{array}$$

A *Rule a* is just a function from *Symbolized a* to *Report*. Thus if r is a *Rule*, it can be tested against an s by the function application $r\ s$. However since a *Rule* will often be written inline and be textually larger than the *Symbolized a* it is applied to, it is easier to read code that is written with the s first. This effect can be achieved by writing *flip* (\$) s \$ r. That idiom is encapsulated in \$$\sim$ so a user can simply write s \$$\sim$ r.

$$\begin{array}{l} \textbf{infixr}\ 0\$\sim \\ (\$\sim) :: \mathit{Symbolized}\ a \rightarrow (\mathit{Symbolized}\ a \rightarrow m\ b) \rightarrow m\ b \\ (\$\sim) = \mathit{flip}\ (\$) \end{array}$$

The $=\sim$ operator is a modified version of the *SymbolizedMatch.match* function that generalizes the type signature from returning a *Maybe PatternBindings* to returning a *m PatternBindings* where m is any instance of *MonadPlus*.

$$\begin{array}{l} \textbf{infix}\ 4{=}\sim \\ ({=}\sim) :: (\mathit{Data}\ a, \mathit{MonadPlus}\ m) \Rightarrow \mathit{Symbolized}\ a \rightarrow \mathit{Pattern}\ a \rightarrow m\ \mathit{PatternBindings} \\ s =\sim p = \\ \qquad \textbf{case}\ \mathit{match}\ s\ p\ \textbf{of} \\ \qquad\qquad \mathit{Nothing} \rightarrow \mathit{mzero} \\ \qquad\qquad \mathit{Just}\ x \rightarrow \mathit{return}\ x \end{array}$$

The *matchSuccess* function maps the result of an application of *match* or $=\sim$ to a *Bool*. It returns *True* if the match was successful and *False* otherwise.

$$\begin{array}{l} \mathit{matchSuccess} :: \mathit{Maybe}\ \mathit{PatternBindings} \rightarrow \mathit{Bool} \\ \mathit{matchSuccess}\ \mathit{Nothing} = \mathit{False} \\ \mathit{matchSuccess}\ (\mathit{Just}\ \_) = \mathit{True} \end{array}$$

A close kin of *extQ* from the *Generics* libarary, the *extMatch* function evaluates to *ext* when x matches the *guard*. Otherwise it evaluates to *def*. When it evaluates to *ext*, the *Env* of the monad is set to the *PatternBindings* that resulted from the match and the *SymbolTable* associated with x. No equivalent of *mkQ* from the *Generics* libary is provided because that easily written as *extMatch* (*const* \$ *def*) *guard ext*.

$$\begin{array}{l} \mathit{extMatch} :: (\mathit{MonadReader}\ \mathit{Env}\ m, \mathit{Data}\ a) \\ \qquad \Rightarrow (\mathit{Symbolized}\ a \rightarrow m\ b) \rightarrow \mathit{Pattern}\ a \rightarrow m\ b \rightarrow \mathit{Symbolized}\ a \rightarrow m\ b \\ \mathit{extMatch}\ \mathit{def}\ \mathit{guard}\ \mathit{ext}\ x = \\ \qquad \textbf{case}\ x =\sim \mathit{guard}\ \textbf{of} \\ \qquad\qquad \mathit{Nothing} \rightarrow \mathit{def}\ x \\ \qquad\qquad \mathit{Just}\ \mathit{pb} \rightarrow \mathit{local} \\ \qquad\qquad\qquad (\mathit{const}\ (\mathit{Env}\ (\mathit{getSymbolTable}\ x)\ \mathit{pb})) \\ \qquad\qquad\qquad \mathit{ext} \end{array}$$

As was done with \$$\sim$, the *extMatch* function can be easier to use when written as a ternary operator. Using the standard trick [6] *extMatch def guard ext* can be written as *def* $\sim\!\sim\!\dashv$ *guard* $\models\!\sim\!\sim$ *ext*.

$$\textbf{data}\ (\mathit{Monad}\ m) \Rightarrow \mathit{ExtMatch}\ a\ m\ b = \mathit{Pattern}\ a :\sim\sim m\ b$$

$$\begin{array}{l} \textbf{infixl}\ 2{\sim\!\sim\!\dashv} \\ \textbf{infix}\ 3{\models\!\sim\!\sim} \end{array}$$

```
(~~|) :: (MonadReader Env m, Data a)
        ⇒ (Symbolized a → m b)
        → ExtMatch a m b
        → Symbolized a
        → m b
def ~~| guard :~~ ext = (def `extMatch` guard) ext

(|~~) :: (Monad m) ⇒ Pattern a → m b → ExtMatch a m b
(|~~) = (:~~)
```

When written in this way it becomes easy to chain multiple guarded rules together. For example,

```
extExample :: Symbolized (Either String Int) → ConstraintM Int
extExample =
        (const $ fatalFailure "Unexpected pattern failure in extExample")
        ~~|Left ?? "left"|~~
                do
                (left :: Symbolized String) ← derefPattern "left"
                return $ length (collapse left)
        ~~|Right ?? "right"|~~
                do
                (right :: Symbolized Int) ← derefPattern "right"
                return $ (collapse right)
```

When writing constraints over a type it is very common to want to write a *Rule* where some test is evaluated but only if a guard pattern matches and the rule *passes* otherwise. This is expressed by the ==== operator.

```
infix 1====
(====) :: (Data a)
        ⇒ Pattern a → Report
        → Rule a
(====) guard result =
        (const passes)
        ~~|guard|~~
                result
```

The user can then write things such as the following which specifies the constraint every element of a list of strings must have an even length.

```
allStringsHaveEvenLength :: Rule [String]
allStringsHaveEvenLength =
        (:) ?? "head" ?? "tail"
        ====
        do
        (head :: Symbolized String) ← derefPattern "head"
        (tail :: Symbolized [String]) ← derefPattern "tail"
        if length (collapse head) `mod` 2 ≡ 1
                then fails$
                        "Element '" ++ collapse head ++ "'of list has odd length."
                else allStringsHaveEvenLength tail
```

This function may at first appear to have to base condition, but remember that ==== returns *passes* if the guard pattern it not satisfied. So in this case when the end of the list reached, the checking will halt. Any elements that had an odd length will be in the list of rule failures.

9.3 Report Operations

One *Report* can be combined with another using *reportBoth*. Any *RuleFailures* in either *Report* will be combined. However if either *Report* has a *FatalError*, the *FatalError* will override any *RuleFailures*.

$$
\begin{array}{l}
\mathit{reportBoth} :: \mathit{Report} \to \mathit{Report} \to \mathit{Report} \\
\mathit{reportBoth}\ x\ y = \\
\qquad \mathbf{do} \\
\qquad x' \leftarrow \mathit{getError}\ x \\
\qquad y' \leftarrow \mathit{getError}\ y \\
\qquad \mathbf{case}\ (x', y')\ \mathbf{of} \\
\qquad\qquad (\mathit{Nothing}, \mathit{Nothing}) \to \mathit{passes} \\
\qquad\qquad (\mathit{Just}\ (\mathit{FatalError}\ x''), \_) \to \mathit{fatalFailure}\ x'' \\
\qquad\qquad (\_, \mathit{Just}\ (\mathit{FatalError}\ y'')) \to \mathit{fatalFailure}\ y'' \\
\qquad\qquad (\mathit{Just}\ (\mathit{RuleFailures}\ x''), \mathit{Nothing}) \to \mathit{throwError}\ \$\ \mathit{RuleFailures}\ x'' \\
\qquad\qquad (\mathit{Nothing}, \mathit{Just}\ (\mathit{RuleFailures}\ y'')) \to \mathit{throwError}\ \$\ \mathit{RuleFailures}\ y'' \\
\qquad\qquad (\mathit{Just}\ (\mathit{RuleFailures}\ x''), \mathit{Just}\ (\mathit{RuleFailures}\ y'')) \\
\qquad\qquad\qquad \to \mathit{throwError}\ \$\ \mathit{RuleFailures}\ (x'' +\!\!+\ y'') \\
\qquad \mathbf{where} \\
\qquad\qquad \mathit{getError} :: \mathit{Report} \to \mathit{ConstraintM}\ (\mathit{Maybe}\ \mathit{ErrorType}) \\
\qquad\qquad \mathit{getError}\ x = \mathit{catchError}\ (x \gg (\mathit{return}\ \mathit{Nothing}))\ (\mathit{return} \circ \mathit{Just})
\end{array}
$$

Several reports can also be combined using *reportAll*. It simply folds *reportBoth* across a list of reports.

$$
\begin{array}{l}
\mathit{reportAll} :: [\mathit{Report}] \to \mathit{Report} \\
\mathit{reportAll}\ \mathit{xs} = \mathit{foldl}\ \mathit{reportBoth}\ \mathit{passes}\ \mathit{xs}
\end{array}
$$

The *deepRule* function applies a *Rule* to a *Symbolized* and all its children. The results of all these are combined using *reportBoth*. This function is a prime example of using *MetaGenerics* to traverse the a part of a $t\ a$. In this case the function traverses the a in a *Symbolized* a and ignores the details of how *Symbolized* represents the a.

$$
\begin{array}{l}
\mathit{deepRule} :: (\mathit{Typeable}\ a, \mathit{Data}\ b) \Rightarrow \mathit{Rule}\ a \to \mathit{Rule}\ b \\
\mathit{deepRule}\ r = \mathit{mgeverything}\ \mathit{reportBoth}\ (\mathit{passes}\ \text{`}\mathit{mkQ}\text{`}\ r)
\end{array}
$$

9.4 Environment Operations

Access to the current *PatternBindings* and *SymbolTable* in effect are provided by *envPatternBindings* and *envSymbolTable*.

$$
\begin{array}{l}
\mathit{envPatternBindings} :: (\mathit{MonadReader}\ \mathit{Env}\ m) \\
\qquad \Rightarrow m\ (\mathit{PatternBindings}) \\
\mathit{envPatternBindings} = \\
\qquad \mathbf{do} \\
\qquad (\mathit{Env}\ \_\ \mathit{pb}) \leftarrow \mathit{ask} \\
\qquad \mathit{return}\ \mathit{pb}
\end{array}
$$

$$
\begin{array}{l}
\mathit{envSymbolTable} :: (\mathit{MonadReader}\ \mathit{Env}\ m) \\
\qquad \Rightarrow m\ (\mathit{SymbolTable}) \\
\mathit{envSymbolTable} = \\
\qquad \mathbf{do} \\
\qquad (\mathit{Env}\ \mathit{st}\ \_) \leftarrow \mathit{ask} \\
\qquad \mathit{return}\ \mathit{st}
\end{array}
$$

When the user does not want to handle failure cases during a lookup on elements of the *PatternBindings* or *SymbolTable* currently in effect, the user may simply use these monadic variations. Versions that both do and do not perform the needed cast are provided. In the case of an error these functions will throw a generic error message.

$$
\begin{aligned}
&\mathit{derefSymbolTarget} :: (\mathit{MonadError}\ \mathit{ErrorType}\ m, \mathit{MonadReader}\ \mathit{Env}\ m)\\
&\qquad \Rightarrow \mathit{String} \rightarrow m\ \mathit{SymbolTarget}\\
&\mathit{derefSymbolTarget}\ \mathit{key} =\\
&\qquad \mathit{envSymbolTable} \gg\!= \mathit{reportGet}\ \mathit{key}\ \mathit{getSymbolTarget}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{derefPatternTarget} :: (\mathit{MonadError}\ \mathit{ErrorType}\ m, \mathit{MonadReader}\ \mathit{Env}\ m)\\
&\qquad \Rightarrow \mathit{String} \rightarrow m\ \mathit{PatternTarget}\\
&\mathit{derefPatternTarget}\ \mathit{key} =\\
&\qquad \mathit{envPatternBindings} \gg\!= \mathit{reportGet}\ \mathit{key}\ \mathit{getPatternTarget}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{derefSymbol} :: (\mathit{Typeable}\ a, \mathit{MonadError}\ \mathit{ErrorType}\ m, \mathit{MonadReader}\ \mathit{Env}\ m)\\
&\qquad \Rightarrow \mathit{String} \rightarrow m\ (\mathit{Symbolized}\ a)\\
&\mathit{derefSymbol}\ \mathit{key} =\\
&\qquad \mathit{derefSymbolTarget}\ \mathit{key} \gg\!= \mathit{reportCast}\ \mathit{key}\ \mathit{castSymbolTarget}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{derefPattern} :: (\mathit{Typeable}\ a, \mathit{MonadError}\ \mathit{ErrorType}\ m, \mathit{MonadReader}\ \mathit{Env}\ m)\\
&\qquad \Rightarrow \mathit{String} \rightarrow m\ (\mathit{Symbolized}\ a)\\
&\mathit{derefPattern}\ \mathit{key} =\\
&\qquad \mathit{derefPatternTarget}\ \mathit{key} \gg\!= \mathit{reportCast}\ \mathit{key}\ \mathit{castPatternTarget}
\end{aligned}
$$

If the target of a *derefPattern* is a *Pattern String* that is intended to refer to an entry in the current *SymbolTable* then *derefSymbolFromPattern* may be used as a shortcut.

$$
\begin{aligned}
&\mathit{derefSymbolFromPattern} ::\\
&\qquad (\mathit{Typeable}\ a,\\
&\qquad \mathit{MonadError}\ \mathit{ErrorType}\ m,\\
&\qquad \mathit{MonadReader}\ \mathit{Env}\ m)\\
&\qquad \Rightarrow \mathit{String} \rightarrow m\ (\mathit{Symbolized}\ a)\\
&\mathit{derefSymbolFromPattern}\ \mathit{name} =\\
&\qquad \mathit{derefPattern}\ \mathit{name} \gg\!= (\mathit{derefSymbol} \circ \mathit{collapse})
\end{aligned}
$$

The default handling of errors when the user calls one of the *deref* functions are defined by *reportGet* and *reportCast* which are private to the module and simply throw a fatal error containing a short message.

$$
\begin{aligned}
&\mathit{reportGet}\ \mathit{key}\ \mathit{get}\ \mathit{fm} =\\
&\qquad \mathbf{case}\ \mathit{get}\ \mathit{fm}\ \mathit{key}\ \mathbf{of}\\
&\qquad\qquad \mathit{Nothing} \rightarrow \mathit{fatalFailure}\\
&\qquad\qquad\qquad (\texttt{"By name lookup for '"} +\!\!+ \mathit{key} +\!\!+ \texttt{"' failed."})\\
&\qquad\qquad \mathit{Just}\ x \rightarrow \mathit{return}\ x
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{reportCast}\ \mathit{key}\ \mathit{cast}\ x =\\
&\qquad \mathbf{case}\ \mathit{cast}\ x\ \mathbf{of}\\
&\qquad\qquad \mathit{Nothing} \rightarrow \mathit{fatalFailure}\\
&\qquad\qquad\qquad (\texttt{"Value for name '"} +\!\!+ \mathit{key} +\!\!+ \texttt{"' of wrong type'"})\\
&\qquad\qquad \mathit{Just}\ y \rightarrow \mathit{return}\ y
\end{aligned}
$$

9.5 User Library Code

The following functions represent the beginnings of a user library. It is written using only function calls that are available to the user. Once it becomes more complete it will probably become a separate module.

The *forallInList* function transforms a *Rule* over *a* into a *Rule* over [*a*] such that the rule passes only if all of the children pass.

```
forallInList :: (Data a)
        ⇒ Rule a → Rule [a]
forallInList p =
        ((:) %% ("car" @@ _) %% ("cdr" @@ _))
        ====
        do
        car ← derefPattern "car"
        cdr ← derefPattern "cdr"
        local
                (const emptyEnv)
                (reportAll [p car, forallInList p cdr])
```

Because the *Rule* that one is using with *forallInList* is often expressed inline as a large piece of code it is usually more convenient to specify the list to operate on before the *Rule* to apply to it. The *forallInList′* function is a version of *forallInList* with the arguments flipped to allow the user to do that. If production code shows that this version is used significantly more than the plain *forallInList*, then *forallInList′* will probably replace *forallInList* entirely.

```
forallInList′ p = flip forallInList p
```

The standard *Prelude* in Haskell provides several list operations that are quite useful, but these functions can't operate on a *Symbolized* [*a*]. The *collapseList* function monadically collapses the outermost level so one has a [*Symbolized a*]. In theory, for any type that allows it's contents to be parameterized such a function could also be written.

```
collapseList :: (MonadError ErrorType m, MonadReader Env m, Data a)
        ⇒ Symbolized [a] → m [Symbolized a]
collapseList =
        const (return [])
        ~~⊣(:) ?? "car" ?? "cdr"⊢~~
        do
        car ← derefPattern "car"
        cdr ← derefPattern "cdr"
        liftM (car:) (collapseList cdr)
```

Chapter 10

Conclusion

The *Annotated* type allows the storage of extra information in a pre-existing type. It does this while requiring minimal modification to the original type. Frequently this modification is as trivial as adding to the type declaration a deriving clause for *Data* and *Typeable*.

An *Annotated* achieves results similar to those seen with the fixed point of a type functor composition. Even so *Annotated* is not exactly the same. An *Annotated* wraps each constructor with extra information. The functor based approach wraps each child. This poses a limitation on *Annotated* not present in the functor approach. For example, the functor approach could wrap each child with a list, thus representing a tree in which each node has several potential values. Currently *Annotated* cannot denote this, but a modification of *Annoated* that wraps the children instead of the constructor would add this ability. That in turn poses its own problem. The current version of *MetaGenerics* can not be used with such a variation on *Annotated*. Writing an instance of *mgfoldl* would be impossible because the type signatures of the functions passed to it are incorrect. Further research is necessary to resolve this problem.

For ease of manipulation of an *Annotated*, the *MetaGenerics* library extends the work from Boilerplate I [1] to be applicable to the *Annotated* type. During the development of *MetaGenerics*, a means was found for *mtmapQl* to avoid the type-safety hazards of *tmapQl* from the *Generics* library. With the introduction by Boilerplate II [2] of *gzip* forms to the *Generics* library as a replacement of *tfoldl*, it is desirable to perform the analog in *MetaGenerics*. The question, though, remains of how to incorporate the safety that was added to *tmapQl* by this paper into the new *gzip* based forms.

The use of an *Annotated* to attach symbol table information to an abstract syntax tree means that constraint predicates no longer need to be limited to operating in only one scope nor is the predicate required to have special knowledge about how to generate the symbol table. It also demonstrates the use of *Annotated* for attaching simple data to nodes in a tree. Other potential uses include adding information to an abstract syntax tree about what line in a source file a particular node came from.

The construction of first class patterns in *Annotated* shows how more complex applications of *Annotated* may be useful. Refinement is still necessary to make variable bindings in pattern matching type-safe. The current implementation wraps each matched variable in an existential type and requires casting to the appropriate type. In theory if variable bindings were positional instead of named, the types of the bound variables could be embedded in the type signature of the pattern, thus making pattern matching type-safe.

Finally all these features are combined into a language for describing constraints. The portions of an abstract syntax tree that are not of interest may be filtered out by the use of a guarding pattern. Constraints may easily employ non-local information through the use of the symbol table attached to each node. Thus, this language solved the difficulties that other approaches have with changes in scope.

Bibliography

[1] LÄMMEL, R., AND JONES, S. P. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation* (2003), ACM Press, pp. 26–37.

[2] LÄMMEL, R., AND PEYTON JONES, S. Scrap more boilerplate: reflection, zips, and generalised casts. Draft; Submitted to ICFP 2004, 16 Mar. 2004.

[3] LIANG, S., HUDAK, P., AND JONES, M. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1995), ACM Press, pp. 333–343.

[4] MEIJER, E., FOKKINGA, M., AND PATERSON, R. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture* (1991), Springer-Verlag New York, Inc., pp. 124–144.

[5] MEIJER, E., AND HUTTON, G. Bananas in space: extending fold and unfold to exponential types. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture* (1995), ACM Press, pp. 324–333.

[6] SHAN, K. Infix expressions. The Haskell Cafe Mailing List, Jul 2002.

[7] SHIELDS, M. B., AND PEYTON JONES, S. First class modules for Haskell. pp. 28–40.

[8] WADLER, P. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1992), ACM Press, pp. 1–14.