Alloy: Fast Generic Transformations for Haskell

Neil C. C. Brown   Adam T. Sampson

Computing Laboratory, University of Kent, UK, CT2 7NF

neil@twistedsquare.com, ats@offog.org

Abstract

Data-type generic programming can be used to traverse and manipulate specific parts of large heterogeneously-typed tree structures, without the need for tedious boilerplate. Generic programming is often approached from a theoretical perspective, where the emphasis lies on the power of the representation rather than on efficiency. We describe use cases for a generic system derived from our work on a nanopass compiler, where efficiency is a real concern, and detail a new generics approach (Alloy) that we have developed in Haskell to allow our compiler passes to traverse the abstract syntax tree quickly. We benchmark our approach against several other Haskell generics approaches and statistically analyse the results, finding that Alloy is fastest on heterogeneously-typed trees.

Categories and Subject Descriptors D.1.1 [*Applicative (Functional) Programming*]

General Terms Generic Programming, Haskell

Keywords Generic Programming, Haskell, Alloy

1. Introduction

Data-type generic programming concerns functions that depend on the structure of data-types, such as pretty-printing. A very common use is the automatic application of a function that operates on sub-elements of a larger type. This avoids the need for large amounts of systematic boilerplate code to traverse all the types *not* of interest to apply functions to the types that *are* of interest.

Generic programming research has become popular over the last ten years, particularly in the functional programming language Haskell (for a review, see Rodriguez et al. 2008). The approaches mainly differ by theoretical approach or the use of different language features to achieve generic programming (including several language extensions for generic programming).

Our interest in generic programming is pragmatic. We use generic programming in a compiler to eliminate boilerplate, and we require a straightforward API backed by a very fast generics approach (see section 2 for more detail of our requirements). We began by using a pre-existing generics system, but found that it was not fast enough for our needs.

We thus developed our own generics library for Haskell, Alloy, that blends together features of several existing generics approaches into an efficient whole. Our contributions are as follows:

[Copyright notice will appear here once 'preprint' option is removed.]

- We describe the basic algorithm, implementation and API of Alloy, a library for generic traversals and transformations built using Haskell type-classes (section 3). We later describe a further improvement to our approach (section 7).
- We explain several real use cases of data-type generic programming in our compiler, and examine how to implement them efficiently (section 4).
- We benchmark and statistically analyse the results of Alloy and existing generics approaches (sections 5, 6 and 6.5). The results show that Alloy is faster than existing approaches for traversing heterogeneously-typed trees (we conclude in section 8).

2. Motivation

We develop Tock, a compiler for imperative parallel languages such as occam-π (Welch and Barnes 2005), in Haskell. Tock is currently over 20,000 non-blank lines of Haskell code. Tock is a nanopass compiler (Sarkar et al. 2004), meaning that its design consists of many (currently around 40) small passes that operate on the Abstract Syntax Tree (AST) of the program, each performing one simple operation, for example: making names unique, or checking that variables declared constant are not modified.

A pass that makes names unique must traverse the entire AST, operating on all names. A constant folding pass must traverse the entire AST, operating on all expressions. To avoid writing boilerplate for each traversal, we use generic programming. To ensure fast compilation of occam-π code, the 40 traversals of the tree must be as fast as possible.

Our passes typically operate on one or two types, but the most complex passes (such as the type-checker) operate on up to nine types in one traversal, with complicated rules for when the traversal must descend further into the tree, and when it must not. Our AST currently consists of around 40 different algebraic data types, with around 170 constructors between them. If all the basic sub-types (lists, pairs, primitive types, etc) are also included, we have around 110 different types.

We began by using the Scrap Your Boilerplate (SYB) library (Lämmel and Peyton Jones 2003), we found it was too slow for our purposes, leading us to first augment SYB, and then replace it altogether with Alloy.

We require the following generics facilities:

- **Monadic transformations.** Most transformation functions must run in our compiler monad, so that they have access to the compiler's state and can report errors. As we will see later, while we require the full power of monads for the compiler, our generics approach only requires the more general applicative functors (McBride and Paterson 2008).
- **Multiple target types.** Several passes – particularly those that walk the tree updating some internal state – need to operate upon multiple target types at once.

- **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 decision logic in many cases, since traversal strategies are often pass-specific.)
- **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 by hand.** Tock's AST representation is complex, and sometimes extended or refactored. Writing type class instances by hand would require a lot of effort (and be prone to mistakes); we must be able to generate them automatically, such as with an external tool.
- **Decent performance.** Walking the entire tree for every pass is unacceptably inefficient; each traversal should examine as few nodes as possible.
- **Library-level.** We want it to be easy to distribute and build Tock. Therefore any generics approach that we use must be in the form of a library that uses existing Glasgow Haskell Compiler (GHC) features, so that it can be built with a standard distribution of GHC by our end-users. Ideally, we would depend only on extensions to the Haskell language that are likely to end up in the next Haskell standard, Haskell Prime.

In section 4 we will detail several use cases that show examples of where we need these different features of generic programming. There are several features of generic programming in the literature that we do *not* require. We refer to them, where possible, by the names given in Rodriguez et al. (2008):

- **Multiple arguments:** This is required by operations such as generic zipping, or generic equality. In Tock we always operate on a part of the AST and do not need this.
- **Constructor names:** This is required by operations such as *gshow*. While Alloy could easily be extended to support this, we do not require this functionality in Tock.
- **Type-altering transformations:** We need transformations of the form *a* -> *a* (and *a* -> *m a*), but we do not need type-altering transformations of the form *a* -> *b*.
- **Extensibility:** Several authors (Hinze 2004; d. S. Oliveira et al. 2007; Lämmel and Peyton Jones 2005) have identified the problem that once generic functions have been defined as a list of specific cases (also known as tying the recursive knot), a new case cannot easily be added. This is not a problem in Tock, where we never need to extend pass functions with additional specific cases outside of the definition of the pass.

3. Alloy

Alloy, our generics library, is centred on applying type-preserving transformation operations to all of the largest instances of those types in a heterogeneously-typed tree. The largest instances are all those not contained within any other instances of the type-set of interest (see figure 1 for an illustration). The transformations can then descend further if required.

We do this by taking a set of transformation operations (opset for short) and comparing the type that the operation acts on with a current *suspect* type (think of the type being investigated for matches; hence a suspect). If there is a match, the transformation is applied. If there is no match, the operations are applied to the children (immediate sub-elements) of the suspect type and so on until the largest types have all been transformed in such a way.

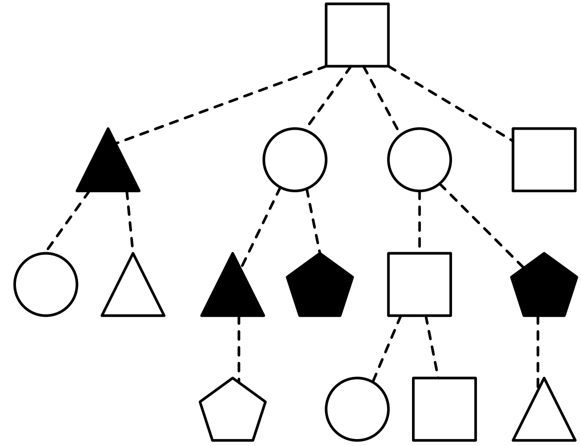

Figure 1. An illustration of the largest types in a tree. The shape of a node indicates its type. The shaded shapes are the largest instances when the types of interest are triangles and pentagons.

Our basic algorithm is to have a *queued* opset ready to be compared to the suspect type, and a *descent* opset ready to be applied to the suspect's children if no exact match is found. We repeatedly take one operation from the queued opset, and compare it to the suspect type. There can be three possible results of this comparison:

1. the suspect type matches the operation type,
2. the suspect type can contain the operation type, or
3. the suspect type cannot contain the operation type.

In case 1, the operation is applied and the result returned. No further work is done by the current call. In case 2, the operation is retained, by moving it onto the descent opset. In case 3, the operation is discarded.

As an example, consider the following type:

```
data Foo = FooInt Int Int | FooFloat Float
```

We wish to apply transformations to everything of type *Float*, *Int* and *String* that might be contained in the suspect type *Foo*.

Figure 2 demonstrates our opset being compared against the suspect type *Foo*. The operations on *Float* and *Int* are retained (because *Foo* can contain those types), whereas the operation on type *String* is discarded.

Alloy is similar to several other approaches, such as Uniplate (Mitchell and Runciman 2007), SYB (Lämmel and Peyton Jones 2003) and Smash (Kiselyov 2006). The two key features of Alloy, intended to increase its efficiency, are that:

1. All our decisions about types are made statically via the Haskell type-checker, rather than dynamically at run-time. Smash and Uniplate take the same approach, in contrast to SYB's use of dynamic typing.
2. Unlike Smash or SYB, we discard operations that can no longer be applied anywhere inside the suspect type. Uniplate, which only supports one target type, stops the traversal when this target type cannot possibly be found anywhere inside the suspect type. We extend this optimisation to multiple types. Not only do we stop when no operations can be further applied, but we also dynamically discard each operation individually when it cannot be applied anywhere inside the suspect type. This is a primary contribution of Alloy.

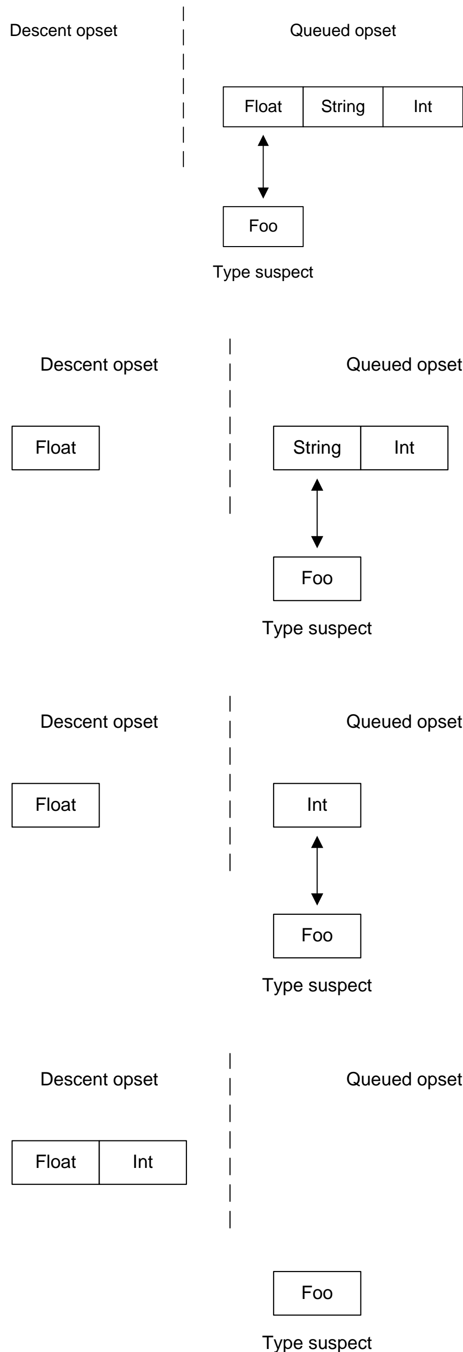

Figure 2. An example of processing an opset with respect to a suspect type. The types of the transformations in the queued opset are progressively compared to the suspect type. If, like *String*, they cannot be contained in the suspect type, they are discarded. If they can be contained, like *Float* and *Int*, they are retained by being moved to the descent opset.

3.1 The Type-Class

Haskell's type-classes are a form of ad-hoc polymorphism that allow functions to be specialised differently for different types. Like Smash and Uniplate, we use Haskell's type-classes to implement Alloy; the library is centred around a type-class of the same name:

```
class Alloy opsQueued opsDescent suspect where
  transform :: opsQueued -> opsDescent -> suspect -> suspect
```

The type-class has three parameters. The first is the queued opset, the second is the descent opset and the third is the is the suspect type, all of which were described in the previous section. Our opsets are implemented in a cons fashion (with terminator *BaseOp*):

```
data BaseOp = BaseOp
data t :- ops = (t -> t) :- ops
infixr 7 :-
```

This allows the value of the opsets to directly mirror the type; a sample opset that works on *String*, *Float* and *Int* is:

```
ops :: String :- Float :- Int :- BaseOp
ops = processString :- processFloat :- processInt :- BaseOp
```

Most of our use of Alloy is via two simple helper functions. The *descend* function[1] is used to apply the transformations to a value's children, which is done by using the *transform* function with an empty queued opset and a full descent opset – which will result in an application of the descent opset to all the children of the value. In contrast, our *apply* helper function begins with a full queued opset and an empty descent opset, and will attempt to apply the operations directly to the time, before descending if none can be applied:

```
descend :: Alloy BaseOp ops t => ops -> t -> t
descend ops = transform BaseOp ops

apply :: Alloy ops BaseOp t => ops -> t -> t
apply ops = transform ops BaseOp
```

We can thus write a compiler pass (that has no automatic descent) as follows:

```
alterNames :: AST -> AST
alterNames = apply ops
  where
    ops = doName :- BaseOp

    doName :: Name -> Name
    doName = ...
```

3.2 Instances

As an example for instances we will consider again the type from the previous section:

```
data Foo = FooInt Int Int | FooFloat Float
```

To aid understanding by those not familiar with Haskell type-classes and manipulations at the type-level, we will also provide a Haskell-like pseudo-code for the instances, of the form:

```
alloyInst :: [Op] -> [Op] -> a -> a
alloyInst queued descent x = ...
```

[1] The *descend* function has the same behaviour as the *compos* operator defined by Bringert and Ranta (2008).

3.2.1 Base Case

We require a base case instance, for when there are no operations left in either opset – none to try to apply to the suspect type, and none to apply to its children. In this case we are no longer interested in this element or anything beneath it, and the identity operation is used on the data:

```
instance Alloy BaseOp BaseOp Foo where
  transform _ _ x = x
```

This is equivalent in our pseudo-code to:

```
alloyInst [] [] x = x
```

3.2.2 Matching Case

We require a case where the type of the operation matches the current type:

```
instance Alloy (Foo :- opsQueued) opsDescent Foo where
  transform (f :- _) _ x = f x
```

Here, we have found a type of interest and the appropriate operation to apply. Therefore we simply apply the operation, ignoring the remaining queued and descent opsets (any required further descent will be done by the *f* function). This is analogous to:

```
alloyInst (f:_) _ x | typeOfOp f == typeOf x = f x
```

The matching of the *Foo* type in our instance declaration is here converted into a guard that uses notional type-getting functions.

3.2.3 Descent Case

We require an instance dealing with the case where there are no operations remaining in the queued opset to try to apply to the suspect type, but there are operations remaining in the descent opset to apply to all the sub-elements:

```
instance (Alloy (t :- ops) BaseOp Int,
          Alloy (t :- ops) BaseOp Float) =>
    Alloy BaseOp (t :- ops) Foo where

  transform _ opsD (FooInt m n)
    = FooInt (transform opsD BaseOp m) (transform opsD BaseOp n)
  transform _ opsD (FooFloat f)
    = FooFloat (transform opsD BaseOp f)
```

The type *t* can be anything here; expressing the opset as a *t*:-*ops* indicates to the type system that it is distinct from *BaseOp*, to prevent the instances overlapping (unlike Haskell's normal in-order pattern-matching, with type-classes every instance must be uniquely determinable from the head). One can think of the constructor *BaseOp* as being the type-level equivalent of the empty list pattern, `[]`, whereas the pattern (*t* :- *ops*) is akin to the cons pattern (*x*:*xs*). This is reflected in the two cases added to our pseudo-code:

```
alloyInst [] opsD@(_:_) (FooInt m n)
  = FooInt ( alloyInst opsD [] m) ( alloyInst opsD [] n)
alloyInst [] opsD@(_:_) (FooFloat f)
  = FooFloat ( alloyInst opsD [] f)
```

The instance body has a case for each constructor of the algebraic data type, and processes each sub-element with a further traversal, where the descent opset is moved to be processed anew on the sub-element type as the queued opset (and the descent opset is emptied).

The head of the instance declaration lists the type-class requirements for these new traversals. In this case, the two types *Int* and *Float* need to be processed with an empty descent opset and a full queued opset.

3.2.4 Sliding Cases

The descent cases had generic opsets – that is, they did not examine what types were in the opsets. The remaining instances must all consider whether the type of the operation at the head of the opset matches, can be contained, or cannot be contained by the suspect type. We perform this check at compile-time, by generating different instances for each combination of suspect type and head of the opset. A couple of the relevant instances for Foo are:

```
instance Alloy opsQueued (Int :- opsDescent) Foo =>
    Alloy (Int :- opsQueued) opsDescent Foo where
  transform (f :- opsQ) opsD x = transform opsQ (f :- opsD) x

instance Alloy opsQueued (Float :- opsDescent) Foo =>
    Alloy (Float :- opsQueued) opsDescent Foo where
  transform (f :- opsQ) opsD x = transform opsQ (f :- opsD) x
```

These instances are processing operations on *Float* and *Int* – two types that *can* be contained in *Foo*. The instance moves the operations from the queued opset to the descent opset, and continues processing the remainder of the queued opset.

Contrast this with the instance for *String*:

```
instance Alloy opsQueued opsDescent Foo =>
    Alloy (String :- opsQueued) opsDescent Foo where
  transform (f :- opsQ) opsD x = transform opsQ opsD x
```

Here, the operation is discarded (*String* cannot be contained by *Foo*), and then we continue to process the remainder of the queued opset. As well as not being applied to *Foo*, the operation will not be checked against any of *Foo*'s children, because it is not added to the descent opset. If *Foo* were a large data-type with many possible sub-elements, this would save a lot of time.

These instances are reflected in the final case in our pseudo-code, now presented alongside the rest of the code:

```
alloyInst [] [] x = x
alloyInst (f:_) _ x | typeOfOp f == typeOf x = f x
alloyInst [] opsD@(_:_) (FooInt m n)
  = FooInt ( alloyInst opsD [] m) ( alloyInst opsD [] n)
alloyInst [] opsD@(_:_) (FooFloat f)
  = FooFloat ( alloyInst opsD [] f)
alloyInst (f:fs) opsD x
  | typeOfOp f `canBeContainedIn` typeOf x
     = alloyInst fs (f : opsD) x
  | otherwise = alloyInst fs opsD x
```

Recall that type-class instances must have a unique match – unlike Haskell functions, they are not matched in-order. Hence our pseudo-code has the same property; none of the pattern matches (plus guards) overlap; this is the reason for the explicit pattern for *opsD* on the third and fourth lines.

We could generate our instances using an approach like Smash, where the information on type relations could be abstracted out into one type-class, and the descent instances put into another, with only four or so instances of *Alloy* to traverse the opset and build on these type-classes. Some preliminary testing indicated that this alternative approach ended up being slower at run-time – but it would be easy to change to this model.

3.2.5 Polymorphic Types

In our compiler application, we have only one polymorphic type, *Structured* (as well as uses of *Maybe* and lists). Typically, we want to apply different operations to the instantiations of these types, e.g. process *Structured Process* differently than *Structured Expression* and `[Char]` differently than `[Formal]`.

Alloy thus does not currently provide any special support for polymorphic types (e.g. processing all *Maybe a*, for all *a*). *Maybe Int* and *Maybe Float* are treated as two entirely separate types, just as *Int* and *Float* are.

3.3 Monadic Alloy

As mentioned earlier, in our compiler nearly all of our passes operate inside a monad. To support monadic transformations, all we strictly need is support for applicative functors – every monad can be made an applicative functor (McBride and Paterson 2008). We must define a new type-class to support this:

```
class AlloyA opsQ opsD t where
  transformA :: Applicative f => opsQ f -> opsD f -> t -> f t
```

In order for it to be apparent to the type system that the applicative functor that *transformA* operates in is the same applicative functor that the opsets use, we parameterise the opsets with the functor. To support this we define our new opsets as follows:

```
data (t :-* ops) f = ((t -> f t) :-* ops f)
infixr 7 :-*
data BaseOpA f = BaseOpA
```

The use of this opset becomes apparent in an example:

```
fixNames :: AlloyA (Name :-* BaseOpA) BaseOpA a => a -> PassM a
fixNames = applyA (doName :-* BaseOpA)
  where
    doName :: Name -> PassM Name
    doName = ...
```

The opset *Name* :-**BaseOpA* is ready to be parameterised by an applicative functor, and the functor being used is not mentioned in the class constraint. The design of the :-* type is such that we guarantee that all operations in the opset are using the same functor, which a plain *HList* (Kiselyov et al. 2004) could not.

The instances for *AlloyA* are nearly identical to those given for *Alloy* in the previous sections. The operations are of type (for example) *Int* -> *f Int* rather than *Int* -> *Int*, and two cases are slightly different – the base case and descent case:

```
-- Base case:
instance AlloyA BaseOpA BaseOpA Foo where
  transformA _ _ = pure

-- Descent case:
instance (AlloyA (t :-* ops) BaseOpA Int,
          AlloyA (t :-* ops) BaseOpA Float)
  => AlloyA BaseOpA (t :-* ops) Foo where

  transformA _ opsD (FooInt m n)
    = pure FooInt <*> transformA opsD BaseOpA m
                  <*> transformA opsD BaseOpA n
  transformA _ opsD (FooFloat f)
    = pure FooFloat <*> transformA opsD BaseOpA f
```

The instances for *Alloy* and *AlloyA* are so similar that we do not have to generate the instances for both *Alloy* and *AlloyA*. We can generate instances for *AlloyA* (the more general case), and define *Alloy* in terms of *AlloyA* by converting each of the operations (using some trivial type-level programming) in the opsets into operations in the *Identity* monad[2]. However, this is not as fast (at run-time) as generating specific instances for Alloy. Defining the pure version in terms of the more general applicative functor version, and the definitions the descent case is very similar to the *ComposOp* module (Bringert and Ranta 2008).

3.4 Common Operations

The *Alloy* type-class we have shown is used to apply transformations to the largest values belonging to types of interest[3] in a tree.

[2] We do this in Tock, for the very few passes that are pure functions.

[3] Recall that the largest types of interest are those not contained by any other types of interest – see figure 1.

Often we actually want to apply a transformation to *all* types of interest in a tree, which we can do by first wrapping each of the transformation functions as follows:

```
makeBottomUp, makeTopDown :: Alloy BaseOp opsDescent t =>
  opsDescent -> (t -> t) -> t -> t
makeBottomUp ops f = f . descend
makeTopDown ops f = descend . f
```

The difference between these two functions is whether the function is applied before or after the descent, which results in the transformation either being bottom-up or top-down. We provide top-down transformations for completeness; Mitchell and Runciman (2007) rightly caution against the use of such transformations.

These functions can then be used in convenience functions to apply functions to two different types in a large tree:

```
applyBottomUp2 :: (Alloy (sA :- sB :- BaseOp) BaseOp t,
                   Alloy BaseOp (sA :- sB :- BaseOp) sA,
                   Alloy BaseOp (sA :- sB :- BaseOp) sB) =>
     (sA -> sA) -> (sB -> sB) -> t -> t
applyBottomUp2 fA fB = apply ops
  where
    ops = makeBottomUp ops fA :- makeBottomUp ops fB :- BaseOp
```

Note that the opset is used in its own definition, because the wrappers for the two functions need to know what operations to apply when recursing. Our type-class constraints indicate what calls to *transform* need to be made:

- One call will be on the top-level type *t* with the full set of queued operations (and an empty descent opset).
- A call will be made on the *sA* type to apply the operations to all of its children. To force this descent into the *sA* type (rather than applying the *sA* transformation again), we pass an empty queued opset, but a full descent opset. This will cause all the operations to be applied to *sA*'s children. If *sA* does not contain *sB*, for example, the opset will be pruned on the next step because therefore none of *sA*'s children contain *sB*.
- The same call will be made on the *sB* type.

It is important to note that *applyBottomUp2 f g* is not guaranteed to be the same as *applyBottomUp f* . *applyBottomUp g* (and *applyBottomUp g* . *applyBottomUp f*) unless the types that *f* and *g* operate on are entirely disjoint. Consider:

```
g :: Maybe Int -> Maybe Int
g = const $ Just 3
f :: Int -> Int
f = succ
x :: Maybe Int
x = Nothing

(applyBottomUp f . applyBottomUp g $ x) == Just 4
applyBottomUp2 f g x == Just 3
applyBottomUp2 g f x == Just 3
```

The composition will apply the second function to children of the result of the first – something that *applyBottomUp2* will not do.

Unlike Uniplate, we do not provide a great variety of helper functions. As well as the simple *descend* and *apply* functions explained in section 3.1, and *applyBottomUp* and *applyBottomUp2* (and applicative versions of each using *AlloyA*), the only other function we need for Tock is a query function akin to SYB's *listify* :

```
findAll :: (AlloyA (s :-* BaseOpA) BaseOpA t,
            AlloyA BaseOpA (s :-* BaseOpA) s) =>
     (s -> Bool) -> t -> [s]
findAll qf x = execState (applyBottomUpA examine x) []
  where
    examine y = do when (qf y) $ modify (y:)
                   return y
```

3.5 Instance Generation

Instance generation is regular and systematic. Naturally, we do not wish users of Alloy to write instances by hand. While there are tools, such as Derive (Mitchell and O'Rear 2009) and DrIFT (Winstanley 1997), for generating Haskell instances (as well as Template Haskell (Sheard and Peyton Jones 2002)), we opted to build our own simple instance generator using SYB.

The advantage of using SYB is that no external tools or libraries are required. SYB requires language extensions in GHC, and SYB is supplied with GHC. We can use its traversals to discover the necessary information (the relations between types in terms of can-contain) to generate Alloy instances for any type that derives the *Data* type-class in the standard way.

4. Use Cases

In this section, we present and discuss some of the uses we make of generic operations. Our approach to designing our passes is guided by the knowledge (backed up by the results in tables 2 and 3 on page 12) that the traversal of large trees such as ours is a large time cost which dwarfs the cost of the operation at particular nodes. We present several use cases in the subsequent sections, discussing a simple way to implement them, and possible efficient refactorings. We accompany each example with some code that makes correct use of Alloy, but that uses a simplified version of our AST.

We characterise our traversals via two orthogonal distinctions: bottom-up (descent before transformation) versus top-down, and depth-first (each child is processed entirely before its sibling) versus breadth-first.

4.1 Correcting Names

The occam naming rules, originally designed over twenty years ago, permit dots in names but not underscores. In order to compile to C, we simply turn each dot in a name into an underscore. This is easily accomplished with our helper functions:

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

The majority of the passes in Tock are implemented using *applyBottomUpA* or *applyBottomUpA2*, but the other examples we give here are the more interesting cases that require a different approach.

4.2 Parallel Usage Check

The languages we compile have parallel constructs that execute several code branches in parallel. We have a pass that checks that each parallel construct obeys the CREW rule: Concurrent-Read, Exclusive Write. We must check that each variable is either:

- not written-to, or
- only written-to in one part of the parallel construct while not used at all (for reading or writing) in any others.

The algorithm is simple, once we have used our traversals to collect sets of written-to and read-from names for each part of the parallel construct, for which we can use a generic operation.

A straightforward implementation would be to use a generic traversal to descend to each parallel construct – then, further generic queries could be used to find all written-to names (by looking for all elements that could be involved in writing to a name, such as assignments and procedure calls) and all read-from names (which can be done by just finding all other names), followed by checking our CREW rule, and descending to find further nested parallel constructs. This would be an implementation of an $O(N^2)$ pass, however, with each instance of name processed once for each parallel construct it is contained within.

We refactor our pass as follows. We perform a traversal of the tree with explicit descent and a monad with a record of used names. When we encounter a name, we add it to this record. At each parallel construct, we explicitly descend separately into each branch with a fresh blank record of names, and when these traversals finish, we use these different name records for our CREW check. Afterwards, we combine all these name records into the state. In this way, we can perform one descent of the entire tree to deal with all the nested parallel constructs. The code is:

```
-- Issues an error if the CREW rule is broken
checkSets :: [Set.Set String] -> PassM ()

checkCREW :: AST -> PassM AST
checkCREW x = execWriterT (applyA ops x) Set.empty
  where
    ops = doProc :-* doName :-* BaseOpA

    doProc (Par ps)
      = do ns <- mapM (flip execWriterT mempty . applyA ops) ps
           checkSets ns
           tell $ mappend ns
           return $ Par ps
    doProc other = descendA ops other

    doName (Name n) = do tell $ Set.singleton n
                         return $ Name n
```

Note that we have two cases for *doProc*; one to handle the constructor we are interested in, and a default case to descend into the children of all the other constructors. If we used *applyA* here we would get an infinite loop (*applyA* would apply *doProc* again from the opset), so we must use *descendA*.

Several other passes in Tock make use of this pattern: manipulating/checking parts of the AST in a manner dependent on their sub-nodes. For example: removing unused variables, pulling up free names in procedures to become parameters, or pulling up sub-expressions into temporary variables. The latter two are actually rearrangements of the tree, pulling up sub-trees to a higher-level, which we do by recording the pulled-up trees in a monad on a descent, then inserting them higher up.

4.3 Adding Channel Directions

Our source languages feature communication channels. Channels can be used with direction specifiers that indicate the writing or reading end of a channel, but we allow these specifiers to be omitted where they can be inferred. For example, if a channel is used in an output statement, we can infer that the writing end of the channel is required. The compiler must therefore infer the direction specifiers on all uses of the channel (in our AST, a variable can be a directed channel variable, much as we can have subscripted array variables). Even though our interest is in modifying the variable, our traversal must descend to the level of output/input statements, and then further descend into the variable to see if a direction specifier is necessary (or, indeed, if an invalid specifier has been given).

This pass illustrates two concepts that feature in other passes:

1. The concept of descent in a context is used more extensively by our type-checker, where the processing of an inner node is dependent on the value of a parent node.
2. Output and input statements are just two of many values a statement can take. We wish to process these constructors specifically, and descend further into other statements without processing. It is typical that we only want to process one or two constructors of a particular data type, and either descend, or ignore the rest.

4.4 Directly Contains

We are often concerned with processing every element of a particular type, but we need to take care when processing types that can be recursive. Consider the case of lists: strings, for example. A Haskell list is a directly recursive data-type. We may want to write a function to append a prime symbol to all strings. If we blindly apply a function like SYB's *everywhere* or Alloy's *applyTopDown* with the operation `(++ "'")`, we will get multiple primes added to a string; the string `"foo"` technically contains four strings (`"foo"`, `"oo"`, `"o"` and `""`), and a generic traversal appending to strings, applied everywhere, will prime each of them before joining it back to the rest of the string, resulting in `"foo''''"`. This is a simple example that a programmer should see to avoid. However, the issue becomes more complicated with more complicated data-types.

We have a pass to pull up (hoist) array literals from expressions into variables. In occam-π, array literals are delimited by square brackets, much as list literals are in Haskell. We may have some occam-π code such as:

```
a := doubleEach([xs, [0,1], doubleEach([2,3]), ys])
```

We need to pull up any array literals that are not directly nested inside other array literals, yielding the new code:

```
temp IS doubleEach([2,3]):
temp2 IS [xs, [0,1], temp, ys]:
a := doubleEach(temp2)
```

Note that we do not need to pull up the `[0,1]` literal directly nested inside one literal (our multidimensional arrays compile to a flattened single-dimension array) – so we do not want to blindly pull up all array literals. We still want to pull up the array from the inner function call, though. To deal with these sorts of issues, we usually find the largest expression, then write some code to explicitly descend (ignoring directly nested array literals) and revert back to generic descent when we encounter other nodes (such as the inner function call to `doubleEach`). We also have to deal with pulling up the temporaries to the nearest appropriate place:

```
makeUniqueTemp :: PassM Name

applyItems :: [(Name, Expression)] -> Struct -> Struct

pullUpArrayLiterals :: Struct -> PassM Struct
pullUpArrayLiterals x = evalWriterT (doStruct x) []
  where
    ops = doExpr :-* doStruct :-* BaseOpA

    doExpr (ArrayLit es) = do es' <- mapM doArrayLit es
                              t <- makeUniqueTemp
                              tell [(t, ArrayLit es')]
                              return $ ExprVariable t
    doExpr e = descendA ops other

    doArrayLit (ArrayLit es) = liftM ArrayLit (mapM doArrayLit es)
    doArrayLit e = descendA ops e

    doStruct s = do (s', items) <- listen $ descendA ops s
                    return $ applyItems items s'
```

4.5 Making Names Unique

Making names unique, or 'uniquifying' names, is the process of renaming declared names to be unique in the program, and resolving all uses of that name to match the new declared name. Thus, we want to find declarations, and alter their name, followed by recursing down the tree to resolve all uses of that name, doing so in a top-down manner (name shadowing is allowed!).

If names are declared in two different AST element types (as used to be the case in Tock), we could not resolve the names correctly by resolving names for each AST declaration type separately – a name declared in one fashion may shadow a name declared in another fashion. So we would require one pass that operates on two types, and could not use two passes that each operate on one type.

One way to implement name resolution non-monadically is to search for name declarations in a *bottom-up* fashion, then process the scope of the declaration to uniquify all uses of the given name. This would resolve all names to their closest declaration, but the run-time would be $O(N^2)$. We can avoid the use of a reader monad for the name stack but we choose to retain a state monad for assigning a unique suffix to the name, and the error monad for issuing errors:

```
addUniqueSuffix :: String -> PassM String

uniquifyNames :: AST -> PassM AST
uniquifyNames = applyA (ops [])
  where
    ops nameStack
      = doDecl nameStack :-* doName nameStack :-* BaseOpA

    doName nameStack (Name n)
      = case lookup n nameStack of
          Nothing -> throwError $ "Name " ++ n ++ " not found"
          Just resolved -> return $ Name resolved

    doDecl nameStack (Decl n body)
      = do unique <- addUniqueSuffix n
           liftM (Decl unique) $
             applyA (ops $ (n, unique) : nameStack) body
    doDecl nameStack other = descendA (ops nameStack) other
```

We omit the irrelevant details of the *addUniqueSuffix* function. When processing names, we look for the most recent entry on the stack and use that as the new name. We need not descend further, because there are no elements of interest inside *Name*.

For declarations, we make a unique version of the name, and then descend into the body of the declaration with the adjusted name stack. This example demonstrates an interesting mix of pure programming (the name stack) and effectful programming (to get the unique identifier for the names).

4.6 Summary

We have described several ways in which we make use of monads in our passes. Allowing transformations to be monadic/applicative is the most flexible way to augment and implement much of the dependence involved in our passes (i.e. where one part of the transformation depends on the results of another part).

The cost involved in descending the tree guides much of the design of our passes, so that we traverse the tree as few times as possible. However, for clarity of design, we stop short of combining several passes into one (although we have considered attempting to do so automatically).

5. Related Work

Rodriguez et al. (2008) provide a comprehensive review of generic programming libraries, and Hinze et al. (2006a) provide a slightly older review of generic programming approaches (including non-library approaches). In this section we summarise the features and approaches of various generic libraries.

Scrap Your Boilerplate (Lämmel and Peyton Jones 2003) is a system based on dynamic type examination. Lists of operations can be constructed, and the correct operation to apply is chosen by dynamically comparing the type of the suspect data item to the type of the operation. This is slow, but SYB is well-maintained and is effectively built-in to GHC, making it easily available.

SYB with class (Lämmel and Peyton Jones 2005) reworks SYB to allow extensible generic functions – something not all other

approaches (including Alloy) can achieve. The Spine work (Hinze et al. 2006b) also built on SYB, transforming SYB into a more type-theoretic approach that removed the dynamic polymorphism.

Smash (Kiselyov 2006) has an HList (Kiselyov et al. 2004) of operations that provided inspiration for our opsets. Smash uses static type-level techniques to compare the type of the operation against the suspect type. Alloy includes the extra optimisation for discarding operations by using more information about the types involved, but does not support as many forms of transformation and traversal as Smash.

Uniplate (Mitchell and Runciman 2007) and its related library Biplate use type-classes instances to descend into types, looking for the largest instances, which also influenced Alloy's design. The main restriction of Uniplate and Biplate is that they can operate on only one target type – Alloy lifts this restriction to allow operation on multiple types by using type-level opsets.

Compos (Bringert and Ranta 2008) has an explicit descent mechanism similar to our *descend* function, and also allowed use with applicative functors, much as our *AlloyA* class does. However, Compos adopts a GADT approach and again lacks our optimisation for discarding operations.

Many generics libraries focus on transforming Haskell data-types into a simpler (and more theoretical) representation, with special cases for primitive types (*Int* and similar) and a sum-of-products view for all other types. Both RepLib and EMGM (Hinze 2004; d. S. Oliveira et al. 2007) and take this approach. This builds a layer of abstraction that simplifies traversals. The performance of EMGM reported later in this paper demonstrates that this layer does not necessarily come at a performance cost.

While this paper has focused on libraries for generic programming, there are also several other approaches to generic programming in Haskell. Template Haskell (Sheard and Peyton Jones 2002) allows code to be run by the compiler, using compiler-level information on types and other aspects of the program to generate further code before compilation continues. EMGM uses Template Haskell to generate instances, but Template Haskell could equally be used to generate traversal code. There are also language extensions such as PolyP (Jansson and Jeuring 1997) and Generic Haskell (Clarke and Löh 2003), as well as external tools such as DrIFT (Winstanley 1997) and Derive (Mitchell and Runciman 2007), but as stated earlier we required a library-level approach.

6. Benchmarks

Our primary motivation for creating Alloy was to increase the speed of our generic traversals, and in this section we describe some benchmarks and analyse the results. We have used only transformations in our benchmarks: this is the majority of use in Tock, and frequently used elsewhere too (Rodriguez et al. 2008).

We first tried using the GPBench generic programming benchmarks (`http://www.haskell.org/haskellwiki/GPBench`) but found that those did not sufficiently distinguish the different approaches. Instead, we used the following benchmarks, taking data from Tock to provide larger-scale benchmarks:

- **BTree:** Common binary tree data structure, with a transformation that alters the value at each leaf node. There are only two types: the binary tree, and the leaf type. Data instances are perfectly symmetric, with a given depth.
- **OmniName:** The real AST structure from Tock, using parsed existing compiler tests as input data values, transforming every *Name* item in the AST.
- **FPName:** The same AST structure, but only transforming *Name*s that are function calls or procedure calls (requires examining three types).

Our expectations with respect to Alloy were that its worst relative performance would be on the BTree example (homogeneous data-type, everything can contain the target type), with mid-level performance on OmniName, and best relative performance on FPName, since this involves traversing less of the tree than OmniName (an optimisation that will not be spotted by the other approaches).

Although Alloy was written to be faster in Tock, there is no Tock-specific code in Alloy that confers it an advantage in the latter two benchmarks. We believe that using a real example of a complex tree structure will give the best idea of real performance of the techniques.

We chose to benchmark Alloy against SYB (Lämmel and Peyton Jones 2003), Smash (Kiselyov 2006) and EMGM (Hinze 2004; d. S. Oliveira et al. 2007). Uniplate (Mitchell and Runciman 2007) would not support the FPName benchmark, and thus we did not test it. Uniplate, EMGM and Smash were indicated by Rodriguez et al. (2008) to be the fastest generics approaches, and we include SYB due to its popularity and it being the approach that we have replaced with Alloy in Tock.

6.1 Modifications

After some initial explorations with simple implementations of the benchmarks for each approach, it was clear that Alloy was an order of magnitude faster than the other approaches. We knew from experience why this was. Our AST data structure is filled with *String*s – variable names, function names, source code filenames, and so on. Naïvely applying generic approaches such as EMGM, Smash and SYB leads them to descend into every character of every *String*, attempting to apply transformations. Alloy naturally avoids this due to its optimisation to discard operations that cannot be applied – when processing a *String*, all operations not targeted at *String* or *Char* will have been discarded, which in practical terms means *String*s are never descended into. We felt it realistic to add special cases to EMGM and Smash and SYB that skip over *String*s, as we did with SYB in earlier versions of Tock. Thus, EMGM and Smash both have *standard* versions (without this special case) and *modified* (with this special case). The performance of SYB (both in terms of speed and memory) without this special case was such that we were unable to complete the benchmarks, so SYB only features *with* this special case.

6.2 Parameters

Rodriguez et al. (2008) found that the performance of generics benchmarks were sensitive to differing compiler versions and optimisation levels. We aimed to compensate for this difference by testing two compiler versions (GHC 6.8.2 and 6.10.1), each with three optimisation levels (O0, O1 and O2) and used statistical analysis to try to abstract from these differences to get an overall measure of how performance differed by generics approach.

For the OmniName and FPName benchmarks, we used ten different ASTs taken from an occam compiler test suite that predates Tock. We first timed (wall-clock time) five instances (per combination of approach/compiler/optimisation/AST) of applying the operation a fixed number of times (50) and forcing evaluation of the output, with the following approaches (the difference between modified and standard is explained in section 6.1):

1. Alloy
2. Smash (modified implementation)
3. EMGM (modified implementation)
4. SYB (modified implementation)
5. Smash (standard implementation)
6. EMGM (standard implementation)

Factor Levels	OmniName	FPName	BTree
EMGM / Alloy	1.57	1.32	0.72
Smash / Alloy	1.91	2.17	1.06
Smash / EMGM	1.21	1.64	1.46
Opt 1 / Opt 0	0.61	0.56	0.52
Opt 2 / Opt 0	0.61	0.56	0.51
Opt 2 / Opt 1	0.99	1.00	0.99
GHC 6.10 / GHC 6.8	0.92	1.07	1.08

Table 1. The relative difference in the factors according to separate fitted generalised linear models for each benchmark. For example, the linear model indicates that Smash took 1.91 times as long as Alloy to complete the OmniName benchmark, discounting the effect of other factors.

It was apparent after running these benchmarks that approaches 4–6 were considerably slower than 1–3 (see tables 2 and 3 on page 12), and thus we only used approaches 1–3 in a subsequent experiment where we measured 30 instances of each combination (this showed no difference in means than our original run, but had a smaller standard error, allowing us to be more confident in our results).

For the BTree benchmark, we used a symmetric tree of height 14, and timed 50 instances (per combination of approach/compiler/optimisation) of applying 100 increments on all leaves in the tree. The results are shown in table 4 on page 12.

6.3 Analysis

There was no guarantee of a relationship between a technique's performance on one benchmark and its performance on another. Therefore we analysed each benchmark separately. For the AST benchmarks (OmniName and FPName), our dependent variable was the time measurement, and our four independent variables were: generics approach, compiler version, optimisation level and AST input. The last factor was included because the ASTs varied greatly in size, and thus we could not meaningfully directly compare results across the different ASTs, such as averaging the time taken across the ASTs. Note that in this section, for OmniName and FPName we only discuss the analysis of the three fastest approaches (Alloy, EMGM modified and Smash modified that were run 30 times).

We performed an initial analysis using a four-way ($3 \times 2 \times 3 \times 10$) analysis of variance (ANOVA). This revealed that all factors had a significant main effects, and all the interactions[4] of factors were also significant (significance at the 1% level, all p-values $< .001$). This is because our benchmarks are deterministic, and thus the variance is primarily measurement error, differences in cache state and similar.

We primarily wish to compare the performance of the three remaining generics approaches to each other. To do this, we fitted a generalised linear model to our data, which assigns to each level of each factor a weight that describes how many seconds that factor level adds/subtracts from a baseline. The absolute values are of little interest; instead we look at the *relative* difference between the levels. The values of the differences can be seen in table 1 (without the factors for task input, which are not of interest).

Table 1 is thus the most concise summary of all our results. This tells us that Alloy is around twice as fast as Smash on our AST-based benchmarks, and offers similar performance in the binary tree example. EMGM is around a third to a half slower in the AST benchmarks, but a quarter faster in the binary tree example. These figures include the *String* optimisations for EMGM and Smash (in the AST examples).

[4] An interaction is a difference in the effect of a given factor as a function of the level of another.

6.4 Compiler Versions and Optimisation Levels

Our analysis of variance revealed that there is a statistically significant interaction between approach, compiler and optimisation level. Figure 4a (page 12) illustrates this for one task input to the OmniName benchmark. It can be seen that GHC 6.10 is not always better than GHC 6.8 – EMGM in particular is much slower in the newer compiler, and Alloy too. Optimisation level 1 usually offers a big improvement over no optimisation (level 0), but optimisation level 2 is only sometimes better than level 1 – this is noted in the GHC 6.10 user manual: "At the moment, -O2 is unlikely to produce better code than -O1." This is backed up for the BTree example in figure 4b (page 12) and by the figures in table 1.

It can be seen in table 3 (page 12) that in GHC 6.8 at optimisation level 1, EMGM and Alloy have the same performance (a t-test confirms there is no significant difference between the two at the 5% level), but this is not true in GHC 6.10, where Alloy has become faster, and EMGM slower. This illustrates some of the effect compiler and optimisation can have – but, broadly, the compiler and optimisation levels did not affect the *ranking* of the approaches.

6.5 Discussion

Our benchmarks showed that for the AST-based benchmarks, Alloy was faster than the other approaches – although close enough to EMGM that the difference may not matter for many users. The benchmarks confirmed that SYB was an order of magnitude slower than the other approaches on the AST-based benchmarks (see tables 2 and 3). For homogeneous types, Alloy is not the fastest (and probably not the most suitable, either), while SYB has less of a performance gap (see table 4).

Unexpectedly, the gap between EMGM and Alloy was narrower on FPName (see table 1) where we had expected Alloy to be the clear winner. The FPName benchmark took around the same amount of time in Alloy as the OmniName benchmark – it took the same amount of time to process all the names at specific places in the AST than it did to process all the names. It is possible that the cost of having three types in our opset counterbalanced the savings of being able to discard those operations. One of the target AST types, *Expression*, occurs throughout the tree very frequently, so this may have also contributed to the lack of savings for Alloy.

7. Opening the Closed World

There are several issues with the design of Alloy as described thus far, stemming from one cause: for each type, Alloy requires instances for all the types contained and not contained within it. Consider the type-relation table shown in figure 3a for the following types:

data *Foo* = *Foo Int Int*
data *Bar* = *Bar1 Foo* | *Bar2 Baz*
data *Baz* = *Baz1 Foo* | *Baz2 Bar*

Each row in the figure corresponds to a type-suspect. If the type-suspect contains the type at the top of a column, a 'c' is present. An 'n' indicates the type-suspect cannot contain the column type, and the equals signs on the leading diagonal show where the types match. Each entry in the type-relation square will have a corresponding instance (the row-type being the type suspect, and the column-type being the operation type of the head of the opset).

With N types, this means there will be N^2 instances. This large number of instances is the first problem. It can also be seen that adding further types at a later date (perhaps generated by someone using the original types in a library) requires several new instances. To add a new type, one must supply not only the instances for the

	Foo	Bar	Baz	Int
Foo	=	n	n	c
Bar	c	=	c	c
Baz	c	c	=	c
Int	n	n	n	=

(a)

	Foo	Bar	Baz	Int	Quux
Foo	=	n	n	c	n
Bar	c	=	c	c	n
Baz	c	c	=	c	n
Int	n	n	n	=	n
Quux	c	n	n	c	=

(b)

	Foo	Bar	Baz	Int	Quux
Foo	=			c	
Bar	c	=	c	c	
Baz	c	c	=	c	
Int				=	
Quux	c			c	=

(c)

Figure 3. Type-relation squares for (a) four types, (b) with a fifth type added, requiring nine new instances, and (c) the same square with overlapping instances. A 'c' indicates the row-type contains the column-type, an 'n' indicates the row-type does not contain the column-type, and an '=' indicates type equality.

types contained with the new type, but also the instances for all the existing types (that the new type is not contained within). Consider the extra type:

```
data Quux = Quux Foo
```

The new type-relation square can be seen in figure 3b, and the user must add all the shaded types: both the bottom row regarding types contained in *Quux*, but also the right-hand column with all the instances stating that *Quux* is not contained by any of the existing types. If *Quux* is added on later in a separate module, it is inherent that *Quux* cannot be contained in any of the original types, so this new column will always be filled with 'n' instances (ignoring the equals instance). Having to add all these instances is a second problem.

Both of these problems can be solved by the use of the overlapping instances Haskell language extension. Overlapping instances allows us to still have specific instances for all the cases where the current type matches, or is contained within the type of the latest operation, but then allows us to provide a single generic instance for the not-contained-within case:

```
instance Alloy opsDescent opsQueued =>
         Alloy opsDescent (t :- opsQueued) where

  transform opsDescent (_ :- opsQ) x = transform opsDescent opsQ x
```

This instance means that adding a new type only requires adding (at most) half the previous number of instances, and in general the number of instances is greatly reduced. Consider the new type-relation square in figure 3c, where all the 'n' decisions have been replaced with empty squares (they are covered by our overlapping instance). The number of instances is almost halved. In Tock, with this overlapping instance we generate around 5200 instances – without it, we generate around 13000.

It can be seen that adding *Quux* requires only instances for the types contained within *Quux*, and no more. This transforms our approach from a closed-world system into more of an open-world approach where new types can easily be added on later. It also reduces compilation times (see section 8.1 for more discussion of this issue).

8. Conclusions

We presented a new generics approach, Alloy, whose development and design was motivated by our use of generics in a compiler. Alloy is a powerful and fast library that can be used in any application where transformations and traversals of large complex data structures are required.

Our benchmarks confirmed that the generics approaches closest in performance to our own (that had to be optimised slightly for the application) are 30–100% slower than Alloy on large heterogeneous data types, and showed that SYB is around 3000% slower. Alloy's advantage is eliminated on more homogeneous data types. We showed that our results persist across compiler version and optimisation level, suggesting that generics comparisons are perhaps not as sensitive to these factors as previously thought.

We described use cases for generics in our compiler, and examined how they need to be rewritten to ensure that the number of passes required is kept to a minimum. On large tree structures, the time taken by the traversals outweighs the processing at each node. In our compiler, Tock, switching from (an augmented and optimised) SYB to Alloy approximately halved our entire compile-time for occam-π code (which includes parsing and code generation), giving a strong indication of where much of the time is spent.

8.1 Limitations and Future Work

Most generics approaches require an instance to be generated per type. In Alloy, the number of instances is proportional to the square of the number of types (see section 7 for more details). Our improved performance at run-time comes at a cost at compile-time. GHC takes in the order of five to ten minutes to compile our thousands of generated instances. These instances are only re-compiled, however, when our AST type changes – which is very infrequently compared to how often we compile the compiler. For projects that have complex types that change often, this is a drawback to using Alloy and reveals a potential limitation for the scalability of our particular approach, even with the overlapping instances improvement detailed in section 7. In future we would like to investigate ways to alleviate this problem.

Some of the passes in Tock operate on only one constructor of a type; it descends into all the rest to continue the traversal. Several generics systems have support for special cases for particular constructors, but Alloy does not. We could perhaps alter the opsets so that an operation could either be a transformation on a whole type, or a particular case for a constructor. It is unclear whether this would bring benefits, either in speed or in terms of code clarity.

Our AST contains one polymorphic type, *Structured*, which supports name declarations surrounding an inner type. *Structured* is used with seven different types as a parameter at various places in our AST. Some functions, such as those modifying name declarations, operate on all the different variants of *Structured*, while others are only interested in manipulating one specific *Structured* instance. Currently, our only support for manipulating all variants of *Structured* is to instantiate the operation for all variants and put

all of them in an opset. In future we would like to investigate neater and more efficient solutions to this problem.

8.1.1 API

Our API was originally based on SYB (`:-*` is akin to *extM*). When we began developing Alloy, Tock was already using SYB-based traversals so keeping the API similar to SYB was advantageous in order to ease the transition. Our API can be contrasted with Uniplate's API (Mitchell and Runciman 2007), which is of the form:

```
class Uniplate a where
  uniplate :: a -> ([a], [a] -> a)
```

The *uniplate* function takes a data item, and gives back a list of all the largest sub-elements of that type, along with a function that can take a corresponding list (same length, same order) of values, and reassemble them back into the original item.

The immediate problem with Alloy compared to Uniplate is that multiple types are involved. Still, if we use type-level programming to transform an opset into a corresponding type-level list of types, we could add a front-end class such as:

```
class ConvertOpsToTypes ops ts => Alloy' t ops where
  transform :: t -> ops -> (ts, ts -> t)
```

The instances would need a little alteration so that when an operation is dropped from the opsets, an empty list is put at the correct point in the return type.

8.2 Further Details

We will soon package and release our Alloy library for general use (currently it can be found in Tock at `http://offog.org/darcs/tock/`). We also hope to release our benchmarks, ideally as a contribution to the GPBench (`http://www.haskell.org/haskellwiki/GPBench`) generic programming benchmarks.

8.3 Haskell Extensions

The core idea of Alloy requires a few extensions to the Haskell language (available in the commonly-used GHC compiler). The first is multi-parameter type-classes, and the others are undecidable instances, which allows our type-class recursion (with a corresponding increase in GHC's context reduction stack), as well as flexible contexts and flexible instances for the same purpose, and infix type constructors for our opsets. Multi-parameter type classes and infix type constructors have been accepted for the next Haskell language standard (currently titled Haskell Prime), and the other extensions remain under consideration.

This set of extensions is increased by the use of overlapping instances, although they are not essential for our library. Instance generation takes advantage of GHC's support for automatically deriving the *Data* type-class, but instances could instead be generated by other external tools.

All of these language extensions are pre-existing and have been supported by GHC for many major versions.

References

Björn Bringert and Aarne Ranta. A pattern for almost compositional functions. *Journal of Functional Programming*, 18(5-6):567–598, 2008.

Dave Clarke and Andres Löh. Generic Haskell, specifically. In *Proceedings of the IFIP TC2/WG2.1 Working Conference on Generic Programming*, pages 21–47, Deventer, The Netherlands, 2003. Kluwer, B.V.

Bruno C. d. S. Oliveira, Ralf Hinze, and Andres Loeh. Extensible and modular generics for the masses. In Henrik Nilsson, editor, *Trends in Functional Programming (TFP 2006)*, April 2007.

Ralf Hinze. Generics for the masses. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 236–243, 2004.

Ralf Hinze, Johan Jeuring, and Andres Loeh. *Spring School on Datatype-Generic Programming*, chapter Comparing Approaches to Generic Programming in Haskell. 2006a.

Ralf Hinze, Andres Löh, and Bruno C. d. S. Oliveira. "Scrap Your Boilerplate" Reloaded. In *Proceedings of the Eighth International Symposium on Functional and Logic Programming (FLOPS 2006)*, 2006b.

P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

Oleg Kiselyov. Smash your boilerplate without class and typeable. http://article.gmane.org/gmane.comp.lang.haskell.general/14086, 2006.

Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107, 2004.

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

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *ICFP 2005*, pages 204–215. ACM Press, September 2005.

Ralf Lämmel and Joost 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.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

Neil Mitchell and Stefan O'Rear. Derive home page, May 2009. URL `http://community.haskell.org/~ndm/derive/`.

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.

Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. Comparing libraries for generic programming in haskell. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 111–122, New York, NY, USA, 2008. ACM.

Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. A nanopass infrastructure for compiler education. In *ICFP 2004*, pages 201–212. ACM Press, 2004.

Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.

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.

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

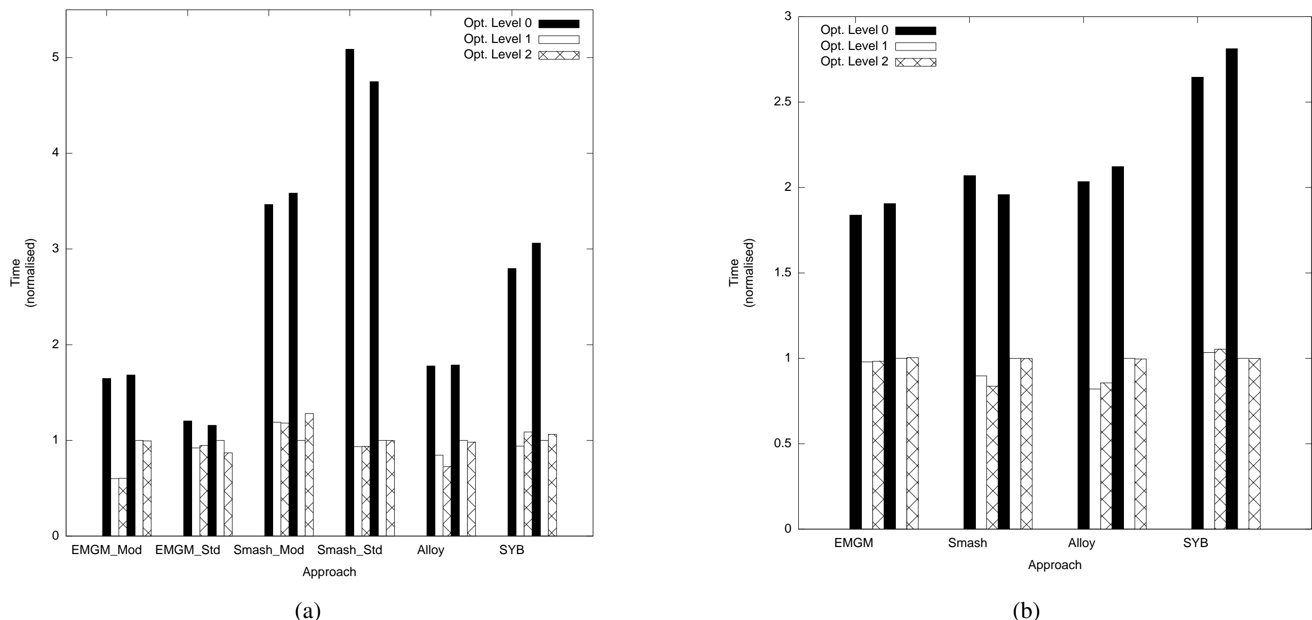

Figure 4. Effect of compiler and optimisation for each approach in (a) the OmniName benchmark and (b) the BTree benchmark. Each approach has two sets of three bars; the left-hand set is GHC 6.8, the right-hand set is GHC 6.10. Each set contains a bar per optimisation level. Each approach has its times (lower is better) normalised to GHC 6.10, Opt. Level 1, so **numbers can only be compared *within* each approach**. There is little difference between optimisation levels 1 and 2 for any approach, but they both show an improvement over optimisation level 0. Speed differs little by compiler version, except that EMGM was much faster under GHC 6.8 at optimisation levels 1 and 2 in OmniName, and in the BTree benchmark Smash and Alloy were slightly faster (at optimisation levels 1 and 2) in GHC 6.8.

Compiler	Optimisation	EMGM Mod.	EMGM Std.	Smash Mod.	Smash Std.	Alloy	SYB Mod.
GHC 6.8	Opt0	3.448 (0.067)	20.669 (0.364)	4.963 (0.056)	34.394 (0.675)	1.536 (0.013)	49.309 (0.275)
	Opt1	1.259 (0.007)	15.832 (0.096)	1.703 (0.015)	6.323 (0.010)	0.730 (0.005)	16.559 (0.233)
	Opt2	1.266 (0.007)	16.278 (0.136)	1.690 (0.017)	6.334 (0.011)	0.627 (0.005)	19.180 (0.061)
GHC 6.10	Opt0	3.526 (0.047)	19.894 (0.143)	5.128 (0.045)	32.101 (0.420)	1.542 (0.012)	53.937 (0.122)
	Opt1	2.096 (0.020)	17.183 (0.165)	1.432 (0.029)	6.760 (0.016)	0.864 (0.015)	17.633 (0.140)
	Opt2	2.085 (0.022)	14.930 (0.087)	1.833 (0.032)	6.754 (0.021)	0.848 (0.011)	18.756 (0.074)

Table 2. An illustrative table of results for one of our test inputs for the OmniName benchmark. Means are wall-clock times (measured in seconds) for 50 traversals, followed in brackets by standard deviations.

Compiler	Optimisation	EMGM Mod.	EMGM Std.	Smash Mod.	Smash Std.	Alloy	SYB Mod.
GHC 6.8	Opt0	3.123 (0.058)	19.189 (0.344)	5.948 (0.074)	39.748 (0.965)	2.066 (0.009)	105.791 (0.548)
	Opt1	0.983 (0.018)	13.118 (0.352)	1.692 (0.049)	6.541 (0.102)	1.013 (0.057)	22.826 (0.055)
	Opt2	1.106 (0.028)	14.169 (0.453)	1.598 (0.056)	6.620 (0.131)	0.598 (0.013)	21.986 (0.170)
GHC 6.10	Opt0	3.219 (0.039)	20.596 (0.152)	5.926 (0.042)	34.415 (0.610)	2.068 (0.013)	109.272 (0.486)
	Opt1	1.560 (0.017)	14.891 (0.152)	1.600 (0.018)	7.056 (0.082)	0.859 (0.006)	17.636 (0.051)
	Opt2	1.432 (0.013)	13.377 (0.092)	1.813 (0.010)	6.896 (0.077)	0.845 (0.003)	19.007 (0.026)

Table 3. An illustrative table of results for one of our test inputs for the FPName benchmark. Means are wall-clock times (measured in seconds) for 50 traversals, followed in brackets by standard deviations.

Compiler	Optimisation	EMGM	Smash	Alloy	SYB
GHC 6.8	Opt0	1.488 (0.025)	2.152 (0.027)	2.112 (0.025)	9.214 (0.074)
	Opt1	0.793 (0.015)	0.868 (0.012)	0.916 (0.022)	3.603 (0.038)
	Opt2	0.796 (0.017)	0.905 (0.017)	0.854 (0.022)	3.668 (0.049)
GHC 6.10	Opt0	1.543 (0.019)	2.245 (0.017)	1.999 (0.016)	9.798 (0.056)
	Opt1	0.810 (0.009)	1.058 (0.010)	1.021 (0.018)	3.484 (0.029)
	Opt2	0.813 (0.010)	1.054 (0.012)	1.019 (0.025)	3.481 (0.031)

Table 4. The results for the BTree benchmark for all four generics approaches. Means are wall-clock times (measured in seconds) for 100 traversals, followed in brackets by standard deviations.