Almost Compositional Functions

Björn Bringert and Aarne Ranta
{bringert,aarne}@cs.chalmers.se

Department of Computer Science and Engineering

Chalmers University of Technology
and Göteborg University

The problem: Boring tree traversals

An abstract syntax tree type:

```
data Exp = EAbs String Exp | EApp Exp Exp
  | EVar String | EAdd Exp Exp
  | EMul Exp Exp | EInt Int
```

Add “X” to all variable names:

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

Boring code

The solution: Abstraction

Apply a function to the children of all nodes:

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

Example: Renaming

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

Boring code

Some other examples

- Substitute a term for a variable.
- Syntactic desugaring.
- Constant folding (e.g. replace *2 + 5* with *7*).

Making the problem more difficult

We often have more than one syntactic category:

```
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
```

Masochist's rename

```
renameStm :: Stm -> Stm
renameStm s = case s of
  SDecl t v -> SDecl t (renameVar v)
  SAss v e  -> SAss (renameVar v) (renameExp e)
  SBlock ss -> SBlock (map renameStm ss)
  SReturn e -> SReturn (renameExp e)

renameExp :: Exp -> Exp
renameExp e = case e of
  EAdd e1 e2 -> EAdd (renameExp e1) (renameExp e2)
  EStm s     -> EStm (renameStm s)
  EVar v     -> EVar (renameVar v)
```

```
renameVar :: Var -> Var
renameVar (V x) = V (x ++ "X")
```

Abstract Syntax with GADTs

Dummy types for categories:

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

The family of syntax tree types:

```
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
```

GADT composOp

A function which can be applied to any syntax tree.

```
composOp :: (forall a. Tree a -> Tree a)
         -> Tree c -> Tree c

composOp f t = case t of
      SDecl typ var  -> SDecl (f typ) (f var)
      SAss var exp   -> SAss (f var) (f exp)
      SBlock stms    -> SBlock (map f stms)
      SReturn exp    -> SReturn (f exp)
      EAdd exp1 exp2 -> EAdd (f exp1) (f exp2)
      EStm stm       -> EStm (f stm)
      EVar var       -> EVar (f var)
      _              -> t
```

A slightly shorter rename

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

Generalizing composOp

- Only simple tree transformations so far.
- Maybe we need to return something else?
- Maybe we need some state?
- Maybe we want to beep once in a while?
- We can make other composOp-like functions.

Compositional folding

When the function does not change the tree:

Result for leaves — `b`; Combine child results — `(b -> b -> b)`

```
composOpFold :: b -> (b -> b -> b)
  -> (forall a. Tree a -> b) -> Tree c -> b
```

Example: Free variables

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

Monadic composOp

When the action changes the tree:

```
composOpM :: Monad m =>
     (forall a. Tree a -> m (Tree a))
  -> Tree c -> m (Tree c)
```

When the action doesn't change the tree:

```
composOpM_ :: Monad m =>
     (forall a. Tree a -> m ())
  -> Tree c -> m ()
```

Examples of composOpM

Example: Beep on assignment

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

Other examples: fresh variables names, failure

Most general composOp

We can express all the composOp* functions with:

The operations of an applicative functor,
Conor McBride and Ross Paterson,
Applicative Programming with Effects.

```
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)
```

Java: Boring traversal code

- Example: Build a symbol table

```
class BuildSymTab implements Stm.Visitor<SymTab> {
  public Stm visit(SDecl d, SymTab tab) {
    tab.put(d.var_, d.typ_);
    return d;
  }
  public Stm visit(SAss p, Map<Var,Typ> arg) {
    Var var_ = p.var_.accept(this, arg);
    Exp exp_ = p.exp_.accept(this, arg);
    return new SAss(var_, exp_);
  }
  ... lots of similar cases ...
}
```

Java: ComposVisitor

- A visitor which visits all the children and reconstructs each node:

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

  public Stm visit(SAss p, A arg) {
    Var var_ = p.var_.accept(this, arg);
    Exp exp_ = p.exp_.accept(this, arg);
    return new SAss(var_, exp_);
  }
  ...
}
```

Handles all categories

Java: Using ComposVisitor

Extend ComposVisitor, override interesting cases.

Example: Build a symbol table

```
class BuildSymTab extends ComposVisitor<SymTab> {
  public Stm visit(SDecl d, SymTab tab) {
    tab.put(d.var_, d.typ_);
    return d; } }
```

Example: Convert increments to assignments

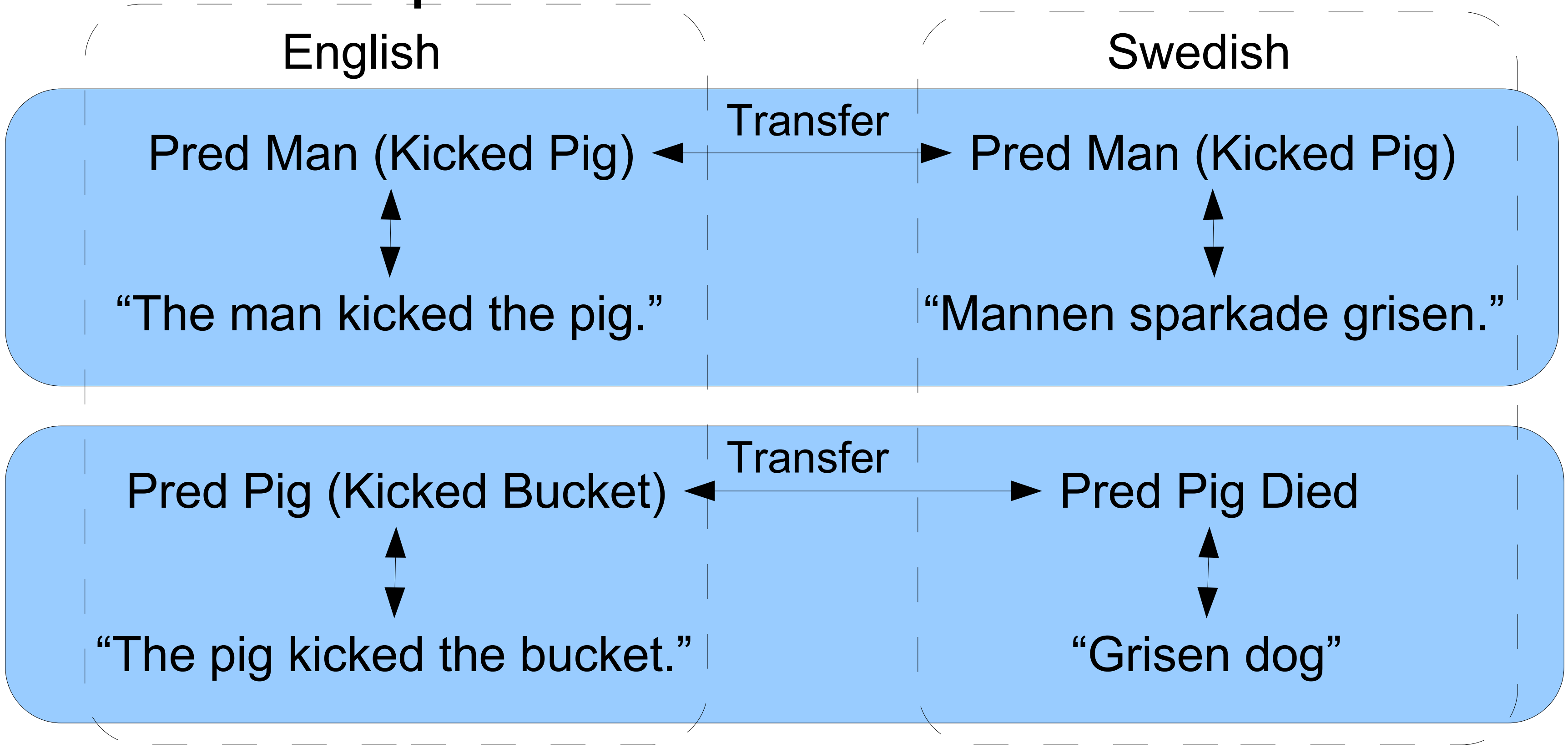
```
class Desugar extends ComposVisitor<Object> {
  public Stm visit(SInc i, Object arg) {
    Exp rhs = new EAdd(new Evar(i.var_), new EInt(1));
    return new SAss(i.var_, rhs); } }
```

BNFC support for composOp

- The BNF Converter produces abstract syntax, lexer, parser and pretty printer from a BNF grammar.
- We have extended BNFC:
  - There is a new Haskell GADT back-end, which generates abstract syntax with composOp* functions.
  - The Java 1.5 back-end now generates a ComposVisitor.

Natural Language Applications

- We can use composOp to translate between languages which use different structures for the same concept:



Kicking the bucket: Grammar

Abstract syntax

```
cat S; NP; VP;
fun Pred : NP -> VP -> S;
    Man : NP;
    Pig : NP;
    Bucket : NP;
    Died : VP;
    Kicked : NP -> VP;
    Ate : NP -> VP;
```

English concrete syntax

```
lin Pred x y = {s = x.s ++ y.s};
    Man      = {s = ["the man"]};
    Pig      = {s = ["the pig"]};
    Bucket   = {s = ["the bucket"]};
    Died     = {s = "died"};
    Kicked x = {s = "kicked" ++ x.s};
    Ate x    = {s = "ate" ++ x.s};
```

Kicking the bucket: Transfer

Generated from the grammar

```
data Cat : Type where {NP:Cat; S:Cat; VP:Cat}

data Tree : Cat -> Type where
  Pred : Tree NP -> Tree VP -> Tree S
  Man : Tree NP
  Pig : Tree NP
  Bucket : Tree NP
  Died : Tree VP
  Kicked : Tree NP -> Tree VP
  Ate : Tree NP -> Tree VP
derive Compos Tree
```

Create composOp automagically

Might be hidden in the future

```
translate : (C : Cat) -> Tree C -> Tree C
translate _ t = case t of
     Kicked Bucket -> Died
     _ -> composOp ? ? compos_Tree ? translate t
```

Related Work

- Scrap Your Boilerplate,
  by Ralf Lämmel and Simon Peyton Jones
  - More general, less intuitive.
  - Requires a type cast operator.
  - SYB: normal types, strange functions.
  - composOp: lift types to GADT, normal functions.
- Tree Sets,
  by Kent Peterson and Dan Synek
- Applicative Programming with Effects, Conor McBride and Ross Paterson.

Future Work

- Generate traversal functions for existing Haskell types automatically.
- Try some more natural language examples.
- Implement in other programming languages?

Conclusions

- Makes writing and maintaining tree processing programs easier.
  - Reduces amount of boilerplate code.
  - When adding new constructors, only functions that care about them need to be changed.
- Works in multiple programming languages.
- Integrated into BNFC.