Generics in Small Doses

Adam T. Sampson   Neil C. C. Brown

Computing Laboratory, University of Kent

Fun in the Afternoon, November 2008

Overview

- Tock, our application
- Why generic programming?
- What's wrong with the existing generics systems?
- What we've done to fix them

Tock

- A compiler for concurrent imperative programming languages
- Written in Haskell
  - Lots of expertise here, and good for student projects
  - Many existing compilers in Haskell
- Uses a nanopass approach

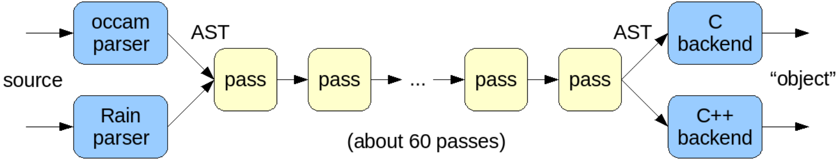
Nanopass compilation (Sarkar et al., 2004)

- Build a compiler as lots of little passes, each of which does one thing to the AST
- Various types of passes:

Simplifications	e.g. “remove multiple assignment”
Restructurings	e.g. “group variable definitions”
Annotations	e.g. “mark parallel usage of channels”
Checks	type-checking, internal consistency

- Easier to write, extend, test…and teach

Structure of Tock



Representing the AST

- Tock's AST is quite complex, since it needs to represent all the intermediate stages too
  - Other nanopass toolkits are in dynamic languages...
- We use algebraic data types
  - 37 data types, 160+ constructors

```
data Process = Seq [Process]
             | Assign [(Variable, Expression)]
             | ...
data Expression = DyadicOp Op Expression Expression
                | ExprVariable Variable
                | ...
data Variable = Variable String
...
```

Writing passes

- A pass is a function from AST to AST
- For example, let's write a pass that converts `occam.style.names` to `c_style_names`

```
cStyleNames :: AST -> AST
cStyleNames = ...
  where
    doName :: Name -> Name
    doName (Name s)
      = Name [if c == '.' then '_' else c | c <- s]
```

- How do we apply doName to all the Names in the AST?

Generics

- This is a job for a generic programming toolkit
- A generics system will let you take type-specific functions, and apply them wherever they match inside a more complex data structure
  - i.e. turn a *type-specific* function into a *generic* function
- There are many existing generics systems for Haskell...

Scrap Your Boilerplate (Lämmel/Peyton-Jones, 2003)

```
cStyleNames :: AST -> AST
cStyleNames = everywhere (mkT doName)
```

- We started out using SYB, because it's included with GHC as Data.Generics
- It's pretty easy to use, and lets you easily build custom traversals
- Unfortunately, it's *very* slow:
  - It works by runtime type introspection
  - Its traversals don't do any pruning, so it'll look at every Char of every String to see if it's a Name

Uniplate (Mitchell/Runciman, 2007)

```
cStyleNames :: AST -> AST
cStyleNames = transform doName
```

- Designed for compiler applications
- Provides a wide variety of ready-made traversal functions
- Works using a primitive defined in a typeclass

```
class Biplate outer inner where
  biplate :: outer -> ([inner], [inner] -> outer)
```

- biplate lets you operate upon the biggest inners in an outer
  - From this, you can build all the higher-level operations
- Much faster – no runtime typing

So why not just use Uniplate?

- Uniplate doesn't support generic operations with more than one target type
  - e.g. matching Processes and Expressions
- This is a problem for us – we have several passes that need to do this
- Can we extend the Biplate primitive to support multiple target types?
  - Yes: we've called it Polyplate

University of Kent | Computing

Operation sets

- We need to be able to build sets of type-specific functions ("operations")
- . . . and we need to be able to parameterise a typeclass over the type of a set of operations
- So we use a standard type-level programming trick. . .
- The empty set of operations is the unit type:

```
type BaseOp = ()

baseOp :: BaseOp
baseOp = ()
```

Operation sets

- We then add type-specific functions to the set by nesting tuples:

```
type Transform t = t -> t

type ExtOp op t = (Transform t, op)

extOp :: op -> Transform t -> ExtOp op t
extOp ops f = (f, ops)
```

University of Kent | Computing

Operation sets

- There's a nice symmetry between the functions used to build an operation set and its type
- Here's an operation set with type-specific functions for Process and Expression

```
myOp :: BaseOp `ExtOp` Process `ExtOp` Expression
myOp = baseOp `extOp` doProcess `extOp` doExpression
```

(in practice the type can usually be inferred)

The Polyplate typeclass

```
class Polyplate ops tops t where
  polyplate :: ops -> tops -> Bool -> t -> t
```

- polyplate applies the type-specific functions in its operation set to the *largest subtrees* of the appropriate types within a value of type t
- If no functions match, it behaves like the identity function
- It takes two sets of operations:
  - ops to apply to the current value;
  - tops to apply to children of the value when recursing into it
- I'll come back to the Bool flag in a minute; for now we'll just pass it through

An example data type

- We'll use the following pair of data types for our examples:

```
data Outer = Foo Inner | Bar
data Inner = Baz | Quux
```

- The constructors here aren't really important, but...
- Note that Outer can contain an Inner, but not vice versa

Polyplate instances: “hits”

- When the set is not empty, and the outermost type-specific function in the set can be applied to the value type, we simply apply it:

```
instance Polyplate (Transform Inner, r) tops Inner where
  polyplate (f, _) _ _ v = f v

instance Polyplate (Transform Outer, r) tops Outer where
  polyplate (f, _) _ _ v = f v
```

Polyplate instances: "misses"

- When the set is not empty, and the outermost type-specific function *cannot* be applied to the value type, then we recurse to try the next function in the set:

```
instance Polyplate r tops Inner =>
          Polyplate (Transform Outer, r) tops Inner where
  polyplate (_, rest) topOps b v
    = polyplate rest topOps b v
```

- The recursion in the typeclass constraint matches the recursion in the function itself

University of Kent | Computing

Pruning

```
class Polyplate ops tops t where
  polyplate :: ops -> tops -> Bool -> t -> t
```

- What's that Bool for?
- It's the *descent flag*
- It starts off as False
- If it becomes True while we're trying to apply our functions, then the value type t might contain one of the target types
- We use this to limit our traversal to only the values that might contain the things we're looking for

Polyplate instances: “throughs”

- . . . except in the case where we know that the value type might contain values of the type that the type-specific function is looking for – then we do the same, but we also force the descent flag to True:

```
instance Polyplate r tops Inner =>
         Polyplate (Transform Inner, r) tops Outer where
  polyplate (_, rest) topOps b v
    = polyplate rest topOps True v
```

Polyplate instances: non-trivial empty sets

- When the set of operations is empty, we know we haven't applied any type-specific functions to the current value
- We have to look at the descent flag
- If it's False, none of the types we're looking for can be contained inside this value; we can just return it
- If it's True, we have to apply polyplate recursively to the children of the value
  - ...setting the descent flag back to False

```
instance Polyplate tops tops Inner =>
         Polyplate () tops Outer where
  polyplate () _ False v = v
  polyplate () topOps True (Foo i)
    = let i' = polyplate topOps topOps False i
      in Foo i'
  polyplate () _ True Bar = Bar
```

Polyplate instances: trivial empty sets

- If the set of operations is empty and the value type has no children, we can just return it:

```
instance Polyplate () tops Inner where
  polyplate () _ _ v = v
```

Downsides

- You need lots of instances of Polyplate – $n(n-1)$ where n is the number of types you want to handle
- Fortunately, we can derive them automatically
  - We use SYB's runtime typing to detect which types can contain other types, then generate instance code
- You also need more typeclass constraints on functions using these operations than with SYB
- It takes a *very long time* to compile Polyplate code with GHC…

University of Kent | Computing

In summary…

- We've shown how the Uniplate approach to generics can be extended to allow operations involving multiple types
- This lets us replace SYB – which significantly speeds up our compiler
- I've been glossing over a lot here: ask me for the paper for the full details
  - For example, all the transformations are actually monadic…
- Any questions?

University of Kent | Computing