JFP **18** (5 & 6): 567–598, 2008.
doi:10.1017/S0956796808006898 First published online 2 September 2008 Printed in the United Kingdom

A pattern for almost compositional functions

BJÖRN BRINGERT and AARNE RANTA

Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg SE-412 96, Göteborg, Sweden
(*e-mail:* {bringert,aarne}@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 all of the constructors in the type(s), except a limited number of them, 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 are given, 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. The relationships to the "Scrap Your Boilerplate" approach to generic programming, and to general tree types in dependent type theory, are investigated by reimplementing our operations using those frameworks.

1 Introduction

This paper addresses the issue of repetitive code in operations on 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 that you have an abstract syntax definition with many syntactic types such as statement, expression, and variable.

1. Write a function that prepends an underscore to the names of all variables in a program. Do this with a case expression that has just two branches: one for the variables, and another for the rest.
2. Write a function that gives unique names to all variables in a program. Use only three cases: variable bindings, variable uses, and the rest.
3. Write a function that constructs a symbol table containing all variables declared in a program, and the type of each variable. Do this with only two cases: one for declarations, another for the rest.
4. Write a function that replaces increment statements with the corresponding assignments. Use only two cases: one for increments, and another 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.
- The need for complete traversal code for the whole family of data types in every function can 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 those given earlier 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 in plain Haskell 98 (Peyton Jones, 2003a). 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 (Martin-Löf, 1984; Nordström *et al.*, 1990) with inductive families (Dybjer, 1994) and in Haskell with generalized algebraic data types (Peyton Jones *et al.*, 2006; Augustsson & Petersson, 1994) and rank-2 polymorphism (Leivant, 1983; Peyton Jones *et al.*, 2007). We then prove some properties of our compositional operations, using the laws for applicative functors (McBride & Paterson, 2008). We go on to express the pattern in Java (Gosling *et al.*, 2005) using a variant of the Visitor design pattern (Gamma *et al.*, 1995). We also briefly describe some tools which can be used to automate the process of writing the necessary support code for a given data type. Finally, we discuss some related work in generic programming, type theory, object-oriented programming, and compiler construction, 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 Haskell type defines the abstract

syntax of the 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 (1997) book on compiler construction in ML.

```
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 that 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 (Ranta, 2004), the Exp type has 30 constructors, and *composOp* is used in more than 20 functions, 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 *composFold*.

```
compossFold :: Monoid o ⇒ (Exp → o) → Exp → o
compossFold f e = case e of
                    EAbs x b → f b
                    EApp c a → f c ⊕ f a
                    _        → ∅
```

This function takes an argument which maps terms to a monoid, and combines the results. The Monoid class requires an identity element $\emptyset$, which we return for leaf nodes, and an associative operation ($\oplus$), which we use to combine results from nodes with more than one child.

```
class Monoid a where
  ∅ :: a
  (⊕) :: a → a → a
```

Using *composFold* we can now, for example, write a function which gets the names of all free variables in an expression.

```
free :: Exp → Set String
free e = case e of
           EAbs x b → free b \ {x}
           EVar x   → {x}
           _        → composFold free e
```

This example uses a Set type with the operations $\setminus$ (set minus), $\{\cdot\}$ (singleton set), $\varnothing$ (empty set), and $\cup$ (union), with a Monoid instance such that $\emptyset = \varnothing$ and $(\oplus) = \cup$.

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.

```
composM :: Monad m ⇒ (Exp → m Exp) → Exp → m Exp
composM f e = case e of
  EAbs x b → f b >>= (λb' → return (EAbs x b'))
  EApp c a → f c >>= (λc' → f a >>= (λa' → return (EApp c' a')))
  _        → return e
```

Here, we are using the Monad type class (Peyton Jones, 2003b).

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

If we want to maintain some state across the computation over the tree, we can use *composM* with a state monad (Jones, 1995). In the following example, we will use a state monad State with these operations:

```
readState  :: State s 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 :: [(String, String)] → Exp → State [String] Exp
        f vs t = case t of
                   EAbs x b → do x' : ns ← readState
                                 writeState ns
                                 b' ← f ((x, x') : vs) b
                                 return (EAbs x' b')
                   EVar x   → return (EVar (findWithDefault x x vs))
                   _        → composM (f vs) t
findWithDefault :: Eq a ⇒ b → a → [(a, b)] → b
findWithDefault d _ [] = d
findWithDefault d k ((x, y) : xs) | x == k  = y
                                  | otherwise = findWithDefault d k xs
```

3.2 *Generalizing composOp, composM and composFold*

The three functions which we have introduced earlier, henceforth referred to as *compositional operations*, share a common structure which we will now reveal. McBride and Paterson (2008) introduce *applicative functors*, which generalize monads. An applicative functor has two operations, *pure* and ⊛.

```
class Applicative f where
  pure :: a → f a
  (⊛)  :: f (a → b) → f a → f b
```

The *pure* operation corresponds to the *return* operation of a monad, and ⊛ corresponds to *ap*, which can be defined using ≫=.

```
ap :: Monad m ⇒ m (a → b) → m a → m b
ap mf mx = mf ≫= λf → mx ≫= λx → return (f x)
```

We can rewrite *composM* to use *ap* instead of ≫=.

```
composM :: Monad m ⇒ (Exp → m Exp) → Exp → m Exp
composM 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
```

Since *composM* uses only *return* and *ap*, it actually works on all applicative functors, not just on monads. We call this generalized version *compos*.

```
compos :: Applicative f ⇒ (Exp → f Exp) → Exp → f Exp
compos f e = case e of
               EAbs x b → pure EAbs ⊛ pure x ⊛ f b
               EApp g h → pure EApp ⊛ f g ⊛ f h
               _        → pure e
```

By using wrapper types with appropriate Applicative instances (McBride & Paterson, 2008), we can now define *composOp*, *composM*, and *composFold* in terms of *compos*. The definitions of *composOp* and *composFold* are identical to McBride and Paterson's definitions of *fmap* and *accumulate* in terms of *traverse*, and the definition of *composM* follows directly from the relationship between applicative functors and monads.

```
composOp :: (Exp → Exp) → Exp → Exp
composOp f = runIdentity ∘ compos (Identity ∘ f)

newtype Identity a = Identity {runIdentity :: a}

instance Applicative Identity where
  pure                    = Identity
  Identity f ⊛ Identity x = Identity (f x)

composM :: Monad m ⇒ (Exp → m Exp) → Exp → m Exp
composM f = unwrapMonad ∘ compos (WrapMonad ∘ f)

newtype WrappedMonad m a = WrapMonad {unwrapMonad :: m a}

instance Monad m ⇒ Applicative (WrappedMonad m) where
  pure                        = WrapMonad ∘ return
  WrapMonad f ⊛ WrapMonad v = WrapMonad (f `ap` v)

composFold :: Monoid o ⇒ (Exp → o) → Exp → o
composFold f = getConst ∘ compos (Const ∘ f)

newtype Const a b = Const {getConst :: a}

instance Monoid m ⇒ Applicative (Const m) where
  pure _              = Const ∅
  Const f ⊛ Const v = Const (f ⊕ v)
```

Further compositional operations, such as *composM_*, can be defined by using other wrapper types.

```
composM_ :: Monad m ⇒ (Exp → m ()) → Exp → m ()
composM_ f = unwrapMonad_ ∘ composFold (WrapMonad_ ∘ f)

newtype WrappedMonad_ m = WrapMonad_ {unwrapMonad_ :: m ()}

instance Monad m ⇒ Monoid (WrappedMonad_ m) where
  ∅                           = WrapMonad_ (return ())
  WrapMonad_ x ⊕ WrapMonad_ y = WrapMonad_ (x ≫ y)
```

4 Systems of data types

4.1 Several algebraic 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 (such as assignments and 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 = TInt | TFloat
```

We now need one *compos* function for each recursive type, and some of the recursive calls must be made on terms which have types different from those on which the function was called on. This can be solved by taking several functions as arguments, one for each type.

```
composStm :: Applicative f ⇒
             (Stm → f Stm, Exp → f Exp, Var → f Var, Typ → f Typ)
             → Stm → f Stm
composStm (fs,fe,fv,ft) s = case s of
                              SDecl x y  → pure SDecl  ⊛ ft x ⊛ fv y
                              SAss x y   → pure SAss   ⊛ fv x ⊛ fe y
                              SBlock xs  → pure SBlock ⊛ traverse fs xs
                              SReturn x  → pure SReturn ⊛ fe x
composExp :: Applicative f ⇒
             (Stm → f Stm, Exp → f Exp, Var → f Var, Typ → f Typ)
             → Exp → f Exp
composExp (fs,fe,fv,ft) e = case e of
                              EAdd x y → pure EAdd ⊛ fe x ⊛ fe y
                              EStm x   → pure EStm ⊛ fs x
                              EVar x   → pure EVar ⊛ fv x
```

Note that the Typ function is not actually required in *composExp*, but we include it here for the sake of uniformity. We would also need to implement *composOp*, *composM*, *composFold* etc. for each of the types. Even though these implementations would be identical for all type families, it is difficult to provide generic implementations of them without resorting to multiparameter type classes and functional dependencies, since the type of the function tuple will depend on the type family. With these functions, we can define a renaming function more easily than without *composOp*.

```
renameStm :: Stm → Stm
renameStm t = composOpStm (renameStm, renameExp,
                           renameVar, renameTyp)
```

```
renameExp :: Exp → Exp
renameExp t = composOpExp (renameStm, renameExp,
                           renameVar, renameTyp)
renameVar :: Var → Var
renameVar (V x) = V ("_" ++ x)
renameTyp :: Typ → Typ
renameTyp t = t
```

We now need up to one extra function per type (for nonrecursive types we can get away with passing *id*). In a large system, this can result in significant overhead. For example, the abstract syntax used in the Glasgow Haskell Compiler contains more than 50 data types (Peyton Jones, 2007).

4.2 Categories and trees

An alternative to separate mutual data types for abstract syntax is to define just one type Tree, whose constructors take Trees 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 | TInt | TFloat
```

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

A solution to this problem is provided by dependent types (Martin-Löf, 1984; Nordström *et al.*, 1990). Instead of a constant type Tree, we define an **inductive family** (Dybjer, 1994) Tree *c*, indexed by a **category** *c*. The category is just a label to distinguish between different types of trees. Inductive families have previously been used for representing the abstract syntax of **well-typed expressions**: the family Exp *a* gives separate, yet related, types to integer expressions, boolean expressions, etc. (Augustsson & Petersson, 1994). The extension from such a family to one comprising all syntactic categories (expressions, statements, etc.) seems to be a novelty of our work. We must now leave standard Haskell and use a Haskell-like language with dependent types and inductive families. Agda (Coquand & Coquand, 1999; Norell, 2007) 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 later, 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 version 6.4 of the Glasgow Haskell Compiler (GHC): **Generalized Algebraic Data Types** (GADTs) (Augustsson & Petersson, 1994; Peyton Jones *et al.*, 2006). 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
  TInt    :: Tree Typ
  TFloat  :: 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 targeting this type, $\bot$ is the only element in it.

4.3 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 tree-traversing programs concisely.

```
compos :: Applicative f ⇒ (∀a. Tree a → f (Tree a)) → Tree c → f (Tree c)
compos f t = case t of
               SDecl x y  → pure SDecl  ⊛ f x ⊛ f y
               SAss x y   → pure SAss   ⊛ f x ⊛ f y
               SBlock xs  → pure SBlock ⊛ traverse f xs
               SReturn x  → pure SReturn ⊛ f x
               EAdd x y   → pure EAdd   ⊛ f x ⊛ f y
               EStm x     → pure EStm   ⊛ f x
               EVar x     → pure EVar   ⊛ f x
               _          → pure t
```

The first argument must now be polymorphic, since it is applied to subtrees of different types. This requires rank-2 polymorphism (Leivant, 1983; Peyton Jones

```
class Compos t where
  compos :: Applicative f => (forall a . t a -> f (t a)) -> t c -> f (t c)

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

composFold :: (Monoid o, Compos t) => (forall a . t a -> o) -> t c -> o
composFold f = getConst . compos (Const . f)

composM :: (Compos t, Monad m) => (forall a . t a -> m (t a)) -> t c -> m (t c)
composM f = unwrapMonad . compos (WrapMonad . f)

composM_ :: (Compos t, Monad m) => (forall a . t a -> m ()) -> t c -> m ()
composM_ f = unwrapMonad_ . composFold (WrapMonad_ . f)
```

Fig. 1. The Compos module.

et al., 2007), a widely supported Haskell extension. The argument to the SBlock constructor is a list of statements, which we handle by visiting the list elements from left to right, using the *traverse* function (McBride & Paterson, 2008), which generalizes *mapM*.

```
traverse :: Applicative f => (a -> f b) -> [a] -> f [b]
traverse f []       = pure []
traverse f (x : xs) = pure (:) <*> f x <*> traverse f xs
```

The other compositional operations are special cases of *compos* in the same way as before.

4.4 A library of compositional operations

Since all the other compositional operations can be defined in terms of *compos*, we create a type class containing the *compos* function, and define the other operations in terms of it. The code for this is shown in Figure 1.

4.5 Migrating existing programs

Replacing a family of data types with a generalized algebraic data type (GADT) does not change the appearance of the expressions and patterns in the syntax tree types. However, the types now have the form Tree *c*. If we want, we can give the dummy types names other than those of the original categories, 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_
```

This allows us to modify existing programs to switch from a family of data types to a GADT, simply by replacing the type definitions. All existing functions remain valid with the new type definitions, which makes it possible to take advantage of our compositional operations when writing new functions, without being forced to change any existing ones. There are a few minor issues: the limitations on type inference for GADTs (Peyton Jones *et al.*, 2006) and rank-2 polymorphism (Peyton Jones *et al.*, 2007) may require type signatures for some functions, and since GHC does not currently support type class instance deriving for GADTs, we have to write instances of common type classes such as Show and Eq for our type family by hand.

4.6 Examples

4.6.1 Example: Variable renaming

It is laborious to define a renaming function for the original Haskell definition with separate data types (as shown in Section 4.1), but now it is easy.

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

4.6.2 Example: Symbol table construction

This function constructs a variable symbol table by folding over the syntax tree. We use the Monoid instance for lists, where the associative operation is ++, and the identity element is [].

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

4.6.3 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 that you want to add increment statements. This means a new branch in the definition of Tree c from Section 4.2.

```
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
```

4.6.4 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 *composM_* (like *composM* but without a tree result, see Section 3.2 for its definition).

```
warnAssign :: Tree c → IO ()
warnAssign t = case t of
                 SAss _ _ → putChar (chr 7)
                 _        → composM_ warnAssign t
```

4.6.5 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 the pattern shown earlier. 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.

4.7 *Writing* Compos *instances*

Until now, we have only shown *compos* functions for example data types. But what is the general pattern? We will consider types of the following form

```
data T : * → * where
  C1 :: t1,1 → ··· → t1,a1 → T c1
  ⋮
  Cn :: tn,1 → ··· → tn,an → T cn
```

where $n \geqslant 0$ is the number of data constructors, $a_x \geqslant 0$ is the arity of data constructor C_x, $t_{x,y}$ is the type of argument y of constructor C_x, and c_x is the type argument to T in the type of constructor C_x. The argument types $t_{x,y}$ cannot be

type variables, since it must be possible to determine statically whether or not each argument belongs to the type family T. All Compos instances have the following general form:

```
instance Compos T where
  compos f t = case t of
                 C1 b1 ... ba1 → pure C1 ⊛ g1,1 b1 ⊛ ··· ⊛ g1,a1 ba1
                 ⋮
                 Cn b1 ... ban → pure Cn ⊛ gn,1 b1 ⊛ ··· ⊛ gn,an ban
```

where each $g_{x,y}$ function depends on the type $t_{x,y}$ of the corresponding constructor argument. There is some freedom in how $g_{x,y}$ is chosen. The simplest choice is to only use f on children which have a type in the type family T:

$$g_{x,y} = \begin{cases} f & \text{if } \exists c.t_{x,y} = \mathsf{T}\ c \\ \mathit{pure} & \text{otherwise.} \end{cases}$$

In the *compos* implementation shown in Section 4.3, we used *traverse* to map f over any lists containing elements in the type family T. This can be generalized to any traversable type, using the Traversable type class by McBride and Paterson (2008).

$$g_{x,y} = \begin{cases} f & \text{if } \exists c.t_{x,y} = \mathsf{T}\ c \\ \mathit{traverse}\ f & \text{if } \exists c.t_{x,y} = \mathsf{F}\ (\mathsf{T}\ c) \wedge \mathsf{Traversable\ F} \\ \mathit{pure} & \text{otherwise} \end{cases}$$

4.7.1 Parameterized abstract syntax

We may want to have type parameters for the entire type family. For example, GHC's abstract syntax is parameterized over the type of identifiers. This makes it possible to use the same abstract syntax, with different identifier types, for the input before and after name resolution. We can add extra type parameters to our type family to support this. For example,

```
data Decl
data Exp
data Tree :: * → * → * where
  Decl :: i → Tree i Exp → Tree i Decl
  App  :: Tree i Exp → Tree i Exp → Tree i Exp
  Var  :: i → Tree i Exp.
```

Earlier, we said that constructors should not have type variable arguments, but when we implement *compos*, we can choose to treat i as a non-Tree type.

4.7.2 An optimization

As done in the *compos* implementations in Sections 3.2 and 4.3, the cases for all nonrecursive constructors (i.e., constructors C_x such that $\forall y.g_{x,y} = \mathit{pure}$) can be optimized to a single catch-all case: $\_ \rightarrow \mathit{pure}\ t$. This can be done since

$pure\ \mathsf{C}_x \circledast pure\ b_1 \circledast \cdots \circledast pure\ b_{a_x} = pure\ (\mathsf{C}_x\ b_1 \ldots b_{a_x})$ by the homomorphism law for applicative functors (see Section 4.8).

4.8 Properties of compositional operations

The following laws hold for our compositional operations:

Identity 1	$compos\ pure = pure$
Identity 2	$composOp\ id = id$
Identity 3	$composFold\ (\lambda\_ \to \emptyset) = \lambda\_ \to \emptyset$
Composition	$composOp\ f \circ composOp\ g = composOp\ (f \circ g)$.

Here, $=$ denotes extensional function equality at some type T for which we have defined *compos* according to the scheme shown in Section 4.7. That is, $f = g$ means that for all total values $t :: \mathsf{T}$, $f\ t = g\ t$. In the proofs, we will make use of the following laws for applicative functors (McBride & Paterson, 2008):

Identity	$pure\ id \circledast u = u$
Composition	$pure\ (\circ) \circledast u \circledast v \circledast w = u \circledast (v \circledast w)$
Homomorphism	$pure\ f \circledast pure\ x = pure\ (f\ x)$
Interchange	$u \circledast pure\ x = pure\ (\lambda f \to f\ x) \circledast u$.

We would like *compos* to have the property that it does not modify the term on its own, i.e.,

Theorem 1
For all total values $t :: \mathsf{T}$, $compos\ pure\ t = pure\ t$.

Proof
Consider some $t = \mathsf{C}\ t_1 \ldots t_n$, where C is an arbitrary constructor of T with arity n. The relevant part of the *compos* function is then

$$compos\ f\ t = \textbf{case}\ t\ \textbf{of}$$
$$\mathsf{C}\ x_1 \ldots x_n \to pure\ \mathsf{C} \circledast g_1\ x_1 \circledast \cdots \circledast g_n\ x_n$$

where each g_i is either *pure*, f, or *traverse* f, depending on the type of x_i. Since $f = pure$ in the case that we are reasoning about, the functions $g_1 \ldots g_n$ are either *pure* or *traverse pure*. As noted by Gibbons and Oliveira (2006), all implementations of *traverse* should satisfy the "purity law" *traverse pure* = *pure*. Thus, all the $g_1 \ldots g_n$ functions are *pure* and all constructor cases have the following form:

$$\mathsf{C}\ x_1 \ldots x_n \to pure\ \mathsf{C} \circledast pure\ x_1 \circledast \cdots \circledast pure\ x_n$$

By the repeated use of the homomorphism law for applicative functors, we have

$$pure\ \mathsf{C} \circledast pure\ x_1 \circledast \cdots \circledast pure\ x_n = pure\ (\mathsf{C}\ x_1 \ldots x_n)$$

Thus, for all total $t : \mathsf{T}$, $compos\ pure\ t = pure\ t$. □

With the definitions of *composOp* and *composFold* given in Section 4.3, **Identity 2** and **Identity 3** follow straightforwardly from Theorem 1.

Theorem 2
For all total $t :: \mathsf{T}$, $composOp\ f\ (composOp\ g\ t) = composOp\ (f \circ g)\ t$.

Proof
Consider some $t = \mathsf{C}\ t_1 \ldots t_n$, where C is an arbitrary constructor of T, with arity n. As in the proof of Theorem 1, the interesting part of *compos* is

$$compos\ f\ t = \textbf{case}\ t\ \textbf{of}$$
$$\mathsf{C}\ x_1 \ldots x_n \rightarrow pure\ \mathsf{C} \circledast g_1\ x_1 \circledast \cdots \circledast g_n\ x_n$$

Lemma 1

$$composOp\ g\ (\mathsf{C}\ x_1 \ldots x_n) = \mathsf{C}\ (g'_1\ x_1) \ldots (g'_n\ x_n)$$

where each g'_i is id, g or $fmap\ g$, depending on the type of x_i.

Proof

$composOp\ g\ (\mathsf{C}\ x_1 \ldots x_n)$
$=$ { Definition of $composOp$ }
$runIdentity\ (compos\ (\mathsf{Identity} \circ g)\ (\mathsf{C}\ x_1 \ldots x_n))$
$=$ { Definition of $compos$ }
$runIdentity\ (pure\ \mathsf{C} \circledast g_1\ x_1 \circledast \cdots \circledast g_n\ x_n)$
$=$ { Definition of $pure$ for $\mathsf{Identity}$ }
$runIdentity\ (\mathsf{Identity}\ \mathsf{C} \circledast g_1\ x_1 \circledast \cdots \circledast g_n\ x_n)$
$=$ { Definition of $\circledast$ for $\mathsf{Identity}$ }
$runIdentity\ (\mathsf{Identity}\ (\mathsf{C}\ (runIdentity\ (g_1\ x_1))) \circledast \cdots \circledast g_n\ x_n)$
$=$ { Definition of $\circledast$ for $\mathsf{Identity}$ }
$runIdentity\ (\mathsf{Identity}\ (\mathsf{C}\ (runIdentity\ (g_1\ x_1)) \cdots (runIdentity\ (g_n\ x_n))))$
$=$ { Introduce $g'_i = runIdentity \circ g_i$ }
$runIdentity\ (\mathsf{Identity}\ (\mathsf{C}\ (g'_1\ x_1) \ldots (g'_n\ x_n)))$
$=$ { Definition of $runIdentity$ }
$\mathsf{C}\ (g'_1\ x_1) \ldots (g'_n\ x_n)$

Since each g_i is $\mathsf{Identity}$, $\mathsf{Identity} \circ g$ or $traverse\ (\mathsf{Identity} \circ g)$, each g'_i is id, g or $fmap\ g$. The last case relies on the observation by Gibbons and Oliveira (2006) that all implementations of *traverse* should satisfy $traverse\ (\mathsf{Identity} \circ f) = \mathsf{Identity} \circ fmap\ f$. □

Now,

$composOp\ f\ (composOp\ g\ (\mathsf{C}\ x_1 \ldots x_n))$
$=$ { Lemma 1 }
$composOp\ f\ (\mathsf{C}\ (g'_1\ x_1) \ldots (g'_n\ x_n))$
$=$ { Lemma 1 }
$\mathsf{C}\ (f'_1\ (g'_1\ x_1)) \ldots (f'_n\ (g'_n\ x_n))$
$=$ { Definition of $\circ$ }

$$
\begin{aligned}
&\mathsf{C}\ ((f'_1 \circ g'_1)\ x_1) \ldots ((f'_n \circ g'_n)\ x_n) \\
&= \quad \{ \text{ Lemma 1 and } \mathit{fmap}\ f \circ \mathit{fmap}\ g = \mathit{fmap}\ (f \circ g) \ \} \\
&\mathit{composOp}\ (f \circ g)\ (\mathsf{C}\ x_1 \ldots x_n)
\end{aligned}
$$

□

One may think that the stronger $\mathit{compos}\ g\ t \gg\!= \mathit{compos}\ f = \mathit{compos}\ (\lambda x \rightarrow g\ x \gg\!= f)\ t$ would hold for any Applicative type that is also a Monad, but it does not, as it changes the order of the monadic computations.

It should also be possible to perform formal reasoning about our compositional operations using dependent type theory with tree sets, as discussed in Section 7.4.

5 Almost compositional functions and the Visitor design pattern

The Visitor design pattern (Gamma *et al.*, 1995) 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 earlier 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. Our examples are written in Java 1.5 (Gosling *et al.*, 2005) and make use of its parametric polymorphism (Bracha *et al.*, 1998).

5.1 Abstract syntax representation

We use a standard encoding of abstract syntax trees in Java (Appel, 2002), along with the support code for a type-parametrized version of the Visitor design pattern. For each algebraic data type in the Haskell version (as shown in Section 4.1), 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. The Visitor interface has two type parameters: R is the type of the value returned by the Visitor, and A is the type of an auxiliary argument which is threaded through the traversal. 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, for example,

```
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 in the Haskell 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. A visitor for a given base class corresponds to a Haskell case expression on an algebraic data type. Multiple interface inheritance lets us write a single visitor which can handle multiple classes. Such a visitor is then like a case expression on an entire type family. This use of multiple interface inheritance is what makes it possible to handle the multiple-type recursion issue that forced us to use GADTs and rank-2 polymorphism in Haskell.

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 unchanged, 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 constructor argument, the ComposVisitor class in itself is essentially 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 given below implements the desugaring example from Section 4.6.3 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 do not have to do anything special to thread a state through the computation. Here, is the symbol table construction function from Section 4.6.2 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 *composFold* 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

When using the method we have described, one needs to define the Haskell Compos instance or Java ComposVisitor class manually for each type or type family. 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 (Sheard & Peyton Jones, 2002), DrIFT (Winstanley *et al.*, 2007), or Derive (Mitchell & O'Rear, 2007), but these tools do not yet support GADTs.

```
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";
```

Fig. 2. LBNF grammar for the simple imperative language.

We have added a new back-end to the Backus–Naur Form Converter (BNFC) (Forsberg, 2007; Forsberg & Ranta, 2006) 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 earlier, 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 in Figure 2. It is written in LBNF (Labeled Backus–Naur Form), the input language for BNFC.

7 Related work

7.1 Scrap Your Boilerplate

The part of this work dealing with functional programming languages can be seen as a solution to a subset of the problems solved by generic programming systems. Like "Scrap Your Boilerplate" (SYB) (Lämmel & Peyton Jones, 2003), we focus on traversal operations that make it easier to write functions over a given rich data type or set of data types when there are only a few "interesting" cases. Our approach does not aim at defining functions such as equality, hashing, or pretty-printing, which need to consider every constructor in the type or type family. We also do not address the problem of writing *polytypic functions* (Jansson & Jeuring, 1997; Hinze, 2004), that is, functions that work on any data type, even those which are yet to be defined.

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 with various traversal schemes. Data types must have instances of the Typeable and Data type classes to be used with SYB.

The original "Scrap Your Boilerplate" paper (Lämmel & Peyton Jones, 2003) 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 following 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))
```

The *everywhere* function applies a generic transformation to every node, bottom-up, and *mkT* makes a type specific transformation generic. More advanced traversal schemes are also supported. Following 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'
```

The *gmapT* function applies a generic transformation to the immediate subterms. SYB also supports queries, that is, functions that compute some result from the data structure rather than returning a modified structure. A type-specific query is made generic by *mkQ*, whose first argument is a constant that is returned for all other types. Following example computes the sum of the salaries of everyone in the company:

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

```
billS :: Salary → Float
billS (S f) = f
```

The *everything* function applies a generic query everywhere in a term, and summarizes the results using the function given as the first argument.

7.1.2 SYB examples using compositional operations

We will now show the above examples implemented using our compositional operations. We 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 described in Section 4.7, and use the operations from the library of compositional operations from Section 4.4 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 (given a Monoid instance where $\emptyset = 0$ and $(\oplus) = (+)$).

```
salaryBill :: Tree c → Float
salaryBill c = case c of
                 S s → s
                 _   → composFold salaryBill c
```

These examples can all be written as single functions, whereas with SYB each consist of two or three functions. SYB requires at least one function for each type-specific case, and one function that extends a generic traversal with the type-specific cases.

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 the Data and Typeable classes used by SYB).

7.1.3 Using SYB to implement compositional operations

Single data type: 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 *composM* can be implemented with *gmapT* and *gmapM* (a monadic version of *gmapT*). The *gmapQ* function, which returns a list of the results of applying a query to the immediate subterms, can be used to write *composFold*. Our *compos* function can be written in terms of *gfoldl*, the one SYB function which can be used to implement all the others. Their definitions for the Exp type are as follows:

```
composOp :: (Exp → Exp) → Exp → Exp
composOp f = gmapT (mkT f)

composM :: Monad m ⇒ (Exp → m Exp) → Exp → m Exp
composM f = gmapM (mkM f)

composFold :: Monoid o ⇒ (Exp → o) → Exp → o
composFold f = foldl (⊕) ∅ ∘ gmapQ (mkQ ∅ f)

compos :: Applicative f ⇒ (Exp → f Exp) → Exp → f Exp
compos f = gfoldl (λx y → x ⊛ extM pure f y) pure
```

Here the *extM* function, which adds a type-specific case to a generic transformation, has been generalized to arbitrary functors (the *extM* from SYB requires a Monad).

Families of data types: For the multiple data type case, it is difficult to use SYB to implement our examples with the desired type. When using *composOp*, the

type restriction is achieved as a by-product 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. We can implement functions with types that are too general or too specific. For example, this is too general:

```
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 that are not abstract syntax trees. With a family of normal Haskell data types, the 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.

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 as follows:

```
gmapT :: Data a ⇒ (∀b. Data b ⇒ b → b) → a → a
gmapT f = mkT (composOp f)
gmapM :: (Data a, Monad m) ⇒ (∀b ∘ Data b ⇒ b → m b) → a → m a
gmapM f = mkM (composM f)
gmapQ :: Data a ⇒ (∀b. Data b ⇒ b → u) → a → [u]
gmapQ f = mkQ [] (composFold (λx → [f x]))
```

Note that 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* requires 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 The Spine *data type*

In *"Scrap Your Boilerplate" Reloaded* (Hinze *et al.*, 2006), SYB is explained by using a GADT called Type to lift all types into a single type Spine. For our type family example, this becomes

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

data Type :: * → * where
  Stm    :: Type Stm
  Exp    :: Type Exp
  Var    :: Type Var
  Typ    :: Type Typ
  List   :: Type a → Type [a]
  Int    :: Type Int
  String :: Type String

data Typed a = a : Type a

data Spine :: * → * where
  Constr :: a → Spine a
  (◇)    :: Spine (a → b) → Typed a → Spine b
```

For example, the value EVar (V "x") is represented as Constr EVar ◇ V "x" : Var. Compared to our representation, the Spine data type only lifts the top-level (or spine) of the value, rather than the entire value. The Spine type adds another level above the existing types, instead of replacing them, which changes how values are written. It also decouples constructors from their arguments, making it impossible to do pattern matching directly. While this means that the Spine type cannot be used to replace our type family representation, it can be used to implement the SYB combinators. Thus it can be used to implement *compos* as shown in Section 7.1.3.

7.1.6 Scrap Your Boilerplate conclusions

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

- 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 pattern matching, recursion, and reconstruction code otherwise written by hand. SYB uses runtime type representations and type casts, which gives more genericity, at the expense of transparency and understandability.

7.2 Catamorphisms and folds

The *composFold* function may appear to be similar to a *catamorphism* or *fold* (Meijer *et al*., 1991). However, none of the compositional operations are recursive, as they just apply a given function to the immediate children of the current term. When using a fold, the behavior for each constructor is specified, and the recursion is done by the fold operator. With *composFold*, there is a default behavior for each constructor, and any recursion must be done explicitly.

7.3 Two-level types

Two-level types, as described by Sheard and Pasalic (2004), also address a problem that can lead to repetitive code. Their solution is to break the data type up into two levels, one level for the structures that the algorithm manipulates and one "recursive knot-tying level." The problem which the two-level types approach solves is dual to the problem described in this paper: we want to reduce the amount of repeated code when writing many similar functions over the same data type, and they want to reduce the amount of repeated code when writing the same function for many similar data types.

Using the idea of splitting a type into two levels can give us some insight into the relationship between compositional operations and *idiomatic traversals* (the term used by Gibbons (2007) to describe McBride and Paterson's (2008) *traverse* function). We split the Exp type into two levels, making Exp a fixed point of the *structure operator* E. Now *compos* becomes an idiomatic traversal, without changing anything but the type signature (and expanding the catch-all case). The intuition is that E is a container of expressions, and *compos* maps a function over the expressions that it contains. This is only done at the top level, just as a regular map on lists does not descend into any nested lists.

```
data E e = EAbs String e | EApp e e | EVar String

newtype Exp = Wrap (E Exp)

compos :: Applicative f ⇒ (a → f b) → E a → f (E b)
compos f e = case e of
               EAbs x b → pure EAbs ⊛ pure x ⊛ f b
               EApp g h → pure EApp ⊛ f g ⊛ f h
               EVar v   → pure EVar ⊛ pure v
```

We define *composOp*, *composM*, and *composFold* as before, but with different types.

```
composOp :: (a → b) → E a → E b
composM  :: Monad m ⇒ (a → m b) → E a → m (E b)
composFold :: Monoid o ⇒ (a → o) → E a → o
```

Functions such as *rename* which work on the Exp type now need to use the Wrap constructor, but apart from that, the code is unchanged.

```
rename :: Exp → Exp
rename (Wrap e) = Wrap $ case e of
                    EAbs x b → EAbs ("_" ++ x) (rename b)
                    EVar x   → EVar ("_" ++ x)
                    _        → composOp rename e
```

7.4 The Tree set constructor

7.4.1 Introduction

Petersson and Synek (1989) introduce a set constructor for tree types into Martin-Löf's (1984) intuitionistic type theory. Their tree types are similar to the inductive families in, for example, Agda (Norell, 2007), 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

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

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, $\mathscr{T}(a)$ is used, instead of $Tree(A,B,C,d,a)$.

Tree values are constructed using the following rule:

TREE VALUE INTRODUCTION

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

Here, a is the name of the set to which the tree belongs, b is the constructor, and 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 following computation rule:

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

Here, f is applied to the tree set name a, the constructor b, the children c, and the results of recursive calls on each of the children. The type of *treerec* is given by

TREE VALUE ELIMINATION

$$\frac{\begin{array}{l} D(x,t) : set[x : A, t : \mathscr{T}(x)] \qquad a : A \qquad t : \mathscr{T}(a) \\ f(x,y,z,u) : D(x, tree(x,y,z)) \qquad [x : A, y : B(x), z(v) : \mathscr{T}(d(x,y,v))[v : C(x,y)], \\ \quad u(v) : D(d(x,y,v), z(v))[v : C(x,y)]] \end{array}}{treerec(t, f) : D(a, t)}$$

7.4.2 Relationship to GADTs

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

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

For example, the value SDecl TInt (V "foo")::Tree Stm in our Haskell representation would be represented as the term shown below. We use *"string"* to stand for some appropriate tree representation of a string.

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

7.4.3 Tree types and compositional operations

We can implement a *composOp*-equivalent in type theory by using *treerec*.

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

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

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

One advantage over the Haskell solution is that *treerec* is a catamorphism for arbitrary tree types, as it gives us access not only to the original child values (c in

the above example), but also to the results of the recursive calls (c' in the above example). This would simplify functions which need to use the results of recursive calls, for example, the constant folding example in Section 4.6.5. As compositional operations are not catamorphisms (see Section 7.2), *composOp* itself does not make use of the c' argument.

7.5 Related work in object-oriented programming

The ComposVisitor class looks deceptively simple, but it has a number of features in what appears to be a novel combination:

- It uses type-parameterized visitor interfaces, which require powerful features such as C++ templates or Java generics. Similar parameterized visitor interfaces can be found in the Loki C++ library (Alexandrescu, 2001).
- It is a depth-first traversal combinator whose behavior can be overridden for each concrete class. A similar traversal can be achieved by using the BottomUp and Identity combinators from Visser's (2001) work on visitor combinators, and with the depth-first traversal function in the Boost Graph Library (Lee *et al.*, 2002).
- 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.

7.6 Nanopass framework for compiler education

The idea of structuring compilers as a large number of simple passes is central to the work on the Nanopass framework for compiler education (Sarkar *et al.*, 2005), a domain-specific language embedded in Scheme. Using the Nanopass framework, a compiler is implemented as a sequence of transformations between a number of intermediate languages, each of which is defined using a set of mutually recursive data types. Transformations are implemented by pattern matching, and a *pass expander* adds any missing cases, a role similar to that of our *composOp*.

One notable feature of the Nanopass framework is that a language can be declared to inherit from an existing language, with new constructors added or existing ones removed. This makes it possible to give more accurate types to functions which add or remove constructions, without having to define completely separate languages which differ only in the presence or absence of a few constructors. While this is a very useful feature, it is difficult to implement in languages such as Haskell or Java whose notions of data types are more rigid than Scheme's. In Haskell, we model abstract syntax with algebraic datatypes, but Haskell does not allow the extension or restriction of datatypes. In Java, we could add subclasses to encode new constructors, and create new Visitor interfaces for each set of constructors we want to handle, but this would require writing a new ComposVisitor class for each new Visitor interface.

8 Conclusions

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

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 (Bringert, 2006), we use a front-end generated by BNFC (Forsberg & Ranta, 2006; Forsberg, 2007), including a Compos instance for the abstract syntax. The abstract syntax has 70 constructors, and in the (still very small) compiler compositional operations are currently used in 12 places. The typical function that uses compositional operations 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. However, using many simple steps is likely to have a negative impact on performance, as a complete traversal is potentially done in every step. This problem could perhaps be ameliorated by developing deforestation techniques (Wadler, 1990) for compositional operations.

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, Thomas Schilling, Andres Löh, the anonymous ICFP and JFP referees, and everyone who offered comments during the talks at the Chalmers CS Winter Meeting, at Galois Connections, and at ICFP 2006. The code in this paper has been typeset using `lhs2TeX`, with the help from Andres Löh and Jeremy Gibbons. This work has been partly funded by the EU TALK project, IST-507802.

References

Alexandrescu, Andrei. (2001) *Modern C++ Design: Generic Programming and Design Patterns Applied*. Indianapolis: Addison-Wesley Professional.

Appel, Andrew W. (1997) *Modern Compiler Implementation in ML*. Cambridge, UK: Cambridge University Press.

Appel, Andrew W. (2002) *Modern Compiler Implementation in Java*, 2nd ed. Cambridge, UK: Cambridge University Press.

Augustsson, Lennart & Petersson, Kent. (1994) Silly type families. `http://www.cs.pdx.edu/sheard/papers/silly.pdf`.

Bracha, Gilad, Odersky, Martin, Stoutamire, David & Wadler, Philip. (1998) Making the future safe for the past: Adding genericity to the java programming language. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, Vancouver, BC, Chambers, Craig (ed.), pp. 183–200.

Bringert, Björn. (2006) The Transfer programming language. `http://www.cs.chalmers.se/Cs/Research/Language-technology/GF/doc/transfer.html`.

Coquand, Catarina & Coquand, Thierry. (September 1999) Structured type theory. *Workshop on Logical Frameworks and Meta-languages (LFM'99)*, Paris, France.

Dybjer, Peter. (1994) Inductive families. *Formal Aspect Comput*. **6**(4), 440–465.

Forsberg, Markus. (September 2007) *Three Tools for Language Processing: BNF Converter, Functional Morphology, and Extract*. Ph.D. Thesis, Göteborg University and Chalmers University of Technology.

Forsberg, Markus & Ranta, Aarne. (2006) BNF Converter homepage. `http://www.cs.chalmers.se/markus/BNFC/`.

Gamma, Erich, Helm, Richard, Johnson, Ralph & Vlissides, John. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley Longman Publishing Co.

Gibbons, Jeremy. (2007) Datatype-generic programming. In *Spring School on Datatype-Generic Programming*, Backhouse, Roland, Gibbons, Jeremy, Hinze, Ralf & Jeuring, Johan (eds). Lecture Notes in Computer Science, Vol. 4719. Heidelberg: Springer-Verlag.

Gibbons, Jeremy & Oliveira, Bruno C. (July 2006) The essence of the iterator pattern. In *Workshop on Mathematically-Structured Functional Programming (MSFP 2006)*, McBride, Conor & Uustalu, Tarmo (eds). Kuressaare Estonia Swindon, UK: British Computer Society.

Gosling, James, Joy, Bill, Steele, Guy & Bracha, Gilad. (2005) *Java Language Specification*, 3rd ed. Indianapolis: Addison-Wesley Professional.

Hinze, Ralf. (2004) Generics for the masses. In *ICFP '04: Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, Vol. 39. New York: ACM Press, pp. 236–243.

Hinze, Ralf, Löh, Andres & Oliveira, Bruno C. D. S. (2006) "Scrap Your Boilerplate" Reloaded. In 8th International Symposium on Functional and Logic Programming (*FLOPS 2006*), Hagiya, Masami & Wadler, Philip (eds). Lecture Notes in Computer Science, Vol. 3945. Heidelberg: Springer, pp. 13–29.

Jansson, Patrik & Jeuring, Johan. (1997) PolyP—A polytypic programming language extension. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. New York: ACM Press, pp. 470–482.

Jones, Mark P. (1995) 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: Springer-Verlag, pp. 97–136.

Lämmel, Ralf & Peyton Jones, Simon. (January 2003) Scrap Your Boilerplate: A practical design pattern for generic programming. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*. New Orleans, LA, New York: ACM.

Lee, Lie-Quan, Lumsdaine, Andrew & Siek, Jeremy G. (2002) *The Boost Graph Library: User Guide and Reference Manual*. Boston, MA: Addison-Wesley Longman Publishing Co.

Leivant, Daniel. (1983) Polymorphic type inference. In *POPL'83: Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. New York: ACM Press, pp. 88–98.

Martin-Löf, Per. (1984) *Intuitionistic Type Theory*. Naples, Italy: Bibliopolis.

McBride, Conor & Paterson, Ross. (2008) Applicative programming with effects. *J. Funct. Program.* **18**(1), 1–13.

Meijer, Erik, Fokkinga, Maarten & Paterson, Ross. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*. New York: Springer-Verlag, pp. 124–144.

Mitchell, Neil & O'Rear, Stefan. (August 2007) Data Derive: A User Manual. `http://www.cs.york.ac.uk/fp/darcs/derive/derive.htm`.

Nordström, Bengt, Petersson, Kent & Smith, Jan M. (1990) *Programming in Martin-Löf's Type Theory: An Introduction*. USA: Oxford University Press. `http://www.cs.chalmers.se/Cs/Research/Logic/book/`.

Norell, Ulf. (2007) *Towards a Practical Programming Language Based on Dependent Type Theory*. Ph.D. Thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden.

Petersson, Kent & Synek, Dan. (1989) A set constructor for inductive sets in Martin-Löf's type theory. In *Category Theory and Computer Science*. Lecture Notes in Computer Science, Vol. 389. Heidelberg: Springer, pp. 128–140.

Peyton Jones, Simon. (2003a) The Haskell 98 language. *J. Funct. Program.* **13**(1), 1–146.

Peyton Jones, Simon. (2003b) The Haskell 98 libraries. *J. Funct. Program.* **13**(1), 149–240.

Peyton Jones, Simon. (August 2007). The GHC Commentary. `http://hackage.haskell.org/trac/ghc/wiki/Commentary`.

Peyton Jones, Simon, Vytiniotis, Dimitrios, Weirich, Stephanie & Washburn, Geoffrey. (2006) Simple unification-based type inference for GADTs. In *ICFP'06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*. New York: ACM Press, pp. 50–61.

Peyton Jones, Simon, Vytiniotis, Dimitrios, Weirich, Stephanie & Shields, Mark. (2007) Practical type inference for arbitrary-rank types. *J. Funct. Program.* **17**(1), 1–82.

Ranta, Aarne. (2004) Grammatical framework: A type-theoretical grammar formalism. *J. Funct. Program.* **14**(2), 145–189.

Sarkar, Dipanwita, Waddell, Oscar & Dybvig, Kent R. (2005) EDUCATIONAL PEARL: A Nanopass framework for compiler education. *J. Funct. Program.* **15**(5), 653–667.

Sheard, Tim & Pasalic, Emir. (2004) Two-level types and parameterized modules. *J. Funct. Program.* **14**(5), 547–587.

Sheard, Tim & Peyton Jones, Simon. (2002) Template meta-programming for Haskell. In *Haskell'02: Proceedings of the ACM SIGPLAN workshop on Haskell*. New York: ACM Press, pp. 1–16.

Visser, Joost. (2001) Visitor combination and traversal control. In *OOPSLA'01: Proceedings of the 16th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, Vol. 36. New York: ACM Press, pp. 270–280.

Wadler, Philip. (1990) Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.* **73**(2), 231–248.

Winstanley, Noel, Wallace, Malcom & Meacham, John. (2007) The DrIFT homepage. `http://repetae.net/ john/computer/haskell/DrIFT/`.