Preprint version.

A Pattern for Almost Compositional Functions

Björn Bringert Aarne Ranta

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
{bringert,aarne}@cs.chalmers.se

Abstract

This paper introduces a pattern for almost compositional functions over recursive data types, and over families of mutually recursive data types. Here "almost compositional" means that for a number of the constructors in the type(s), the result of the function depends only on the constructor and the results of calling the function on the constructor's arguments. The pattern consists of a generic part constructed once for each data type or family of data types, and a task-specific part. The generic part contains the code for the predictable compositional cases, leaving the interesting work to the task-specific part. Examples of the pattern implemented in dependent type theory with inductive families, in Haskell with generalized algebraic data types and rank-2 polymorphism, and in Java using a variant of the Visitor design pattern are given. The relationship to the "Scrap Your Boilerplate" approach to generic programming, and to general tree types in dependent type theory are also investigated.

Categories and Subject Descriptors D1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D3.3 [*Programming Languages*]: Language Constructs and Features—Patterns

General Terms Languages, Design

Keywords Traversal, Abstract syntax, Haskell, Java, Visitor pattern, Dependent type theory

1. Introduction

This paper addresses the issue of repetitive code when defining operations over rich data structures. To give concrete examples of what we would like to be able to do, we start by giving some motivating problems.

1.1 Some motivating problems

Suppose you have an abstract syntax definition with many syntactic types such as statement, expression, and variable.

1. Write a function that renames all variables in a program by prepending an underscore to their names. Do this with a case expression that has just two branches: one for the variables, another for the rest.
2. Write a function that constructs a symbol table containing all variables declared in a program, and the type of each variable. Do this with a case expression that has just two branches: one for declarations, another for the rest.
3. Write a function which gives fresh names to all variables in a program. Do this using only three cases: one for variable bindings, another for variable uses, and a third for the rest.

One problem when writing recursive functions which need to traverse rich data structures is that the straightforward way to write them involves large amounts of traversal code which tends to be repeated in each function. There are several problems with this:

- The repeated traversals are probably implemented using copy-and-paste or retyping, both of which are error-prone and can lead to maintenance problems.
- When we add a constructor to the data type, we need to change all functions that traverse the data type, many of which may not need any specific behavior for the new constructor.
- Repeated traversal code obscures the interesting cases where the functions do their real work.
- Forcing complete traversal code for the whole family of data types when implementing even the simplest function could encourage a less modular programming style where multiple operations are collected in a single function.

1.2 The solution

The pattern which we present in this paper allows the programmer to solve problems such as the above in a (hopefully) intuitive way. First we write the traversal code once and for all for our data type or family of data types. We then reuse this component to succinctly express the operations which we want to define.

1.3 Article overview

We first present the simple case of a single recursive algebraic data type, and show examples of using the pattern for this case, with examples in plain Haskell 98 [12]. After that, we generalize this to the more complex case of a family of data types, and show how the pattern can be used in dependent type theory and Haskell with generalized algebraic data types and rank-2 polymorphism. We go on to express the same pattern in Java with parametric polymorphism, using a variant of the Visitor design pattern. In the following section, we briefly describe some tools which can be used to automate the process of writing the necessary support code for the data type at hand. Finally, we discuss some related work in generic programming, type theory and object-oriented programming, and provide some conclusions.

2. Abstract Syntax and Algebraic Data Types

Algebraic data types provide a natural way to implement the abstract syntax in a compiler. To give an example, the following type in Haskell gives the abstract syntax of lambda calculus with abstractions, applications, and variables. For more information about using algebraic data types to represent abstract syntax for programming languages, see for example Appel's text books on compiler construction [2].

```
data Exp = EAbs String Exp
         | EApp Exp Exp
         | EVar String
```

Pattern matching is the technique for defining functions on algebraic data types. These functions are typically recursive. An example is a function that renames all the variables in an expression by prepending an underscore to their names:

```
rename :: Exp -> Exp
rename e = case e of
  EAbs x b -> EAbs ("_" ++ x) (rename b)
  EApp c a -> EApp (rename c) (rename a)
  EVar x   -> EVar ("_" ++ x)
```

3. Compositional Functions

Many functions used in compilers are *compositional*, in the sense that the result for a complex argument is constructed from the results for its parts. The `rename` function is an example of this. The essence of compositional functions is defined by the following higher-order function:

```
composOp :: (Exp -> Exp) -> Exp -> Exp
composOp f e = case e of
  EAbs x b -> EAbs x     (f b)
  EApp c a -> EApp (f c) (f a)
  _        -> e
```

Its power lies in that it can be used when defining other functions, to take care of cases which are just compositional. Such is the `EApp` case in `rename`, which we thus omit by writing:

```
rename :: Exp -> Exp
rename e = case e of
  EAbs x b -> EAbs ("_" ++ x) (rename b)
  EVar x   -> EVar ("_" ++ x)
  _        -> composOp rename e
```

In general, an abstract syntax has many more constructors, and this pattern saves much more work. For instance, in the implementation of GF [19], the `Exp` type has 30 constructors, and `composOp` is used over 20 times, typically covering 90 % of all cases.

A major restriction of `composOp` is that its return type is `Exp`. How do we use it if we want to return something else? If we simply want to compute some result using the abstract syntax tree, without modifying the tree, we can use `composOpFold`:

```
composOpFold :: b -> (b -> b -> b)
             -> (Exp -> b) -> Exp -> b
composOpFold zero combine f e =
  case e of
    EAbs x b -> f b
    EApp c a -> f c `combine` f a
    _        -> zero
```

The first argument is the value that is returned for leaf nodes, the second is a function used to combine the results for children of nodes with more than one child, and the third is a function which is applied to all the children of the given node.

Using `composOpFold` we can now, for example, write a function which gets the names of all free variables in an expression:

```
free :: Exp -> [String]
free e = case e of
  EAbs x b -> delete x (free b)
  EVar x -> [x]
  _ -> composOpFold [] union free e
```

3.1 Monadic compositional functions

When defining a compiler in Haskell, it is convenient to use monads instead of plain functions, to deal with errors, state, etc. To this end, we generalize `composOp` to a monadic variant:

```
composOpM :: Monad m =>
             (Exp -> m Exp) -> Exp -> m Exp
composOpM f e = case e of
  EAbs x b -> return EAbs `ap` return x `ap` f b
  EApp c a -> return EApp `ap` f c `ap` f a
  _        -> return e
```

Here we are using the `Monad` type class and the `ap` function from the Haskell 98 libraries [12]:

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a

ap :: (Monad m) => m (a -> b) -> m a -> m b
```

We can define the ordinary `composOp` as a special case. Using the `Identity` monad [11], we write:

```
composOp :: (Exp -> Exp) -> Exp -> Exp
composOp f =
  runIdentity . composOpM (Identity . f)
```

If we want to maintain some state across the computation over the tree, we can use use `composOpM` with a state monad [11]. In the example below, we will use a state monad `State` with these operations:

```
readState :: State s s
updateState :: (s -> s) -> State s ()
writeState :: s -> State s ()
runState :: s -> State s a -> (a,s)
```

Now we can, for example, write a function that gives fresh names of the form $\_n$, where n is an integer, to all bound variables in an expression. Here the state is an infinite supply of fresh variable names, and we pass a table of the new names for the bound variables to the recursive calls.

```
fresh :: Exp -> Exp
fresh = fst . runState names . f []
  where
  names = ["_" ++ show n | n <- [0..]]
  f vs t = case t of
    EAbs x b -> do
                y:fs <- readState
                writeState fs
                liftM (EAbs y) (f ((x,y):vs) b)
    EVar x ->
      return (EVar (fromMaybe x (lookup x vs)))
    _ -> composOpM (f vs) t
```

3.2 Generalizing `composOpM` and `composOpFold`

We can generalize `composOpM` and `composOpFold` to a single function `compos`. It is basically the same as `composOpM` but it takes the `return` and `ap` functions as arguments:

```
compos :: (forall a. a -> m a)
       -> (forall a b. m (a -> b) -> m a -> m b)
       -> (Exp -> m Exp) -> Exp -> m Exp
compos return ap f e = case e of
  EAbs x b -> return EAbs `ap` return x `ap` f b
  EApp g h -> return EApp `ap` f g `ap` f h
  _        -> return e
```

We define `composOpM` and `composOpFold` in terms of `compos`:

```
composOpM :: Monad m =>
             (Exp -> m Exp) -> Exp -> m Exp
composOpM = compos return ap

newtype C b a = C { unC :: b }
composOpFold :: b -> (b -> b -> b)
             -> (Exp -> b) -> Exp -> b
composOpFold z c f =
  unC . compos (\_ -> C z)
               (\(C x) (C y) -> C (c x y)) (C . f)
```

The definition of `composOpFold` requires a dummy type `C` which is used to throw away the tree result, keeping the `b` result which we are interested in.

3.3 Compositional operations and applicative functors

The first two arguments of `compos` have the same types as the operations of an *applicative functor*, as introduced by McBride and Paterson [16]. Applicative functors are more general than monads, in the sense that while all monads are applicative functors, not all applicative functors are monads. Applicative functors have two operations, `pure` and `*`, which correspond to the `return` and `ap` operations of a `Monad`:

```
class Applicative f where
  pure :: a -> f a
  (*) :: f (a -> b) -> f a -> f b
```

The operations must satisfy these laws:

Identity `pure id * u = u`

Composition `pure (.) * u * v * w = u * (v * w)`

Homomorphism `pure f * pure x = pure (f x)`

Interchange `u * pure x = pure (\f -> f x) * u`

We could use these laws to prove properties of our compositional operations. For example, we would like `compos` not to modify the tree on its own, i.e. that:

```
compos pure (*) pure t = pure t
```

3.4 More examples for the functional language

These are some more examples of operations we could implement for our small functional language using the general functions defined above:

1. Count the number of free occurrences of a given variable in a term.
2. Perform capture-avoiding substitution of a term for a variable. This could use the `fresh` function from Section 3.1.
3. Reduce all β-redexes where the abstracted variable occurs free exactly once in the body of the abstraction. This can be implemented using the previous two operations.

The implementation of these functions are left as an exercise to the reader.

4. Systems of Data Types

For many languages, the abstract syntax is not just one data type, but many, which are often defined by mutual induction. An example is the following simple imperative language with statements, expressions, variables, and types. In this language, statements that return values (for example assignments or maybe blocks that end with a return statement) can be used as expressions.

```
data Stm = SDecl  Typ   Var
         | SAss   Var   Exp
         | SBlock [Stm]
         | SReturn Exp

data Exp = EStm   Stm
         | EAdd   Exp   Exp
         | EVar   Var
         | EInt   Int

data Var = V String

data Typ = T_int | T_float
```

Now we cannot any longer easily define general `composOp` functions, as some of the recursive calls must be done on terms which have different types than the value on which the function was called. Implementing operations such as α-conversion on this kind of family of data types quickly becomes very laborious.

4.1 Categories and trees

An alternative to separate mutual data types for abstract syntax is to define just one type `Tree`, whose constructors take `Tree`s as arguments:

```
data Tree = SDecl   Tree Tree
          | SAss    Tree Tree
          | SBlock  [Tree]
          | SReturn Tree
          | EStm    Tree
          | EAdd    Tree Tree
          | EVar    Tree
          | EInt    Int
          | V       String
          | T_int
          | T_float
```

This approach, however, does not constrain the combinations enough for our liking: there are many `Tree`s that are even syntactically nonsense. This is essentially the representation one would use in a dynamically typed language.

A solution to this problem is provided by dependent types. Instead of a constant type `Tree`, we define an **inductive family** `Tree c`, indexed on a **category**, `c`. The category is just a label to distinguish between different types of trees. We must now leave standard Haskell and use a Haskell-like language with dependent types and inductive families. Agda [5] is one such language. What one would define in Agda is an enumerated type:

```
data Cat = Stm | Exp | Var | Typ
```

followed by an `idata` (inductive data type, or in this case an inductive family of data types) definition of `Tree`, indexed on `Cat`. We omit the Agda definitions of the `Tree` family and the `compos` function as they are virtually identical to the Haskell versions shown below, except that in Agda the index for `Tree` is a value of type `Cat`, whereas in Haskell the index is a dummy data type.

We can also do our exercise with the limited form of dependent types provided by Haskell since GHC 6.4: **Generalized Algebraic Data Types** (GADTs) [13]. We cannot quite define a *type* of categories, but we can define a set of dummy data types:

```
data Stm; data Exp; data Var; data Typ
```

To define the inductive family of trees, we write, in this extension of Haskell:

```
data Tree :: * -> * where
  SDecl   :: Tree Typ -> Tree Var -> Tree Stm
  SAss    :: Tree Var -> Tree Exp -> Tree Stm
  SBlock  :: [Tree Stm] -> Tree Stm
  SReturn :: Tree Exp -> Tree Stm
  EStm    :: Tree Stm -> Tree Exp
  EAdd    :: Tree Exp -> Tree Exp -> Tree Exp
  EVar    :: Tree Var -> Tree Exp
  EInt    :: Int    -> Tree Exp
  V       :: String -> Tree Var
  T_int   :: Tree Typ
  T_float :: Tree Typ
```

In Haskell we cannot restrict the types used as indices in the `Tree` family, which makes it entirely possible to construct types such as `Tree String`. However, since there are no constructors of this type, $\perp$ is the only element in it.

Note that the canonical expressions of syntax trees look just the same as they did in the case of mutual data types and the universal tree type. However, their types now have the form `Tree c` for some `c`. If we want, we can give the dummy types different names, for example `Stm_`, `Exp_`,`Var_`, and `Typ_`, and use type synonyms to make the types also look like they did when we had multiple data types:

```
type Stm = Tree Stm_; type Exp = Tree Exp_
type Var = Tree Var_; type Typ = Tree Typ_
```

4.2 Compositional operations

The power of inductive families is shown in the definition of the function `compos`. We now define it simultaneously for the whole syntax, and can then use it to define any tree-traversing programs concisely.

```
compos :: (forall a. a -> m a)
       -> (forall a b. m (a -> b) -> m a -> m b)
       -> (forall a. Tree a -> m (Tree a))
       -> Tree c -> m (Tree c)
compos return ap f t = case t of
  SDecl x y -> return SDecl `ap` f x `ap` f y
  SAss x y  -> return SAss `ap` f x `ap` f y
  SBlock xs -> return SBlock `ap` mapM f xs
  SReturn x -> return SReturn `ap` f x
  EAdd x y  -> return EAdd `ap` f x `ap` f y
  EStm x    -> return EStm `ap` f x
  EVar x    -> return EVar `ap` f x
  _         -> return t
 where mapM g =
    foldr (ap . ap (return (:)) . g) (return [])
```

The third argument to `compos`, the function to apply to the subtrees, is now a polymorphic function, since it is applied to subtrees of different types. The `mapM` function with the unreadable implementation simply does the same thing as the normal `mapM`, but using the given functions. The other `composOp*` functions are special cases of `compos` in the same way as before.

4.3 A library of compositional operations

In order to provide generic implementations of the different functions, we overload `compos` and define the other operations in terms of it. See Figure 1 for definitions of the other functions in terms of the overloaded `compos` function.

4.4 Examples

4.4.1 Example: Rename variables

Defining a renaming function for the original Haskell definition with separate data types would be very laborious. But now it is easy:

```
rename :: Tree c -> Tree c
rename t = case t of
  V x -> V ("_" ++ x)
  _   -> composOp rename t
```

4.4.2 Example: Warnings for assignments

To encourage pure functionality, this function sounds the bell each time an assignment occurs. Since we are not interested in the return value of the function, but only in its `IO` outputs, we use the function `composOpM_` (like `composOpM` but without a tree result, see Figure 1 for its definition).

```
warnAssign :: Tree c -> IO ()
warnAssign t = case t of
  SAss _ _ -> putChar (chr 7)
  _ -> composOpM_ warnAssign t
```

4.4.3 Example: Symbol table construction

This function constructs a variable symbol table by folding over the syntax tree. Once again, the return value is of no interest. We use `composOpMonoid`, which is simply `composOpFold` specialized to the `Monoid` type class [11]. This uses the `Monoid` instance for lists.

```
symbols :: Tree c -> [(Tree Var,Tree Typ)]
symbols t = case t of
  SDecl typ var -> [(var,typ)]
  _ -> composOpMonoid symbols t
```

4.4.4 Example: Constant folding

We want to replace additions of constants by their result. Here is a first attempt:

```
constFold :: Tree c -> Tree c
constFold e = case e of
  EAdd (EInt x) (EInt y) -> EInt (x+y)
  _ -> composOp constFold e
```

This works for simple cases, but what about for example $1 + (2 + 3)$? This is an addition of constants, but is not matched by our pattern above. We have to look at the results of the recursive calls:

```
constFold' :: Tree c -> Tree c
constFold' e = case e of
  EAdd x y -> case (constFold' x, constFold' y) of
                (EInt n, EInt m) -> EInt (n+m)
                (x',y') -> EAdd x' y'
  _ -> composOp constFold' e
```

This illustrates a common pattern used when the recursive calls can introduce terms which we want to handle.

```
{-# OPTIONS_GHC -fglasgow-exts #-}
module ComposOp (Compos(..),composOp,composOpM,composOpM_,composOpMonoid,
                 composOpMPlus,composOpFold) where

import Control.Monad.Identity
import Data.Monoid

class Compos t where
  compos :: (forall a. a -> m a) -> (forall a b. m (a -> b) -> m a -> m b)
         -> (forall a. t a -> m (t a)) -> t c -> m (t c)

composOp :: Compos t => (forall a. t a -> t a) -> t c -> t c
composOp f = runIdentity . composOpM (Identity . f)

composOpM :: (Compos t, Monad m) => (forall a. t a -> m (t a)) -> t c -> m (t c)
composOpM = compos return ap

composOpM_ :: (Compos t, Monad m) => (forall a. t a -> m ()) -> t c -> m ()
composOpM_ = composOpFold (return ()) (>>)

composOpMonoid :: (Compos t, Monoid m) => (forall a. t a -> m) -> t c -> m
composOpMonoid = composOpFold mempty mappend

composOpMPlus :: (Compos t, MonadPlus m) => (forall a. t a -> m b) -> t c -> m b
composOpMPlus = composOpFold mzero mplus

composOpFold :: Compos t => b -> (b -> b -> b) -> (forall a. t a -> b) -> t c -> b
composOpFold z c f = unC . compos (\_ -> C z) (\(C x) (C y) -> C (c x y)) (C . f)

newtype C b a = C { unC :: b }
```

Figure 1. The ComposOp Module

4.4.5 Example: Syntactic sugar

This example shows how easy it is to add syntax constructs as syntactic sugar, i.e. syntactic constructs that can be eliminated. Suppose you want to add increment statements. This means a new branch in the definition of `Tree c` from Section 4.1:

```
SIncr :: Tree Var -> Tree Stm
```

Increments are eliminated by translation to assignments as follows:

```
elimIncr :: Tree c -> Tree c
elimIncr t = case t of
  SIncr v -> SAss v (EAdd (EVar v) (EInt 1))
  _ -> composOp elimIncr t
```

4.4.6 More examples for the imperative language

Here are some more examples which can benefit from being implemented using the almost compositional function pattern. Their implementations are again left as exercises to the reader.

1. α-conversion of x to y without caring about captures.
2. α-conversion avoiding captures.
3. Substitution of a variable by an expression, avoiding capture.
4. Optimizations: constant propagation, remove addition with 0, remove unused assignments.
5. Line counter: assume each declaration and assignment is one line.
6. Symbol table with failure if the variable is already declared.
7. Type checker: report on ill-typed expressions and assignments, and undeclared variables.
8. Type-annotating type checker: introduce new constructors for typed addition and typed variables, and translate all addition and variable expressions in a program to these forms.

5. Almost Compositional Functions and the Visitor Design Pattern

The Visitor design pattern [7] is a pattern used in object-oriented programming to define an operation for each of the concrete elements of an object hierarchy. We will show how an adaptation of the Visitor pattern can be used to define almost compositional functions in object-oriented languages, in a manner quite similar to that shown above for languages with algebraic data types and pattern matching.

First we present the object hierarchies corresponding to the algebraic data types. Each object hierarchy has a generic Visitor interface. We then show a concrete visitor that corresponds to the `composOp` function.

5.1 Data representation

We use a standard encoding of abstract syntax trees in Java [3], along with the support code for a type-parametrized version of Visitor design pattern. For each algebraic data type in the Haskell version, we have an abstract base class in the Java representation:

```
public abstract class Stm {
  public abstract <R,A> R accept(Visitor<R,A> v,
                                 A arg);
  public interface Visitor <R,A> {
```

```
    public R visit(SDecl p, A arg);
    public R visit(SAss p, A arg);
    public R visit(SBlock p, A arg);
    public R visit(SReturn p, A arg);
    public R visit(SInc p, A arg);
  }
}
```

The base class contains an interface for visitors with methods for visiting each of the inheriting classes. It also specifies that each inheriting class must have a method for accepting the visitor. This method dispatches the call to the correct method in the visitor.

For each data constructor in the algebraic data type, we have a concrete class which inherits from the abstract base class:

```
public class SDecl extends Stm {
  public final Typ typ_; public final Var var_;
  public SDecl(Typ p1, Var p2) { typ_ = p1;
                                 var_ = p2; }
  public <R,A> R accept(Visitor<R,A> v, A arg) {
    return v.visit(this, arg);
  }
}
```

The Visitor interface can be used to define operations on all the concrete classes in one or more of the hierarchies (when defining an operation on more than one hierarchy, the visitor implements multiple Visitor interfaces). This corresponds to the initial examples of pattern matching on all of the constructors, as shown in Section 2. It suffers from the same problem: lots of repetitive traversal code.

5.2 `ComposVisitor`

We can create a class which does all of the traversal and tree rebuilding. This corresponds to the `composOp` function from the Hakell implementation.

```
public class ComposVisitor<A> implements
  Stm.Visitor<Stm,A>, Exp.Visitor<Exp,A>,
  Var.Visitor<Var,A>, Typ.Visitor<Typ,A> {

  public Stm visit(SDecl p, A arg) {
    Typ typ_ = p.typ_.accept(this, arg);
    Var var_ = p.var_.accept(this, arg);
    return new SDecl(typ_, var_);
  }

  // ...
}
```

The `ComposVisitor` class implements all the `Visitor` interfaces in the abstract syntax, and can thus visit all of the constructors in all of the types. Each `visit()` method visits the children of the current node, and then constructs a new node with the results returned from these visits.

The code above could be optimized to eliminate the reconstruction overhead when the recursive calls do not modify the subtrees. For example, if all the objects which are being traversed are immutable, unnecessary copying could be avoided by doing a pointer comparison between the old and the new child. If all the children are the same as the old, we do not need to construct a new parent.

5.3 Using `ComposVisitor`

While the `composOp` function takes a function as a parameter, and applies that function to each node in the tree, the `ComposVisitor` class in itself is pretty much just a complicated implementation of the identity function. Its power comes from the fact that we can override individual `visit()` methods.

When using the standard Visitor pattern, adding new operations is easy, but adding new elements to the object hierarchy is difficult, since it requires changing the code for all the operations. Having a `ComposVisitor` changes this as we can add a new element, and only have to change the `Visitor` interface, the `ComposVisitor`, and any operations which need to have special behavior for the new class.

The Java code below implements the desugaring example from Section 4.4.5 where increments are replaced by addition and assignment. Note that in Java we only need the interesting case, all the other cases are taken care of by the parent class.

```
class Desugar extends ComposVisitor<Object> {
  public Stm visit(SInc i, Object arg) {
    Exp rhs = new EAdd(new EVar (i.var_),
                       new EInt(1));
    return new SAss(i.var_, rhs);
  }
}

Stm desugar(Stm stm) {
  return stm.accept(new Desugar(), null);
}
```

The `Object` argument to the `visit()` method is a dummy since this visitor does not need any extra arguments. The `desugar()` method at the end is just a wrapper used to hide the details of getting the visitor to visit the statement, and passing in the dummy argument.

This being an imperative language, we don't have to do anything special to be able to thread a state though the computation. Here is the symbol table construction function from Section 4.4.3 in Java:

```
class BuildSymTab extends ComposVisitor<Object> {
  Map<Var,Typ> symTab = new HashMap<Var,Typ>();

  public Stm visit(SDecl d, Object arg) {
    symTab.put(d.var_, d.typ_);
    return d;
  }
}

Map<Var,Typ> symbolTable(Stm stm) {
  BuildSymTab v = new BuildSymTab();
  stm.accept(v, null);
  return v.symTab;
}
```

You may wonder why this function was implemented as a stateful computation instead of as a fold like in the Haskell version. Creating a visitor which corresponds to `composOpFold` would be less elegant in Java, since we would have to pass a combining function and a base case value to the visitor. This could be done by adding abstract methods in the visitor, but in most cases the stateful implementation is probably more idiomatic in Java.

Our final Java example is the example from Section 3, where we compute the set of free variables in a term in the small functional language introduced in Section 2.

```
class Free extends ComposVisitor<Set<String>> {
  public Exp visit(EAbs e, Set<String> vs) {
    Set<String> xs = new TreeSet<String>();
    e.exp_.accept(this, xs);
    xs.remove(e.ident_);
    vs.addAll(xs);
    return e;
  }
```

```
  public Exp visit(EVar e, Set<String> vs) {
    vs.add(e.ident_);
    return e;
  }
}

Set<String> freeVars(Exp exp) {
  Set<String> vs = new TreeSet<String>();
  exp.accept(new Free(), vs);
  return vs;
}
```

Here we make use of the possibility of passing an extra argument to the `visit()` methods. The argument is a set to which the `visit()` method adds all the free variables in the visited term.

6. Language and Tool Support for Compositional Operations

A drawback of using the method we have described is that one needs to define the `compos` function for each type or type family. Another problem when working in Haskell is that the current version of GHC does not support type class derivation for GADTs, which means that we often also have to write instances for the common built-in type classes, such as Eq, Ord and Show.

To create `Compos` instances automatically, we could extend the Haskell compiler to allow deriving instances of `Compos`. Another possibility would be to generate the instances using Template Haskell [20] or DrIFT [22], though these tools do not yet support GADTs.

We have added a new back-end to the BNF Converter (BNFC) [17, 6] tool which generates a Haskell GADT abstract syntax type along with instances of Compos, Eq, Ord and Show. We have also extended the BNFC Java 1.5 back-end to generate the Java abstract syntax representation shown above, along with the `ComposVisitor` class. In addition to the abstract syntax types and traversal components described in this paper, the generated code also includes a lexer, a parser, and a pretty printer. We can generate all the Haskell or Java code for our simple imperative language example using the grammar shown below. It is written in LBNF (Labelled Backus-Naur Form), the input language for BNFC.

```
SDecl.   Stm  ::= Typ Var ";" ;
SAss.    Stm  ::= Var "=" Exp ";" ;
SBlock.  Stm  ::= "{" [Stm] "}" ;
SReturn. Stm  ::= "return" Exp ";" ;
SInc.    Stm  ::= Var "++" ";" ;
separator Stm "" ;

EStm.    Exp1 ::= Stm ;
EAdd.    Exp1 ::= Exp1 "+" Exp2 ;
EVar.    Exp2 ::= Var ;
EInt.    Exp2 ::= Integer ;
EDbl.    Exp2 ::= Double ;
coercions Exp 2 ;

V.       Var  ::= Ident ;

TInt.    Typ  ::= "int";
TDbl.    Typ  ::= "double";
```

7. Related Work

7.1 Scrap Your Boilerplate

The part of this work dealing with functional programming languages can be seen as a light-weight solution to a subset of the problems solved by generic programming systems. We use traversal operations similar to those in the "Scrap Your Boilerplate" (SYB) [14] approach. However, no attempt is made to support completely generic functions such as those in "Generics for the Masses" [8] or PolyP [10]. In this section we attempt to compare and contrast our work and SYB.

7.1.1 Introduction to Scrap Your Boilerplate

SYB uses generic traversal functions along with a type safe cast operation implemented by the use of type classes. This allows the programmer to extend fully generic operations with type-specific cases, and use these in various traversal schemes. Data types must have instances of the `Typeable` and `Data` type classes to be used with SYB. Figure 2 lists the SYB type classes and functions which we will use below.

The original "Scrap Your Boilerplate" paper [14] contains a number of examples, some of which we will show as an introduction and later use for comparison. In the examples, some type synonyms (`GenericT` and `GenericQ`) have been inlined to make the function types more transparent. The examples work on a family of data types:

```
data Company  = C [Dept]
                deriving (Typeable,Data)
data Dept     = D Name Manager [Unit]
                deriving (Typeable,Data)
data Unit     = PU Employee | DU Dept
                deriving (Typeable,Data)
data Employee = E Person Salary
                deriving (Typeable,Data)
data Person   = P Name Address
                deriving (Typeable,Data)
data Salary   = S Float
                deriving (Typeable,Data)
type Manager  = Employee
type Name     = String
type Address  = String
```

The first example increases the salary of all employees:

```
increase :: Data a => Float -> a -> a
increase k = everywhere (mkT (incS k))

incS :: Float -> Salary -> Salary
incS k (S s) = S (s * (1+k))
```

More advanced traversal schemes are also supported. This example increases the salary of everyone in a named department:

```
incrOne :: Data a => Name -> Float -> a -> a
incrOne n k a | isDept n a = increase k a
              | otherwise = gmapT (incrOne n k) a

isDept :: Data a => Name -> a -> Bool
isDept n = False `mkQ` isDeptD n

isDeptD :: Name -> Dept -> Bool
isDeptD n (D n' _ _) = n==n'
```

SYB also supports queries, that is, functions which compute some result from the data structure rather than returning a modified structure. This example computes the sum of the salaries of everyone in the company:

```
salaryBill :: Company -> Float
salaryBill = everything (+) (0 `mkQ` billS)
```

```
class Typeable a where
  -- Takes a value of type a and returns a concrete representation of that type.
  typeOf :: a -> TypeRep

class Typeable a => Data a where
  -- Left-associative fold operation for constructor applications
  gfoldl :: (forall a b. Data a => c (a -> b) -> a -> c b) -> (forall g. g -> c g) -> a -> c a
  -- A generic monadic transformation that maps over the immediate subterms, defined in terms of gfoldl
  gmapM :: Monad m => (forall b. Data b => b -> m b) -> a -> m a
  -- A generic query that processes the immediate subterms and returns a list, defined in terms of gfoldl
  gmapQ :: (forall b. Data b => b -> u) -> a -> [u]
  -- A generic transformation that maps over the immediate subterms, defined in terms of gfoldl
  gmapT :: (forall b. Data b => b -> b) -> a -> a
  ...

-- Make a generic transformation: start from a type-specific case; preserve the term otherwise
mkT :: (Typeable b, Typeable a) => (b -> b) -> a -> a
-- Make a generic query: start from a type-specific case; return a constant otherwise
mkQ :: (Typeable b, Typeable a) => r -> (b -> r) -> a -> r
-- Make a generic monadic transformation: start from a type-specific case; resort to return otherwise
mkM :: (Typeable b, Typeable a, Monad m) => (b -> m b) -> a -> m a

-- Extend a generic transformation by a type-specific case
extT :: (Typeable b, Typeable a) => (a -> a) -> (b -> b) -> a -> a
-- Extend a generic query by a type-specific case
extQ :: (Typeable b, Typeable a) => (a -> q) -> (b -> q) -> a -> q
-- Extend a generic monadic transformation by a type-specific case
extM :: (Typeable b, Typeable a, Monad m) => (a -> m a) -> (b -> m b) -> a -> m a

-- Apply a transformation everywhere in bottom-up manner
everywhere :: Data a => (forall a. Data a => a -> a) -> a -> a
-- Summarise all nodes in top-down, left-to-right order
everything :: Data a => (r -> r -> r) -> (forall a. Data a => a -> r) -> a -> r
```

Figure 2. The type classes and some functions from Scrap Your Boilerplate.

```
billS :: Salary -> Float
billS (S f) = f
```

7.1.2 SYB examples using compositional operations

We will now show the above examples implemented using our compositional operations. We first lift the family of data types from the previous section into a GADT:

```
data Company; data Dept
data Unit;    data Employee
data Person;  data Salary

type Manager  = Employee
type Name     = String
type Address  = String

data Tree :: * -> * where
  C :: [Tree Dept] -> Tree Company
  D :: Name -> Tree Manager -> [Tree Unit]
         -> Tree Dept
  PU :: Tree Employee -> Tree Unit
  DU :: Tree Dept -> Tree Unit
  E :: Tree Person -> Tree Salary -> Tree Employee
  P :: Name -> Address -> Tree Person
  S :: Float -> Tree Salary
```

We define `compos` as in Section 4.2, and use the operations from the library of compositional operations described in Section 4.3 to implement the examples.

```
increase :: Float -> Tree c -> Tree c
increase k c = case c of
  S s -> S (s * (1+k))
  _ -> composOp (increase k) c
```

Here is the richer traversal example:

```
incrOne :: Name -> Float -> Tree c -> Tree c
incrOne d k c = case c of
  D n _ _ | n == d -> increase k c
  _ -> composOp (incrOne d k) c
```

Query functions are also easy to implement:

```
salaryBill :: Tree c -> Float
salaryBill c = case c of
  S s -> s
  _ -> composOpFold 0 (+) salaryBill c
```

These examples can all be written as single functions, whereas with SYB they each consist of two or three functions. With SYB, the type class based system for type-specific cases forces functions that have specific cases for multiple types to be split into multiple definitions.

SYB is a powerful system, but for many common uses such as the examples presented here, we believe that the `composOp` approach is more intuitive and easy to use. The drawback is that the data type family has to be lifted to a GADT, and that the `compos` function must be implemented. However, this only needs to be done once, and at least the latter can be automated, either by using

BNFC, or by extending the Haskell compiler to generate instances of `Compos` (as is done for SYB).

7.1.3 Using SYB to implement compositional operations

Single data type Above we have shown how to replace simple uses of SYB with compositional operations. We will now show the opposite, and investigate to what extent the compositional operations can be reimplemented using SYB. The renaming example for the simple functional language, as shown in Section 3, looks very similar when implemented using SYB:

```
rename :: Exp -> Exp
rename e = case e of
  EAbs x b -> EAbs ("_" ++ x) (rename b)
  EVar x   -> EVar ("_" ++ x)
  _        -> gmapT (mkT rename) e
```

For the single data type case, our `composOp` and `composOpM` can be implemented with `gmapT` and `gmapM`, `composOpFold` is like `gmapQ` with a built-in fold, and our `compos` corresponds to `gfoldl`. Here are their definitions for the `Exp` type:

```
composOp :: (Exp -> Exp) -> Exp -> Exp
composOp f = gmapT (mkT f)

composOpM :: Monad m => (Exp -> m Exp)
                     -> Exp -> m Exp
composOpM f = gmapM (mkM f)

composOpFold :: b -> (b -> b -> b)
             -> (Exp -> b) -> Exp -> b
composOpFold z c f = foldl c z . gmapQ (mkQ z f)

compos :: (forall a. a -> m a)
       -> (forall a b. m (a -> b) -> m a -> m b)
       -> (Exp -> m Exp) -> Exp -> m Exp
compos r a f e = gfoldl (\x -> a x . extM r f) r e
```

The `extM` function used above has been generalized to arbitrary unary type constructors (the `extM` from SYB requires the type constructor to be in the `Monad` class).

Families of data types For the multiple data type case, it is difficult to use SYB to implement our examples with the desired type. We can implement functions with a type which is too general or too specific, for example:

```
rename :: Data a => a -> a
rename = gmapT (rename `extT` renameVar)
  where renameVar :: Var -> Var
        renameVar (V x) = V ("_" ++ x)

renameStm :: Stm -> Stm
renameStm = rename
```

What we would like to have is a `rename` function which can be applied to any abstract syntax tree, but not to things which are not abstract syntax trees. Using a family of normal Haskell data types, this restriction could be achieved by the use of a dummy type class:

```
class Data a => Tree a
instance Tree Stm
instance Tree Exp
instance Tree Var
instance Tree Typ

renameTree :: Tree a => a -> a
renameTree = rename
```

However, we would like the class `Tree` to be closed, something which is currently only achievable using hacks such as not exporting the class.

When using `composOp`, the type restriction is achieved as a side effect of lifting the family of data types into a GADT. Using a GADT to restrict the function types when using SYB is currently not practical, since current GHC versions cannot derive `Data` and `Typeable` instances automatically for GADTs.

7.1.4 Using compositional operations to implement SYB

We can also try to implement the SYB functions in terms of our functions. If we are only interested in our single data type, this works:

```
gmapT :: Data a =>
         (forall b. Data b => b -> b) -> a -> a
gmapT f = mkT (composOp f)

gmapM :: (Data a, Monad m) =>
         (forall b. Data b => b -> m b)
      -> a -> m a
gmapM f = mkM (composOpM f)

gmapQ :: Data a =>
         (forall b. Data b => b -> u) -> a -> [u]
gmapQ f =
  mkQ [] (composOpFold [] (++) ((:[]) . f))
```

Of course these functions are no longer truly generic: even though their types are the same as the SYB versions', they will only apply the function that they are given to values in the single data type `Exp`. Defining `gfoldl` turns out to be problematic, since the combining operation that `gfoldl` accepts cannot be constructed from the operations of an applicative functor.

For the type family case, it does not seem possible to use compositional operations to implement SYB operations. It is even unclear what this would mean, since type families are implemented in different ways in the two approaches.

7.1.5 Scrap Your Boilerplate conclusions

We consider the main differences between Scrap Your Boilerplate and our compositional operations to be that:

- When using SYB, no changes to the data types are required (except some type class deriving), but the way in which functions over the data types are written is changed drastically. With compositional operations on the other hand, the data type family must be lifted to a GADT, while the style in which functions are written remains more natural.
- SYB functions over multiple data types are too generic, in that they are not restricted to the type family for which they are intended.
- Our approach is a general pattern which can be translated rather directly to other programming languages and paradigms.
- Compositional operations directly abstract out the code for pattern matches, recursion and reconstruction otherwise written by hand. SYB uses runtime type representations and type casts, which makes for more genericity, at the expense of transparency and understandability.

7.2 The Tree set constructor

7.2.1 Introduction

Petersson and Synek [18] introduce a set constructor for tree types into Martin-Löf's intuitionistic type theory. Their tree types are

similar to the inductive families in for example Agda, and, for our purposes, to Haskell's GADTs. The value representation, however, is quite different. There is only one constructor for trees, and it takes as arguments the type index, the data constructor and the data constructor arguments.

Tree types are constructed by the following rule:

TREE SET FORMATION

$$\frac{\begin{array}{c} A : set \\ B(x) : set[x : A] \qquad C(x, y) : set[x : A, y : B(x)] \\ d(x, y, z) : A[x : A, y : B(x), z : C(x, y)] \qquad a : A \end{array}}{Tree(A, B, C, d, a) : set}$$

Here A is the set of names (type indices) of the mutually dependent sets. $B(x)$ is the set of constructors in the set with name x. $C(x, y)$ is the set of argument labels (or selector names) for the arguments of the constructor y in the set with name x. d is a function which assigns types to constructor arguments: for constructor y in the set with name x, $d(x, y, z)$ is the name of the set to which the argument with label z belongs. For simplicity, $\mathcal{T}(a)$ is used below, instead of $Tree(A, B, C, d, a)$.

Tree values are constructed using this rule:

TREE VALUE INTRODUCTION

$$\frac{a : A \qquad b : B(a) \qquad c(z) : \mathcal{T}(d(a, b, z))[z : C(a, b)]}{tree(a, b, c) : \mathcal{T}(a)}$$

Here a is the name of the set to which the tree belongs. b is the constructor. c is a function which assigns values to the arguments of the constructor (children of the node), where $c(z)$ is the value of the argument with label z.

Trees are eliminated using the $treerec$ constant, with the computation rule:

$$treerec(tree(a, b, c), f) \rightarrow f(a, b, c, \lambda z.\, treerec(c(z), f))$$

7.2.2 Relationship to GADTs

As we have seen above, trees are built using the single constructor $tree$, with the type, constructor and node children as arguments to $tree$. We can use this structure to represent GADT values, as long as all children are also trees. Using the constants $l_1 \ldots$ as argument labels for all constructors, we can represent GADT values in the following way:

$$\texttt{b t}_1 \ldots \texttt{t}_n :: \texttt{Tree a} \equiv$$
$$tree(a, b, \lambda z.\ \text{case } z \text{ of } \{l_1 : t_1; \ldots; l_n : t_n\})$$

For example, the value `SDecl T_int (V "x") :: Tree Stm` in our Haskell representation would be represented as the term shown below. We use "x" as syntactic sugar for some appropriate tree representation of a string.

$$\begin{array}{l} tree(Stm, SDecl, \lambda x.\ \text{case } x \text{ of } \{ \\ \qquad l_1 : tree(Typ, T\_int, \lambda y.\ \text{case } y \text{ of } \{\}); \\ \qquad l_2 : tree(Var, V, \lambda y.\ \text{case } y \text{ of } \{l_1 : \text{"}x\text{"}\}) \\ \}) \end{array}$$

7.2.3 Tree types and compositional operations

We can implement a `composOp`-equivalent in type theory by using $treerec$:

$$composOp(f, t) =$$
$$treerec(t, \lambda(a, b, c, c').\ tree(a, b, \lambda z.\ f(c(z))))$$

What makes this so easy is that all values have the same representation, and c which contains the child trees is just a function which we can compose with our function f. With this definition, we can use `composOp` like in Haskell. The code below assumes that we have wild card patterns in case expressions, and that $+\!\!+$ is a concatenation operation for whatever string representation we have.

$$\begin{array}{l} rename(t) = treerec(t, \lambda(a, b, c, c').\ \text{case } b \text{ of } \{ \\ \qquad V : tree(Var, V, \lambda l.\ \text{"\_"} +\!\!+ c(l)); \\ \qquad \_ : composOp(rename, t) \\ \}) \end{array}$$

One advantage over the Haskell solution is that we have access to both the original child values (c in the example above), and the results of the recursive calls (c' in the example above) when writing our functions. This would simplify functions which need to use the results of the recursive calls, for example the constant folding example in Section 4.4.4.

7.3 Related work in object-oriented programming

The `ComposVisitor` class looks deceptively simple, but it combines a number of features, some of which are already known in the object-oriented programming community. It does however appear that the combination which we have presented is relatively novel.

- It uses type-parameterized visitor interfaces, which can only be implemented in a few object-oriented languages. Similar parameterized visitor interfaces can be found in the Loki C++ library [1].
- It is a depth-first traversal combinator whose behavior can be overridden for each concrete class. A similar effect can be achieved by using the `BottomUp` and `Identity` combinators from Joost Visser's work on visitor combinators [21], and with the depth-first traversal function in the the Boost Graph Library [15].
- It allows modification of the data structure in a functional and compositional way. The fact that functional modification is not widely used in imperative object-oriented programming is probably the main reason why this area has not been explored further.

8. Future Work

8.1 Automatic generation of `compos` for existing types

Some way of automatically declaring new `Compos` instances for existing data types should be developed. At the moment, none of the meta-programming and generic programming tools which we have looked at support reflection over GADTs.

8.2 Applications in natural language processing

While most of the examples in this paper are related to compiler writing, we think that this technique could also be useful in natural language processing, for example in rule-based translation. One example of this would be aggregation, e.g. by transforming sentence conjunction, for example "John walks and Mary walks", to noun phrase conjunction, such as "John and Mary walk". We want to be able to do this transformation wherever sentences of this form appear in a phrase, for example in "I know that John walks and Mary walks". The transformation is done on the level of abstract syntax, and is similar to the ones for formal languages shown earlier in this paper. Since a natural language grammar may have a very large number of constructors, using `composOp` for this kind of transformation could be very beneficial. We will explore this further in the Transfer language [4], which is intended for writing functions over GF [19] abstract syntax terms. The language is dependently typed,

and has support for inductive families and automatic generation of `composOp` functions.

8.3 Tree types and generic programming

In *"Scrap Your Boilerplate" Reloaded* [9], SYB is explained in terms of a lifting of all types to a GADT. We have already seen that the tree types of Petersson and Synek [18] are a very powerful construct which can be used to represent GADTs and perform generic operations on them. It would be interesting to see to what extent generic programming systems such as Scrap Your Boilerplate can be explained using dependent type theory with these tree types.

9. Conclusions

We have presented a pattern for implementing almost compositional operations over rich data structures such as abstract syntax trees easily.

We have ourselves started to use this pattern for real implementation tasks, and we feel that it has been very successful. In the compiler for the Transfer language [4] we use a BNFC [17, 6]-generated front-end with a generated `Compos` instance for the abstract syntax. The abstract syntax has 70 constructors, and in the (still very small) compiler the various `composOp*` functions are currently used in 12 places. The typical function using `composOp*` pattern matches on between 1 and 5 of the constructors, saving hundreds of lines of code. Some of the functions include: replacing infix operator use with function calls, beta reduction, simultaneous substitution, getting the set of variables bound by a pattern, getting the free variables in an expression, assigning fresh names to all bound variables, numbering meta-variables, changing pattern equations to simple declarations using case expressions, and replacing unused variable bindings in patterns with wild cards. Furthermore, we have noticed that using compositional operations to implement a compiler makes it easy to structure it as a sequence of simple steps, without having to repeat large amounts of traversal code for each step. Modifying the abstract syntax, for example by adding new constructs to the front-end language, is also made easier since only the functions which care about this new construct need to be changed.

Acknowledgments

We would like to thank the following people for their comments on earlier versions of this work: Thierry Coquand, Bengt Nordström, Patrik Jansson, Josef Svenningsson, Sibylle Schupp, Marcin Zalewski, Andreas Priesnitz, Markus Forsberg, Alejandro Russo, and everyone who offered comments during the talks at the Chalmers CS Winter Meeting and at Galois Connections. Last but not least we would like to thank the anonymous referees who provided many valuable comments. This work has been partly funded by the EU TALK project, IST-507802.

References

[1] ALEXANDRESCU, A. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, Feb. 2001.

[2] APPEL, A. W. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[3] APPEL, A. W. *Modern Compiler Implementation in Java*, second ed. Cambridge University Press, 2002.

[4] BRINGERT, B. The Transfer programming language. `http://www.cs.chalmers.se/~aarne/GF/doc/transfer.html`.

[5] COQUAND, C. The Agda homepage. `http://www.cs.chalmers.se/~catarina/agda/`.

[6] FORSBERG, M., AND RANTA, A. BNF Converter homepage. `http://www.cs.chalmers.se/~markus/BNFC/`.

[7] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1994.

[8] HINZE, R. Generics for the masses. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, 2004), ACM Press, pp. 236–243.

[9] HINZE, R., LÖH, A., AND OLIVEIRA, B. C. D. S. "Scrap Your Boilerplate" reloaded. In *FLOPS* (2006), M. Hagiya and P. Wadler, Eds., vol. 3945 of *Lecture Notes in Computer Science*, Springer, pp. 13–29.

[10] JANSSON, P., AND JEURING, J. PolyP - a polytypic programming language extension. In *POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1997), ACM Press, pp. 470–482.

[11] JONES, M. P. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text* (London, UK, 1995), Springer-Verlag, pp. 97–136.

[12] JONES, S. P. The Haskell 98 language. *Journal of Functional Programming 13*, 1 (2003).

[13] JONES, S. P., VYTINIOTIS, D., WEIRICH, S., AND WASHBURN, G. Simple unification-based type inference for GADTs. Submitted to ICFP'06. `http://research.microsoft.com/Users/simonpj/papers/gadt/`, Apr. 2006.

[14] LÄMMEL, R., AND PEYTON JONES, S. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices 38*, 3 (Mar. 2003), 26–37. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[15] LEE, L.-Q., LUMSDAINE, A., AND SIEK, J. G. *The Boost graph library: user guide and reference manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[16] MCBRIDE, C., AND PATERSON, R. Applicative programming with effects. Submitted to Journal of Functional Programming. `http://www.soi.city.ac.uk/~ross/papers/Applicative.pdf`.

[17] PELLAUER, M., FORSBERG, M., AND RANTA, A. BNF Converter: Multilingual front-end generation from labelled BNF. Tech. Rep. 2004-09, Department of Computing Science, Chalmers University of Technology and Göteborg University, 2004. `http://www.cs.chalmers.se/~markus/tech2004.pdf`.

[18] PETERSSON, K., AND SYNEK, D. A set constructor for inductive sets in Martin-Löf's type theory. In *Category Theory and Computer Science* (1989), pp. 128–140.

[19] RANTA, A. Grammatical Framework, a type-theoretical grammar formalism. *The Journal of Functional Programming 14*, 2 (2004), 145–189. `http://www.cs.chalmers.se/~aarne/articles/gf-jfp.ps.gz`.

[20] SHEARD, T., AND PEYTON JONES, S. Template metaprogramming for Haskell. In *ACM SIGPLAN Haskell Workshop 02* (Oct. 2002), M. M. T. Chakravarty, Ed., ACM Press, pp. 1–16.

[21] VISSER, J. Visitor combination and traversal control. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications* (New York, NY, USA, 2001), ACM Press, pp. 270–282.

[22] WINSTANLEY, N., WALLACE, M., AND MEACHAM, J. The DrIFT homepage. `http://repetae.net/~john/computer/haskell/DrIFT/`.