Generics in Small Doses: Nanopass Compilation with Haskell

Adam T. Sampson
University of Kent, UK
ats@offog.org

Neil C. C. Brown
University of Kent, UK
neil@twistedsquare.com

Abstract

Tock is a new compiler for concurrent imperative programming languages, designed using nanopass techniques. Nanopass compilers transform a program from source code to the target form through the application of a series of transformation passes. Because these passes are usually small and self-contained, the resulting compiler is highly modular, and easy to test and extend. Most existing nanopass compilers are implemented in dynamically-typed languages, but Tock is written in Haskell. We describe the generic programming interface that we have designed for building nanopass compilers in Haskell, and show how it can be implemented efficiently by combining techniques from the "Scrap Your Boilerplate" and Uniplate generic programming systems.

Categories and Subject Descriptors D.3 [*Software*]: Programming Languages

General Terms Languages, Performance

1. Introduction

The occam-π programming language (Welch and Barnes 2005) supports very large numbers of lightweight concurrent processes which communicate using channels and barriers, and ensures safety through compile-time static analysis. The existing occam-π compiler is nearly twenty years old and written in C. It is difficult to maintain and develop, and generates very poor native code for modern processors.

Tock is a new compiler for concurrent imperative languages such as occam-π and Rain (Brown 2006). It generates portable C99 or C++ code, which can be translated to efficient native object code by any standard compiler. Its primary aim is to be extremely flexible for language experimentation, particularly with a view to use in student projects.

To further this goal, Tock is a nanopass compiler (Sarkar et al. 2004): it is built from a series of small passes which operate upon an abstract syntax tree (AST) in various ways, transforming it step by step into the target language (figure 1). The types of passes found in a nanopass compiler may include:

simplifications which translate a complex language feature into a series of simpler operations (for example, turning parallel assignment into sequential assignment through temporary variables);

restructurings which move items around in the AST (for example, grouping variable declarations);

annotations which add extra information to the AST that can be used by later passes (for example, marking free variables in definitions); and

checks which ensure that properties hold on the AST (for example, type-checking).

Since passes are typically very simple, they are straightforward to write, modify and test. The resulting compiler is extremely modular and easy to navigate, and new features can be implemented by writing additional passes.

Tock is implemented using Haskell. We chose Haskell because it is the most commonly-used functional language at the University of Kent – all of our undergraduates will have at least some experience with it – and because it has been used successfully to write a number of existing compilers, such as the Glasgow Haskell Compiler (GHC), Yhc and Pugs.

Nanopass compilers are usually written using dynamically-typed languages such as Scheme. The implementation of a nanopass compiler in a statically-typed language such as Haskell raises a number of interesting questions. In this paper, we will discuss our generics-based approach to the implementation of a nanopass compiler using Haskell.

We start in section 2 with an overview of Tock's structure and major components. In section 3, we describe our experience with the "Scrap Your Boilerplate" and "Uniplate" generics systems, and give our requirements for generic operations. In section 4, we describe Tock's generic traversal interface, and we show in sections 5 and 6 how it can be implemented using approaches drawn from both generics systems.

2. An Overview of Tock

In this section, we will describe some of the design decisions that we have taken during Tock's development.

2.1 AST Representation

The representation of the AST is absolutely fundamental to the structure of a nanopass compiler, since nearly every part of the compiler is concerned with generating, searching or manipulating it. Not only must it be flexible enough to describe the user's program at every stage of its transformation, it must also be possible to operate efficiently upon it.

The simplest approach in Haskell is to use a single algebraic data type with constructors for every possible node in the AST:

```
data Node = Seq [Node]
          | Assign [(Node, Node)]
          | Plus Node Node
          | Variable String
          | ...
```

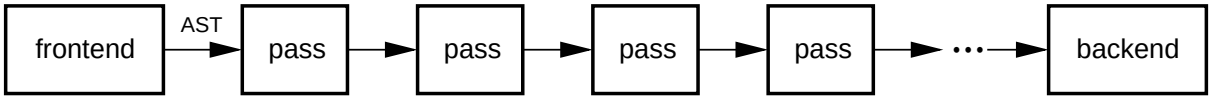

Figure 1. Basic structure of a nanopass compiler

This is somewhat analogous to the AST representation in the existing monolithic occam-π compiler, which uses a single large C union type – or to the representations generally used in dynamically-typed languages where there are no type constraints. The obvious downside is that it is possible for faulty code in the compiler to construct an invalid AST; for example, assignment to an expression. We have found that such programmer errors are common when writing and refactoring passes.

Tock's AST therefore uses a more natural Haskell representation, with a separate algebraic data type for each type of AST node:

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

This representation catches many programmer errors at compile time; it is now impossible to construct an syntactically invalid AST (although you can, of course, still construct an AST corresponding to a program that does not work).

To support annotation passes, and to simplify error reporting, most constructors in the AST have a first argument of type Meta, a record type that contains metadata about a particular node in the AST: the source position corresponding to the node, and any annotations that have been applied to it.

occam-π is a reasonably large and complex language: Tock's AST currently contains 37 algebraic data types with more than 160 constructors between them.

2.2 Monadic Operations

Tock makes heavy use of monads and monad transformers (Jones 1995). Nearly all of Tock runs in the PassM monad, which is defined as:

```
type PassM = ErrorT ErrorReport (StateT CompilerState IO)
```

ErrorReport is the representation of an error message, containing a source position and a string. This allows errors to be handled consistently in a single location for all parts of Tock. CompilerState is a record containing compiler options, name definitions, and various other state useful to all passes. IO is included in the stack because several passes need access to files – and because being able to call putStrLn is sometimes useful as a last resort when debugging.

Some parts of Tock need to track local state, accumulate lists of output, and so on. This can be achieved by stacking additional monad transformers on top of PassM:

```
type AnalyseM = StateT (Map String FunctionInfo) PassM
```

A type class is provided to allow access to PassM's facilities from other monads stacked on top of it, using the same approach as the MonadState (etc.) classes in the monad transformer library.

2.3 Parsing

Tock's occam-π parser is built using Parsec, a backtracking monadic parser combinator library (Leijen and Meijer 2001). Parsec makes writing a parser very straightforward; in many cases the parser functions are simply translations into Haskell syntax of the equivalent productions in the occam-π grammar, returning the corresponding piece of AST:

```
type OccParser = GenParser Token CompilerState

process :: OccParser Process
process = seqProcess <|> assignProcess <|> ...

seqProcess :: OccParser Process
seqProcess
  = do reserved "SEQ" >> eol
       indent
       ps <- many process
       outdent
       return $ Seq ps
```

Our only complaint is that Parsec is not yet provided in monad transformer form, so we cannot stack it on top of PassM directly; some additional manipulation is necessary to get the state and error reports into and out of the OccParser monad.

It is possible to do lexical analysis inside Parsec itself, but we found it advantageous to use a separate lexer, written using the Alex lexer generator. Tock's lexer returns a stream of Tokens, which are then fed through a series of small passes prior to parsing (figure 2). Currently, token-stream passes are used to apply the occam-π C-like preprocessor rules (themselves parsed using a separate Parsec parser), insert included files, and turn occam-π's indentation-based syntax into one with explicit marker tokens. These passes run in PassM, and can perform IO and report errors just as the later AST passes do.

2.4 Nanopasses

The bulk of Tock's code consists of nanopasses that operate upon the AST. Each pass is a function in the PassM monad that takes an AST and returns a new AST:

```
type PassType = AST -> PassM AST
```

A typical nanopass operates only upon selected parts of the AST: it performs what is in effect a recursive pattern-match across the entire AST, looking for nodes of interest. Haskell's built-in pattern-matching does not provide recursion – there is no way of specifying a pattern that will match any Process anywhere within the AST – so a pass written using only Haskell pattern-matching would need to include patterns to match every possible type and constructor in the AST and then explicitly recurse. This would clearly be very inconvenient.

Instead, we write just the functions that operate upon the parts of the tree we are interested in, and use *generic programming* to apply them to the appropriate parts of the tree. This is another reason for using multiple types in our AST representation: we can usually specify our operations in terms of a function that modifies a tree node of the appropriate type.

In practice, it is useful to leaves passes themselves as generic functions that can be applied to any type found in the AST, rather than restricting them to the AST type itself. This makes it easier to construct unit tests for passes, since you can run a pass against

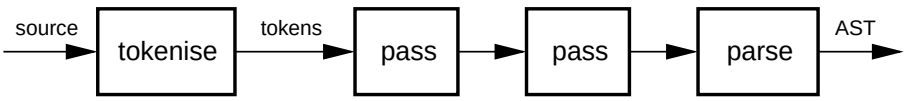

Figure 2. Tock's occam-π frontend

any fragment of the AST without needing to construct a complete AST around it – for example, a pass that operated primarily upon Expressions could simply be called with an Expression as an argument in its unit tests.

Tock's performance depends largely upon the generics system used to implement its passes. Including basic Haskell types, a large occam-π program will have upwards of a million data values in its initial AST, and a typical Tock compilation involves approximately fifty nanopasses, each of which must traverse the AST looking for items to modify. It is therefore important for AST traversals to be as efficient as reasonably possible; a traversal strategy that performs acceptably in a conventional compiler with one or two passes is not necessarily appropriate for a nanopass compiler.

2.5 Ordering Passes

We must combine our passes in the correct sequence to give the whole compiler. Since each pass is a monadic function on the AST, this is simply monadic sequencing at the lowest level:

```
compile x = pass1 x >>= pass2 >>= pass3 ...
```

We use >>= directly when a single logical pass requires multiple traversals of the AST, composing several functions to form the pass. However, it would be awkward to manage the complete set of passes in a nanopass compiler this way; we probably want to give the user some feedback about which pass is running, and there are other useful properties of passes that we want to keep track of.

One of the problems of nanopass compiler design is running the passes in the correct order. The ordering must take into account:

- **The semantic ordering of passes.** A pass that removes an element from the AST must run before other passes that expect that element to have been removed.
- **Efficiency concerns.** The size of the AST should be minimised to reduce traversal costs.
- **User-interface concerns.** Errors must be found and reported in the user's program before the information necessary to report those errors in an intelligible fashion has been removed from the AST.

Tock addresses this problem by providing a set of properties that the AST can have – such as "user datatypes have been resolved", or "parallel assignment has been removed" – and allowing passes to specify the lists of properties that they depend on and provide:

```
resolveUserTypes :: Pass
resolveUserTypes = pass "Resolve user types"
                        [Prop.typesChecked]
                        [Prop.userTypesResolved]
                        (applyDepthM doType)
  where ...
```

Each property is described by the Property data type, which contains a name for the property along with a function that takes an AST and fails if the property does not hold – usually using a generic query:

```
userTypesResolved :: Property
userTypesResolved
    = Property "User types resolved"
               (assertNull . listify findUserType)
```

Given this information, Tock can ensure that passes are being run in a reasonable order, and that passes are doing their jobs correctly. Since some properties are quite expensive to check, the checks are enabled only if Tock is invoked with an appropriate debugging option.

It is possible to compute a valid pass ordering automatically given the dependency information; Tock was initially implemented this way. However, we found that this generally resulted in a non-optimal – and often surprising – pass ordering, and we were forced to define uncheckable properties in order to force the desired ordering. We now specify the pass ordering explicitly. Tock checks that the property dependencies are satisfied by the specified ordering; this helps to detect dependency problems introduced when adding or reordering passes.

Many passes will be conditional on particular compiler options being specified – for example, some checks are only applicable to a particular frontend or backend, and some optimising transformations should only be applied if optimisation has been enabled. Pass specifications can therefore include a function to decide whether the pass is enabled. Maintaining dependency information makes it possible to ensure that pass dependencies are satisfied regardless of the compiler options selected.

2.6 Output

Once the main sequence of passes has completed, the AST is in a form essentially equivalent to the code that will be generated – AST features that do not have a direct translation in the target language will have been rewritten into more appropriate forms. A final backend pass writes the AST out to a file in the appropriate target language. Tock currently has three backends: C99, C++, and XML, the latter just dumping the final AST in a form that can be easily parsed by other programs (such as IDEs or documentation extractors).

The C99 and C++ backends have a large quantity of code in common owing to the similarity between the two languages; while they use different runtime libraries, they translate expressions, basic structures and many datatypes in the same way. We could simply write a set of helper functions and a set of backend-specific functions, but this becomes awkward when a shared helper function needs to call a backend-specific function (figure 3) – we would have to add checks for which backend was in use whenever this was necessary.

To avoid this problem, we use a Haskell approximation of a virtual function table. The C99 and C++ backends run in the CGen monad, which is built on top of PassM (and thus includes IO):

```
type CGen = ReaderT (Handle, GenOps) PassM
```

The Handle is the file handle to write the generated output to, and GenOps is a record of generator functions for various AST constructs:

```
data GenOps = GenOps {
    genExpression :: Expression -> CGen (),
    genProcess :: Process -> CGen (),
    ...
```

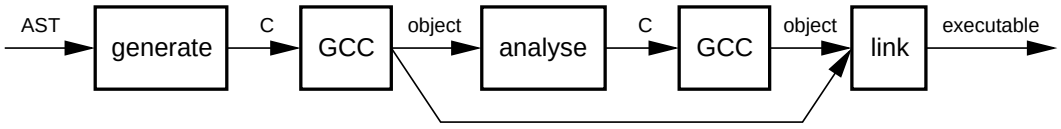

Figure 4. Tock's C backend

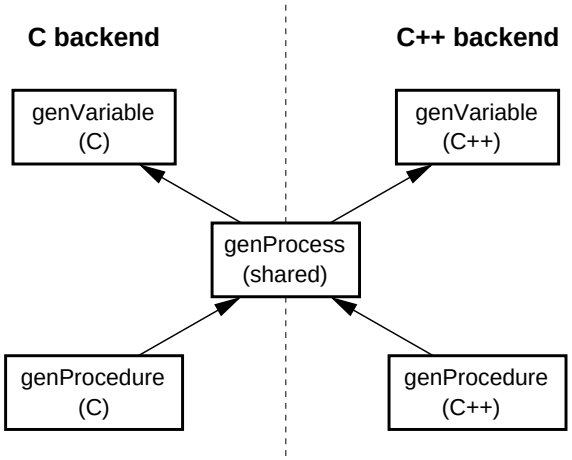

Figure 3. Calling between common and backend-specific functions

```
}
```

Some of these functions are common to both backends; others have different implementations for the C99 and C++ backends. When a backend is started, the GenOps structure is initialised with the appropriate functions for that backend.

A type class CGenCall includes an operation to call any one of these functions, and an instance is provided for each of the possible function kinds that gets the current GenOps from the monad and looks up the requested function.

```
class CGenCall a where
  call :: (GenOps -> a) -> a

instance CGenCall (a -> CGen z) where
  call f x0 = do (_, ops) <- ask
                 f ops x0
```

As a result, the backend code can just say call genProcess a to invoke the appropriate implementation of genProcess. This mechanism is particularly useful when writing unit tests for the backend: it is very easy for a unit test to replace an uninteresting function with something that outputs a dummy value.

After the entire AST has been written out, Tock invokes the system's C or C++ compiler to produce native assembly code and a native object file. In order to allocate a reasonable amount of stack space for each lightweight process at runtime, Tock analyses the generated assembly code, computing the stack usage of each function. The stack usage information is written out to a separate C file, compiled, and finally linked with the main object file and the runtime libraries to give an executable (figure 4).

3. Haskell Generics

There are a number of existing generic programming systems for Haskell (Rodriguez et al. 2008). Tock uses a hybrid approach based upon the "Scrap Your Boilerplate" and "Uniplate" systems.

3.1 Scrap Your Boilerplate

The "Scrap Your Boilerplate" approach (SYB) extends the Haskell runtime system with introspection facilities that allow the types and structures of values to be inspected at runtime, and a cast operation that allows a value to be coerced to a different type (Lämmel and Peyton Jones 2003). To operate upon a data type using SYB, it needs instances of the Data and Typeable type classes. These can be derived automatically for most algebraic types by GHC, and can easily be defined by hand in the few cases where GHC is unable to derive them; we have found this to be necessary for some parameterised types.

SYB is designed around two fundamental operations: mkT, which converts a function of type a -> a into a generic function that can be applied to any type, and gmap which applies a generic function to all the immediate children of any type. It is particularly important to note that gmap does not implicitly recurse: this makes it easy to write generic operations that only traverse parts of a data structure based on the types and values they find within it.

A number of variations upon these fundamentals are also provided to support other types of generic operations – for example, generic queries of type a -> b, and generic monadic operations of type a -> m a. It is also possible to extend a generic operation so that it operates on multiple types, with a different function for each type; this makes it possible to write traversals that look for or modify several types at once.

From these it is possible to construct a variety of higher-level operations, some of which are also included in the SYB library. For example, the everywhere function applies a function recursively everywhere that it matches in an arbitrary data structure:

```
cStyleNames :: Data t => t -> t
cStyleNames = everywhere (mkT doName)
  where
    doName :: Name -> Name
    doName (Name s) =
      Name [if c == '.' then '_' else c | c <- s]
```

The set of higher-level operations included in SYB is fairly limited; it is, however, very straightforward for the user to define their own operations. We found it useful to define higher-level operations that provided more control over which parts of the AST to descend into.

SYB's major disadvantage is efficiency, since it relies entirely upon run-time introspection of the values it is operating upon. We have found the performance of the provided high-level operations such as everywhere to be particularly unsatisfactory when used on a large data structure such as Tock's AST, because they must examine every value in the structure – not just our algebraic data types, but every Char in every String!

Since this means examining millions of values in each traversal of the AST for a large occam-π program, this makes SYB's default high-level operations unacceptably slow for use in a nanopass

compiler. Profiling our initial version of Tock showed that it was spending most of its time in the cast operation.

We addressed this problem at first by providing a "pruning" traversal that avoided recursing into String, Meta, and other types that we knew would never contain types we would want to operate upon. While each pass still examined many unnecessary nodes, the efficiency improvement was sufficiently great to make Tock usable for medium-sized programs.

SYB ships with GHC as the Data.Generics module, requiring no additional libraries or tools. This makes it attractive for use in Tock, since the effort necessary to install and work on it is minimised. On the other hand, other Haskell implementations do not yet support SYB; we do not consider this a problem, since Tock already depends on a number of other GHC extensions.

3.2 Uniplate

The Uniplate library (Mitchell and Runciman 2007) takes an alternative approach to generic operations upon data structures containing multiple types. For each pair of types x, y where x may contain values of type y, Uniplate requires a type class instance Biplate x y, which provides a biplate operation to efficiently map over the "largest" (in terms of subtree size) y values within an x value.

Since these instances would be extremely tedious to write by hand – Tock would need over a thousand of them – and GHC cannot derive them automatically, they are usually generated using an external tool such as DrIFT (Winstanley 1997). Alternatively, the PlateData module implements Biplate in terms of SYB's Data, at a small performance cost.

Like SYB, Uniplate provides a number of higher-level utility functions built on top of the low-level operations; for example, transform is equivalent to SYB's everywhere:

```
cStyleNames :: Biplate t Name => t -> t
cStyleNames = transform doName
  where
    doName :: Name -> Name
    doName (Name s) = ...
```

Uniplate also provides descend, which is similar to gmap, specifically to support the kind of explicit-descent traversals that gmap makes possible: it applies a function to the largest values of the target type in the value it is given. In general, Uniplate's library of high-level traversal functions is richer and better suited to use in compilers than SYB's; this reflects Uniplate's history of use in compilation-related projects such as Yhc.

Uniplate's greatest advantage over SYB is that it is massively more efficient for complex data structures. Since the low-level Uniplate operations always know what type they are looking for, they can avoid recursing into parts of the data type that cannot possibly contain anything interesting – and therefore the "every Char" problem from SYB goes away.

However, Uniplate does not support generic operations over more than one type – a deliberate design decision on the part of Uniplate's authors, who observed that "most traversals have value-specific behaviour for just one type" (Mitchell and Runciman 2007). While we agree with this assessment, the result is that Uniplate cannot be used for the handful of operations in Tock that *do* need to match more than one type.

In addition, Uniplate is not presently included with GHC, nor is it yet included in many Linux distributions; we would need to include it as part of the Tock distribution, or require our users to install it separately.

3.3 Desiderata

From our experience with both SYB and Uniplate, we identified the features that we considered necessary to implement Tock nanopasses using generic operations:

- **Monadic transformations.** Transformation functions must run in PassM (or a similar monad), so they have access to the compiler's state and can report errors. The few passes that are pure functions can be wrapped with return.
- **Explicit descent.** Some passes must be able to decide whether – and when – to descend into a subtree. A convenient way to do this is to provide a function like gmap or descend. (An alternative used by Strafunski (Lämmel and Visser 2002) is to define tree traversal strategies separately from the transformation functions, but in Tock this would mean duplicating code in many cases.)
- **High-level common operations.** Most passes do not need explicit descent; we need helper functions like everywhere to apply simple depth-first transformations and checks to the tree.
- **No need to define instances.** Tock's AST representation is complex, and often extended or refactored. Writing type class instances by hand would require a lot of effort; we must be able to generate code to implement them automatically with an external tool, or, ideally, have GHC derive them for us.
- **Applied operations are themselves generic.** That is, a pass – the application of a set of type-specific functions – can be applied to both an AST and an AST fragment of another type, for testing purposes.
- **Multiple target types.** Several passes – particularly those that walk the tree updating some internal state – need to operate upon multiple target types at once. (This is where Uniplate fails to satisfy our requirements.)
- **Decent performance.** Walking the entire tree for every pass is unacceptably inefficient; each traversal should examine only the AST nodes that could contain the types affected by the generic operation. (This is where SYB fails to satisfy our requirements.)

Both SYB and Uniplate very nearly meet our requirements. We will describe the interface that is provided for generic operations in Tock, and show how it can be implemented using approaches based on both SYB and Uniplate.

4. Tock's Generic Traversal Interface

Tock's Traversal module provides an abstract interface for performing generic operations that (mostly) hides the details of the implementation from the programmer. The interface includes both low-level and high-level functionality.

The types in the interface will depend on the underlying implementation. For this explanation, the types are shown as they are for the SYB implementation, simplified somewhat: in the real interface, everything is parameterised over the monad rather than being fixed to PassM.

4.1 The Low-Level Interface

Performing a generic operation using the low-level interface is a two-step process: the operation is first constructed, then applied to a value giving a new value. An operation consists of a set of type-specific monadic functions that will be applied if their types match; for example, a type-specific function that operates upon Process would be defined as:

```
type Transform t = Data t => t -> PassM t
```

```
doProcess :: Transform Process
doProcess = ...
```

Traversal provides an empty set of operations, baseOp, and a combinator function extOp that adds a new function to the set. (Presently, each set may contain only one function for each target type; this restriction may be lifted in the future.)

```
baseOp :: Ops
extOp :: Ops -> Transform t -> Ops
```

Given a set of Ops, two functions are provided that construct generic functions from them:

```
makeDescend :: Ops -> (forall t. Data t => Transform t)
makeRecurse :: Ops -> (forall t. Data t => Transform t)
```

For each operation, these are used to generate descend and recurse functions:

- descend behaves much like Uniplate's descend: it attempts to apply each function in the set to the largest values of each target type contained inside the value it is given as an argument. If none of the functions match, it simply returns the value it was given.
- recurse, on the other hand, first tries to apply each of the functions in the set to the value it is given as an argument; if none of them match, then it behaves like descend.

As an example, suppose we have the following data structure representing a list of Bools:

```
data Cell = Cons Bool Cell | Nil

value :: Cell
value = Cons False (Cons False (Cons False Nil))
```

We can construct a set of operations that matches the Cons constructor, and sets the Bool to True:

```
ops = baseOp `extOp` doCell
  where
    doCell :: Transform Cell
    doCell (Cons _ c) = return (Cons True c)
    doCell c = return c
```

If we apply makeDescend ops to value, the operation will be applied to the (single) largest subtree of value that is of type Cons, resulting in:

```
Cons False (Cons True (Cons False Nil))
```

If we apply makeRecurse ops instead, the operation will be applied to value itself, with the result:

```
Cons True (Cons False (Cons False Nil))
```

Note that the innermost False remains unchanged in both cases, because in neither case is there implicit descent: once a type-specific function matches, it is up to that function to explicitly descend into the corresponding subtree if desired.

This interface may seem counter-intuitive, but it can be used to construct all the common types of pass in Tock with minimal effort. Let us take as an example a pass that walks the AST, printing messages as it enters and leaves Seq nodes. This follows the usual pattern for explicit-descent Tock passes, showing the typical uses for descend and recurse:

```
printStructure :: PassType
printStructure = recurse
  where
    ops = baseOp `extOp` doProcess
    descend = makeDescend ops
    recurse = makeRecurse ops

    doProcess :: Transform Process
    doProcess (Seq ps)
      = do print "Seq {"
           ps' <- recurse ps
           print "}"
           return ps'
    doProcess p = descend p
```

descend is used when a type-specific function has been applied to a value in the AST and not found anything interesting, in order to descend into its children without applying the same type-specific function again. recurse is used to descend into child nodes from a value you have matched, and to start the generic operation at the top level.

4.2 Implicit Descent

The majority of passes in Tock do not need explicit descent; they are perfectly content to have their type-specific functions applied wherever they match in the tree. To write a type-specific function that recursed depth-first throughout the tree, we would make it call descend on the value it was passed before examining the result:

```
doProcess :: Transform Process
doProcess p
  = do p' <- descend p
       case p' of
         ...
```

To avoid writing this boilerplate in every type-specific function that should recurse, we can abstract this out into a higher-order function that transforms a type-specific function into a recursing version of itself:

```
makeDepth :: Ops -> Transform t -> Transform t
makeDepth ops f
  = do v' <- (makeDescend ops) v
       f v'
```

We can now use makeDepth when building a set of operations, to add operations that should be applied recursively:

```
defrobulate :: PassType
defrobulate = makeRecurse ops
  where
    ops = baseOp `extOp` makeDepth ops doProcess
                 `extOp` makeDepth ops doExpression

    doProcess :: Transform Process
    doExpression :: Transform Expression
    ...
```

Tock provides a similar helper function, makeCheck, for converting a check function of type t -> PassM () into an implicit-descent Transform t; this is used in passes that want to check properties of the AST, such as the typechecker.

4.3 The High-Level Interface

For the majority of passes that apply just one or two functions with implicit descent, this is still needlessly verbose. We therefore provide a number of helper functions to apply a small number of type-specific functions recursively throughout a data structure, implemented in terms of the operations above:

```
applyDepthM :: Data t1 => Transform t1
```

```
                -> ( forall s. Data s => Transform s)
applyDepthM f1 = makeRecurse ops
  where
    ops = baseOp `extOp` makeDepth ops f1

applyDepthM2 :: (Data t1, Data t2) =>
                Transform t1 -> Transform t2
                -> ( forall s. Data s => Transform s)
applyDepthM2 f1 f2 = ...
```

Note that applyDepthM is equivalent to SYB's everywhere or Uniplate's transform. Passes using these functions are very simple:

```
disentangle :: PassType
disentangle = applyDepthM doProcess
  where
    doProcess :: Transform Process
    ...
```

5. Implementing Traversals with SYB

To implement our traversal interface using SYB, we must find a cure for SYB's performance problems.

5.1 Brute Force

If performance were not a concern – that is, if we did not mind traversing the entire tree every time – we can implement the interface above on top of SYB in a very straightforward manner. A set of operations is represented as a function that, given the descend function, builds the recurse function which tries to apply all the type-specific functions or otherwise descend:

```
type MakeRecurse = (forall t. Data t => Transform t)
                   -> ( forall t. Data t => Transform t)
type Ops = MakeRecurse
```

The empty set of operations simply returns the descend function as-is:

```
baseOp :: Ops
baseOp = id
```

Adding a type-specific function to the set means extending the generic function with the new function:

```
extOp :: Data t => Ops -> Transform t -> Ops
extOp ops f = (\descend -> (ops descend) `extM` f)
```

recurse is then just that generic function:

```
makeRecurse ops = ops (makeDescend ops)
```

descend uses the monadic version of gmap to apply recurse to all the children of the value it is given:

```
makeDescend ops = gmapM (makeRecurse ops)
```

5.2 Tracking the Types

To avoid traversing the entire tree with each operation, we need to know when to stop – that is, when it is not worth recursing into a data value, because it cannot possibly contain any of the types we are interested in.

For each type we encounter, we need to determine whether it is a "hit" (one of the target types), a "through" (a type that may contain one or more of the target types), or a "miss" (a type that cannot contain any of the target types). We can do this by reusing part of Uniplate: the PlateData module has to make exactly this decision in order to implement Biplate using the SYB facilities, albeit for only one target type. Two functions adapted from the PlateData code allow us to find a unique type identifier for a value, and to construct a map from type identifiers to decisions:

```
data TypeDecision = Hit | Through | Miss
type TypeSet = Map TypeKey TypeDecision

typeKey :: Typeable a => a -> TypeKey
makeTypeSet :: [TypeKey] -> TypeSet
```

We provide gmapMWith, which takes a TypeSet as an extra argument, and acts upon the decisions in it for each child:

```
gmapMWith ts f = gmapM (each f)
  where
    each f x
        = case Map.lookup (typeKey x) ts of
            Just Hit -> f x
            Just Through -> gmapM (each f) x
            Just Miss -> return x
            Nothing -> return x
```

This is essentially gmapM with the behaviour of biplate: it applies a generic function to the largest subtrees of any of the target types.

We now modify Ops to keep track of the target types, and make baseOp and extOp update the set of types when a new function is added:

```
type Ops = ([TypeKey], TypeSet, MakeRecurse)

baseOp :: Ops
baseOp = ([], makeTypeSet [], id)

extOp :: forall t. Data t => Ops -> Transform t -> Ops
extOp (tks, _, mk) f
    = (tks',
       makeTypeSet tks',
       (\descend -> (mk descend) `extM` f))
  where
    tks' = typeKey (undefined :: t) : tks
```

We include the TypeSet in Ops because it is rather expensive to build, since makeTypeSet must walk the graph of type relationships to determine which types may contain the target types; doing it in extOp means it is only computed once per traversal.

recurse is the generic function, as before:

```
makeRecurse ops@(_, _, f)
    = f (makeDescend ops)
```

descend can now be implemented in terms of gmapMWith:

```
makeDescend ops@(_, ts, _)
    = gmapMWith ts (makeRecurse ops)
```

The resulting code meets our criteria above, performing significantly better than the plain-SYB implementation. However, profiling Tock compiling a typical program reveals that it is now spending most of its time inside typeKey (which involves an unsafePerformIO). We are still paying a penalty for the use of type introspection.

6. Implementing Traversals with Uniplate

We have attempted to fix the performance problem with SYB, with some measure of success. Can we instead fix the problem that we had with Uniplate – that is, can we make Uniplate operate upon multiple target types?

The answer is a qualified "yes". We have not extended Uniplate; instead, we have used a similar approach to provide a type-class-based generic operation that can be used to implement Tock's generics interface.

6.1 Polyplate

The type class in question – called Polyplate, since it is, at least in spirit, a generalisation of Biplate – defines a single function:

```
class Polyplate ops tops t where
  polyplateM :: ops -> tops -> Bool -> t -> PassM t
```

polyplateM, like gmapMWith, applies a set of type-specific functions to the largest subtrees of the appropriate types within a value, behaving like return if none of the type-specific functions match. It differs in that it takes *two* sets of operations: one to apply to the current value (ops), and one to apply to children of the value when recursing into it (tops). We will explain the additional Bool argument shortly.

The type class is parameterised over both sets of functions and the value type. To allow the former, we must encode the set of target types in Haskell's type system. The empty set of operations has the unit type:

```
type BaseOp = ()

baseOp :: BaseOp
baseOp = ()
```

Additional types are added to the set using nested tuples:

```
type ExtOp op t = (Transform t, op)

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

There is a pleasant symmetry here between the functions used to build the set of operations and the type constructors used to build their types; indeed, the ExtOp type constructor can be used infix in the same way as extOp. An operation upon Process and Expression can therefore be defined as:

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

For the following examples, we will use this simple data structure containing only two types.

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

Each instance of Polyplate describes how to apply a set of operations to a value. When the set of operations is not empty, the behaviour of polyplateM depends on whether the current value is a "hit", a "through" or a "miss" for the type of the outermost type-specific function in the set. If it is a "hit", polyplateM just applies the type-specific function:

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

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

If the value is a through or a miss, polyplateM recurses to try the next function in the set. The Bool argument is used here to record whether it will be necessary to descend into the children of the value. For a "miss", it is left alone:

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

For a "through", the descent flag is forced to True:

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

Once the set of operations is empty, we know whether we need to recurse into the children of the type using the topOps set of operations. For a type such as Inner that has no children, the instance is trivial since the behaviour is the same either way:

```
instance Polyplate () tops Inner where
  polyplateM () _ _ v = return v
```

For Outer, we must look at the descent flag. If it is False, we can return the value immediately. If it is True, then we must recursively apply topOps to each of the children of each production, then construct a new value to return.

```
instance Polyplate tops tops Inner =>
         Polyplate () tops Outer where
  polyplateM () _ False v = return v
  polyplateM () topOps True (Foo i)
    = do i' <- polyplateM topOps topOps False i
         return (Foo i')
  polyplateM () _ True Bar = return Bar
```

We can now define recurse and descend functions in terms of polyplateM. recurse applies its set of operations, initialising the descent flag to False:

```
makeRecurse :: Polyplate ops ops t => ops -> Transform t
makeRecurse ops = polyplateM ops ops False
```

Finally, descend forces immediate descent into the value given:

```
makeDescend :: Polyplate ops ops t => ops -> Transform t
makeDescend ops = polyplateM () ops True
```

6.2 Successes and Annoyances

This implementation also meets our criteria, and has excellent performance characteristics: it is about four times faster on typical Tock compilation runs than the type-tracking SYB implementation – and it no longer requires any GHC extensions other than multi-parameter type classes. There is ample opportunity for a smart Haskell compiler to take advantage of inlining and specialisation to further improve the performance of polyplateM.

One downside is that the types involved have become much more complicated. The type class constraints of the recurse and descend functions now depend on the operations involved, rather than only requiring a Data instance for the value. While the types can be inferred automatically for most simple passes, any pass that applies recurse or descend to more than one type will require their types to be explicitly specified (else GHC will infer a type that is too specific for the first use, then complain when it does not match the second).

The complexity can be hidden to some degree through the use of type synonyms (with Recurse ops and Descend ops types, for example), but it is still necessary to introduce many type constraints that were not previously present in the code. The proposed addition of type class synonyms to the language may simplify matters here.

As with Uniplate, the number of instances required to make a complex data structure traversable can be very large: for a data structure containing n types, it is necessary to define $n(n+1)$ instances of Polyplate. This would be extraordinarily tedious to do by hand – Tock needs nearly five thousand instances – but can be readily automated if the structure of the types is known. For Tock, we wrote a helper program that uses SYB to determine the type relationships in the AST, and automatically generates Haskell code to define the necessary Polyplate instances.

Finally, programs using Polyplate take a pathologically long time to compile with GHC; a Haskell file of a few hundred lines that performs a handful of generic operations may take nearly ten minutes to compile. We presume that this is owing to the complexity of the typeclass instances involved, since GHC must traverse each possible path through the graph of typeclass dependencies for each operation to ensure that all the necessary instances are present.

To work around this problem, we could provide – in addition to the automatically-generated instances – an implementation of Polyplate in terms of SYB's Data, using the PlateData approach. This would run more slowly but require only a couple of instances, allowing the Tock developer to trade off Haskell compilation speed against runtime performance.

7. Conclusion

We have used Haskell to implement a nanopass compiler for the occam-π concurrent programming language. Our new compiler is both flexible and maintainable: multiple frontends and backends are supported, new language features can easily be added, and the codebase is smaller, easier to navigate and easier to test than the existing occam-π compiler. The use of Haskell's powerful type system along with test-driven development and automatic property checking helps to prevent several common types of programmer error during Tock development.

The compiler's output via C and C++ is highly portable and performs well; initial benchmarks suggest an eightfold performance increase on straight-line numerical code over the existing compiler. Automatic stack usage analysis makes it possible to run very large numbers of lightweight processes with minimal memory overheads.

We have defined an interface for generic operations that can be used to construct various types of generic transformations and checks across a complex AST, supporting operations upon multiple target types. We have shown how this interface can be implemented using approaches based upon both "Scrap Your Boilerplate" (with run-time typing) and Uniplate (with automatically-generated type class instances). The efficiency of these implementations allows large numbers of nanopasses to be applied to data structures containing millions of nodes without adversely affecting the performance of the compiler.

Tock is open source software, and its components are freely available for reuse in other projects. The source code and more details are available from `http://offog.org/tock`.

Acknowledgments

Our work on Tock is supported by EPSRC grants EP/P50029X/1 and EP/E049419/1. The authors would like to thank Matt Jadud for enthusing them about nanopass compilation, and Tom Shackell for suggesting the use of Uniplate.

References

Neil C. C. Brown. Rain: A New Concurrent Process-Oriented Programming Language. In *Communicating Process Architectures 2006*, pages 237–251, September 2006. ISBN 1-58603-671-8.

Mark P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In *First International Spring School on Advanced Functional Programming Techniques*, pages 97–136, London, UK, 1995. Springer-Verlag. ISBN 3-540-59451-5.

R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *Proc. Practical Aspects of Declarative Programming PADL 2002*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, January 2002.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *TLDI*, pages 26–37, 2003.

Daan Leijen and Erik Meijer. Parsec: A Practical Parser Library. In *ACM SIGPLAN Haskell Workshop*, 2001.

Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 49–60, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-674-5. doi: http://doi.acm.org/10.1145/1291201.1291208.

Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C.d.S. Oliveira. Comparing Libraries for Generic Programming in Haskell. 2008. To appear.

Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. A nanopass infrastructure for compiler education. In *ICFP 2004*, pages 201–212. ACM Press, 2004. ISBN 1-58113-905-5. doi: http://doi.acm.org/10.1145/1016850.1016878.

Peter H. Welch and Fred R. M. Barnes. Communicating Mobile Processes: introducing occam-pi. In *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005. ISBN 3-540-25813-2.

Noel Winstanley. Reflections on instance derivation. In *1997 Glasgow Workshop on Functional Programming*. BCS Workshops in Computer Science, September 1997.